Isabelle/HOL — Higher-Order Logic

August 29, 2014

Contents

1 Code-Generator: Loading the code generator and related modules 29

2 HOL: The basis of Higher-Order Logic 30
   2.1 Primitive logic ........................................... 31
      2.1.1 Core syntax ....................................... 31
      2.1.2 Additional concrete syntax ......................... 32
      2.1.3 Axioms and basic definitions ....................... 33
   2.2 Fundamental rules ...................................... 34
      2.2.1 Equality ........................................... 34
      2.2.2 Congruence rules for application ................... 35
      2.2.3 Equality of booleans – iff ......................... 35
      2.2.4 True .............................................. 36
      2.2.5 Universal quantifier ............................... 36
      2.2.6 False ............................................. 36
      2.2.7 Negation ......................................... 37
      2.2.8 Implication ....................................... 37
      2.2.9 Existential quantifier ............................. 38
      2.2.10 Conjunction .................................... 38
      2.2.11 Disjunction ..................................... 39
      2.2.12 Classical logic .................................. 39
      2.2.13 Unique existence ................................. 40
      2.2.14 THE: definite description operator ............. 40
      2.2.15 Classical intro rules for disjunction and existential quantifiers ........................................ 42
      2.2.16 Intuitionistic Reasoning ......................... 43
      2.2.17 Atomizing meta-level connectives ................. 44
      2.2.18 Atomizing elimination rules ...................... 45
   2.3 Package setup ......................................... 45
      2.3.1 Sledgehammer setup ................................ 45
      2.3.2 Classical Reasoner setup .......................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Simplifier</td>
<td>48</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Generic cases and induction</td>
<td>57</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Coherent logic</td>
<td>61</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Reorienting equalities</td>
<td>61</td>
</tr>
<tr>
<td>2.4</td>
<td>Other simple lemmas and lemma duplicates</td>
<td>62</td>
</tr>
<tr>
<td>2.5</td>
<td>Basic ML bindings</td>
<td>62</td>
</tr>
<tr>
<td>2.6</td>
<td>Code generator setup</td>
<td>63</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Generic code generator preprocessor setup</td>
<td>63</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Equality</td>
<td>64</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Generic code generator foundation</td>
<td>64</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Generic code generator target languages</td>
<td>66</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Evaluation and normalization by evaluation</td>
<td>68</td>
</tr>
<tr>
<td>2.7</td>
<td>Counterexample Search Units</td>
<td>68</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Quickcheck</td>
<td>68</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Nitpick setup</td>
<td>68</td>
</tr>
<tr>
<td>2.8</td>
<td>Preprocessing for the predicate compiler</td>
<td>69</td>
</tr>
<tr>
<td>2.9</td>
<td>Legacy tactics and ML bindings</td>
<td>69</td>
</tr>
<tr>
<td>3</td>
<td>Orderings: Abstract orderings</td>
<td>70</td>
</tr>
<tr>
<td>3.1</td>
<td>Abstract ordering</td>
<td>70</td>
</tr>
<tr>
<td>3.2</td>
<td>Syntactic orders</td>
<td>72</td>
</tr>
<tr>
<td>3.3</td>
<td>Quasi orders</td>
<td>72</td>
</tr>
<tr>
<td>3.4</td>
<td>Partial orders</td>
<td>73</td>
</tr>
<tr>
<td>3.5</td>
<td>Linear (total) orders</td>
<td>75</td>
</tr>
<tr>
<td>3.6</td>
<td>Reasoning tools setup</td>
<td>77</td>
</tr>
<tr>
<td>3.7</td>
<td>Bounded quantifiers</td>
<td>83</td>
</tr>
<tr>
<td>3.8</td>
<td>Transitivity reasoning</td>
<td>85</td>
</tr>
<tr>
<td>3.9</td>
<td>Monotonicity</td>
<td>90</td>
</tr>
<tr>
<td>3.10</td>
<td>min and max – fundamental</td>
<td>93</td>
</tr>
<tr>
<td>3.11</td>
<td>(Unique) top and bottom elements</td>
<td>93</td>
</tr>
<tr>
<td>3.12</td>
<td>Dense orders</td>
<td>95</td>
</tr>
<tr>
<td>3.13</td>
<td>Wellorders</td>
<td>96</td>
</tr>
<tr>
<td>3.14</td>
<td>Order on bool</td>
<td>98</td>
</tr>
<tr>
<td>3.15</td>
<td>Order on - ⇒ -</td>
<td>99</td>
</tr>
<tr>
<td>3.16</td>
<td>Order on unary and binary predicates</td>
<td>100</td>
</tr>
<tr>
<td>3.17</td>
<td>Name duplicates</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>Groups: Groups, also combined with orderings</td>
<td>102</td>
</tr>
<tr>
<td>4.1</td>
<td>Fact collections</td>
<td>102</td>
</tr>
<tr>
<td>4.2</td>
<td>Abstract structures</td>
<td>103</td>
</tr>
<tr>
<td>4.3</td>
<td>Generic operations</td>
<td>104</td>
</tr>
<tr>
<td>4.4</td>
<td>Semigroups and Monoids</td>
<td>105</td>
</tr>
<tr>
<td>4.5</td>
<td>Groups</td>
<td>109</td>
</tr>
<tr>
<td>4.6</td>
<td>(Partially) Ordered Groups</td>
<td>114</td>
</tr>
</tbody>
</table>
5 Lattices: Abstract lattices
5.1 Abstract semilattice
5.2 Syntactic infimum and supremum operations
5.3 Concrete lattices
  5.3.1 Intro and elim rules
  5.3.2 Equational laws
  5.3.3 Strict order
5.4 Distributive lattices
5.5 Bounded lattices and boolean algebras
5.6 \textit{min}/\textit{max} as special case of lattice
5.7 Uniqueness of inf and sup
5.8 Lattice on \textit{bool}
5.9 Lattice on \textit{-} \Rightarrow -
5.10 Lattice on unary and binary predicates

6 Set: Set theory for higher-order logic
6.1 Sets as predicates
6.2 Subsets and bounded quantifiers
6.3 Basic operations
  6.3.1 Subsets
  6.3.2 Equality
  6.3.3 The empty set
  6.3.4 The universal set – UNIV
  6.3.5 The Powerset operator – Pow
  6.3.6 Set complement
  6.3.7 Binary intersection
  6.3.8 Binary union
  6.3.9 Set difference
  6.3.10 Augmenting a set – \textit{insert}
  6.3.11 Singletons, using \textit{insert}
  6.3.12 Image of a set under a function
  6.3.13 Some rules with \textit{if}
6.4 Further operations and lemmas
  6.4.1 The “proper subset” relation
  6.4.2 Derived rules involving subsets
  6.4.3 Equalities involving union, intersection, inclusion, etc.
  6.4.4 Monotonicity of various operations
  6.4.5 Inverse image of a function
  6.4.6 Getting the Contents of a Singleton Set
  6.4.7 Least value operator
  6.4.8 Monad operation
6.4.9 Operations for execution 187

7 Typedef: HOL type definitions 189

8 Fun: Notions about functions 191
  8.1 The Identity Function \( id \) 192
  8.2 The Composition Operator \( f \circ g \) 192
  8.3 The Forward Composition Operator \( fcomp \) 193
  8.4 Mapping functions 193
  8.5 Injectivity and Bijectivity 194
  8.6 Function Updating 203
  8.7 \( \text{override-on} \) 204
  8.8 \( \text{swap} \) 204
  8.9 Inversion of injective functions 206
  8.10 Cantor’s Paradox 207
  8.11 Setup 207
    8.11.1 Proof tools 207
    8.11.2 Functorial structure of types 208

9 Complete-Lattices: Complete lattices 209
  9.1 Syntactic infimum and supremum operations 209
  9.2 Abstract complete lattices 211
  9.3 Complete lattice on \( \text{bool} \) 223
  9.4 Complete lattice on \( \Rightarrow \) 224
  9.5 Complete lattice on unary and binary predicates 225
  9.6 Complete lattice on \( \text{set} \) 226
    9.6.1 Inter 227
    9.6.2 Intersections of families 228
    9.6.3 Union 230
    9.6.4 Unions of families 232
    9.6.5 Distributive laws 235
  9.7 Injections and bijections 236
    9.7.1 Complement 237
    9.7.2 Miniscoping and maxiscoping 238

10 Ctr-Sugar: Wrapping Existing Freely Generated Type’s Constructors 239

11 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions 240
  11.1 Least and greatest fixed points 240
  11.2 Proof of Knaster-Tarski Theorem using \( lfp \) 241
  11.3 General induction rules for least fixed points 241
  11.4 Proof of Knaster-Tarski Theorem using \( gfp \) 242
11.5 Coinduction rules for greatest fixed points 243
11.6 Even Stronger Coinduction Rule, by Martin Coen 244
11.7 Inductive predicates and sets 245
11.8 Inductive datatypes and primitive recursion 245

12 Product-Type: Cartesian products 246
12.1 \texttt{bool} is a datatype 246
12.2 The \texttt{unit} type 247
12.3 The product type 250
  12.3.1 Type definition 250
  12.3.2 Tuple syntax 251
  12.3.3 Code generator setup 254
  12.3.4 Fundamental operations and properties 254
  12.3.5 Derived operations 262
12.4 Simproc for rewriting a set comprehension into a pointfree expression 271
12.5 Inductively defined sets 272
12.6 Legacy theorem bindings and duplicates 272

13 Sum-Type: The Disjoint Sum of Two Types 273
13.1 Construction of the sum type and its basic abstract operations 273
13.2 Projections 275
13.3 The Disjoint Sum of Sets 277

14 Rings: Rings 277

15 Fields: Fields 302
15.1 Division rings 302
15.2 Fields 307
15.3 Ordered fields 312

16 Nat: Natural numbers 325
16.1 Type \texttt{ind} 326
16.2 Type \texttt{nat} 326
16.3 Arithmetic operators 329
  16.3.1 Addition 330
  16.3.2 Difference 331
  16.3.3 Multiplication 332
16.4 Orders on \texttt{nat} 333
  16.4.1 Operation definition 333
  16.4.2 Introduction properties 335
  16.4.3 Elimination properties 335
  16.4.4 Inductive (?) properties 336
  16.4.5 Monotonicity of Addition 339
19 Relation: Relations – as sets of pairs, and binary predicates

19.1 Fundamental ............................................. 424
  19.1.1 Relations as sets of pairs .............................. 424
  19.1.2 Conversions between set and predicate relations ... 425
19.2 Properties of relations ..................................... 426
  19.2.1 Reflexivity ........................................... 426
  19.2.2 Irreflexivity ........................................... 428
  19.2.3 Asymmetry ............................................ 428
  19.2.4 Symmetry .............................................. 429
  19.2.5 Antisymmetry ......................................... 430
  19.2.6 Transitivity ........................................... 431
  19.2.7 Totality ............................................... 432
  19.2.8 Single valued relations ............................... 432
19.3 Relation operations ........................................ 433
  19.3.1 The identity relation ................................ 433
  19.3.2 Diagonal: identity over a set ........................ 434
  19.3.3 Composition ........................................... 435
  19.3.4 Converse .............................................. 437
  19.3.5 Domain, range and field ............................. 440
  19.3.6 Image of a set under a relation ..................... 443
  19.3.7 Inverse image ....................................... 445
  19.3.8 Powerset .............................................. 445
  19.3.9 Expressing relation operations via \textit{Finite-Set.fold} ... 446

20 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

20.1 Equivalence relations – set version ....................... 448
20.2 Equivalence classes ....................................... 448
20.3 Quotients .................................................. 449
20.4 Defining unary operations upon equivalence classes ...... 451
20.5 Defining binary operations upon equivalence classes ...... 452
20.6 Quotients and finiteness ................................ 454
20.7 Projection .................................................. 454
20.8 Equivalence relations – predicate version ................ 455

21 Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

21.1 The datatype universe .................................... 457
21.2 Freeness: Distinctness of Constructors .................. 460
21.3 Set Constructions .......................................... 463
22 Transitive-Closure: Reflexive and Transitive closure of a relation

22.1 Reflexive closure 469
22.2 Reflexive-transitive closure 469
22.3 Transitive closure 474
22.4 The power operation on relations 482
22.5 Bounded transitive closure 490
22.6 Acyclic relations 491
22.7 Setup of transitivity reasoner 492

23 Wellfounded: Well-founded Recursion 494

23.1 Basic Definitions 494
23.2 Basic Results 495
23.3 Well-Foundedness Results for Unions 499
23.4 Acyclic relations 501
23.5 nat is well-founded 502
23.6 Accessible Part 503
23.7 Tools for building wellfounded relations 506

24 Wfrec: Well-Founded Recursion Combinator 510

25 Order-Relation: Orders as Relations 512

25.1 Orders on a set 512
25.2 Orders on the field 513
25.3 Orders on a type 515
25.4 Order-like relations 515
25.4.1 Auxiliaries 515
25.4.2 The upper and lower bounds operators 516
25.5 Variations on Well-Founded Relations 518
25.5.1 Well-founded recursion via genuine fixpoints 519
25.5.2 Characterizations of well-foundedness 519
25.5.3 Characterizations of well-foundedness 521

26 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice 522

26.1 Hilbert’s epsilon 522
26.2 Hilbert’s Epsilon-operator 523
26.3 Axiom of Choice, Proved Using the Description Operator 523
26.4 Function Inverse 524
26.5 The Cantor-Bernstein Theorem 531
26.6 Other Consequences of Hilbert’s Epsilon 533
26.7 Least value operator 534
26.8 Greatest value operator 535
30.5 $\prec o$ is well-founded ........................................ 606
30.6 Copy via direct images ........................................ 607
30.7 Bounded square ............................................... 610

31 BNF-Cardinal-Order-Relation: Cardinal-Order Relations as Needed by Bounded Natural Functors 624
31.1 Cardinal orders ............................................... 624
31.2 Cardinal of a set .............................................. 626
31.3 Cardinals versus set operations on arbitrary sets .......... 631
31.4 Cardinals versus set operations involving infinite sets .... 640
31.5 The cardinal $\omega$ and the finite cardinals .................. 647
   31.5.1 First as well-orders ................................... 647
   31.5.2 Then as cardinals ..................................... 649
31.6 The successor of a cardinal .................................. 649
31.7 Regular cardinals ............................................. 654
31.8 Others .......................................................... 656

32 BNF-Cardinal-Arithmetic: Cardinal Arithmetic as Needed by Bounded Natural Functors 658
32.1 Zero ............................................................ 659
32.2 (In)finite cardinals ........................................... 660
32.3 Binary sum .................................................... 660
32.4 One ............................................................. 663
32.5 Two ............................................................. 663
32.6 Family sum .................................................... 663
32.7 Product ........................................................ 664
32.8 Exponentiation ................................................. 665

33 Fun-Def-Base: Function Definition Base 672

34 BNF-Def: Definition of Bounded Natural Functors 673

35 BNF-Comp: Composition of Bounded Natural Functors 677

36 Basic-BNFs: Registration of Basic Types as Bounded Natural Functors 680

37 BNF-FP-Base: Shared Fixed Point Operations on Bounded Natural Functors 685

38 BNF-LFP: Least Fixed Point Operation on Bounded Natural Functors 690
39 Num: Binary Numerals

39.1 The num type ................................................. 694
39.2 Numeral operations ........................................... 695
39.3 Binary numerals .................................................. 698
39.4 Class-specific numeral rules ................................. 700
  39.4.1 Structures with addition: class numeral ............... 700
  39.4.2 Structures with negation: class neg-numeral ......... 701
  39.4.3 Structures with multiplication: class semiring-numeral 704
  39.4.4 Structures with a zero: class semiring-1 ............ 704
  39.4.5 Equality: class semiring-char-0 ...................... 705
  39.4.6 Comparisons: class linordered-semidom .............. 705
  39.4.7 Multiplication and negation: class ring-1 .......... 707
  39.4.8 Equality using iszero for rings with non-zero character- 
             tic .................................................. 707
  39.4.9 Equality and negation: class ring-char-0 .......... 708
  39.4.10 Structures with negation and order: class linordered-idom 710
  39.4.11 Natural numbers ........................................ 712
39.5 Numeral equations as default simplification rules ...... 715
39.6 Setting up simprocs .......................................... 715
  39.6.1 Simplification of arithmetic operations on integer con- 
         stants .................................................. 716
  39.6.2 Simplification of arithmetic when nested to the right. 717
39.7 code module namespace ...................................... 718

40 Power: Exponentiation

40.1 Powers for Arbitrary Monoids ............................... 718
40.2 Exponentiation on ordered types ............................ 724
40.3 Miscellaneous rules ......................................... 730
40.4 Exponentiation for the Natural Numbers ................. 732
  40.4.1 Cardinality of the Powerset .......................... 733
  40.4.2 Generalized sum over a set .......................... 733
  40.4.3 Generalized product over a set ..................... 734
40.5 Code generator tweak ........................................ 735

41 Meson: MESON Proof Method

41.1 Negation Normal Form ....................................... 735
41.2 Pulling out the existential quantifiers ................... 736
41.3 Lemmas for Forward Proof .................................. 737
41.4 Clausification helper ....................................... 737
41.5 Skolemization helpers ....................................... 738
41.6 Meson package .............................................. 738
48 Complete-Partial-Order: Chain-complete partial orders and their fixpoints 789
48.1 Monotone functions 789
48.2 Chains 789
48.3 Chain-complete partial orders 790
48.4 Transfinite iteration of a function 790
48.5 Fixpoint combinator 791
48.6 Fixpoint induction 792

49 Partial-Function: Partial Function Definitions 794
49.1 Axiomatic setup 795
49.2 Flat interpretation: tailrec and option 798

50 SAT: Reconstructing external resolution proofs for propositional logic 803

51 Fun-Def: Function Definitions and Termination Proofs 803
51.1 Definitions with default value 803
51.2 Measure functions 805
51.3 Congruence rules 806
51.4 Simp rules for termination proofs 806
51.5 Decomposition 807
51.6 Reduction pairs 807
51.7 Concrete orders for SCNP termination proofs 807
51.8 Tool setup 809

52 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers 810
52.1 Definition of integers as a quotient type 810
52.2 Integers form a commutative ring 810
52.3 Integers are totally ordered 811
52.4 Ordering properties of arithmetic operations 812
52.5 Embedding of the Integers into any ring-1: of-int 814
52.6 Magnitude of an Integer, as a Natural Number: nat 816
52.7 Lemmas about the Function of-nat and Orderings 818
52.8 Cases and induction 819
52.8.1 Binary comparisons 820
52.8.2 Comparisons, for Ordered Rings 821
52.9 The Set of Integers 822
52.10 setsum and setprod 824
52.11 Setting up simplification procedures 825
52.12 More Inequality Reasoning 825
52.13 The functions nat and int 825
52.14 Induction principles for int 827
52.15 Intermediate value theorems .............................................. 830
52.16 Products and 1, by T. M. Rasmussen ................................. 830
52.17 Further theorems on numerals ......................................... 831
  52.17.1 Special Simplification for Constants ......................... 831
  52.17.2 Optional Simplification Rules Involving Constants .......... 833
52.18 The divides relation .................................................. 834
52.19 Finiteness of intervals .............................................. 837
52.20 Configuration of the code generator ................................ 838
52.21 Legacy theorems ...................................................... 841

53 Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back. 842
  53.1 Generic transfer machinery ......................................... 842
  53.2 Set up transfer from nat to int ................................... 842
  53.3 Set up transfer from int to nat ................................... 847

54 Divides: The division operators div and mod 850
  54.1 Syntactic division operations .................................... 850
  54.2 Abstract division in commutative semirings ..................... 850
  54.3 Generic numeral division with a pragmatic type class ....... 860
  54.4 Division on nat .................................................... 864
    54.4.1 Quotient .................................................... 869
    54.4.2 Remainder .................................................. 869
    54.4.3 Quotient and Remainder .................................. 870
    54.4.4 Further Facts about Quotient and Remainder ............. 871
    54.4.5 An “induction” law for modulus arithmetic ............... 875
  54.5 Division on int .................................................. 879
    54.5.1 Uniqueness and Monotonicity of Quotients and Remainders 881
    54.5.2 Correctness of posDivAlg, the Algorithm for Non-Negative Dividends 881
    54.5.3 Correctness of negDivAlg, the Algorithm for Negative Dividends 882
    54.5.4 Existence Shown by Proving the Division Algorithm to be Correct 883
    54.5.5 General Properties of div and mod ........................ 885
    54.5.6 Laws for div and mod with Unary Minus .................... 886
    54.5.7 Computation of Division and Remainder ................... 887
    54.5.8 Monotonicity in the First Argument (Dividend) ........... 889
    54.5.9 Monotonicity in the Second Argument (Divisor) .......... 890
    54.5.10 More Algebraic Laws for div and mod ..................... 891
    54.5.11 Proving \( a \div (b \times c) = a \div b \div c \) ............... 892
    54.5.12 Splitting Rules for div and mod .......................... 894
    54.5.13 Computing div and mod with shifting ..................... 895
54.5.14 Quotients of Signs ........................................ 897
54.5.15 The Divides Relation ..................................... 898
54.5.16 Further properties ........................................ 898
54.5.17 Tools setup ............................................... 902
54.5.18 Code generation .......................................... 902

55 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions 903

56 Semiring-Normalization: Semiring normalization 909

57 Groebner-Basis: Groebner bases 914
   57.1 Groebner Bases ........................................... 914

58 Lattices-Big: Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets 916
   58.1 Generic lattice operations over a set ................. 916
   58.1.1 Without neutral element .............................. 916
   58.1.2 With neutral element ................................. 920
   58.2 Lattice operations on finite sets ...................... 922
   58.3 Infimum and Supremum over non-empty sets .......... 923
   58.4 Minimum and Maximum over non-empty sets .......... 926

59 Set-Interval: Set intervals .................................. 932
   59.1 Various equivalences .................................... 934
   59.2 Logical Equivalences for Set Inclusion and Equality .. 935
   59.3 Two-sided intervals .................................... 936
      59.3.1 Emptyness, singletons, subset .................... 936
   59.4 Infinite intervals ...................................... 942
      59.4.1 Intersection ....................................... 944
   59.5 Intervals of natural numbers ......................... 945
      59.5.1 The Constant lessThan ............................. 945
      59.5.2 The Constant greaterThan .......................... 946
      59.5.3 The Constant atLeast .............................. 946
      59.5.4 The Constant atMost ................................ 947
      59.5.5 The Constant atLeastLessThan .................... 947
      59.5.6 Intervals of nats with Suc ........................ 947
      59.5.7 Intervals and numerals .............................. 948
      59.5.8 Image ............................................... 948
      59.5.9 Finiteness .......................................... 951
      59.5.10 Proving Inclusions and Equalities between Unions .. 952
      59.5.11 Cardinality ...................................... 953
   59.6 Intervals of integers .................................. 955
      59.6.1 Finiteness .......................................... 955
59.6.2 Cardinality .............................................. 956
59.7 Lemmas useful with the summation operator setsum ........... 957
   59.7.1 Disjoint Unions ..................................... 958
   59.7.2 Disjoint Intersections ............................... 958
   59.7.3 Some Differences .................................... 959
   59.7.4 Some Subset Conditions ............................. 959
59.8 Summation indexed over intervals ............................ 959
59.9 Shifting bounds ......................................... 962
59.10 The formula for geometric sums ........................... 964
59.11 The formula for arithmetic sums ........................... 964
59.12 Products indexed over intervals ........................... 966
59.13 Transfer setup ........................................... 967

60 Presburger: Decision Procedure for Presburger Arithmetic968
   60.1 The $-\infty$ and $+\infty$ Properties ..................... 968
   60.2 The A and B sets ...................................... 969
   60.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version .......... 972
      60.3.1 First some trivial facts about periodic sets or predicates 972
      60.3.2 The $-\infty$ Version ................................ 973
      60.3.3 The $+\infty$ Version ................................ 974
   60.4 Try0 ................................................. 978

61 SMT2: Bindings to Satisfiability Modulo Theories (SMT)
solvers based on SMT-LIB 2 .................................. 978
   61.1 Triggers for quantifier instantiation ..................... 978
   61.2 Higher-order encoding ................................ 978
   61.3 Normalization ......................................... 979
   61.4 Integer division and modulo for Z3 ...................... 979
   61.5 Setup ................................................ 979
   61.6 Configuration ......................................... 980
   61.7 General configuration options ........................ 980
   61.8 Certificates .......................................... 981
   61.9 Tracing .............................................. 981
   61.10 Schematic rules for Z3 proof reconstruction ........... 982

62 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup ............... 984

63 Code-Numeral: Numeric types for code generation onto target
language numerals only ........................................ 985
   63.1 Type of target language integers ....................... 985
   63.2 Code theorems for target language integers ............ 990
   63.3 Serializer setup for target language integers .......... 996
   63.4 Type of target language naturals ....................... 998
   63.5 Inductive representation of target language naturals ... 1001
63.6 Code refinement for target language naturals 1002

64 Lifting-Set: Setup for Lifting/Transfer for the set type 1004
64.1 Relator and predicator properties 1004
64.2 Quotient theorem for the Lifting package 1006
64.3 Transfer rules for the Transfer package 1007
   64.3.1 Unconditional transfer rules 1007
   64.3.2 Rules requiring bi-unique, bi-total or right-total relations 1008

64.1 Relator and predicator properties 1010
64.2 Transfer rules for the Transfer package 1011

65 Lifting-Option: Setup for Lifting/Transfer for the option type 1010
65.1 Relator and predicator properties 1010
65.2 Transfer rules for the Transfer package 1011

66 Lifting-Product: Setup for Lifting/Transfer for the product type 1011
66.1 Transfer rules for the Transfer package 1011

67 List: The datatype of finite lists 1012
67.1 Basic list processing functions 1013
   67.1.1 List comprehension 1019
   67.1.2 [] and op # 1027
   67.1.3 length 1028
   67.1.4 @ – append 1031
   67.1.5 map 1033
   67.1.6 rev 1036
   67.1.7 set 1037
   67.1.8 filter 1041
   67.1.9 List partitioning 1043
   67.1.10 concat 1044
   67.1.11 op ! 1045
   67.1.12 list-update 1048
   67.1.13 last and butlast 1050
   67.1.14 take and drop 1052
   67.1.15 takeWhile and dropWhile 1057
   67.1.16 zip 1061
   67.1.17 list-all2 1064
   67.1.18 List.product and product-lists 1067
   67.1.19 fold with natural argument order 1068
   67.1.20 Fold variants: foldr and foldl 1071
   67.1.21 uptime 1072
   67.1.22 uptime: interval-list on int 1075
   67.1.23 distinct and remdups and remdups-adj 1076
67.1.24 List summation: \textit{listsum} and $\sum$  
67.1.25 \textit{insert}  
67.1.26 \textit{List.union}  
67.1.27 \textit{List.find}  
67.1.28 \textit{List.extract}  
67.1.29 \textit{remove1}  
67.1.30 \textit{removeAll}  
67.1.31 \textit{replicate}  
67.1.32 \textit{enumerate}  
67.1.33 \textit{rotate1} and \textit{rotate}  
67.1.34 \textit{sublist} — a generalization of \textit{op !} to sets  
67.1.35 \textit{sublists} and \textit{List.n-lists}  
67.1.36 \textit{splice}  
67.1.37 Transpose  
67.1.38 (In)\textit{finiteness}  

67.2 Sorting  
67.2.1 \textit{transpose} on sorted lists  
67.2.2 \textit{sorted-list-of-set}  
67.2.3 \textit{sorted-list-of-set}  
67.2.4 \textit{lists}: the list-forming operator over sets  
67.2.5 Inductive definition for membership  
67.2.6 Lists as Cartesian products  

67.3 Relations on Lists  
67.3.1 Length Lexicographic Ordering  
67.3.2 Lexicographic Ordering  
67.3.3 Lexicographic combination of measure functions  
67.3.4 Lifting Relations to Lists: one element  
67.3.5 Lifting Relations to Lists: all elements  

67.4 Size function  
67.5 Monad operation  
67.6 Transfer  
67.7 Code generation  
67.7.1 Counterparts for set-related operations  
67.7.2 Optimizing by rewriting  
67.7.3 Pretty lists  
67.7.4 Use convenient predefined operations  
67.7.5 Implementation of sets by lists  

67.8 Setup for Lifting/Transfer  
67.8.1 Transfer rules for the Transfer package  

68 Random: A HOL random engine  
68.1 Auxiliary functions  
68.2 Random seeds  
68.3 Base selectors
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>74.3.1</td>
<td>General lazy sequence operation</td>
<td>1216</td>
</tr>
<tr>
<td>74.3.2</td>
<td>Small lazy typeclasses</td>
<td>1216</td>
</tr>
<tr>
<td>74.4</td>
<td>With Hit Bound Value</td>
<td>1217</td>
</tr>
<tr>
<td>75</td>
<td>Limited-Sequence: Depth-Limited Sequences with failure element</td>
<td>1219</td>
</tr>
<tr>
<td>75.1</td>
<td>Depth-Limited Sequence</td>
<td>1219</td>
</tr>
<tr>
<td>75.2</td>
<td>Positive Depth-Limited Sequence</td>
<td>1220</td>
</tr>
<tr>
<td>75.3</td>
<td>Negative Depth-Limited Sequence</td>
<td>1221</td>
</tr>
<tr>
<td>75.4</td>
<td>Negation</td>
<td>1222</td>
</tr>
<tr>
<td>76</td>
<td>Code-Evaluation: Term evaluation using the generic code generator</td>
<td>1223</td>
</tr>
<tr>
<td>76.1</td>
<td>Term representation</td>
<td>1224</td>
</tr>
<tr>
<td>76.1.1</td>
<td>Terms and class term-of</td>
<td>1224</td>
</tr>
<tr>
<td>76.1.2</td>
<td>Syntax</td>
<td>1224</td>
</tr>
<tr>
<td>76.2</td>
<td>Tools setup and evaluation</td>
<td>1225</td>
</tr>
<tr>
<td>76.3</td>
<td>term-of instances</td>
<td>1225</td>
</tr>
<tr>
<td>76.4</td>
<td>Generic reification</td>
<td>1226</td>
</tr>
<tr>
<td>76.5</td>
<td>Diagnostic</td>
<td>1226</td>
</tr>
<tr>
<td>77</td>
<td>Quickcheck-Random: A simple counterexample generator performing random testing</td>
<td>1226</td>
</tr>
<tr>
<td>77.1</td>
<td>Catching Match exceptions</td>
<td>1227</td>
</tr>
<tr>
<td>77.2</td>
<td>The random class</td>
<td>1227</td>
</tr>
<tr>
<td>77.3</td>
<td>Fundamental and numeric types</td>
<td>1227</td>
</tr>
<tr>
<td>77.4</td>
<td>Complex generators</td>
<td>1229</td>
</tr>
<tr>
<td>77.5</td>
<td>Deriving random generators for datatypes</td>
<td>1231</td>
</tr>
<tr>
<td>77.6</td>
<td>Code setup</td>
<td>1231</td>
</tr>
<tr>
<td>78</td>
<td>Random-Pred: The Random-Predicate Monad</td>
<td>1231</td>
</tr>
<tr>
<td>79</td>
<td>Random-Sequence: Various kind of sequences inside the random monad</td>
<td>1233</td>
</tr>
<tr>
<td>80</td>
<td>Quickcheck-Exhaustive: A simple counterexample generator performing exhaustive testing</td>
<td>1237</td>
</tr>
<tr>
<td>80.1</td>
<td>basic operations for exhaustive generators</td>
<td>1237</td>
</tr>
<tr>
<td>80.2</td>
<td>exhaustive generator type classes</td>
<td>1237</td>
</tr>
<tr>
<td>80.2.1</td>
<td>A smarter enumeration scheme for functions over finite datatypes</td>
<td>1242</td>
</tr>
<tr>
<td>80.3</td>
<td>Bounded universal quantifiers</td>
<td>1248</td>
</tr>
<tr>
<td>80.4</td>
<td>Fast exhaustive combinators</td>
<td>1248</td>
</tr>
<tr>
<td>80.5</td>
<td>Continuation passing style functions as plus monad</td>
<td>1249</td>
</tr>
<tr>
<td>80.6</td>
<td>Defining generators for any first-order data type</td>
<td>1251</td>
</tr>
</tbody>
</table>
80.7 Defining generators for abstract types 

81 Predicate-Compile: A compiler for predicates defined by introduction rules  
81.1 Set membership as a generator predicate

82 Quickcheck-Narrowing: Counterexample generator performing narrowing-based testing  
82.1 Counterexample generator  
82.1.1 Code generation setup  
82.1.2 Narrowing’s deep representation of types and terms  
82.1.3 From narrowing’s deep representation of terms to Code-Evaluation’s terms  
82.1.4 Auxilary functions for Narrowing  
82.1.5 Narrowing’s basic operations  
82.1.6 Narrowing generator type class  
82.1.7 class is-testable  
82.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way  
82.1.9 Setting up the counterexample generator  
82.2 Narrowing for sets  
82.3 Narrowing for integers  
82.4 The find-unused-assms command  
82.5 Closing up

83 Extraction: Program extraction for HOL  
83.1 Setup  
83.2 Type of extracted program  
83.3 Realizability  
83.4 Computational content of basic inference rules

84 Lifting-Sum: Setup for Lifting/Transfer for the sum type
84.1 Transfer rules for the Transfer package

85 Coinduction: Coinduction Method

86 Record: Extensible records with structural subtyping  
86.1 Introduction  
86.2 Operators and lemmas for types isomorphic to tuples  
86.3 Logical infrastructure for records  
86.4 Concrete record syntax  
86.5 Record package

87 Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL
88 BNF-GFP: Greatest Fixed Point Operation on Bounded Natural Functors

88.1 Equivalence relations, quotients, and Hilbert’s choice

89 SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers

89.1 Triggers for quantifier instantiation
89.2 Quantifier weights
89.3 Higher-order encoding
89.4 First-order logic
89.5 Integer division and modulo for Z3
89.6 Setup
89.7 Configuration
89.8 General configuration options
89.9 Certificates
89.10 Tracing
89.11 Schematic rules for Z3 proof reconstruction

90 Main: Main HOL

91 Fact: Factorial Function

91.1 Set up Transfer
91.2 Factorial
91.3 fact and of-nat

92 Parity: Even and Odd for int and nat

92.1 Even and odd are mutually exclusive
92.2 Behavior under integer arithmetic operations
92.3 Equivalent definitions
92.4 even and odd for nats
92.5 Equivalent definitions
92.6 Parity and powers
92.7 More Even/Odd Results
92.8 An Equivalence for \( 0 \leq a^n \)
92.9 Miscellaneous

93 GCD: Greatest common divisor and least common multiple

93.1 GCD and LCM definitions
93.2 Transfer setup
93.3 GCD properties
93.4 Coprimality
93.5 Bezout’s theorem
93.6 LCM properties
93.7 The complete divisibility lattice
93.7.1 Setwise gcd and lcm for integers . . . . . . . . . . . . 1346

94 Archimedean-Field: Archimedean Fields, Floor and Ceiling Functions 1347
94.1 Class of Archimedean fields . . . . . . . . . . . . . . . . . . . 1347
94.2 Existence and uniqueness of floor function . . . . . . . . . 1349
94.3 Floor function . . . . . . . . . . . . . . . . . . . . . . . . . . 1350
94.4 Ceiling function . . . . . . . . . . . . . . . . . . . . . . . . . 1353
94.5 Negation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1356

95 Rat: Rational numbers 1356
95.1 Rational numbers as quotient . . . . . . . . . . . . . . . . . 1356
95.1.1 Construction of the type of rational numbers . . . . . . 1356
95.1.2 Representation and basic operations . . . . . . . . . . . 1357
95.1.3 Function normalize . . . . . . . . . . . . . . . . . . . . 1361
95.1.4 Various . . . . . . . . . . . . . . . . . . . . . . . . . . 1364
95.1.5 The ordered field of rational numbers . . . . . . . . . . 1364
95.1.6 Rationals are an Archimedean field . . . . . . . . . . . 1368
95.2 Linear arithmetic setup . . . . . . . . . . . . . . . . . . . . . 1369
95.3 Embedding from Rationals to other Fields . . . . . . . . . . 1369
95.4 The Set of Rational Numbers . . . . . . . . . . . . . . . . . 1372
95.5 Implementation of rational numbers as pairs of integers . . 1375
95.6 Setup for Nitpick . . . . . . . . . . . . . . . . . . . . . . . . 1379
95.7 Float syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 1380
95.8 Hiding implementation details . . . . . . . . . . . . . . . . . 1380

96 Conditionally-Complete-Lattices: Conditionally-complete Lattices 1381

97 Real: Development of the Reals using Cauchy Sequences 1394
97.1 Preliminary lemmas . . . . . . . . . . . . . . . . . . . . . . . 1394
97.2 Sequences that converge to zero . . . . . . . . . . . . . . . 1395
97.3 Cauchy sequences . . . . . . . . . . . . . . . . . . . . . . . . 1397
97.4 Equivalence relation on Cauchy sequences . . . . . . . . . . 1401
97.5 The field of real numbers . . . . . . . . . . . . . . . . . . . 1402
97.6 Positive reals . . . . . . . . . . . . . . . . . . . . . . . . . . 1404
97.7 Completeness . . . . . . . . . . . . . . . . . . . . . . . . . . 1408
97.8 Supremum of a set of reals . . . . . . . . . . . . . . . . . . . 1413
97.9 Hiding implementation details . . . . . . . . . . . . . . . . 1414
97.10 More Lemmas . . . . . . . . . . . . . . . . . . . . . . . . . . 1414
97.11 Embedding numbers into the Reals . . . . . . . . . . . . . . 1414
97.12 Embedding the Naturals into the Reals . . . . . . . . . . . . 1418
97.13 The Archimedean Property of the Reals . . . . . . . . . . . 1421
97.14 Rationals . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1421
97.15 Density of the Rational Reals in the Reals  1423
97.16 Numerals and Arithmetic  1423
97.17 Simprules combining x+y and 0: ARE THEY NEEDED?  1424
97.18 Lemmas about powers  1425
97.19 Density of the Reals  1426
97.20 Absolute Value Function for the Reals  1426
97.21 Floor and Ceiling Functions from the Reals to the Integers  1427
97.21.1 Versions for the natural numbers  1431
97.22 Exponentiation with floor  1435
97.23 Implementation of rational real numbers  1436
97.24 Setup for Nitpick  1439
97.25 Setup for SMT  1439

98 Topological-Spaces: Topological Spaces  1440
98.1 Topological space  1440
98.2 Hausdorff and other separation properties  1443
98.3 Generators for topologies  1444
98.4 Order topologies  1444
98.5 Filters  1446
  98.5.1 Eventually  1447
  98.5.2 Finer-than relation  1449
  98.5.3 Map function for filters  1453
  98.5.4 Standard filters  1454
  98.5.5 Order filters  1455
98.6 Sequentially  1457
  98.6.1 Topological filters  1458
98.7 Limits  1461
  98.7.1 Tendsto  1463
  98.7.2 Rules about \( \text{Lim} \)  1467
98.8 Limits to \( \text{at-top} \) and \( \text{at-bot} \)  1467
98.9 Limits on sequences  1470
  98.9.1 Monotone sequences and subsequences  1471
  98.9.2 Increasing and Decreasing Series  1476
98.10 First countable topologies  1476
98.11 Function limit at a point  1479
  98.11.1 Relation of LIM and LIMSEQ  1480
98.12 Continuity  1483
  98.12.1 Continuity on a set  1483
  98.12.2 Continuity at a point  1486
  98.12.3 Open-cover compactness  1488
98.13 Connectedness  1493
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>101.7</td>
<td>Power Sequences</td>
<td>1570</td>
</tr>
<tr>
<td>101.8</td>
<td>Limits of Functions</td>
<td>1571</td>
</tr>
<tr>
<td>101.9</td>
<td>Continuity</td>
<td>1573</td>
</tr>
<tr>
<td>101.10</td>
<td>Uniform Continuity</td>
<td>1575</td>
</tr>
<tr>
<td>101.11</td>
<td>Nested Intervals and Bisection – Needed for Compactness</td>
<td>1576</td>
</tr>
<tr>
<td>101.12</td>
<td>Boundedness of continuous functions</td>
<td>1578</td>
</tr>
</tbody>
</table>

**102 Series: Infinite Series**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>102.1</td>
<td>Definition of infinite summability</td>
<td>1581</td>
</tr>
<tr>
<td>102.2</td>
<td>Infinite summability on topological monoids</td>
<td>1582</td>
</tr>
<tr>
<td>102.3</td>
<td>Infinite summability on ordered, topological monoids</td>
<td>1584</td>
</tr>
<tr>
<td>102.4</td>
<td>Infinite summability on real normed vector spaces</td>
<td>1586</td>
</tr>
<tr>
<td>102.5</td>
<td>Infinite summability on real normed algebras</td>
<td>1589</td>
</tr>
<tr>
<td>102.6</td>
<td>Infinite summability on real normed fields</td>
<td>1589</td>
</tr>
<tr>
<td>102.7</td>
<td>Infinite summability on Banach spaces</td>
<td>1590</td>
</tr>
<tr>
<td>102.8</td>
<td>The Ratio Test</td>
<td>1592</td>
</tr>
<tr>
<td>102.9</td>
<td>Cauchy Product Formula</td>
<td>1593</td>
</tr>
<tr>
<td>102.18</td>
<td>Series on reals</td>
<td>1595</td>
</tr>
</tbody>
</table>

**103 Deriv: Differentiation**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>103.1</td>
<td>Frechet derivative</td>
<td>1596</td>
</tr>
<tr>
<td>103.2</td>
<td>Continuity</td>
<td>1600</td>
</tr>
<tr>
<td>103.3</td>
<td>Composition</td>
<td>1600</td>
</tr>
<tr>
<td>103.4</td>
<td>Uniqueness</td>
<td>1605</td>
</tr>
<tr>
<td>103.5</td>
<td>Differentiability predicate</td>
<td>1606</td>
</tr>
<tr>
<td>103.6</td>
<td>Derivatives</td>
<td>1609</td>
</tr>
<tr>
<td>103.7</td>
<td>Local extrema</td>
<td>1614</td>
</tr>
<tr>
<td>103.8</td>
<td>Rolle’s Theorem</td>
<td>1617</td>
</tr>
<tr>
<td>103.9</td>
<td>Mean Value Theorem</td>
<td>1619</td>
</tr>
<tr>
<td>103.16</td>
<td>Generalized Mean Value Theorem</td>
<td>1626</td>
</tr>
<tr>
<td>103.1L</td>
<td>L’Hopital’s rule</td>
<td>1628</td>
</tr>
</tbody>
</table>

**104 NthRoot: Nth Roots of Real Numbers**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>104.1</td>
<td>Existence of Nth Root</td>
<td>1634</td>
</tr>
<tr>
<td>104.2</td>
<td>Nth Root</td>
<td>1635</td>
</tr>
<tr>
<td>104.3</td>
<td>Square Root</td>
<td>1641</td>
</tr>
<tr>
<td>104.4</td>
<td>Square Root of Sum of Squares</td>
<td>1645</td>
</tr>
</tbody>
</table>

**105 Transcendental: Power Series, Transcendental Functions etc**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>105.1</td>
<td>Properties of Power Series</td>
<td>1648</td>
</tr>
<tr>
<td>105.2</td>
<td>Alternating series test / Leibniz formula</td>
<td>1652</td>
</tr>
<tr>
<td>105.3</td>
<td>Term-by-Term Differentiability of Power Series</td>
<td>1656</td>
</tr>
<tr>
<td>105.4</td>
<td>Derivability of power series</td>
<td>1662</td>
</tr>
<tr>
<td>105.5</td>
<td>Exponential Function</td>
<td>1667</td>
</tr>
</tbody>
</table>
1 Code-Generator: Loading the code generator and related modules

theory Code-Generator
imports Pure
keywords
print-codeproc code-thms code-deps :: diag and
export-code code-identifier code-printing code-reserved
code-monad code-reflect :: thy-decl and
datatypes functions module-name file checking
constant type-constructor type-class class-relation class-instance code-module
begin
ML-file ~/src/Tools/cache-io.ML
ML-file ~/src/Tools/Code/code-preproc.ML

attribute-setup code-preproc-trace = ⟨⟨
(Scan.lift (Args.$$ off >> K Code-Preproc.trace-none)
|| (Scan.lift (Args.$$ only |-- Args.colon |-- Scan.repeat1 Parse.term))
>> Code-Preproc.trace-only-ext
|| Scan.succeed Code-Preproc.trace-all)
>> (Thm.declaration-attribute o K)
⟩⟩ tracing of the code generator preprocessor

ML-file ~/src/Tools/Code/code-symbol.ML
ML-file ~/src/Tools/Code/code-thingol.ML
ML-file ~/src/Tools/Code/code-simp.ML
ML-file ~/src/Tools/Code/code-printer.ML
ML-file ~/src/Tools/Code/code-target.ML
ML-file ~/src/Tools/Code/code-namespace.ML
ML-file ~/src/Tools/Code/code-ml.ML
ML-file ~/src/Tools/Code/code-haskell.ML
ML-file ~/src/Tools/Code/code-scala.ML

setup ⟨⟨
  Code-Simp.setup
#> Code-Target.setup
#> Code-ML.setup
#> Code-Haskell.setup
#> Code-Scala.setup
⟩⟩

code-datatype TYPE('a::{})

definition holds :: prop where
  holds ≡ ((λx::prop. x) ≡ (λx. x))

lemma holds: PROP holds
by (unfold holds-def) (rule reflexive)
**THEORY “HOL”**

**THEORY “HOL”**

code-datatype holds

lemma implies-code [code]:

\[(PROP \text{ holds } \implies \text{ PROP P}) \equiv \text{ PROP P} \]

\[(PROP P \implies \text{ PROP holds}) \equiv \text{ PROP holds} \]

proof –

show \((PROP \text{ holds } \implies \text{ PROP P}) \equiv \text{ PROP P}\)

proof

assume \(PROP \text{ holds } \implies \text{ PROP P}\)

then show \(PROP P\) using holds .

next

assume \(PROP P\)

then show \(PROP P\).

qed

next

show \((PROP P \implies \text{ PROP holds}) \equiv \text{ PROP holds}\)

by rule (rule holds)+

qed

ML-file ~~/src/Tools/Code/code-runtime.ML

ML-file ~~/src/Tools/nbe.ML

hide-const (open) holds

end

2 **HOL: The basis of Higher-Order Logic**

theory HOL

imports Pure ~~/src/Tools/Code-Generator

keywords

try solve-direct quickcheck print-coercions print-claset

print-induct-rules :: diag and

quickcheck-params :: thy-decl

begin

ML-file ~~/src/Tools/misc-legacy.ML

ML-file ~~/src/Tools/try.ML

ML-file ~~/src/Tools/quickcheck.ML

ML-file ~~/src/Tools/solve-direct.ML

ML-file ~~/src/Tools/IsaPlanner/zipper.ML

ML-file ~~/src/Tools/IsaPlanner/isand.ML

ML-file ~~/src/Tools/IsaPlanner/rw-inst.ML

ML-file ~~/src/Provers/hypsubst.ML

ML-file ~~/src/Provers/splitter.ML

ML-file ~~/src/Provers/classical.ML

ML-file ~~/src/Provers/blast.ML

ML-file ~~/src/Provers/clasimp.ML
ML-file ∼∼/src/Tools/eqsubst.ML
ML-file ∼∼/src/Provers/quantifier1.ML
ML-file ∼∼/src/Tools/atomize-elim.ML
ML-file ∼∼/src/Tools/induct.ML
ML-file ∼∼/src/Tools/cong-tac.ML
ML-file ∼∼/src/Tools/intuitionistic.ML
ML-file ∼∼/src/Tools/project-rule.ML
ML-file ∼∼/src/Tools/subtyping.ML
ML-file ∼∼/src/Tools/case-product.ML

setup ⟨⟨ Intuitionistic.method-setup @{binding iprover} ⟩⟩
≥ Subtyping.setup
≥ Case-Product.setup
⟩⟩

2.1 Primitive logic
2.1.1 Core syntax

setup ⟨⟨ Axclass.class-axiomatization @{binding type}, [] ⟩⟩
default-sort type
setup ⟨⟨ Object-Logic.add-base-sort @{sort type} ⟩⟩

axiomatization where fun-arity: OFCLASS('a ⇒ 'b, type-class)
instance fun :: (type, type) type by (rule fun-arity)

axiomatization where itself-arity: OFCLASS('a itself, type-class)
instance itself :: (type) type by (rule itself-arity)

typedecl bool

judgment
    Trueprop :: bool => prop                       ((-5))

axiomatization
    implies :: [bool, bool] => bool               (infixr −− 25) and
    eq :: ['a, 'a] => bool                       (infixl = 50) and
    The :: ('a => bool) => 'a

consts
    True :: bool
    False :: bool
    Not :: bool => bool                         (~ [40] 40)
    conj :: [bool, bool] => bool                (infixr & 35)
    disj :: [bool, bool] => bool                (infixr | 30)
    All :: ('a => bool) => bool                 (binder ALL 10)
    Ex :: ('a => bool) => bool                  (binder EX 10)
2.1.2 Additional concrete syntax

notation (output)
  eq (infix = 50)

abbreviation
  not-equal :: ['a, 'a] => bool (infixl ~ = 50) where
  x ~ = y == ~ (x = y)

notation (output)
  not-equal (infix ~ = 50)

notation (xsymbols output)
  not-equal (infix ~ = 50)

notation (HTML output)
  not-equal (infix ~ = 50)

abbreviation (iff)
  iff :: [bool, bool] => bool (infixr <-> 25) where
  A <-> B == A = B

notation (xsymbols)
  iff (infixr <-> 25)

syntax -The :: [pttrn, bool] => 'a ((3THE ./ -) [0, 10] 10)
translations THE x. P == CONST The (%x. P)
print-translation ⟨⟨
  ([@{const-syntax The}, fn - => fn [Abs abs] =>
    let val (x, t) = Syntax-Trans.atomic-abs-tr' abs
    in Syntax.const @(syntax-const -The} $ x $ t end])
⟩⟩ — To avoid eta-contraction of body

nonterminal letbinds and letbind

syntax
  -bind :: [pttrn, 'a] => letbind ((2- =/ -) 10)
  :: letbind => letbinds (-)
THEORY "HOL"

-binds :: [letbind, letbinds] => letbinds (-/ -)
-let :: [letbinds, 'a] => 'a ((let (-)/ in (-)) [0, 10] 10)

nonterminal case-syn and cases-syn

syntax
-case-syntax :: ['a, cases-syn] => 'b ((case - of / -) 10)
-case1 :: ['a, 'b] => case-syn ((2. =>/ -) 10)
  :: case-syn => cases-syn (-)
-case2 :: [case-syn, cases-syn] => cases-syn (-/ | -)

syntax (xsymbols)
-case1 :: ['a, 'b] => case-syn ((2. =>/ -) 10)

notation (xsymbols)
All (binder ∀ 10) and
Ex (binder ∃ 10) and
Ex1 (binder ∃! 10)

notation (HTML output)
All (binder ∀ 10) and
Ex (binder ∃ 10) and
Ex1 (binder ∃! 10)

notation (HOL)
All (binder ′ 10) and
Ex (binder ″ 10) and
Ex1 (binder ′′ 10)

2.1.3 Axioms and basic definitions

axiomatization where
refl: t = (t: 'a) and
subst: s = t => P s => P t and
ext: (!x:'a. (f x :: 'b) = g x) =>> (%x. f x) = (%x. g x)

the-eq-trivial: (THE x. x = a) = (a::'a)

axiomatization where
impI: (P =>> Q) =>> P => Q and
mp: || P =>> Q; P || =>> Q and
iff: (P =>> Q) =>> (Q =>> P) =>> (P=Q) and
True-or-False: (P=True) | (P=False)

defs
True-def: True == ((%x:bool. x) = (%x. x))
All-def: All(P) == (P = (%x. True))
Ex-def: \( \text{Ex}(P) == \neg Q \land (\forall x. P x \rightarrow Q) \rightarrow Q \)

False-def: \( \text{False} == (\neg P) \)

not-def: \( \neg P == P \rightarrow \text{False} \)

and-def: \( P \land Q == \neg R \land (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \)

or-def: \( P \lor Q == \neg R \land (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \)

Ex1-def: \( \text{Ex1}(P) == (\exists x. P x) \land \neg (\forall y. P y) \land (y = x) \)

definition \text{If} :: \text{bool} \rightarrow 'a \rightarrow 'a \rightarrow 'a ((\text{if} (-) \text{ then} (-) \text{ else} (-)) [0, 0, 10] 10)

where \text{If} P x y \equiv (\text{THE} z::'a. (P=\text{True} \rightarrow z=x) \& (P=\text{False} \rightarrow z=y))

definition \text{Let} :: 'a \rightarrow ('a \rightarrow 'b) \rightarrow 'b

where \text{Let} s f \equiv f s

translations
\(-\text{Let} ((\text{-binds} b \ bs) \ e) == -\text{Let} b ((\text{-Let} bs \ e)) \)

let \( x = a \) in \( e \) \( == \) \text{CONST} \( \text{Let} a (\%x. e) \)

axiomatization \text{undefined} :: 'a

class \text{default} = \text{fixes} default :: 'a

2.2 Fundamental rules

2.2.1 Equality

lemma sym: \( s = t \rightarrow t = s \)

by (erule subst) (rule refl)

lemma ssubst: \( t = s \rightarrow P s \rightarrow P t \)

by (erule sym) (erule subst)

lemma trans: \( \left[ r = s; s = t \right] \rightarrow r = t \)

by (erule subst)

lemma trans-sym \( \text{[Pure.elim?]}: r = s \rightarrow t = s \rightarrow r = t \)

by (rule trans \[ OF - sym\])

lemma meta-eq-to-obj-eq:

assumes meq: \( A == B \)

shows \( A = B \)

by (unfold meq) (rule refl)

Useful with erule for proving equalities from known equalities.

lemma box-equals: \( \left[ a = b; a = c; b = d \right] \rightarrow c = d \)

apply (rule trans)

apply (rule trans)

apply (rule sym)

apply assumption+

done
For calculational reasoning:

**lemma** forw-subst: \( a = b \implies P \) \( b \implies P \) \( a \)
   by (rule ss subst)

**lemma** back-subst: \( P \) \( a \implies a = b \implies P \) \( b \)
   by (rule subst)

### 2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon’s HOL.

**lemma** fun-cong: \( (f::'a=>'b) = g \implies f(x)=g(x) \)
apply (erule subst)
apply (rule refl)
done

Similar to *AP-TERM* in Gordon’s HOL and FOL’s *subst-context*.

**lemma** arg-cong: \( x=y \implies f(x)=f(y) \)
apply (erule subst)
apply (rule refl)
done

**lemma** arg-cong2: \[ a = b; c = d \] \( \implies f a c = f b d \)
apply (erule ss subst)+
apply (rule refl)
done

**lemma** cong: \[ f = g; (x::'a) = y \] \( \implies f x = g y \)
apply (erule subst)+
apply (rule refl)
done

ML \( \langle\langle\text{val cong-tac = Cong-Tac.cong-tac @\{thm cong\}}\rangle\rangle \)

### 2.2.3 Equality of booleans – iff

**lemma** iffI: assumes \( P \implies Q \) and \( Q \implies P \) shows \( P = Q \)
by (iprover intro: iff [THEN mp, THEN mp] impI assms)

**lemma** iffD2: \[ P=Q; Q \] \( \implies P \)
by (erule ss subst)

**lemma** rev-iffD2: \[ Q; P=Q \] \( \implies Q \)
by (erule iffD2)

**lemma** iffD1: \( Q = P \implies Q \implies P \)
by (drule sym) (rule iffD2)

**lemma** rev-iffD1: \( Q \implies Q = P \implies P \)
THEORY “HOL”

by (drule sym) (rule rev-iffD2)

lemma iffE:
assumes major: \( P = Q \)
  and minor: \( P \longrightarrow Q ; Q \longrightarrow P \) \( \implies \) \( R \)
shows \( R \)
by (iprover intro: minor impI major [THEN iffD2] major [THEN iffD1])

2.2.4 True

lemma TrueI: \( \text{True} \)
  unfolding True-def by (rule refl)

lemma eqTrueI: \( P \implies P = \text{True} \)
  by (iprover intro: iffI TrueI)

lemma eqTrueE: \( P = \text{True} \implies P \)
  by (erule iffD2) (rule TrueI)

2.2.5 Universal quantifier

lemma allI: assumes \(!x::'a. P(x)\) shows \( \forall x. P(x) \)
  unfolding All-def by (iprover intro: ext eqTrueI assms)

lemma spec: \( \forall x::'a. P(x) \implies P(x) \)
apply (unfold All-def)
apply (rule eqTrueE)
apply (erule fun-cong)
done

lemma allE:
assumes major: \( \forall x. P(x) \)
  and minor: \( P(x) \implies R \)
shows \( R \)
by (iprover intro: minor major [THEN spec])

lemma all-dupE:
assumes major: \( \forall x. P(x) \)
  and minor: \( \forall x; \forall x. P(x) \) \( \implies \) \( R \)
shows \( R \)
by (iprover intro: minor major major [THEN spec])

2.2.6 False

Depends upon spec; it is impossible to do propositional logic before quantifiers!

lemma FalseE: \( \text{False} \implies P \)
apply (unfold False-def)
apply (erule spec)
done

lemma False-neq-True: \( \text{False} = \text{True} \implies P \)
by (erule eqTrueE [THEN FalseE])

2.2.7 Negation

lemma notI:
  assumes \( P \implies \text{False} \)
  shows \( \neg P \)
  apply (unfold not-def)
  apply (iprover intro: impI assms)
  done

lemma False-not-True: \( \text{False} \neg= \text{True} \)
apply (rule notI)
apply (erule False-neq-True)
done

lemma True-not-False: \( \text{True} \neg= \text{False} \)
apply (rule notI)
apply (drule sym)
apply (erule False-neq-True)
done

lemma notE: \[ \neg P; \ P \] \( \implies R \)
apply (unfold not-def)
apply (erule mp [THEN FalseE])
apply assumption
done

lemma notI2: \( P \implies \neg Pa \implies (P \implies Pa) \implies \neg P \)
by (erule notE [THEN notI]) (erule meta-mp)

2.2.8 Implication

lemma impE:
  assumes \( P \implies Q \implies Q \implies R \)
  shows \( R \)
  by (iprover intro: assms mp)

lemma rev-mp: \[ P; \ P \implies Q \] \( \implies Q \)
by (iprover intro: mp)

lemma contrapos-nn:
  assumes major: \( \neg Q \)
  and minor: \( P \implies Q \)
  shows \( \neg P \)
  by (iprover intro: notI minor major [THEN notE])
lemma contrapos-pn:
assumes major: \( Q \)
and minor: \( P \implies \neg Q \)
shows \( \neg P \)
by (iprover intro: notI minor major notE)

lemma not-sym: \( t \sim s \implies s \sim t \)
by (erule contrapos-nn) (erule sym)

lemma eq-neq-eq-imp-neq: \( | x = a ; a \sim b ; b = y | \implies x \sim y \)
by (erule subst, erule ssubst, assumption)

2.2.9 Existential quantifier

lemma exI: \( P x \implies EX x \:: \cdot a. P x \)
apply (unfold Ex-def)
apply (iprover intro: allI allE impI mp)
done

lemma exE:
assumes major: \( EX x \:: \cdot a. P(x) \)
and minor: \( \forall x. P(x) \implies Q \)
shows \( Q \)
apply (rule major [unfolded Ex-def, THEN spec, THEN mp])
apply (iprover intro: impI [THEN allI] minor)
done

2.2.10 Conjunction

lemma conjI: \( | P ; Q | \implies P \& Q \)
apply (unfold and-def)
apply (iprover intro: impI [THEN allI] mp)
done

lemma conjunct1: \( | P \& Q | \implies P \)
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

lemma conjunct2: \( | P \& Q | \implies Q \)
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

lemma conjE:
assumes major: \( P \& Q \)
and minor: \( | P ; Q | \implies R \)
shows \( R \)
apply (rule minor)
apply (rule major [THEN conjunct1])
apply (rule major [THEN conjunct2])
done

lemma context-conjI:
  assumes P P ==> Q shows P & Q
by (iprover intro: conjI assms)

2.2.11 Disjunction

lemma disjI1: P ==> P|Q
apply (unfold or-def)
apply (iprover intro: allI impI mp)
done

lemma disjI2: Q ==> P|Q
apply (unfold or-def)
apply (iprover intro: allI impI mp)
done

lemma disjE:
  assumes major: P|Q
  and minorP: P ==> R
  and minorQ: Q ==> R
  shows R
by (iprover intro: minorP minorQ impI
  major [unfolded or-def, THEN spec, THEN mp, THEN mp])

2.2.12 Classical logic

lemma classical:
  assumes prem: ~P ==> P
  shows P
apply (rule True-or-False [THEN disjE, THEN eqTrueE])
apply assumption
apply (rule notI [THEN prem, THEN eqTrueI])
apply (erule subst)
apply assumption
done

lemmas ccontr = FalseE [THEN classical]

lemma rev-notE:
  assumes premp: P
  and premnt: ~R ==> ~P
  shows R
apply (rule ccontr)
apply (erule notE [OF premnt premp])
done

lemma notnotD: \( \sim \sim P \implies P \)
apply (rule classical)
apply (erule notE)
apply assumption
done

lemma contrapos-pp:
  assumes p1: \( Q \)
  and p2: \( \sim P \implies \sim Q \)
  shows \( P \)
by (iprover intro: classical p1 p2 notE)

2.2.13 Unique existence

lemma ex1I:
  assumes P a (!x. \( P(x) \implies x=a \))
  shows \( \exists! x. P(x) \)
by (unfold Ex1-def, iprover intro: assms exI conjI allI impI)

Sometimes easier to use: the premises have no shared variables. Safe!

lemma ex-ex1I:
  assumes ex-prem: \( \exists! x. P(x) \)
  and eq: (!x y. [| P(x); P(y) |] \implies x=y)
  shows \( \exists! x. P(x) \)
by (iprover intro: ex-prem [THEN exE] ex1I eq)

lemma ex1E:
  assumes major: \( \exists! x. P(x) \)
  and minor: (!x. [| P(x); \( \forall y. P(y) \implies y=x \) |] \implies R)
  shows \( R \)
apply (rule major [unfolded Ex1-def, THEN exE])
apply (erule conjE)
apply (iprover intro: minor)
done

lemma ex1-implies-ex: \( \exists! x. P(x) \implies \exists x. P(x) \)
apply (erule ex1E)
apply (rule ex1)
apply assumption
done

2.2.14 THE: definite description operator

lemma the-equality:
  assumes prema: \( P(a) \)
  and premx: (!x. \( P(x) \implies x=a \))
  shows \( (\text{THE } x. P(x)) = a \)
apply (rule trans [OF - the-eq-trivial])
apply (rule_tac f = The in arg-cong)
apply (rule ext)
apply (rule iffI)
apply (erule premx)
apply (erule ssubst, rule prema)
done

lemma theI:
  assumes P a and !!x. P x ==> x=a
  shows P (THE x. P x)
by (iprover intro: assms the-equality [THEN subst])

lemma theI': EX! x. P x ==> P (THE x. P x)
apply (erule ex1E)
apply (erule theI)
apply (erule allE)
apply (erule mp)
apply assumption
done

lemma theI2:
  assumes P a !!x. P x ==> x=a !!x. P x ==> Q x
  shows Q (THE x. P x)
by (iprover intro: assms theI)

lemma theI2: assumes EX! x. P x \A x. P x ==> Q x shows Q (THE x. P x)
by (iprover intro: assms(2) theI2[where P=P and Q=Q] ex1E[OF assms(1)]
  elim:allE impE)

lemma the1-equality [elim?]: \EX!x. P x; \A a | ==>(THE x. P x) = a
apply (rule the-equality)
apply assumption
apply (erule ex1E)
apply (erule all-dupE)
apply (drule mp)
apply assumption
apply (erule ssubst)
apply (erule allE)
apply (erule mp)
apply assumption
done

lemma the-sym-eq-trivial: (THE y. x=y) = x
apply (rule the-equality)
apply (rule refl)
apply (erule sym)
done
2.2.15 Classical intro rules for disjunction and existential quantifiers

lemma disjCI:
  assumes \( \neg Q \Longrightarrow P \)
  shows \( P \lor Q \)
apply (rule classical)
apply (iprover intro: assms disjI1 disjI2 notI elim: notE)
done

lemma excluded-middle: \( \neg P \lor P \)
by (iprover intro: disjCI)
case distinction as a natural deduction rule. Note that \( \neg P \)
not the first case, not the first
lemma case-split [case-names True False]:
  assumes prem1: \( P \Longrightarrow Q \)
     and prem2: \( \neg P \Longrightarrow Q \)
  shows Q
apply (rule excluded-middle [THEN disjE])
apply (erule prem2)
apply (erule prem1)
done

lemma impCE:
  assumes major: \( P \Longrightarrow Q \)
     and minor: \( \neg P \Longrightarrow R \)
  shows \( Q \)
apply (rule excluded-middle [THEN disjE])
apply (iprover intro: minor major [THEN mp])
done

lemma impCE':
  assumes major: \( P \Longrightarrow Q \)
     and minor: \( Q \Longrightarrow R \)
  shows \( \neg P \Longrightarrow R \)
apply (rule excluded-middle [THEN disjE])
apply (iprover intro: minor major [THEN mp])
done

lemma iffCE:
  assumes major: \( P = Q \)
     and minor: \( \| P \; Q \| \Longrightarrow R \; \| \neg P ; \neg Q \| \Longrightarrow R \)
  shows \( R \)
apply (rule major [THEN iffE])
apply (iprover intro: minor elim: impCE notE)
done
lemma \textit{exCI}:
assumes ALL x. \sim P(x) ==> P(a)
shows EX x. P(x)
apply (rule ccontr)
apply (prove intro: assms exI allI notI notE [of \exists x. P x])
done

2.2.16 Intuitionistic Reasoning

lemma \textit{impE'}:
assumes 1: P ---\rightarrow Q
and 2: Q ---\rightarrow R
and 3: P ---\rightarrow Q ---\rightarrow P
shows R
proof
  from 3 and 1 have P .
  with 1 have Q by (rule impE)
  with 2 show R .
qed

lemma \textit{allE'}:
assumes 1: ALL x. P x
and 2: P x ---\rightarrow ALL x. P x ---\rightarrow Q
shows Q
proof
  from 1 have P x by (rule spec)
  from this and 1 show Q by (rule 2)
qed

lemma \textit{notE'}:
assumes 1: \sim P
and 2: \sim P ---\rightarrow P
shows R
proof
  from 2 and 1 have P .
  with 1 show R by (rule notE)
qed

lemma \textit{TrueE}: True ---\rightarrow P ---\rightarrow P .
lemma \textit{notFalseE}: \sim False ---\rightarrow P ---\rightarrow P .

lemmas [Pure.clim!] = disjE iffE FalseE conjE exE TrueE notFalseE
and [Pure.intro!] = iffI conjI impl TrueI notI allI refl
and [Pure.clim 2] = allE notE' impI'
and [Pure.intro] = exI disjI2 disjI1

lemmas [trans] = trans
and [sym] = sym not-sym
2.2.17 Atomizing meta-level connectives

axiomatization where

\textit{eq-reflection}: x = y \implies x \equiv y

\textbf{lemma} \ atomize-all [\textit{atomize}]: (!!x. P x) == \textit{Trueprop} (\textit{ALL} x. P x)
\textbf{proof}
\begin{itemize}
  \item assume !!x. P x
  \item then show \textit{ALL} x. P x ..
\end{itemize}
\textbf{qed}

\textbf{lemma} \ atomize-imp [\textit{atomize}]: (A ==\> B) == \textit{Trueprop} (A \land\lor B)
\textbf{proof}
\begin{itemize}
  \item assume r: A ==\> B
  \item show A \land\lor B by (rule impI) (rule r)
\end{itemize}
\textbf{next}
\begin{itemize}
  \item assume A \land\lor B and A
  \item then show B by (rule mp)
\end{itemize}
\textbf{qed}

\textbf{lemma} \ atomize-not: (A ==\> False) == \textit{Trueprop} (\textit{\~}A)
\textbf{proof}
\begin{itemize}
  \item assume r: A ==\> False
  \item show \textit{\~}A by (rule notI) (rule r)
\end{itemize}
\textbf{next}
\begin{itemize}
  \item assume \textit{\~}A and A
  \item then show False by (rule notE)
\end{itemize}
\textbf{qed}

\textbf{lemma} \ atomize-eq [\textit{atomize, code}]: (x == y) == \textit{Trueprop} (x = y)
\textbf{proof}
\begin{itemize}
  \item assume x == y
  \item show x = y by (unfold \langle x == y \rangle) (rule refl)
\end{itemize}
\textbf{next}
\begin{itemize}
  \item assume x = y
  \item then show x == y by (rule eq-reflection)
\end{itemize}
\textbf{qed}

\textbf{lemma} \ atomize-conj [\textit{atomize}]: (A \&\& B) == \textit{Trueprop} (A \& B)
\textbf{proof}
\begin{itemize}
  \item assume conj: A \&\& B
  \item show A \& B
  \begin{itemize}
    \item \textbf{proof} (rule conjI)
    \begin{itemize}
      \item \textbf{from} conj \textbf{show} A by (rule conjunctionD1)
    \end{itemize}
  \end{itemize}
\end{itemize}
\textbf{qed}
from conj show B by (rule conjunctionD2)
qed
next
assume conj: A & B
show A &&& B
proof
  from conj show A ..
  from conj show B ..
qed
qed

lemmas [symmetric, rulify] = atomize-all atomize-imp
and [symmetric, defn] = atomize-all atomize-imp atomize-eq

2.2.18 Atomizing elimination rules
setup AtomizeElim.setup

lemma atomize-exL[atomize-elim]: (!!x. P x ==> Q) ==> ((EX x. P x) ==> Q)
  by rule iprover+

lemma atomize-conjL[atomize-elim]: (A ==> B ==> C) ==> (A & B ==> C)
  by rule iprover+

lemma atomize-disjL[atomize-elim]: ((A ==> C) ==> (B ==> C) ==> C)
  ==> ((A | B ==> C) ==> C)
  by rule iprover+

lemma atomize-elimL[atomize-elim]: (!!B. (A ==> B) ==> B) ==> Trueprop A
  ..

2.3 Package setup
ML-file Tools/hologic.ML

2.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

ML ⟨⟨
structure No-ATPs = Named-Thms
{
  val name = @{binding no-atp}
  val description = theorems that should be filtered out by Sledgehammer
}
⟩⟩

setup ⟨⟨ No-ATPs.setup ⟩⟩
2.3.2 Classical Reasoner setup

lemma imp-elim: \( P \rightarrow Q \rightarrow (\neg R \rightarrow P) \rightarrow (Q \rightarrow R) \rightarrow R \)
by (rule classical) iprover

lemma swap: \( \neg P \rightarrow (\neg R \rightarrow P) \rightarrow R \)
by (rule classical) iprover

lemma thin-refl:
\[ \forall x. [ x=x; PROP W ] \Rightarrow PROP W . \]

ML ⟨⟨
structure Hypsubst = Hypsubst
{
  val dest-eq = HOLogic.dest-eq
  val dest-Trueprop = HOLogic.dest-Trueprop
  val dest-imp = HOLogic.dest-imp
  val eq-reflection = \{thm eq-reflection\}
  val rev-eq-reflection = \{thm meta-eq-to-obj-eq\}
  val imp-intr = \{thm impI\}
  val rev-mp = \{thm rev-mp\}
  val subst = \{thm subst\}
  val sym = \{thm sym\}
  val thin-refl = \{thm thin-refl\};
};
open Hypsubst;

structure Classical = Classical
{
  val imp-elim = \{thm imp-elim\}
  val not-elim = \{thm notE\}
  val swap = \{thm swap\}
  val classical = \{thm classical\}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
};

structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;
⟩⟩

setup Classical.setup

setup ⟨⟨
let
  fun non-bool-eq (\{const-name HOL.eq\}, Type (-, [T, -])) = T <> \{typ bool\}
| non-bool-eq - = false;
  fun hyp-subst-tac' ctxt =
    SUBGOAL (fn (goal, i) =>
      if Term.exists-Const non-bool-eq goal

then Hypsubst.hyp-subst-tac ctxt i
    else no-tac);
in
Hypsubst.hypsubst-setup
(*prevent substitution on bool*)
#> Context-Rules.addSWrapper (fn ctxt => fn tac => hyp-subst-tac' ctxt ORELSE' tac)
end
⟩⟩

declare iffI [intro!]
and notI [intro!]
and impI [intro!]
and disjCI [intro!]
and conjI [intro!]
and TrueI [intro!]
and refl [intro!]

declare iffCE [elim!]
and FalseE [elim!]
and impCE [elim!]
and disjE [elim!]
and conjE [elim!]

declare ex-ex1I [intro!]
and allI [intro]
and the-equality [intro]
and exI [intro]

declare exE [elim!]
allE [elim]

ML ⟨⟨ val HOL-cs = claset-of @\{context\} ⟩⟩

lemma contrapos-np: ~ Q == ( ~ P == Q ) == P
apply (erule swap)
apply (erule (1) meta-mp)
done

declare ex-ex1I [rule del, intro! 2]
and exI [intro]

declare ext [intro]

lemmas [intro?] = ext
and [elim?] = ex1-implies-ex

lemma alt-ex1E [elim!]:
assumes major: \( \exists x. \, P \)  
and prem: \( \forall x. \, [ \, P \; x \land \forall y \; y' \; . \; P \; y \land P \; y' \rightarrow y = y' \, ] \rightarrow R \)  
shows \( R \)  
apply (rule ex1E [OF major])  
apply (rule prem)  
apply (tactic \( \langle{\text{ares-tac @\{thms allI\} \, 1}}\rangle \))  
apply (tactic \( \langle{\text{etac (Classical.dup-elim @\{thm allE\} \, 1}}\rangle \))  
apply iprover  
done

ML  
\[
\text{structure Blast} = \text{Blast}
\]
\[
\text{structure Classical} = \text{Classical}
\]
val Trueprop-const = dest-Const @\{const Trueprop\}  
val equality-name = @\{const-name HOL.eq\}  
val not-name = @\{const-name Not\}  
val notE = @\{thm notE\}  
val ccontr = @\{thm ccontr\}  
val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
\]
val blast-tac = Blast.blast-tac;
\]
setup Blast.setup

2.3.3 Simplifier

lemma eta-contract-eq: \( (\%s. \, f \; s) = f \, . \)  

lemma simp-thms:
shows not-not: \( (\sim \sim P) = P \)  
and Not-iff: \( (\sim P) = (\sim Q) = (P = Q) \)  
and \( (\sim P) = (P = (\sim Q)) \rightarrow (P \sim Q) \)  
\( (P \sim Q) \) = True  
\( (\sim P \sim P) \) = True  
\( (x = x) \) = True  
and not-True-iff-False [code]: \( (\sim \text{True}) = \text{False} \)  
and not-False-iff-True [code]: \( (\sim \text{False}) = \text{True} \)  
and \( (\sim P) \sim (P \sim P) \)  
\( (\text{True} = P) = P \)  
and \( (\sim P) \) = True  
\( (P = P) = \text{True} \)  
and \( (\sim P) \) = True  
\( (\text{False} = P) = (\sim P) \)  
and \( (\sim P) \) = False  
\( (P = \text{False}) = (\sim P) \)  
and \( (\text{True} \rightarrow P) = (\text{False} \rightarrow P) = \text{True} \)  
\( (P \rightarrow \text{True}) = \text{True} \)  
\( (P \rightarrow P) = \text{True} \)  
\( (P \rightarrow \text{False}) = (\sim P) \)  
\( (P \rightarrow (\sim P)) = (\sim P) \)
THEORY “HOL”

\[(P \& \text{True}) = P \, (\text{True} \& P) = P\]
\[(P \& \text{False}) = \text{False} \, (\text{False} \& P) = \text{False}\]
\[(P \& P) = P \, (P \& (P \& Q)) = (P \& Q)\]
\[(P \& \lnot P) = \text{False} \, (\lnot P \& P) = \text{False}\]
\[(P | \text{True}) = \text{True} \, (\text{True} | P) = \text{True}\]
\[(P | \text{False}) = P \, (\text{False} | P) = P\]
\[(P | P) = P \, (P | (P | Q)) = (P | Q)\]
and
\[(\forall x. P) = P \, (\exists x. P) = P \, \exists x. x=t \, \exists x. t=x\]

\textbf{lemma disj-absorb:} \((A \mid A) = A\) 
\textbf{by blast}

\textbf{lemma disj-left-absorb:} \((A \mid (A \mid B)) = (A \mid B)\) 
\textbf{by blast}

\textbf{lemma conj-absorb:} \((A \& A) = A\) 
\textbf{by blast}

\textbf{lemma conj-left-absorb:} \((A \& (A \& B)) = (A \& B)\) 
\textbf{by blast}

\textbf{lemma eq-ac:}
\textbf{shows} eq-commute: \(a = b \iff b = a\)
\textbf{and} iff-left-commute: \((P \iff (Q \iff R)) \iff (Q \iff (P \iff R))\)
\textbf{and} iff-assoc: \((P \iff Q) \iff (P \iff (Q \iff R))\) \textbf{by (iprover, blast+)}

\textbf{lemma neq-commute:} \(a \neq b \iff b \neq a\) \textbf{by iprover}

\textbf{lemma conj-commoms:}
\textbf{shows} conj-commute: \((P\&Q) = (Q\&P)\)
\textbf{and} conj-left-commute: \((P\&(Q\&R)) = (Q\&(P\&R))\) \textbf{by iprover+}

\textbf{lemma conj-assoc:} \(((P\&Q)\&R) = (P\&(Q\&R))\) \textbf{by iprover}

\textbf{lemmas} conj-ac = conj-commute conj-left-commute conj-assoc

\textbf{lemma disj-commoms:}
\textbf{shows} disj-commute: \((P|Q) = (Q|P)\)
\textbf{and} disj-left-commute: \((P|(Q|R)) = (Q|(P|R))\) \textbf{by iprover+}

\textbf{lemma disj-assoc:} \(((P|Q)|R) = (P|(Q|R))\) \textbf{by iprover}

\textbf{lemmas} disj-ac = disj-commute disj-left-commute disj-assoc

\textbf{lemma conj-disj-distribL:} \((P\&(Q|R)) = (P\&Q \mid P\&R)\) \textbf{by iprover}
lemma conj-disj-distribR: \((P|Q)&R) = (P&R | Q&R)\) by iprover

lemma disj-conj-distribL: \((P|(Q&R)) = ((P|Q) & (P|R))\) by iprover
lemma disj-conj-distribR: \((P&Q)|R) = ((P|R) & (Q|R))\) by iprover

lemma imp-conjR: \((P \rightarrow (Q&R)) = ((P\rightarrow Q) & (P\rightarrow R))\) by iprover
lemma imp-conjL: \((P&Q) \rightarrow R) = (P \rightarrow (Q \rightarrow R))\) by iprover
lemma imp-disjL: \((P|Q) \rightarrow R) = ((P\rightarrow R)&(Q\rightarrow R))\) by iprover

These two are specialized, but imp-disj-not1 is useful in Auth/Yahalom.

lemma imp-disj-not1: \((P \rightarrow Q | R) = (\sim Q \rightarrow P \rightarrow R)\) by blast
lemma imp-disj-not2: \((P \rightarrow Q | R) = (\sim R \rightarrow P \rightarrow Q)\) by blast

lemma imp-cong: \((P = P') \implies (P' \implies (Q = Q')) \implies ((P \implies Q) = (P' \implies Q'))\) by iprover

lemma de-Morgan-disj: \((\sim (P | Q)) = (\sim P & \sim Q)\) by iprover
lemma de-Morgan-conj: \((\sim (P & Q)) = (\sim P | \sim Q)\) by blast
lemma not-imp: \((\sim (P \rightarrow Q)) = (P & \sim Q)\) by blast
lemma not-iff: \((P\sim Q) = (P = (\sim Q))\) by blast
lemma disj-not1: \((\sim P | Q) = (P \rightarrow Q)\) by blast
lemma disj-not2: \((P | \sim Q) = (Q \rightarrow P)\) — changes orientation :-)
   by blast
lemma imp-conv-disj: \((P \rightarrow Q) = ((\sim P) | Q)\) by blast

lemma iff-conc-conv-imp: \((P = Q) = ((P \rightarrow Q) & (Q \rightarrow P))\) by iprover

lemma cases-simp: \(((P \rightarrow Q) & (\sim P \rightarrow Q)) = Q\)
— Avoids duplication of subgoals after split-if, when the true and false
— cases boil down to the same thing.
   by blast

lemma not-all: \((\sim (! x. P(x))) = (? x.\sim P(x))\) by blast
lemma imp-all: \(((! x. P x) \rightarrow Q) = (? x. \sim P x \rightarrow Q)\) by blast
lemma not-ex: \((\sim (? x. P(x))) = (! x. \sim P(x))\) by iprover
lemma imp-ex: \(((? x. P x) \rightarrow Q) = (! x. P x \rightarrow Q)\) by iprover
lemma all-not-ex: \((\forall x. P x) = (\sim (EX x. \sim P x))\) by blast

declare All-def [no-atp]

lemma ex-disj-distrib: \((? x. P(x) | Q(x)) = (((? x. P(x)) | (? x. Q(x))))\) by iprover
lemma all-conj-distrib: \((! x. P(x) & Q(x)) = ((! x. P(x)) & (! x. Q(x)))\) by iprover

The \& congruence rule: not included by default! May slow rewrite proofs
THEORY “HOL”

down by as much as 50%

**lemma** conj-cong:

\[(P = P') \implies (Q = Q') \implies ((P \land Q) = (P' \land Q'))\]

*by iprover*

**lemma** rev-conj-cong:

\[(Q = Q') \implies (Q' = Q) \implies ((P' \land Q) = (P \land Q'))\]

*by iprover*

The \(\mid\) congruence rule: not included by default!

**lemma** disj-cong:

\[(P = P') \implies \neg P' \implies (Q = Q') \implies ((P \lor Q) = (P' \lor Q'))\]

*by blast*

if-then-else rules

**lemma** if-True [code]: \((\text{if True then } x \text{ else } y) = x\)

*by (unfold If-def) blast*

**lemma** if-False [code]: \((\text{if False then } x \text{ else } y) = y\)

*by (unfold If-def) blast*

**lemma** if-P: \(P \implies (\text{if } P \text{ then } x \text{ else } y) = x\)

*by (unfold If-def) blast*

**lemma** if-not-P: \(\neg P \implies (\text{if } P \text{ then } x \text{ else } y) = y\)

*by (unfold If-def) blast*

**lemma** split-if: \((\text{if } Q \text{ then } x \text{ else } y) = ((Q \implies P(x)) \land (\neg Q \implies P(y)))\)

*apply (rule case-split [of Q])*

*apply (simplesubst if-P)*

*prefer 3 apply (simplesubst if-not-P, blast+)

done*

**lemma** split-if-asn: \((\text{if } Q \text{ then } x \text{ else } y) = (\neg(\neg(\neg(\neg Q \land \neg P) \mid Q) \land P x) \mid \neg(\neg Q \land \neg P) y))\)

*by (simplesubst split-if, blast)*

**lemmas** if-splits [no-atp] = split-if split-if-asn

**lemma** if-cancel: \((\text{if } c \text{ then } x \text{ else } y) = x\)

*by (simplesubst split-if, blast)*

**lemma** if-eq-cancel: \((\text{if } x = y \text{ then } y \text{ else } x) = x\)

*by (simplesubst split-if, blast)*

**lemma** if-bool-eq-conj: \((\text{if } P \text{ then } Q \text{ else } R) = ((\neg P \implies Q) \land (\neg P \implies R))\)

— This form is useful for expanding if-s on the RIGHT of the \(\implies\) symbol.

*by (rule split-if)*
lemma if-bool-eq-disj: \( (\text{if } P \text{ then } Q \text{ else } R) = ((P \& Q) \mid (\neg P \& R)) \)
— And this form is useful for expanding if's on the LEFT.
apply (simplesubst split-if, blast)
done

lemma Eq-TrueI: \( P = \Rightarrow P = \text{True} \) by (unfold atomize-eq) iprover
lemma Eq-FalseI: \( \neg P = \Rightarrow P = \text{False} \) by (unfold atomize-eq) iprover

let rules for simproc

lemma Let-folded: \( f \equiv g \Rightarrow \text{Let } x f \equiv \text{Let } x g \)
by (unfold Let-def)

lemma Let-unfold: \( f \equiv g \Rightarrow \text{Let } x f \equiv g \)
by (unfold Let-def)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.
definition simp-implies :: \([\text{prop}, \text{prop}] = \Rightarrow\) where
  simp-implies \equiv op ==>

lemma simp-impliesI:
  assumes PQ: \( \text{PROP } P \Rightarrow \text{PROP } Q \)
  shows \( \text{PROP } P \Rightarrow \text{PROP } Q \)
  apply (unfold simp-implies-def)
  apply (rule PQ)
  apply assumption
done

lemma simp-impliesE:
  assumes PQ: \( \text{PROP } P \Rightarrow \text{PROP } Q \)
  and P: \( \text{PROP } P \)
  and QR: \( \text{PROP } Q \Rightarrow \text{PROP } R \)
  shows \( \text{PROP } R \)
  apply (rule QR)
  apply (rule PQ [unfolded simp-implies-def])
  apply (rule P)
done

lemma simp-implies-cong:
  assumes PP': \( \text{PROP } P \Rightarrow \text{PROP } P' \)
  and P'QQ': \( \text{PROP } P' \Rightarrow (\text{PROP } Q \Rightarrow \text{PROP } Q') \)
  shows \( (\text{PROP } P \Rightarrow \text{PROP } Q) \Rightarrow (\text{PROP } P' \Rightarrow \text{PROP } Q') \)
proof (unfold simp-implies-def, rule equal-intr-rule)
  assume PQ: \( \text{PROP } P \Rightarrow \text{PROP } Q \)
  and P': \( \text{PROP } P' \)
  from PP' [symmetric] and P' have PROP P
  by (rule equal-elim-rule1)
then have $PROP Q$ by (rule $PQ$)
with $P'QQ'$ [of $P'$] show $PROP Q'$ by (rule equal-elim-rule1)
next
  assume $P'Q'$; $PROP P' \implies PROP Q'$
  and $P$: $PROP P$
  from $PP'$ and $P$ have $P'$: $PROP P'$ by (rule equal-elim-rule1)
  then have $PROP Q'$ by (rule $P'Q'$)
  with $P'QQ'$ [of $P'$, symmetric] show $PROP Q$
    by (rule equal-elim-rule1)
qed

lemma uncurry:
  assumes $P \implies Q \implies R$
  shows $P \land Q \implies R$
  using assms by blast

lemma iff-allI:
  assumes $\forall x. P x = Q x$
  shows $(\forall x. P x) = (\forall x. Q x)$
  using assms by blast

lemma iff-exI:
  assumes $\forall x. P x = Q x$
  shows $(\exists x. P x) = (\exists x. Q x)$
  using assms by blast

lemma all-comm:
  $(\forall x y. P x y) = (\forall y x. P x y)$
  by blast

lemma ex-comm:
  $(\exists x y. P x y) = (\exists y x. P x y)$
  by blast

ML-file Tools/simpdata.ML
ML "open Simpdata"
setup "map-theory-simpset (put-simpset HOL-basic-ss)"

simproc-setup defined-Ex (EX $x. P x) = "{ fn - => Quantifier1.rearrange-ex }"
simproc-setup defined-All (ALL $x. P x) = "{ fn - => Quantifier1.rearrange-all }"

setup "{ Simplifier.method-setup Splitter.split-modifiers
# > Splitter.setup
# > clasimp-setup
# > EqSubst.setup }"
Simproc for proving \((y = x) == False\) from premise \(\neg(x = y)\):

**simproc-setup** neq \((x = y) = \langle fn - =>

```ml
let
val neq-to-EQ-False = @\{thm not-sym\} RS @\{thm Eq-FalseI\};
fun is-neq eq lhs rhs thm =
  (case Thm.prop-of thm of
   _ $ (Not $ (eq $ t' $ r')) =>>
     Not = HOLogic.Not andalso eq' = eq andalso
     r' aconv lhs andalso t' aconv rhs
   | - => false);
fun proc ss ct =
  (case Thm.term-of ct of
   eq $ lhs $ rhs =>>
     (case find-first (is-neq eq lhs rhs) (Simplifier.prems-of ss) of
       SOME thm =>> SOME (thm RS neq-to-EQ-False)
       | NONE =>> NONE)
   | - =>> NONE);
in proc end;
```

```ml
)
```
then SOME $\circ\{\text{thm Let-def}\}$
else
    let
    val n = case f of (Abs (x, _, _)) => x | _ => x;
    val cx = cterm-of thy x;
    val {T = xT, ...} = rep-cterm cx;
    val cf = cterm-of thy f;
    val fx-g = Simplifier.rewrite ctxt (Thm.apply cf cx);
    val (g $ g) = prop-of fx-g;
    val g' = abstract-over (x, g);
    val abs-g' = Abs (n, xT, g');
in (if (g aconv g')
    then
        let
            val rl = cterm-instantiate [(f-Let-unfold, cf), (x-Let-unfold, cx)] $\circ\{\text{thm Let-unfold}\};
        in SOME (rl $ [fx-g]) end
    else
        if (Envir.beta-eta-contract f) aconv (Envir.beta-eta-contract abs-g')
        then NONE (*avoid identity conversion*)
        else let
            val g'x = abs-g'$x;
            val g-g'x = Thm.symmetric (Thm.beta-conversion false (cterm-of thy g'x));
        in
            val rl = cterm-instantiate
            [(f-Let-folded, cterm-of thy f), (x-Let-folded, cx),
             (g-Let-folded, cterm-of thy abs-g')]
            $\circ\{\text{thm Let-folded}\};
            in SOME (rl $ [Thm.transitive fx-g g-g'x])
        end
    end
| _ => NONE
end
end
⟩⟩

lemma True-implies-equals: (True $ PROP P) $ PROP P
proof
    assume True $ PROP P
    from this [OF TrueI] show PROP P.
next
    assume PROP P
    then show PROP P.
qed

lemma ex-simps:
$\forall P. Q. (\exists x. P x \& Q) = ((\exists x. P x) \& Q)$
$\forall P. Q. (\exists x. P \& Q x) = (P \& (\exists x. Q x))$
$\forall P. Q. (\exists x. P x \mid Q) = ((\exists x. P x) \mid Q)$
$\forall P. (\exists x. P \mid Q x) = (P \mid (\exists x. Q x))$
THEORY “HOL”

!!P Q. (EX x. P x ---> Q) = ((ALL x. P x) ---> Q)
!!P Q. (EX x. P ---> Q x) = (P ---> (EX x. Q x))
— Miniscoping: pushing in existential quantifiers.
by (iprover | blast)+

lemma all-simps:
!!P Q. (ALL x. P x & Q) = ((ALL x. P x) & Q)
!!P Q. (ALL x. P & Q x) = (P & (ALL x. Q x))
!!P Q. (ALL x. P x | Q) = ((ALL x. P x) | Q)
!!P Q. (ALL x. P | Q x) = (P | (ALL x. Q x))
!!P Q. (ALL x. P x ---> Q) = ((EX x. P x) ---> Q)
!!P Q. (ALL x. P ---> Q x) = (P ---> (ALL x. Q x))
— Miniscoping: pushing in universal quantifiers.
by (iprover | blast)+

lemmas [simp] =
  triv-forall-equality
  True-implies-equals
  if-True
  if-False
  if-cancel
  if-eq-cancel
  imp-disjL
  conj-assoc
  disj-assoc
  de-Morgan-conj
  de-Morgan-disj
  imp-disj1
  imp-disj2
  not-imp
  disj-not1
  not-all
  not-ex
  cases-simp
  the-eq-trivial
  the-sym-eq-trivial
  ex-simps
  all-simps
  simp-thms

lemmas [cong] = imp-cong simp-implies-cong
lemmas [split] = split-if

ML ⟨⟨ val HOL-ss = simpset-of @{context} ⟩⟩
Simplifies x assuming c and y assuming  c

lemma if-cong:
  assumes b = c
and \( c \implies x = u \)
and \( \neg c \implies y = v \)
shows \((\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)\)
using assms by simp

Prevents simplification of \( x \) and \( y \): faster and allows the execution of functional programs.

lemma if-weak-cong [cong]:
assumes \( b = c \)
shows \((\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)\)
using assms by (rule arg-cong)

Prevents simplification of \( t \): much faster

lemma let-weak-cong:
assumes \( a = b \)
shows \((\text{let } x = a \text{ in } t \ x) = (\text{let } x = b \text{ in } t \ x)\)
using assms by (rule arg-cong)

To tidy up the result of a simproc. Only the RHS will be simplified.

lemma eq-cong2:
assumes \( u = u' \)
shows \((t \equiv u) \equiv (t \equiv u')\)
using assms by simp

lemma if-distrib:
\( f \ (\text{if } c \text{ then } x \text{ else } y) = (\text{if } c \text{ then } f \ x \text{ else } f \ y)\)
by simp

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma fun-eq-iff: \( f = g \longleftrightarrow (\forall x. f \ x = g \ x)\)
by auto

2.3.4 Generic cases and induction

Rule projections:

ML ⟨⟨
structure Project-Rule = Project-Rule
{
  val conjunct1 = \{thm conjunct1\}
  val conjunct2 = \{thm conjunct2\}
  val mp = \{thm mp\}
}
⟩⟩

definition induct-forall where
induct-forall \( P \equiv \forall x. P \ x \)
definition induct-implies where
induct-implies A B == A \rightarrow B

definition induct-equal where
induct-equal x y == x = y

definition induct-conj where
induct-conj A B == A \land B

definition induct-true where
induct-true == True

definition induct-false where
induct-false == False

lemma induct-forall-eq: (\!x. P x) == Trueprop (induct-forall (\!x. P x))
by (unfold atomize-all induct-forall-def)

lemma induct-implies-eq: (A ==> B) == Trueprop (induct-implies A B)
by (unfold atomize-imp induct-implies-def)

lemma induct-equal-eq: (x == y) == Trueprop (induct-equal x y)
by (unfold atomize-eq induct-equal-def)

lemma induct-conj-eq: (A &&& B) == Trueprop (induct-conj A B)
by (unfold atomize-conj induct-conj-def)

lemmas induct-atomize' = induct-forall-eq induct-implies-eq induct-conj-eq
lemmas induct-atomize = induct-atomize' induct-equal-eq
lemmas induct-rulify'[symmetric] = induct-atomize'
lemmas induct-rulify [symmetric] = induct-atomize
lemmas induct-rulify-fallback =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def

lemma induct-forall-conj: induct-forall (\!x. induct-conj (A x) (B x)) =
  induct-conj (induct-forall A) (induct-forall B)
by (unfold induct-forall-def induct-conj-def) iprover

lemma induct-implies-conj: induct-implies C (induct-conj A B) =
  induct-conj (induct-implies C A) (induct-implies C B)
by (unfold induct-implies-def induct-conj-def) iprover

lemma induct-conj-curry: (induct-conj A B ==> PROP C) == (A ==> B ==> PROP C)
proof
  assume r: induct-conj A B ==> PROP C and A B
show PROP C by (rule r) (simp add: induct-conj-def ⟨A⟩ ⟨B⟩)
next
  assume r: A ==⇒ B ==⇒ PROP C and induct-conj A B
  show PROP C by (rule r) (simp-all add: ⟕induct-conj A B\[
\])
qed

lemmas induct-conj = induct-forall-conj induct-implies-conj induct-conj-curry

lemma induct-trueI: induct-true
  by (simp add: induct-true-def)

Method setup.

ML ⟨⟨
structure Induct = Induct
{
  val cases-default = @{thm case-split}
  val atomize = @{thms induct-atomize}
  val rulify = @{thms induct-rulify}
  val rulify-fallback = @{thms induct-rulify-fallback}
  val equal-def = @{thm induct-equal-def}
  fun dest-def (Const (@{const-name induct-equal}, -) $ t $ u) = SOME (t, u)
    | dest-def = NONE
  val trivial-tac = match-tac @{thms induct-trueI}
}⟩⟩

ML-file ~/src/Tools/induction.ML

setup ⟨⟨
Induct.setup #> Induction.setup #>
Context.theory-map (Induct.map-simpset (fn ss => ss addsimprocs
  [Simplifier.simproc-global @{theory} swap-induct-false
    [induct-false ==⇒ PROP P ==⇒ PROP Q]
    (fn - =>
      (fn - $ (P as -$ @{const induct-false}) $ (- $ Q $ -) =>
        if P <-> Q then SOME Drule.swap-prems-eq else NONE
      | - => NONE)),
  Simpifier.simproc-global @{theory} induct-equal-conj-curry
  [induct-conj P Q ==⇒ PROP R]
  (fn - =>
    (fn - $ (- $ P) $ - =>
      let
        fun is-conj ( @{const induct-conj} $ P $ Q) =
          is-conj P andalso is-conj Q
        | is-conj (Const (@{const-name induct-equal}, -) $ - $ -) = true
        | is-conj @{const induct-true} = true
        | is-conj @{const induct-false} = true
        | is-conj _ = false
      in
        
      )
    )
  )
⟩⟩
Pre-simplification of induction and cases rules

lemma [induct-simp]: (!!x. induct-equal x t ==> PROP P x) == PROP P t
  unfolding induct-equal-def
  proof
    assume R: !!x. x = t ==> PROP P x
    show PROP P t by (rule R [OF refl])
  next
    fix x assume PROP P t x = t
    then show PROP P x by simp
  qed

lemma [induct-simp]: (!!x. induct-equal t x ==> PROP P x) == PROP P t
  unfolding induct-equal-def
  proof
    assume R: !!x. t = x ==> PROP P x
    show PROP P t by (rule R [OF refl])
  next
    fix x assume PROP P t t = x
    then show PROP P x by simp
  qed

lemma [induct-simp]: (induct-false ==> P) == Trueprop induct-true
  unfolding induct-false-def induct-true-def
  by (iprover intro: equal-intr-rule)

lemma [induct-simp]: (induct-true ==> PROP P) == PROP P
  unfolding induct-true-def
  proof
    assume R: True ==> PROP P
    from TrueI show PROP P by (rule R)
  next
    assume PROP P .
    then show PROP P .
  qed

lemma [induct-simp]: (PROP P ==> induct-true) == Trueprop induct-true
  unfolding induct-true-def
  by (iprover intro: equal-intr-rule)

lemma [induct-simp]: (!!x. induct-true) == Trueprop induct-true
  unfolding induct-true-def
  by (iprover intro: equal-intr-rule)
lemma [induct-simp]: \( \text{induct-implies \ induct-true \ P = \ P} \)
by (simp add: induct-implies-def induct-true-def)

lemma [induct-simp]: \((x = x) = \text{True}\)
by (rule simp-thms)

hide-const induct-forall induct-implies induct-equal induct-conj induct-true induct-false

ML-file ~/src/Tools/induct-tacs.ML
setup Induct-Tacs.setup

2.3.5 Coherent logic

ML-file ~/src/Tools/coherent.ML
ML ⟨⟨
structure Coherent = Coherent |
  (val atomize-elimL = @{thm atomize-elimL};
val atomize-exL = @{thm atomize-exL};
val atomize-conjL = @{thm atomize-conjL};
val atomize-disjL = @{thm atomize-disjL};
val operator-names = [@{const-name HOL.disj}, @{const-name HOL.conj}, @{const-name Ex}];)
⟩⟩

2.3.6 Reorienting equalities

ML ⟨⟨
signature REORIENT-PROC =
  sig
val add : (term -> bool) -> theory -> theory
val proc : morphism -> Proof.context -> cterm -> thm option
end;
structure Reorient-Proc : REORIENT-PROC =
struct
structure Data = Theory-Data |
  (type T = ((term -> bool) * stamp) list;
val empty = [];
val extend = I;
fun merge data : T = Library.merge (eq-snd op =) data;
);
fun add m = Data.map (cons (m, stamp ()));
fun matches thy t = exists (fn (m, _) => m t) (Data.get thy);

val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
fun proc phi ctxt ct =
let

val thy = Proof-Context.theory-of ctxt;

in

case Thm.term-of ct of

| - $ t \# u ) => if matches thy u then NONE else SOME meta-reorient
| - => NONE
end;
end;

⟩⟩

2.4 Other simple lemmas and lemma duplicates

lemma ex1-eq [iff]: EX! x. x = t EX! x. t = x
by blast+

lemma choice-eq: (ALL x. EX! y. P x y) = (EX! f. ALL x. P x (f x))
apply (rule iffI)
apply (rule-tac a = %x. THE y. P x y in exII)
apply (fast dest!: theI')
apply (fast intro: the1-equality [symmetric])
apply (erule ex1E)
apply (rule allI)
apply (rule ex1II)
apply (erule spec)
apply (erule-tac x = %z. if z = x then y else f z in allE)
apply (erule simpE)
apply (rule allI)
apply (erule case-tac x = x)
apply (erule-tac x = x)
done

lemmas eq-sym-conv = eq-commute

lemma nnf-simps:

(¬(P \land Q)) = (¬ P \lor ¬ Q) (¬ (P \lor Q)) = (¬ P \land ¬ Q) (P \rightarrow Q) = (¬P \lor Q)
(P = Q) = ((P \land Q) \lor (¬P \land ¬ Q)) (¬(P = Q)) = ((P \land ¬ Q) \lor (¬P \land Q))
(¬ ¬(P)) = P
by blast+

2.5 Basic ML bindings

ML ⟨⟨
val FalseE = @\{thm FalseE\}
val Let-def = @\{thm Let-def\}
val TrueI = @\{thm TrueI\}
val allE = @\{thm allE\}
val allI = @\{thm allI\}
val all-dupE = @\{thm all-dupE\}
val arg-cong = @\{thm arg-cong\}⟩⟩
val box-equals = @ {thm box-equals}
val ccontr = @ {thm ccontr}
val classical = @ {thm classical}
val conjE = @ {thm conjE}
val conjI = @ {thm conjI}
val conjunct1 = @ {thm conjunct1}
val conjunct2 = @ {thm conjunct2}
val disjCI = @ {thm disjCI}
val disjE = @ {thm disjE}
val disjI1 = @ {thm disjI1}
val disjI2 = @ {thm disjI2}
val eq-reflection = @ {thm eq-reflection}
val ex1E = @ {thm ex1E}
val ex1I = @ {thm ex1I}
val ex1-implies-ex = @ {thm ex1-implies-ex}
val exE = @ {thm exE}
val exI = @ {thm exI}
val excluded-middle = @ {thm excluded-middle}
val ext = @ {thm ext}
val fun-cong = @ {thm fun-cong}
val iffD1 = @ {thm iffD1}
val iffD2 = @ {thm iffD2}
val iffI = @ {thm iffI}
val impE = @ {thm impE}
val impI = @ {thm impI}
val meta-eq-to-obj-eq = @ {thm meta-eq-to-obj-eq}
val mp = @ {thm mp}
val notE = @ {thm notE}
val notI = @ {thm notI}
val not-all = @ {thm not-all}
val not-ex = @ {thm not-ex}
val not-iff = @ {thm not-iff}
val not-not = @ {thm not-not}
val not-sym = @ {thm not-sym}
val refl = @ {thm refl}
val rev-mp = @ {thm rev-mp}
val spec = @ {thm spec}
val sssubst = @ {thm sssubst}
val subst = @ {thm subst}
val sym = @ {thm sym}
val trans = @ {thm trans}
⟩⟩
\[ P \leftrightarrow Q \Rightarrow P \land R \leftrightarrow Q \land R \]
\textbf{by (fact arg-cong)}

\textbf{lemma disj-left-cong:}
\[ P \leftrightarrow Q \Rightarrow P \lor R \leftrightarrow Q \lor R \]
\textbf{by (fact arg-cong)}

\textbf{setup} \langle
code{Code-Preproc.map-pre (put-simpset HOL-basic-ss)}
\#> Code-Preproc.map-post (put-simpset HOL-basic-ss)
\#> Code-Simp.map-ss (put-simpset HOL-basic-ss)
\#> Simplifier.add-cong @{thm conj-left-cong} \#> Simplifier.add-cong @{thm disj-left-cong}
\rangle

\textbf{2.6.2 Equality}
\textbf{class} equal =
\textbf{fixes} equal :: 'a ⇒ 'a ⇒ bool
\textbf{assumes} equal-eq: equal x y \iff x = y
\begin{flushleft}
\textbf{lemma} equal: equal = (op =)
\textbf{by (rule ext equal-eq)+}
\end{flushleft}

\textbf{lemma} equal-refl: equal x x \iff True
\textbf{unfolding} equal \textbf{by rule+}

\textbf{lemma} eq-equal: (op =) ≡ equal
\textbf{by (rule eq-reflection)} (rule ext, rule ext, rule sym, rule equal-eq)
\end{flushleft}

\textbf{end}

\textbf{declare} eq-equal \[\text{[symmetric, code-post]}\]
\textbf{declare} eq-equal \[\text{[code]}\]

\textbf{setup} \langle
code{Code-Preproc.map-pre (fn ctxt =>
cxt.addsimprocs [Simplifier.simproc-global-i @{theory} equal [@{term HOL.eq}]
  (fn _ => fn Const (_, Type (fun, [Type _, _])) => SOME @{thm eq-equal} | _ => NONE))}
\rangle

\textbf{2.6.3 Generic code generator foundation}

\textbf{Datatype} bool
\textbf{code-datatype} True False

\textbf{lemma} [code]:
shows $\text{False} \land P \iff \text{False}$
and $\text{True} \land P \iff P$
and $P \land \text{False} \iff \text{False}$
and $P \land \text{True} \iff P$ by simp-all

lemma [code]:
shows $\text{False} \lor P \iff P$
and $\text{True} \lor P \iff \text{True}$
and $P \lor \text{False} \iff P$
and $P \lor \text{True} \iff \text{True}$ by simp-all

lemma [code]:
shows $(\text{False} \rightarrow P) \iff \text{True}$
and $(\text{True} \rightarrow P) \iff P$
and $(P \rightarrow \text{False}) \iff \neg P$
and $(P \rightarrow \text{True}) \iff \text{True}$ by simp-all

More about prop

lemma [code nbe]:
shows $(\text{True} \Rightarrow PROP Q) \equiv PROP Q$
and $(PROP Q \Rightarrow True) \equiv \text{Trueprop True}$
and $(P \Rightarrow R) \equiv \text{Trueprop} (P \rightarrow R)$ by (auto intro!: equal-intr-rule)

lemma Trueprop-code [code]:
$\text{Trueprop True} \equiv \text{Code-Generator.holds}$
by (auto intro!: equal-intr-rule holds)

declare Trueprop-code [symmetric, code-post]

Equality

declare simp-thms(6) [code nbe]

instantiation itself :: (type) equal
begin

definition equal-itself :: "'a itself ⇒ 'a itself ⇒ bool" where
  equal-itself $x \ y \iff x = y$

instance proof
qed (fact equal-itself-def)
end

lemma equal-itself-code [code]:
$\text{equal TYPE('a) TYPE('a) \iff True}$
by (simp add: equal)

setup "Sign.add-const-constraint @{const-name equal}, SOME @{typ 'a::type ⇒ 'a ⇒ \}
lemma equal-alias-cert: OFCLASS('a, equal-class) ≡ ((op = :: 'a ⇒ 'a ⇒ bool) ≡ equal) (is ?ofclass ≡ ?equal)

proof
  assume PROP ?ofclass
  show PROP ?equal
  by (tactic (ALLGOALS (rtac (Thm.unconstrT @{thm eq-equal}))))

next
  assume PROP ?equal
  show PROP ?ofclass proof
  qed (simp add: PROP ?equal)
qed

setup ⟨⟨
  Sign.add-const-constraint (@{const-name equal}, SOME @{typ 'a::equal ⇒ 'a ⇒ bool})
⟩⟩

setup ⟨⟨
  Nbe.add-const-alias @{thm equal-alias-cert}
⟩⟩

Cases

lemma Let-case-cert:
  assumes CASE ≡ (λx. Let x f)
  shows CASE x ≡ f x
  using assms by simp-all

setup ⟨⟨
  Code.add-case @{thm Let-case-cert}
  #> Code.add-undefined @{const-name undefined}
⟩⟩

declare [[code abort: undefined]]

2.6.4 Generic code generator target languages

type bool
code-printing
type-constructor bool →
  (SML) bool and (OCaml) bool and (Haskell) Bool and (Scala) Boolean
| constant True →
  (SML) true and (OCaml) true and (Haskell) True and (Scala) true
| constant False →
  (SML) false and (OCaml) false and (Haskell) False and (Scala) false
code-reserved SML
  bool true false

code-reserved OCaml
  bool

code-reserved Scala
  Boolean

code-printing
  constant Not ⇀
    (SML) not and (OCaml) not and (Haskell) not and (Scala) ⇑ -
    | constant HOL.conj →
      (SML) infixl 1 andalso and (OCaml) infixl 3 && and (Haskell) infixr 3 &&
      and (Scala) infixl 3 &&
    | constant HOL.disj →
      (SML) infixl 0 orelse and (OCaml) infixl 2 || and (Haskell) infixl 2 || and
      (Scala) infixl 1 ||
    | constant HOL.implies →
      (SML) !(if (-)/ then (-)/ else true)
      and (OCaml) !(if (-)/ then (-)/ else true)
      and (Haskell) !(if (-)/ then (-)/ else True)
      and (Scala) !(if ((-))/ (-)/ else true)
    | constant If →
      (SML) !(if (-)/ then (-)/ else (-))
      and (OCaml) !(if (-)/ then (-)/ else (-))
      and (Haskell) !(if (-)/ then (-)/ else (-))
      and (Scala) !(if ((-))/ (-)/ else (-))

code-reserved SML
  not

code-reserved OCaml
  not

code-identifier
  code-module Pure →
    (SML) HOL and (OCaml) HOL and (Haskell) HOL and (Scala) HOL

using built-in Haskell equality

code-printing
  type-class equal → (Haskell) Eq
  | constant HOL.equal → (Haskell) infix 4 ==
  | constant HOL.eq → (Haskell) infix 4 ==

undefined

code-printing
  constant undefined →
    (SML) !(raise/ Fail/ undefined)
and (OCaml) failwith/undefined
and (Haskell) error/undefined
and (Scala) !sys.error(undefined)

### 2.6.5 Evaluation and normalization by evaluation

**method-setup** eval = \!

et
fun eval-tac ctxt =
  let val conv = Code-Runtime.dynamic-holds-conv ctxt
  in CONVERSION (Conv.params-conv ~1 (K (Conv.concl-conv ~1 conv)) ctxt)
  THEN’ rtac TrueI end
in
  Scan.succeed (SIMPLE-METHOD’ o eval-tac)
end
\!

**method-setup** normalization = \!
Scan.succeed (fn ctxt =>
  SIMPLE-METHOD’
  (CHANGED-PROP o
    (CONVERSION (Nbe.dynamic-conv ctxt)
      THEN-ALL-NEW (TRY o rtac TrueI))))
\!

### 2.7 Counterexample Search Units

#### 2.7.1 Quickcheck

**quickcheck-params** [size = 5, iterations = 50]

#### 2.7.2 Nitpick setup

**ML**

```ml
structure Nitpick-Unfolds = Named-Thms
{
  val name = @{binding nitpick-unfold}
  val description = alternative definitions of constants as needed by Nitpick
}
structure Nitpick-Simps = Named-Thms
{
  val name = @{binding nitpick-simp}
  val description = equational specification of constants as needed by Nitpick
}
structure Nitpick-Psimps = Named-Thms
{
  val name = @{binding nitpick-psimp}
  val description = partial equational specification of constants as needed by Nitpick
}
structure Nitpick-Choice-Specs = Named-Thms
```
THEORY “HOL”

(val name = @{binding nitpick-choice-spec}
val description = choice specification of constants as needed by Nitpick)

setup ⟨⟨Nitpick-Unfolds.setup
#> Nitpick-Simps.setup
#> Nitpick-Psimps.setup
#> Nitpick-Choice-Specs.setup⟩⟩

declare if-boe-eq-conj [nitpick-unfold, no-atp]
     if-boe-eq-disj [no-atp]

2.8 Preprocessing for the predicate compiler

ML ⟨⟨structure Predicate-Compile-Alternative-Defs = Named-Thms
(     val name = @{binding code-pred-def}
     val description = alternative definitions of constants for the Predicate Compiler)
structure Predicate-Compile-Inline-Defs = Named-Thms
(     val name = @{binding code-pred-inline}
     val description = inlining definitions for the Predicate Compiler)
structure Predicate-Compile-Simps = Named-Thms
(     val name = @{binding code-pred-simp}
     val description = simplification rules for the optimisations in the Predicate Compiler)
⟩⟩

setup ⟨⟨Predicate-Compile-Alternative-Defs.setup
#> Predicate-Compile-Inline-Defs.setup
#> Predicate-Compile-Simps.setup⟩⟩

2.9 Legacy tactics and ML bindings

ML ⟨⟨(* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)
local
fun wrong-prem (Const (@{const-name All}, -) $ Abs (_, _, t)) = wrong-prem t
    | wrong-prem (Bound '-') = true
    | wrong-prem _ = false;⟩⟩
val filter-right = filter (not o wrong-prem o HO Logic.dest-Trueprop o hd o Thm.prems-of);

in
fun smp i = funpow i (fn m => filter-right ([spec] RL m)) ([mp]);

fun smp-tac j = EVERY' [dresolve-tac (smp j), atac];
end;

local
val nnf-ss = simpset-of (put-simpset HOL-basic-ss @ {context} addsimps @ {thms simp-thms nnf-simps});

in
fun nnf-conv ctxt = Simplifier.rewrite (put-simpset nnf-ss ctxt);
end

hide-const (open) eq equal

end

3 Orderings: Abstract orderings

theory Orderings
imports HOL
keywords print-orders :: diag

ML-file ~/src/Provers/order.ML
ML-file ~/src/Provers/quasi.ML

3.1 Abstract ordering

locale ordering =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix 50)
  and less :: 'a ⇒ 'a ⇒ bool (infix 50)
  assumes strict-iff-order: a ≺ b ⇔ a ≤ b ∧ a ≠ b
  assumes refl: a ≤ a — not iff: makes problems due to multiple (dual) interpretations
  and antisym: a ≤ b ⇒ b ≤ a ⇒ a = b
  and trans: a ≤ b ⇒ b ≤ c ⇒ a ≤ c
begin

lemma strict-implies-order:
  a ≺ b ⇒ a ≤ b
  by (simp add: strict-iff-order)

lemma strict-implies-not-eq:
  a ≺ b ⇒ a ≠ b
  by (simp add: strict-iff-order)
lemma not-eq-order-implies-strict:
\[ a \neq b \implies a \leq b \implies a \prec b \]
by (simp add: strict-iff-order)

lemma order-iff-strict:
\[ a \leq b \iff a \prec b \lor a = b \]
by (auto simp add: strict-iff-order refl)

lemma irrefl: — not iff: makes problems due to multiple (dual) interpretations
\[ \neg a \prec a \]
by (simp add: strict-iff-order)

lemma asym:
\[ a \prec b \implies b \prec a \implies False \]
by (auto intro: antisym)

lemma strict-trans1:
\[ a \leq b \implies b \prec c \implies a \prec c \]
by (auto simp add: strict-iff-order intro: trans antisym)

lemma strict-trans2:
\[ a \prec b \implies b \leq c \implies a \prec c \]
by (auto simp add: strict-iff-order intro: trans antisym)

lemma strict-trans:
\[ a \prec b \implies b \prec c \implies a \prec c \]
by (auto intro: strict-trans1 strict-implies-order)

locale ordering-top = ordering +
fixes top :: 'a
assumes extremum [simp]: \[ a \leq top \]
begin

lemma extremum-unique1:
\[ top \leq a \implies a = top \]
by (rule antisym) auto

lemma extremum-unique:
\[ top \leq a \iff a = top \]
by (auto intro: antisym)

lemma extremum-strict [simp]:
\[ \neg (top \prec a) \]
using extremum [of a] by (auto simp add: order-iff-strict intro: asym irrefl)

lemma not-eq-extremum:
\[ a \neq top \iff a \prec top \]
by (auto simp add: order-iff-strict intro: not-eq-order-implies-strict extremum)

end

3.2 Syntactic orders

class ord =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool
    and less :: 'a ⇒ 'a ⇒ bool
begin

notation
  less-eq (op <) and
  less-eq ((/ <) [51, 51] 50) and
  less (op <) and
  less ((/ <) [51, 51] 50)

notation (xsymbols)
  less-eq (op ≤) and
  less-eq ((/ ≤) [51, 51] 50)

notation (HTML output)
  less-eq (op ≤) and
  less-eq ((/ ≤) [51, 51] 50)

abbreviation (input)
  greater-eq (infix ≥ 50) where
  x ≥ y ≡ y ≤ x

notation (input)
  greater-eq (infix ≥ 50)

abbreviation (input)
  greater (infix > 50) where
  x > y ≡ y < x

end

3.3 Quasi orders

class preorder = ord +
  assumes less-le-not-le: x < y ↔ x ≤ y ∧ ¬ (y ≤ x)
    and order-refl [iff]: x ≤ x
    and order-trans: x ≤ y =⇒ y ≤ z =⇒ x ≤ z
begin

Reflexivity.

lemma eq-refl: x = y =⇒ x ≤ y
  — This form is useful with the classical reasoner.
by (erule ssubst) (rule order-refl)

lemma less-irrefl [iff]: \( \neg x < x \)
by (simp add: less-le-not-le)

lemma less-imp-le: \( x < y \implies x \leq y \)
unfolding less-le-not-le by blast

Asymmetry.

lemma less-not-sym: \( x < y \implies \neg (y < x) \)
by (simp add: less-le-not-le)

lemma less-asym: \( x < y \implies (\neg P \implies y < x) \implies P \)
by (drule less-not-sym, erule contrapos-np) simp

Transitivity.

lemma less-trans: \( x < y \implies y < z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

lemma le-less-trans: \( x \leq y \implies y < z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

lemma less-le-trans: \( x < y \implies y \leq z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

Useful for simplification, but too risky to include by default.

lemma less-imp-not-less: \( x < y \implies (\neg y < x) \iff \text{True} \)
by (blast elim: less-asym)

lemma less-imp-triv: \( x < y \implies (y < x \implies P) \iff \text{True} \)
by (blast elim: less-asym)

Transitivity rules for calculational reasoning

lemma less-asym': \( a < b \implies b < a \implies P \)
by (rule less-asym)

Dual order

lemma dual-preorder:
  class preorder \((\text{op} \geq)\) \((\text{op} >)\)
proof qed (auto simp add: less-le-not-le intro: order-trans)

end

3.4 Partial orders

class order = preorder +
  assumes antisym: \( x \leq y \implies y \leq x \implies x = y \)
begin
lemma less-le: \( x < y \iff x \leq y \land x \neq y \)
by (auto simp add: less-le-not-le intro: antisym)

sublocale order!: ordering less-eq less + dual-order!: ordering greater-eq greater
by default (auto intro: antisym order-trans simp add: less-le)

Reflexivity.
lemma le-less: \( x \leq y \iff x < y \lor x = y \)
— NOT suitable for iff, since it can cause PROOF FAILED.
by (fact order.order-iff-strict)

lemma unfolding less-le by blast

Useful for simplification, but too risky to include by default.
lemma less-imp-not-eq: \( x < y \Longrightarrow (x = y) \iff False \)
by auto

lemma less-imp-not-eq2: \( x < y \Longrightarrow (y = x) \iff False \)
by auto

Transitivity rules for calculational reasoning
lemma neq-le-trans: \( a \neq b \Longrightarrow a \leq b \Longrightarrow a < b \)
by (fact order.not-eq-order-implies-strict)

lemma le-neq-trans: \( a \leq b \Longrightarrow a \neq b \Longrightarrow a < b \)
by (rule order.not-eq-order-implies-strict)

Asymmetry.
lemma eq-iff: \( x = y \iff x \leq y \land y \leq x \)
by (blast intro: antisym)

lemma antisym-conv: \( y \leq x \Longrightarrow x \leq y \iff x = y \)
by (blast intro: antisym)

lemma less-imp-neq: \( x < y \Longrightarrow x \neq y \)
by (fact order.strict-implies-not-eq)

Least value operator
definition (in ord)
Least :: \( 'a \Rightarrow bool \Rightarrow 'a \) where
Least \( P = (\text{THE } x. \ P x \land (\forall y. \ P y \Longrightarrow x \leq y)) \)

lemma Least-equality:
assumes \( P x \)
and \( \forall y. \ P y \Longrightarrow x \leq y \)
shows \( \text{Least } P = x \)
THEORY “Orderings”

unfolding Least-def by (rule the-equality)
(blast intro: assms antisym)+

lemma LeastI2-order:
assumes P x
and \( \forall y. P y \implies x \leq y \)
and \( \forall x. P x \implies \forall y. P y \implies x \leq y \implies Q x \)
shows Q (Least P)
unfolding Least-def by (rule theI2)
(blast intro: assms antisym)+

Dual order

lemma dual-order:
class ord (op ≥) (op >)
by (intro-locales, rule dual-preorder) (unfold-locales, rule antisym)
end

Alternative introduction rule with bias towards strict order

lemma order-strictI:
fixes less (infix ⊏ 50)
and less-eq (infix ⊑ 50)
assumes less-eq-less: \( \forall a. a ⊑ b \iff a ⊏ b \lor a = b \)
assumes asym: \( \forall a b. a ⊏ b \implies ¬ b ⊏ a \)
assumes irrefl: \( \forall a. ¬ a ⊏ a \)
assumes trans: \( \forall a b c. a ⊏ b \land b ⊏ c \implies a ⊏ c \)
shows class.ord less-eq less
proof
fix a b
show a ⊏ b \iff a ⊑ b ∧ ¬ b ⊑ a
  by (auto simp add: less-eq-less asym irrefl)
next
fix a
show a ⊏ a
  by (auto simp add: less-eq-less)
next
fix a b c
assume a ⊑ b and b ⊑ c then show a ⊑ c
  by (auto simp add: less-eq-less asym intro: trans)
next
fix a b
assume a ⊑ b and b ⊑ a then show a = b
  by (auto simp add: less-eq-less asym)
qed

3.5 Linear (total) orders

class linorder = order +
assumes linear: \( x ≤ y \lor y ≤ x \)
begin

lemma less-linear: \( x < y \lor x = y \lor y < x \)
unfolding less-le using less-le linear by blast

lemma le-less-linear: \( x \leq y \lor y < x \)
by (simp add: le-less less-linear)

lemma le-cases [case-names le ge]:
\[
(x \leq y \implies P) \implies (y \leq x \implies P) \implies P
\]
using linear by blast

lemma linorder-cases [case-names less equal greater]:
\[
(x < y \implies P) = \implies (x = y \implies (y < x \implies P) \implies P)
\]
using less-linear by blast

lemma linorder-wlog [case-names le sym]:
\[
(\forall a b. a \leq b \implies P a b) \implies (\forall a b. P b a \implies P a b) \implies P a b
\]
by (cases rule: le-cases [of a b] ) blast+

lemma not-less: \( \neg x < y \iff y \leq x \)
apply (simp add: less-le)
using linear apply (blast intro: antisym)
done

lemma not-less-iff-gr-or-eq:
\( \neg(x < y) \iff (x > y \mid x = y) \)
apply (simp add: not-less le-less)
apply blast
done

lemma not-le: \( \neg x \leq y \iff y < x \)
apply (simp add: less-le)
using linear apply (blast intro: antisym)
done

lemma neq-iff: \( x \neq y \iff x < y \lor y < x \)
by (cut-tac x = x and y = y in less-linear, auto)

lemma neqE: \( x \neq y \implies (x < y \implies R) \implies (y < x \implies R) \implies R \)
by (simp add: neq-iff) blast

lemma antisym-conv1: \( \neg x < y \implies x \leq y \iff x = y \)
by (blastic intro: antisym dest: not-less [THEN iffD1])

lemma antisym-conv2: \( x \leq y \implies \neg x < y \iff x = y \)
by (blastic intro: antisym dest: not-less [THEN iffD1])

lemma antisym-conv3: \( y < x \implies \neg x < y \iff x = y \)
by (blast intro: antisym dest: not-less \[THEN\] iffD1))

**lemma** leI: \( \neg x < y \implies y \leq x \)**

**unfolding** not-less .

**lemma** leD: \( y \leq x \implies \neg x < y \)**

**unfolding** not-less .

**lemma** not-leE: \( \neg y \leq x \implies x < y \)**

**unfolding** not-le .

Dual order

**lemma** dual-linorder:

\[
\text{class.linorder (op \(\geq\)) (op \(>\))}
\]

**by** (rule class.linorder.intro, rule dual-order) (unfold-locales, rule linear)

end

Alternative introduction rule with bias towards strict order

**lemma** linorder-strictI:

**fixes** less (infix \(\sqsubseteq\) 50)

**and** less-eq (infix \(\sqsubseteq\) 50)

**assumes** class.order less-eq less

**assumes** trichotomy: \( \forall a \ b. \ a \sqsubseteq b \lor a = b \lor b \sqsubseteq a \)

**shows** class.linorder less-eq less

**proof** -

**interpret** order less-eq less

**by** (fact \(\langle\text{class.order less-eq less}\rangle\))

**show** ?thesis

**proof**

**fix** a b

**show** a \(\subseteq\) b \(\lor\) b \(\subseteq\) a

**using** trichotomy **by** (auto simp add: le-less)

**qed**

**qed**

### 3.6 Reasoning tools setup

**ML** \(\langle\langle\text{signature ORDERS =}

\text{sig}

\text{val print-structures: Proof.context \to\ unit}

\text{val order-tac: Proof.context \to\ thm list \to\ int \to\ tactic}

\text{end};

\text{structure Orders: ORDERS =}

\text{struct}

\rangle\rangle\)
fun struct-eq ((s1: string, ts1), (s2, ts2)) = 
s1 = s2 andalso eq-list (op aconv) (ts1, ts2);
\( T = \text{HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT } \)

end;

fun rel (bin-op $ t1 $ t2) =
    if excluded t1 then NONE
    else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
    else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
    else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
    else NONE
| rel - = NONE;

fun dec (Const (@{ const-name Not }, -) $ t) =
    (case rel t of NONE =>
      NONE
    | SOME (t1, rel, t2) => SOME (t1, ¬ rel, t2))

in decomp - - = NONE;

in
    (case s of
      order => Order-Tac.partial-tac decomp thms ctxt facts
    | linorder => Order-Tac.linear-tac decomp thms ctxt facts
    | - => error (Unknown order kind "quote s " encountered in transitivity reasoner))
end

fun order-tac ctxt facts =
    FIRST' (map (fn s => CHANGED o struct-tac s ctxt facts) (Data.get (Context.Proof ctxt)));

(* attributes *)

fun add-struct-thm s tag =
    Thm.declaration-attribute
    (fn thm => Data.map (AList.map-default struct-eq (s, Order-Tac.empty TrueI) (Order-Tac.update tag thm)));

fun del-struct s =
    Thm.declaration-attribute
    (fn - => Data.map (AList.delete struct-eq s));

val - =
    Theory.setup
    (Attrib.setup @{ binding order }
    (Scan.lift ((Args.add -- Args.name >> (fn (_, s) => SOME s) || Args.del >> K NONE)) --
      Args.colon (* FIXME || Scan.succeed true *) -- Scan.lift Args.name --
      Scan.repeat Args.term
    >> (fn ((SOME tag, n), ts) => add-struct-thm (n, ts) tag
      | ((NONE, n), ts) => del-struct (n, ts))
    theorems controlling transitivity reasoner);
THEORY “Orderings”

end;
⟩⟩

method-setup order = ⟨⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (Orders.order-tac ctxt []))
⟩⟩ transitivity reasoner

Declarations to set up transitivity reasoner of partial and linear orders.

context order
begin

declare less-irrefl [THEN notE, order add less-reflE: order op = :: 'a ⇒ 'a ⇒
  bool op <= op <]

declare order-refl [order add le-refl: order op = :: 'a => 'a => bool op <= op <]

declare less-imp-le [order add less-imp-le: order op = :: 'a => 'a => bool op <=
  op <]

declare antisym [order add eqI: order op = :: 'a => 'a => bool op <= op <]

declare eq-refl [order add eqD1: order op = :: 'a => 'a => bool op <= op <]

declare sym [THEN eq-refl, order add eqD2: order op = :: 'a => 'a => bool op
  <= op <]

declare less-trans [order add less-trans: order op = :: 'a => 'a => bool op <=
  op <]

declare less-le-trans [order add less-le-trans: order op = :: 'a => 'a => bool op
  <= op <]

declare le-less-trans [order add le-less-trans: order op = :: 'a => 'a => bool op
  <= op <]

declare order-trans [order add le-trans: order op = :: 'a => 'a => bool op <=
  op <]

declare le-neq-trans [order add le-neq-trans: order op = :: 'a => 'a => bool op
  <= op <]

declare neq-le-trans [order add neq-le-trans: order op = :: 'a => 'a => bool op
  <= op <]

declare less-imp-neq [order add less-imp-neq: order op = :: 'a => 'a => bool op
  <]
THEORY “Orderings”

\[ \leq \ op \leq \]

**declare** eq-neq-imp-neq | order add eq-neq-imp-neq: order op = :: 'a => 'a => bool op \leq op <]

**declare** not-sym | order add not-sym: order op = :: 'a => 'a => bool op \leq op <]

end

class context linorder

**declare** [[order del]: order op = :: 'a => 'a => bool op \leq op <]

**declare** less-irrefl | THEN notE, order add less-reflE: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** order-refl | order add le-refl: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** less-imp-le | order add less-imp-le: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** not-less | THEN iffD2, order add not-lessI: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** not-le | THEN iffD2, order add not-leI: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** not-less | THEN iffD1, order add not-lessD: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** not-le | THEN iffD1, order add not-leD: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** antisym | order add eqI: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** eq-refl | order add eqD1: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** sym | THEN eq-refl, order add eqD2: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** less-trans | order add less-trans: linorder op = :: 'a => 'a => bool op \leq op <]

**declare** less-le-trans | order add less-le-trans: linorder op = :: 'a => 'a => bool op \leq op <]
THEORY “Orderings” 82

declare le-less-trans [order add le-less-trans: linorder op = :: 'a => 'a => bool op <= op <]
declare order-trans [order add le-trans: linorder op = :: 'a => 'a => bool op <= op <]
declare le-neq-trans [order add le-neq-trans: linorder op = :: 'a => 'a => bool op <= op <]
declare neq-le-trans [order add neq-le-trans: linorder op = :: 'a => 'a => bool op <= op <]
declare less-imp-neq [order add less-imp-neq: linorder op = :: 'a => 'a => bool op <= op <]
declare eq-neq-eq-imp-neq [order add eq-neq-eq-imp-neq: linorder op = :: 'a => 'a => bool op <= op <]
declare not-sym [order add not-sym: linorder op = :: 'a => 'a => bool op <= op <]
end

setup ⟨⟨
map-theory-simpset (fn ctxt0 => ctxt0 addSolver
  mk-solver Transitivity (fn ctxt => Orders.order-tac ctxt (Simplifier.prems-of ctxt)))
(*Adding the transitivity reasoners also as safe solvers showed a slight speed up, but the reasoning strength appears to be not higher (at least no breaking of additional proofs in the entire HOL distribution, as of 5 March 2004, was observed).*
⟩⟩
⟩⟩

ML ⟨⟨
lolocal
  fun prp t thm = Thm.prop-of thm = t; (*FIXME proper aconv!? *)
in

  fun antisym-le-simproc ctxt ct =
    (case term-of ct of
      (le as Const (_, T)) $ r $ s =>
      (let
        val prems = Simplifier.prems-of ctxt;
        val less = Const (@{const-name less}, T);
        val t = HOLogic.mkTrueprop(le $ s $ r);
        in
        (case find-first (prp t) prems of
          NONE =>
          let val t = HOLogic.mkTrueprop(HOLogic.Not ($ (less $ r $ s))) in

  end

⟩⟩}
THEORY “Orderings”

(case find-first (prp t) prems of
  NONE => NONE
  | SOME thm => SOME (mk-meta-eq (thm RS @ {thm linorder-class.antisym-conv1})))
end
| SOME thm => SOME (mk-meta-eq (thm RS @ {thm order-class.antisym-cone})))
end handle THM - => NONE)
| - => NONE);

fun antisym-less-simproc ctxt ct =
  (case term-of ct of
     NotC $ ((less as Const (\, T)) $ r $ s) =>
       (let
        val prems = Simplifier.prems-of ctxt;
        val le = Const (@ {const-name less-eq}, T);
        val t = HOLogic.mk-Trueprop (le $ r $ s);
       in
        (case find-first (prp t) prems of
          NONE =>
            let val t = HOLogic.mk-Trueprop (NotC $ (less $ s $ r)) in
            (case find-first (prp t) prems of
              NONE => NONE
              | SOME thm => SOME (mk-meta-eq (thm RS @ {thm linorder-class.antisym-conv3})))
            end
            | SOME thm => SOME (mk-meta-eq (thm RS @ {thm linorder-class.antisym-conv2})))
        end
        | - => NONE);
      |
end;

simproc-setup antisym-le ((x::'a::order) ≤ y) = K antisym-le-simproc
simproc-setup antisym-less (∀ (x::'a::linorder) < y) = K antisym-less-simproc

3.7 Bounded quantifiers

syntax
- All-less :: [idt, 'a, bool] => bool ((\ALL -<-/-) [0, 0, 10] 10)
- Ez-less :: [idt, 'a, bool] => bool ((\EX -<-/-) [0, 0, 10] 10)
- All-less-eq :: [idt, 'a, bool] => bool ((\ALL -<=/-) [0, 0, 10] 10)
- Ez-less-eq :: [idt, 'a, bool] => bool ((\EX -<=/-) [0, 0, 10] 10)
- All-greater :: [idt, 'a, bool] => bool ((\ALL ->/-) [0, 0, 10] 10)
- Ez-greater :: [idt, 'a, bool] => bool ((\EX ->/-) [0, 0, 10] 10)
- All-greater-eq :: [idt, 'a, bool] => bool ((\ALL ->=/-) [0, 0, 10] 10)
- Ez-greater-eq :: [idt, 'a, bool] => bool ((\EX ->=/-) [0, 0, 10] 10)

syntax (asymbols)
- All-less :: [idt, 'a, bool] => bool ((\@ -<-/-) [0, 0, 10] 10)
- Ez-less :: [idt, 'a, bool] => bool ((\@ -<-/-) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool ((\forall \leq -/.) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool ((\exists \leq -/.) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool ((\forall \rightarrow -/.) [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool ((\exists \rightarrow -/.) [0, 0, 10] 10)

-translations

syntax (HOL)
-All-less :: [idt, 'a, bool] => bool ((\forall \langle -/.) [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool ((\exists \langle -/.) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool ((\forall \leq = -/.) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool ((\exists \leq = -/.) [0, 0, 10] 10)

syntax (HTML output)
-All-less :: [idt, 'a, bool] => bool ((\forall \langle -/.) [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool ((\exists \langle -/.) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool ((\forall \leq = -/.) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool ((\exists \leq = -/.) [0, 0, 10] 10)

translations
\[
\begin{align*}
&\text{ALL } x < y. \ P \Rightarrow \ \text{ALL } x. \ x < y \rightarrow P \\
&\text{EX } x < y. \ P \Rightarrow \ \text{EX } x. \ x < y \land P \\
&\text{ALL } x \leq y. \ P \Rightarrow \ \text{ALL } x. \ x \leq y \rightarrow P \\
&\text{EX } x \leq y. \ P \Rightarrow \ \text{EX } x. \ x \leq y \land P \\
&\text{ALL } x > y. \ P \Rightarrow \ \text{ALL } x. \ x > y \rightarrow P \\
&\text{EX } x > y. \ P \Rightarrow \ \text{EX } x. \ x > y \land P \\
&\text{ALL } x \geq y. \ P \Rightarrow \ \text{ALL } x. \ x \geq y \rightarrow P \\
&\text{EX } x \geq y. \ P \Rightarrow \ \text{EX } x. \ x \geq y \land P 
\end{align*}
\]

print-translation \langle

let
val All-binder = Mixfix.binder-name @{const-syntax All};
val Ex-binder = Mixfix.binder-name @{const-syntax Ex};
val impl = @{const-syntax HOL.implies};
val conj = @{const-syntax HOL.conj};
val less = @{const-syntax less};
val less-eq = @{const-syntax less-eq};

val trans =
[[(All-binder, impl, less),
  (@{syntax-const -All-less}, @{syntax-const -All-greater})],
[(All-binder, impl, less-eq),
  (@{syntax-const -All-less-eq}, @{syntax-const -All-greater-eq})],
fun matches-bound v t =
  (case t of
   Const (@{syntax-const -bound}, -) $ Free (v', -) => v = v'
   | - => false);

fun contains-var v = Term.exists-subterm (fn Free (x, -) => x = v | - => false);

fun mk x c n P = Syntax.const c $ Syntax.Trans.mark-bound-body x $ n $ P;

fun tr' q = (q, fn - =>
  (fn [Const (@{syntax-const -bound}, -) $ Free (v, T),
       Const (c, -) $ (Const (d, -) $ t $ u) $ P] =>
       (case AList.lookup (op =) trans (q, c, d) of
        NONE => raise Match
        | SOME (l, g) =>
           if matches-bound v t andalso not (contains-var v u) then mk (v, T) l u P
           else if matches-bound v u andalso not (contains-var v t) then mk (v, T) g t P
           else raise Match)
   | - => raise Match));

in [tr' All-binder, tr' Ex-binder] end

3.8 Transitivity reasoning

context ord

begin

lemma ord-le-eq-trans: a ≤ b => b = c => a ≤ c
  by (rule subst)
lemma ord-eq-le-trans: a = b => b ≤ c => a ≤ c
  by (rule ssubst)
lemma ord-less-eq-trans: a < b => b = c => a < c
  by (rule subst)
lemma ord-eq-less-trans: a = b => b < c => a < c
  by (rule ssubst)

end

lemma order-less-subs1: a::'a::order < b => f b < (c::'c::order) =>
  (!x y. x < y => f x < f y) => f a < c
proof
  assume r: !x y. x < y => f x < f y
assume $a < b$ hence $f a < f b$ by (rule $r$)
also assume $f b < c$
finally (less-trans) show ?thesis.
qed

lemma order-less-subst1: $(a::'a::order) < f b ==> (b::'b::order) < c ==> ($!!x. x < y ==> f x < f y) ==> a < f c$
proof
  assume $r$: $!!x. x < y ==> f x < f y$
  assume $a < f b$
  also assume $b < c$ hence $f b < f c$ by (rule $r$)
  finally (less-trans) show ?thesis.
 qed

lemma order-le-less-subst2: $(a::'a::order) <= b ==> f b < (c::'c::order) ==> ($!!x. x <= y ==> f x <= f y) ==> f a < c$
proof
  assume $r$: $!!x. x <= y ==> f x <= f y$
  assume $a <= f b$
  also assume $f b <= c$
  finally (le-less-trans) show ?thesis.
 qed

lemma order-le-less-subst1: $(a::'a::order) <= f b ==> (b::'b::order) <= c ==> ($!!x. x < y ==> f x < f y) ==> f a < c$
proof
  assume $r$: $!!x. x < y ==> f x < f y$
  assume $a < f b$
  also assume $f b <= c$
  finally (le-less-trans) show ?thesis.
 qed

lemma order-less-le-subst2: $(a::'a::order) < f b ==> (b::'b::order) <= c ==> ($!!x. x < y ==> f x < f y) ==> a < f c$
proof
  assume $r$: $!!x. x < y ==> f x < f y$
  assume $a < f b$
  also assume $f b <= c$
  finally (less-le-trans) show ?thesis.
 qed

lemma order-less-le-subst1: $(a::'a::order) < f b ==> (b::'b::order) <= c ==> ($!!x. x <= y ==> f x <= f y) ==> a < f c$
proof
  assume $r$: $!!x. x <= y ==> f x <= f y$
  assume $a < f b$
  also assume $b <= c$ hence $f b <= f c$ by (rule $r$)
  finally (less-le-trans) show ?thesis.
 qed
lemma order-subst1: (a:a:order) <= f b ==> (b:b:order) <= c ==> 
((!!x y. x <= y ==> f x <= f y) ==> a <= f c)
proof -
  assume r: !!x y. x <= y ==> f x <= f y
  assume a <= f b
  also assume b <= c hence f b <= f c by (rule r)
finally (order-trans) show ?thesis .
qed

lemma order-subst2: (a:a:order) <= b ==> f b <= (c:c:order) ==> 
((!!x y. x <= y ==> f x <= f y) ==> f a <= c)
proof -
  assume r: !!x y. x <= y ==> f x <= f y
  assume a <= b hence f a <= f b by (rule r)
  also assume f b <= c
finally (order-trans) show ?thesis .
qed

lemma ord-le-eq-subst: a <= b ==> f b = c ==> 
((!!x y. x <= y ==> f x <= f y) ==> f a <= c)
proof -
  assume r: !!x y. x <= y ==> f x <= f y
  assume a <= b hence f a <= f b by (rule r)
  also assume f b = c
qed

lemma ord-eq-le-subst: a = f b ==> b <= c ==> 
((!!x y. x <= y ==> f x <= f y) ==> a <= f c)
proof -
  assume r: !!x y. x <= y ==> f x <= f y
  assume a = f b
  also assume b <= c hence f b <= c by (rule r)
qed

lemma ord-less-eq-subst: a < b ==> f b = c ==> 
((!!x y. x < y ==> f x < f y) ==> f a < c)
proof -
  assume r: !!x y. x < y ==> f x < f y
  assume a < b hence f a < f b by (rule r)
  also assume f b = c
finally (ord-less-eq-trans) show ?thesis .
qed

lemma ord-eq-less-subst: a = f b ==> b < c ==> 
((!!x y. x < y ==> f x < f y) ==> a < f c)
proof -

**THEORY “Orderings”**

**assume** \( r :: !x\ y\ .\ x < y \Rightarrow f\ x < f\ y \)**

**assume** \( a = f\ b \)**

*also assume* \( b < c\) *hence* \( f\ b < f\ c\) *by* (rule \( r \))


**qed**

Note that this list of rules is in reverse order of priorities.

**lemmas** [trans] =

- order-less-subst2
- order-less-subst1
- order-le-less-subst2
- order-le-less-subst1
- order-less-le-subst2
- order-less-le-subst1
- order-subst2
- order-subst1
- ord-le-eq-subst
- ord-eq-le-subst
- ord-less-eq-subst
- ord-eq-less-subst
- forw-subst
- back-subst
- rev-mp
- mp

**lemmas (in order) [trans] =**

- neq-le-trans
- le-neq-trans

**lemmas (in preorder) [trans] =**

- less-trans
- less-asym'
- le-less-trans
- less-le-trans
- order-trans

**lemmas (in order) [trans] =**

- antisym

**lemmas (in ord) [trans] =**

- ord-le-eq-trans
- ord-eq-le-trans
- ord-less-eq-trans
- ord-eq-less-trans

**lemmas [trans] =**

- trans

**lemmas order-trans-rules =**
order-less-subst2  
order-less-subst1  
order-le-less-subst2  
order-le-less-subst1  
order-less-le-subst2  
order-less-le-subst1  
order-subst2  
order-subst1  
ord-le-eq-subst  
ord-eq-le-subst  
ord-less-eq-subst  
ord-eq-less-subst  
forw-subst  
back-subst  
rev-mp  
mp  
neq-le-trans  
le-neq-trans  
less-trans  
less-asym  
le-less-trans  
less-le-trans  
order-trans  
antisym  
ord-le-eq-trans  
ord-eq-le-trans  
ord-less-eq-trans  
ord-eq-less-trans  
trans

These support proving chains of decreasing inequalities $a \leq b \leq c \ldots$ in Isar proofs.

**lemma xt1 [no-atp]:**

$a = b \Longrightarrow b > c \Longrightarrow a > c$
$a > b \Longrightarrow b = c \Longrightarrow a > c$
$a = b \Longrightarrow b > c \Longrightarrow a >= c$
$a >= b \Longrightarrow b = c \Longrightarrow a >= c$
$(x::a::order) >= y ==> y >= x ==> x = y$
$(x::a::order) >= y ==> y >= z ==> x >= z$
$(x::a::order) >= y ==> y >= z ==> x > z$
$(x::a::order) >= y ==> y > z ==> x > z$
$(a::a::order) > b ==> b > a ==> P$
$(x::a::order) > y ==> y > z ==> x > z$
$(a::a::order) >= b ==> a <= b ==> a > b$
(a::a::order) <= b ==> a >= b ==> a >= b$
$a = f b ==> b > c ==> (!x y. x > y ==> f x > f y) ==> a > f c$
$a > b ==> f b = c ==> (!x y. x > y ==> f x > f y) ==> f a > c$
$a = f b ==> b >= c ==> (!x y. x >= y ==> f x >= f y) ==> a >= f c$
$a >= b ==> f b = c ==> (! x y. x >= y ==> f x >= f y) ==> f a >= c$
by auto

lemma xt2 [no-atp]:
  (a::'a::order) >= f b ==> b >= c ==> (!x y. x >= y ==> f x >= f y) ==> a >= f c
by (subgoal-tac f b >= f c, force, force)

lemma xt3 [no-atp]: (a::'a::order) >= b ==> (f b::'b::order) >= c ==> (!x y. x >= y ==> f x >= f y) ==> a >= c
by (subgoal-tac f a >= f c, force, force)

lemma xt4 [no-atp]: (a::'a::order) > f b ==> (b::'b::order) >= c ==> (!x y. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f a > f c, force, force)

lemma xt5 [no-atp]: (a::'a::order) >= b ==> (f b::'b::order) > c ==> (!x y. x >= y ==> f x > f y) ==> a > c
by (subgoal-tac f a > f c, force, force)

lemma xt6 [no-atp]: (a::'a::order) >= f b ==> b > c ==> (!x y. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f a > f c, force, force)

lemma xt7 [no-atp]: (a::'a::order) >= b ==> (f b::'b::order) > c ==> (!x y. x >= y ==> f x > f y) ==> a > c
by (subgoal-tac f a > f b, force, force)

lemma xt8 [no-atp]: (a::'a::order) > f b ==> (b::'b::order) > c ==> (!x y. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

lemma xt9 [no-atp]: (a::'a::order) > b ==> (f b::'b::order) > c ==> (!x y. x > y ==> f x > f y) ==> f a > c
by (subgoal-tac f a > f b, force, force)

lemmas xtrans = xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9

3.9 Monotonicity
context order
begin

definition mono :: ('a ⇒ 'b::order) ⇒ bool where
  mono f ≡ (∀ x y. x ≤ y → f x ≤ f y)

lemma monoI [intro?]:
  fixes f :: 'a ⇒ 'b::order
  shows (∀ x y. x ≤ y → f x ≤ f y) → mono f
unfolding mono-def by iprover
lemma monoD [dest?]:
  fixes f :: 'a ⇒ 'b::order
  shows mono f ⇒ x ≤ y ⇒ f x ≤ f y
  unfolding mono-def by iprover

lemma monoE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes x ≤ y
  obtains f x ≤ f y
  proof
    from assms show f x ≤ f y by (simp add: mono-def)
  qed

definition antimono :: ('a ⇒ 'b::order) ⇒ bool where
  antimono f ←→ (∀x y. x ≤ y → f x ≥ f y)

lemma antimonoI [intro?]:
  fixes f :: 'a ⇒ 'b::order
  shows (∀x y. x ≤ y → f x ≥ f y) ⇒ antimono f
  unfolding antimono-def by iprover

lemma antimonoD [dest?]:
  fixes f :: 'a ⇒ 'b::order
  shows antimono f ⇒ x ≤ y ⇒ f x ≥ f y
  unfolding antimono-def by iprover

lemma antimonoE:
  fixes f :: 'a ⇒ 'b::order
  assumes antimono f
  assumes x ≤ y
  obtains f x ≥ f y
  proof
    from assms show f x ≥ f y by (simp add: antimono-def)
  qed

definition strict-mono :: ('a ⇒ 'b::order) ⇒ bool where
  strict-mono f ←→ (∀x y. x < y → f x < f y)

lemma strict-monoI [intro?]:
  assumes (∀x y. x < y → f x < f y)
  shows strict-mono f
  using assms unfolding strict-mono-def by auto

lemma strict-monoD [dest?]:
  strict-mono f ⇒ x < y ⇒ f x < f y
  unfolding strict-mono-def by auto
lemma strict-mono-mono [dest?):
  assumes strict-mono f
  shows mono f
proof (rule monoI)
  fix x y
  assume x ≤ y
  show f x ≤ f y
  proof (cases x = y)
    case True then show thesis by simp
  next
    case False with ⟨x ≤ y⟩ have x < y by simp
    with assms strict-monoD have f x < f y by auto
    then show thesis by simp
  qed
qed

context linorder
begin

lemma mono-invE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes f x < f y
  obtains x ≤ y
proof
  show x ≤ y
  proof (rule ccontr)
    assume ¬ x ≤ y
    then have y ≤ x by simp
    with ⟨mono f⟩ obtain f y ≤ f x by (rule monoE)
    with ⟨f x < f y⟩ show False by simp
  qed
qed

lemma strict-mono-eq:
  assumes strict-mono f
  shows f x = f y ⟷ x = y
proof
  assume f x = f y
  show x = y
  proof (cases x y rule: linorder-cases)
    case less with assms strict-monoD have f x < f y by auto
    with ⟨f x = f y⟩ show thesis by simp
  next
    case equal then show thesis .
  next
    case greater with assms strict-monoD have f y < f x by auto
    with ⟨f x = f y⟩ show thesis by simp
**THEORY “Orderings”**

qed
qed simp

**lemma** strict-mono-less-eq:
assumes strict-mono f
shows \( f x \leq f y \iff x \leq y \)
proof
assume \( x \leq y \)
with assms strict-mono monoD show \( f x \leq f y \) by auto
next
assume \( f x \leq f y \)
show \( x \leq y \)
proof (rule ccontr)
  assume \( \neg x \leq y \)
  then have \( y < x \) by simp
  with assms strict-monoD have \( f y < f x \) by auto
  with \( f x \leq f y \) show False by simp
qed
qed

**lemma** strict-mono-less:
assumes strict-mono f
shows \( f x < f y \iff x < y \)
using assms
  by (auto simp add: less-le Orderings.less-le strict-mono-eq strict-mono-less-eq)
end

### 3.10 min and max – fundamental

**definition** (in ord) \( \text{min ::= } \cdot \Rightarrow \cdot \Rightarrow \cdot \) where
\[
\text{min} \ a \ b = (\text{if } a \leq b \text{ then } a \text{ else } b)
\]

**definition** (in ord) \( \text{max ::= } \cdot \Rightarrow \cdot \Rightarrow \cdot \) where
\[
\text{max} \ a \ b = (\text{if } a \leq b \text{ then } b \text{ else } a)
\]

**lemma** min-absorb1: \( x \leq y \implies \text{min} \ x \ y = x \)
  by (simp add: min-def)

**lemma** max-absorb2: \( x \leq y \implies \text{max} \ x \ y = y \)
  by (simp add: max-def)

**lemma** min-absorb2: \( y :: \text{a::order} \leq x \implies \text{min} \ x \ y = y \)
  by (simp add:min-def)

**lemma** max-absorb1: \( y :: \text{a::order} \leq x \implies \text{max} \ x \ y = x \)
  by (simp add: max-def)

### 3.11 (Unique) top and bottom elements

**class** bot =
  **fixes** bot :: ‘a (\bot)
class order_bot = order + bot + 
    assumes bot_least: ⊥ ≤ a 
begin 

sublocale bot!: ordering-top greater-eq greater bot 
    by default (fact bot_least) 

lemma le_bot: 
    a ≤ ⊥ ⇒ a = ⊥ 
    by (fact bot.extremum-unIQUEI) 

lemma bot_unique: 
    a ≤ ⊥ ↔ a = ⊥ 
    by (fact bot.extremum_unique) 

lemma not_less_bot: 
    ¬ a < ⊥ 
    by (fact bot.extremum_strict) 

lemma bot_less: 
    a ≠ ⊥ ↔ ⊥ < a 
    by (fact bot.not_eq_extremum) 

end 

class top = 
    fixes top :: 'a (⊤) 

class order_top = order + top + 
    assumes top_greatest: a ≤ ⊤ 
begin 

sublocale top!: ordering-top less-eq less top 
    by default (fact top_greatest) 

lemma top_le: 
    ⊤ ≤ a ⇒ a = ⊤ 
    by (fact top.extremum_uniqueI) 

lemma top_unique: 
    ⊤ ≤ a ↔ a = ⊤ 
    by (fact top.extremum_unique) 

lemma not_top_less: 
    ¬ ⊤ < a 
    by (fact top.extremum_strict) 

lemma less_top: 

THEORY "Orderings"

\[ a \neq \top \iff a < \top \]
by (fact top.not-eq-extremum)

end

3.12 Dense orders

class dense-order = order +
\[ \text{assumes dense: } x < y \implies (\exists z. x < z \land z < y) \]
class dense-linorder = linorder + dense-order
begin

lemma dense-le:
fixes \( y, z :: 'a \)
assumes \( \forall x. x < y \implies x \leq z \)
shows \( y \leq z \)
proof (rule ccontr)
assume \( \neg \thesis \)
hence \( z < y \) by simp
from dense [OF this]
obtain \( x \) where \( x < y \) and \( z < x \) by safe
moreover have \( x \leq z \) using assms [OF \<\langle x < y \rangle\>]
ultimately show False by auto
qed

lemma dense-le-bounded:
fixes \( x, y, z :: 'a \)
assumes \( x < y \)
assumes \( \ast: \forall w. \langle x < w ; w < y \rangle \implies w \leq z \)
shows \( y \leq z \)
proof (rule dense-le)
fix \( w \) assume \( w < y \)
from dense [OF \<\langle x < y \rangle\>]
obtain \( u \) where \( x < u \) \( u < y \) by safe
from linear [of \( u \) \( w \)]
show \( w \leq z \)
proof (rule disjE)
assume \( u \leq w \)
from less-le-trans [OF \<\langle x < w \rangle \langle u \leq w \rangle \langle w < y \rangle\>]
show \( w \leq z \) by (rule \ast)
next
assume \( w \leq u \)
from \<\langle w \leq u \rangle \ast[OF \langle x < w \rangle \langle u < y \rangle \rangle\>
show \( w \leq z \) by (rule order-trans)
qed
qued

lemma dense-ge:
fixes \( y, z :: 'a \)
assumes $\forall x, z < x \Rightarrow y \leq x$
shows $y \leq z$
proof (rule ccontr)
  assume $\neg \thesis$
  hence $z < y$ by simp
  from dense[OF this]
  obtain $x$ where $x < y$ and $z < x$ by safe
moreover have $y \leq x$ using assms[OF $z < x$]
ultimately show $\false$ by auto
qed

lemma dense-ge-bounded:
  fixes $x, y, z$ :: 'a
  assumes $z < x$
  assumes $\ast$: $\forall w. \exists z < w ; w < x \Rightarrow y \leq w$
  shows $y \leq z$
proof (rule dense-ge)
  fix $w$
  assume $z < w$
  from dense[OF $z < x$] obtain $u$ where $z < u u < x$ by safe
  from linear[of $u w$]
  show $y \leq w$
    proof (rule disjE)
      assume $w \leq u$
      from $z < w$ le-less-trans[OF $w \leq u$]
      show $y \leq w$ by (rule $\ast$)
    next
      assume $u \leq w$
      from $\ast$[OF $z < w$ $(u < x)$] $(u \leq w)$
      show $y \leq w$ by (rule order-trans)
    qed
  qed
qed

class no-top = order +
  assumes gt-ex: $\exists y. x < y$

class no-bot = order +
  assumes lt-ex: $\exists y. y < x$

class unbounded-dense-linorder = dense-linorder + no-top + no-bot

3.13 Wellorders

class wellorder = linorder +
  assumes less-induct [case-names less]: $(\forall x. (\forall y < x \Rightarrow P y) \Rightarrow P x) \Rightarrow P a$
begin
THEORY “Orderings”

lemma wellorder-Least-lemma:
fixes k :: 'a
assumes P k
shows LeastI: P (LEAST x. P x) and Least-le: (LEAST x. P x) ≤ k
proof -
  have P (LEAST x. P x) ∧ (LEAST x. P x) ≤ k
  using assms proof (induct k rule: less-induct)
  case (less x) then have P x by simp
  show ?case proof (rule classical)
    assume assm: ¬ (P (LEAST a. P a) ∧ (LEAST a. P a) ≤ x)
    have ∨ y. P y ⇒ x ≤ y
    proof (rule classical)
      fix y
      assume P y and ¬ x ≤ y
      with less have P (LEAST a. P a) and (LEAST a. P a) ≤ y
      by (auto simp add: not-le)
      with assm have x < (LEAST a. P a) and (LEAST a. P a) ≤ y
      by auto
      then show x ≤ y by auto
    qed
    with (P x) have Least: (LEAST a. P a) = x
    by (rule Least-equality)
    with (P x) show ?thesis by simp
  qed
  qed
then show P (LEAST x. P x) and (LEAST x. P x) ≤ k by auto
qed

— The following 3 lemmas are due to Brian Huffman
lemma LeastI-ex: ∃ x. P x ⇒ P (Least P)
  by (erule exE) (erule LeastI)

lemma LeastI2:
P a ⇒ (∀ x. P x ⇒ Q x) ⇒ Q (Least P)
  by (blast intro: LeastI)

lemma LeastI2-ex:
∃ a. P a ⇒ (∀ x. P x ⇒ Q x) ⇒ Q (Least P)
  by (blast intro: LeastI-ex)

lemma LeastI2-wellorder:
assumes P a
  and ∀ a. [ P a; ∀ b. P b ⇒ a ≤ b ] ⇒ Q a
shows Q (Least P)
proof (rule LeastI2-order)
  show P (Least P) using P a by (rule LeastI)
next
  fix y assume P y thus Least P ≤ y by (rule Least-le)
next
THEORY “Orderings”

fix x assume P x ∀ y. P y → x ≤ y thus Q x by (rule assms(2))
qed

lemma not-less-Least: k < (LEAST x. P x) →¬ P k
apply (simp (no asm-use) add: not-le [symmetric])
apply (erule contrapos nn)
apply (erule Least-le)
done

end

3.14 Order on bool

instantiation bool :: {order-bot, order-top, linorder}
begnin

definition le-bool-def [simp]: P ≤ Q ←→ P → Q

definition [simp]: (P :: bool) < Q ←→ ¬ P ∧ Q

definition [simp]: ⊥ ←→ False

definition [simp]: ⊤ ←→ True

instance proof
qed auto
end

lemma le-boolI: (P ⇒ Q) ⇒ P ≤ Q
by simp

lemma le-boolI': P ⇒ Q ⇒ P ≤ Q
by simp

lemma le-boolE: P ≤ Q ⇒ P ⇒ (Q ⇒ R) ⇒ R
by simp

lemma le-boolD: P ≤ Q ⇒ P → Q
by simp

lemma bot-boolE: ⊥ ⇒ P
by simp

lemma top-boolI: ⊤
by simp

lemma [code]:
False ≤ b ←→ True
True ≤ b ←→ b
False < b ←→ b
True < b ←→ False
by simp-all

3.15 Order on - ⇒ -

instantiation fun :: (type, ord) ord begin

definition le-fun-def: f ≤ g ←→ (∀ x. f x ≤ g x)

definition (f::'a ⇒ 'b) < g ←→ f ≤ g ∧ ¬ (g ≤ f)

instance ..

end

instance fun :: (type, preorder) preorder proof
qed (auto simp add: le-fun-def less-fun-def intro: order-trans antisym)

instance fun :: (type, order) order proof
qed (auto simp add: le-fun-def intro: antisym)

instantiation fun :: (type, bot) bot begin

definition ⊥ = (λx. ⊥)

instance ..

end

instance fun :: (type, order-bot) order-bot begin

lemma bot-apply [simp, code]:
⊥ x = ⊥
by (simp add: bot-fun-def)

instance proof
THEORY “Orderings”

qed (simp add: le-fun-def)

end

instantiation fun :: (type, top) top
begin

definition
    [no-atp]: ⊤ = (λx. ⊤)

instance ..

end

instantiation fun :: (type, order-top) order-top
begin

lemma top-apply [simp, code]:
    ⊤ x = ⊤
    by (simp add: top-fun-def)

instance proof
    qed (simp add: le-fun-def)

end

lemma le-funI: (]\ x. f x ≤ g x) ⇒ f ≤ g
    unfolding le-fun-def by simp

lemma le-funE: f ≤ g ⇒ (f x ≤ g x ⇒ P) ⇒ P
    unfolding le-fun-def by simp

lemma le-funD: f ≤ g ⇒ f x ≤ g x
    by (rule le-funE)

3.16 Order on unary and binary predicates

lemma predicate1I:
    assumes PQ: ] x. P x ⇒ Q x
    shows P ≤ Q
    apply (rule le-funI)
    apply (rule le-boolI)
    apply (rule PQ)
    apply assumption
    done

lemma predicate1D:
    P ≤ Q ⇒ P x ⇒ Q x
    apply (erule le-funE)
THEORY “Orderings”

apply (erule le_boolE)
apply assumption+
done

lemma rev-predicate1D:
P x \Rightarrow P \leq Q \Rightarrow Q x
by (rule predicate1D)

lemma predicate2I:
assumes PQ: \forall x y. P x y \Rightarrow Q x y
shows P \leq Q
apply (rule le_funI)+
apply (rule le_boolI)
apply assumption
done

lemma predicate2D:
P \leq Q \Rightarrow P x y \Rightarrow Q x y
apply (erule le_funE)+
apply (erule le_boolE)
apply assumption+
done

lemma rev-predicate2D:
P x y \Rightarrow P \leq Q \Rightarrow Q x y
by (rule predicate2D)

lemma bot1E [no-atp]: \bot x \Rightarrow P
by (simp add: bot_fun_def)

lemma bot2E: \bot x y \Rightarrow P
by (simp add: bot_fun_def)

lemma top1I:
\top x
by (simp add: top_fun_def)

lemma top2I:
\top x y
by (simp add: top_fun_def)

3.17 Name duplicates
lemmas order_eq_refl = preorder_class.eq_refl
lemmas order_less_irrefl = preorder_class.less_irrefl
lemmas order_less_imp_le = preorder_class.less_imp_le
lemmas order_less_not_sym = preorder_class.less_not_sym
lemmas order_less_asym = preorder_class.less_asym
lemmas order_less_trans = preorder_class.less_trans
lemmas order_le_less_trans = preorder_class.le_less_trans
lemmas order-less-le-trans = preorder-class.less-le-trans
lemmas order-less-imp-not-less = preorder-class.less-imp-not-less
lemmas order-less-imp-triv = preorder-class.less-imp-triv
lemmas order-less-asym' = preorder-class.less-asym'

lemmas order-less-le = order-class.less-le
lemmas order-le-less = order-class.le-less
lemmas order-le-imp-less-or-eq = order-class.le-imp-less-or-eq
lemmas order-less-imp-not-eq = order-class.less-imp-not-eq
lemmas order-less-imp-not-eq2 = order-class.less-imp-not-eq2
lemmas order-neq-le-trans = order-class.neq-le-trans
lemmas order-le-neq-trans = order-class.le-neq-trans
lemmas order-antisym = order-class.antisym
lemmas order-eq-iff = order-class.eq-iff
lemmas order-antisym-conv = order-class.antisym-conv

lemmas linorder-linear = linorder-class.linear
lemmas linorder-les-linear = linorder-class.less-linear
lemmas linorder-le-less-linear = linorder-class.le-less-linear
lemmas linorder-le-cases = linorder-class.le-cases
lemmas linorder-not-less = linorder-class.not-less
lemmas linorder-not-le = linorder-class.not-le
lemmas linorder-neq-iff = linorder-class.neq-iff
lemmas linorder-neqE = linorder-class.neqE
lemmas linorder-antisym-conv1 = linorder-class.antisym-conv1
lemmas linorder-antisym-conv2 = linorder-class.antisym-conv2
lemmas linorder-antisym-conv3 = linorder-class.antisym-conv3

end

4 Groups: Groups, also combined with orderings

theory Groups
imports Orderings
begin

4.1 Fact collections

ML ⟨⟨
structure Ac-Simps = Named-Thms
{
  val name = @{binding ac-simps}
  val description = associativity and commutativity simplification rules
}
⟩⟩

setup Ac-Simps.setup

The rewrites accumulated in algebra-simps deal with the classical algebraic
structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by field-simps.

ML ⟨⟨
structure Algebra-Simps = Named-Thms
(  
  val name = @{binding algebra-simps}
  val description = algebra simplification rules
)
⟩⟩

setup Algebra-Simps.setup

Lemmas field-simps multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign algebra-simps.

ML ⟨⟨
structure Field-Simps = Named-Thms
(  
  val name = @{binding field-simps}
  val description = algebra simplification rules for fields
)
⟩⟩

setup Field-Simps.setup

4.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale semigroup =
  fixes f :: 'a ⇒ 'a ⇒ 'a (infixl * 70)
  assumes assoc [ac-simps]: a * b * c = a * (b * c)

locale abel-semigroup = semigroup +
  assumes commute [ac-simps]: a * b = b * a

begin

lemma left-commute [ac-simps]:
  b * (a * c) = a * (b * c)

proof −
  have (b * a) * c = (a * b) * c


by (simp only: commute)
then show ?thesis
  by (simp only: assoc)
qed

locale monoid = semigroup +
  fixes z :: 'a (1)
  assumes left-neutral [simp]: 1 ∗ a = a
  assumes right-neutral [simp]: a ∗ 1 = a
locale comm-monoid = abel-semigroup +
  fixes z :: 'a (1)
  assumes comm-neutral: a ∗ 1 = a
begin
sublocale monoid
  by default (simp-all add: commute comm-neutral)
end

4.3 Generic operations

class zero =
  fixes zero :: 'a (0)

class one =
  fixes one :: 'a (1)

hide-const (open) zero one

lemma Let-0 [simp]: Let 0 f = f 0
  unfolding Let-def ..

lemma Let-1 [simp]: Let 1 f = f 1
  unfolding Let-def ..

setup ⟨⟨
Reorient-Proc.add
  (fn Const(0){const-name Groups.zero}, _) => true
  | Const(1){const-name Groups.one}, _ => true
  | _ => false)
⟩}

simproc-setup reorient-zero (0 = x) = Reorient-Proc.proc
simproc-setup reorient-one (1 = x) = Reorient-Proc.proc

typed-print-translation ⟨⟨
THEORY "Groups"

let
fun tr' c = (c, fn ctxt => fn T => fn ts =>
  if null ts andalso Printer.type-emphasis ctxt T then
    Syntax.const @{"syntax-const -constrain"} $ Syntax.const c $
    Syntax-Phases.term-of-typ ctxt T
  else raise Match);
in map tr' [@{syntax-const Groups.one} | @{syntax-const Groups.zero}] end;

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a (infixl + 65)

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a (infixl - 65)

class uminus =
  fixes uminus :: 'a ⇒ 'a (− [81] 80)

class times =
  fixes times :: 'a ⇒ 'a ⇒ 'a (infixl * 70)

4.4 Semigroups and Monoids

class semigroup-add = plus +
  assumes add-assoc [algebra-simps, field-simps]: (a + b) + c = a + (b + c)
begin

sublocale add!: semigroup plus
  by default (fact add-assoc)
end

hide-fact add-assoc

class ab-semigroup-add = semigroup-add +
  assumes add-commute [algebra-simps, field-simps]: a + b = b + a
begin

sublocale add!: abel-semigroup plus
  by default (fact add-commute)

declare add.left-commute [algebra-simps, field-simps]

theorems add-ac = add.assoc add.commute add.left-commute
end

hide-fact add-commute
theorys add-ac = add.assoc add.commute add.left-commute

class semigroup-mult = times +
   assumes mult-assoc [algebra-simps, field-simps]: (a * b) * c = a * (b * c)
begin

sublocale mult!: semigroup times
   by default (fact mult-assoc)
end

hide-fact mult-assoc

class ab-semigroup-mult = semigroup-mult +
   assumes mult-commute [algebra-simps, field-simps]: a * b = b * a
begin

sublocale mult!: abel-semigroup times
   by default (fact mult-commute)

declare mult.left-commute [algebra-simps, field-simps]

theorems mult-ac = mult.assoc mult.commute mult.left-commute
end

hide-fact mult-commute

theorems mult-ac = mult.assoc mult.commute mult.left-commute

class monoid-add = zero + semigroup-add +
   assumes add-0-left: 0 + a = a
   and add-0-right: a + 0 = a
begin

sublocale add!: monoid plus 0
   by default (fact add-0-left add-0-right)+
end

lemma zero-reorient: 0 = x if-then x = 0
   by (fact eq-commute)

class comm-monoid-add = zero + ab-semigroup-add +
   assumes add-0: 0 + a = a
begin

sublocale add!: comm-monoid plus 0
   by default (insert add-0, simp add: ac-simps)
subclasses monoid-add
  by default (fact add.left-neutral add.right-neutral)+

end

class comm-monoid-diff = comm-monoid-add + minus +
  assumes diff-zero [simp]: a - 0 = a
  and zero-diff [simp]: 0 - a = 0
  and add-diff-cancel-left [simp]: (c + a) - (c + b) = a - b
  and diff-diff-add: a - b - c = a - (b + c)
begin

lemma add-diff-cancel-right [simp]:
  (a + c) - (b + c) = a - b
  using add-diff-cancel-left [symmetric] by (simp add: add.commute)

lemma add-diff-cancel-left' [simp]:
  (b + a) - b = a
  proof -
    have (b + a) - (b + 0) = a by (simp only: add-diff-cancel-left diff-zero)
    then show ?thesis by simp
  qed

lemma add-diff-cancel-right' [simp]:
  (a + b) - b = a
  using add-diff-cancel-left' [symmetric] by (simp add: add.commute)

lemma diff-add-zero [simp]:
  a - (a + b) = 0
  proof -
    have a - (a + b) = (a + 0) - (a + b) by simp
    also have ... = 0 by (simp only: add-diff-cancel-left zero-diff)
    finally show ?thesis .
  qed

lemma diff-cancel [simp]:
  a - a = 0
  proof -
    have (a + 0) - (a + 0) = 0 by (simp only: add-diff-cancel-left diff-zero)
    then show ?thesis by simp
  qed

lemma diff-right-commute:
  a - c - b = a - b - c
  by (simp add: diff-diff-add add.commute)

lemma add-implies-diff:
  assumes c + b = a
THEORY "Groups"

shows $c = a - b$

proof
  from assms have $(b + c) - (b + 0) = a - b$ by (simp add: add.commute)
  then show $c = a - b$ by simp
qed

end

class monoid-mult = one + semigroup-mult +
  assumes mult-1-left: $1 * a = a$
    and mult-1-right: $a * 1 = a$
begin

sublocale mult!: monoid times 1
  by default (fact mult-1-left mult-1-right)+
end

lemma one-reorient: $1 = x \iff x = 1$
  by (fact eq-commute)

class comm-monoid-mult = one + ab-semigroup-mult +
  assumes mult-1: $1 * a = a$
begin

sublocale mult!: comm-monoid times 1
  by default (insert mult-1, simp add: ac-simps)

subclass monoid-mult
  by default (fact mult.left-neutral mult.right-neutral)+
end

class cancel-semigroup-add = semigroup-add +
  assumes add-left-imp-eq: $a + b = a + c \Longrightarrow b = c$
  assumes add-right-imp-eq: $b + a = c + a \Longrightarrow b = c$
begin

lemma add-left-cancel [simp]:
  $a + b = a + c \iff b = c$
  by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]:
  $b + a = c + a \iff b = c$
  by (blast dest: add-right-imp-eq)
end

class cancel-ab-semigroup-add = ab-semigroup-add +
assumes add-imp-eq: \( a + b = a + c \Rightarrow b = c \)

begin

subclass cancel-semigroup-add
proof
  fix \( a \ b \ c :: 'a \)
  assume \( a + b = a + c \)
  then show \( b = c \) by (rule add-imp-eq)
next
  fix \( a \ b \ c :: 'a \)
  assume \( b + a = c + a \)
  then have \( a + b = a + c \) by (simp only: add.commute)
  then show \( b = c \) by (rule add-imp-eq)
qed

end

class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add

4.5 Groups

class group-add = minus + uminus + monoid-add +
  assumes left-minus [simp]: \(- a + a = 0\)
  assumes add-uminus-conv-diff [simp]: \( a + (- b) = a - b \)
begin

lemma diff-conv-add-uminus:
  \[ a - b = a + (- b) \]
  by simp

lemma minus-unique:
  assumes \( a + b = 0 \) shows \( - a = b \)
proof -
  have \( - a = - a + (a + b) \) using assms by simp
  also have \( \ldots = b \) by (simp add: add.assoc [symmetric])
  finally show \( \text{thesis} \).
qed

lemma minus-zero [simp]: \(- 0 = 0\)
proof -
  have \( 0 + 0 = 0 \) by (rule add-0-right)
  thus \( - 0 = 0 \) by (rule minus-unique)
qed

lemma minus-minus [simp]: \(- (- a) = a\)
proof -
  have \( - a + a = 0 \) by (rule left-minus)
  thus \( - (- a) = a \) by (rule minus-unique)
qed
lemma right-minus: \( a + - a = 0 \)
proof
  have \( a + - a = - (a) + - a \) by simp
  also have \( \ldots = 0 \) by (rule left-minus)
  finally show \( \text{thesis} \).
qed

lemma diff-self [simp]:
  \( a - a = 0 \)
using right-minus [of a] by simp

subclass cancel-semigroup-add
proof
  fix \( a \ b \ c :: 'a \)
  assume \( a + b = a + c \)
  then have \( - a + a + b = - a + a + c \)
  unfolding add.assoc by simp
  then show \( b = c \) by simp
next
  fix \( a \ b \ c :: 'a \)
  assume \( b + a = c + a \)
  then have \( b + a + - a = c + a + - a \) by simp
  then show \( b = c \) unfolding add.assoc by simp
qed

lemma minus-add-cancel [simp]:
  \(- a + (a + b) = b\)
by (simp add: add.assoc [symmetric])

lemma add-minus-cancel [simp]:
  \( a + (- a + b) = b\)
by (simp add: add.assoc [symmetric])

lemma diff-add-cancel [simp]:
  \( a - b + b = a\)
by (simp only: diff-conv-add-uminus add.assoc) simp

lemma add-diff-cancel [simp]:
  \( a + b - b = a\)
by (simp only: diff-conv-add-uminus add.assoc) simp

lemma minus-add:
  \( - (a + b) = - b + - a\)
proof
  have \( (a + b) + (- b + - a) = 0 \)
  by (simp only: add.assoc add-minus-cancel) simp
  then show \( - (a + b) = - b + - a \)
  by (rule minus-unique)
The next two equations can make the simplifier loop!
lemma equation-minus-iff:
\[ a = -b \iff b = -a \]
proof
  have \(- (a) = -b \iff a = b\) by (rule neg-equal-iff-equal)
  thus thesis by (simp add: eq-commute)
qed

lemma minus-equation-iff:
\(-a = b \iff b = a\)
proof
  have \(-a = -(b) \iff a = -b\) by (rule neg-equal-iff-equal)
  thus thesis by (simp add: eq-commute)
qed

lemma eq-neg-iff-add-eq-0:
\[ a = -b \iff a + b = 0 \]
proof
assume \(a = -b\) then show \(a + b = 0\) by simp
next
  assume \(a + b = 0\)
  moreover have \(a + (b + -b) = (a + b) + -b\)
    by (simp only: add.assoc)
  ultimately show \(a = -b\) by simp
qed

lemma add-eq-0-iff2:
\[ a + b = 0 \iff a = -b \]
by (fact eq-neg-iff-add-eq-0 [symmetric])

lemma neg-eq-iff-add-eq-0:
\[-a = b \iff a + b = 0\]
by (auto simp add: add-eq-0-iff2)

lemma add-eq-0-iff:
\[ a + b = 0 \iff b = -a \]
by (auto simp add: neg-eq-iff-add-eq-0 [symmetric])

lemma minus-diff-eq [simp]:
\[ -(a - b) = b - a \]
by (simp only: neg-eq-iff-add-eq-0 diff-conv-add-uminus add.assoc minus-add-cancel) simp

lemma add-diff-eq [algebra-simps, field-simps]:
\[ a + (b - c) = (a + b) - c \]
by (simp only: diff-conv-add-uminus add.assoc)

lemma diff-add-eq-diff-diff-swap:
\[ a - (b + c) = a - c - b \]
by (simp only: diff-conv-add-uminus add.assoc minus-add)
lemma diff-eq-eq [algebra-simps, field-simps]:
a - b = c \iff a = c + b
by auto

lemma eq-diff-eq [algebra-simps, field-simps]:
a = c - b \iff a + b = c
by auto

lemma diff-diff-eq2 [algebra-simps, field-simps]:
a - (b - c) = (a + c) - b
by (simp only: diff-cone-add-uminus add.assoc) simp

lemma diff-eq-diff-eq:
a - b = c - d \implies a = b \iff c = d
by (simp only: eq-iff-diff-eq-0 [of a b] eq-iff-diff-eq-0 [of c d])

end

class ab-group-add = minus + uminus + comm-monoid-add +
assumes ab-left-minus: - a + a = 0
assumes ab-add-uminus-conv-diff: a - b = a + (- b)
beginsubclass group-add
proof qed (simp-all add: ab-left-minus ab-add-uminus-conv-diff)

subclass cancel-comm-monoid-add
proof
fix a b c :: 'a
assume a + b = a + c
then have - a + a + b = - a + a + c
by (simp only: add.assoc)
then show b = c by simp
qed

lemma uminus-add-conv-diff [simp]:
- a + b = b - a
by (simp add: add.commute)

lemma minus-add-distrib [simp]:
- (a + b) = - a + - b
by (simp add: algebra-simps)

lemma diff-add-eq [algebra-simps, field-simps]:
(a - b) + c = (a + c) - b
by (simp add: algebra-simps)

lemma diff-diff-eq [algebra-simps, field-simps]:
(a - b) - c = a - (b + c)
by (simp add: algebra-simps)

lemma diff-add-eq-diff-diff:
a - (b + c) = a - b - c
using diff-add-eq-diff-diff-swap [of a c b] by (simp add: add.commute)

lemma add-diff-cancel-left [simp]:
(c + a) - (c + b) = a - b
by (simp add: algebra-simps)

end

4.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- [http://www.mathworld.com](http://www.mathworld.com) by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class ordered-ab-semigroup-add = order + ab-semigroup-add +
assumes add-left-mono: a ≤ b =&gt; c + a ≤ c + b
begin

lemma add-right-mono:
  a ≤ b =&gt; a + c ≤ b + c
by (simp add: add.commute [of - c] add-left-mono)

non-strict, in both arguments

lemma add-mono:
  a ≤ b =&gt; c ≤ d =&gt; a + c ≤ b + d
apply (erule add-right-mono [THEN order-trans])
apply (simp add: add.commute add-left-mono)
done

end

class ordered-cancel-ab-semigroup-add =
  ordered-ab-semigroup-add + cancel-ab-semigroup-add
begin
lemma add-strict-left-mono:
a < b \Rightarrow c + a < c + b
by (auto simp add: less-le add-left-mono)

lemma add-strict-right-mono:
a < b \Rightarrow a + c < b + c
by (simp add: add.commute [of - c] add-strict-left-mono)

Strict monotonicity in both arguments

lemma add-strict-mono:
a < b \Rightarrow c < d \Rightarrow a + c < b + d
apply (erule add-strict-right-mono [THEN less-trans])
apply (erule add-strict-left-mono)
done

lemma add-less-le-mono:
a < b \Rightarrow c \leq d \Rightarrow a + c < b + d
apply (erule add-strict-right-mono [THEN less-le-trans])
apply (erule add-left-mono)
done

lemma add-le-less-mono:
a \leq b \Rightarrow c < d \Rightarrow a + c < b + d
apply (erule add-right-mono [THEN le-less-trans])
apply (erule add-strict-left-mono)
done

end

class ordered-ab-semigroup-add-imp-le =
  ordered-cancel-ab-semigroup-add +
  assumes add-le-imp-le-left: c + a \leq c + b \Rightarrow a \leq b
begin

lemma add-less-imp-less-left:
  assumes less: c + a < c + b shows a < b
proof -
  from less have le: c + a \leq c + b by (simp add: order-le-less)
  have a \leq b
    apply (insert le)
    apply (drule add-le-imp-le-left)
    by (insert le, drule add-le-imp-le-left, assumption)
  moreover have a \neq b
  proof (rule ccontr)
    assume "(a \neq b)
    then have a = b by simp
    then have c + a = c + b by simp
    with less show False by simp
  qed

qed
ultimately show $a < b$ by (simp add: order-le-less)

qed

lemma add-less-imp-less-right:
  $a + c < b + c \implies a < b$
apply (rule add-less-imp-less-left [of $c$])
apply (simp add: add.commute)
done

lemma add-less-cancel-left [simp]:
  $c + a < c + b \iff a < b$
by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma add-less-cancel-right [simp]:
  $a + c < b + c \iff a < b$
by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma add-le-cancel-left [simp]:
  $c + a \leq c + b \iff a \leq b$
by (auto, drule add-le-imp-le-left, simp-all add: add-left-mono)

lemma add-le-cancel-right [simp]:
  $a + c \leq b + c \iff a \leq b$
by (simp add: add.commute [of $a$ $c$] add.commute [of $b$ $c$])

lemma add-le-imp-le-right:
  $a + c \leq b + c \implies a \leq b$
by simp

lemma max-add-distrib-left:
  $\max x y + z = \max (x + z) (y + z)$
unfolding max-def by auto

lemma min-add-distrib-left:
  $\min x y + z = \min (x + z) (y + z)$
unfolding min-def by auto

lemma max-add-distrib-right:
  $x + \max y z = \max (x + y) (x + z)$
unfolding max-def by auto

lemma min-add-distrib-right:
  $x + \min y z = \min (x + y) (x + z)$
unfolding min-def by auto

end

class ordered-cancel-comm-monoid-diff = comm-monoid-diff + ordered-ab-semigroup-add-imp-le +
THEORY “Groups”

assumes le-iff-add: \( a \leq b \iff (\exists \ c. \ b = a + c) \)

begin

context

fixes \( a \) \( b \)

assumes \( a \leq b \)

begin

lemma add-diff-inverse:
\( a + (b - a) = b \)
using \( \langle a \leq b \rangle \) by (auto simp add: le-iff-add)

lemma add-diff-assoc:
\( c + (b - a) = c + b - a \)
using \( \langle a \leq b \rangle \) by (auto simp add: le-iff-add add.left-commute [of c])

lemma add-diff-assoc2:
\( b - a + c = b + c - a \)
using \( \langle a \leq b \rangle \) by (auto simp add: le-iff-add add.assoc)

lemma diff-add-assoc:
\( c + b - a = c + (b - a) \)
using \( \langle a \leq b \rangle \) by (simp add: commute add-diff-assoc)

lemma diff-add-assoc2:
\( b + c - a = b - a + c \)
using \( \langle a \leq b \rangle \) by (simp add: commute add-diff-assoc)

lemma diff-diff-right:
\( c - (b - a) = c + a - b \)
by (simp add: add-diff-inverse add-diff-cancel-left [of \ a \ c \ b - \ a, symmetric] add.commute)

lemma diff-add:
\( b - a + a = b \)
by (simp add: commute add-diff-inverse)

lemma le-add-diff:
\( c \leq b + c - a \)
by (auto simp add: commute diff-add-assoc2 le-iff-add)

lemma le-imp-diff-is-add:
\( a \leq b \implies b - a = c \iff b = c + a \)
by (auto simp add: commute add-diff-inverse)

lemma le-diff-conv2:
\( c \leq b - a \iff c + a \leq b \) (is \ ?P \iff \ ?Q)
proof
assume \ ?P
then have \( c + a \leq b - a + a \) by (rule add-right-mono)
then show \(?Q\) by (simp add: add-diff-inverse add.commute)
next
  assume \(?Q\)
  then have \( a + c \leq a + (b - a) \) by (simp add: add-diff-inverse add.commute)
  then show \(?P\) by simp
qed

end

end

4.7 Support for reasoning about signs

class ordered-comm-monoid-add =
  ordered-cancel-ab-semigroup-add + comm-monoid-add
begin

lemma add-pos-nonneg:
  assumes \( 0 < a \) and \( 0 \leq b \) shows \( 0 < a + b \)
proof –
  have \( 0 + 0 < a + b \)
    using assms by (rule add-less-le-mono)
  then show \(?thesis\) by simp
qed

lemma add-pos-pos:
  assumes \( 0 < a \) and \( 0 < b \) shows \( 0 < a + b \)
by (rule add-pos-nonneg) (insert assms, auto)

lemma add-nonneg-pos:
  assumes \( 0 \leq a \) and \( 0 < b \) shows \( 0 < a + b \)
proof –
  have \( 0 + 0 < a + b \)
    using assms by (rule add-le-less-mono)
  then show \(?thesis\) by simp
qed

lemma add-nonneg-nonneg [simp]:
  assumes \( 0 \leq a \) and \( 0 \leq b \) shows \( 0 \leq a + b \)
proof –
  have \( 0 + 0 \leq a + b \)
    using assms by (rule add-mono)
  then show \(?thesis\) by simp
qed

lemma add-neg-nonpos:
  assumes \( a < 0 \) and \( b \leq 0 \) shows \( a + b < 0 \)
proof –
have \( a + b < 0 + 0 \)
  using assms by (rule add-less-le-mono)
then show ?thesis by simp
qed

lemma add-neg-neg:
  assumes \( a < 0 \) and \( b < 0 \) shows \( a + b < 0 \)
by (rule add-neg-nonpos) (insert assms, auto)

lemma add-nonpos-neg:
  assumes \( a \leq 0 \) and \( b < 0 \) shows \( a + b < 0 \)
proof
  have \( a + b < 0 + 0 \)
  using assms by (rule add-le-less-mono)
then show ?thesis by simp
qed

lemma add-nonpos-nonpos:
  assumes \( a \leq 0 \) and \( b \leq 0 \) shows \( a + b \leq 0 \)
proof
  have \( a + b \leq 0 + 0 \)
  using assms by (rule add-mono)
then show ?thesis by simp
qed

lemmas add-sign-intros =
  add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
  add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma add-nonneg-eq-0-iff:
  assumes \( x: 0 \leq x \) and \( y: 0 \leq y \)
  shows \( x + y = 0 \) \iff \( x = 0 \land y = 0 \)
proof (intro iffI conjI)
  have \( x = x + 0 \) by simp
  also have \( x + 0 \leq x + y \) using \( y \) by (rule add-left-mono)
  also assume \( x + y = 0 \)
  also have \( 0 \leq x \) using \( x \).
  finally show \( x = 0 \).
next
  have \( y = 0 + y \) by simp
  also have \( 0 + y \leq x + y \) using \( x \) by (rule add-right-mono)
  also assume \( x + y = 0 \)
  also have \( 0 \leq y \) using \( y \).
  finally show \( y = 0 \).
next
  assume \( x = 0 \land y = 0 \)
  then show \( x + y = 0 \) by simp
qed
lemma add-increasing: 
\( \theta \leq a \Rightarrow b \leq c \Rightarrow b \leq a + c \) 
by (insert add-mono [of \( \theta \) a b c], simp)

lemma add-increasing2: 
\( \theta \leq c \Rightarrow b \leq a \Rightarrow b \leq a + c \) 
by (simp add: add-increasing add.commute [of a])

lemma add-strict-increasing: 
\( \theta < a \Rightarrow b \leq c \Rightarrow b < a + c \) 
by (insert add-less-le-mono [of \( \theta \) a b c], simp)

lemma add-strict-increasing2: 
\( \theta < a \Rightarrow b < c \Rightarrow b < a + c \) 
by (insert add-le-less-mono [of \( \theta \) a b c], simp)

end

class ordered-ab-group-add = 
  ab-group-add + ordered-ab-semigroup-add
begin

subclass ordered-cancel-ab-semigroup-add ..

subclass ordered-ab-semigroup-add-imp-le 
proof 
  fix a b c :: 'a 
  assume c + a \leq c + b 
  hence \((-c) + (c + a) \leq (-c) + (c + b)\) by (rule add-left-mon) 
  hence \(((c + a) + c) \leq ((-c) + c) + b\) by (simp only: add.assoc) 
  thus a \leq b by simp 
qed

subclass ordered-comm-monoid-add ..

lemma add-less-same-cancel1 [simp]: 
  \( b + a < b \leftrightarrow a < 0 \) 
  using add-less-cancel-left [of \( - \) 0] by simp

lemma add-less-same-cancel2 [simp]: 
  \( a + b < b \leftrightarrow a < 0 \) 
  using add-less-cancel-right [of \( - \) 0] by simp

lemma less-add-same-cancel1 [simp]: 
  \( a < a + b \leftrightarrow 0 < b \) 
  using add-less-cancel-left [of \( - \) 0] by simp

lemma less-add-same-cancel2 [simp]: 
  \( a < b + a \leftrightarrow 0 < b \)
using add-less-cancel-right [of 0] by simp

lemma add-le-same-cancel1 [simp]:
  \( b + a \leq b \iff a \leq 0 \)
using add-le-cancel-left [of - - 0] by simp

lemma add-le-same-cancel2 [simp]:
  \( a + b \leq b \iff a \leq 0 \)
using add-le-cancel-right [of - - 0] by simp

lemma le-add-same-cancel1 [simp]:
  \( a \leq a + b \iff 0 \leq b \)
using add-le-cancel-left [of - 0] by simp

lemma le-add-same-cancel2 [simp]:
  \( a \leq b + a \iff 0 \leq b \)
using add-le-cancel-right [of 0] by simp

lemma max-diff-distrib-left:
  shows max x y - z = max (x - z) (y - z)
using max-add-distrib-left [of x y - z] by simp

lemma min-diff-distrib-left:
  shows min x y - z = min (x - z) (y - z)
using min-add-distrib-left [of x y - z] by simp

lemma le-imp-neg-le:
  assumes a \leq b shows -b \leq -a
proof -
  have -a + a \leq -a + b using (a \leq b) by (rule add-left-mono)
  then have 0 \leq -a + b by simp
  then have 0 + (-b) \leq (-a + b) + (-b) by (rule add-right-mono)
  then show ?thesis by (simp add: algebra-simps)
qed

lemma neg-le-iff-le [simp]: -b \leq -a \iff a \leq b
proof
  assume -b \leq -a
  hence -(a) \leq -(b) by (rule le-imp-neg-le)
  thus a \leq b by simp
next
  assume a \leq b
  thus -b \leq -a by (rule le-imp-neg-le)
qed

lemma neg-le-0-iff-le [simp]: -a \leq 0 \iff 0 \leq a
by (subst neg-le-iff-le [symmetric], simp)

lemma neg-0-le-iff-le [simp]: 0 \leq -a \iff a \leq 0
by (subst neg-le-iff-le [symmetric], simp)

lemma neg-less-iff-less [simp]: \(- b < - a \iff a < b\)
by (force simp add: less-le)

lemma neg-less-0-iff-less [simp]: \(- a < 0 \iff 0 < a\)
by (subst neg-less-iff-less [symmetric], simp)

lemma neg-0-less-iff-less [simp]: \(0 < - a \iff a < 0\)
by (subst neg-less-iff-less [symmetric], simp)

The next several equations can make the simplifier loop!

lemma less-minus-iff: \(a < - b \iff b < - a\)
proof -
  have \((- (-a) < - b) = (b < - a)\) by (rule neg-less-iff-less)
  thus \(?thesis by simp\)
qed

lemma minus-less-iff: \(- a < b \iff - b < a\)
proof -
  have \((- a < - (-b)) = (- b < a)\) by (rule neg-less-iff-less)
  thus \(?thesis by simp\)
qed

lemma le-minus-iff: \(a \leq - b \iff b \leq - a\)
proof -
  have mm: \(! a (b::a). (-(-a)) < -b \implies -(b) < -a\) by (simp only: minus-less-iff)
  have \((- (-a) <= -b) = (b <= - a)\)
    apply (auto simp only: le-less)
    apply (drule mm)
    apply (simp-all)
    apply (drule mm[simplified], assumption)
    done
  then show \(?thesis by simp\)
qed

lemma minus-le-iff: \(- a \leq b \iff - b \leq a\)
by (auto simp add: le-less minus-less-iff)

lemma diff-less-0-iff-less [simp]:
  \(a - b < 0 \iff a < b\)
proof -
  have \(a - b < 0 \iff a + (- b) < b + (- b)\) by simp
  also have \(\ldots \iff a < b\) by (simp only: add-less-cancel-right)
  finally show \(?thesis\).
qed

lemmas less-iff-diff-less-0 = diff-less-0-iff-less [symmetric]
lemma diff-less [algebra-simps, field-simps]:
  \( a - b < c \iff a < c + b \)
apply (subst less_iff_diff_less_0 [of a])
apply (rule less_iff_diff_less_0 [of - c, THEN ssubst])
apply (simp add: algebra-simps)
done

lemma less-diff-eq [algebra-simps, field-simps]:
  \( a < c - b \iff a + b < c \)
apply (subst less_iff_diff_less_0 [of a + b])
apply (rule less_iff_diff_less_0 [of a])
apply (simp add: algebra-simps)
done

lemma diff-le-eq [algebra-simps, field-simps]:
  \( a - b \leq c \iff a \leq c + b \)
by (auto simp add: le_less diff_less_eq)

lemma le-diff-eq [algebra-simps, field-simps]:
  \( a \leq c - b \iff a + b \leq c \)
by (auto simp add: le_less_less diff_less_eq)

lemma diff-le-0-iff-le [simp]:
  \( a - b \leq 0 \iff a \leq b \)
by (simp add: algebra-simps)

lemmas le_iff_diff_le_0 = diff_le_iff_diff_le [symmetric]

lemma diff-eq-diff-less:
  \( a - b = c - d \implies a < b \iff c < d \)
by (auto simp only: less_iff_diff_less_0 [of a b] less_iff_diff_less_0 [of c d])

lemma diff-eq-diff-less_eq:
  \( a - b = c - d \implies a \leq b \iff c \leq d \)
by (auto simp only: le_iff_diff_le_0 [of a b] le_iff_diff_le_0 [of c d])

lemma diff-mono: \( a \leq b \implies d \leq c \implies a - c \leq b - d \)
by (simp add: field-simps add_mono)

lemma diff-left-mono: \( b \leq a \implies c - a \leq c - b \)
by (simp add: field-simps)

lemma diff-right-mono: \( a \leq b \implies a - c \leq b - c \)
by (simp add: field-simps)

lemma diff-strict-mono: \( a < b \implies d < c \implies a - c < b - d \)
by (simp add: field-simps add_strict_mono)

lemma diff-strict-left-mono: \( b < a \implies c - a < c - b \)
by (simp add: field-simps)
lemma diff-strict-right-mono: $a < b \implies a - c < b - c$

by (simp add: field-simps)

end

ML-file Tools/group-cancel.ML

simproc-setup group-cancel-add ($a + b::'a::ab-group-add$) =
  ($fn phi => fn ss => try Group-Cancel.cancel-add-conv$)

simproc-setup group-cancel-diff ($a - b::'a::ab-group-add$) =
  ($fn phi => fn ss => try Group-Cancel.cancel-diff-conv$)

simproc-setup group-cancel-eq ($a = (b::'a::ab-group-add)$) =
  ($fn phi => fn ss => try Group-Cancel.cancel-eq-conv$)

simproc-setup group-cancel-le ($a \leq (b::'a::ordered-ab-group-add)$) =
  ($fn phi => fn ss => try Group-Cancel.cancel-le-conv$)

simproc-setup group-cancel-less ($a < (b::'a::ordered-ab-group-add)$) =
  ($fn phi => fn ss => try Group-Cancel.cancel-less-conv$)

class linordered-ab-semigroup-add =
  linorder + ordered-ab-semigroup-add

class linordered-cancel-ab-semigroup-add =
  linorder + ordered-cancel-ab-semigroup-add

begin

subclass linordered-ab-semigroup-add ..

subclass ordered-ab-semigroup-add-imp-le

proof
  fix $a b c :: 'a$
  assume le: $c + a \leq c + b$
  show $a \leq b$
  proof (rule ccontr)
    assume w: $\sim a \leq b$
    hence $b \leq a$ by (simp add: linorder-not-le)
    hence le2: $c + b \leq c + a$ by (rule add-left-mono)
    have $a = b$
      apply (insert le)
      apply (insert le2)
      apply (drule antisym, simp-all)
      done
    with $w$ show False
      by (simp add: linorder-not-le [symmetric])
  qed
  qed
class linordered-ab-group-add = linorder + ordered-ab-group-add begin

subclass linordered-cancel-ab-semigroup-add ..

lemma equal-neg-zero [simp]:
  a = - a ⇔ a = 0
proof
  assume a = 0 then show a = - a by simp
next
  assume A: a = - a show a = 0
proof (cases 0 ≤ a)
    case True with A have 0 ≤ - a by auto
        with le-minus-iff have a ≤ 0 by simp
        with True show ?thesis by (auto intro: order-trans)
next
    case False then have B: a ≤ 0 by auto
        with A have - a ≤ 0 by auto
        with B show ?thesis by (auto intro: order-trans)
qed

lemma neg-equal-zero [simp]:
  - a = a ⇔ a = 0
by (auto dest: sym)

lemma neg-less-eq-nonneg [simp]:
  - a ≤ a ⇔ 0 ≤ a
proof
  assume A: - a ≤ a show 0 ≤ a
proof (rule classical)
    assume ¬ 0 ≤ a
    then have a < 0 by auto
    with A have - a < 0 by (rule le-less-trans)
    then show ?thesis by auto
qed
next
  assume A: 0 ≤ a show - a ≤ a
proof (rule order-trans)
    show - a ≤ 0 using A by (simp add: minus-le-iff)
next
  show 0 ≤ a using A.
qed

lemma neg-less-pos [simp]:
THEORY "Groups"

\[ - a < a \iff 0 < a \]
by (auto simp add: less-le)

**lemma** less-eq-neg-nonpos [simp]:
\[ a \leq -a \iff a \leq 0 \]
using neg-less-eq-nonneg [of \(-a\)] by simp

**lemma** less-neg-neg [simp]:
\[ a < -a \iff a < 0 \]
using neg-less-pos [of \(-a\)] by simp

**lemma** double-zero [simp]:
\[ a + a = 0 \iff a = 0 \]
proof
assume **assm**: \(a + a = 0\)
then have \(-a = a\) by (rule minus-unique)
then show \(a = 0\) by (simp only: neg-equal-zero)
qed simp

**lemma** double-zero-sym [simp]:
\[ 0 = a + a \iff a = 0 \]
by (rule, drule sym) simp-all

**lemma** zero-less-double-add-iff-zero-less-single-add [simp]:
\[ 0 < a + a \iff 0 < a \]
proof
assume \(0 < a + a\)
then have \(0 - a < a\) by (simp only: diff-less-eq)
then have \(-a < a\) by simp
then show \(0 < a\) by simp
next
assume \(0 < a\)
with this have \(0 + 0 < a + a\)
by (rule add-strict-mono)
then show \(0 < a + a\) by simp
qed

**lemma** zero-le-double-add-iff-zero-le-single-add [simp]:
\[ 0 \leq a + a \iff 0 \leq a \]
by (auto simp add: le-less)

**lemma** double-add-less-zero-iff-single-add-less-zero [simp]:
\[ a + a < 0 \iff a < 0 \]
proof
have \(\neg a + a < 0 \iff \neg a < 0\)
by (simp add: not-less)
then show ?thesis by simp
qed
lemma double-add-le-zero-iff-single-add-le-zero [simp]:
\[ a + a \leq 0 \iff a \leq 0 \]
proof
  have \( \neg a + a \leq 0 \iff \neg a \leq 0 \)
  by (simp add: not-le)
  then show \( \text{thesis} \) by simp
qed

lemma minus-max-eq-min:
\[ -\max x y = \min (-x) (-y) \]
by (auto simp add: max-def min-def)

lemma minus-min-eq-max:
\[ -\min x y = \max (-x) (-y) \]
by (auto simp add: max-def min-def)

end

class abs =
  fixes abs :: 'a \Rightarrow 'a
begin
  notation (xsymbols)
  abs \((|\cdot|)\)
  notation (HTML output)
  abs \((|\cdot|)\)
end

class sgn =
  fixes sgn :: 'a \Rightarrow 'a
begin
  class abs-if = minus + uminus + ord + zero + abs +
  assumes abs-if: \[ |a| = (if a < 0 then \(-a\) else a) \]
  class sgn-if = minus + uminus + zero + one + ord + sgn +
  assumes sgn-if: \( \text{sgn } x = (if x = 0 \text{ then } 0 \text{ else if } 0 < x \text{ then } 1 \text{ else } -1) \)
begin
  lemma sgn0 [simp]: \( \text{sgn } 0 = 0 \)
  by (simp add: sgn-if)
end

class ordered-ab-group-add-abs = ordered-ab-group-add + abs +
assumes abs-ge-zero [simp]: \[ |a| \geq 0 \]
and abs-ge-self: \[ a \leq |a| \]
and abs-leI: \[ a \leq b \implies -a \leq b \implies |a| \leq b \]
and abs-minus-cancel [simp]: \(|-a| = |a|\)
and abs-triangle-ineq: \(|a + b| \leq |a| + |b|\)

```
lemma abs-minus-le-zero: \(-|a| \leq 0\)
  unfolding neg-le-0-iff by simp

lemma abs-of-nonneg [simp]:
  assumes nonneg: \(0 \leq a\) shows \(|a| = a\)
  proof (rule antisym)
    from nonneg le-imp-neg-le have \(-a \leq 0\) by simp
    then show \(|a| \leq a\) by (auto intro: abs-leI)
  qed (rule abs-ge-self)

lemma abs-idempotent [simp]: \(||a|| = |a|\)
  by (rule antisym)
    (auto intro!: abs-ge-self abs-leI order-trans [of \(-|a|\) 0 \(|a|\)])

lemma abs-eq-0 [simp]: \(|a| = 0 \iff a = 0\)
  proof
    have \(|a| = 0 \implies a = 0\)
  proof (rule antisym)
    assume zero: \(|a| = 0\)
    with abs-ge-self show \(-a \leq 0\) by auto
    from zero have \(|-a| = 0\) by simp
    with abs-ge-self show \(-a \leq 0\) by auto
    with neg-le-0-iff-le show \(0 \leq a\) by auto
  qed
  then show ?thesis by auto
  qed

lemma abs-zero [simp]: \(|0| = 0\)
  by simp

lemma abs-0-eq [simp]: \(0 = |a| \iff a = 0\)
  proof
    have \(0 = |a| \iff |a| = 0\) by (simp only: eq-ac)
    thus ?thesis by simp
  qed

lemma abs-le-zero-iff [simp]: \(|a| \leq 0 \iff a = 0\)
  proof
    assume \(|a| \leq 0\)
    then have \(|a| = 0\) by (rule antisym) simp
    thus \(a = 0\) by simp
  next
    assume \(a = 0\)
    thus \(|a| \leq 0\) by simp
```
qed

lemma  zero-less-abs-iff [simp]: \(0 < |a| \iff a \neq 0\)
by (simp add: less-le)

lemma  abs-not-less-zero [simp]: \(\neg |a| < 0\)
proof –
  have \(a \wedge x. x \leq y \Longrightarrow \neg y < x\) by auto
  show ?thesis by (simp add: a)
qed

lemma  abs-ge-minus-self: \(- a \leq |a|\)
proof –
  have \(- a \leq | - a|\) by (rule abs-ge-self)
  then show ?thesis by simp
qed

lemma  abs-minus-commute:
  \(|a - b| = |b - a|\)
proof –
  have \(|a - b| = | - (a - b)|\) by (simp only: abs-minus-cancel)
  also have \(\ldots = |b - a|\) by simp
  finally show ?thesis .
qed

lemma  abs-of-pos: \(0 < a \implies |a| = a\)
by (rule abs-of-nonneg, rule less-imp-le)

lemma  abs-of-nonpos [simp]:
  assumes \(a \leq 0\) shows \(|a| = - a\)
proof –
  let \(?b = - a\)
  have \(- ?b \leq 0 \implies | - ?b| = -( - ?b)\)
  unfolding abs-minus-cancel [of ?b]
  unfolding neg-le-0-iff-le [of ?b]
  unfolding minus-minus by (erule abs-of-nonneg)
  then show ?thesis using assms by auto
qed

lemma  abs-of-neg: \(a < 0 \implies |a| = - a\)
by (rule abs-of-nonneg, rule less-imp-le)

lemma  abs-le-D1: \(|a| \leq b \implies a \leq b\)
by (insert abs-ge-self, blast intro: order-trans)

lemma  abs-le-D2: \(|a| \leq b \implies - a \leq b\)
by (insert abs-le-D1 [of \(- a\)], simp)

lemma  abs-le-iff: \(|a| \leq b \iff a \leq b \land - a \leq b\)
by (blast intro: abs-leI dest: abs-le-D1 abs-le-D2)

lemma abs-triangle-ineq2: \(|a| - |b| \leq |a - b|

proof -
  have \(|a| = |b + (a - b)|
  by (simp add: algebra-simps)
  then have \(|a| \leq |b| + |a - b|
  by (simp add: abs-triangle-ineq)
  then show ?thesis
  by (simp add: algebra-simps)
qed

lemma abs-triangle-ineq2-sym: \(|a| - |b| \leq |b - a|

by (simp only: abs-minus-commute [of b] abs-triangle-ineq2)

lemma abs-triangle-ineq3: \(||a| - |b|| \leq |a - b|

by (simp add: abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym)

lemma abs-triangle-ineq4: \(|a - b| \leq |a| + |b|

proof -
  have \(|a - b| = |a + (a - b)|
  by (simp add: algebra-simps)
  also have \(... \leq |a| + |- b|
  by (rule abs-triangle-ineq)
  finally show ?thesis by simp
qed

lemma abs-diff-triangle-ineq: \(|a + b - (c + d)| \leq |a - c| + |b - d|

proof -
  have \(|a + b - (c+d)| = |(a-c) + (b-d)|
  by (simp add: algebra-simps)
  also have \(... \leq |a-c| + |b-d|
  by (rule abs-triangle-ineq)
  finally show ?thesis .
qed

lemma abs-add-abs [simp]:
  \(||a| + ||b|| = |a| + |b|\) (is \(?L = ?R\)

proof (rule antisym)
  show \(?L \geq ?R\) by (rule abs-ge-self)
next
  have \(?L \leq ||a|| + ||b||\) by (rule abs-triangle-ineq)
  also have \(... = ?R\) by simp
  finally show \(?L \leq ?R\).
qed

4.8 Tools setup

lemma add-mono-thms-linordered-semiring:
  fixes i j k :: 'a::ordered-ab-semigroup-add
  shows \(i \leq j \land k \leq l \Longrightarrow i + k \leq j + l\)
and $i = j \land k \leq l \implies i + k \leq j + l$
and $i \leq j \land k = l \implies i + k \leq j + l$
and $i = j \land k = l \implies i + k = j + l$
by (rule add-mono, clarify)+

lemma add-mono-thms-linordered-field:
fixes $i\ j\ k\ ::\ 'a\::\ ordered-cancel-ab-semigroup-add$
shows $i < j \land k = l \implies i + k < j + l$
and $i = j \land k < l \implies i + k < j + l$
and $i < j \land k \leq l \implies i + k < j + l$
and $i \leq j \land k < l \implies i + k < j + l$
and $i < j \land k < l \implies i + k < j + l$
by (auto intro: add-strict-right-mono add-strict-left-mono
add-less-le-mono add-le-less-mono add-strict-mono)

code-identifier

code-module Groups $\rightarrow$ SML Arith and OCaml Arith and Haskell Arith

end

5 Lattices: Abstract lattices

theory Lattices
imports Groups
begin

5.1 Abstract semilattice

These locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

no-notation times (infixl * 70)
no-notation Groups.one (1)

locale semilattice = abel-semigroup +
  assumes idem [simp]: $a \ast a = a$
begin

lemma left-idem [simp]: $a \ast (a \ast b) = a \ast b$
by (simp add: assoc [symmetric])

lemma right-idem [simp]: $(a \ast b) \ast b = a \ast b$
by (simp add: assoc)

end

locale semilattice-neutr = semilattice + comm-monoid
locale semilattice-order = semilattice +
  fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix \leq)
  and less :: 'a ⇒ 'a ⇒ bool (infix \prec)
assumes order-iff: a \leq b ⟷ a = a \ast b
  and semilattice-strict-iff-order: a \prec b ⟷ a \leq b \land a \neq b
begin

lemma orderI:
a = a \ast b \Rightarrow a \leq b
  by (simp add: order-iff)

lemma orderE:
assumes a \leq b
obtains a = a \ast b
  using assms by (unfold order-iff)

sublocale ordering less-eq less
proof
  fix a b
  show a \prec b ⟷ a \less b \land a \neq b
    by (fact semilattice-strict-iff-order)
next
  fix a
  show a \less a
    by (simp add: order-iff)
next
  fix a b
  assume a \less b \less a
  then have a = a \ast b \ast b = b
    by (simp-all add: order-iff commute)
  then show a = b by simp
next
  fix a b c
  assume a \less b \less c
  then have a = a \ast b \ast b = b \ast c
    by (simp-all add: order-iff commute)
  then have a = a \ast (b \ast c)
    by simp
  then have a = (a \ast b) \ast c
    by (simp add: assoc)
  with \langle a = a \ast b \rangle [symmetric] have a = a \ast c by simp
  then show a \less c by (rule orderI)
qed

lemma cobounded1 [simp]:
a \ast b \leq a
  by (simp add: order-iff commute)

lemma cobounded2 [simp]:
THEORY “Lattices”

```
  a * b ≤ b
  by (simp add: order-iff)

lemma boundedI:
  assumes a ≤ b and a ≤ c
  shows a ≤ b * c
proof (rule orderI)
  from assms obtain a * b = a and a * c = a by (auto elim!: orderE)
  then show a = a * (b * c) by (simp add: assoc [symmetric])
qed

lemma boundedE:
  assumes a ≤ b * c
  obtains a ≤ b and a ≤ c
  using assms by (blast intro: trans coboundedI cobounded2)

lemma bounded-iff [simp]:
  a ≤ b * c ←→ a ≤ b ∧ a ≤ c
  by (blast intro: boundedI elim: boundedE)

lemma strict-boundedE:
  assumes a ≺ b * c
  obtains a ≺ b and a ≺ c
  using assms by (auto simp add: commute strict-iff-order elim: orderE intro!: that)+

lemma coboundedI1:
  a ≤ c ⇒ a * b ≤ c
  by (rule trans) auto

lemma coboundedI2:
  b ≤ c ⇒ a * b ≤ c
  by (rule trans) auto

lemma strict-coboundedI1:
  a ≺ c ⇒ a * b ≺ c
  using irrefl
  by (auto intro: not-eq-order-implies-strict coboundedI1 strict-implies-order elim: strict-boundedE)

lemma strict-coboundedI2:
  b ≺ c ⇒ a * b ≺ c
  using strict-coboundedI1 [of b c a] by (simp add: commute)

lemma mono: a ≤ c ⇒ b ≤ d ⇒ a * b ≤ c * d
  by (blast intro: boundedI coboundedI1 coboundedI2)

lemma absorb1: a ≤ b ⇒ a * b = a
  by (rule antisym) (auto simp add: refl)
```
lemma absorb2: $b \preceq a \Rightarrow a \ast b = b$
  by (rule antisym) (auto simp add: refl)

lemma absorb-iff1: $a \preceq b \iff a \ast b = a$
  using order-iff by auto

lemma absorb-iff2: $b \preceq a \iff a \ast b = b$
  using order-iff by (auto simp add: commute)

end

locale semilattice-neutr-order
  begin
    sublocale ordering-top less-eq less 1
      by default (simp add: order-iff)
  end

notation times (infixl \ast 70)
notation Groups. one (1)

5.2 Syntactic infimum and supremum operations

class inf =
  fixes inf :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \cap 70)

class sup =
  fixes sup :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \cup 65)

5.3 Concrete lattices

notation
  less-eq (infix \subseteq 50) and
  less (infix \subseteq 50)

class semilattice-inf = order + inf +
  assumes inf-le1 [simp]: $x \cap y \subseteq x$
  and inf-le2 [simp]: $x \cap y \subseteq y$
  and inf-greatest: $x \subseteq y \Rightarrow x \subseteq z \Rightarrow x \subseteq y \cap z$

class semilattice-sup = order + sup +
  assumes sup-le1 [simp]: $x \subseteq x \cup y$
  and sup-le2 [simp]: $y \subseteq x \cup y$
  and sup-least: $y \subseteq x \Rightarrow z \subseteq x \Rightarrow y \cup z \subseteq x$

begin

Dual lattice

lemma dual-semilattice:
class semilattice-inf sup greater-eq greater
by (rule class_semilattice-inf.intro, rule dual-order)
(unfold-locales, simp-all add: sup-least)
end

class lattice = semilattice-inf + semilattice-sup

5.3.1 Intro and elim rules
context semilattice-inf begin

lemma le-infI1:
  a ⊑ x ⇒ a ∩ b ⊑ x
  by (rule order-trans) auto

lemma le-infI2:
  b ⊑ x ⇒ a ∩ b ⊑ x
  by (rule order-trans) auto

lemma le-infI: x ⊑ a ⇒ x ⊑ b ⇒ x ⊑ a ∩ b
  by (fact inf-greatest)

lemma le-infE: x ⊑ a ∩ b ⇒ (x ⊑ a ⇒ x ⊑ b ⇒ P) ⇒ P
  by (blast intro: order-trans inf-le1 inf-le2)

lemma le-inf-iff:
  x ⊑ y ∩ z ↔ x ⊑ y ∧ x ⊑ z
  by (blast intro: le-infI elim: le-infE)

lemma le-iff-inf:
  x ⊑ y ⊨ x ∩ y = x
  by (auto intro: le-infI1 antisym dest: eq-iff [THEN iffD1] simp add: le-inf-iff)

lemma inf-mono: a ⊑ c ⇒ b ⊑ d ⇒ a ∩ b ⊑ c ∩ d
  by (fast intro: inf-greatest le-infI1 le-infI2)

lemma mono-inf:
  fixes f :: 'a ⇒ 'b::semilattice-inf
  shows mono f ⇒ f (A ∩ B) ⊑ f A ∩ f B
  by (auto simp add: mono-def intro: Lattices.inf-greatest)
end

context semilattice-sup begin

lemma le-supI1:
THEORY “Lattices”

\[ x \subseteq a \implies x \subseteq a \sqcup b \]
by (rule order-trans) auto

lemma le-supI2:
\[ x \subseteq b \implies x \subseteq a \sqcup b \]
by (rule order-trans) auto

lemma le-supI:
\[ a \subseteq x \implies b \subseteq x \implies a \sqcup b \subseteq x \]
by (fact sup-least)

lemma le-supE:
\[ a \sqcup b \subseteq x \implies (a \subseteq x \implies b \subseteq x \implies P) \implies P \]
by (blast intro: order-trans sup-ge1 sup-ge2)

lemma le-sup-iff:
\[ x \sqcup y \subseteq z \iff x \subseteq z \land y \subseteq z \]
by (blast intro: le-supI elim: le-supE)

lemma le-iff-sup:
\[ x \subseteq y \iff x \sqcup y = y \]
by (auto intro: le-supI2 antisym dest: eq-iff \[THEN iffD1\] simp add: le-sup-iff)

lemma sup-mono:
\[ a \subseteq c \implies b \subseteq d \implies a \sqcup b \subseteq c \sqcup d \]
by (fast intro: sup-least le-supI1 le-supI2)

lemma mono-sup:
fixes f :: 'a \Rightarrow 'b::semilattice-sup
shows mono f \[\implies f A \sqcup f B \subseteq f (A \sqcup B)\]
by (auto simp add: mono-def intro: Lattices.sup-least)

end

5.3.2 Equational laws

context semilattice-inf
begin

sublocale inf!: semilattice inf
proof
fix a b c
show \((a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)\)
  by (rule antisym) (auto intro: le-infI1 le-infI2 simp add: le-inf-iff)
show \(a \sqcap b = b \sqcap a\)
  by (rule antisym) (auto simp add: le-inf-iff)
show \(a \sqcap a = a\)
  by (rule antisym) (auto simp add: le-inf-iff)
qed
sublocale inf!: semilattice-order inf less-eq less
  by default (auto simp add: le-iff-inf less-le)

lemma inf-assoc: \((x \cap y) \cap z = x \cap (y \cap z)\)
  by (fact inf.assoc)

lemma inf-commute: \((x \cap y) = (y \cap x)\)
  by (fact inf.commute)

lemma inf-left-commute: \(x \cap (y \cap z) = y \cap (x \cap z)\)
  by (fact inf.left-commute)

lemma inf-idem: \(x \cap x = x\)
  by (fact inf.idem)

lemma inf-left-idem: \(x \cap (x \cap y) = x \cap y\)
  by (fact inf.left-idem)

lemma inf-right-idem: \((x \cap y) \cap y = x \cap y\)
  by (fact inf.right-idem)

lemma inf-absorb1: \(x \sqsubseteq y \implies x \sqcap y = x\)
  by (rule antisym) auto

lemma inf-absorb2: \(y \sqsubseteq x \implies x \sqcap y = y\)
  by (rule antisym) auto

lemmas inf-aci = inf-commute inf-assoc inf-left-commute inf-left-idem

end

context semilattice-sup
begin

sublocale sup!: semilattice sup
  proof
    fix \(a\) \(b\) \(c\)
    show \((a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)\)
      by (rule antisym) (auto intro: le-supI1 le-supI2 simp add: le-sup-iff)
    show \(a \sqcup b = b \sqcup a\)
      by (rule antisym) (auto simp add: le-sup-iff)
    show \(a \sqcup a = a\)
      by (rule antisym) (auto simp add: le-sup-iff)
  qed

sublocale sup!: semilattice-order sup greater-eq greater
  by default (auto simp add: le-iff-sup sup.commute less-le)

lemma sup-assoc: \((x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)\)
by (fact sup.assoc)

lemma sup-commute: $(x \sqcup y) = (y \sqcup x)$
  by (fact sup.commute)

lemma sup-left-commute: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
  by (fact sup.left-commute)

lemma sup-idem: $x \sqcup x = x$
  by (fact sup.idem)

lemma sup-left-idem [simp]: $x \sqcup (x \sqcup y) = x \sqcup y$
  by (fact sup.left-idem)

lemma sup-absorb1: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
  by (rule antisym) auto

lemma sup-absorb2: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
  by (rule antisym) auto

lemmas sup-aci = sup-commute sup-assoc sup-left-commute sup-left-idem

end

context lattice
begin

lemma dual-lattice:
  class.\text{lattice sup } (\text{op } \geq) (\text{op } >) inf
  by (rule class.lattice.intro, rule dual-semilattice, rule class.semilattice-sup.intro, rule dual-order)
  (unfold-\text{locales}, auto)

lemma inf-sup-absorb [simp]: $x \sqcap (x \sqcup y) = x$
  by (blast intro: antisym inf-le1 inf-greatest sup-ge1)

lemma sup-inf-absorb [simp]: $x \sqcup (x \sqcap y) = x$
  by (blast intro: antisym sup-ge1 sup-least inf-le1)

lemmas inf-sup-aci = inf-aci sup-aci

lemmas inf-sup-ord = inf-le1 inf-le2 sup-ge1 sup-ge2

Towards distributivity

lemma distrib-sup-le: $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$
  by (auto intro: le-infl1 le-infl2 le-supI1 le-supI2)

lemma distrib-inf-le: $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$
  by (auto intro: le-infl1 le-infl2 le-supI1 le-supI2)
If you have one of them, you have them all.

**lemma** **distrib-imp1:**
**assumes** $D: \forall x \, y \, z. \ x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$
**shows** $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$
**proof** -
  have $x \cup (y \cap z) = (x \cup (x \cap z)) \cup (y \cap z)$ by simp
  also have \ldots $= x \cup (z \cap (x \cup y))$
    by \,(simp add: $D$ inf-commute sup-assoc del: sup-inf-absorb)
  also have \ldots $= ((x \cup y) \cap x) \cup ((x \cup y) \cap z)$
    by\,(simp add: inf-commute)
  also have \ldots $= (x \cup y) \cap (x \cup z)$ by\,(simp add:$D$)
  finally show \?thesis .
**qed**

**lemma** **distrib-imp2:**
**assumes** $D: \forall x \, y \, z. \ x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$
**shows** $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$
**proof** -
  have $x \cap (y \cup z) = (x \cap (x \cup z)) \cap (y \cup z)$ by simp
  also have \ldots $= x \cap (z \cup (x \cap y))$
    by \,(simp add: $D$ sup-commute inf-assoc del: inf-sup-absorb)
  also have \ldots $= ((x \cup y) \cap x) \cap ((x \cup y) \cap z)$
    by\,(simp add: sup-commute)
  also have \ldots $= (x \cap y) \cup (x \cap z)$ by\,(simp add:$D$)
  finally show \?thesis .
**qed**

end

5.3.3 Strict order

c**ontext** **semilattice-inf**
begin

**lemma** **less-infI1:**
$a \sqsubseteq x \Longrightarrow a \cap b \sqsubseteq x$
by \,(auto simp add: less-le inf-absorb1 intro: le-infI1)

**lemma** **less-infI2:**
$b \sqsubseteq x \Longrightarrow a \cap b \sqsubseteq x$
by \,(auto simp add: less-le inf-absorb2 intro: le-infI2)
end

**context** **semilattice-sup**
begin

**lemma** **less-supI1:**
$x \sqsubseteq a \Longrightarrow x \sqsubseteq a \sqcup b$

end
using dual-semilattice
by (rule semilattice-inf.less-infI1)

lemma less-supI2:
x ⊏ b ⇒ x ⊏ a ⊔ b
using dual-semilattice
by (rule semilattice-inf.less-infI2)

end

5.4 Distributive lattices

class distrib-lattice = lattice +
assumes sup-inf-distrib1: x ⊔ (y ∩ z) = (x ⊔ y) ∩ (x ⊔ z)

context distrib-lattice
begin

lemma sup-inf-distrib2:
(y ∩ z) ⊔ x = (y ∪ x) ∩ (z ∪ x)
by (simp add: sup-commute sup-inf-distrib1)

lemma inf-sup-distrib1:
x ∩ (y ∪ z) = (x ∩ y) ∪ (x ∩ z)
by (rule distrib-imp2 [OF sup-inf-distrib1])

lemma inf-sup-distrib2:
(y ∪ z) ∩ x = (y ∩ x) ∪ (z ∩ x)
by (simp add: inf-commute inf-sup-distrib1)

lemma dual-distrib-lattice:
class distrib-lattice sup (op ≥) (op >) inf
by (rule class.distrib-lattice.intro, rule dual-lattice)
(unfold-locales, fact inf-sup-distrib1)

lemmas sup-inf-distrib =
sup-inf-distrib1 sup-inf-distrib2

lemmas inf-sup-distrib =
inf-sup-distrib1 inf-sup-distrib2

lemmas distrib =
sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

5.5 Bounded lattices and boolean algebras

class bounded-semilattice-inf-top = semilattice-inf + order-top
begin
sublocale inf-top!: semilattice-neutr inf top
+ inf-top!: semilattice-neutr-order inf top less-eq less
proof
fix x
show x \cap \top = x
  by (rule inf-absorb1) simp
qed
end

class bounded-semilattice-sup-bot = semilattice-sup + order-bot
begin

sublocale sup-bot!: semilattice-neutr sup bot
+ sup-bot!: semilattice-neutr-order sup bot greater-eq greater
proof
fix x
show x \sqcup \bot = x
  by (rule sup-absorb1) simp
qed
end

class bounded-lattice-bot = lattice + order-bot
begin
subclass bounded-semilattice-sup-bot ..

lemma inf-bot-left [simp]:
  \bot \cap x = \bot
  by (rule inf-absorb1) simp

lemma inf-bot-right [simp]:
  x \cap \bot = \bot
  by (rule inf-absorb2) simp

lemma sup-bot-left:
  \bot \sqcup x = x
  by (fact sup-bot.left-neutral)

lemma sup-bot-right:
  x \sqcup \bot = x
  by (fact sup-bot.right-neutral)

lemma sup-eq-bot-iff [simp]:
  x \sqcup y = \bot \iff x = \bot \land y = \bot
  by (simp add: eq-iff)
lemma bot-eq-sup-iff [simp]:
\( \bot = x \sqcup y \iff x = \bot \land y = \bot \)
by (simp add: eq-iff)
end

class bounded-lattice-top = lattice + order-top
begin
subclass bounded-semilattice-inf-top ..

lemma sup-top-left [simp]:
\( \top \sqcup x = \top \)
by (rule sup-absorb1) simp

lemma sup-top-right [simp]:
\( x \sqcup \top = \top \)
by (rule sup-absorb2) simp

lemma inf-top-left:
\( \top \sqcap x = x \)
by (fact inf-top.left-neutral)

lemma inf-top-right:
\( x \sqcap \top = x \)
by (fact inf-top.right-neutral)

lemma inf-eq-top-iff [simp]:
\( x \sqcap y = \top \iff x = \top \land y = \top \)
by (simp add: eq-iff)
end

class bounded-lattice = lattice + order-bot + order-top
begin
subclass bounded-lattice-bot ..
subclass bounded-lattice-top ..

lemma dual-bounded-lattice:
class bounded-lattice sup greater-eq greater inf \( \top \bot \)
by unfold-locales (auto simp add: less-le-not-le)
end

class boolean-algebra = distrib-lattice + bounded-lattice + minus + uminus +
assumes inf-compl-bot: \( x \sqcap - x = \bot \)
and sup-compl-top: \( x \sqcup - x = \top \)
assumes diff-eq: \( x - y = x \sqcap - y \)
begin

lemma dual-boolean-algebra:
class boolean-algebra (\lambda x y. x \sqcup - y) uminus sup greater-eq greater inf \top \bot
by (rule class.boolean-algebra.intro, rule dual-bounded-lattice, rule dual-distrib-lattice)
   (unfold-locale, auto simp add: inf-compl-bot sup-compl-top diff-eq)

lemma compl-inf-bot [simp]:
- x \cap x = \bot
by (simp add: inf-commute inf-compl-bot)

lemma compl-sup-top [simp]:
- x \sqcup x = \top
by (simp add: sup-commute sup-compl-top)

lemma compl-unique:
assumes x \cap y = \bot
  and x \sqcup y = \top
shows - x = y
proof -
  have (x \cap - x) \sqcup (- x \cap y) = (x \cap y) \sqcup (- x \cap y)
    using inf-compl-bot assms (1) by simp
  then have (- x \cap x) \sqcup (- x \cap y) = (y \cap x) \sqcup (y \cap - x)
    by (simp add: inf-commute)
  then have - x \cap (x \sqcup y) = y \cap (x \sqcup - x)
    by (simp add: inf-sup-distrib1)
  then have - x \cap \top = y \cap \top
    using sup-compl-top assms (2) by simp
  then show - x = y by simp
qed

lemma double-compl [simp]:
- (- x) = x
using compl-inf-bot compl-sup-top by (rule compl-unique)

lemma compl-eq-compl-iff [simp]:
- x = - y \iff x = y
proof
  assume - x = - y
  then have - (- x) = - (- y) by (rule arg-cong)
  then show x = y by simp
next
  assume x = y
  then show - x = - y by simp
qed

lemma compl-bot-eq [simp]:
- \bot = \top
proof -
from sup-compl-top have ⊥⊔⊥ − ⊥ = ⊤.
then show ?thesis by simp
qed

lemma compl-top-eq [simp]:
− ⊤ = ⊥
proof −
from inf-compl-bot have ⊤∩− ⊤ = ⊥.
then show ?thesis by simp
qed

lemma compl-inf [simp]:
− (x ∩ y) = − x ⊔ − y
proof (rule compl-unique)
  have (x ∩ y) ∩ (− x ⊔ − y) = (y ∩ (x ∩ − x)) ∪ (x ∩ (y ∩ − y))
    by (simp only: inf-sup-distrib inf-aci)
  then show (x ∩ y) ∩ (− x ⊔ − y) = ⊥
    by (simp add: inf-compl-bot)
next
  have (x ∩ y) ∪ (− x ⊔ − y) = (− y ∪ (x ∪ − x)) ∩ (− x ∪ (y ∪ − y))
    by (simp only: sup-inf-distrib sup-aci)
  then show (x ∩ y) ∪ (− x ⊔ − y) = ⊤
    by (simp add: sup-compl-top)
qed

lemma compl-sup [simp]:
− (x ⊔ y) = − x ∩ − y
using dual-boolean-algebra
by (rule boolean-algebra.compl-inf)

lemma compl-mono:
x ⊑ y −→ − y ⊑ − x
proof −
  assume x ⊑ y
  then have x ∪ y = y by (simp only: le-iff-sup)
  then have − (x ∪ y) = − y by simp
  then have − x ∩ − y = − y by simp
  then have − y ∩ − x = − y by (simp only: inf-commute)
  then show − y ⊑ − x by (simp only: le-iff-inf)
qed

lemma compl-le-compl-iff [simp]:
− x ⊑ − y −→ y ⊑ x
by (auto dest: compl-mono)

lemma compl-le-swap1:
  assumes y ⊑ − x shows x ⊑ − y
proof −
  from assms have − (− x) ⊑ − y by (simp only: compl-le-compl-iff)
then show ?thesis by simp
qed

lemma compl-le-swap2:
  assumes − y ⊑ x shows − x ⊑ y
proof
  from assms have − x ⊑ − (− y) by (simp only: compl-le-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-compl-iff:
  − x ⊏ − y ⇔ y ⊏ x
by (auto simp add: less-le)

lemma compl-less-swap1:
  assumes y ⊏ − x shows x ⊏ − y
proof
  from assms have − (− x) ⊏ − y by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-swap2:
  assumes − y ⊏ x shows − x ⊏ y
proof
  from assms have − x ⊏ − (− y) by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed

end

5.6 min/max as special case of lattice

context linorder
begin

sublocale min!: semilattice-order min less-eq less
+ max!: semilattice-order max greater-eq greater
by default (auto simp add: min-def max-def)

lemma min-le iff-disj:
  min x y ≤ z ⇔ x ≤ z ∨ y ≤ z
unfolding min-def using linear by (auto intro: order-trans)

lemma le-max iff-disj:
  z ≤ max x y ⇔ z ≤ x ∨ z ≤ y
unfolding max-def using linear by (auto intro: order-trans)

lemma min-less iff-disj:
  min x y < z ⇔ x < z ∨ y < z
unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma less-max-iff-disj:
  \( z < \max x y \iff z < x \lor z < y \)
unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-less-iff-conj [simp]:
  \( z < \min x y \iff z < x \land z < y \)
unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma max-less-iff-conj [simp]:
  \( \max x y < z \iff x < z \land y < z \)
unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-max-distrib1:
  \( \min (\max b c) a = \max (\min b a) (\min c a) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma min-max-distrib2:
  \( \min a (\max b c) = \max (\min a b) (\min a c) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma max-min-distrib1:
  \( \max (\min b c) a = \min (\max b a) (\max c a) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma max-min-distrib2:
  \( \max a (\min b c) = \min (\max a b) (\max a c) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemmas min-max-distrib = min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2

lemma split-min [no-atp]:
  \( P (\min i j) \iff (i \leq j \rightarrow P i) \land (\neg i \leq j \rightarrow P j) \)
  by (simp add: min-def)

lemma split-max [no-atp]:
  \( P (\max i j) \iff (i \leq j \rightarrow P j) \land (\neg i \leq j \rightarrow P i) \)
  by (simp add: max-def)

lemma min-of-mono:
  fixes \( f :: 'a \Rightarrow 'b::linorder \)
  shows \( mono f \Rightarrow \min (f m) (f n) = f (\min m n) \)
  by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)
lemma max-of-mono:
fixes f :: 'a ⇒ 'b::linorder
shows mono f ⇒ max (f m) (f n) = f (max m n)
by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

end

lemma inf-min: inf = (min :: 'a::{semilattice-inf, linorder} ⇒ 'a ⇒ 'a)
by (auto intro: antisym simp add: min-def fun-eq-iff)

lemma sup-max: sup = (max :: 'a::{semilattice-sup, linorder} ⇒ 'a ⇒ 'a)
by (auto intro: antisym simp add: max-def fun-eq-iff)

5.7 Uniqueness of inf and sup

lemma (in semilattice-inf) inf-unique:
fixes f (infixl △ 70)
assumes le1: ∀x y. x △ y ⊑ x and le2: ∀x y. x △ y ⊑ y
and greatest: ∀x y z. x ⊑ z ⇒ y ⊑ z ⇒ x △ y ⊑ z
shows x ⊓ y = x △ y
proof (rule antisym)
  show x △ y ⊑ x ⊓ y by (rule le-infI) (rule le1, rule le2)
next
  have leI: ∀x y z. x ⊑ z ⇒ y ⊑ z ⇒ x △ y ⊑ z by (blast intro: greatest)
  show x ⊓ y ⊑ x △ y by (rule leI) simp-all
qed

lemma (in semilattice-sup) sup-unique:
fixes f (infixl ∇ 70)
assumes ge1 [simp]: ∀x y. x ⊑ x ∇ y and ge2: ∀x y. y ⊑ x ∇ y
and least: ∀x y z. y ⊑ x ⇒ z ⊑ x ⇒ y ∇ z ⊑ x
shows x ⊔ y = x ∇ y
proof (rule antisym)
  show x ⊔ y ⊑ x ∇ y by (rule le-supI) (rule ge1, rule ge2)
next
  have leI: ∀x y z. x ⊑ z ⇒ y ⊑ z ⇒ x ∇ y ⊑ z by (blast intro: least)
  show x ∇ y ⊑ x ⊔ y by (rule leI) simp-all
qed

5.8 Lattice on bool

instantiation bool :: boolean-algebra
begin

definition bool-Compl-def [simp]: uminus = Not

definition bool-diff-def [simp]: A − B ↔ A ∧ ¬ B
THEORY “Lattices”

definition
[simp]: \( P \sqcap Q \iff P \land Q \)

definition
[simp]: \( P \sqcup Q \iff P \lor Q \)

instance proof
qed auto

end

lemma sup-boolI1:
\( P \implies P \sqcup Q \)
by simp

lemma sup-boolI2:
\( Q \implies P \sqcup Q \)
by simp

lemma sup-boolE:
\( P \sqcup Q \implies (P \implies R) \implies (Q \implies R) \implies R \)
by auto

5.9 Lattice on - \( \Rightarrow \) -

instantiation fun :: (type, semilattice-sup) semilattice-sup
begin

definition
\( f \sqcup g = (\lambda x. f x \sqcup g x) \)

lemma sup-apply [simp, code]:
\((f \sqcup g) x = f x \sqcup g x\)
by (simp add: sup-fun-def)

instance proof
qed (simp-all add: le-fun-def)

end

instantiation fun :: (type, semilattice-inf) semilattice-inf
begin

definition
\( f \sqcap g = (\lambda x. f x \sqcap g x) \)

lemma inf-apply [simp, code]:
\((f \sqcap g) x = f x \sqcap g x\)
5.10 Lattice on unary and binary predicates

**Lemma** `inf1I`: \( A x \implies B x \implies (A \cap B) x \)

*by (simp add: inf-fun-def)*
lemma inf2I: \( A \times y \Rightarrow B \times y \Rightarrow (A \cap B) \times y \) 
by (simp add: inf-fun-def)

lemma inf1E: \((A \cap B) \times \Rightarrow (A \times \Rightarrow B \times \Rightarrow P) \Rightarrow P \) 
by (simp add: inf-fun-def)

lemma inf2E: \((A \cap B) \times y \Rightarrow (A \times y \Rightarrow B \times y \Rightarrow P) \Rightarrow P \) 
by (simp add: inf-fun-def)

lemma inf1D1: \((A \cap B) \times \Rightarrow A \times \) 
by (rule inf1E)

lemma inf2D1: \((A \cap B) \times y \Rightarrow A \times y \) 
by (rule inf2E)

lemma inf1D2: \((A \cap B) \times \Rightarrow B \times \) 
by (rule inf1E)

lemma inf2D2: \((A \cap B) \times y \Rightarrow B \times y \) 
by (rule inf2E)

lemma sup1I1: \( A \times \Rightarrow (A \sqcup B) \times \) 
by (simp add: sup-fun-def)

lemma sup2I1: \( A \times y \Rightarrow (A \sqcup B) \times y \) 
by (simp add: sup-fun-def)

lemma sup1I2: \( B \times \Rightarrow (A \sqcup B) \times \) 
by (simp add: sup-fun-def)

lemma sup2I2: \( B \times y \Rightarrow (A \sqcup B) \times y \) 
by (simp add: sup-fun-def)

lemma sup1E: \( (A \sqcup B) \times \Rightarrow (A \times \Rightarrow P) \Rightarrow (B \times \Rightarrow P) \Rightarrow P \) 
by (simp add: sup-fun-def) iprover

lemma sup2E: \( (A \sqcup B) \times y \Rightarrow (A \times y \Rightarrow P) \Rightarrow (B \times y \Rightarrow P) \Rightarrow P \) 
by (simp add: sup-fun-def) iprover

Classical introduction rule: no commitment to \( A \) vs \( B \).

lemma sup1CI: \( \neg B \times \Rightarrow A \times \Rightarrow (A \sqcup B) \times \) 
by (auto simp add: sup-fun-def)

lemma sup2CI: \( \neg B \times y \Rightarrow A \times y \Rightarrow (A \sqcup B) \times y \) 
by (auto simp add: sup-fun-def)

no-notation
THEORY "Set"

less-eq (infix ⊑ 50) and
less (infix ⊏ 50)

end

6 Set: Set theory for higher-order logic

theory Set
imports Lattices
begin

6.1 Sets as predicates

typedcl 'a set

axiomatization Collect :: ('a ⇒ bool) ⇒ 'a set — comprehension
and member :: 'a ⇒ 'a set ⇒ bool — membership

where
mem-Collect-eq [iff, code-unfold]: member a (Collect P) = P a
and Collect-mem-eq [simp]: Collect (λx. member x A) = A

notation
member (op :)
and
member ((/- : -) [51, 51] 50)

abbreviation not-member where
not-member x A ≡ ~ (x : A) — non-membership

notation
not-member (op :~:)
and
not-member ((/- :~: -) [51, 51] 50)

notation (xsymbols)
member (op ∈)
and
member ((/- ∈ -) [51, 51] 50)
and
not-member (op ∉)
and
not-member ((/- ∉ -) [51, 51] 50)

notation (HTML output)
member (op ∈)
and
member ((/- ∈ -) [51, 51] 50)
and
not-member (op ∉)
and
not-member ((/- ∉ -) [51, 51] 50)

Set comprehensions

syntax
-Coll :: pttrn => bool => 'a set ((1{-/-}))

translations
{x. P} == CONST Collect (%x. P)
THEORY "Set"

syntax
-Collect :: pttrn => 'a set => bool => 'a set ((1{:/ -/- -}))
syntax (xsymbols)
-Collect :: pttrn => 'a set => bool => 'a set ((1{:/ -/- -}))
translations
{p:A. P} => CONST Collect (%p. p:A & P)

lemma CollectI: P a => a ∈ {x. P x}
  by simp

lemma CollectD: a ∈ {x. P x} => P a
  by simp

lemma Collect-cong: (\x. P x = Q x) => {x. P x} = {x. Q x}
  by simp

Simproc for pulling x=t in \{x . . . & x=t & . . .\} to the front (and similarly for t=x):

simproc-setup defined-Collect (\{x. P x & Q x\}) = ⟨⟨
  fn - => Quantifier1.rearrange-Collect
  (fn - =>
    rtac @{thm Collect-cong} 1 THEN
    rtac @{thm iffI} 1 THEN
    ALLGOALS (EVERY' [REPEAT-DETERM o etac @{thm conjE}, DEPTH-SOLVE-1 o
    ares-tac @{thms conjI}]))
⟩⟩

lemmas CollectE = CollectD [elim-format]

lemma set-eqI:
  assumes \x. x ∈ A <-> x ∈ B
  shows A = B
proof
  from assms have {x. x ∈ A} = {x. x ∈ B} by simp
  then show ?thesis by simp
qed

lemma set-eq iff:

A = B <-> (∀x. x ∈ A <-> x ∈ B)
by (auto intro: set-eqI)

Lifting of predicate class instances

instantiation set :: (type) boolean-algebra
begin

definition less-eq-set where
A ≤ B <-> (λx. member x A) ≤ (λx. member x B)


\textbf{definition \textit{less-set} where} \\
\textit{A} < B \iff (\lambda x. \text{member } x \ A) < (\lambda x. \text{member } x \ B)

\textbf{definition \textit{inf-set} where} \\
\textit{A} \cap \textit{B} = \text{Collect} ((\lambda x. \text{member } x \ A) \cap (\lambda x. \text{member } x \ B))

\textbf{definition \textit{sup-set} where} \\
\textit{A} \cup \textit{B} = \text{Collect} ((\lambda x. \text{member } x \ A) \cup (\lambda x. \text{member } x \ B))

\textbf{definition \textit{bot-set} where} \\
\bot = \text{Collect} \bot

\textbf{definition \textit{top-set} where} \\
\top = \text{Collect} \top

\textbf{definition \textit{uminus-set} where} \\
- \textit{A} = \text{Collect} (- (\lambda x. \text{member } x \ A))

\textbf{definition \textit{minus-set} where} \\
\textit{A} - \textit{B} = \text{Collect} ((\lambda x. \text{member } x \ A) - (\lambda x. \text{member } x \ B))

\textbf{instance proof} \\
\textbf{qed \ (simp-all add: less-eq-set-def less-set-def inf-set-def sup-set-def \ bot-set-def top-set-def uminus-set-def \ less-le-not-le inf-compl-bot sup-compl-top sup-inf-distrib1 diff-eq \ set-eqI fun-eq-iff \ del: \ inf-apply sup-apply bot-apply top-apply minus-apply uminus-apply)}

end

Set enumerations

\textbf{abbreviation \textit{empty} :: \textit{a set} \{\} \ where} \\
\{\} \equiv \text{bot}

\textbf{definition \textit{insert} :: \textit{a} \Rightarrow \textit{a set} \Rightarrow \textit{a set} \ where} \\
\textit{insert-compr: insert a B = \{x. x = a \lor x \in B\}}

\textbf{syntax} \\
-Finset :: args \Rightarrow \textit{a set} (\{\})

\textbf{translations} \\
\{x, xs\} == \text{CONST insert x \{xs\}} \\
\{x\} == \text{CONST insert x \{\}}

\textbf{6.2 Subsets and bounded quantifiers}

\textbf{abbreviation} \\
\textit{subset} :: \textit{a set} \Rightarrow \textit{a set} \Rightarrow \textit{bool} \ where \\
\textit{subset} \equiv \text{less}
THEORY "Set"

abbreviation
subset-eq :: 'a set ⇒ 'a set ⇒ bool where
subset-eq ≡ less-eq

notation (output)
subset (op <) and
subset ((/- < -) [51, 51] 50) and
subset-eq (op <=) and
subset-eq ((/- <= -) [51, 51] 50)

notation (xsymbols)
subset (op ⊂) and
subset ((/- ⊂ -) [51, 51] 50) and
subset-eq (op ⊆) and
subset-eq ((/- ⊆ -) [51, 51] 50)

notation (HTML output)
subset (op ⊂) and
subset ((/- ⊂ -) [51, 51] 50) and
subset-eq (op ⊆) and
subset-eq ((/- ⊆ -) [51, 51] 50)

abbreviation (input)
supset :: 'a set ⇒ 'a set ⇒ bool where
supset ≡ greater

abbreviation (input)
supset-eq :: 'a set ⇒ 'a set ⇒ bool where
supset-eq ≡ greater-eq

notation (xsymbols)
supset (op ⊃) and
supset ((/- ⊃ -) [51, 51] 50) and
supset-eq (op ⊇) and
supset-eq ((/- ⊇ -) [51, 51] 50)

definition Ball :: 'a set ⇒ ('a ⇒ bool) ⇒ bool where
Ball A P ←→ (∀ x. x ∈ A → P x) — bounded universal quantifiers

definition Bex :: 'a set ⇒ ('a ⇒ bool) ⇒ bool where
Bex A P ←→ (∃ x. x ∈ A ∧ P x) — bounded existential quantifiers

syntax
-Ball :: pttrn => 'a set => bool => bool ((3ALL ::/- ->) [0, 0, 10] 10)
-Bex :: pttrn => 'a set => bool => bool ((3EX ::/- ->) [0, 0, 10] 10)
-Bex1 :: pttrn => 'a set => bool => bool ((3EX! ::/- ->) [0, 0, 10] 10)
-Bleast :: id => 'a set => bool => 'a ((3LEAST ::/- ->) [0, 0, 10] 10)
THEORY "Set"

**syntax (HOL)**

-Ball :: ptrtn => 'a set => bool => bool ((3! -.-.-) [0, 0, 10] 10)
-Bex :: ptrtn => 'a set => bool => bool ((3? -.-.-) [0, 0, 10] 10)
-Bext :: ptrtn => 'a set => bool => bool ((3?? -.-.-) [0, 0, 10] 10)

**syntax (xsymbols)**

-Ball :: ptrtn => 'a set => bool => bool ((∀ε.-.-.-) [0, 0, 10] 10)
-Bex :: ptrtn => 'a set => bool => bool ((∃ε.-.-.-) [0, 0, 10] 10)
-Bext :: ptrtn => 'a set => bool => bool ((∃∃ε.-.-.-) [0, 0, 10] 10)

**translations**

ALL x:A. P == CONST Ball A (%x. P)
EX x:A. P == CONST Bex A (%x. P)
EX! x:A. P => EX! x. x:A & P
LEAST x:A. P => LEAST x. x:A & P

**syntax (output)**

-setlessAll :: [idt, 'a, bool] => bool ((3ALL -<-/-) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] => bool ((3EX -<-/-) [0, 0, 10] 10)
-setleAll :: [idt, 'a, bool] => bool ((3∃-<-/-) [0, 0, 10] 10)
-setleEx :: [idt, 'a, bool] => bool ((3∃EX -<-/-) [0, 0, 10] 10)
-setleEx1 :: [idt, 'a, bool] => bool ((3∃EX! -<-/-) [0, 0, 10] 10)

**syntax (xsymbols)**

-setlessAll :: [idt, 'a, bool] => bool ((∀ε.-<-/-) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] => bool ((∃ε.-<-/-) [0, 0, 10] 10)
-setleAll :: [idt, 'a, bool] => bool ((∃∃ε.-<-/-) [0, 0, 10] 10)
-setleEx :: [idt, 'a, bool] => bool ((∃∃EX -<-/-) [0, 0, 10] 10)
-setleEx1 :: [idt, 'a, bool] => bool ((∃∃EX! -<-/-) [0, 0, 10] 10)

**syntax (HOL output)**

-setlessAll :: [idt, 'a, bool] => bool ((3! -.-.-) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] => bool ((3? -.-.-) [0, 0, 10] 10)
-setleAll :: [idt, 'a, bool] => bool ((3? -<-/-) [0, 0, 10] 10)
-setleEx :: [idt, 'a, bool] => bool ((3∃ -.-.-) [0, 0, 10] 10)
-setleEx1 :: [idt, 'a, bool] => bool ((3∃! -.-.-) [0, 0, 10] 10)

**syntax (HTML output)**

-setlessAll :: [idt, 'a, bool] => bool ((3! -.-.-) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] => bool ((3? -.-.-) [0, 0, 10] 10)
-setleAll :: [idt, 'a, bool] => bool ((3∃ -.-.-) [0, 0, 10] 10)
THEORY "Set"

-val All-binder = Mixfix.binder-name @{const-syntax All};
-val Ex-binder = Mixfix.binder-name @{const-syntax Ex};
-val conj = @{const-syntax HOL.conj};
-val sbset = @{const-syntax subset};
-val sbset-eq = @{const-syntax subset-eq};
-val trans =
  [((All-binder, conj, sbset), @{syntax-const -setlessAll}),
   ((All-binder, impl, sbset-eq), @{syntax-const -setleAll}),
   ((Ex-binder, conj, sbset), @{syntax-const -setlessEx}),
   ((Ex-binder, conj, sbset-eq), @{syntax-const -setleEx})];

-fun mk v (v', T) c n P =
  if v = v' andalso not (Term.exists-subterm (fn Free (x, _) => x = v | - => false) n)
  then Syntax.const c $ Syntax.Trans.mark-bound-body (v', T) $ n $ P
  else raise Match;

-fun tr' q = (q, fn - =>
  (fn | Const (@{syntax-const -bound}, -) $ Free (v, Type (@{type-name set}, -)),
       Const (c, -) $
       (Const (d, -) $ (Const (@{syntax-const -bound}), -) $ Free (v', T)) $ n) $ P) =>
  (case ALIST.lookup (op =) trans (q, c, d) of
    NONE => raise Match
  | SOME l => mk v (v', T) l n P
  | - => raise Match));

end

]]

Translate between \{e | x1...xn. P\} and \{u. EX x1..xn. u = e & P\}; \{y. EX x1..xn. y = e & P\} is only translated if \{0..n\} subset bvs(e).

syntax
THEORY "Set"

parse-translation ⟨⟨
let
val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX, @{const-syntax Ex}));

fun nvars (Const (@{syntax-const -idts}, -) $ - $ idts) = nvars idts + 1
| nvars = 1;

fun setcompr-tr ctxt [e, idts, b] =
  let
  val eq = Syntax.const @{const-syntax HOL.eq} $ Bound (nvars idts) $ e;
  val P = Syntax.const @{const-syntax HOL.conj} $ eq $ b;
  val exP = ex-tr ctxt [idts, P];
  in Syntax.const @{const-syntax Collect} $ absdummy dummyT exP end;
   
in [(@{syntax-const -Setcompr}, setcompr-tr)] end;
⟩⟩

print-translation ⟨⟨
[Syntax-Trans.preserve-binder-abs2-tr' @{const-syntax Ball} @{syntax-const -Ball},
 Syntax-Trans.preserve-binder-abs2-tr' @{const-syntax Bex} @{syntax-const -Bex}]
⟩⟩ — to avoid eta-contraction of body

print-translation ⟨⟨
let
val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (@{const-syntax Ex}, DUMMY));

fun setcompr-tr' ctxt [Abs (abs as (-, -), P)] =
  let
  fun check (Const (@{const-syntax Ex}, -) $ Abs (-, -), n) = check (P, n + 1)
  | check (Const (@{const-syntax HOL.conj}, -) $ (Const (@{const-syntax HOL.eq}, -) $ Bound m $ e) $ P, n) =
    n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
    subset (op =) (0 upto (n - 1), add-loose-bnos (e, 0, []))
  | check = false;

  fun tr' (- $ abs) =
    let val - $ idts $ (- $ (- $ - $ e) $ Q) = ex-tr' ctxt [abs]
    in Syntax.const @{syntax-const -Setcompr} $ e $ idts $ Q end;
  in
  if check (P, 0) then tr' P
  else let
    val (x as - $ Free(xN, -), t) = Syntax-Trans.atomic-abs-tr' abs;
    val M = Syntax.const @{syntax-const -Coll} $ x $ t;
    in
    case t of
    …
THEORY "Set"

Const (@\{const-syntax HOL.conv\}, -) $158$
(Const (@\{const-syntax Set.member\}, -) $158$
(Const (@\{syntax-const -bound\}, -) $158$ Free (yN, -)) $158$ A) $158$ P =>
if xN = yN then Syntax.const @\{syntax-const -Collect\} $158$ x $158$ A $158$ P else
M
| - => M
end;
in (@\{const-syntax Collect\}, setcompr-tr') end;

simproc-setup defined-Bex (EX x:A. P x & Q x) = ⟨⟨
fn - => Quantifier1.rearrange-bex
(fn ctxt =>
unfold-tac ctxt @\{thms Bex-def\} THEN
Quantifier1.prove-one-point-ex-tac)
⟩⟩

simproc-setup defined-All (ALL x:A. P x --> Q x) = ⟨⟨
fn - => Quantifier1.rearrange-ball
(fn ctxt =>
unfold-tac ctxt @\{thms Ball-def\} THEN
Quantifier1.prove-one-point-all-tac)
⟩⟩

lemma ballI [intro!]: (!!x. x:A ===> P x) ===> ALL x:A. P x
by (simp add: Ball-def)

lemmas strip = impI allI ballI

lemma bspec [dest?!]: ALL x:A. P x ===> x:A ===> P x
by (simp add: Ball-def)

Gives better instantiation for bound:

setup ⟨⟨
map-theory-claset (fn ctxt =>
ctxt addbefore (bspec, fn - => dtac @\{thm bspec\} THEN' assume-tac))
⟩⟩

ML ⟨⟨
structure Simpdata =
struct
open Simpdata;
val mksimps-pairs = [(@\{const-name Ball\}, @\{thms bspec\})] @ mksimps-pairs;
end;
⟩⟩
open Simpdata;

declaration ⟨.fn -⟩
  Simplifier.map-ss (Simplifier.set-mksimps (mksimps mksimps-pairs))
⟩⟩

lemma ballE [elim]: ALL x:A. P x ==⇒ (P x ==⇒ Q) ==⇒ (x : A ==⇒ Q)
  by (unfold Ball-def) blast

lemma bexI [intro]: P x ==⇒ x:A ==⇒ EX x:A. P x
  — Normally the best argument order: P x constrains the choice of x ∈ A.
  by (unfold Bex-def) blast

lemma rev-bexI [intro?!]: x:A ==⇒ P x ==⇒ EX x:A. P x
  — The best argument order when there is only one x ∈ A.
  by (unfold Bex-def) blast

lemma bexCI: (ALL x:A. ~P x ==⇒ P a) ==⇒ a:A ==⇒ EX x:A. P x
  by (unfold Bex-def) blast

lemma ball-triv [simp]: (ALL x:A. P) ==⇒ ((EX x:A) ==⇒ P)
  — Trival rewrite rule.
  by (simp add: Ball-def)

lemma bex-triv [simp]: (EX x:A. P) ==⇒ ((EX x:A) & P)
  — Dual form for existentials.
  by (simp add: Bex-def)

lemma bex-triv-one-point1 [simp]: (EX x:A. x = a) ==⇒ (a:A)
  by blast

lemma bex-triv-one-point2 [simp]: (EX x:A. a = x) ==⇒ (a:A)
  by blast

lemma bex-one-point1 [simp]: (EX x:A. x = a & P x) ==⇒ (a:A & P a)
  by blast

lemma bex-one-point2 [simp]: (EX x:A. a = x & P x) ==⇒ (a:A & P a)
  by blast

lemma ball-one-point1 [simp]: (ALL x:A. x = a ==⇒ P x) ==⇒ (a:A ==⇒ P a)
  by blast

lemma ball-one-point2 [simp]: (ALL x:A. a = x ==⇒ P x) ==⇒ (a:A ==⇒ P a)
THEORY "Set"

by blast

lemma ball-conj-distrib:
(∀ x∈A. P x ∧ Q x) ←→ ((∀ x∈A. P x) ∧ (∀ x∈A. Q x))
by blast

lemma bex-disj-distrib:
(∃ x∈A. P x ∨ Q x) ←→ ((∃ x∈A. P x) ∨ (∃ x∈A. Q x))
by blast

Congruence rules

lemma ball-cong:
A = B ==> (!!x. x:B ==> P x = Q x) ==>
(∀ x:A. P x) = (∀ x:B. Q x)
by (simp add: Ball-def)

lemma strong-ball-cong [cong]:
A = B ==> (!!x. x:B =simp=> P x = Q x) ==>
(ALL x:A. P x) = (ALL x:B. Q x)
by (simp add: simp-implies-def Ball-def)

lemma bex-cong:
A = B ==> (!!x. x:B =simp=> P x = Q x) ==>
(EX x:A. P x) = (EX x:B. Q x)
by (simp add: Bex-def cong: conj-cong)

lemma strong-bex-cong [cong]:
A = B ==> (!!x. x:B =simp=> P x = Q x) ==>
(EX x:A. P x) = (EX x:B. Q x)
by (simp add: simp-implies-def Bex-def cong: conj-cong)

6.3 Basic operations

6.3.1 Subsets

lemma subsetI [intro!]: (∀ x∈A. x ∈ A ==> x ∈ B) ==> A ⊆ B
by (simp add: less-eq-set-def le-fun-def)

Map the type 'a set => anything to just 'a; for overloading constants whose
first argument has type 'a set.

lemma subsetD [elim, intro?]: A ⊆ B ==> c ∈ A ==> c ∈ B
by (simp add: less-eq-set-def le-fun-def)
— Rule in Modus Ponens style.

lemma rev-subsetD [intro?]: c ∈ A ==> A ⊆ B ==> c ∈ B
— The same, with reversed premises for use with erule – cf rev-mp.
by (rule subsetD)

Converts A ⊆ B to x ∈ A ==> x ∈ B.
THEORY “Set”

lemma subsetCE [elim]: \A \subseteq \B \Longrightarrow (c \notin \A \Longrightarrow P) \Longrightarrow (c \in \B \Longrightarrow P) 
— Classical elimination rule.
  by (auto simp add: less-eq-set-def le-fun-def)

lemma subset-eq: \A \leq \B = (\forall x \in \A. x \in \B) by blast

lemma contra-subsetD: \A \subseteq \B \Longrightarrow c \notin \B \Longrightarrow c \notin \A
  by blast

lemma subset-refl: \A \subseteq \A
  by (fact order-refl)

lemma subset-trans: \A \subseteq \B \Longrightarrow \B \subseteq \C \Longrightarrow \A \subseteq \C
  by (fact order-trans)

lemma set-rev-mp: \_\A \Longrightarrow \A \subseteq \B \Longrightarrow x : \B
  by (rule subsetD)

lemma set-mp: \A \subseteq \B \Longrightarrow x : \A \Longrightarrow x : \B
  by (rule subsetD)

lemma subset-not-subset-eq [code]:
\_\A \subseteq \B \iff \A \subseteq \B \land \neg \B \subseteq \A
  by (fact less-le-not-le)

lemma eq-mem-trans: a = b \Longrightarrow b \in \A \Longrightarrow a \in \A
  by simp

lemmas basic-trans-rules [trans] =
  order-trans-rules set-rev-mp set-mp eq-mem-trans

6.3.2 Equality

lemma subset-antisym [intro!]: \A \subseteq \B \Longrightarrow \B \subseteq \A \Longrightarrow \A = \B
— Anti-symmetry of the subset relation.
  by (iprover intro: set-eql subsetD)

Equality rules from ZF set theory – are they appropriate here?

lemma equalityD1: \_\A = \B \Longrightarrow \A \subseteq \B
  by simp

lemma equalityD2: \_\A = \B \Longrightarrow \B \subseteq \A
  by simp

Be careful when adding this to the claset as subset-empty is in the simpset:
\A = \{} goes to \{} \subseteq \A and \A \subseteq \{} and then back to \A = \{}!

lemma equalityE: \_\A = \B \Longrightarrow (\A \subseteq \B \Longrightarrow \B \subseteq \A \Longrightarrow P) \Longrightarrow P
by simp

lemma equalityCE [elim]:
\[ A = B \Longrightarrow (c \in A \Longrightarrow c \in B \Longrightarrow P) \Longrightarrow (c \notin A \Longrightarrow c \notin B \Longrightarrow P) \]
\Longrightarrow P
by blast

lemma eqset-imp-iff: \( A = B \Longrightarrow (x : A) = (x : B) \)
by simp

lemma eqelem-imp-iff: \( x = y \Longrightarrow (x : A) = (y : A) \)
by simp

6.3.3 The empty set

lemma empty-def:
\{\} = \{x. \text{False}\}
by (simp add: bot-set-def bot-fun-def)

lemma empty-iff [simp]: \( c : \{\} \) = False
by (simp add: empty-def)

lemma emptyE [elim!]: \( a : \{\} \Longrightarrow P \)
by simp

lemma empty-subsetI [iff]: \{\} \subseteq A
— One effect is to delete the ASSUMPTION \{\} \subseteq A
by blast

lemma equals0I: (!! y. y \in A \Longrightarrow False) \Longrightarrow A = \{\}
by blast

lemma equals0D: A = \{\} \Longrightarrow a \notin A
— Use for reasoning about disjointness: \( A \cap B = \{\} \)
by blast

lemma ball-empty [simp]: Ball \{\} P = True
by (simp add: Ball-def)

lemma bex-empty [simp]: Bex \{\} P = False
by (simp add: Bex-def)

6.3.4 The universal set – UNIV

abbreviation UNIV :: 'a set where
\( UNIV \equiv \text{top} \)

lemma UNIV-def:
\( UNIV = \{x. \text{True}\} \)
by (simp add: top-set-def top-fun-def)
lemma \text{UNIV-I} [simp]: \( x : \text{UNIV} \)
by (simp add: UNIV-def)

\text{declare \text{UNIV-I} [intro]} — unsafe makes it less likely to cause problems

lemma \text{UNIV-witness} [intro?]: \( \exists x. \ x : \text{UNIV} \)
by simp

\text{lemma \text{subset-UNIV}:} A \subseteq \text{UNIV}
by (fact \text{top-greatest})

Eta-contracting these two rules (to remove \( P \)) causes them to be ignored because of their interaction with congruence rules.

lemma \text{ball-UNIV} [simp]: \( \text{Ball UNIV } P = \text{All } P \)
by (simp add: Ball-def)

lemma \text{bex-UNIV} [simp]: \( \text{Bex UNIV } P = \text{Ex } P \)
by (simp add: Bex-def)

lemma \text{UNIV-eq-I}: \( \forall x. \ x \in A \) \implies \text{UNIV} = A
by auto

lemma \text{UNIV-not-empty} [iff]: \text{UNIV} \neq \{\}
by (blast elim: equalityE)

lemma \text{empty-not-UNIV} [simp]: \( \{\} \neq \text{UNIV} \)
by blast

6.3.5 The Powerset operator – \text{Pow}

definition \text{Pow} :: 'a set => 'a set set where
\text{Pow-def:} \text{Pow } A = \{ B. \ B \subseteq A \}

lemma \text{Pow-Iff} [iff]: \( A \in \text{Pow } B \) = (A \subseteq B)
by (simp add: Pow-def)

lemma \text{PowD}: A \subseteq B \Longrightarrow A \in \text{Pow } B
by (simp add: Pow-def)

lemma \text{Pow-bottom}: \( \{\} \in \text{Pow } B \)
by simp

lemma \text{Pow-top}: A \in \text{Pow } A
by simp
THEORY “Set”

lemma Pow-not-empty: Pow A ≠ {} using Pow-top by blast

6.3.6 Set complement

lemma Compl-iff [simp]: (c ∈ −A) = (c ∉ A)
  by (simp add: fun-Compl-def uminus-set-def)

lemma ComplI [intro!]: (c ∈ A ==> False) ==> c ∈ −A
  by (simp add: fun-Compl-def uminus-set-def) blast

This form, with negated conclusion, works well with the Classical prover.
Negated assumptions behave like formulae on the right side of the notional turnstile ...

lemma ComplD [dest!]: c : −A ==> c ~: A
  by simp

lemmas ComplE = ComplD [elim-format]

lemma Compl-eq: −A = {x. ∼ x : A}
  by blast

6.3.7 Binary intersection

abbreviation inter :: 'a set ⇒ 'a set ⇒ 'a set (infixl Int 70) where
  op Int ≡ inf

notation (xsymbols)
  inter (infixl ∩ 70)

notation (HTML output)
  inter (infixl ∩ 70)

lemma Int-def:
  A ∩ B = {x. x ∈ A ∧ x ∈ B}
  by (simp add: inf-set-def inf-fun-def)

lemma Int-iff [simp]: (c : A Int B) = (c:A & c:B)
  by (unfold Int-def) blast

lemma IntI [intro!]: c:A ==> c:B ==> c : A Int B
  by simp

lemma IntD1: c : A Int B ==> c:A
  by simp

lemma IntD2: c : A Int B ==> c:B
  by simp
lemma \texttt{IntE} [elim!]: \(c : A \text{ Int } B \Longrightarrow (c:A \Longrightarrow c:B \Longrightarrow P) \Longrightarrow P\)

by \texttt{simp}

lemma \texttt{mono-Int}: \(\text{mono } f \Rightarrow f \ (A \cap B) \subseteq f \ A \cap f \ B\)

by \texttt{(fact mono-inf)}

6.3.8 Binary union

abbreviation \texttt{union} :: 'a set ⇒ 'a set ⇒ 'a set (infixl Un 65) where

\texttt{union} ≡ \texttt{sup}

notation \texttt{(xsymbols)}

\texttt{union} (infixl ∪ 65)

notation \texttt{(HTML output)}

\texttt{union} (infixl ∪ 65)

lemma \texttt{Un-def}:

\(A \cup B = \{x. \ x \in A \lor x \in B\}\)

by (simp add: sup-set-def sup-fun-def)

lemma \texttt{Un-iff} [simp]: \((c : A \cup B) = (c:A \mid c:B)\)

by (unfold Un-def) blast

lemma \texttt{UnI1} [elim?]: \(c : A \Longrightarrow c : A \cup B\)

by \texttt{simp}

lemma \texttt{UnI2} [elim?]: \(c : B \Longrightarrow c : A \cup B\)

by \texttt{simp}

Classical introduction rule: no commitment to \(A \) vs \(B\).

lemma \texttt{UnCI} [intro!]: \((c:\sim:B \Longrightarrow c:A) \Longrightarrow c : A \cup B\)

by \texttt{auto}

lemma \texttt{UnE} [elim!]: \(c : A \cup B \Longrightarrow (c:A \Longrightarrow P) \Longrightarrow (c:B \Longrightarrow P) \Longrightarrow P\)

by (unfold Un-def) blast

lemma \texttt{insert-def}: \texttt{insert} a B = \{x. x = a\} \cup B

by (simp add: insert-compr Un-def)

lemma \texttt{mono-Un}: \(\text{mono } f \Rightarrow f \ A \cup f \ B \subseteq f \ (A \cup B)\)

by \texttt{(fact mono-sup)}

6.3.9 Set difference

lemma \texttt{Diff-iff} [simp]: \((c : A - B) = (c:A \& c:\sim:B)\)

by (simp add: minus-set-def fun-diff-def)
lemma DiffI [intro!]: \( c : A \Rightarrow c : A - B \)
   by simp

lemma DiffD1: \( c : A - B \Rightarrow c : A \)
   by simp

lemma DiffD2: \( c : A - B \Rightarrow c : B \Rightarrow P \)
   by simp

lemma DiffE [elim!]: \( c : A - B \Rightarrow (c:A \Rightarrow c\sim:B \Rightarrow P) \Rightarrow P \)
   by simp

lemma set-diff-eq: \( A - B = \{ x \mid x : A \& \sim x : B \} \)
   by blast

lemma Compl-eq-Diff-UNIV: \( -A = (UNIV - A) \)
   by blast

6.3.10 Augmenting a set – insert

lemma insert-iff [simp]: \( (a : insert b A) = (a = b \mid a:A) \)
   by (unfold insert-def) blast

lemma insertI1: a : insert a B
   by simp

lemma insertI2: a : B \Rightarrow a : insert b B
   by simp

lemma insertE [elim!]: a : insert b A \Rightarrow (a = b \Rightarrow P) \Rightarrow a : insert b B
   \Rightarrow P
   by (unfold insert-def) blast

lemma insertCI [intro!]: \( a\sim:B \Rightarrow a = b \Rightarrow a : insert b B \)
   — Classical introduction rule.
   by auto

lemma subset-insert-iff: \( A \subseteq insert x B \Rightarrow (if x:A then A - \{x\} \subseteq B else A \subseteq B) \)
   by auto

lemma set-insert:
   assumes x \in A
   obtains B where A = insert x B and x \notin B
proof
   from assms show A = insert x (A - \{x\}) by blast
next
   show x \notin A - \{x\} by blast
qed
Lemma insert-ident: \( x \sim A \implies x \sim B \implies (\text{insert } x A = \text{insert } x B) = (A = B) \)

by auto

Lemma insert-eq-iff: assumes \( a \notin A, b \notin B \)

shows \( \text{insert } a A = \text{insert } b B \iff (\text{if } a = b \text{ then } A = B \text{ else } \exists C. A = \text{insert } b C \land b \notin C \land B = \text{insert } a C \land a \notin C) \)

(is \( ?L \iff ?R \))

proof

assume \( ?L \)

show \( ?R \)

proof cases

assume \( a = b \) with assms \( (?L) \)

show \( ?R \) by (simp add: insert-ident)

next

assume \( a \neq b \)

let \( ?C = A - \{b\} \)

have \( A = \text{insert } b ?C \land b \notin ?C \land B = \text{insert } a ?C \land a \notin ?C \)

using assms \( (?L) \) \( (a \neq b) \) by auto

thus \( ?R \) using \( (a \neq b) \) by auto

qed

next

assume \( ?R \) thus \( ?L \) by (auto split: if-splits)

qed

6.3.11 Singletons, using insert

Lemma singletonI [intro!]: \( a : \{a\} \)

— Redundant? But unlike insertCI, it proves the subgoal immediately!

by (rule insertI1)

Lemma singletonD [dest!]: \( b : \{a\} \implies b = a \)

by blast

Lemmas singletonE = singletonD [elim-format]

Lemma singleton-if: \( b : \{a\} \implies (b = a) \)

by blast

Lemma singleton-inject [dest!]: \( \{a\} = \{b\} \implies a = b \)

by blast

Lemma singleton-insert-inj-eq [iff]:

\( \{b\} = \text{insert } a A \implies (a = b \land A \subseteq \{b\}) \)

by blast

Lemma singleton-insert-inj-eq' [iff]:

\( \text{insert } a A = \{b\} \implies (a = b \land A \subseteq \{b\}) \)

by blast
lemma subset-singletonD: \( A \subseteq \{x\} \Longrightarrow A = \{\} \mid A = \{x\} \)
by fast

lemma singleton-conv [simp]: \( \{.x = a\} = \{a\} \)
by blast

lemma singleton-conv2 [simp]: \( \{.a = x\} = \{a\} \)
by blast

lemma diff-single-insert: \( A - \{x\} \subseteq B \Longrightarrow A \subseteq \text{insert } x \ B \)
by blast

lemma doubleton-eq-iff: \((\{a,b\} = \{c,d\}) = (a=c \& b=d \mid a=d \& b=c)\)
by (blast elim: equalityE)

lemma Un-singleton-iff:
\((A \cup B = \{x\}) = (A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\}))\)
by auto

lemma singleton-Un-iff:
\((\{x\} = A \cup B) = (A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\}))\)
by auto

6.3.12 Image of a set under a function

Frequently \( b \) does not have the syntactic form of \( f x \).

definition image :: \(\{a \Rightarrow 'b\} \Rightarrow 'a \ set \Rightarrow 'b \ set\) (infixr \('90\))
where
\( f \cdot A = \{.y . \exists x \in A. y = f x\} \)

lemma image-eqI [simp, intro]:
\( b = f x \Longrightarrow x \in A \Longrightarrow b \in f \cdot A \)
by (unfold image-def) blast

lemma imageI:
\( x \in A \Longrightarrow f x \in f \cdot A \)
by (rule image-eqI) (rule refl)

lemma rev-image-eqI:
\( x \in A \Longrightarrow b = f x \Longrightarrow b \in f \cdot A \)
— This version’s more effective when we already have the required \( x \).
by (rule image-eqI)

lemma imageE [elim!]:
assumes \( b \in (\lambda x. f x) \cdot A \) — The eta-expansion gives variable-name preservation.
obtains $x$ where $b = f\ x$ and $x \in A$
using assms by (unfold image-def) blast

lemma Compr-image-eq:
$\{x \in f\ A. P\ x\} = f\ \{x \in A. P\ (f\ x)\}$
by auto

lemma image-Un:
$f\ (A \cup B) = f\ A \cup f\ B$
by blast

lemma image-iff:
$z \in f\ A \iff (\exists x \in A. z = f\ x)$
by blast

lemma image-subsetI:
$(\forall x. x \in A \implies f\ x \in B) \implies f\ A \subseteq B$
— Replaces the three steps subsetI, imageE, hypsubst, but breaks too many existing proofs.
by blast

lemma image-subset-iff:
$f\ A \subseteq B \iff (\forall x \in A. f\ x \in B)$
— This rewrite rule would confuse users if made default.
by blast

lemma subset-imageE:
assumes $B \subseteq f\ A$
obtains $C$ where $C \subseteq A$ and $B = f\ C$
proof —
from assms have $B = f\ \{a \in A. f\ a \in B\}$ by fast
moreover have $\{a \in A. f\ a \in B\} \subseteq A$ by blast
ultimately show thesis by (blast intro: that)
qed

lemma subset-image-iff:
$B \subseteq f\ A \iff (\exists AA \subseteq A. B = f\ AA)$
by (blast elim: subset-imageE)

lemma image-ident [simp]:
$(\lambda x. x)\ Y = Y$
by blast

lemma image-empty [simp]:
$f\ \{\}\ = \{\}$
by blast

lemma image-insert [simp]:
$f\ \text{insert } a\ B = \text{insert } (f\ a)\ (f\ B)$
**THEORY “Set”**

by blast

**lemma image-constant:**

\[ x \in A \implies (\lambda x. \, c) \cdot A = \{ c \} \]

by auto

**lemma image-constant-conv:**

\[ (\lambda x. \, c) \cdot A = (\text{if } A = \{ \} \text{ then } \{ \} \text{ else } \{ c \}) \]

by auto

**lemma image-image:**

\[ f \cdot (g \cdot A) = (\lambda x. \, f \cdot (g \cdot x)) \cdot A \]

by blast

**lemma insert-image [simp]:**

\[ x \in A \implies \text{insert} \cdot (f \cdot x) \cdot (f \cdot A) = f \cdot A \]

by blast

**lemma image-is-empty [iff]:**

\[ f \cdot A = \{ \} \iff A = \{ \} \]

by blast

**lemma empty-is-image [iff]:**

\[ \{ \} = f \cdot A \iff A = \{ \} \]

by blast

**lemma image-Collect:**

\[ f \cdot \{ x. \, P \cdot x \} = \{ f \cdot x \mid x \cdot P \cdot x \} \]

— NOT suitable as a default simprule: the RHS isn’t simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.

by blast

**lemma if-image-distrib [simp]:**

\[ (\lambda x. \, \text{if } P \cdot x \text{ then } f \cdot x \text{ else } g \cdot x) \cdot S = (f \cdot (S \cap \{ x. \, P \cdot x \}) \cup (g \cdot (S \cap \{ x. \, \neg P \cdot x \})) \]

by auto

**lemma image-cong:**

\[ M = N \implies (\forall x. \, x \in N \implies f \cdot x = g \cdot x) \implies f \cdot M = g \cdot N \]

by (simp add: image-def)

**lemma image-Int-subset:**

\[ f \cdot (A \cap B) \subseteq f \cdot A \cap f \cdot B \]

by blast

**lemma image-diff-subset:**

\[ f \cdot A - f \cdot B \subseteq f \cdot (A - B) \]

by blast
lemma ball-imageD:
  assumes \( \forall x \in f \cdot A \cdot P x \)
  shows \( \forall x \in A \cdot P (f x) \)
  using assms by simp

lemma bex-imageD:
  assumes \( \exists x \in f \cdot A \cdot P x \)
  shows \( \exists x \in A \cdot P (f x) \)
  using assms by auto

Range of a function – just a translation for image!

abbreviation range :: \((\cdot a \Rightarrow \cdot b) \Rightarrow \cdot b \) set
  where — of function
  range f \equiv f \cdot \text{UNIV}

lemma range-eqI:
  \( b = f x \Longrightarrow b \in \text{range } f \)
  by simp

lemma rangeI:
  \( f x \in \text{range } f \)
  by simp

lemma rangeE [elim?]:
  \( b \in \text{range } (\lambda x. f x) \Longrightarrow (\forall x. b = f x \Longrightarrow P) \Longrightarrow P \)
  by (rule imageE)

lemma full-SetCompr-eq:
  \( \{ u. \exists x. u = f x \} = \text{range } f \)
  by auto

lemma range-composition:
  \( \text{range } (\lambda x. f (g x)) = f \cdot \text{range } g \)
  by auto

6.3.13 Some rules with if

Elimination of \( \{ x. \ldots & x=t & \ldots \} \).

lemma Collect-conv-if: \( \{ x. x=a & P x \} = (if P a then \{ a \} else \{ \}) \)
  by auto

lemma Collect-conv-if2: \( \{ x. a=x & P x \} = (if P a then \{ a \} else \{ \}) \)
  by auto

Rewrite rules for boolean case-splitting: faster than split-if [split].

lemma split-if-eq1: \( ((if Q then x else y) = b) = ((Q \rightarrow x = b) & (\sim Q \rightarrow y = b)) \)
by (rule split-if)

lemma split-if-eq2: (a = (if Q then x else y)) = ((Q --> a = x) & (~ Q --> a = y))
  by (rule split-if)

Split ifs on either side of the membership relation. Not for [simp] – can cause goals to blow up!

lemma split-if-mem1: ((if Q then x else y) : b) = ((Q --> x : b) & (~ Q --> y : b))
  by (rule split-if)

lemma split-if-mem2: (a : (if Q then x else y)) = ((Q --> a : x) & (~ Q --> a : y))
  by (rule split-if [where \( P = S \). a : S])

lemmas split-ifs = if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2

6.4 Further operations and lemmas

6.4.1 The “proper subset” relation

lemma psubsetI [intro!]: \( A \subseteq B \implies A \neq B \implies A \subset B \)
  by (unfold less-le) blast

lemma psubsetE [elim!]:
  \( [[[A \subset B]; [A \subseteq B]; \sim (B \subseteq A)]]] \implies R \implies R \)
  by (unfold less-le) blast

lemma psubset-insert-iff:
  (A \subseteq insert x B) = (if x \in B then A \subseteq B else if x \in A then A \setminus \{x\} \subseteq B else A \subseteq B)
  by (auto simp add: less-le subset-insert-iff)

lemma psubset-eq: (A \subseteq B) = (A \subseteq B & A \neq B)
  by (simp only: less-le)

lemma psubset-imp-subset: A \subseteq B \implies A \subseteq B
  by (simp add: psubset-eq)

lemma psubset-trans: \( [[[A \subseteq B]; [B \subseteq C]]] \implies A \subseteq C \)
apply (unfold less-le)
apply (auto dest: subset-antisym)
done

lemma psubsetD: \( [[[A \subseteq B]; [c \in A]] \implies c \in B \)
apply (unfold less-le)
apply (auto dest: subsetD)
done


lemma psubset-subset-trans: \( A \subset B \implies B \subseteq C \implies A \subset C \)
by (auto simp add: psubset-eq)

lemma subset-psubset-trans: \( A \subseteq B \implies B \subset C \implies A \subset C \)
by (auto simp add: psubset-eq)

lemma psubset-imp-ex-mem: \( A \subset B \implies \exists b.\ b \in (B - A) \)
by (unfold less-le blast)

lemma atomize-ball:
(\( \forall x.\ x \in A \implies P x \) ) \( \equiv \) Trueprop (\( \forall x \in A.\ P x \) )
by (simp only: Ball-def atomize-all atomize-imp)

lemmas [symmetric, rulify] = atomize-ball
and [symmetric, defn] = atomize-ball

lemma image-Pow-mono:
assumes \( f ^ ' A \subseteq B \) 
shows image f ^ ' Pow A \( \subseteq \) Pow B
using assms by blast

lemma image-Pow-surj:
assumes \( f ^ ' A = B \) 
shows image f ^ ' Pow A = Pow B
using assms by (blast elim: subset-imageE)

6.4.2 Derived rules involving subsets.

insert.

lemma subset-insertI: \( B \subseteq \operatorname{insert} a B \)
by (rule subsetI) (erule insertI2)

lemma subset-insertI2: \( A \subseteq B \implies A \subseteq \operatorname{insert} b B \)
by blast

lemma subset-insert: \( x \notin A \implies (A \subseteq \operatorname{insert} x B) = (A \subseteq B) \)
by blast

Finite Union – the least upper bound of two sets.

lemma Un-upper1: \( A \subseteq A \cup B \)
by (fact sup-le1)

lemma Un-upper2: \( B \subseteq A \cup B \)
by (fact sup-le2)

lemma Un-least: \( A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C \)
by (fact sup-least)
Finite Intersection – the greatest lower bound of two sets.

**lemma** `Int-lower1`: \( A \cap B \subseteq A \)
  
  **by** (fact `inf-le1`)

**lemma** `Int-lower2`: \( A \cap B \subseteq B \)
  
  **by** (fact `inf-le2`)

**lemma** `Int-greatest`: \( C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B \)
  
  **by** (fact `inf-greatest`)

Set difference.

**lemma** `Diff-subset`: \( A - B \subseteq A \)
  
  **by** blast

**lemma** `Diff-subset-conv`: \( (A - B \subseteq C) = (A \subseteq B \cup C) \)
  
  **by** blast

### 6.4.3 Equalities involving union, intersection, inclusion, etc.

\{
\}

**lemma** `Collect-const [simp]`: \( \{s. \, P\} = (if \, P \, then \, UNIV \, else \, \{\}) \)
  
  — supersedes `Collect-False-empty`
  
  **by** auto

**lemma** `subset-empty [simp]`: \( (A \subseteq \{\}) = (A = \{\}) \)
  
  **by** (fact `bot-unique`)

**lemma** `not-psubset-empty [iff]`: \( \neg \, (A \subset \{\}) \)
  
  **by** (fact `not-less-bot`)

**lemma** `Collect-empty-eq [simp]`: \( (Collect \, P \, = \, \{\}) = (\forall \, x. \, \neg \, P \, x) \)
  
  **by** blast

**lemma** `empty-Collect-eq [simp]`: \( \{\} = Collect \, P \)
  
  **by** blast

**lemma** `Collect-neg-eq`: \( \{x. \, \neg \, P \, x\} = \neg \{x. \, P \, x\} \)
  
  **by** blast

**lemma** `Collect-disj-eq`: \( \{x. \, P \, x \mid Q \, x\} = \{x. \, P \, x\} \cup \{x. \, Q \, x\} \)
  
  **by** blast

**lemma** `Collect-imp-eq`: \( \{x. \, P \, x \Longrightarrow Q \, x\} = \neg \{x. \, P \, x\} \cup \{x. \, Q \, x\} \)
  
  **by** blast

**lemma** `Collect-conj-eq`: \( \{x. \, P \, x \& \, Q \, x\} = \{x. \, P \, x\} \cap \{x. \, Q \, x\} \)
  
  **by** blast
insert.

**lemma** insert-is-Un: insert a A = \{a\} Un A  
— NOT SUITABLE FOR REWRITING since \{a\} == insert a {}  
by blast

**lemma** insert-not-empty**: simp\]: insert a A \neq {}  
by blast

**lemmas** empty-not-insert = insert-not-empty [symmetric]  
**declare** empty-not-insert**: simp\]

**lemma** insert-absorb**: simp\]: a \in A ==\> insert a A = A  
— [simp] causes recursive calls when there are nested inserts  
— with quadratic running time  
by blast

**lemma** insert-absorb2**: simp\]: insert x (insert x A) = insert x A  
by blast

**lemma** insert-commute: insert x (insert y A) = insert y (insert x A)  
by blast

**lemma** insert-subset**: simp\]: (insert x A \subseteq B) = (x \in B & A \subseteq B)  
by blast

**lemma** mk-disjoint-insert: a \in A ==\> \exists B. A = insert a B & a \notin B  
— use new B rather than A - \{a\} to avoid infinite unfolding  
apply (rule-tac x = A - \{a\} in exI, blast)  
done

**lemma** insert-Collect: insert a (Collect P) = \{ u. u \neq a \<-- B P u\}  
by auto

**lemma** insert-inter-insert**: simp\]: insert a A \cap insert a B = insert a (A \cap B)  
by blast

**lemma** insert-disjoint**: simp\]: \langle insert a A \cap B = {} \rangle = (a \notin B \& A \cap B = {})  
\langle {} = insert a A \cap B \rangle = (a \notin B \& {} = A \cap B)  
by auto

**lemma** disjoint-insert**: simp\]: \langle B \cap insert a A = {} \rangle = (a \notin B \& B \cap A = {})  
\langle {} = A \cap insert b B \rangle = (b \notin A \& {} = A \cap B)  
by auto

Int

**lemma** Int-absorb: A \cap A = A
THEORY "Set"

by (fact inf-idem)

lemma Int-left-absorb: \( A \cap (A \cap B) = A \cap B \)
by (fact inf-left-idem)

lemma Int-commute: \( A \cap B = B \cap A \)
by (fact inf-commute)

lemma Int-left-commute: \( A \cap (B \cap C) = B \cap (A \cap C) \)
by (fact inf-left-commute)

lemma Int-assoc: \( (A \cap B) \cap C = A \cap (B \cap C) \)
by (fact inf-assoc)

lemmas Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute
— Intersection is an AC-operator

lemma Int-absorb1: \( B \subseteq A \Longrightarrow A \cap B = B \)
by (fact inf-absorb2)

lemma Int-absorb2: \( A \subseteq B \Longrightarrow A \cap B = A \)
by (fact inf-absorb1)

lemma Int-empty-left: \( \emptyset \cap B = \emptyset \)
by (fact inf-bot-left)

lemma Int-empty-right: \( A \cap \emptyset = \emptyset \)
by (fact inf-bot-right)

lemma disjoint-eq-subset-Compl: \( A \cap B = \emptyset \) = \( A \subseteq -B \)
by blast

lemma disjoint-iff-not-equal: \( A \cap B = \emptyset \) = \( \forall x \in A. \forall y \in B. x \neq y \)
by blast

lemma Int-UNIV-left: \( \text{UNIV} \cap B = B \)
by (fact inf-top-left)

lemma Int-UNIV-right: \( A \cap \text{UNIV} = A \)
by (fact inf-top-right)

lemma Int-Un-distrib: \( A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \)
by (fact inf-sup-distrib)

lemma Int-Un-distrib2: \( (B \cup C) \cap A = (B \cap A) \cup (C \cap A) \)
by (fact inf-sup-distrib2)

lemma Int-UNIV [simp]: \( A \cap B = \text{UNIV} \) = \( A = \text{UNIV} \& B = \text{UNIV} \)
by (fact inf-eq-top-iff)
lemma *Int-subset-iff* [simp]: \((C \subseteq A \cap B) = (C \subseteq A \& C \subseteq B)\)
by (fact le-inf-iff)

lemma *Int-Collect*: \((x \in A \cap \{x. \ P \ x\}) = (x \in A \& P \ x)\)
by blast

Un.

lemma *Un-absorb*: \(A \cup A = A\)
by (fact sup-idem)

lemma *Un-left-absorb*: \(A \cup (A \cup B) = A \cup B\)
by (fact sup-left-idem)

lemma *Un-commute*: \(A \cup B = B \cup A\)
by (fact sup-commute)

lemma *Un-left-commute*: \(A \cup (B \cup C) = B \cup (A \cup C)\)
by (fact sup-left-commute)

lemma *Un-assoc*: \((A \cup B) \cup C = A \cup (B \cup C)\)
by (fact sup-assoc)

lemmas *Un-ac* = Un-assoc Un-left-absorb Un-commute Un-left-commute
— Union is an AC-operator

lemma *Un-absorb1*: \(A \subseteq B \Longrightarrow A \cup B = B\)
by (fact sup-absorb2)

lemma *Un-absorb2*: \(B \subseteq A \Longrightarrow A \cup B = A\)
by (fact sup-absorb1)

lemma *Un-empty-left*: \(\{} \cup B = B\)
by (fact sup-bot-left)

lemma *Un-empty-right*: \(A \cup \{} = A\)
by (fact sup-bot-right)

lemma *Un-UNIV-left*: \(\text{UNIV} \cup B = \text{UNIV}\)
by (fact sup-top-left)

lemma *Un-UNIV-right*: \(A \cup \text{UNIV} = \text{UNIV}\)
by (fact sup-top-right)

lemma *Un-insert-left* [simp]: \((\text{insert} \ a \ B) \cup C = \text{insert} \ a \ (B \cup C)\)
by blast

lemma *Un-insert-right* [simp]: \(A \cup (\text{insert} \ a \ B) = \text{insert} \ a \ (A \cup B)\)
by blast
lemma Int-insert-left:
  \( (\text{insert } a \ B) \ \text{Int} \ C = (\text{if } a \in C \ \text{then insert } a \ (B \cap C) \ \text{else } B \cap C) \)
  by auto

lemma Int-insert-left-if0[simp]:
  \( a \notin C \implies (\text{insert } a \ B) \ \text{Int} \ C = B \cap C \)
  by auto

lemma Int-insert-left-if1[simp]:
  \( a \in C \implies (\text{insert } a \ B) \ \text{Int} \ C = \text{insert } a \ (B \ \text{Int} C) \)
  by auto

lemma Int-insert-right:
  \( A \cap (\text{insert } a \ B) = (\text{if } a \in A \ \text{then insert } a \ (A \cap B) \ \text{else } A \cap B) \)
  by auto

lemma Int-insert-right-if0[simp]:
  \( a \notin A \implies A \ \text{Int} \ (\text{insert } a \ B) = A \ \text{Int} B \)
  by auto

lemma Int-insert-right-if1[simp]:
  \( a \in A \implies A \ \text{Int} \ (\text{insert } a \ B) = \text{insert } a \ (A \ \text{Int} B) \)
  by auto

lemma Un-Int-distrib: \( A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \)
  by (fact sup-inf-distrib1)

lemma Un-Int-distrib2: \( (B \cap C) \cup A = (B \cup A) \cap (C \cup A) \)
  by (fact sup-inf-distrib2)

lemma Un-Int-crazy:
  \( (A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A) \)
  by blast

lemma subset-Un-eq: \( A \subseteq B \) = \( A \cup B = B \)
  by (fact le-iff-sup)

lemma Un-empty [iff]: \( A \cup B = \{\} \) = \( A = \{\} \ & \ B = \{\} \)
  by (fact sup-eq-bot-iff)

lemma Un-subset-iff [simp]: \( A \cup B \subseteq C \) = \( A \subseteq C \ & \ B \subseteq C \)
  by (fact le-sup-iff)

lemma Un-Diff-Int: \( A - B \) \cup (A \cap B) = A
  by blast

lemma Diff-Int2: \( A \cap C - B \cap C = A \cap C - B \)
  by blast
Set complement

**Lemma** Compl-disjoint [simp]:  \( A \cap -A = {} \)
by (fact inf-compl-bot)

**Lemma** Compl-disjoint2 [simp]:  \(-A \cap A = {}\)
by (fact compl-inf-bot)

**Lemma** Compl-partition:  \( A \cup -A = \text{UNIV} \)
by (fact sup-compl-top)

**Lemma** Compl-partition2:  \(-A \cup A = \text{UNIV} \)
by (fact compl-sup-top)

**Lemma** double-complement:  \(-(-A) = (A::'a set) \)
bym (fact double-compl)

**Lemma** Compl-Un:  \(- (A \cup B) = (-A) \cap (-B) \)
bym (fact compl-sup)

**Lemma** Compl-Int:  \(- (A \cap B) = (-A) \cup (-B) \)
bym (fact compl-inf)

**Lemma** subset-Compl-self-eq:  \((A \subseteq -A) = (A = \{\}) \)
bym blast

**Lemma** Un-Int-assoc-eq:  \((\forall x \in A \cup B. \ P x) = (\forall x \in A. \ P x) \& (\forall x \in B. \ P x) \)
— Halmos, Naive Set Theory, page 16.
bym blast

**Lemma** Compl-UNIV-eq:  \(- \text{UNIV} = \{\} \)
bym (fact compl-top-eq)

**Lemma** Compl-empty-eq:  \(-\{\} = \text{UNIV} \)
bym (fact compl-bot-eq)

**Lemma** Compl-subset-Compl-iff [iff]:  \((-A \subseteq -B) = (B \subseteq A) \)
bym (fact compl-le-compl-iff)

**Lemma** Compl-eq-Compl-iff [iff]:  \((-A = -B) = (A = (B::'a set)) \)
bym (fact compl-eq-compl-iff)

**Lemma** Compl-insert:  \(- \text{insert } x A = (-A) \setminus \{x\} \)
bym blast

Bounded quantifiers.
The following are not added to the default simpset because (a) they duplicate
the body and (b) there are no similar rules for Int.

**Lemma** ball-Un:  \((\forall x \in A \cup B. \ P x) = ((\forall x \in A. \ P x) \& (\forall x \in B. \ P x)) \)
THEORY “Set”

by blast

lemma bex-Un: \( (\exists x \in A \cup B. \ P x) = ( (\exists x \in A. \ P x) \ | \ (\exists x \in B. \ P x)) \)
by blast

Set difference.

lemma Diff-eq: \( A - B = A \cap (-B) \)
by blast

lemma Diff-eq-empty-iff [simp]: \( (A - B = \{\}) = (A \subseteq B) \)
by blast

lemma Diff-cancel [simp]: \( A - A = \{\} \)
by blast

lemma Diff-idemp [simp]: \( (A - B) - B = A - (B::'a set) \)
by blast

lemma Diff-triv: \( A \cap B = \{\} \implies A - B = A \)
by (blast elim: equalityE)

lemma empty-Diff [simp]: \( \{\} - A = \{\} \)
by blast

lemma Diff-empty [simp]: \( A - \{\} = A \)
by blast

lemma Diff-UNIV [simp]: \( A - \text{UNIV} = \{\} \)
by blast

lemma Diff-insert0 [simp]: \( x \notin A \implies A - \text{insert} \ x \ B = A - B \)
by blast

lemma Diff-insert: \( A - \text{insert} \ a \ B = A - B - \{a\} \)
— NOT SUITABLE FOR REWRITING since \( \{a\} \equiv \text{insert} \ a \ 0 \)
by blast

lemma Diff-insert2: \( A - \text{insert} \ a \ B = A - \{a\} - B \)
— NOT SUITABLE FOR REWRITING since \( \{a\} \equiv \text{insert} \ a \ 0 \)
by blast

lemma insert-Diff-if: \( \text{insert} \ x A - B = (if \ x \in B \ then \ A - B \ else \ \text{insert} \ x \ (A - B)) \)
by auto

lemma insert-Diff1 [simp]: \( x \in B \implies \text{insert} \ x A - B = A - B \)
by blast

lemma insert-Diff-single [simp]: \( \text{insert} \ a \ (A - \{a\}) = \text{insert} \ a \ A \)
by blast

lemma insert-Diff: \( a \in A \implies insert a (A - \{a\}) = A \)
  by blast

lemma Diff-insert-absorb: \( x \notin A \implies (insert x A) - \{x\} = A \)
  by auto

lemma Diff-disjoint [simp]: \( A \cap (B - A) = \{\} \)
  by blast

lemma Diff-partition: \( A \subseteq B \implies A \cup (B - A) = B \)
  by blast

lemma double-diff: \( A \subseteq B \implies B \subseteq C \implies B - (C - A) = A \)
  by blast

lemma Un-Diff-cancel [simp]: \( A \cup (B - A) = A \cup B \)
  by blast

lemma Un-Diff-cancel2 [simp]: \( (B - A) \cup A = B \cup A \)
  by blast

lemma Diff-Un: \( A - (B \cup C) = (A - B) \cap (A - C) \)
  by blast

lemma Diff-Int: \( A - (B \cap C) = (A - B) \cup (A - C) \)
  by blast

lemma Un-Diff: \( (A \cup B) - C = (A - C) \cup (B - C) \)
  by blast

lemma Int-Diff: \( (A \cap B) - C = A \cap (B - C) \)
  by blast

lemma Diff-Int-distrib: \( C \cap (A - B) = (C \cap A) - (C \cap B) \)
  by blast

lemma Diff-Int-distrib2: \( (A - B) \cap C = (A \cap C) - (B \cap C) \)
  by blast

lemma Diff-Compl [simp]: \( A - (-B) = A \cap B \)
  by auto

lemma Compl-Diff-eq [simp]: \( - (A - B) = -A \cup B \)
  by blast

Quantification over type bool.

lemma bool-induct: \( P \ True \implies P \ False \implies P \ x \)
by (cases x) auto

lemma all-bool-eq: \( (\forall b. P b) \iff P \text{True} \land P \text{False} \)
  by (auto intro: bool-induct)

lemma bool-contrapos: \( P x \implies \neg P \text{False} \implies P \text{True} \)
  by (cases x) auto

lemma ex-bool-eq: \( (\exists b. P b) \iff P \text{True} \lor P \text{False} \)
  by (auto intro: bool-contrapos)

lemma UNIV-bool: \( \text{UNIV} = \{ \text{False}, \text{True} \} \)
  by (auto intro: bool-induct)

P

lemma Pow-empty [simp]: \( \text{Pow} \{ \} = \{ \{ \} \} \)
  by (auto simp add: Pow-def)

lemma Pow-insert: \( \text{Pow} (\text{insert} \ a \ A) = \text{Pow} \ A \cup (\text{insert} \ a \ \text{Pow} \ A) \)
  by (blast intro: image-eqI [where ?x = u - \{a\} for u])

lemma Pow-Compl: \( \text{Pow} (\neg A) = \{\neg B \mid B. A \in \text{Pow} B\} \)
  by (blast intro: exI [where ?x = \neg u for u])

lemma Pow-UNIV [simp]: \( \text{Pow} \ \text{UNIV} = \text{UNIV} \)
  by blast

lemma Un-Pow-subset: \( \text{Pow} \ A \cup \text{Pow} \ B \subseteq \text{Pow} (\ A \cup \ B) \)
  by blast

lemma Pow-Int-eq [simp]: \( \text{Pow} (\ A \cap \ B) = \text{Pow} \ A \cap \text{Pow} \ B \)
  by blast

Miscellany.

lemma set-eq-subset: \( (A = B) = (A \subseteq B \land B \subseteq A) \)
  by blast

lemma subset-if: \( (A \subseteq B) = (\forall t. t \in A \longrightarrow t \in B) \)
  by blast

lemma subset-if-psubset-eq: \( (A \subseteq B) = ((A \subseteq B) \mid (A = B)) \)
  by (unfold less-le) blast

lemma all-not-in-conv [simp]: \( (\forall x. x \notin A) = (A = \{\}) \)
  by blast

lemma ex-in-conv: \( (\exists x. x \in A) = (A \neq \{\}) \)
  by blast
lemma ball-simps [simp, no-apt]:
\[ \land A \land P \land Q. (\forall x \in A. P \lor Q) \iff ((\forall x \in A. P) \lor Q) \]
\[ \land A \land P \land Q. (\forall x \in A. P \lor Q x) \iff (P \lor (\forall x \in A. Q x)) \]
\[ \land A \land P \land Q. (\forall x \in A. P \rightarrow Q x) \iff (P \rightarrow (\forall x \in A. Q x)) \]
\[ \land A \land P \land Q. (\forall x \in A. P x \rightarrow Q) \iff ((\exists x \in A. P x) \rightarrow Q) \]
\[ \land P. (\forall x \in \text{UNIV}. P x) \iff (\forall x. P x) \]
\[ \land a \land B \land P. (\forall x \in \text{insert} \ a \ B \land P x) \iff (P a \land (\forall x \in B. P x)) \]
\[ \land P \land Q. (\forall x \in \text{Collect} \ Q \land P x) \iff (\forall x. Q x \rightarrow P x) \]
\[ \land A \land P \land f. (\forall x \in f' A \land P x) \iff (\forall x \in A. P (f x)) \]
\[ \land A \land P. (\neg (\forall x \in A. P x)) \iff (\exists x \in A. \neg P x) \]
by auto

lemma bex-simps [simp, no-apt]:
\[ \land A \land P \land Q. (\exists x \in A. P x \land Q) \iff ((\exists x \in A. P x) \land Q) \]
\[ \land A \land P \land Q. (\exists x \in A. P \land Q x) \iff (P \land (\exists x \in A. Q x)) \]
\[ \land P. (\exists x \in \text{UNIV}. P x) \iff (\exists x. P x) \]
\[ \land a \land B \land P. (\exists x \in \text{insert} \ a \ B \land P x) \iff (P a \mid (\exists x \in B. P x)) \]
\[ \land P \land Q. (\exists x \in \text{Collect} \ Q \land P x) \iff (\exists x. Q x \land P x) \]
\[ \land A \land P \land f. (\exists x \in f' A \land P x) \iff (\exists x \in A. P (f x)) \]
\[ \land A \land P. (\neg (\exists x \in A. P x)) \iff (\forall x \in A. \neg P x) \]
by auto

6.4.4 Monotonicity of various operations

lemma image-mono: \( A \subseteq B \implies fA \subseteq fB \)
by blast

lemma Pow-mono: \( A \subseteq B \implies \text{Pow} A \subseteq \text{Pow} B \)
by blast

lemma insert-mono: \( C \subseteq D \implies \text{insert} \ a \ C \subseteq \text{insert} \ a \ D \)
by blast

lemma Un-mono: \( A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D \)
by (fact sup-mono)

lemma Int-mono: \( A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D \)
by (fact inf-mono)

lemma Diff-mono: \( A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D \)
by blast

lemma Compl-anti-mono: \( A \subseteq B \implies \neg B \subseteq \neg A \)
by (fact compl-mono)

Monotonicity of implications.
THEORY "Set" 184

lemma in-mono: \( A \subseteq B \implies x \in A \implies x \in B \)
apply (rule impl)
apply (erule subsetD, assumption)
done

lemma conj-mono: \( P_1 \implies Q_1 \implies P_2 \implies Q_2 \implies (P_1 \land P_2) \implies (Q_1 \land Q_2) \)
by iprover

lemma disj-mono: \( P_1 \implies Q_1 \implies P_2 \implies Q_2 \implies (P_1 \lor P_2) \implies (Q_1 \lor Q_2) \)
by iprover

lemma imp-mono: \( Q_1 \implies P_1 \implies P_2 \implies Q_2 \implies (P_1 \implies P_2) \implies (Q_1 \implies Q_2) \)
by iprover

lemma imp-refl: \( P \implies P \) ..

lemma not-mono: \( Q \implies P \implies \neg P \implies \neg Q \)
by iprover

lemma ex-mono: \( (!x. P x \implies Q x) \implies (EX x. P x) \implies (EX x. Q x) \)
by iprover

lemma all-mono: \( (!x. P x \implies Q x) \implies (ALL x. P x) \implies (ALL x. Q x) \)
by iprover

lemma Collect-mono: \( (!x. P x \implies Q x) \implies \text{Collect } P \subseteq \text{Collect } Q \)
by blast

lemma Int-Collect-mono:
\[ A \subseteq B \implies (!x. x \in A \implies P x \implies Q x) \implies A \cap \text{Collect } P \subseteq B \cap \text{Collect } Q \]
by blast

lemmas basic-monos =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma eq-to-mono: \( a = b \implies c = d \impliedby b \implies d \impliedby a \implies c \)
by iprover

6.4.5 Inverse image of a function

definition vimage :: \( \{ 'a \implies 'b \} \implies 'b \implies \{ a \in \text{set} (\text{infixr} \ '90) \mid \text{where} f \in 'B \} \)
by blast
by (unfold vimage-def) blast

lemma vimage-singleton-eq: (a : f −' {b}) = (f a = b)
  by simp

lemma vimageI [intro]: f a = b ==> b:B ==> a : f −' B
  by (unfold vimage-def) blast

lemma vimageI2: f a : A ==> a : f −' A
  by (unfold vimage-def) fast

lemma vimageE [elim!]: a: f −' B ==> (!x. f a = x ==> x:B ==> P) ==> P
  by (unfold vimage-def) blast

lemma vimageD: a : f −' A ==> f a : A
  by (unfold vimage-def) fast

lemma vimage-empty [simp]: f −' {} = {}
  by blast

lemma vimage-Compl: f −' (-A) = -(f −' A)
  by blast

lemma vimage-Un [simp]: f −' (A Un B) = (f −' A) Un (f −' B)
  by blast

lemma vimage-Int [simp]: f −' (A Int B) = (f −' A) Int (f −' B)
  by fast

lemma vimage-Collect-eq [simp]: f −' Collect P = {y. P (f y)}
  by blast

lemma vimage-Collect: (!x. P (f x) = Q x) ==> f −' (Collect P) = Collect Q
  by blast

lemma vimage-insert: f −'(insert a B) = (f −'{a}) Un (f −'B)
  — NOT suitable for rewriting because of the recurrence of {a}.
  by blast

lemma vimage-Diff: f −' (A - B) = (f −' A) - (f −' B)
  by blast

lemma vimage-UNIV [simp]: f −' UNIV = UNIV
  by blast

lemma vimage-mono: A ⊆ B ==> f −' A ⊆ f −' B
  — monotonicity
  by blast
lemma vimage-image-eq: \( f^{-1}(f \cdot A) = \{ y. \ \exists x:A. \ f x = f y \} \)
by (blast intro: sym)

lemma image-vimage-subset: \( f^{-1}(f \cdot A) \subseteq A \)
by blast

lemma image-vimage-eq [simp]: \( f^{-1}(f \cdot A) = A \cap \text{range } f \)
by blast

lemma image-subset-iff-subset-vimage: \( f^{-1} A \subseteq B \iff A \subseteq f^{-1} B \)
by blast

lemma vimage-const [simp]: \( (\lambda x. \ c)^{-1} A = \{ \text{if } c \in A \text{ then UNIV else } \} \)
by auto

lemma vimage-if [simp]: \( (\lambda x. \ \text{if } x \in B \text{ then } c \text{ else } d)^{-1} A = \{ \text{if } c \in A \text{ then } (\text{if } d \in A \text{ then UNIV else } B) \text{ else if } d \in A \text{ then } -B \text{ else } \} \)
by (auto simp add: vimage-def)

lemma vimage-inter-cong:
\( (\forall w. \ w \in S \implies f w = g w) \implies f^{-1} \cap S = g^{-1} \cap S \)
by auto

lemma vimage-ident [simp]: \( (\% x. \ x)^{-1} Y = Y \)
by blast

6.4.6 Getting the Contents of a Singleton Set

definition the-elem :: 'a set \Rightarrow 'a where
the-elem \( X = (\text{THE } x. \ X = \{ x \}) \)

lemma the-elem-eq [simp]: the-elem \( \{ x \} = x \)
by (simp add: the-elem-def)

lemma the-elem-image-unique:
assumes \( A \neq \{ \} \)
assumes *: \( \forall y. \ y \in A \implies f y = f x \)
shows the-elem \( (f \cdot A) = f x \)

unfolding the-elem-def proof (rule the1-equality)
from \( A \neq \{ \} \) obtain \( y \) where \( y \in A \) by auto
with * have \( f x = f y \) by simp
with \( y \in A \) have \( f x \in f \cdot A \) by blast
with * show \( f^{-1} A = \{ f x \} \) by auto
then show \( \exists! x. f^{-1} A = \{ x \} \) by auto
qed
6.4.7 Least value operator

**lemma** Least-mono:

```
mono (f :: 'a::order => 'b::order) ==> EX x:S. ALL y:S. x <= y
===> (LEAST y. y : f ' S) = f (LEAST x. x : S)
```

— Courtesy of Stephan Merz

**apply** clarify

**apply** (erule-tac P = %x. x : S in LeastI2-order, fast)

**apply** (rule LeastI2-order)

**apply** (auto elim: monoD intro: order-antisym)

**done**

6.4.8 Monad operation

**definition** bind :: 'a set => ('a => 'b set) => 'b set where

```
bind A f = {x. \exists B \in f'A. x \in B}
```

**hide-const** (open) bind

**lemma** bind-bind:

```
fixes A :: 'a set
shows Set.bind (Set.bind A B) C = Set.bind A (\lambda x. Set.bind (B x) C)
by (auto simp add: bind-def)
```

**lemma** empty-bind [simp]:

```
Set.bind {} f = {}
```

**by** (simp add: bind-def)

**lemma** nonempty-bind-const:

```
A eq {} => Set.bind A (\lambda-. B) = B
```

**by** (auto simp add: bind-def)

**lemma** bind-const: Set.bind A (\lambda-. B) = (if A = {} then {} else B)

**by** (auto simp add: bind-def)

6.4.9 Operations for execution

**definition** is-empty :: 'a set => bool where

```
[code-abbrev]: is-empty A <-> A eq {}
```

**hide-const** (open) is-empty

**definition** remove :: 'a => 'a set => 'a set where

```
[code-abbrev]: remove x A = A - {x}
```

**hide-const** (open) remove

**lemma** member-remove [simp]:

```
x \in Set.remove y A <-> x \in A \land x \neq y
```

**by** (simp add: remove-def)
definition filter :: ('a ⇒ bool) ⇒ 'a set ⇒ 'a set where
\[ \text{filter } P \ A = \{ a \in A. \ P \ a \} \]

hide-const (open) filter

lemma member-filter [simp]:
\[ x \in \text{Set} \text{.filter } P \ A \leftrightarrow x \in A \land P \ x \]
by (simp add: filter-def)

instantiation set :: (equal) equal
begin

definition HOL.equal A B \leftrightarrow A \subseteq B \land B \subseteq A

instance proof
qed (auto simp add: equal-set-def)

end

Misc

hide-const (open) member not-member

lemmas equalityI = subset-antisym

ML \[ \}}} val Ball-def = \{ thm Ball-def \} val Bex-def = \{ thm Bex-def \} val CollectD = \{ thm CollectD \} val CollectE = \{ thm CollectE \} val CollectI = \{ thm CollectI \} val Collect-conj-eq = \{ thm Collect-conj-eq \} val Collect-mem-eq = \{ thm Collect-mem-eq \} val IntD1 = \{ thm IntD1 \} val IntD2 = \{ thm IntD2 \} val IntE = \{ thm IntE \} val IntI = \{ thm IntI \} val Int-Collect = \{ thm Int-Collect \} val UNIV-I = \{ thm UNIV-I \} val UNIV-witness = \{ thm UNIV-witness \} val UnE = \{ thm UnE \} val UnI1 = \{ thm UnI1 \} val UnI2 = \{ thm UnI2 \} val ballE = \{ thm ballE \} val ballI = \{ thm ballI \} val bexCI = \{ thm bexCI \} val bexE = \{ thm bexE \} val bexI = \{ thm bexI \} \]
7 Typedef: HOL type definitions

theory Typedef
imports Set
keywords typedef :: thy-goal and morphisms
begin

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
      and Rep-inverse: Abs (Rep x) = x
      and Abs-inverse: y ∈ A ==> Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin
lemma Rep-inject:
(Rep x = Rep y) = (x = y)
proof
  assume Rep x = Rep y
  then have Abs (Rep x) = Abs (Rep y) by (simp only:)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  moreover have Abs (Rep y) = y by (rule Rep-inverse)
  ultimately show x = y by simp
next
  assume x = y
  thus Rep x = Rep y by (simp only:)
qed

lemma Abs-inject:
  assumes x: x ∈ A and y: y ∈ A
  shows (Abs x = Abs y) = (x = y)
proof
  assume Abs x = Abs y
  then have Rep (Abs x) = Rep (Abs y) by (simp only:)
  moreover from x have Rep (Abs x) = x by (rule Abs-inverse)
  moreover from y have Rep (Abs y) = y by (rule Abs-inverse)
  ultimately show x = y by simp
next
  assume x = y
  thus Abs x = Abs y by (simp only:)
qed

lemma Rep-cases [cases set]:
  assumes y: y ∈ A
  and hyp: !!x. y = Rep x ==> P
  shows P
proof (rule hyp)
  from y have Rep (Abs y) = y by (rule Abs-inverse)
  thus y = Rep (Abs y) ..
qed

lemma Abs-cases [cases type]:
  assumes r: !!y. x = Abs y ==> y ∈ A ==> P
  shows P
proof (rule r)
  have Abs (Rep x) = x by (rule Rep-inverse)
  thus x = Abs (Rep x) ..
  show Rep x ∈ A by (rule Rep)
qed

lemma Rep-induct [induct set]:
  assumes y: y ∈ A
  and hyp: !!x. P (Rep x)
  shows P y
proof
  have P (Rep (Abs y)) by (rule hyp)
  moreover from y have Rep (Abs y) = y by (rule Abs-inverse)
  ultimately show P y by simp
qed

lemma Abs-induct [induct type]:
  assumes r: !!y. y ∈ A ==> P (Abs y)
  shows P x
proof
  have Rep x ∈ A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range: range Rep = A
proof
  show range Rep ⊆ A using Rep by (auto simp add: image-def)
  show A ⊆ range Rep
proof
    fix x assume x : A
    hence x = Rep (Abs x) by (rule Abs-inverse [symmetric])
    thus x : range Rep by (rule range-eqI)
  qed
qed

lemma Abs-image: Abs ' A = UNIV
proof
  show Abs ' A ⊆ UNIV by (rule subset-UNIV)
next
  show UNIV ⊆ Abs ' A
proof
    fix x
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x : A by (rule Rep)
    ultimately show x : Abs ' A by (rule image-eqI)
  qed
qed

end

ML-file Tools/typedef.ML setup Typedef.setup

end

8 Fun: Notions about functions

theory Fun
imports Set
keywords functor :: thy-goal
begin

lemma apply-inverse:
f x = u \implies (\forall x. P x \implies g (f x) = x) \implies P x \implies x = g u
by auto

8.1 The Identity Function id

definition id :: 'a => 'a where
  id = (\lambda x. x)

lemma id-apply [simp]: id x = x
  by (simp add: id-def)

lemma image-id [simp]: image id = id
  by (simp add: id-def fun-eq-iff)

lemma vimage-id [simp]: vimage id = id
  by (simp add: id-def fun-eq-iff)

code-printing
  constant id -> (Haskell) id

8.2 The Composition Operator f \circ g

definition comp :: ('b => 'c) => ('a => 'b) => 'a => 'c (infixl o 55) where
  f o g = (\lambda x. f (g x))

notation (xsymbols)
  comp (infixl \circ 55)

notation (HTML output)
  comp (infixl \circ 55)

lemma comp-apply [simp]: (f o g) x = f (g x)
  by (simp add: comp-def)

lemma comp-assoc: (f o g) o h = f o (g o h)
  by (simp add: fun-eq-iff)

lemma id-comp [simp]: id o g = g
  by (simp add: fun-eq-iff)

lemma comp-id [simp]: f o id = f
  by (simp add: fun-eq-iff)

lemma comp-eq-dest:
a o b = c o d \implies a (b v) = c (d v)
by \((\text{simp add: fun-eq-iff})\)

**Lemma** \(\text{comp-eq-elim}::\)
\[
a \circ b = c \circ d \implies (\forall v. \ (a \ (b \ v)) = (c \ (d \ v))) \implies R \implies R
\]
by \((\text{simp add: fun-eq-iff})\)

**Lemma** \(\text{comp-eq-dest-lhs}::\)
\[
a \circ b = c \implies a \ (b \ v) = c \ v
\]
by \text{clarsimp}

**Lemma** \(\text{comp-eq-id-dest}::\)
\[
a \circ b = \text{id} \circ c \implies a \ (b \ v) = c \ v
\]
by \text{clarsimp}

**Code-printing**
\[
\text{constant \ comp \ \rightarrow \ (SML) infixl 5 \ circ \ and \ (Haskell) infixr 9 .}
\]

### 8.3 The Forward Composition Operator \(\text{fcomp}\)

**Definition** \(\text{fcomp}::\) \((\Rightarrow 'a \Rightarrow 'b) \Rightarrow (\Rightarrow 'b \Rightarrow 'c) \Rightarrow (\Rightarrow 'a \Rightarrow 'c)\) \((\text{infixl} \circ> 60)\) where
\[
f \circ> g = (\lambda x. \ g (f x))
\]

**Lemma** \(\text{fcomp-apply} [\text{simp}]:\)
\[
(f \circ> g) \ x = g \ (f \ x)
\]
by \((\text{simp add: fcomp-def})\)

**Lemma** \(\text{fcomp-assoc}::\)
\[
(f \circ> g) \circ> h = f \circ> (g \circ> h)
\]
by \((\text{simp add: fcomp-def})\)

**Lemma** \(\text{id-fcomp} [\text{simp}]:\)
\[
id \circ> g = g
\]
by \((\text{simp add: fcomp-def})\)

**Lemma** \(\text{fcomp-id} [\text{simp}]:\)
\[
f \circ id = f
\]
by \((\text{simp add: fcomp-def})\)

**Code-printing**
\[
\text{constant \ fcomp \ \rightarrow \ (Eval) infixl 1 \ #>}
\]

**No-notation** \(\text{fcomp} \ (\text{infixl} \circ> 60)\)

### 8.4 Mapping functions

**Definition** \(\text{map-fun}::\) \((\Rightarrow 'c \Rightarrow 'a) \Rightarrow (\Rightarrow 'b \Rightarrow 'd) \Rightarrow (\Rightarrow 'a \Rightarrow 'b) \Rightarrow (\Rightarrow 'c \Rightarrow 'd)\) where
\[
\text{map-fun} \ f \ g \ h = g \circ h \circ f
\]
lemma map-fun-apply [simp]:
  map-fun f g h x = g (h (f x))
  by (simp add: map-fun-def)

8.5 Injectivity and Bijectivity

definition inj-on :: ('a ⇒ 'b) ⇒ 'a set ⇒ bool where — injective
  inj-on f A ←→ (∀x ∈ A. ∀y ∈ A. f x = f y → x = y)

definition bij-betw :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool where — bijective
  bij-betw f A B ←→ inj-on f A ∧ f ' A = B

A common special case: functions injective, surjective or bijective over the
entire domain type.

abbreviation
  inj f ≡ inj-on f UNIV

abbreviation surj :: ('a ⇒ 'b) ⇒ bool where — surjective
  surj f ≡ (range f = UNIV)

abbreviation
  bij f ≡ bij-betw f UNIV UNIV

The negated case:

translations
  ¬ CONST surj f <== CONST range f ≠ CONST UNIV

lemma injI:
  assumes ∃x y. f x = f y → x = y
  shows inj f
  using assms unfolding inj-on-def by auto

theorem range-ex1-eq: inj f → b : range f = (EX x. b = f x)
  by (unfold inj-on-def, blast)

lemma injD: ∃x. inj(f); f(x) = f(y) |===> x=y
  by (simp add: inj-on-def)

lemma inj-on-eq-iff: inj-on f A ===> x:A ===> y:A ===> (f(x) = f(y)) = (x=y)
  by (force simp add: inj-on-def)

lemma inj-on-cong:
  (∀a. a : A → f a = g a) → inj-on f A = inj-on g A
  unfolding inj-on-def by auto

lemma inj-on-strict-subset:
  inj-on f B → A ⊂ B → f ' A ⊂ f ' B
  unfolding inj-on-def by blast
lemma inj-comp:
inj f \implies inj g \implies inj (f \circ g)
by (simp add: inj-on-def)

lemma inj-fun: inj f \implies inj (\lambda x. f x)
by (simp add: inj-on-def fun-eq-iff)

lemma inj-eq: inj f \implies (f(x) = f(y)) = (x = y)
by (simp add: inj-on-eq-iff)

lemma inj-on-id[simp]: inj-on id A
by (simp add: inj-on-def)

lemma inj-on-id2[simp]: inj-on (%x. A)
by (simp add: inj-on-def)

lemma inj-on-Int: inj-on f A \vee inj-on f B \implies inj-on f (A \cap B)
unfolding inj-on-def by blast

lemma surj-id: surj id
by simp

lemma bij-id[simp]: bij id
by (simp add: bij-betw-def)

lemma inj-onI: (!!! x y. \[ | x:A; y:A; f(x) = f(y) | \] \implies x = y) \implies inj-on f A
by (simp add: inj-on-def)

lemma inj-on-inverseI: (!!! x:A \implies g(f(x)) = x) \implies inj-on f A
by (auto dest: arg-cong [of concl: g] simp add: inj-on-def)

lemma inj-onD: [ | inj-on f A; f(x) = f(y); x:A; y:A | \] \implies x = y
by (unfold inj-on-def, blast)

lemma inj-on-iff: [ | inj-on f A; x:A; y:A | \] \implies (f(x) = f(y)) = (x = y)
by (fact inj-on-eq-iff)

lemma comp-inj-on:
[ | inj-on f A; inj-on g (f:A) | \] \implies inj-on (g o f) A
by (simp add: comp-def inj-on-def)

lemma inj-on-imageI: inj-on (g o f) A \implies inj-on g (f' A)
by (simp add: inj-on-def) blast

lemma inj-on-image-iff: [ ALL x:A. ALL y:A. (g(f x) = g(f y)) = (g x = g y); inj-on f A \] \implies inj-on g (f' A) = inj-on g A
apply (unfold inj-on-def)
apply blast
THEORY “Fun”

done

lemma inj-on- contraD: \[ \text{inj-on} f A; \; \sim x = y; \; x:A; \; y:A \implies \sim f(x) = f(y) \]
by (unfold inj-on-def, blast)

lemma inj-singleton: inj \( \% s. \{ s \} \)
by (simp add: inj-on-def)

lemma inj-on-empty iff: inj-on f \{\}
by (simp add: inj-on-def)

lemma subset-inj-on: \[ \text{inj-on} f B; \; A \subseteq B \implies \text{inj-on} f A \]
by (unfold inj-on-def, blast)

lemma inj-on-Un:
\[ \text{inj-on} f (A \cup B) = \\]
\[ (\text{inj-on} f A \& \text{inj-on} f B \& f'(A-B) \cap f'(B-A) = \{\}) \]
apply (unfold inj-on-def)
apply (blast intro:sym)
done

lemma inj-on-insert iff:
\[ \text{inj-on} f (\text{insert} a A) = (\text{inj-on} f A \& f a \sim: f'(A-\{a\})) \]
apply (unfold inj-on-def)
apply (blast intro:sym)
done

lemma inj-on-diff: \[ \text{inj-on} f A \implies \text{inj-on} f (A-B) \]
apply (unfold inj-on-def)
apply (blast)
done

lemma comp-inj-on-iff:
\[ \text{inj-on} f A \implies \text{inj-on} f' (f' A) \iff \text{inj-on} (f' o f) A \]
by (auto simp add: comp-inj-on inj-on-def)

lemma inj-on-imageI2:
\[ \text{inj-on} (f' o f) A \implies \text{inj-on} f A \]
by (auto simp add: comp-inj-on inj-on-def)

lemma inj-img-insertE:
assumes inj-on f A
assumes \( x \notin B \) and insert x B = f' A
obtains \( x' A' \) where \( x' \notin A' \) and A = insert x' A'
and \( x = f x' \) and B = f' A'
proof -
  from assms have \( x \in f' A \) by auto
then obtain \( x' \) where \( x' \in A \) \( x = f x' \) by auto
then have A = insert x' (A - \{x'\}) by auto
with assms * have \( B = f' \{x'\} \)
  by (auto dest: inj-on-contraD)
have \( x' \notin A - \{x'\} \) by simp
from \( x' \notin A - \{x'\} \) \( A = insert x' (A - \{x'\}) \) \( x = f x' \) \( B = image f (A - \{x'\}) \) show \( \text{thesis} \).
qed

lemma linorder-injI:
  assumes \( \forall x y. x < (y::'a::linorder) \rightarrow f x \neq f y \)
  shows \( \text{inj} f \)
  — Courtesy of Stephan Merz
proof (rule inj-onI)
  fix \( x y \)
  assume f-eq: \( f x = f y \)
  show \( x = y \) by (rule linorder-cases) (auto dest: hyp simp: f-eq)
qed

lemma surj-def: \( \text{surj} f \longleftrightarrow (\forall y. \exists x. y = f x) \)
  by auto

lemma surjI: \( \text{assumes \( \forall x \). \( g(f x) = x \) shows \( \text{surj} g \) \) \)
  using \( \text{symmetric} \) by auto

lemma surjD: \( \text{surj} f \Longrightarrow \exists x. y = f x \)
  by (simp add: surj-def)

lemma surjE: \( \text{surj} f \Longrightarrow (\forall x. y = f x 
  \Longrightarrow C) \Longrightarrow C \)
  by (simp add: surj-def, blast)

lemma comp-surj: \( \text{|| surj f; surj g ||} \Longrightarrow \text{surj} (g \circ f) \)
apply (simp add: comp-def surj-def, clarify)
apply (drule-tac x = y in spec, clarify)
apply (drule-tac x = x in spec, blast)
done

lemma bij-betw-imageI:
  \( \text{inj-on} f A; f' A = B \) \( \Longrightarrow \) \( \text{bij-betw} f A B \)
unfolding bij-betw-def by clarify

lemma bij-betw-imp-surj-on: \( \text{bij-betw} f A B \Longrightarrow f' A = B \)
unfolding bij-betw-def by clarify

lemma bij-betw-imp-surj: \( \text{bij-betw} f A \text{UNIV} 
  \Longrightarrow \text{surj} f \)
unfolding bij-betw-def by auto

lemma bij-betw-emptyI:
  \( \text{assumes \( \text{bij-betw} f \{\} A \) shows \( A = \{\} \) \) 

using assms unfolding bij-betw-def by blast

lemma bij-betw-empty2:
  assumes bij-betw f A {}
  shows A = {}
using assms unfolding bij-betw-def by blast

lemma inj-on-imp-bij-betw:
  inj-on f A ⇒ bij-betw f A (f ' A)
unfolding bij-betw-def by simp

lemma bij-def: bij f ☐ inj f ∧ surj f
unfolding bij-def ..

lemma bijI: [| inj f; surj f |] ==> bij f
by (simp add: bij-def)

lemma bij-is-inj: bij f ==> inj f
by (simp add: bij-def)

lemma bij-is-surj: bij f ==> surj f
by (simp add: bij-def)

lemma bij-betw-imp-inj-on:
  bij-betw f A B ⇒ inj-on f A
by (simp add: bij-betw-def)

lemma bij-betw-trans:
  bij-betw f A B ⇒ bij-betw g B C ⇒
  bij-betw (g o f) A C
by (auto simp add: bij-betw-def comp-inj-on)

lemma bij-comp: bij f ==> bij g ==> bij (g o f)
by (rule bij-betw-trans)

lemma bij-betw-comp-iff:
  bij-betw f A A' ⇒ bij-betw f' A' A'' ⇔
  bij-betw (f' o f) A A''
by(auto simp add: bij-betw-def inj-on-def)

lemma bij-betw-comp-iff2:
  assumes BIJ: bij-betw f' A' A'' and IM: f' A ≤ A'
  shows bij-betw f A A' ⇔
  bij-betw (f' o f) A A''
using assms
proof(auto simp add: bij-betw-comp-iff)
  assume *: bij-betw (f' o f) A A''
  thus bij-betw f A A'
  using IM
proof(auto simp add: bij-betw-def)
  assume inj-on (f' o f) A
  thus inj-on f A using inj-on-imageI2 by blast
next
fix \( a' \) assume \( \forall a': a' \in A' \)
hence \( f' a' \in A'' \) using BIJ unfolding bij-betw-def by auto
then obtain \( a \) where \( \exists 1: a \in A \land f'(f a) = f' a' \) using \( \ast \)
unfolding bij-betw-def by force
hence \( f a \in A'' \) using IM by auto
thus \( a' \in f' A \) using \( \ast \) unfolding bij-betw-def inj-on-def by auto
qed

lemma bij-betw-inv: assumes bij-betw \( f A B \) shows \( \exists g. \text{bij-betw } g B A \)
proof
  have \( i: \text{inj-on } f A \) and \( s: f' A = B \)
  using assms by(auto simp:bij-betw-def)
  let \( ?P = \lambda b a. a:A \land f a = b \) let \( ?g = \lambda b. \text{The } (\exists a. ?P b a) \)
  { fix \( a b \) assume \( \exists 1: \exists a. ?P b a \) using \( s \) by blast
    hence \( \exists x1: \exists a. ?P b a \) using \( s \) by blast
    hence \( \exists y1: \exists a. ?P b a \) by(blast dest:inj-onD[OF \( i \)])
  }
  note \( g = \text{this} \)
  have \( \text{inj-on } ?g B \)
  proof(rule inj-onI)
    fix \( x y \) assume \( x:B \) \( y:B \) \( ?g x = ?g y \)
    from \( s' :a:B\) obtain \( a1 \) where \( a1: ?P x a1 \) by blast
    from \( s' :y:B\) obtain \( a2 \) where \( a2: ?P y a2 \) by blast
    from \( g[\text{OF } a1] a1 \) \( g[\text{OF } a2] a2 \) \( \exists g x = ?g y \) show \( x=y \) by simp
  qed
  moreover have \( ?g' B = A \)
  proof(auto simp:image-def)
    fix \( b \) assume \( b:B \)
    with \( s \) obtain \( a \) where \( \exists P: ?P b a \) by blast
    thus \( \exists g b \in A \) using \( g[\text{OF } P] \) by auto
  next
    fix \( a \) assume \( a:A \)
    then obtain \( b \) where \( ?P b a \) by blast
    then have \( b:B \) using \( s \) by blast
    with \( g[\text{OF } P] \) show \( \exists b \in B. a = ?g b \) by blast
  qed
  ultimately show \( \exists \)thesis by(auto simp:bij-betw-def)
qed

lemma bij-betw-cong: \( (\forall a. a \in A \Rightarrow f a = g a) \Rightarrow \text{bij-betw } f A A' = \text{bij-betw } g A A' \)
unfolding bij-betw-def inj-on-def by force

lemma bij-betw-id[intro, simp]:
  \( \text{bij-betw } id A A \)
unfolding bij-betw-def id-def by auto
lemma bij-betw-id-iff:
  bij-betw id A B \iff A = B
by(auto simp add: bij-betw-def)

lemma bij-betw-combine:
  assumes bij-betw f A B bij-betw f C D B \cap D = {}
  shows bij-betw f (A \cup C) (B \cup D)
using assms unfolding bij-betw-def inj-on-Un image-Un by auto

lemma bij-betw-subset:
  assumes BIJ: bij-betw f A A' and
         SUB: B \leq A and IM: f ' B = B'
  shows bij-betw f B B'
using assms
by(unfold bij-betw-def inj-on-def, auto simp add: inj-on-def)

lemma surj-image-vimage-eq: surj f \Longrightarrow f ' (f -' A) = A
by simp

lemma surj-vimage-empty: surj f shows f -' A = {} \iff A = {} by (intro iffI) fastforce+

lemma inj-vimage-image-eq: inj f \Longrightarrow f -' (f ' A) = A
by (simp add: inj-on-def, blast)

lemma vimage-subsetD: surj f \Longrightarrow f -' B \leq A \Longrightarrow B \leq f ' A
by (blast intro: sym)

lemma vimage-subsetI: inj f \Longrightarrow B \leq f ' A \Longrightarrow f -' B \leq A
by (unfold inj-on-def, blast)

lemma vimage-subset-eq: bij f \Longrightarrow (f -' B \leq A) = (B \leq f ' A)
apply (unfold bij-def)
apply (blast del: subsetI intro: vimage-subset1 vimage-subsetD)
done

lemma inj-on-image-eq-iff: \[ inj-on f C; A \subseteq C; B \subseteq C \] \Longrightarrow f ' A = f ' B \iff A = B
by(fastforce simp add: inj-on-def)

lemma inj-on-Un-image-eq-iff: inj-on f (A \cup B) \Longrightarrow f ' A = f ' B \iff A = B
by(erule inj-on-image-eq-iff) simp-all

lemma inj-on-image-Int:
  \[ inj-on f C; A\subseteq C; B\subseteq C \] \Longrightarrow f '(A \cap B) = f 'A \cap f 'B
apply (simp add: inj-on-def, blast)
done
lemma inj-on-image-set-diff: 
\[ | \text{inj-on } f \ C; \quad A\subseteq C; \quad B\subseteq C | \implies f'(A-B) = f'A - f'B \]
apply (simp add: inj-on-def, blast)
done

lemma image-Int: inj f \implies f'(A \cap B) = f'A \cap f'B
by (simp add: inj-on-def, blast)

lemma image-set-diff: inj f \implies f'(A-B) = f'A - f'B
by (simp add: inj-on-def, blast)

lemma inj-image-mem-iff: inj f \implies (f a : A') = (a : A)
by (blast dest: injD)

lemma inj-image-subset-iff: inj f \implies (f'A \subseteq f'B) = (A \subseteq B)
by (blast dest: injD)

lemma inj-image-eq-iff: inj f \implies (f'A = f'B) = (A = B)
by (blast dest: injD)

lemma surj-Compl-image-subset: surj f \implies -f'A \subseteq f'(-A)
by auto

lemma inj-image-Compl-subset: inj f \implies f'(-A) \subseteq -f'A
by (auto simp add: inj-on-def)

lemma bij-image-Compl-eq: bij f \implies f'(-A) = -f'A
apply (simp add: bij-def)
apply (rule equalityI)
apply (simp all (no asm simp) add: inj-image-Compl-subset surj-Compl-image-subset)
done

lemma inj-vimage-singleton: inj f \implies f^{-1} \{a\} \subseteq \{x. f x = a\}
— The inverse image of a singleton under an injective function is included in a singleton.
apply (auto simp add: inj-on-def)
apply (blast intro: the-equality [symmetric])
done

lemma inj-vimage-singleton:
\[ \text{inj-on } f A \implies f^{-1} \{a\} \cap A \subseteq \{x. f x = a\} \]
by (auto simp add: inj-on-def intro: the-equality [symmetric])

lemma (in ordered-ab-group-add) inj-uminus(simp, intro): inj-on uminus A
by (auto intro: inj-onI)

lemma (in linorder) strict-mono-imp-inj-on: strict-mono f \implies inj-on f A
by (auto intro!: inj-onI dest: strict-mono-eq)
lemma bij-betw-byWitness:
assumes \( \forall a \in A. f'(f a) = a \quad \text{and} \quad \forall a' \in A'. f'(f a) = a' \quad \text{and} \quad f' : A \leq A' \quad \text{and} \quad f': A' \leq A \)
shows bij-betw \( f \) \( A \) \( A' \)
using assms
proof
  unfold bij-betw-def inj-on-def, safe
  fix \( a \ \text{and} \ b \) assume \( a \in A \) \( b \in A \) and \( f(a) = f(b) \)
  hence \( f'(f(a)) = f'(f(b)) \)
  with \( f'(f(a)) = f'(f(b)) \)
  have \( a = b \) by simp
  next
  fix \( a' \) assume \( a' \in A' \)
  hence \( f'(a') \in A' \)
  moreover
  have \( a' = f'(a') \)
  ultimately show \( a' \in f' A \)
qed

corollary notIn-Un-bij-betw:
assumes \( \exists b : f(b) \notin A' \) \( f' : A' \)
shows \( \forall a \in A. \exists a'. f'(a) = a' \)
proof
  assume \( \forall a \in A. \exists a'. f'(a) = a' \)
  hence \( \forall a \in A. \exists a'. f'(a) = a' \)
  thus \( \forall a \in A. \exists a'. f'(a) = a' \)
  using assms notIn-Un-bij-betw[of \( f \) \( A \) \( A' \)] by blast
next
assumes \( \forall a \in A. \exists a'. f'(a) = a' \)
have \( f' A = A' \)
proof(auto)
  fix \( a \)
  assume \( a \in A \)
  hence \( f(a) \in A' \)
  moreover
  { assume \( f(a) = f(b) \)
    hence \( a = b \)
    with \( \forall a \in A. \exists a'. f'(a) = a' \)
    have False by blast
  }
  ultimately show \( f(a) \in A' \)
qed
next
  fix \texttt{a'} assume \texttt{**: a' \in A'}
  hence \texttt{a' \in f'(A \cup \{b\})}
  using \texttt{* by (auto simp add: bij-betw-def)}
  then obtain \texttt{a} where \texttt{1: a \in A \cup \{b\} \land f a = a'} by blast
  moreover
  \{assume \texttt{a = b with 1 ** NIN'} have False by blast\}
  ultimately have \texttt{a' \in f' (A \cup \{b\})} using \texttt{* by (auto simp add: bij-betw-def)}
  then obtain \texttt{a} where \texttt{1: a \in A \cup \{b\} \land f a = a'} by blast
  qed

thus \texttt{bij-betw f A A'} using \texttt{* bij-betw-subset[of f A \cup \{b\} - A]} by blast
qed

8.6 Function Updating

definition \texttt{fun-upd} :: \langle\texttt{a => b} \rangle = \langle\texttt{a'} => \texttt{b'} \rangle where
\texttt{fun-upd f a b == \% x. if x=a then b else f x}

nonterminal \texttt{updbinds and updbind}

syntax
-\texttt{updbind :: ['a, 'a] => updbind ((2- :=/ -))}
  :: updbind => updbinds (-)
-\texttt{updbinds:: [updbind, updbinds] => updbinds (-/ -)}
-\texttt{Update :: ['a, updbinds] => 'a (-/(-) [1000, 0] 900)}

translations
-\texttt{Update f (-updbinds b bs) == -Update (-Update f b) bs}
\texttt{f(x:=y) == CONST fun-upd f x y}

lemma \texttt{fun-upd-idem-iff}: \texttt{(f(x:=y) = f) = (f x = y)}
apply (simp add: fun-upd-def, safe)
apply (erule subst)
apply (rule-tac [2] ext, auto)
done

lemma \texttt{fun-upd-idem}: \texttt{f x = y ==> f(x:=y) = f}
by (simp only: fun-upd-idem-iff)

lemma \texttt{fun-upd-triv [iff]}: \texttt{f(x := f x) = f}
by (simp only: fun-upd-idem)

lemma \texttt{fun-upd-apply [simp]}: \texttt{(f(x:=y))z = (if z=x then y else f z)}
by (simp add: fun-upd-def)
lemma fun-upd-same: \((f(x:=y)) x = y\)
by simp

lemma fun-upd-other: \(z \sim x \Longrightarrow (f(x:=y)) z = f z\)
by simp

lemma fun-upd-upd \[simp\]:
\(f(x:=y, x:=z) = f(x:=z)\)
bwSIMP: fun-eq-iff

lemma fun-upd-twist: \(a \sim c \Longrightarrow (m(a:=b))(c:=d) = (m(c:=d))(a:=b)\)
by (rule ext, auto)

lemma inj-on-fun-updI:
\(\text{inj-on } f \ A \Longrightarrow y/\in f ' A \Longrightarrow \text{inj-on } (f(x:=y)) \ A\)
bwSIMP: inj-on-def

lemma fun-upd-image:
\(f(x:=y) ' A = (if x \in A then insert y (f ' (A\-\{x\})) else f ' A)\)
bwauto

lemma fun-upd-comp:
\(f \circ (g(x:=y)) = (f \circ g)(x:=f y)\)
bwauto

8.7 override-on

definition override-on :: \('a \Rightarrow 'b \Rightarrow 'a set \Rightarrow 'a \Rightarrow 'b\ where 
\(\text{override-on } f g A = (\lambda a. \text{ if } a \in A \text{ then } g a \text{ else } f a)\)

lemma override-on-emptyset\[simp\]: override-on \(f g \{\} = f\)
bw(simp add:override-on-def)

lemma override-on-notin\[simp\]: \(a \sim: A \Longrightarrow (\text{override-on } f g A) a = f a\)
bw(simp add:override-on-def)

lemma override-on-in\[simp\]: \(a : A \Longrightarrow (\text{override-on } f g A) a = g a\)
bw(simp add:override-on-def)

8.8 swap

definition swap :: \('a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b\ where 
\(\text{swap } a : b f = f (a := f b, b := f a)\)

lemma swap-apply \[simp\]:
\(\text{swap } a : b f a = f b\)
\(\text{swap } a : b f b = f a\)
\(c \neq a \Longrightarrow c \neq b \Longrightarrow \text{swap } a : b c = f c\)
bw(simp-all add: swap-def)

lemma swap-self \[simp\]:
THEORY "Fun"

```plaintext
swap a a f = f
by (simp add: swap-def)

lemma swap-commute:
swap a b f = swap b a f
by (simp add: fun-upd-def swap-def fun-eq-iff)

lemma swap-nilpotent [simp]:
swap a b (swap a b f) = f
by (rule ext, simp add: fun-upd-def swap-def)

lemma swap-comp-involutory [simp]:
swap a b ◦ swap a b = id
by (rule ext, simp)

lemma swap-triple:
assumes a ≠ c and b ≠ c
shows swap a b (swap b c (swap a b f)) = swap a c f
using assms by (simp add: fun-eq-iff swap-def)

lemma comp-swap:
f ◦ swap a b g = swap a b (f ◦ g)
by (rule ext, simp add: fun-upd-def swap-def)

lemma swap-image-eq [simp]:
assumes a ∈ A b ∈ A
shows swap a b f ' A = f ' A
proof
  have subset: ∃ f. swap a b f ' A ⊆ f ' A
    using assms by (auto simp: image_iff swap-def)
  then have swap a b (swap a b f) ' A ⊆ (swap a b f) ' A
    with subset[of f] show ?thesis by auto
qed

lemma inj-on-imp-inj-on-swap:
[inj-on f A; a ∈ A; b ∈ A] ⇒ inj-on (swap a b f) A
by (simp add: inj-on-def swap-def, blast)

lemma inj-on-swap-iff [simp]:
assumes A: a ∈ A b ∈ A shows inj-on (swap a b f) A ⇔ inj-on f A
proof
  assume inj-on (swap a b f) A
  with A have inj-on (swap a b (swap a b f)) A
    by (iprover intro: inj-on-imp-inj-on-swap)
  thus inj-on f A by simp
next
  assume inj-on f A
  with A show inj-on (swap a b f) A by (iprover intro: inj-on-imp-inj-on-swap)
qed

lemma surj-imp-surj-swap: surj f ⇒ surj (swap a b f)
```
by simp

lemma surj-swap-iff [simp]: surj (swap a b f) ↔ surj f  
  by simp

lemma bij-betw-swap-iff [simp]: 
  [ x ∈ A; y ∈ A ] → bij-betw (swap x y f) A B ↔ bij-betw f A B  
  by (auto simp: bij-betw-def)

lemma bij-swap-iff [simp]: bij (swap a b f) ↔ bij f  
  by simp

hide-const (open) swap

8.9 Inversion of injective functions

definition the-inv-into :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a) where  
  the-inv-into A f = %x. THE y. y : A & f y = x

lemma the-inv-into-f-f:
  inj-on f A; x : A ] ===> the-inv-into A f (f x) = x
  apply (simp add: the-inv-into-def inj-on-def)
  apply blast
  done

lemma f-the-inv-into-f:
  inj-on f A ===> y : f' A ===> f (the-inv-into A f y) = y
  apply (simp add: the-inv-into-def)
  apply (rule the1I2)
  apply(blast dest: inj-onD)
  apply blast
  done

lemma the-inv-into-into:
  inj-on f A; x : f' A; A <= B ] ===> the-inv-into A f x : B
  apply (simp add: the-inv-into-def)
  apply (rule the1I2)
  apply(blast dest: inj-onD)
  apply blast
  done

lemma the-inv-into-onto[simp]:
  inj-on f A ===> the-inv-into A f' (f' A) = A
  by (fast intro:the-inv-into-into-the-inv-into-f-f[symmetric])

lemma the-inv-into-f-eq:
  inj-on f A; f x = y; x : A ] ===> the-inv-into A f y = x
  apply (erule subst)
  apply (erule the-inv-into-f-f, assumption)
done

lemma the-inv-into-comp:
|| inj-on f (g ' A); inj-on g A; x : f ' g ' A || ==>
the-inv-into A (f o g) x = (the-inv-into A g o the-inv-into (g ' A) f) x
apply (rule the-inv-into-f-eq)
apply (fast intro: comp-inj-on)
apply (simp add: f-the-inv-into-f the-inv-into-into)
done

lemma inj-on-the-inv-into:
inj-on f A == inj-on (the-inv-into A f) (f ' A)
by (auto intro: inj-onI simp: the-inv-into-f-f)

lemma bij-betw-the-inv-into:
bij-betw f A B == bij-betw (the-inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into)

abbreviation the-inv :: ('a => 'b) => ('b => 'a) where
the-inv f == the-inv-into UNIV f

lemma the-inv-f-f:
assumes inj f
shows the-inv f (f x) = x using assms UNIV-I
by (rule the-inv-into-f-f)

8.10 Cantor’s Paradox

lemma Cantors-paradox:
~(∃ f. f ' A = Pow A)
proof clarify
fix f assume f ' A = Pow A hence *: Pow A ≤ f ' A by blast
let ?X = { a ∈ A. a /∈ f a } 
have *: ?X ∈ Pow A unfolding Pow-def by auto
with *: obtain x where x ∈ A ∧ f x = ?X by blast 
thus False by best
qed

8.11 Setup

8.11.1 Proof tools

simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...)
simproc-setup fun-upd2 (f(v := w, x := y)) = (f n - =>
let
fun gen-fun-upd NONE T - - = NONE
| gen-fun-upd (SOME f) T x y = SOME (Const (@{const-name fun-upd}, T)
$ f $ x $ y)
fun dest-fun-T1 (Type (_, T :: Ts)) = T

fun find-double (t as Const (@{const-name fun-upd}, T) $ f $ x $ y) =
  let
    fun find (Const (@{const-name fun-upd}, T) $ g $ v $ w) =
      if v aconv x then SOME g else gen-fun-upd (find g) T v w
    in
    dest-fun-T1 T, gen-fun-upd (find f) T x y end

val ss = simpset_of @{context}

fun proc ctxt ct =
  let
    val t = Thm.term_of ct
  in
    case find-double t of
      (T, NONE) => NONE
    | (T, SOME rhs) =>
      SOME (Goal.prove ctxt [] [] (Logic.mk_equals (t, rhs)))
        (fn _ => rtac eq-reflection 1 THEN rtac @{thm ext} 1 THEN simp-tac (put-simpset ss ctxt) 1))
  end
in proc end

8.11.2 Functorial structure of types

ML-file Tools/functor.ML

functor map-fun: map-fun
  by (simp-all add: fun-eq-iff)

functor vimage
  by (simp-all add: fun-eq-iff vimage-comp)

Legacy theorem names

lemmas o-def = comp-def
lemmas o-apply = comp-apply
lemmas o-assoc = comp-assoc [symmetric]
lemmas id-o = id-comp
lemmas o-id = comp-id
lemmas o-eq-dest = comp-eq-dest
lemmas o-eq-elim = comp-eq-elim
lemmas o-eq-dest-lhs = comp-eq-dest-lhs
lemmas o-eq-id-dest = comp-eq-id-dest

end
9 Complete-Lattices: Complete lattices

theory Complete-Lattices
imports Fun
begin

notation
less-eq (infix ⊑ 50) and
less (infix ⊏ 50)

9.1 Syntactic infimum and supremum operations

class Inf =
fixes Inf :: 'a set ⇒ 'a (⨍_ [900] 900)
begin

definition INFIMUM :: 'b set ⇒ ('b ⇒ 'a) ⇒ 'a where
INF-def: INFIMUM A f = ∏(f ' A)

lemma Inf-image-eq [simp]:
asurer (f ' A) = INFIMUM A f
by (simp add: INF-def)

lemma INF-image [simp]:
INFIMUM (f ' A) g = INFIMUM A (g ◦ f)
by (simp only: INF-def image-comp)

lemma INF-identity-eq [simp]:
INFIMUM A (λx. x) = ∏A
by (simp add: INF-def)

lemma INF-id-eq [simp]:
INFIMUM A id = ∏A
by (simp add: id-def)

lemma INF-cong:
A = B ⇒ (∀x. x ∈ B ⇒ C x = D x) ⇒ INFIMUM A C = INFIMUM B D
by (simp add: INF-def image-def)

lemma strong-INF-cong [cong]:
A = B ⇒ (∀x. x ∈ B =simp=> C x = D x) ⇒ INFIMUM A C = INFIMUM B D
unfolding simp-implies-def by (fact INF-cong)

end

class Sup =
fixes Sup :: 'a set ⇒ 'a (⨌_ [900] 900)
begin
definition SUPREMUM :: 'b set ⇒ ('b ⇒ 'a) ⇒ 'a where
SUP-def: SUPREMUM A f = ⨆(f • A)

lemma Sup-image-eq [simp]:
⨆(f • A) = SUPREMUM A f
by (simp add: SUP-def)

lemma SUP-image [simp]:
SUPREMUM (f • A) g = SUPREMUM A (g ∘ f)
by (simp only: SUP-def image-comp)

lemma SUP-identity-eq [simp]:
SUPREMUM A (λx. x) = ⨆A
by (simp add: SUP-def)

lemma SUP-id-eq [simp]:
SUPREMUM A id = ⨆A
by (simp add: id-def)

lemma SUP-cong:
A = B ⇒ (∀x. x ∈ B ⇒ C x = D x) ⇒ SUPREMUM A C = SUPREMUM B D
by (simp add: SUP-def image-def)

lemma strong-SUP-cong [cong]:
A = B ⇒ (∀x. x ∈ B =simp⇒ C x = D x) ⇒ SUPREMUM A C = SUPREMUM B D
unfolding simp-implies-def by (fact SUP-cong)

end

Note: must use names INFIMUM and SUPREMUM here instead of INF and SUP to allow the following syntax coexist with the plain constant names.

syntax
-INF1 :: pattrns ⇒ 'b ⇒ 'b
-INF :: pattrn ⇒ 'a set ⇒ 'b ⇒ 'b
-SUP1 :: pattrns ⇒ 'b ⇒ 'b
-SUP :: pattrn ⇒ 'a set ⇒ 'b ⇒ 'b

translations
INF x y. B == INF x. INF y. B
INF x. B == CONST INFIMUM CONST UNIV (%x. B)
INF x. B == INF x:CONST UNIV. B
9.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into the following type class, along with assumptions that define bottom and top in terms of infimum and supremum.

```plaintext
class complete-lattice = lattice + Inf + Sup + bot + top +
  assumes Inf-lower: x ∈ A ⇒ \bigcap A ⊑ x
  and Inf-greatest: (∀ x. x ∈ A ⇒ z ⊑ x) ⇒ z ⊑ \bigcap A
  assumes Sup-upper: x ∈ A ⇒ x ⊑ \bigcup A
  and Sup-least: (∀ x. x ∈ A ⇒ x ⊑ z) ⇒ \bigcup A ⊑ z
  assumes Inf-empty [simp]: ∅ ⊑ T
  assumes Sup-empty [simp]: \bigcup \{\} = ⊥
begin
subclass bounded-lattice
proof
  fix a
  show ⊥ ≤ a by (auto intro: Sup-least simp only: Sup-empty [symmetric])
  show a ≤ ⊤ by (auto intro: Inf-greatest simp only: Inf-empty [symmetric])
qed

lemma dual-complete-lattice:
  class.complete-lattice Sup Inf sup (op ≥) (op >) inf ⊤ ⊥
  by (auto intro!: class.complete-lattice.intro dual-lattice)
  (unfold-locales, (fact Inf-empty Sup-empty
                 Sup-upper Sup-least Inf-lower Inf-greatest)+)
end
```

context complete-lattice

begin

lemma INF-foundation-dual:
  Sup.SUPREMUM Inf = INFIMUM
  by (simp add: fun-eq-iff Sup.SUP-def)
lemma SUP-foundation-dual:
Inf.INFIMUM Sup = SUPREMUM
by (simp add: fun-eq-iff Inf.INF-def)

lemma Sup-eqI:
(∀ y ∈ A ⇒ y ≤ x) ⇒ (∀ y. (∀ z ∈ A ⇒ z ≤ y) ⇒ x ≤ y) ⇒ ⋃ A = x
by (blast intro: antisym Sup-least Sup-upper)

lemma INF-eqI:
(∀ i ∈ A ⇒ x ≤ i) ⇒ (∀ y. (∀ i ∈ A ⇒ y ≤ i) ⇒ y ≤ x) ⇒ ⋂ A = x
by (blast intro: antisym Inf-greatest Inf-lower)

lemma INF-lower:
i ∈ A ⇒ (⋂ i∈A. f i) ⊆ f i
using INF-lower [of i A f] by auto

lemma INF-upper:
u ∈ A ⇒ f i ⊆ u ⇒ (⋃ i∈A. f i) ⊆ u
using INF-upper [of u A] by auto

lemma le-Inf-iff:
b ⊆ ⋂ A ←→ (∀ a ∈ A. b ⊆ a)
by (auto intro: Inf-greatest dest: Inf-lower)
lemma \textit{le-INF-iff} : \(u \sqsubseteq (\bigsqcap i \in A. f i) \iff (\forall i \in A. u \sqsubseteq f i)\)
using \textit{le-Inf-iff \[of \ f \ A\]} by simp

lemma \textit{Sup-le-iff} : \(\bigvee A \sqsubseteq b \iff (\forall a \in A. a \sqsubseteq b)\)
by (auto intro: Sup-least dest: Sup-upper)

lemma \textit{SUP-le-iff} : \((\bigsqcup i \in A. f i) \sqsubseteq u \iff (\forall i \in A. f i \sqsubseteq u)\)
using \textit{Sup-le-iff \[of f \ A\]} by simp

lemma \textit{Inf-insert} [simp] : \(\bigsqcap\{a\in A. f x\} = f a \sqcap \text{INFIMUM } A\ f\)
unfolding \textit{INF-def Inf-insert} by simp

lemma \textit{Sup-insert} [simp] : \(\bigvee\{a\in A. f x\} = f a \sqcup \text{SUPREMUM } A\ f\)
unfolding \textit{SUP-def Sup-insert} by simp

lemma \textit{INF-empty} [simp] : \(\bigsqcap\{x\in\{\}. f x\} = \top\)
by (simp add: INF-def)

lemma \textit{SUP-empty} [simp] : \(\bigvee\{x\in\{\}. f x\} = \bot\)
by (simp add: SUP-def)

lemma \textit{Inf-UNIV} [simp] :
\(\bigsqcap UNIV = \bot\)
by (auto intro!: antisym Inf-lower)

lemma \textit{Sup-UNIV} [simp] :
\(\bigvee UNIV = \top\)
by (auto intro!: antisym Sup-upper)

lemma \textit{Inf-Sup} : \(\bigsqcap A = \bigsqcap\{b. \forall a \in A. b \sqsubseteq a\}\)
by (auto intro: antisym Inf-greatest Sup-upper Sup-least)

lemma \textit{Sup-Inf} : \(\bigvee A = \bigvee\{b. \forall a \in A. a \sqsubseteq b\}\)
by (auto intro: antisym Inf-lower Inf-greatest Sup-upper Sup-least)

lemma \textit{Inf-superset-mono} : \(B \subseteq A \implies \bigsqcap A \sqsubseteq \bigsqcap B\)
by (auto intro: Inf-greatest Inf-lower)

lemma \textit{Sup-subset-mono} : \(A \subseteq B \implies \bigvee A \subseteq \bigvee B\)
by (auto intro: Sup-least Sup-upper)

lemma \textit{Inf-mono}:
assumes $\bigwedge b, b \in B \implies \exists a \in A. a \subseteq b$
shows $\bigcap A \subseteq \bigcap B$
proof (rule Inf-greatest)
  fix $b$ assume $b \in B$
  with assms obtain $a$ where $a \in A$ and $a \subseteq b$ by blast
  from $a \in A$ have $\bigcap A \subseteq a$ by (rule Inf-lower)
  with $a \subseteq b$ show $\bigcap A \subseteq b$ by auto
qed

lemma INF-mono:
\[ (\forall m. m \in B \implies \exists n \in A. f n \subseteq g m) \implies (\bigcap n \in A. f n) \subseteq (\bigcap n \in B. g n) \]
using Inf-mono [of $f \cdot A$ $f \cdot B$] by auto

lemma Sup-mono:
assumes $\bigwedge a, a \in A \implies \exists b \in B. a \subseteq b$
shows $\bigcup A \subseteq \bigcup B$
proof (rule Sup-least)
  fix $a$ assume $a \in A$
  with assms obtain $b$ where $b \in B$ and $a \subseteq b$ by blast
  from $b \in B$ have $\bigcup B \subseteq b$ by (rule Sup-upper)
  with $a \subseteq b$ show $a \subseteq \bigcup B$ by auto
qed

lemma SUP-mono:
\[ (\forall n. n \in A \implies \exists m \in B. f n \subseteq g m) \implies (\bigcup n \in A. f n) \subseteq (\bigcup n \in B. g n) \]
using Sup-mono [of $f \cdot A$ $g \cdot B$] by auto

lemma INF-superset-mono:
$B \subseteq A \implies (\forall x. x \in B \implies f x \subseteq g x) \implies (\bigcap x \in A. f x) \subseteq (\bigcap x \in B. g x)$$—$ The last inclusion is POSITIVE!
by (blast intro: INF-mono dest: subsetD)

lemma SUP-subset-mono:
$A \subseteq B \implies (\forall x. x \in A \implies f x \subseteq g x) \implies (\bigcup x \in A. f x) \subseteq (\bigcup x \in B. g x)$$by (blast intro: SUP-mono dest: subsetD)

lemma Inf-less-eq:
assumes $\bigwedge v, v \in A \implies v \subseteq u$
\ and $A \neq \{\}$
shows $\bigcap A \subseteq u$
proof
  from $A \neq \{\}$ obtain $v$ where $v \in A$ by blast
  moreover from $v \in A$ assms(1) have $v \subseteq u$ by blast
  ultimately show $\neg$thesis by (rule Inf-lower2)
qed

lemma less-eq-Sup:
assumes $\bigwedge v, v \in A \implies u \subseteq v$
\ and $A \neq \{\}$
shows $u \subseteq \bigsqcup A$

proof –

from $\langle A \neq \emptyset \rangle$ obtain $v$ where $v \in A$ by blast

moreover from $\langle v \in A \rangle$ have $u \sqsubseteq v$ by blast

ultimately show ?thesis by (rule Sup-upper2)

qed

lemma SUP-eq:

assumes $\textstyle \bigwedge i. i \in A \Longrightarrow \exists j \in B. f_i \leq g_j$

assumes $\textstyle \bigwedge j. j \in B \Longrightarrow \exists i \in A. g_j \leq f_i$

shows $\bigcap i \in A. f_i = \bigcap j \in B. g_j$

by (intro antisym SUP-least) (blast intro: SUP-upper2 dest: assms)+

lemma INF-eq:

assumes $\textstyle \bigwedge i. i \in A \Longrightarrow \exists j \in B. f_i \geq g_j$

assumes $\textstyle \bigwedge j. j \in B \Longrightarrow \exists i \in A. g_j \geq f_i$

shows $\bigcap i \in A. f_i \leq \bigcap j \in B. g_j$

by (intro antisym INF-greatest) (blast intro: INF-lower2 dest: assms)+
qed

lemma \textit{Inf-top-conv} \([\text{simp}]:\)
\[
\prod A = \top \iff (\forall x \in A. \ x = \top)
\]
\[
\top = \prod A \iff (\forall x \in A. \ x = \top)
\]
proof
  show \(\prod A = \top \iff (\forall x \in A. \ x = \top)\)
  proof
    assume \(\forall x \in A. \ x = \top\)
    then have \(A = \{\} \lor A = \{\top\}\) by auto
    then show \(\prod A = \top\) by auto
  next
    assume \(\prod A = \top\)
    show \(\forall x \in A. \ x = \top\)
    proof
      (rule ccontr)
      assume \(\neg (\forall x \in A. \ x = \top)\)
      then obtain \(x\) where \(x \in A\) and \(x \neq \top\) by blast
      then obtain \(B\) where \(A = \text{insert} \ x B\) by blast
      with \(\prod A = \top\) \(\langle x \neq \top\rangle\) show False by simp
  qed
  qed
then show \(\top = \prod A \iff (\forall x \in A. \ x = \top)\) by auto
qed

lemma \textit{INF-top-conv} \([\text{simp}]:\)
\[
(\prod x \in A. \ B x) = \top \iff (\forall x \in A. \ B x = \top)
\]
\[
\top = (\prod x \in A. \ B x) \iff (\forall x \in A. \ B x = \top)
\]
using \textit{Inf-top-conv} \([\text{of} \ B ' \ A]\) by simp-all

lemma \textit{Sup-bot-conv} \([\text{simp}]:\)
\[
\bigsqcup A = \bot \iff (\forall x \in A. \ x = \bot)
\]
\[
\bot = \bigsqcup A \iff (\forall x \in A. \ x = \bot)
\]
using \textit{dual-complete-lattice}
by (rule complete-lattice.\textit{Inf-top-conv})+

lemma \textit{SUP-bot-conv} \([\text{simp}]:\)
\[
(\bigsqcup x \in A. \ B x) = \bot \iff (\forall x \in A. \ B x = \bot)
\]
\[
\bot = (\bigsqcup x \in A. \ B x) \iff (\forall x \in A. \ B x = \bot)
\]
using \textit{Sup-bot-conv} \([\text{of} \ B ' \ A]\) by simp-all

lemma \textit{INF-const} \([\text{simp}]:\) \(A \neq \{\} \implies (\prod i \in A. \ f) = f\)
by (auto intro: antisym \textit{INF-lower} \textit{INF-greatest})

lemma \textit{SUP-const} \([\text{simp}]:\) \(A \neq \{\} \implies (\bigsqcup i \in A. \ f) = f\)
by (auto intro: antisym \textit{SUP-upper} \textit{SUP-least})

lemma \textit{INF-top} \([\text{simp}]:\) \((\prod x \in A. \ \top) = \top\)
by (cases \(A = \{\})\) simp-all
lemma SUP-bot [simp]: \( \bigcup_{x \in A. \bot} = \bot \)
  by (cases A = \{\}) simp-all

lemma INF-commute: \( \bigcap_{i \in A. \bigcap_j B. f \ i \ j} = (\bigcap_j B. \bigcap_{i \in A. f \ i \ j}) \)
  by (iprover intro: INF-upper INF-least order-trans antisym)

lemma SUP-commute: \( \bigcup_{i \in A. \bigcup_j B. f \ i \ j} = (\bigcup_j B. \bigcup_{i \in A. f \ i \ j}) \)
  by (iprover intro: SUP-upper SUP-least order-trans antisym)

lemma INF-absorb:
  assumes \( k \in I \)
  shows \( A \ k \sqcap (\bigcap_{i \in I. A \ i}) = (\bigcap_{i \in I. A \ i}) \)
proof (note \( \langle y < (\bigcap_{i \in A. f \ i}) \rangle \)
  also have \( (\bigcap_{i \in A. f \ i}) \leq f \ i \) using \( \langle i \in A \rangle \)
  by (rule INF-lower)
finally show \( y < f \ i \).

lemma SUP-absorb:
  assumes \( k \in I \)
  shows \( A \ k \sqcup (\bigcup_{i \in I. A \ i}) = (\bigcup_{i \in I. A \ i}) \)
proof (note \( \langle y < (\bigcap_{i \in A. f \ i}) \rangle \)
  also have \( (\bigcap_{i \in A. f \ i}) \leq f \ i \) using \( \langle i \in A \rangle \)
  by (rule INF-lower)
finally show \( y < f \ i \).

qed

lemma SUP-lessD:
  assumes \(( \bigsqcup i \in A. f i ) < y \) \( i \in A \) shows \( f i < y \)
proof
  have \( f i \leq ( \bigsqcup i \in A. f i ) \) using \(( i \in A)\) by (rule SUP-upper)
  also note \(( \bigsqcup i \in A. f i ) < y \)
  finally show \( f i < y \).
qed

lemma INF-UNIV-bool-expand:
\(( \prod b. A b ) = A \text{ True } \cap A \text{ False}\)
by (simp add: UNIV-bool inf-commute)

lemma SUP-UNIV-bool-expand:
\(( \bigsqcup b. A b ) = A \text{ True } \cup A \text{ False}\)
by (simp add: UNIV-bool sup-commute)

lemma Inf-le-Sup:
\( A \neq \{} \Rightarrow \text{Inf } A \leq \text{Sup } A\)
by (blast intro: Sup-upper2 Inf-lower ex-in-conv)

lemma INF-le-SUP:
\( A \neq \{} \Rightarrow \text{INFIMUM } A f \leq \text{SUPREMUM } A f\)
using Inf-le-Sup \([of f ' A]\) by simp

lemma INF-eq-const:
\( I \neq \{} \Rightarrow ( \forall i. i \in I \Rightarrow f i = x ) \Rightarrow \text{INFIMUM } I f = x\)
by (auto intro: INF-eqI)

lemma SUP-eq-const:
\( I \neq \{} \Rightarrow ( \forall i. i \in I \Rightarrow f i = x ) \Rightarrow \text{SUPREMUM } I f = x\)
by (auto intro: SUP-eqI)

lemma INF-eq-iff:
\( I \neq \{} \Rightarrow ( \forall i. i \in I \Rightarrow f i \leq c ) \Rightarrow ( \text{INFIMUM } I f = c ) \iff ( \forall i \in I. f i = c )\)
using INF-eq-const \([of f c]\) INF-lower \([of - I f]\)
by (auto intro: antisym cong del: strong-INF-cong)

lemma SUP-eq-iff:
\( I \neq \{} \Rightarrow ( \forall i. i \in I \Rightarrow c \leq f i ) \Rightarrow ( \text{SUPREMUM } I f = c ) \iff ( \forall i \in I. f i = c )\)
using SUP-eq-const \([of f c]\) SUP-upper \([of - I f]\)
by (auto intro: antisym cong del: strong-SUP-cong)

end

class complete-distrib-lattice = complete-lattice +
  assumes sap-Inf: \( a \sqcup \prod B = ( \prod b \in B. a \sqcup b)\)
assumes \( \text{inf-Sup}: \ a \cap \bigsqcup B = (\bigsqcup b \in B. \ a \cap b) \)

\begin{verbatim}
lemma sup-INF:
  \( a \sqcup (\bigsqcap b \in B. \ f(b)) = (\bigsqcap b \in B. \ a \sqcup f(b)) \)
  by (simp only: INF-def sup-Inf image-image)

lemma inf-SUP:
  \( a \sqcap (\bigsqcup b \in B. \ f(b)) = (\bigsqcup b \in B. \ a \sqcap f(b)) \)
  by (simp only: SUP-def inf-Sup image-image)

lemma dual-complete-distrib-lattice:
  class complete-distrib-lattice Sup Inf sup (op ≥) (op >) inf ⊤⊥
  apply (rule class.complete-distrib-lattice).intro)
  apply (fact dual-complete-lattice)
  apply (rule class.complete-distrib-lattice-axioms.intro)
  apply (simp-all only: INF-foundation-dual SUP-foundation-dual inf-Sup sup-Inf)
  done

subclass distrib-lattice proof
  fix a b c
  from sup-Inf have a ⊔ \( \bigsqcap \{b, c\} = (\bigsqcap d \in \{b, c\}. \ a \sqcup d) \).
  then show a ⊔ b ∩ c = (a ⊔ b) ∩ (a ∪ c) by (simp add: INF-def)
  qed

lemma Inf-sup:
  \( \bigsqcap B \sqcup a = (\bigsqcap b \in B. \ b \sqcup a) \)
  by (simp add: sup-Inf sup-commute)

lemma Sup-inf:
  \( \bigsqcup B \sqcap a = (\bigsqcup b \in B. \ b \sqcap a) \)
  by (simp add: inf-Sup inf-commute)

lemma INF-sup:
  \( (\bigsqcap b \in B. \ f(b)) \sqcup a = (\bigsqcap b \in B. \ f(b) \sqcup a) \)
  by (simp add: sup-INF sup-commute)

lemma SUP-inf:
  \( (\bigsqcup b \in B. \ f(b)) \sqcap a = (\bigsqcup b \in B. \ f(b) \sqcap a) \)
  by (simp add: inf-SUP inf-commute)

lemma Inf-sup-eq-top-iff:
  \( \bigsqcap B \sqcup a = \top \) ☐ \( \forall b \in B. \ b \sqcup a = \top \)
  by (simp only: Inf-sup INF-top-conv)

lemma Sup-inf-eq-bot-iff:
  \( \bigsqcup B \sqcap a = \bot \) ☐ \( \forall b \in B. \ b \sqcap a = \bot \)
  by (simp only: Sup-inf SUP-bot-conv)
\end{verbatim}
lemma INF-sup-distrib2:
\((\bigsqcup a \in A. f a) \sqcup (\bigsqcup b \in B. g b) = (\bigsqcup a \in A. \bigsqcup b \in B. f a \sqcup g b)\)
by (subst INF-commute) (simp add: sup-INF INF-sup)

lemma SUP-inf-distrib2:
\((\bigmeet a \in A. f a) \sqcap (\bigsqcap b \in B. g b) = (\bigsqcup a \in A. \bigmeet b \in B. f a \sqcap g b)\)
by (subst SUP-commute) (simp add: inf-SUP SUP-inf)

context
  fixes f :: 'a ⇒ 'b::complete-lattice
begin

lemma mono-Inf:
  shows \(f (\bigsqcap A) \leq (\bigsqcap x \in A. f x)\)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.INF-greatest Inf-lower dest: monoD)

lemma mono-Sup:
  shows \((\bigsqcup x \in A. f x) \leq f (\bigsqcup A)\)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.SUP-least Sup-upper dest: monoD)

end

end

class complete-boolean-algebra = boolean-algebra + complete-distrib-lattice
begin

lemma dual-complete-boolean-algebra:
  class.complete-boolean-algebra Sup Inf sup (op ≥) (op >) inf ⊤⊥ (λx y. x ⊔ − y) uminus
by (rule class.complete-boolean-algebra:intro, rule dual-complete-distrib-lattice, rule dual-boolean-algebra)

lemma uminus-Inf:
  − (\bigsqcap A) = \bigsqcup (uminus ' A)
proof (rule antisym)
  show − (\bigsqcap A) ≤ \bigsqcup (uminus ' A)
  by (rule compl-le-swap2, rule Inf-greatest, rule compl-le-swap2, rule Sup-upper)
simp
  show \bigsqcup (uminus ' A) ≤ − (\bigsqcap A)
  by (rule Sup-least, rule compl-le-swap1, rule Inf-lower) auto
qed

lemma uminus-INF:
  − (\bigsqcap x \in A. B x) = (\bigsqcup x \in A. − B x)
by (simp only: INF-def SUP-def uminus-Inf image-image)
lemma uminus-Sup:
\( - (\bigsqcup A) = \bigcap (uminus \cdot A) \)
proof
  have \( \bigsqcup A = - \bigcap (uminus \cdot A) \) by (simp add: image-image uminus-INF)
  then show \( \text{thesis} \) by simp
qed

lemma uminus-SUP:
\( - (\bigsqcup x \in A. B x) = (\bigcap x \in A. - B x) \)
by (simp only: INF-def SUP-def uminus-Sup image-image)

end

class complete-linorder = linorder + complete-lattice
begin

lemma dual-complete-linorder:
  class complete-linorder Sup Inf sup (op \( \geq \)) (op \( > \)) inf \( \top \) \( \bot \)
by (rule class.complete-linorder.intro, rule dual-complete-lattice, rule dual-linorder)

lemma complete-linorder-inf-min:
\( \inf = \min \)
by (auto intro: antisym simp add: min-def fun-eq-iff)

lemma complete-linorder-sup-max:
\( \sup = \max \)
by (auto intro: antisym simp add: max-def fun-eq-iff)

lemma Inf-less-iff:
\( \prod S \sqsubset a \iff (\exists x \in S. x \sqsubset a) \)
unfolding not-le [symmetric] le-Inf-iff by auto

lemma INF-less-iff:
(\( \prod i \in A. f i \) \( \sqsubset a \) \( \iff (\exists x \in A. f x \sqsubset a) \)
using Inf-less-iff [of f ' A] by simp

lemma less-Sup iff:
\( a \sqsubset \bigsqcup S \iff (\exists x \in S. a \sqsubset x) \)
unfolding not-le [symmetric] Sup-le-iff by auto

lemma less-SUP iff:
\( a \sqsubset (\bigsqcup i \in A. f i) \iff (\exists x \in A. a \sqsubset f x) \)
using less-Sup iff [of - f ' A] by simp

lemma Sup-eq-top iff [simp]:
\( \bigsqcup A = \top \iff (\forall x < \top. \exists i \in A. x < i) \)
proof
  assume \( *: \bigsqcup A = \top \)
  show \( (\forall x < \top. \exists i \in A. x < i) \) unfolding * [symmetric]
  proof (intro allI impl)
    fix \( x \) assume \( x < \bigsqcup A \) then show \( \exists i \in A. x < i \)
    unfolding less-Sup iff by auto
  qed
assumption *: \( \forall x < \top. \exists i \in A. x < i \)

show \( \bigsqcup A = \top \)

proof (rule ccontr)

assume \( \bigsqcup A \neq \top \)

with top-greatest [of \( \bigsqcup A \)]

have \( \bigsqcup A < \top \) unfolding le-less by auto

then have \( \bigsqcup A < \bigsqcup A \)

using * unfolding less-Sup-iff by auto

then show False by auto

qed

lemma SUP-eq-top-iff [simp]:

\( \bigsqcup (i \in A. f i) = \top \rightleftharpoons (\forall x < \top. \exists i \in A. x < f i) \)

using Sup-eq-top-iff [of f ' A] by simp

lemma INF-eq-bot-iff [simp]:

\( \bigcap A = \bot \rightleftharpoons (\forall x > \bot. \exists i \in A. i < x) \)

using dual-complete-linorder by (rule complete-linorder.Sup-eq-top-iff)

lemma INF-le-iff: \( \bigcap A \leq x \rightleftharpoons (\forall y > x. \exists a \in A. y > a) \)

proof safe

fix y assume \( x \geq \bigcap A \)

then have \( y > \bigcap A \) by auto

then show \( \exists a \in A. y > a \)

unfolding Inf-less-iff .

qed (auto elim!: allE[of - \bigcap A] simp add: not-le[symmetric] Inf-lower)

lemma INF-le-iff:

\( \infimum A f \leq x \rightleftharpoons (\forall x > x. \exists i \in A. y > f i) \)

using Inf-le-iff [of f ' A] by simp

lemma le-Sup-iff: \( x \leq \bigsqcup A \rightleftharpoons (\forall y < x. \exists a \in A. y < a) \)

proof safe

fix y assume \( x \leq \bigsqcup A \)

then have \( y < \bigsqcup A \) by auto

then show \( \exists a \in A. y < a \)

unfolding less-Sup-iff .

qed (auto elim!: allE[of - \bigsqcup A] simp add: not-le[symmetric] Sup-upper)

lemma le-SUP-iff: \( x \leq \supremum A f \rightleftharpoons (\forall y < x. \exists i \in A. y < f i) \)

using le-Sup-iff [of f ' A] by simp
subclass complete-distrib-lattice
proof
  fix a and B
  show a \sqcup \bigcap B = (\bigcap b \in B. a \sqcup b) and a \sqcap \bigcup B = (\bigcup b \in B. a \sqcap b)
  by (safe intro: INF-eqI [symmetric] sup-mono Inf-lower SUP-eqI [symmetric] inf-mono Sup-upper)
  (auto simp: not-less [symmetric] Inf-less-iff less-Sup-iff le-max-iff-disj complete-linorder-sup-max min-le-iff-disj complete-linorder-inf-min)
qed

end

9.3 Complete lattice on bool

instantiation bool :: complete-lattice begin

definition [simp, code]: \bigcap A \longleftrightarrow \text{False} \notin A

definition [simp, code]: \bigcup A \longleftrightarrow \text{True} \in A

instance proof qed (auto intro: bool-induct)

end

lemma not-False-in-image-Ball [simp]:
  \text{False} \notin P \cdot A \longleftrightarrow \text{Ball} A P
  by auto

lemma True-in-image-Bex [simp]:
  \text{True} \in P \cdot A \longleftrightarrow \text{Bex} A P
  by auto

lemma INF-bool-eq [simp]:
  \text{INFIMUM} = \text{Ball}
  by (simp add: fun-eq-iff INF-def)

lemma SUP-bool-eq [simp]:
  \text{SUPREMUM} = \text{Bex}
  by (simp add: fun-eq-iff SUP-def)

instance bool :: complete-boolean-algebra proof
  qed (auto intro: bool-induct)
9.4 Complete lattice on \(-\Rightarrow\)-

**instantiation** fun :: (type, Inf) Inf 

**begin**

**definition**
\(\bigwedge A = (\lambda x. \bigcap f \in A. f x)\)

**lemma** Inf-apply [simp, code]:
\((\bigwedge A) \ x = (\bigcap f \in A. f x)\)

**by** (simp add: Inf-fun-def)

**instance** ..

**end**

**instantiation** fun :: (type, Sup) Sup 

**begin**

**definition**
\(\bigvee A = (\lambda x. \bigvee f \in A. f x)\)

**lemma** Sup-apply [simp, code]:
\((\bigvee A) \ x = (\bigvee f \in A. f x)\)

**by** (simp add: Sup-fun-def)

**instance** ..

**end**

**instantiation** fun :: (type, complete-lattice) complete-lattice 

**begin**

**instance proof**
**qed** (auto simp add: le-fun-def intro: INF-lower INF-greatest SUP-upper SUP-least)

**end**

**lemma** INF-apply [simp]:
\((\bigcap y \in A. f \ y) \ x = (\bigcap y \in A. f \ y \ x)\)

**using** Inf-apply [of \(f \ A\)] **by** (simp add: comp-def)

**lemma** SUP-apply [simp]:
\((\bigvee y \in A. f \ y) \ x = (\bigvee y \in A. f \ y \ x)\)

**using** Sup-apply [of \(f \ A\)] **by** (simp add: comp-def)

**instance** fun :: (type, complete-distrib-lattice) complete-distrib-lattice **proof**

**qed** (auto simp add: INF-def SUP-def inf-Sup sup-Inf fun-eq-iff image-image simp del: Inf-image-eq Sup-image-eq)
instance fun :: (type, complete-boolean-algebra) complete-boolean-algebra ..

9.5 Complete lattice on unary and binary predicates

lemma Inf1-I:
\((\forall P. P \in A \Rightarrow P a) \Rightarrow (\bigwedge A) a\)
by auto

lemma INF1-I:
\((\forall x. x \in A \Rightarrow B x b) \Rightarrow (\bigwedge x \in A. B x) b\)
by simp

lemma INF2-I:
\((\forall x. x \in A \Rightarrow B x b c) \Rightarrow (\bigwedge x \in A. B x) b c\)
by simp

lemma Inf2-I:
\((\forall x. x \in A \Rightarrow B x b c) \Rightarrow (\bigwedge x \in A. B x) b c\)
by simp

lemma Inf1-D:
\((\bigwedge A) a \Rightarrow P \in A \Rightarrow P a\)
by auto

lemma INF1-D:
\((\bigwedge x \in A. B x) b \Rightarrow a \in A \Rightarrow B a b\)
by simp

lemma Inf2-D:
\((\bigwedge x \in A. B x) b c \Rightarrow a \in A \Rightarrow B a b c\)
by simp

lemma Inf1-E:
assumes \((\bigwedge A) a\)
obtains \(P a \mid P \notin A\)
using assms by auto

lemma INF1-E:
assumes \((\bigwedge x \in A. B x) b\)
obtains \(B a b \mid a \notin A\)
using assms by auto

lemma Inf2-E:
assumes \((\bigwedge A) a b\)
obtains \(r a b \mid r \notin A\)
using assms by auto

lemma INF2-E:
  assumes \( (\prod x \in A. B \cdot x) \cdot b \cdot c \)
  obtains \( B \cdot a \cdot b \cdot c \mid a \notin A \)
  using assms by auto

lemma Sup1-I:
  \( P \in A \Rightarrow P \cdot a \Rightarrow (\bigcup A) \cdot a \)
  by auto

lemma SUP1-I:
  \( a \in A \Rightarrow B \cdot a \cdot b \Rightarrow (\bigcup x \in A. B \cdot x) \cdot b \)
  by auto

lemma Sup2-I:
  \( r \in A \Rightarrow r \cdot a \cdot b \Rightarrow (\bigcup A) \cdot a \cdot b \)
  by auto

lemma SUP2-I:
  \( a \in A \Rightarrow B \cdot a \cdot b \cdot c \Rightarrow (\bigcup x \in A. B \cdot x) \cdot b \cdot c \)
  by auto

lemma Sup1-E:
  assumes \( (\bigcup A) \cdot a \)
  obtains \( P \) where \( P \in A \) and \( P \cdot a \)
  using assms by auto

lemma SUP1-E:
  assumes \( (\bigcup x \in A. B \cdot x) \cdot b \)
  obtains \( x \) where \( x \in A \) and \( B \cdot x \cdot b \)
  using assms by auto

lemma Sup2-E:
  assumes \( (\bigcup A) \cdot a \cdot b \)
  obtains \( r \) where \( r \in A \) and \( r \cdot a \cdot b \)
  using assms by auto

lemma SUP2-E:
  assumes \( (\bigcup x \in A. B \cdot x) \cdot b \cdot c \)
  obtains \( x \) where \( x \in A \) and \( B \cdot x \cdot b \cdot c \)
  using assms by auto

9.6 Complete lattice on 

instantiation set :: (type) complete-lattice
begin

definition
\[
\bigcap A = \{ x. \bigcap((\lambda B. x \in B) \cdot A) \}
\]
definition
\[
\bigcup A = \{ x. \bigcup((\lambda B. x \in B) \cdot A) \}
\]
instance proof
qed (auto simp add: less-eq-set-def Inf-set-def Sup-set-def le-fun-def)
end

instance set :: (type) complete-boolean-algebra
proof
qed (auto simp add: INF-def SUP-def Inf-set-def Sup-set-def image-def)

9.6.1 Inter
abbreviation Inter :: 'a set set ⇒ 'a set where
\[
\text{Inter } S ≡ d S
\]
notation (xsymbols)
\[
\text{Inter } (\bigcap - [900] 900)
\]
lemma Inter-eq [simp]: \( \bigcap A = \{ x. \forall B \in A. x \in B \} \)
proof (rule set-eqI)
fix \( x \)
have \((\forall Q \in \{ P. \exists B \in A. P \iff x \in B \}. Q) \iff (\forall B \in A. x \in B)\)
  by auto
then show \( x \in \bigcap A \iff x \in \{ x. \forall B \in A. x \in B \} \)
  by (simp add: Inf-set-def image-def)
qed

lemma Inter-iff [simp]: \( A \in \bigcap C \iff (\forall X \in C. A \in X) \)
by (unfold Inter-eq) blast

lemma InterI [intro!]: \((\bigcap X. X \in C \Longrightarrow A \in X) \Longrightarrow A \in \bigcap C\)
by (simp add: Inter-eq)

A “destruct” rule – every \( X \) in \( C \) contains \( A \) as an element, but \( A \in X \) can
hold when \( X \in C \) does not! This rule is analogous to spec.

lemma InterD [elim, Pure.elim]: \( A \in \bigcap C \Longrightarrow X \in C \Longrightarrow A \in X \)
by auto

lemma InterE [elim]: \( A \in \bigcap C \Longrightarrow (X \notin C \Longrightarrow R) \Longrightarrow (A \in X \Longrightarrow R) \Longrightarrow R\)
— “Classical” elimination rule – does not require proving \( X \in C \).
by (unfold Inter-eq) blast

lemma Inter-lower: \( B \in A \Longrightarrow \bigcap A \subseteq B \)
by (fact Inf-lower)
lemma Inter-subset: 
\( (\bigwedge X. X \in A \Rightarrow X \subseteq B) \Rightarrow A \neq \{\} \Rightarrow \bigcap A \subseteq B \) 
by (fact Inf-less-eq)

lemma Inter-greatest: \( (\bigwedge X. X \in A \Rightarrow C \subseteq X) \Rightarrow C \subseteq \bigcap A \) 
by (fact Inf-greatest)

lemma Inter-empty: \( \bigcap \{\} = \text{UNIV} \) 
by (fact Inf-empty)

lemma Inter-UNIV: \( \bigcap \text{UNIV} = \{\} \) 
by (fact Inf-UNIV)

lemma Inter-insert: \( \bigcap (\text{insert } a \ B) = a \cap \bigcap \ B \) 
by (fact Inf-insert)

lemma Inter-Un-subset: \( \bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B) \) 
by (fact less-eq-Inf-inter)

lemma Inter-Un-distrib: \( \bigcap (A \cup B) = \bigcap A \cap \bigcap B \) 
by (fact Inf-union-distrib)

lemma Inter-UNIV-conv [simp]: 
\( \bigcap A = \text{UNIV} \iff (\forall x \in A. x = \text{UNIV}) \) 
\( \text{UNIV} = \bigcap A \iff (\forall x \in A. x = \text{UNIV}) \) 
by (fact Inf-top-conv)

lemma Inter-anti-mono: \( B \subseteq A \Rightarrow \bigcap A \subseteq \bigcap B \) 
by (fact Inf-superset-mono)

9.6.2 Intersections of families

abbreviation INTER :: 
\( 'a \set \Rightarrow ('a \Rightarrow 'b \set) \Rightarrow 'b \set \) where 
INTER \equiv INFIMUM

Note: must use name INTER here instead of INT to allow the following syntax coexist with the plain constant name.

syntax

-INTER1 :: pttrns => 'b set => 'b set 
\(((3\set ./-) [0, 10] 10)\)

-INTER :: pttrn => 'a set => 'b set => 'b set 
\(((3\set ./- ./-) [0, 0, 10] 10)\)

syntax (xsymbols)

-INTER1 :: pttrns => 'b set => 'b set 
\(((3\bigcap ./-) [0, 10] 10)\)

-INTER :: pttrn => 'a set => 'b set => 'b set 
\(((3\bigcap ./- ./-) [0, 0, 10] 10)\)

syntax (latex output)

-INTER1 :: pttrns => 'b set => 'b set 
\(((3\bigcap (00 ./-) [0, 10] 10)\)
THEORY "Complete-Lattices"

-INTER :: pttrn => 'a set => 'b set => 'b set ((3\cap (00..)/ -) [0, 0, 10] 10)

translations

INT x y. B == INT x. INT y. B
INT x. B == CONST INTER CONST UNIV (%x. B)
INT x. B == INT x:CONST UNIV. B
INT x:A. B == CONST INTER A (%x. B)

print-translation `\langle Syntax-Trans.preserve-binder-abs2-tr @{const-syntax INTER} @{syntax-const -INTER}\rangle` — to avoid eta-contraction of body

lemma INTER-eq:
(\bigcap x\in A. B x) = \{y. \forall x\in A. y \in B x\}
by (auto intro!: INF-eqI)

lemma Inter-image-eq:
\bigcap (B' A) = (\bigcap x\in A. B x)
by (fact Inf-image-eq)

lemma INT-iff [simp]: b \in (\bigcap x\in A. B x) \iff (\forall x\in A. b \in B x)
using INTER-I [intro!]: (\forall x. x \in A \implies b \in B x) \implies b \in (\bigcap x\in A. B x)
by (auto simp add: INF-def image-def)

lemma INT-D [elim, Pure.elim]: b \in (\bigcap x\in A. B x) \implies a \in A \implies b \in B a
by auto

lemma INT-E [elim]: b \in (\bigcap x\in A. B x) \implies (b \in B a \implies R) \implies (a \notin A \implies R) \implies R
— "Classical" elimination – by the Excluded Middle on a \in A.
by (auto simp add: INF-def image-def)

lemma Collect-ball-eq: \{x. \forall y\in A. P x y\} = (\bigcap y\in A. \{x. P x y\})
by blast

lemma Collect-all-eq: \{x. \forall y. P x y\} = (\bigcap y. \{x. P x y\})
by blast

lemma INT-lower: a \in A \implies (\bigcap x\in A. B x) \subseteq B a
by (fact INF-lower)

lemma INT-greatest: (\forall x. x \in A \implies C \subseteq B x) \implies C \subseteq (\bigcap x\in A. B x)
by (fact INF-greatest)

lemma INT-empty: (\bigcap x\in\{}. B x) = UNIV
THEORY "Complete-Lattices"

by (fact INF-empty)

lemma INT-absorb: \( k \in I \implies A \cap (\bigcap_{i \in I} A \ i) = (\bigcap_{i \in I} A \ i) \)
by (fact INF-absorb)

lemma INT-subset-iff: \( B \subseteq (\bigcap_{i \in I} A \ i) \iff (\forall i \in I. B \subseteq A \ i) \)
by (fact le-INF-iff)

lemma INT-insert [simp]: \( (\bigcap_{x \in \text{insert} \ a \ A} B \ x) = (\bigcap_{x \in A} B \ x) \)
by (fact INF-insert)

lemma INT-constant [simp]: \( (\bigcap_{y \in A} c) = (\text{if } A = \{\} \text{ then UNIV else } c) \)
by (fact INF-constant)

lemma INTER-UNIV-conv:
(UNIV = (\bigcap_{x \in A} B \ x)) = (\forall x \in A. B \ x = \text{UNIV})
((\bigcap_{x \in A} B \ x) = \text{UNIV}) = (\forall x \in A. B \ x = \text{UNIV})
by (fact INF-top-conv+)

lemma INT-bool-eq: (\bigcap_{b. A \ b} A \ True \cap A \ False)
by (fact INF-UNIV-bool-expand)

lemma INT-anti-mono:
A \subseteq B \implies (\forall x. x \in A \implies f \ x \subseteq g \ x) \implies (\bigcap_{x \in B} f \ x) \subseteq (\bigcap_{x \in A} g \ x)
— The last inclusion is POSITIVE!
by (fact INF-superset-mono)

lemma Pow-INT-eq: Pow (\bigcap_{x \in A} B \ x) = (\bigcap_{x \in A} \text{Pow} (B \ x))
by blast

lemma vimage-INT: f −′ (\bigcap_{x \in A} B \ x) = (\bigcap_{x \in A} f −′ B \ x)
by blast

9.6.3 Union

abbreviation Union :: 'a set set ⇒ 'a set where
Union S ≡ ⨆ S

notation (xsymbols)
Union  (∪ · [900] 900)

lemma Union-eq:
\[ \bigcup A = \{ x. \exists B \in A. x \in B \} \]

**proof** (rule set-eqI)

fix \( x \)

have \((\exists Q \in \{ P. \exists B \in A. P \iff x \in B \}). Q) \iff (\exists B \in A. x \in B)\)
  by auto

then show \( x \in \bigcup A \iff x \in \{ x. \exists B \in A. x \in B \} \)
  by (simp add: Sup-set-def image-def)

qed

**lemma** Union-iff [simp]:
\[
A \in \bigcup C \iff (\exists X \in C. A \in X)
\]
by (unfold Union-eq) blast

**lemma** UnionI [intro]:
\[
X \in C \Longrightarrow A \in X \Longrightarrow A \in \bigcup C
\]
— The order of the premises presupposes that \( C \) is rigid; \( A \) may be flexible.
by auto

**lemma** UnionE [elim!]:
\[
A \in \bigcup C \Longrightarrow (\forall X. A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R
\]
by auto

**lemma** Union-upper:
\[ B \in A \Longrightarrow B \subseteq \bigcup A \]
by (fact Sup-upper)

**lemma** Union-least:
\[ (\forall X. X \in A \Longrightarrow X \subseteq C) \Longrightarrow \bigcup A \subseteq C \]
by (fact Sup-least)

**lemma** Union-empty:
\[ \bigcup \{ \} = \{ \} \]
by (fact Sup-empty)

**lemma** Union-UNIV:
\[ \bigcup \text{UNIV} = \text{UNIV} \]
by (fact Sup-UNIV)

**lemma** Union-insert:
\[ \bigcup \text{insert } a \ B = a \cup \bigcup B \]
by (fact Sup-insert)

**lemma** Union-Un-distrib [simp]:
\[ \bigcup (A \cup B) = \bigcup A \cup \bigcup B \]
by (fact Sup-union-distrib)

**lemma** Union-Int-subset:
\[ \bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B \]
by (fact Sup-inter-less-eq)

**lemma** Union-empty-conv:
\[ (\bigcup A = \{ \}) \iff (\forall x \in A. x = \{ \}) \]
by (fact Sup-bot-conv)

**lemma** empty-Union-conv:
\[ (\{ \} = \bigcup A) \iff (\forall x \in A. x = \{ \}) \]
by (fact Sup-bot-conv)
lemma subset-Pow-Union: $A \subseteq \text{Pow}(\bigcup A)$
by blast

lemma Union-Pow-eq [simp]: $\bigcup (\text{Pow } A) = A$
by blast

lemma Union-mono: $A \subseteq B \implies \bigcup A \subseteq \bigcup B$
by (fact Sup-subset-mono)

9.6.4 Unions of families

abbreviation UNION :: ('a set) ⇒ ('a ⇒ 'b set) ⇒ 'b set where
UNION ≡ SUPREMUM

Note: must use name UNION here instead of UN to allow the following syntax coexist

syntax
-UNION1 :: pttrns => 'b set => 'b set => 'b set
  ((3UN -. / -) [0, 10] 10)
-UNION :: pttrn => 'a set => 'b set => 'b set
  ((3UN -. / -) [0, 0, 10] 10)

translations
\begin{align*}
  UN x y. B & = \text{ UN } x. \text{ UN } y. B \\
  UN x. B & = \text{ CONST UNION CONST UNIV } (%x. B) \\
  UN x. B & = \text{ UN } x. \text{ CONST UNIV. } B \\
  UN x:A. B & = \text{ CONST UNION } A \ (%x. B)
\end{align*}

Note the difference between ordinary xsymbol syntax of indexed unions and intersections
(e.g. $\bigcup a_1 \in A_1. B$) and their \texttt{\LaTeX} rendition: $\bigcup_{a_1 \in A_1} B$. The former
does not make the index expression a subscript of the union/intersection symbol because
this leads to problems with nested subscripts in Proof General.

print-translation ⟨⟨ Syntax-Trans.preserve-binder-abs2-tr' \{\texttt{const-syntax UNION}\} \{\texttt{syntax-const -UNION}\} ⟩⟩ — to avoid eta-contraction of body

lemma UNION-eq:
\[
(\bigcup x \in A. B \ x) = \{ y. \exists x \in A. y \in B \ x \}
THEORY "Complete-Lattices"

by (auto intro!: SUP-eqI)

lemma bind-UNION [code]:
  Set.bind A f = UNION A f
by (simp add: bind-def UNION-eq)

lemma member-bind [simp]:
  x ∈ Set.bind P f ←→ x ∈ UNION P f
by (simp add: bind-UNION)

lemma Union-image-eq:
  ⋃ (B · A) = (⋃ x∈A. B x)
by (fact Sup-image-eq)

lemma UN-iff [simp]:
  b ∈ (⋃ x∈A. B x) ←→ (∃ x∈A. b ∈ B x)
using Union-iff [of - B · A] by simp

lemma UN-I [intro]:
  a ∈ A =⇒ b ∈ B a =⇒ b ∈ (⋃ x∈A. B x)
— The order of the premises presupposes that A is rigid; b may be flexible.
by auto

lemma UN-E [elim!]:
  b ∈ (⋃ x∈A. B x) =⇒ (∀ x∈A. b ∈ B x =⇒ R) =⇒ R
  by (auto simp add: SUP-def image-def)

lemma image-eq-UN:
  f · A = (⋃ x∈A. {f x})
by blast

lemma UN-upper: a ∈ A =⇒ B a ⊆ (⋃ x∈A. B x)
by (fact SUP-upper)

lemma UN-least: (∀ x∈A. B x ⊆ C) =⇒ (⋃ x∈A. B x) ⊆ C
by (fact SUP-least)

lemma Collect-bex-eq:
  {x. ∃ y∈A. P x y} = (⋃ y∈A. {x. P x y})
by blast

lemma UN-insert-distrib:
  u ∈ A =⇒ (⋃ x∈A. insert a (B x)) = insert a (⋃ x∈A. B x)
by blast

lemma UN-empty: (∪ x∈{}. B x) = {}
by (fact SUP-empty)

lemma UN-empty2: (∪ x∈A. {}) = {}
by (fact SUP-bot)

lemma UN-absorb:
  k ∈ I =⇒ A k ∪ (⋃ i∈I. A i) = (⋃ i∈I. A i)
by (fact SUP-absorb)
lemma UN-insert [simp]: \( \bigcup x \in \text{insert} \ a \ B \ x = B \ a \cup \text{UNION} \ A \ B \)
by (fact SUP-insert)

lemma UN-Un [simp]: \( \bigcup i \in A \cup B. \ M \ i = (\bigcup i \in A. \ M \ i) \cup (\bigcup i \in B. \ M \ i) \)
by (fact SUP-union)

lemma UN-UN-flatten: \( \bigcup x \in (\bigcup y \in A. \ B \ y). \ C \ x = (\bigcup y \in A. \ \bigcup x \in B \ y. \ C \ x) \)
by blast

lemma UN-subset-iff: \( (\bigcap x \in I. \ A \ x) \subseteq B \) = \( (\forall i \in I. \ A \ i \subseteq B) \)
by (fact SUP-le-iff)

lemma UN-constant [simp]: \( \bigcup y \in A. \ c \) = \( \text{if } A = \{\} \text{ then } \{\} \text{ else } c \)
by (fact SUP-constant)

lemma Collect-ex-eq: \( \{x. \ \exists y. \ P \ x \ y\} = (\bigcup y. \ \{x. \ P \ x \ y\}) \)
by blast

lemma image-Union: \( f ' \bigcup S = (\bigcup x \in S. \ f ' x) \)
by blast

lemma UNION-empty-conv:
\( \{\} = (\bigcup x \in A. \ B \ x) \longleftrightarrow (\forall x \in A. \ B \ x = \{\}) \)
\( (\bigcup x \in A. \ B \ x) = \{\} \longleftrightarrow (\forall x \in A. \ B \ x = \{\}) \)
by (fact SUP-bot-conv)

lemma Collect-ex-eq: \( \{x. \ \exists y. \ P \ x \ y\} = (\bigcup y. \ \{x. \ P \ x \ y\}) \)
by blast

lemma ball-UN: \( (\forall z \in \text{UNION} \ A \ B. \ P \ z) \longleftrightarrow (\forall x \in A. \forall z \in B \ x. \ P \ z) \)
by blast

lemma bex-UN: \( (\exists z \in \text{UNION} \ A \ B. \ P \ z) \longleftrightarrow (\exists x \in A. \exists z \in B \ x. \ P \ z) \)
by blast

lemma Un-eq-UN: \( A \cup B = \bigcup b. \text{if } b \text{ then } A \text{ else } B \)
by (auto simp add: split-if-mem2)

lemma UN-bool-eq: \( \bigcup \ b. \ A \ b = (A \ True \cup A \ False) \)
by (fact SUP-UNIV-bool-expand)

lemma UN-Pow-subset: \( (\bigcup x \in A. \ \text{Pow} (B \ x)) \subseteq \text{Pow} (\bigcup x \in A. \ B \ x) \)
by blast

lemma UN-mono:
\( A \subseteq B \Longrightarrow (\bigcup x. \ x \in A \Longrightarrow f \ x \subseteq g \ x) \Longrightarrow \)
\( (\bigcup x \in A. \ f \ x) \subseteq (\bigcup x \in B. \ g \ x) \)
by (fact SUP-subset-mono)

lemma vimage-Union: \( f ^-\ (\bigcup A) = (\bigcup X \in A. \ f ^-\ X) \)
by blast
lemma vimage-UN: \( f^{-1}(\bigcup_{x \in A} B \ x) = (\bigcup_{x \in A} f^{-1} B \ x) \)
by blast

lemma vimage-eq-UN: \( f^{-1} B = (\bigcup_{y \in B} f^{-1} \{y\}) \)
— NOT suitable for rewriting
by blast

lemma image-UN: \( f^{-} \text{UNION} A B = (\bigcup_{x \in A} f^{-} B \ x) \)
by blast

lemma UN-singleton [simp]: \( (\bigcup_{x \in A} \{x\}) = A \)
by blast

9.6.5 Distributive laws

lemma Int-Union: \( A \cap \bigcup B = (\bigcup C \in B \ A \cap C) \)
by (fact inf-Sup)

lemma Un-Inter: \( A \cup \bigcap B = (\bigcap C \in B \ A \cup C) \)
by (fact sup-Inf)

lemma Int-Union2: \( \bigcup B \cap A = (\bigcup C \in B \ C \cap A) \)
by (fact Sup-inf)

lemma INT-Int-distrib: \( (\bigcap_{i \in I} A \ i \cap B \ i) = (\bigcap_{i \in I} A \ i) \cap (\bigcap_{i \in I} B \ i) \)
by (rule sym) (rule INF-inf-distrib)

lemma UN-Un-distrib: \( (\bigcup_{i \in I} A \ i \cup B \ i) = (\bigcup_{i \in I} A \ i) \cup (\bigcup_{i \in I} B \ i) \)
by (rule sym) (rule SUP-sup-distrib)

lemma Int-Inter-image: \( (\bigcap_{x \in C} A \ x \cap B \ x) = (\bigcap (A^{-} C) \cap (B^{-} C)) \)
— FIXME
drop by (simp add: INT-Int-distrib)

lemma Un-Union-image: \( (\bigcup_{x \in C} A \ x \cup B \ x) = (\bigcup (A^{-} C) \cup (B^{-} C)) \)
— FIXME drop
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by (simp add: UN-Un-distrib)

lemma Un-INT-distrib: \( B \cup (\bigcap_{i \in I} A \ i) = (\bigcap_{i \in I} B \cup A \ i) \)
by (fact sup-INF)

lemma Int-UN-distrib: \( B \cap (\bigcup_{i \in I} A \ i) = (\bigcup_{i \in I} B \cap A \ i) \)
— Halmos, Naive Set Theory, page 35.
by (fact inf-SUP)

lemma Int-UN-distrib2: \( (\bigcup_{i \in I} A \ i) \cap (\bigcup_{j \in J} B \ j) = (\bigcup_{i \in I, j \in J} A \ i \cap B \ j) \)
THEORY "Complete-Lattices"

236

j)

by (fact SUP-inf-distrib2)

lemma Un-INT-distrib2: (∩ i∈I. A i) ∪ (∩ j∈J. B j) = (∩ i∈I. ∩ j∈J. A i ∪ B j)

by (fact INF-sup-distrib2)

lemma Union-disjoint: (∪ C ⊆ A = {}) ↔ (∀ B∈C. B ⊆ A = {})

by (fact Sup-inf-eq-bot-iff)

9.7 Injections and bijections

lemma inj-on-Inter:

S ≠ {} ⇒ (∀ A. A ∈ S ⇒ inj-on f A) ⇒ inj-on f (∩ S)

unfolding inj-on-def by blast

lemma inj-on-INTER:

I ≠ {} ⇒ (∀ i. i ∈ I ⇒ inj-on f (A i)) ⇒ inj-on f (∩ i ∈ I. A i)

unfolding inj-on-def by blast

lemma inj-on-UNION-chain:

assumes CH: \( \land i j. [i \in I; j \in I] \Rightarrow A i \leq A j \lor A j \leq A i \) and

INJ: \( \land i. i \in I \Rightarrow inj-on f (A i) \)

shows inj-on f (∪ i ∈ I. A i)

proof –

{ fix i j x y

assume *: i ∈ I j ∈ I and **: x ∈ A i y ∈ A j

and ***: f x = f y

have x = y

proof –

{ assume A i ≤ A j

with ** have x ∈ A j by auto

with INJ * ** *** have ?thesis

by(auto simp add: inj-on-def)
}

moreover

{ assume A j ≤ A i

with ** have y ∈ A i by auto

with INJ * ** *** have ?thesis

by(auto simp add: inj-on-def)
}

ultimately show ?thesis using CH * by blast

qed

then show ?thesis by (unfold inj-on-def UNION-eq) auto

qed
lemma bij-betw-UNION-chain:
  assumes CH: \( \forall i, j \in I; j \in I \Longrightarrow A i \leq A j \vee A j \leq A i \) and
  BIJ: \( \forall i \in I \Longrightarrow \text{bij-betw } f (A i) (A' i) \)
shows bij-betw \( (\bigcup i \in I. A i) (\bigcup i \in I. A' i) \)
proof (unfold bij-betw-def, auto)
  have \( \forall i \in I; f (A i) = \bigcup i \in I. A i \) using BIJ inj-on-def[of f] by auto
  thus inj-on f (\bigcup i \in I. A i) using CH inj-on-UNION-chain[of I A f] by auto
next
  fix i x
  assume \( \ast: i \in I \land x \in A i \)
  hence \( f x \in A' i \) using BIJ bij-betw-def[of f] by auto
  thus \( \exists j \in I. f x \in A' j \) using \( \ast \) by blast
next
  fix i x'
  assume \( \ast: i \in I ; x' \in A' i \)
  hence \( \exists x \in A i. x' = f x \) using BIJ bij-betw-def[of f] by blast
  then have \( \exists j \in I. \exists x \in A j. x' = f x \)
    using \( \ast \) by blast
  then show \( x' \in f' (\bigcup i \in I. A i) \) by blast
qed

lemma image-INT:
  \[ [ \forall i \in I. f C \leq C; \text{ ALL } x. A B x \leq C; j:A ] \]
  ==> \( f' (\bigcap x \in A. B x) = (\bigcap x \in A. f' B x) \)
apply (simp add: inj-on-def, blast)
done

lemma bij-image-INT: bij f ==> \( f' (\bigcap A B) = (\bigcap x:A. f' B x) \)
apply (simp add: bij-def)
apply (simp add: inj-on-def surj-def, blast)
done

lemma UNION-fun-upd:
  UNION J (A(i:=B)) = (UNION (J\{-i\}) A \cup (if i \in J then B else {}))
by (auto split: if-defs)

9.7.1 Complement

lemma Compl-INT [simp]: \( \neg (\bigcap x \in A. B x) = (\bigcup x \in A. \neg B x) \)
by (fact uminus-INF)

lemma Compl-UN [simp]: \( \neg (\bigcup x \in A. B x) = (\bigcap x \in A. \neg B x) \)
by (fact uminus-SUP)
9.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

**lemma** UN-simps [simp]:
\[ \forall a \, B \, C . \ ( \bigcup x \in C . \ insert a \ (B \ x)) = (if \ C=\{} \ then \ \{} \ else \ insert a \ (\bigcup x \in C . \ B \ x)) \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A \ x) = (if \ C=\{} \ then \ \{} \ else \ (\bigcup x \in C . \ A \ x) \cup B) \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A \cup B \ x) = (if \ C=\{} \ then \ \{} \ else \ A \cup (\bigcup x \in C . \ B \ x)) \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A \ x \cap B \ x) = (\bigcup x \in C . \ A \ x) \cap B \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A \cap B \ x) = (A \cap (\bigcup x \in C . \ B \ x)) \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A \ x - B \ x) = (\bigcup x \in C . \ A \ x) - B \]
\[ \forall A \, B \, C . \ (\bigcup x \in C . \ A - B \ x) = (A - (\bigcup x \in C . \ B \ x)) \]
\[ \forall A \, B . \ (\bigcup x \in A . \ B \ x) = (\bigcup y \in A . \ x \in y . \ B \ x) \]
\[ \forall A \, B . \ (\bigcup z \in \bigcup A . \ B . \ C \ x) = (\bigcup z \in x A . \ x \in B x . \ C \ x) \]
\[ \forall A \, B . \ (\bigcup x \in A . \ B x) = (\bigcup a \in A . \ B \ (f a)) \]
**by** auto

**lemma** INT-simps [simp]:
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \ x \cap B \ x) = (if \ C=\{} \ then \ UNIV \ else \ (\bigcap x \in C . \ A \ x) \cap B) \]
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \ x \cup B) = (if \ C=\{} \ then \ UNIV \ else \ (\bigcap x \in C . \ A \ x) \cup B) \]
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \ x \cup B \ x) = (A \cup (\bigcap x \in C . \ B \ x)) \]
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \ x \cup B \ x) = (\bigcap x \in C . \ A \ x \cup B) \]
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \cap B \ x) = (A \cap (\bigcap x \in C . \ B \ x)) \]
\[ \forall a \, B \, C . \ (\bigcap x \in C . \ A \cap B \ x) = (\bigcap x \in C . \ A \ x \cap B) \]
\[ \forall a \, B \, C . \ (\bigcap z \in \bigcap A . \ B . \ C \ x) = (\bigcap z \in x A . \ x \in z B x . \ C \ x) \]
\[ \forall a \, B . \ (\bigcap x \in A . \ B \ x) = (\bigcap a \in A . \ B \ (f a)) \]
**by** auto

**lemma** UN-ball-bex-simps [simp]:
\[ \forall A \, B \, C . \ (\forall x \in \bigcup A . \ P \ x) \leftrightarrow (\forall y \in A . \forall x \in y . \ P \ x) \]
\[ \forall A \, B \, P . \ (\forall x \in \bigcup A . \ B \ x) = (\forall a \in A . \forall x \in B a . \ P \ x) \]
\[ \forall A \, P . \ (\exists x \in \bigcup A . \ P \ x) \leftrightarrow (\exists y \in A . \exists x \in y . \ P \ x) \]
\[ \forall A \, B \ P . \ (\exists x \in \bigcup A . \ B \ x) \leftrightarrow (\exists a \in A . \exists x \in B a . \ P \ x) \]
**by** auto

Maxiscoping: pulling out big Unions and Intersections.

**lemma** UN-extend-simps:
\[ \forall a \, B \, C . \ insert a \ (\bigcup x \in C . \ B \ x) = (if \ C=\{} \ then \ \{} \ else \ (\bigcup x \in C . \ insert a \ (B \ x)) \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x) = (if \ C=\{} \ then \ \{} \ else \ (\bigcup x \in C . \ A \ x) \cup B) \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x \cup B \ x) = (if \ C=\{} \ then \ \{} \ else \ A \cup (\bigcup x \in C . \ B \ x)) \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x \cap B \ x) = (\bigcup x \in C . \ A \ x) \cap B \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x \cap B \ x) = (A \cap (\bigcup x \in C . \ B \ x)) \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x - B \ x) = (\bigcup x \in C . \ A \ x) - B \]
\[ \forall a \, B \, C . \ (\bigcup x \in C . \ A \ x - B \ x) = (A - (\bigcup x \in C . \ B \ x)) \]
\[ \forall a \, B . \ (\bigcup y \in A . \ x \in y . \ B \ x) = (\bigcup x \in \bigcup A . \ B \ x) \]
THEORY "Ctr-Sugar"

\[ \forall A B C. (\bigcup x \in A. \bigcup z \in B. C z) = (\bigcup z \in \text{UNION } A. B. C z) \]
\[ \forall A B f. (\bigcup a \in A. B (f a)) = (\bigcup x \in \text{f } A. B x) \]
by auto

lemma INT-extend-simps:
\[ \forall A B C. (\bigcap x \in C. A x) \cap B = (\text{if } C=\{\} \text{ then } B \text{ else } (\bigcap x \in C. A x \cap B)) \]
\[ \forall A B C. A \cap (\bigcap x \in C. B x) = (\text{if } C=\{\} \text{ then } A \text{ else } (\bigcap x \in C. A \cap B x)) \]
\[ \forall A B C. (\bigcap x \in C. A x) - B = (\text{if } C=\{\} \text{ then } \text{UNIV} - B \text{ else } (\bigcap x \in C. A x - B)) \]
\[ \forall a B C. \text{insert } a (\bigcap x \in C. B x) = (\text{if } C=\{\} \text{ then } A - (\bigcap x \in C. A - B x)) \]
\[ \forall A B C. A - (\bigcup x \in C. B x) = (\text{if } C=\{\} \text{ then } A \text{ else } (\bigcap x \in C. A - B x)) \]
\[ \forall a B C. \text{insert } a (\bigcap x \in C. B x) = (\bigcap x \in C. \text{insert } a (B x)) \]
\[ \forall A B C. (\bigcap x \in C. A x) \cup B = (\bigcap x \in C. A x \cup B) \]
\[ \forall A B C. A \cup (\bigcap x \in C. B x) = (\bigcap x \in C. A \cup B x) \]
\[ \forall A B C. (\bigcap y \in A. \bigcap x \in y. B x) = (\bigcap x \in \bigcup A. B x) \]
\[ \forall A B C. (\bigcap x \in A. \bigcap z \in B. C z) = (\bigcap z \in \text{UNION } A. B. C z) \]
\[ \forall A B f. (\bigcap a \in A. B (f a)) = (\bigcap x \in \text{f } A. B x) \]
by auto

Finally

no-notation
less-eq (infix \(\subseteq\) 50) and
less (infix \(\subset\) 50)

lemmas mem-simps =
insert-if empty-if Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [simp] above.

end

10 Ctr-Sugar: Wrapping Existing Freely Generated Type’s Constructors

theory Ctr-Sugar
imports HOL
keywords
print-case-translations :: diag and
free-constructors :: thy-goal
begin

consts
case-guard :: bool ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ 'b
case-nil :: 'a ⇒ 'b
case-cons :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b
case-elem :: 'a ⇒ 'b ⇒ 'a ⇒ 'b
case-abs :: ('c ⇒ 'b) ⇒ 'b

declare [[coercion-args case-guard - + -]]
```
declare [[coercion-args case-cons −]]
declare [[coercion-args case-abs −]]
declare [[coercion-args case-elem − +]]

ML-file Tools/Ctr-Sugar/case-translation.ML

lemma iffI-np: \[ x \implies \neg y; \neg x \implies y \] \implies \neg x \iff y
by (erule iffI) (erule contrapos-pn)

lemma iff-contradict:
\neg P \implies P \iff Q \implies R
\neg Q \implies P \iff Q \implies P \implies R
by blast+

ML-file Tools/Ctr-Sugar/ctr-sugar-util.ML
ML-file Tools/Ctr-Sugar/ctr-sugar-tactics.ML
ML-file Tools/Ctr-Sugar/ctr-sugar-code.ML
ML-file Tools/Ctr-Sugar/ctr-sugar.ML

end

11 Inductive: Knaster-Tarski Fixpoint Theorem and
inductive definitions

theory Inductive
imports Complete-Lattices Ctr-Sugar
keywords
  inductive coinductive inductive-cases inductive-simps :: thy-decl and
  monos and
  print-inductives :: diag and
  rep-datatype :: thy-goal and
  primrec :: thy-decl
begin

11.1 Least and greatest fixed points

context complete-lattice
begin

definition lfp :: ('a ⇒ 'a) ⇒ 'a where
lfp f = Inf \{ u. f u ≤ u\} — least fixed point

definition gfp :: ('a ⇒ 'a) ⇒ 'a where
gfp f = Sup \{ u. u ≤ f u\} — greatest fixed point
```
11.2 Proof of Knaster-Tarski Theorem using $\text{lfp}$

$lfp$ is the least upper bound of the set $\{u. f u \leq u\}$

**Lemma lfp-lowerbound**: $f A \leq A \implies lfp f \leq A$

by (auto simp add: lfp-def intro: Inf-lower)

**Lemma lfp-greatest**: $(\forall u. f u \leq u \implies A \leq u) \implies A \leq lfp f$

by (auto simp add: lfp-def intro: Inf-greatest)

end

**Lemma lfp-lemma2**: $\text{mono } f \implies f (\text{lfp } f) \leq \text{lfp } f$

by (iprover intro: lfp-greatest order-trans monoD lfp-lowerbound)

**Lemma lfp-lemma3**: $\text{mono } f \implies \text{lfp } f \leq f (\text{lfp } f)$

by (iprover intro: lfp-lemma2 monoD lfp-lowerbound)

**Lemma lfp-unfold**: $\text{mono } f \implies \text{lfp } f = f (\text{lfp } f)$

by (iprover intro: order-antisym lfp-lemma2 lfp-lemma3)

**Lemma lfp-const**: $\text{lfp } (\lambda x. t) = t$

by (rule lfp-unfold) (simp add: mono-def)

11.3 General induction rules for least fixed points

**Theorem lfp-induct**:  
assumes $\text{mono: mono } f$ and $\text{ind}: f (\inf (\text{lfp } f) P) \leq P$

shows $\text{lfp } f \leq P$

proof –

have $\inf (\text{lfp } f) P \leq \text{lfp } f$ by (rule inf-le1)

with $\text{mono}$ have $f (\inf (\text{lfp } f) P) \leq \text{lfp } f$ ..

also from $\text{mono}$ have $f (\text{lfp } f) = \text{lfp } f$ by (rule lfp-unfold [symmetric])

finally have $f (\inf (\text{lfp } f) P) \leq \text{lfp } f$

from this and $\text{ind}$ have $f (\inf (\text{lfp } f) P) \leq \inf (\text{lfp } f) P$ by (rule le-infI)

hence $\text{lfp } f \leq \inf (\text{lfp } f) P$ by (rule lfp-lowerbound)

also have $\inf (\text{lfp } f) P \leq P$ by (rule inf-le2)

finally show $?thesis$.

qed

**Lemma lfp-induct-set**:  
assumes $\text{lfp: a: lfp}(f)$ and $\text{mono: mono}(f)$

and $\text{indhyp: } !!x. [| x: f(\text{lfp} f) \text{ Int } \{x. P(x)\}) |] \implies P(x)$

shows $P(a)$

by (rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp])

(auto simp: intro: indhyp)

**Lemma lfp-ordinal-induct**:  
fixes $f :: 'a::complete-lattice \Rightarrow 'a$
assumes mono: mono f
and P-f: \( \forall S. P S \implies P (f S) \)
and P-Union: \( \forall M. \forall S \in M. P S \implies P (\text{Sup } M) \)
shows \( P (\text{lfp } f) \)
proof
let \( ?M = \{ S. S \leq \text{lfp } f \land P S \} \)
have \( P (\text{Sup } ?M) \) using P-Union by simp
also have \( \text{Sup } ?M = \text{lfp } f \)
proof (rule antisym)
  show \( \text{Sup } ?M \leq \text{lfp } f \) by (blast intro: Sup-least)
  hence \( f (\text{Sup } ?M) \leq \text{lfp } f \) by (rule mono [THEN lfp-ordinal-induct])
  hence \( f (\text{Sup } ?M) \in ?M \) using P-f by simp
  hence \( f (\text{Sup } ?M) \leq \text{Sup } ?M \) by (rule Sup-upper)
thus \( \text{lfp } f \leq \text{Sup } ?M \) by (rule lfp-lowerbound)
qed
finally show \( ?\text{thesis} \).
qed

lemma lfp-ordinal-induct-set:
assumes mono: mono f
and P-f: \( \forall S. P S \implies P (f S) \)
and P-Union: \( \forall M. \forall S \in M. P S \implies P (\text{Union } M) \)
shows \( P (\text{lfp } f) \)
using assms by (rule lfp-ordinal-induct)

Definition forms of \text{lfp-unfold} and \text{lfp-induct}, to control unfolding

lemma def-lfp-unfold: \( \| h = \text{lfp } f \); mono(f) \| \implies h = f(h) \)
by (auto intro!: lfp-unfold)

lemma def-lfp-induct:
\( \| A =\text{lfp}(f); \text{mono}(f); f (\text{inf } A P) \leq P \| \implies A \leq P \)
by (blast intro!: lfp-induct)

lemma def-lfp-induct-set:
\( \| A =\text{lfp}(f); \text{mono}(f); a:A; \forall x. \| x: f(A \text{ Int } \{ x. P(x)\}) \| \implies P(a) \| \implies P(a) \)
by (blast intro!: lfp-induct-set)

lemma lfp-mono: \( \| !Z. f Z \leq g Z \| \implies lfp f \leq lfp g \)
by (rule lfp-lowerbound [THEN lfp-greatest], blast intro: order-trans)

11.4 Proof of Knaster-Tarski Theorem using \text{gfp}

\text{gfp } f \text{ is the greatest lower bound of the set } \{ u. u \leq f u \}
**11.5 Coinduction rules for greatest fixed points**

weak version

lemma weak-coinduct: \(\forall a : X; \ X \subseteq f(X)\) \(\implies a : \text{gfp}(f)\)

by (rule gfp-upperbound [THEN subsetD]) auto

lemma weak-coinduct-image: \(\forall !. x : X; \ g'X \subseteq f (g'X)\) \(\implies g \ a : \text{gfp} f\)

apply (erule gfp-upperbound [THEN subsetD])

apply (erule imageI)

done

lemma coinduct-lemma:

\(\forall X \leq f (\text{sup } X (\text{gfp } f)); \ mono f\) \(\implies \text{sup } X (\text{gfp } f) \leq f (\text{sup } X (\text{gfp } f))\)

apply (frule gfp-lemma2)

apply (erule mono-sup)

apply (rule le-supI)

apply assumption

apply (rule order-trans)

apply (rule order-trans)

apply assumption

apply (rule sup-cn2)

apply assumption

done

strong version, thanks to Coen and Frost

lemma coinduct-set: \(\forall mono(f); \ a : X; \ X \subseteq f(X \cup \text{gfp}(f))\) \(\implies a : \text{gfp}(f)\)

by (rule weak-coinduct[rotated], rule coinduct-lemma) blast+

lemma coinduct: \(\forall mono(f); \ X \leq f (\text{sup } X (\text{gfp } f))\) \(\implies X \leq \text{gfp}(f)\)

apply (rule order-trans)

apply (rule sup-cn1)

apply (rule gfp-upperbound)

apply (erule coinduct-lemma)

apply assumption
THEORY "Inductive"

done

lemma gfp-fun-UnI2: [\[ mono(f); a: gfp(f) \] ==\>= a: f(X Un gfp(f))
  by (blast dest: gfp-lemma2 mono-Un)

11.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f X$ to one expressed using both $lfp$ and $gfp$

lemma coinduct3-mono-lemma: mono(f) ==\> mono(\%x. f(x) Un X Un B)
  by (iprover intro: subset-refl monoI Un-mono monoD)

lemma coinduct3-lemma:
  [\[ X \subseteq f(lfp(\%x. f(x) Un X Un gfp(f)))); mono(f) \] ==\> lfp(\%x. f(x) Un X Un gfp(f)) \subseteq f(lfp(\%x. f(x) Un X Un gfp(f))))
  apply (rule subset-trans)
  apply (erule coinduct3-mono-lemma [THEN lfp-lemma3])
  apply (rule Un-least [THEN Un-least])
  apply (rule subset-refl, assumption)
  apply (rule gfp-unfold [THEN equalityD1, THEN subset-trans], assumption)
  apply (rule monoD, assumption)
  apply (subst coinduct3-mono-lemma [THEN lfp-unfold], auto)
  done

definition forms of $gfp$-unfold and coinduct, to control unfolding

lemma def-gfp-unfold: [\[ A==gfp(f); mono(f) \] ==\> A = f(A)
  by (auto intro!: gfp-unfold)

lemma def-coinduct:
  [\[ A==gfp(f); mono(f); X \subseteq f(sup X A) \] ==\> X \subseteq A
  by (iprover intro!: coinduct)

lemma def-coinduct-set:
  [\[ A==gfp(f); mono(f); a:X; X \subseteq f(X Un A) \] ==\> a: A
  by (auto intro!: coinduct-set)

lemma def-Collect-coinduct:
  [\[ A == gfp(\%w. Collect(P(w)))); mono(\%w. Collect(P(w)))
    a: X; !!z. z: X ==\> P (X Un A) z \] ==\>
  a : A
  by (erule def-coinduct-set) auto
lemma def-coinduct3:
  \[ A \Rightarrow \mathsf{gfp}(f); \ \mathsf{mono}(f); \ a : \mathcal{X}; \ X \subseteq f(lfp(%x. f(x) \cap X \cap A)) \] 
  \Rightarrow a : A
by (auto intro!: coinduct3)

Monotonicity of \( \mathsf{gfp} \)

lemma gfp-mono: (!!! Z. f Z \leq g Z) \Rightarrow \mathsf{gfp} f \leq \mathsf{gfp} g
by (rule gfp-upperbound [THEN gfp-least], blast intro: order-trans)

11.7 Inductive predicates and sets

Package setup.

theorems basic-monos =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool eq-conj
Collect-mono in-mono vimage-mono

ML-file Tools/inductive.ML
setup Inductive.setup

theorems [mono] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-mono not-mono
Ball-def Bex-def
induct-rulify-fallback

11.8 Inductive datatypes and primitive recursion

Package setup.

ML-file Tools/Datatype/datatype-aux.ML
ML-file Tools/Datatype/datatype-prop.ML
ML-file Tools/Datatype/datatype-data.ML setup Datatype-Data.setup
ML-file Tools/Datatype/rep-datatype.ML
ML-file Tools/Datatype/datatype-codegen.ML
ML-file Tools/Datatype/primrec.ML
ML-file Tools/BNF/bnf-fp-rec-sugar-util.ML
ML-file Tools/BNF/bnf-lfp-rec-sugar.ML

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: cases-syn => 'a => 'b
((%-) 10)
syntax (xsymbols)
-lam-pats-syntax :: cases-syn => 'a => 'b
((\-\) 10)

parse-translation \langle
let
  fun fun-tr cctx [cs] =
  let
THEORY “Product-Type”

val x = Syntax.free (fst (Name.variant x (Term.declare-term-frees cs Name.context)));
val ft = Case-Translation.case-tr true ctxt [x, cs];
in lambda x ft end
end

12 Product-Type: Cartesian products

theory Product-Type
imports Typedef Inductive Fun
keywords inductive-set coinductive-set :: thy-decl
begin

12.1 bool is a datatype

free-constructors case-bool for True | False
by auto

Avoid name clashes by prefixing the output of rep-datatype with old.
setup ⟨⟨ Sign.mandatory-path old ⟩⟩

rep-datatype True False by (auto intro: bool-induct)

setup ⟨⟨ Sign.parent-path ⟩⟩

But erase the prefix for properties that are not generated by free-constructors.
setup ⟨⟨ Sign.mandatory-path bool ⟩⟩

lemmas induct = old.bool.induct
lemmas inducts = old.bool.inducts
lemmas rec = old.bool.rec
lemmas simps = bool.distinct bool.case bool.rec

setup ⟨⟨ Sign.parent-path ⟩⟩

declare case-split [cases type: bool]
— prefer plain propositional version

lemma
  shows [code]: HOL.equal False P ⟷ ¬ P
  and [code]: HOL.equal True P ⟷ P
  and [code]: HOL.equal P False ⟷ ¬ P
  and [code]: HOL.equal P True ⟷ P
  and [code nbe]: HOL.equal P P ⟷ True
by (simp-all add: equal)
lemma If-case-cert:
  assumes CASE ≡ (λb. If b f g)
  shows (CASE True ≡ f) && (CASE False ≡ g)
  using assms by simp-all

setup (Code.add-case @{thm If-case-cert})

code-printing
  constant HOL.equal :: bool ⇒ bool ⇒ bool ⇒ (Haskell) infix 4 ==
  class-instance bool :: equal ⇒ (Haskell)

12.2 The unit type

typedef unit = {True} by auto

definition Unity :: unit ("'\)\('"")
  where () = Abs-unit True

lemma unit-eq [no-atp]: u = ()
  by (induct u) (simp add: Unity-def)

Simplification procedure for unit-eq. Cannot use this rule directly — it loops!

simproc-setup unit-eq (x::unit) = (fn - => fn - => fn ct =>>
  if HOLogic.is-unit (term-of ct) then NONE
  else SOME (mk-meta-eq @{thm unit-eq}))

free-constructors case-unit for () by auto

Avoid name clashes by prefixing the output of rep-datatype with old.

setup (Sign.mandatory-path old)

rep-datatype () by simp

setup (Sign.parent-path)

But erase the prefix for properties that are not generated by free-constructors.

setup (Sign.mandatory-path unit)

lemmas induct = old.unit.induct
lemmas inducts = old.unit.inducts
lemmas rec = old.unit.rec
lemmas simps = unit.case unit.rec

setup "Sign.parent-path"

lemma unit-all-eq1: (!!x::unit. PROP P x) == PROP P ()
  by simp

lemma unit-all-eq2: (!!x::unit. PROP P) == PROP P
  by (rule triv-forall-equality)

This rewrite counters the effect of simproc unit-eq on \%u::unit. f u, replacing
it by f rather than by \%u. f ()..

lemma unit-abs-eta-conv [simp]: (%u::unit. f ()) = f
  by (rule ext) simp

lemma UNIV-unit:
  UNIV = {()}
  by auto

instantiation unit :: default
begin

definition default = ()

instance ..
end

instantiation unit :: {complete-boolean-algebra, complete-linorder, wellorder}
begin

definition less-eq-unit :: unit ⇒ unit ⇒ bool
  where (_::unit) ≤ - ⟷ True

lemma less-eq-unit [iff]:
  (u::unit) ≤ v
  by (simp add: less-eq-unit-def)

definition less-unit :: unit ⇒ unit ⇒ bool
  where (_::unit) < - ⟷ False

lemma less-unit [iff]:
  ¬ (u::unit) < v
  by (simp-all add: less-eq-unit-def less-unit-def)

definition bot-unit :: unit
  where [code-unfold]: ⊥ = ()
**THEORY “Product-Type”**

*definition* `top-unit :: unit`

where

```
[code-unfold]: T = ()
```

*definition* `inf-unit :: unit ⇒ unit ⇒ unit`

where

```
[simp]: - ∩ - = ()
```

*definition* `sup-unit :: unit ⇒ unit ⇒ unit`

where

```
[simp]: - ⊔ - = ()
```

*definition* `Inf-unit :: unit set ⇒ unit`

where

```
[simp]: ⋂ - = ()
```

*definition* `Sup-unit :: unit set ⇒ unit`

where

```
[simp]: ⋃ - = ()
```

*definition* `uminus-unit :: unit ⇒ unit`

where

```
[simp]: - - = ()
```

*declare* `less-eq-unit-def [abs-def, code-unfold]

less-unit-def [abs-def, code-unfold]

inf-unit-def [abs-def, code-unfold]

sup-unit-def [abs-def, code-unfold]

Inf-unit-def [abs-def, code-unfold]

Sup-unit-def [abs-def, code-unfold]

uminus-unit-def [abs-def, code-unfold]

*instance*

by intro-classes auto

*end*

**lemma** [code]:

\[ \text{HOL.equal } (u::unit) \leftrightarrow \text{True} \]

unfolding \text{equal unit-eq [of u] unit-eq [of v]} by rule+

**code-printing**

*type-constructor* `unit →`

(SML) unit

and (OCaml) unit

and (Haskell) ()

and (Scala) Unit

| constant `Unity` →
12.3 The product type

12.3.1 Type definition

definition Pair-Rep :: 
    'a ⇒ 'b ⇒ 'a ⇒ 'b ⇒ bool
where
Pair-Rep a b = (λ x y. x = a ∧ y = b)

typedef ('a, 'b) prod (infixr * 20) = prod :: ('a ⇒ 'b ⇒ bool) set

typedef ('a, 'b) prod (infixr ∗ 20)

lemma prod-cases: (∀ a b. P (Pair a b)) ⟹ P p
by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

free-constructors case-prod for Pair fst snd

proof –
  fix P :: bool and p :: 'a × 'b
  show (∀ x1 x2. p = Pair x1 x2 ⟹ P) ⟹ P
    by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

next
  fix a c :: 'a and b d :: 'b
  have Pair-Rep a b = Pair-Rep c d ⟷ a = c ∧ b = d
    by (auto simp add: Pair-Rep-def fun-eq-iff)
moreover have Pair-Rep \(a \ b \in \text{prod}\) and \(Pair-Rep \ c \ d \in \text{prod}\)
by (auto simp add: prod-def)
ultimately show \(Pair \ a \ b = Pair \ c \ d\) \(\iff\) \(a = c \wedge b = d\)
by (simp add: Pair-def Abs-prod-inject)
qed

Avoid name clashes by prefixing the output of rep-datatype with \(old\).

setup \(\langle\langle\text{Sign.mandatory-path old}\rangle\rangle\)

rep-datatype \(Pair\)
by (erule prod-cases) (rule prod.inject)

setup \(\langle\langle\text{Sign.parent-path}\rangle\rangle\)
But erase the prefix for properties that are not generated by free-constructors.

setup \(\langle\langle\text{Sign.mandatory-path prod}\rangle\rangle\)

declare \(old.prod.inject[currant \iff\ del]\)

lemmas induct = old.prod.induct
lemmas inducts = old.prod.inducts
lemmas rec = old.prod.rec
lemmas simps = prod.inject prod.case prod.rec

setup \(\langle\langle\text{Sign.parent-path}\rangle\rangle\)

declare prod.case [nitpick-simp \(\del\)]
declare prod.weak-case-cong [cong \(\del\)]

12.3.2 Tuple syntax

abbreviation \((input)\) \(\text{split} :: (\ 'a \Rightarrow \ 'b \Rightarrow \ 'c) \Rightarrow \ 'a \times \ 'b \Rightarrow \ 'c\) \(where\)
\(\text{split} \equiv \text{case-prod}\)

Patterns – extends pre-defined type pttrn used in abstractions.

nonterminal \(\text{tuple-args}\) and \(\text{patterns}\)

syntax
\(-\text{tuple} :: \ 'a \Rightarrow \ \text{tuple-args} \Rightarrow \ 'a \times \ 'b \ (\text{\(I'\)}\ (-/-)\))
\(-\text{tuple-arg} :: \ 'a \Rightarrow \ \text{tuple-args} \ (-)\)
\(-\text{tuple-args} :: \ 'a \Rightarrow \ \text{tuple-args} \Rightarrow \ \text{tuple-args} \ (-/-)\)
\(-\text{-pattern} :: [\text{pttrn}, \text{patterns}] \Rightarrow \ \text{pttrn} \ (\text{\(I'\)}\ (-/-)\))
\(-\text{-patterns} :: [\text{pttrn}, \text{patterns}] \Rightarrow \ \text{patterns} \ (-/-)\)

translations
\((x, y) \equiv \text{CONST Pair} \ x \ y\)
\(-\text{-pattern} x \ y \Rightarrow \text{CONST Pair} \ x \ y\)
THEORY “Product-Type”

-\textit{patterns} $x, y \Rightarrow \text{CONST} \ Pair \ x \ y$

-\textit{tuple} $x \ (-\textit{tuple-args} \ y \ z) = \text{-\textit{tuple}} \ x \ (-\textit{tuple} \ (-\textit{tuple} \ y \ z))$

$\%(x, y, zs), \ b = \text{CONST} \ \text{case-prod} \ \%(x, y, zs), \ b \)$

$\%(x, y), \ b = \text{CONST} \ \text{case-prod} \ \%(x, y, b) \)$

-\textit{abs} $\ (\text{CONST} \ \text{Pair} \ x \ y) \ t \Rightarrow \%(x, y). \ t \)$

— The last rule accommodates tuples in ‘case C ... (x,y) ... =¿ ...’ The (x,y) is parsed as ‘Pair x y’ because it is logic, not pttrn

\textbf{print-translation} \ll
\begin{verbatim}
let
  fan split-tr' [Abs (x, T, t as (Abs abs))] =
  (* split ($\%x \ y, t) => ($\%x, y, t) * *)
  let
    val (y, t') = Syntax-Trans.atomic-abs-tr' abs;
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, T, t');
    in
      Syntax.const @{syntax-const -abs} $(
        Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
  end
| split-tr' [Abs (x, T, (s as Const (@{const-syntax case-prod}, -) $ t))] =
  (* split ($\%x. \ (split ($\%y z. t))) => ($\%x, y, z, t) * *)
  let
    val Const (@{syntax-const -abs}, -) $ (Const (@{syntax-const -pattern}, -) $ y $ z) $ t' = split-tr' [t];
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, T, t');
    in
      Syntax.const @{syntax-const -abs} $(
        Syntax.const @{syntax-const -pattern} $ x' $ y $ z) $ t''
  end
| split-tr' [Const (@{const-syntax case-prod}, -) $ t] =
  (* split ($\%x \ y \ z. t) => ($\%x, y, z, t) * *)
  split-tr' [split-tr' [t]] (* inner split-tr' creates next pattern *)
| split-tr' [Const (@{syntax-const -abs}, -) $ x-y $ Abs abs] =
  (* split ($\%pttrn \ z. \ t) => ($\%pttrn, z, t) * *)
  let val (z, t) = Syntax-Trans.atomic-abs-tr' abs in
    Syntax.const @{syntax-const -abs} $(
      Syntax.const @{syntax-const -pattern} $ x-y $ z) $ t
  end
| split-tr' - = raise Match;
  in
    ([@{const-syntax case-prod}, K split-tr'] end
\end{verbatim}

\textbf{typed-print-translation} \ll
\begin{verbatim}
let
  fan split-guess-names-tr' T [Abs (x, - as Abs -)] = raise Match
| split-guess-names-tr' T [Abs (x, xT, t)] =
\end{verbatim}
(case (head-of t) of
  Const (@{const-syntax case-prod}, -) => raise Match
| - =>
  let
    val (:: yT :: -) = binder-types (domain-type T) handle Bind => raise Match;
    val (y, t) = Syntax.Trans.atomic-abs-tr' (y, yT, incr-boundvars 1 t $ Bound 0);
    val (x', t') = Syntax.Trans.atomic-abs-tr' (x, xT, t');
    in
      Syntax.const @{syntax-const -abs} $ (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
    end)
  | split-guess-names-tr' T [t] =
    (case head-of t of
      Const (@{const-syntax case-prod}, -) => raise Match
      | - =>
        let
          val (xT :: yT :: -) = binder-types (domain-type T) handle Bind => raise Match;
          val (y, t) = Syntax.Trans.atomic-abs-tr' (y, yT, incr-boundvars 1 t)
          in
            Syntax.const @{syntax-const -abs} $ (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
          end)
      | split-guess-names-tr' - - = raise Match;
      in
        [(@{const-syntax case-prod}, K split-guess-names-tr')] end
    )
  }>

print-translation ≡
  let
    fun contract Q tr ctxt ts =
      (case ts of
        [A, Abs (-, -, (s as Const (@{const-syntax case-prod},-) $ t) $ Bound 0)] =>>
          if Term.is-dependent t then tr ctxt ts
          else Syntax.const Q $ A $ s
        | - => tr ctxt ts);
      in
        Syntax.Trans.preserve-binder-abs2-tr' @{syntax-const Ball} @{syntax-const -Ball},
        Syntax.Trans.preserve-binder-abs2-tr' @{syntax-const Bex} @{syntax-const -Bex},
        Syntax.Trans.preserve-binder-abs2-tr' @{syntax-const INFIMUM} @{syntax-const -INF},
12.3.3 Code generator setup

code-printing

type-constructor prod $\mapsto$

| (SML) infix 2 *
| and (OCaml) infix 2 *
| and (Haskell) !((-,)/ (-))
| and (Scala) !((-),/ (-))

| constant Pair $\mapsto$
| (SML) !((-,)/ (-))
| and (OCaml) !((-,)/ (-))
| and (Haskell) !((-,)/ (-))
| and (Scala) !((-),/ (-))

| class-instance prod :: equal $\mapsto$
| (Haskell) -
| constant HOL.equal :: 'a × 'b $\Rightarrow$ 'a × 'b $\Rightarrow$ bool $\mapsto$
| (Haskell) infix 4 ==

12.3.4 Fundamental operations and properties

lemma Pair-inject:

- assumes $(a, b) = (a', b')$
- and $a = a' ==> b = b' ==> R$
- shows $R$
- using assms by simp

lemma surj-pair [simp]: EX x y. p = (x, y)
- by (cases p) simp

code-printing

| constant fst $\mapsto$ (Haskell) fst
| constant snd $\mapsto$ (Haskell) snd

lemma case-prod-unfold [nitpick-unfold]: case-prod = (%c p. c (fst p) (snd p))
- by (simp add: fun-eq-iff split: prod.split)

lemma fst-eqD: fst (x, y) = a ==> x = a
- by simp

lemma snd-eqD: snd (x, y) = a ==> y = a
- by simp

lemmas surjective-pairing = prod.collapse [symmetric]
theory "Product-Type"

lemma prod-eq-iff: s = t ←→ fst s = fst t ∧ snd s = snd t
  by (cases s, cases t) simp

lemma prod-eqI [intro?]: fst p = fst q ⇒ snd p = snd q ⇒ p = q
  by (simp add: prod-eq-iff)

lemma split-conv [simp, code]: split f (a, b) = f a b
  by (fact prod.case)

lemma splitI: f a b ⇒ split f (a, b)
  by (rule split-conv [THEN iffD2])

lemma splitD: split f (a, b) ⇒ f a b
  by (rule split-conv [THEN iffD1])

lemma split-Pair [simp]: (λ(x, y). (x, y)) = id
  by (simp add: fun-eq-iff split prod.split)

lemma split-comp: split (f ∘ g) x = f (g (fst x)) (snd x)
  by (cases x) simp

lemma split-twice: split f (split g p) = split (λx y. split f (g x y)) p
  by (cases p) simp

lemma The-split: The (split P) = (THE xy. P (fst xy) (snd xy))
  by (simp add: case-prod-unfold)

lemma split-weak-cong: p = q ⇒ split c p = split c q
  — Prevents simplification of c: much faster
  by (fact prod.weak-case-cong)

lemma cond-split-eta: (!!(x, y). f x y = g (x, y)) ==⇒ (%(x, y). f x y) = g
  by (simp add: split-eta)

lemma split-paired-all [no-atp]: (!!x. PROP P x) == (!!a b. PROP P (a, b))
  proof
    fix a b
    assume !!x. PROP P x
    then show PROP P (a, b).
  next
    fix x
    assume !!a b. PROP P (a, b)
    from ⟨PROP P (fst x, snd x)⟩ show PROP P x by simp
  qed
lemma case-prod-distrib: \( f \text{(case } x \text{ of } (x, y) \Rightarrow g x y) = \text{(case } x \text{ of } (x, y) \Rightarrow f (g x y)) \)  
by (cases \( x \)) simp

The rule `split-paired-all` does not work with the Simplifier because it also affects premises in congruence rules, where this can lead to premises of the form \( \forall a \ b \ldots = ?P(a, b) \) which cannot be solved by reflexivity.

lemmas split-tupled-all = split-paired-all unit-all-eq2

ML ⟨⟨  
(* replace parameters of product type by individual component parameters *)  
local (* filtering with exists-paired-all is an essential optimization *)  
fun exists-paired-all (Const (@{const-name Pure.all}, _) $ Abs (_, T, t)) =  
  can HOLogic.dest_prodT T orelse exists-paired-all t  
| exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u  
| exists-paired-all (Abs (_, _, t)) = exists-paired-all t  
| exists-paired-all _ = false;  
val ss =  
simpsset-of  
(let simpset = HOLBasicSet
  putsimps (@{context} [thm split-paired-all], @{thm unit-all-eq2}, @{thm unit-abs-eta-conv})
  addsimpprocs (@{simproc unit-eq})
) ;
in  
fun split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then safe-full-simp-tac (put-simpset ss ctxt) i else no-tac);
fun unsafe-split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then full-simp-tac (put-simpset ss ctxt) i else no-tac);

fun split-all ctxt th =
  if exists-paired-all (Thm.prop_of th)
  then full-simplify (put-simpset ss ctxt) th else th;
end;  
⟩⟩

setup ⟨⟨ map-theory-claset (fn ctxt => ctxt addSbefore (split-all-tac, split-all-tac)) ⟩⟩

lemma split-paired-All [simp, no-atp]: \( \forall x \cdot P x = \forall a \ b \cdot P (a, b) \)  
— [iff] is not a good idea because it makes `blast` loop  
by fast

lemma split-paired-Ex [simp, no-atp]: \( \exists x \cdot P x = \exists a \ b \cdot P (a, b) \)  
by fast

lemma split-paired-The [no-atp]: \( \text{THE} x \cdot P x = \text{THE} (a, b) \cdot P (a, b) \)  
— Can’t be added to simpset: loops!  
by (simp add: split-eta)
Simplification procedure for \textit{cond-split-eta}. Using \textit{split-eta} as a rewrite rule is not general enough, and using \textit{cond-split-eta} directly would render some existing proofs very inefficient; similarly for \textit{split-beta}.

\begin{verbatim}
ML \langle\langle
  local
  val cond-split-eta-ss = simpset-of (put-simpset HOL-basic-ss @{context} addsimps @{thms cond-split-eta});
  fun Pair-pat k 0 (Bound m) = (m = k)
    | Pair-pat k i (Const @{const Pair}, _) $ Bound m $ t) = 
i > 0 andalso m = k + i andalso Pair-pat k (i - 1) t
    | Pair-pat _ _ _ false;
  fun no-args k i (Abs (_, _, t)) = no-args (k + 1) i t
    | no-args k i (t $ u) = no-args k (i t) andalso no-args k i u
    | no-args _ _ _ true;
  fun split-pat tp i = if tp 0 i t then SOME (i, t) else NONE
    | split-pat tp i (Const @{const_case_prod}, _) $ Abs (_, _, t)) = split-pat tp (i + 1) t
    | split-pat tp i = NONE;
  fun metaeq ctxt lhs rhs = mk-meta-eq (Goal.prove ctxt [] []
    (HOLogic.mk_Trueprop (HOLogic.mk_eq (lhs, rhs)))
    (K (simp-tac (put-simpset cond-split-eta-ss ctxt) 1)))

  fun beta-term-pat k i (Abs (_, _, t)) = beta-term-pat (k + 1) i t
    | beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse beta-term-pat k i t andalso beta-term-pat k i u
    | beta-term-pat _ _ _ false;
  fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
    | eta-term-pat _ _ _ false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
    | subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
      else (subst arg k i t) $ subst arg k i u
    | subst arg k i t = t;
  in
  fun beta-proc ctxt (s as Const @{const_case_prod}, _) $ Abs (_, _, t) $ arg) =
    (case split-pat beta-term-pat 1 t of
      SOME (i, f) => SOME (metaeq ctxt s (subst arg 0 i f))
    | NONE => NONE)
    | beta-proc _ _ _ NONE;
  fun eta-proc ctxt (s as Const @{const_case_prod}, _) $ Abs (_, _, t) =
    (case split-pat eta-term-pat 1 t of
      SOME (_, ft) => SOME (metaeq ctxt s (let val (f $ arg) = ft in ft end))
    | NONE => NONE)
    | eta-proc _ _ _ NONE;
  end;

simproc-setup split-beta (split f z) = \langle\langle fn - => fn ctxt => fn ct => beta-proc
\end{verbatim}
THEORY "Product-Type"

\[
\begin{align*}
\text{ctxt} & \text{ (term-of ct) } \\
\text{simproc-setup} & \text{ split-eta (split f) } = \langle \text{fn - => fn ctxt => fn ct => eta-proc ctxt (term-of ct)} \rangle
\end{align*}
\]

\textbf{lemma} split-beta \[\text{mono}]: \((\lambda(x, y). P x y) z = P (\text{fst } z) (\text{snd } z)\]
\begin{itemize}
\item by \((\text{subst surjective-pairing, rule split-cone})\)
\end{itemize}

\textbf{lemma} split-beta': \((\lambda(x, y). f x y) = (\lambda x. f (\text{fst } x) (\text{snd } x))\)
\begin{itemize}
\item by \((\text{auto simp: fun-eq-iff})\)
\end{itemize}

\textbf{lemma} split-split \[\text{no-atp}]: \(R (\text{split } c p) = (\text{ALL } x y. p = (x, y) -\rightarrow R(c x y))\)
\begin{itemize}
\item for use with \text{split} and the Simplifier.
\item by \((\text{insert surj-pair [of p], clarify, simp})\)
\end{itemize}

\textbf{case-prod} used as a logical connective or set former.

These rules are for use with \text{blast}; could instead call \text{simp} using \text{prod.split} as rewrite.

\textbf{lemma} splitI2: \(!! p. [!] a b. p = (a, b) -\rightarrow c a b [!] -\rightarrow \text{split } c p\)
\begin{itemize}
\item apply \((\text{simp only: split-tupled-all})\)
\item apply \((\text{simp (no-asm-simp)})\)
\item done
\end{itemize}

\textbf{lemma} splitI2': \(!! p. [!] a b. (a, b) = p -\rightarrow c a b x [!] -\rightarrow \text{split } c p x\)
\begin{itemize}
\item apply \((\text{simp only: split-tupled-all})\)
\item apply \((\text{simp (no-asm-simp)})\)
\item done
\end{itemize}

\textbf{lemma} splitE: \(\text{split } c p -\rightarrow (\text{ALL } x y. p = (x, y) -\rightarrow c x y -\rightarrow Q) -\rightarrow Q\)
\begin{itemize}
\item by \((\text{induct p}) \text{ auto}\)
\end{itemize}

\textbf{lemma} splitE': \(\text{split } c p z -\rightarrow (\text{ALL } x y. p = (x, y) -\rightarrow c x y z -\rightarrow Q) -\rightarrow Q\)
\begin{itemize}
\item by \((\text{induct p}) \text{ auto}\)
\end{itemize}

\textbf{lemma} splitE2:
\[
[! Q (\text{split } P z); \text{!!y. [!] z = (x, y); Q (P x y)]} -\rightarrow R]\text{ [!] -\rightarrow R}
\]
\begin{itemize}
\item proof
\item assume \(q: Q (\text{split } P z)\)
\item assume \(r: \text{!!y. [!] z = (x, y); Q (P x y)] -\rightarrow R}\)
\item show \(R\)
\end{itemize}
apply (rule r surjective-pairing)+
apply (rule split-beta [THEN subst], rule q)
done

qed

lemma splitD': split R (a,b) c == R a b c
  by simp

lemma mem-splitI: z: c a b ==> z: split c (a, b)
  by simp

lemma mem-splitI2: !!p. [| !!a. p (a, b) ==> z: c a b |] ==> z: split c p
  by (simp only: split-tupled-all, simp)

lemma mem-splitE:
  assumes major: z ∈ split c p
  and cases: ⋀ x y. p = (x, y) ==> z ∈ c x y ==> Q
  shows Q
  by (rule major [unfolded case-prod-unfold] cases surjective-pairing)+

declare mem-splitI2 [intro!] mem-splitI [intro!]
splitI2' [intro!] splitI2 [intro!]
splitI [intro!]
declare mem-splitE [elim!] splitE' [elim!] splitE [elim!]

ML ⟨⟨
local (* filtering with exists-p-split is an essential optimization *)
  fun exists-p-split (Const (@{const-name case-prod}, - $ (Const (@{const-name Pair}, - $ -)))) = true
    | exists-p-split (t $ u) = exists-p-split t orelse exists-p-split u
    | exists-p-split (Abs (_, - , t)) = exists-p-split t
    | exists-p-split _ = false;
in
fun split-conv-tac ctxt = SUBGOAL (fn (t, i) =>
if exists-p-split t
  then safe-full-simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms split-conv})
i
  else no-tac);
end;
⟩⟩

setup ⟨⟨
map-theory-claset (fn ctxt => ctxt addsbefore (split-conv-tac, split-conv-tac))
⟩⟩

lemma split-eta-SetCompr [simp, no-atp]: (%u. EX x y. u = (x, y) & P (x, y))
  = P
  by (rule ext) fast

lemma split-eta-SetCompr2 [simp, no-atp]: (%u. EX x y. u = (x, y) & P x y) =
split \( P \)
by \( \text{rule ext} \) fast

lemma split-part \( \text{[simp]}: \langle \% (a,b). \ P \land Q \ a \ b \rangle = \langle \% \text{ab.} \ P \land \text{split} \ Q \ \text{ab} \rangle \)
— Allows simplifications of nested splits in case of independent predicates.
by \( \text{rule ext} \) blast

lemma split-comp-eq:
fixes \( \text{f} :: \text{'}a = \text{'}b = \text{'}c \ \text{and} \ g :: \text{'}d = \text{'}a \)
shows \( \langle \% u. \ (g \ (\text{fst} \ u)) \ (\text{snd} \ u) \rangle = \langle \text{split} \ \langle \% x. \ f \ (g \ x) \rangle \rangle \)
by \( \text{rule ext} \) auto

lemma pair-imageI \( \text{[intro]}: (a, b) : A ==> f a b : \langle \% (a, b). f a b \rangle \ \text{'} A \)
apply \( \text{rule-tac} \ x = (a, b) \ \text{in} \ \text{image-eqI} \)
apply auto
done

lemma The-split-eq \( \text{[simp]}: \langle \text{THE} \ (x',y'). \ x = x' \land y = y' \rangle = (x, y) \)
by blast

Setup of internal \text{split-rule}.

lemmas case-prodI = prod.case \( \text{[THEN iffD2]} \)

lemma case-prodI2: \( \text{[]} \ p. \ [] \text{[]} a b. \ p = (a, b) ==> c a b [] \Longrightarrow \text{case-prod} \ c \ p \)
by \( \text{fact splitI2} \)

lemma case-prodI2': \( \text{[]} \ p. \ [] \text{[]} a b. \ (a, b) = p ==> c a b x [] \Longrightarrow \text{case-prod} \ c \ p \ x \)
by \( \text{fact splitI2'} \)

lemma case-prodE: case-prod \ c \ p ==> \( \text{[]} \text{[]} \ y. \ p = (x, y) ==> c x y ==> Q \)
==> Q
by \( \text{fact splitE} \)

lemma case-prodE': case-prod \ c \ p \ z ==> \( \text{[]} \text{[]} \ y. \ p = (x, y) ==> c x y z ==> Q \)
==> Q
by \( \text{fact splitE'} \)

declare case-prodI \( \text{[intro]} \)

lemma case-prod-beta:
case-prod \ f \ p = f \ (\text{fst} \ p) \ (\text{snd} \ p) \)
by \( \text{fact split-beta} \)

lemma prod-cases3 \( \text{[cases type]}: \)
obtains \( \text{fields} \ a \ b \ c \ \text{where} \ y = (a, b, c) \)
by \( \text{cases} \ y, \ \text{case-tac} \ b \) blast

lemma prod-induct3 \( \text{[case-names fields, induct type]}: \)
THEORY “Product-Type”

(!!a b c. P (a, b, c)) ==> P x
by (cases x) blast

lemma prod-cases4 [cases type]:
 obtains (fields) a b c d where y = (a, b, c, d)
by (cases y, case-tac c) blast

lemma prod-induct4 [case-names fields, induct type]:
 (!!a b c d. P (a, b, c, d)) ==> P x
by (cases x) blast

lemma prod-cases5 [cases type]:
 obtains (fields) a b c d e where y = (a, b, c, d, e)
by (cases y, case-tac d) blast

lemma prod-induct5 [case-names fields, induct type]:
 (!!a b c d e. P (a, b, c, d, e)) ==> P x
by (cases x) blast

lemma prod-cases6 [cases type]:
 obtains (fields) a b c d e f where y = (a, b, c, d, e, f)
by (cases y, case-tac e) blast

lemma prod-induct6 [case-names fields, induct type]:
 (!!a b c d e f. P (a, b, c, d, e, f)) ==> P x
by (cases x) blast

lemma prod-cases7 [cases type]:
 obtains (fields) a b c d e f g where y = (a, b, c, d, e, f, g)
by (cases y, case-tac f) blast

lemma prod-induct7 [case-names fields, induct type]:
 (!!a b c d e f g. P (a, b, c, d, e, f, g)) ==> P x
by (cases x) blast

lemma split-def:
 split = (\lambda c p. c (fst p) (snd p))
by (fact case-prod-unfold)

definition internal-split :: ('a => 'b => 'c) => 'a * 'b => 'c where
 internal-split == split

lemma internal-split-conv: internal-split c (a, b) = c a b
by (simp only: internal-split-def split-conv)

ML-file Tools/split-rule.ML
setup Split-Rule.setup

hide-const internal-split
12.3.5 Derived operations

**definition curry** :: ('a × 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c where

* curry = (λc x y. c (x, y))

**lemma curry-conv** [simp, code]: curry f a b = f (a, b)  
by (simp add: curry-def)

**lemma curryI** [intro!]: f (a, b) ⇒ curry f a b  
by (simp add: curry-def)

**lemma curryD** [dest!]: curry f a b ⇒ f (a, b)  
by (simp add: curry-def)

**lemma curryE**: curry f a b ⇒ (f (a, b) ⇒ Q) ⇒ Q  
by (simp add: curry-def)

**lemma curry-K**: curry (λx. c) = (λx y. c)  
by (simp add: fun-eq-iff)

The composition-uncurry combinator.

**notation fcomp** (infixl ◦> 60)

**definition scomp** :: ('a ⇒ 'b × 'c) ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'd (infixl ◦→ 60)  
where

* f ◦→ g = (λx. case-prod g (f x))

**lemma scomp-unfold**: scomp = (λf g x. g (fst (f x)) (snd (f x)))  
by (simp add: fun-eq-iff scomp-def case-prod-unfold)

**lemma scomp-apply** [simp]: (f ◦→ g) x = case-prod g (f x)  
by (simp add: scomp-unfold case-prod-unfold)

**lemma Pair-scomp**: Pair x ◦→ f = f x  
by (simp add: fun-eq-iff)

**lemma scomp-Pair**: x ◦→ Pair = x  
by (simp add: fun-eq-iff)

**lemma scomp-scomp**: (f ◦→ g) ◦→ h = f ◦→ (λx. g x ◦→ h)  
by (simp add: fun-eq-iff scomp-unfold)

**lemma scomp-fcomp**: (f ◦→ g) ◦> h = f ◦→ (λx. g x ◦> h)  
by (simp add: fun-eq-iff scomp-unfold fcomp-def)
THEORY "Product-Type"

lemma fcomp-scomp: \((f \circ g) \circ h = f \circ (g \circ h)\)
  by (simp add: fun-eq-iff scomp-unfold)

code-printing
  constant scomp \(\mapsto\) (Eval)
  infixl 3 \#ә

no-notation fcomp (infixl \(\circ\) 60)
no-notation scomp (infixl \(\circ\) \#ә 60)

map-prod — action of the product functor upon functions.

definition map-prod :: \('
\) \('a \Rightarrow \') \(\Rightarrow \) \('b \Rightarrow \') \(\Rightarrow \) \('c \times \) \('d \Rightarrow \) \('a \times \) \('b \Rightarrow \) \('c \times \) \('d\) where

  map-prod f g = (\(\lambda\) (x, y). (f x, g y))

lemma map-prod-simp [simp, code]:

  map-prod f g (a, b) = (f a, g b)
  by (simp add: map-prod-def)

functor map-prod: map-prod
  by (auto simp add: split-paired-all)

lemma fst-map-prod [simp]:

  fst (map-prod f g x) = f (fst x)
  by (cases x) simp-all

lemma snd-prod-fun [simp]:

  snd (map-prod f g x) = g (snd x)
  by (cases x) simp-all

lemma fst-comp-map-prod [simp]:

  fst \circ map-prod f g = f \circ fst
  by (rule ext) simp-all

lemma snd-comp-map-prod [simp]:

  snd \circ map-prod f g = g \circ snd
  by (rule ext) simp-all

lemma map-prod-compose:

  map-prod (f1 \circ f2) (g1 \circ g2) = (map-prod f1 g1 \circ map-prod f2 g2)
  by (rule ext) (simp add: map-prod.compositionality comp-def)

lemma map-prod-ident [simp]:

  map-prod (%x. x) (%y. y) = (%z. z)
  by (rule ext) (simp add: map-prod.identity)

lemma map-prod-imageI [intro]:

  \((a, b) \in R \implies (f a, g b) \in map-prod f g \circ R\)
  by (rule image-eqI) simp-all
lemma prod-fun-imageE [elim!]:
assumes major: \( c \in \text{map-prod} \ f \ g \ R \)
and cases: \( \forall x \ y. \ c = (f \ x, \ g \ y) \iff (x, \ y) \in R \implies P \)
shows \( P \)
apply (rule major [THEN imageE])
apply (case-tac x)
apply (rule cases)
apply simp-all
done

definition apfst :: \( 'a \Rightarrow 'c \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b \) where
apfst \( f \) = map-prod \( f \) id

definition apsnd :: \( 'b \Rightarrow 'c \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c \) where
apsnd \( f \) = map-prod id \( f \)

lemma apfst-conv [simp, code]:
apfst \( f \) \((x, y)\) = \((f \ x, y)\)
by (simp add: apfst-def)

lemma apsnd-conv [simp, code]:
apsnd \( f \) \((x, y)\) = \((x, f \ y)\)
by (simp add: apsnd-def)

lemma fst-apfst [simp]:
fst (apfst \( f \) \( x \)) = \( f \) (fst \( x \))
by (cases \( x \)) simp

lemma fst-comp-apfst [simp]:
fst \circ apfst \( f \) = \( f \) \circ fst
by (simp add: fun-eq-iff)

lemma fst-apsnd [simp]:
fst (apsnd \( f \) \( x \)) = fst \( x \)
by (cases \( x \)) simp

lemma fst-comp-apsnd [simp]:
fst \circ apsnd \( f \) = \( f \)
by (simp add: fun-eq-iff)

lemma snd-apfst [simp]:
snd (apfst \( f \) \( x \)) = snd \( x \)
by (cases \( x \)) simp

lemma snd-comp-apfst [simp]:
snd \circ apfst \( f \) = snd
by (simp add: fun-eq-iff)

lemma snd-apsnd [simp]:
THEORY "Product-Type"

\[
\text{snd} \ (\text{apsnd} \ f \ x) = f \ (\text{snd} \ x)
\]
by (cases x) simp

**lemma** \text{snd-comp-apsnd} [simp]:
\[
\text{snd} \circ \text{apsnd} \ f = f \circ \text{snd}
\]
by (simp add: fun-eq-iff)

**lemma** \text{apfst-compose}:
\[
\text{apfst} \ f \ (\text{apfst} \ g \ x) = \text{apfst} \ (f \circ g) \ x
\]
by (cases x) simp

**lemma** \text{apsnd-compose}:
\[
\text{apsnd} \ f \ (\text{apsnd} \ g \ x) = \text{apsnd} \ (f \circ g) \ x
\]
by (cases x) simp

**lemma** \text{apfst-apsnd} [simp]:
\[
\text{apfst} \ f \ (\text{apsnd} \ g \ x) = (f \ (\text{fst} \ x), g \ (\text{snd} \ x))
\]
by (cases x) simp

**lemma** \text{apsnd-apfst} [simp]:
\[
\text{apsnd} \ f \ (\text{apfst} \ g \ x) = (g \ (\text{fst} \ x), f \ (\text{snd} \ x))
\]
by (cases x) simp

**lemma** \text{apfst-id} [simp] :
\[
\text{apfst} \ id = id
\]
by (simp add: fun-eq-iff)

**lemma** \text{apsnd-id} [simp] :
\[
\text{apsnd} \ id = id
\]
by (simp add: fun-eq-iff)

**lemma** \text{apfst-eq-conv} [simp]:
\[
\text{apfst} \ f \ x = \text{apfst} \ g \ x \iff f \ (\text{fst} \ x) = g \ (\text{fst} \ x)
\]
by (cases x) simp

**lemma** \text{apsnd-eq-conv} [simp]:
\[
\text{apsnd} \ f \ x = \text{apsnd} \ g \ x \iff f \ (\text{snd} \ x) = g \ (\text{snd} \ x)
\]
by (cases x) simp

**lemma** \text{apsnd-apfst-commute}:
\[
\text{apsnd} \ f \ (\text{apfst} \ g \ p) = \text{apfst} \ g \ (\text{apsnd} \ f \ p)
\]
by simp

**context**

**begin**

**local-setup** 「Local-Theory.map-naming (Name-Space.mandatory-path prod)」

**definition** \text{swap} :: 'a × 'b ⇒ 'b × 'a
where
  \(\text{swap } p = (\text{snd } p, \text{fst } p)\)
end

lemma \text{swap-simp} [simp]:
  \(\text{prod.swap } (x, y) = (y, x)\)
  by (simp add: prod.swap-def)

lemma \text{pair-in-swap-image} [simp]:
  \((y, x) \in \text{prod.swap } ' A \iff (x, y) \in ' A\)
  by (auto intro!: image-eqI)

lemma \text{inj-swap} [simp]:
  inj-on prod.swap ' A
  by (rule inj-onI) auto

lemma \text{swap-inj-on}:
  inj-on (\lambda (i, j). (j, i)) ' A
  by (rule inj-onI) auto

lemma \text{case-swap} [simp]:
  (case prod.swap p of (y, x) => f x y) = (case p of (x, y) => f x y)
  by (cases p) simp

Disjoint union of a family of sets – Sigma.
definition \text{Sigma} :: 'a set => ('a => 'b set) => ('a × 'b) set where
  \text{Sigma-def}: Sigma A B == UN x:A. UN y:B x.{Pair x y}

abbreviation \text{Times} :: 'a set => 'b set => ('a × 'b) set
  (infixr "\*" 80) where
  A "\*" B == Sigma A (%-. B)

notation (xsymbols)
  Times (infixr \times 80)

notation (HTML output)
  Times (infixr \times 80)

hide-const (open) Times

syntax
  -Sigma :: [pttrn, 'a set, 'b set] => ('a × 'b) set ((3SIGMA ::= /) [0, 0, 10] 10)

translations
  \(\text{SIGMA } x:A. B \iff \text{CONST Sigma } A \%x. B\)

lemma \text{SigmaI} [intro!]: \| a:A; b:B(a) \| ==> (a,b) : Sigma A B
  by (unfold Sigma-def) blast
lemma SigmaE [elim!]:
    ![x y.] x:A; y:B(x); c=(x,y) ![]==> P
    ![]==> P
    — The general elimination rule.
    by (unfold Sigma-def) blast

Elimination of \((a, b) \in A \times B\) – introduces no eigenvariables.

lemma SigmaD1: (a, b) : Sigma A B ==> a : A
    by blast

lemma SigmaD2: (a, b) : Sigma A B ==> b : B a
    by blast

lemma SigmaE2:
    ![ (a, b) : Sigma A B;
        ![ a:A; b:B(a) ![]==> P
    ![]==> P
    by blast

lemma Sigma-cong:
    ![ A = B; ![x. x \in B \Longrightarrow C x = D x]
    ![==> (SIGMA x: A. C x) = (SIGMA x: B. D x)
    by auto

lemma Sigma-mono: ![ A <= C; ![x. x:A ==> B x <= D x ![]==> Sigma A B
    <= Sigma C D
    by blast

lemma Sigma-empty1 [simp]: Sigma {} B = {}
    by blast

lemma Sigma-empty2 [simp]: A ** {} = {}
    by blast

lemma UNIV-Times-UNIV [simp]: UNIV ** UNIV = UNIV
    by auto

lemma Compl-Times-UNIV1 [simp]: – (UNIV ** A) = UNIV ** (–A)
    by auto

lemma Compl-Times-UNIV2 [simp]: – (A ** UNIV) = (–A) ** UNIV
    by auto

lemma mem-Sigma-iff [iff]: ((a,b): Sigma A B) = (a:A & b:B(a))
    by blast

lemma Times-subset-cancel2: x:C ==> (A ** C <= B ** C) = (A <= B)
by blast

lemma Times-eq-cancel2: \( x: C \mapsto (A \leftrightarrow C = B \leftrightarrow C) = (A = B) \)
by (blast elim: equalityE)

lemma SetCompr-Sigma-eq:
Collect (split (%x y. P x & Q x y)) = (SIGMA x:Collect P. Collect (Q x))
by blast

lemma Collect-split [simp]: \{ (a,b). P a & Q b \} = Collect P \leftrightarrow Collect Q
by blast

lemma UN-Times-distrib:
(UN (a,b):(A \leftrightarrow B). E a \leftrightarrow F b) = (UNION A E) \leftrightarrow (UNION B F)
— Suggested by Pierre Chartier
by blast

lemma split-paired-Ball-Sigma [simp, no-atp]:
(ALL z: Sigma A B. P z) = (ALL x:A. ALL y: B x. P(x,y))
by blast

lemma split-paired-Bex-Sigma [simp, no-atp]:
(EX z: Sigma A B. P z) = (EX x:A. EX y: B x. P(x,y))
by blast

lemma Sigma-Un-distrib1: (SIGMA i: I Un J. C(i)) = (SIGMA i: I. C(i)) Un (SIGMA j: J. C(j))
by blast

lemma Sigma-Un-distrib2: (SIGMA i: I. A(i) Un B(i)) = (SIGMA i: I. A(i)) Un (SIGMA i: I. B(i))
by blast

lemma Sigma-Int-distrib1: (SIGMA i: I Int J. C(i)) = (SIGMA i: I. C(i)) Int (SIGMA j: J. C(j))
by blast

lemma Sigma-Int-distrib2: (SIGMA i: I. A(i) Int B(i)) = (SIGMA i: I. A(i)) Int (SIGMA i: I. B(i))
by blast

lemma Sigma-Diff-distrib1: (SIGMA i: I − J. C(i)) = (SIGMA i: I. C(i)) − (SIGMA j: J. C(j))
by blast

lemma Sigma-Diff-distrib2: (SIGMA i: I. A(i) − B(i)) = (SIGMA i: I. A(i)) − (SIGMA i: I. B(i))
by blast
lemma Sigma-Union: Sigma (Union X) B = (UN A:X. Sigma A B)
  by blast

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma Times-Un-distrib1: (A Un B) <**> C = (A <**> C) Un (B <**> C)
  by (fact Sigma-Un-distrib1)

lemma Times-Int-distrib1: (A Int B) <**> C = (A <**> C) Int (B <**> C)
  by (fact Sigma-Int-distrib1)

lemma Times-Diff-distrib1: (A - B) <**> C = (A <**> C) - (B <**> C)
  by (fact Sigma-Diff-distrib1)

lemma Times-empty[simp]: A × B = {} ↔ A = {} ∨ B = {}
  by auto

lemma times-eq-iff: A × B = C × D ↔ A = C ∧ B = D ∨ ((A = {}) ∨ B = {} ∧ (C = {}) ∨ D = {}))
  by auto

lemma fst-image-times[simp]: fst ' (A × B) = (if B = {} then {} else A)
  by force

lemma snd-image-times[simp]: snd ' (A × B) = (if A = {} then {} else B)
  by force

lemma vimage-fst:
  fst −' A = A × UNIV
  by auto

lemma vimage-snd:
  snd −' A = UNIV × A
  by auto

lemma insert-times-insert[simp]:
  insert a A × insert b B =
  insert (a,b) (A × insert b B ∪ insert a A × B)
  by blast

lemma vimage-Times: f −' (A × B) = ((fst o f) −' A) ∩ ((snd o f) −' B)
  apply auto
  apply (case-tac f x)
  apply auto
  done

lemma times-Int-times: A × B ∩ C × D = (A ∩ C) × (B ∩ D)
  by auto
lemma product-swap:

\[ \text{prod.swap : } (A \times B) = B \times A \]

by (auto simp add: set-eq-iff)

lemma swap-product:

\[ (\lambda (i, j). (j, i)) : (A \times B) = B \times A \]

by (auto simp add: set-eq-iff)

lemma image-split-eq-Sigma:

\[ (\lambda x. (f x, g x)) : A = \Sigma (f : A) (\lambda x. g : (f \setminus \{x\}) \cap A) \]

proof (safe intro!: imageI)

fix a b assume \(*\): a \in A b \in A and eq: f a = f b

show \((f b, g a) \in (\lambda x. (f x, g x)) : A \]

using \(*\ eq[\text{symmetric}] \) by auto

qed simp-all

definition product :: 

\[ 'a set \Rightarrow 'b set \Rightarrow ('a \times 'b) set \]

where [code-abbrev]: \( \text{product } A \ B = A \times B \)

hide-const (open) product

lemma member-product:

\( x : \text{Product-Type} . \text{product } A \ B \leftrightarrow x \in A \times B \)

by (simp add: product-def)

The following map-prod lemmas are due to Joachim Breitner:

lemma map-prod-inj-on:

assumes inj-on f A and inj-on g B

shows inj-on (map-prod f g) (A \times B)

proof (rule inj-onI)

fix x :: 'a \times 'c and y :: 'a \times 'c

assume x \in A \times B hence fst x \in A and snd x \in B by (auto)

assume y \in A \times B hence fst y \in A and snd y \in B by (auto)

assume map-prod f g x = map-prod f g y

hence fst (map-prod f g x) = fst (map-prod f g y) by (auto)

hence \( f (\text{fst } x) = f (\text{fst } y) \) by (cases x,cases y,auto)

with \( \text{inj-on } f A \) and \( \text{fst } x \in A \) and \( \text{fst } y \in A \)

have \( \text{fst } x = \text{fst } y \) by (auto dest:inj-onD)

moreover from \( \text{map-prod } f g x = \text{map-prod } f g y \)

have \( \text{snd } (\text{map-prod } f g x) = \text{snd } (\text{map-prod } f g y) \) by (auto)

hence \( g (\text{snd } x) = g (\text{snd } y) \) by (cases x,cases y,auto)

with \( \text{inj-on } g B \) and \( \text{snd } x \in B \) and \( \text{snd } y \in B \)

have \( \text{snd } x = \text{snd } y \) by (auto dest:inj-onD)

ultimately show \( x = y \) by (rule prod-eqI)

qed

lemma map-prod-surj:

fixes f :: 'a \Rightarrow 'b and g :: 'c \Rightarrow 'd

assumes surj f and surj g
shows surj \((\text{map-prod } f \ g)\)
unsurfolding surj-def
proof
  fix y :: 'b × 'd
  from \(\text{srule } f\) obtain a where \(\text{fst } y = f \ a\) by (auto elim:surjE)
  moreover
  from \(\text{srule } g\) obtain b where \(\text{snd } y = g \ b\) by (auto elim:surjE)
  ultimately have \((\text{fst } y, \text{snd } y) = \text{map-prod } f \ g(a,b)\) by auto
  thus \(\exists x. \ y = \text{map-prod } f \ g \ x\) by auto
qed

lemma map-prod-surj-on:
  assumes \(f' \cdot A = A'\) and \(g' \cdot B = B'\)
  shows \(\text{map-prod } f \ g' \cdot (A \times B) = A' \times B'\)
unsurfolding image-def
proof (rule set-eqI, rule iffI)
  fix x :: 'a × 'c
  assume \(x \in \{ y::'a \times 'c. \exists x::'b \times 'd\in A \times B. \ y = \text{map-prod } f \ g \ x\}\)
  then obtain y where \(y \in A \times B\) and \(x = \text{map-prod } f \ g \ y\) by blast
  from \(\text{image } f \ A = A'\) and \(\text{y } \in A \times B\) have \(f \ (\text{fst } y) \in A'\) by auto
  moreover from \(\text{image } g \ B = B'\) and \(\text{y } \in A \times B\) have \(g \ (\text{snd } y) \in B'\) by auto
  ultimately have \((f \ (\text{fst } y), g \ (\text{snd } y)) \in (A' \times B')\) by auto
  with \((x = \text{map-prod } f \ g \ y)\) show \(x \in A' \times B'\) by (cases y, auto)
next
  fix x :: 'a × 'c
  assume \(x \in A' \times B'\) hence \(\text{fst } x \in A'\) and \(\text{snd } x \in B'\) by auto
  from \(\text{image } f \ A = A'\) and \(\text{fst } x \in A'\) have \(\text{fst } x \in \text{image } f \ A\) by auto
  then obtain a where \(a \in A\) and \(\text{fst } x = f \ a\) by (rule imageE)
  moreover from \(\text{image } g \ B = B'\) and \(\text{snd } x \in B'\)
  obtain \(\text{b } \in B\) and \(\text{snd } x = g \ b\) by auto
  ultimately have \((\text{fst } x, \text{snd } x) = \text{map-prod } f \ g(a,b)\) by auto
  moreover from \(a \in A\) and \(\text{b } \in B\) have \((a, b) \in A \times B\) by auto
  ultimately have \(\exists y \in A \times B. x = \text{map-prod } f \ g \ y\) by auto
  thus \(x \in \{x. \ \exists y \in A \times B. x = \text{map-prod } f \ g \ y\}\) by auto
qed

12.4 Simproc for rewriting a set comprehension into a point-free expression

**ML-file** Tools/set-comprehension-pointfree.ML

**setup**

\[
\text{Code-Preproc.map-pre } (\text{fn ctxt } => \text{ctxt addsimprocs})
\]
\[
[\text{Raw-Simplifier.make-simproc \{name = set comprehension, lhss = [@\{cpat Collect ?P\}],}
\]
\[
\text{proc = K Set-Comprehension-Pointfree.code-simproc, identifier = []}])
\]
\]
12.5 Inductively defined sets

\textbf{simproc-setup} \ Collection-mem \ (Collect \ t) = \ (\langle \langle \ (fn - \Rightarrow fn \ ctxt => fn \ ct =>
\text{(case term-of ct of}
S as \ Const \ (@\{\text{const-name Collect}\}, \ Type \ (@\{\text{type-name fun}\}, [\cdot, T]\}) \ s =>
let \ val \ (u, \ -, \ ps) = HOLogic.strip-psplits \ t \ in
\text{(case u of}
(c as \ Const \ (@\{\text{const-name Set-member}\}, \ -)) \ s \ q \ S' =>
\text{(case try \ (HOLogic.strip-ptuple \ ps) \ q \ of}
NONE => NONE
| SOME \ ts \ =>
\text{if not \ (Term.is-open \ S') \ andalso}
\text{ts = map \ Bound \ (length \ ps \ downto} \ 0)
\text{then}
\text{let \ val \ simp =}
\text{full-simp-tac \ (put-simpset \ HOL-basic-ss \ ctxt}
\text{addsimps \ (@\{\text{thm \ split-paired-all}\}, @\{\text{thm \ split-conv}\}) \ 1}
\text{in}
SOME \ (Goal.prove \ ctxt \ [] \ [\]
\text{(Const \ (@\{\text{const-name Pure.eq}\}, \ T => T => propT)} \ s \ S
\text{S')}
\text{(K} \ (EVERY}
\text{rtac \ eq-reflection \ 1, \ rtac \ @\{\text{thm \ subset-antisym}\} \ 1,}
\text{rtac \ subsetI \ 1, \ d tac \ CollectD \ 1, \ simp,}
\text{rtac \ subsetI \ 1, \ rtac \ CollectI \ 1, \ simp)))\text{))}
end
else NONE)
| - => NONE\text{)}
end
\text{|} - => NONE\text{)}
\text{\rangle \rangle}

ML-file \ Tools/inductive-set.ML

12.6 Legacy theorem bindings and duplicates

\textbf{lemma} \ PairE:\n\textbf{obtains} \ x \ y \ \textbf{where} \ p = (x, y)\n\textbf{by} \ (\text{fact \ prod.exhaust})

\textbf{lemmas} \ Pair-eq = prod.inject
\textbf{lemmas} \ fst-conv = prod.sel(1)
\textbf{lemmas} \ snd-conv = prod.sel(2)
\textbf{lemmas} \ pair-collapse = prod-collapse
\textbf{lemmas} \ split = split-conv
\textbf{lemmas} \ Pair-fst-snd-eq = prod-eq-iff

hide-const \ (open) \ prod

end
13  Sum-Type: The Disjoint Sum of Two Types

theory Sum-Type
imports Typedef Inductive Fun
begin

13.1  Construction of the sum type and its basic abstract operations

definition Inl-Rep :: 'a ⇒ 'a ⇒ 'b ⇒ bool ⇒ bool
where
Inl-Rep a x y p ←→ x = a ∧ p

definition Inr-Rep :: 'b ⇒ 'a ⇒ 'b ⇒ bool ⇒ bool
where
Inr-Rep b x y p ←→ y = b ∧ ¬ p

definition sum = { f. (∃ a. f = Inl-Rep (a::'a)) ∨ (∃ b. f = Inr-Rep (b::'b))}

typedef ('a, 'b) sum (infixr + 10) = sum :: ('a => 'b => bool => bool) set

unfolding sum-def by auto

lemma Inl-RepI: Inl-Rep a ∈ sum
  by (auto simp add: sum-def)

lemma Inr-RepI: Inr-Rep b ∈ sum
  by (auto simp add: sum-def)

lemma inj-on-Abs-sum: A ⊆ sum =⇒ inj-on Abs-sum A
  by (rule inj-on-inverseI, rule Abs-sum-inverse) auto

lemma Inl-Rep-inject: inj-on Inl-Rep A
proof (rule inj-onI)
  show ∀a c. Inl-Rep a = Inl-Rep c =⇒ a = c
    by (auto simp add: Inl-Rep-def fun-eq-iff)
qed

lemma Inr-Rep-inject: inj-on Inr-Rep A
proof (rule inj-onI)
  show ∀b d. Inr-Rep b = Inr-Rep d =⇒ b = d
    by (auto simp add: Inr-Rep-def fun-eq-iff)
qed

lemma Inl-Rep-not-Inr-Rep: Inl-Rep a ≠ Inr-Rep b
  by (auto simp add: Inl-Rep-def Inr-Rep-def fun-eq-iff)

definition Inl :: 'a ⇒ 'a + 'b where
Inl = Abs-sum ◦ Inl-Rep

definition Inr :: 'b ⇒ 'a + 'b where
Inr = Abs-sum ◦ Inr-Rep
lemma inj-Inl [simp]: inj-on Inl A
by (auto simp add: Inl-def intro!: comp-inj-on Inl-Rep-inject inj-on-Abs-sum Inl-RepI)

lemma Inl-inject: Inl x = Inl y =⇒ x = y
using inj-Inl by (rule injD)

lemma inj-Inr [simp]: inj-on Inr A
by (auto simp add: Inr-def intro!: comp-inj-on Inr-Rep-inject inj-on-Abs-sum Inr-RepI)

lemma Inr-inject: Inr x = Inr y =⇒ x = y
using inj-Inr by (rule injD)

lemma Inl-not-Inr: Inl a \neq Inr b
proof
  from Inl-RepI [of a] Inr-RepI [of b] have {Inl-Rep a, Inr-Rep b} ⊆ sum by auto
  with inj-on-Abs-sum have inj-on Abs-sum {Inl-Rep a, Inr-Rep b}.
  with Inl-Rep-not-Inr-Rep [of a b] inj-on-contraD have Abs-sum (Inl-Rep a) \neq Abs-sum (Inr-Rep b) by auto
  then show ?thesis by (simp add: Inl-def Inr-def)
qed

lemma Inr-not-Inl: Inr b \neq Inl a
using Inl-not-Inr by (rule not-sym)

lemma sumE:
  assumes \( \forall x :: 'a. s = \text{Inl } x \Rightarrow P \)
  and \( \forall y :: 'b. s = \text{Inr } y \Rightarrow P \)
  shows P
proof (rule Abs-sum-cases [of s])
  fix f
  assume s = Abs-sum f and f ∈ sum
  with assms show P by (auto simp add: sum-def Inl-def Inr-def)
qed

free-constructors case-sum for
  isl: Inl projl
  | Inr projr
by (erule sumE, assumption) (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

Avoid name clashes by prefixing the output of rep-datatype with old.
setup "\langle Sign.mandatory-path old \rangle"

rep-datatype Inl Inr
proof
  fix P
  fix s :: 'a + 'b
  assume x: \( \forall x :: 'a. P \text{ (Inl } x) \) and y: \( \forall y :: 'b. P \text{ (Inr } y) \)
  then show P s by (auto intro: sumE [of s])
THEORY "Sum-Type"

qed (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

setup ⟨⟨ Sign.parent-path ⟩⟩

But erase the prefix for properties that are not generated by free-constructors.

setup ⟨⟨ Sign.mandatory-path sum ⟩⟩

declare

old.sum.inject[iff del]
old.sum.distinct(1)[simp del, induct-simp del]

lemmas induct = old.sum.induct
lemmas inducts = old.sum.inducts
lemmas rec = old.sum.rec
lemmas simps = sum.inject sum.distinct sum.case sum.rec

setup ⟨⟨ Sign.parent-path ⟩⟩

primrec map-sum :: (′a ⇒ ′c) ⇒ (′b ⇒ ′d) ⇒ ′a + ′b ⇒ ′c + ′d where
map-sum f1 f2 (Inl a) = Inl (f1 a)
| map-sum f1 f2 (Inr a) = Inr (f2 a)

functor map-sum: map-sum proof –

fix g h i

show map-sum f g o map-sum h i = map-sum (f o h) (g o i)
proof

fix s

show (map-sum f g o map-sum h i) s = map-sum (f o h) (g o i) s

by (cases s) simp-all

qed

next

fix s

show map-sum id id = id

proof

fix s

show map-sum id id s = id s

by (cases s) simp-all

qed

13.2 Projections

lemma case-sum-KK [simp]: case-sum (λx. a) (λx. a) = (λx. a)
THEORY “Sum-Type”

by (rule ext) (simp split: sum.split)

lemma surjective-sum: case-sum ($\lambda x::'a. f (Inl x)) (\lambda y::'b. f (Inr y)) = f
proof
  fix s :: 'a + 'b
  show (case s of Inl (x::'a) ⇒ f (Inl x) | Inr (y::'b) ⇒ f (Inr y)) = f s
    by (cases s) simp-all
qed

lemma case-sum-inject:
  assumes a: case-sum f1 f2 = case-sum g1 g2
  assumes r: f1 = g1 ⇒ f2 = g2 ⇒ P
  shows P
proof (rule r)
  show f1 = g1 proof
    fix x :: 'a
    from a have case-sum f1 f2 (Inl x) = case-sum g1 g2 (Inl x) by simp
    then show f1 x = g1 x by simp
  qed
  show f2 = g2 proof
    fix y :: 'b
    from a have case-sum f1 f2 (Inr y) = case-sum g1 g2 (Inr y) by simp
    then show f2 y = g2 y by simp
  qed
qed

primrec Suml :: ('a ⇒ 'c) ⇒ 'a + 'b ⇒ 'c where
  Suml f (Inl x) = f x

primrec Sumr :: ('b ⇒ 'c) ⇒ 'a + 'b ⇒ 'c where
  Sumr f (Inr x) = f x

lemma Suml-inject:
  assumes Suml f = Suml g shows f = g
proof
  fix x :: 'a
  let ?s = Inl x :: 'a + 'b
  from assms have Suml f ?s = Suml g ?s by simp
  then show f x = g x by simp
qed

lemma Sumr-inject:
  assumes Sumr f = Sumr g shows f = g
proof
  fix x :: 'b
  let ?s = Inr x :: 'a + 'b
  from assms have Sumr f ?s = Sumr g ?s by simp
  then show f x = g x by simp
qed
13.3 The Disjoint Sum of Sets

**definition** \texttt{Plus :: 'a set ⇒ 'b set ⇒ ('a + 'b) set (infixr <++> 65) where}
\newline \texttt{A <++> B = Inl ' A ∪ Inr ' B}

**hide-const (open) Plus** — Valuable identifier

**lemma** \texttt{InlI [intro!]: a ∈ A ⇒ Inl a ∈ A <++> B}
by \texttt{(simp add: Plus-def)}

**lemma** \texttt{InrI [intro!]: b ∈ B ⇒ Inr b ∈ A <++> B}
by \texttt{(simp add: Plus-def)}

Exhaustion rule for sums, a degenerate form of induction

**lemma** \texttt{PlusE [elim!]:}
\newline \texttt{u ∈ A <++> B ⇒ (∀ x. x ∈ A ⇒ u = Inl x ⇒ P) ⇒ (∀ y. y ∈ B ⇒ u = Inr y ⇒ P) ⇒ P}
by \texttt{(auto simp add: Plus-def)}

**lemma** \texttt{Plus-eq-empty-conv [simp]: A <++> B = {} ←→ A = {} ∧ B = {}}
by \texttt{auto}

**lemma** \texttt{UNIV-Plus-UNIV [simp]: UNIV <++> UNIV = UNIV}
proof \texttt{(rule set-eqI)}
\newline \texttt{fix u :: 'a + 'b}
\newline \texttt{show u ∈ UNIV <++> UNIV ←→ u ∈ UNIV by (cases u) auto}
qed

**lemma** \texttt{UNIV-sum:}
\newline \texttt{UNIV = Inl ' UNIV ∪ Inr ' UNIV}
proof –
\newline \texttt{fix x :: 'a + 'b}
\texttt{assume x /∈ range Inr}
\texttt{then have x ∈ range Inl}
\texttt{by (cases x) simp-all}
\texttt{then show ?thesis by auto}
qed

**hide-const (open) Suml Sumr sum**

end

14 Rings: Rings

**theory** Rings
**imports** Groups
begin

**class** \texttt{semiring = ab-semigroup-add + semigroup-mult +}
assumes distrib-right[algebra-simps, field-simps]: \((a + b) \ast c = a \ast c + b \ast c\)
assumes distrib-left[algebra-simps, field-simps]: \(a \ast (b + c) = a \ast b + a \ast c\)

For the combine-numerals simproc

lemma combine-common-factor:
\(a \ast e + (b \ast e + c) = (a + b) \ast e + c\)
by (simp add: distrib-right ac-simps)

end

class mult-zero = times + zero +
assumes mult-zero-left [simp]: \(0 \ast a = 0\)
assumes mult-zero-right [simp]: \(a \ast 0 = 0\)

class semiring-0 = semiring + comm-monoid-add + mult-zero

class semiring-0-cancel = semiring + cancel-comm-monoid-add

begin
subclass semiring-0
proof
  fix \(a::'a\)
  have \(0 \ast a + 0 \ast a = 0 \ast a + 0\) by (simp add: distrib-right [symmetric])
  thus \(0 \ast a = 0\) by (simp only: add-left-cancel)
next
  fix \(a::'a\)
  have \(a \ast 0 + a \ast 0 = a \ast 0 + 0\) by (simp add: distrib-left [symmetric])
  thus \(a \ast 0 = 0\) by (simp only: add-left-cancel)
qed

end

class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
assumes distrib: \((a + b) \ast c = a \ast c + b \ast c\)

begin
subclass semiring
proof
  fix \(a \ b \ c::'a\)
  show \((a + b) \ast c = a \ast c + b \ast c\) by (simp add: distrib)
  have \(a \ast (b + c) = (b + c) \ast a\) by (simp add: ac-simps)
  also have \...( = b \ast a + c \ast a\) by (simp only: distrib)
  also have \...( = a \ast b + a \ast c\) by (simp add: ac-simps)
  finally show \(a \ast (b + c) = a \ast b + a \ast c\) by blast
qed

end
class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin
subclass semiring-0 ..
end

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin
subclass semiring-0-cancel ..
subclass comm-semiring-0 ..
end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]: 0 ≠ 1
begin
lemma one-neq-zero [simp]: 1 ≠ 0
by (rule not-sym) (rule zero-neq-one)
definition of-bool :: bool ⇒ 'a
where
  of-bool p = (if p then 1 else 0)
lemma of-bool-eq [simp, code]:
  of-bool False = 0
  of-bool True = 1
by (simp-all add: of-bool-def)
lemma of-bool-eq-iff:
  of-bool p = of-bool q ⟷ p = q
by (simp add: of-bool-def)
lemma split-of-bool [split]:
  P (of-bool p) ⟷ (p ⟷ P 1) ∧ (¬ p ⟷ P 0)
by (cases p) simp-all
lemma split-of-bool-asm:
  P (of-bool p) ⟷ ¬ (p ∧ ¬ P 1 ∨ ¬ p ∧ ¬ P 0)
by (cases p) simp-all
end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult

Abstract divisibility
class dvd = times
begin

definition dvd :: 'a ⇒ 'a ⇒ bool (infix dvd 50) where
  b dvd a ⟷ (∃k. a = b * k)

lemma dvdI [intro?]: a = b * k ⟹ b dvd a
  unfolding dvd-def ..

lemma dvdE [elim?]: b dvd a ⟹ (∀k. a = b * k ⟹ P) ⟹ P
  unfolding dvd-def by blast

end

class comm-semiring-1 = zero-neq-one + comm-semiring-0 + comm-monoid-mult + dvd
begin

subclass semiring-1 ..

lemma dvd-refl [simp]: a dvd a
  proof
    show a = a * 1 by simp
  qed

lemma dvd-trans:
  assumes a dvd b and b dvd c
  shows a dvd c
  proof –
    from assms obtain v where b = a * v by (auto elim!: dvdE)
    moreover from assms obtain w where c = b * w by (auto elim!: dvdE)
    ultimately have c = a * (v * w) by (simp add: mult.assoc)
    then show ?thesis ..
  qed

lemma dvd-0-left-iff [simp]: 0 dvd a ⟷ a = 0
  by (auto intro: dvd-refl elim!: dvdE)

lemma dvd-0-right [iff]: a dvd 0
  proof
    show 0 = a * 0 by simp
  qed

lemma one-dvd [simp]: 1 dvd a
  by (auto intro!: dvdI)

lemma dvd-mult [simp]: a dvd c ⟹ a dvd (b * c)
  by (auto intro!: mult.left-commute dvdI elim!: dvdE)
lemma dvd-mult2 [simp]: \(a \text{ dvd } b \implies a \text{ dvd } (b \ast c)\)
apply (subst mult.commute)
apply (erule dvd-mult)
done

lemma dvd-triv-right [simp]: \(a \text{ dvd } b \ast a\)
by (rule dvd-mult) (rule dvd-refl)

lemma dvd-triv-left [simp]: \(a \text{ dvd } a \ast b\)
by (rule dvd-mult2) (rule dvd-refl)

lemma mult-dvd-mono:
  assumes \(a \text{ dvd } b\) and \(c \text{ dvd } d\)
  shows \(a \ast c \text{ dvd } b \ast d\)
proof
from \(\langle a \text{ dvd } b\rangle\) obtain \(b'\) where \(b = a \ast b'\).
moreover from \(\langle c \text{ dvd } d\rangle\) obtain \(d'\) where \(d = c \ast d'\).
ultimately have \(b + c = (a \ast c) \ast (b' + d')\) by (simp add: ac-simps)
then show \(?thesis\) ..
qed

lemma dvd-mult-left: \(a \ast b \text{ dvd } c \implies a \text{ dvd } c\)
by (simp add: dvd-def mult.assoc, blast)

lemma dvd-mult-right: \(a \ast b \text{ dvd } c \implies b \text{ dvd } c\)
  unfolding mult.commute [of a] by (rule dvd-mult-left)

lemma dvd-0-left: \(0 \text{ dvd } a \implies a = 0\)
by simp

lemma dvd-add [simp]:
  assumes \(a \text{ dvd } b\) and \(a \text{ dvd } c\)
  shows \(a \text{ dvd } (b + c)\)
proof
from \(\langle a \text{ dvd } b\rangle\) obtain \(b'\) where \(b = a \ast b'\).
moreover from \(\langle a \text{ dvd } c\rangle\) obtain \(c'\) where \(c = a \ast c'\).
ultimately have \(b + c = a \ast (b' + c')\) by (simp add: distrib-left)
then show \(?thesis\) ..
qed

end

class no-zero-divisors = zero + times +
  assumes no-zero-divisors: \(a \neq 0 \implies b \neq 0 \implies a \ast b \neq 0\)
begin

lemma divisors-zero:
  assumes \(a \ast b = 0\)
shows $a = 0 \lor b = 0$

proof (rule classical)
  assume $\neg (a = 0 \lor b = 0)$
  then have $a \neq 0$ and $b \neq 0$ by auto
  with no-zero-divisors have $a \cdot b \neq 0$ by blast
  with assms show ?thesis by simp
qed

end

class semiring-1-cancel = semiring + cancel-comm-monoid-add
  + zero-neq-one + monoid-mult
begin

subclass semiring-0-cancel ..

subclass semiring-1 ..

end

class comm-semiring-1-cancel = comm-semiring + cancel-comm-monoid-add
  + zero-neq-one + comm-monoid-mult
begin

subclass semiring-1-cancel ..
subclass comm-semiring-0-cancel ..
subclass comm-semiring-1 ..

end

class ring = semiring + ab-group-add
begin

subclass semiring-0-cancel ..

Distribution rules

lemma minus-mult-left: $- (a \cdot b) = - a \cdot b$
by (rule minus-unique) (simp add: distrib-right [symmetric])

lemma minus-mult-right: $- (a \cdot b) = a \cdot - b$
by (rule minus-unique) (simp add: distrib-left [symmetric])

Extract signs from products

lemmas mult-minus-left [simp] = minus-mult-left [symmetric]
lemmas mult-minus-right [simp] = minus-mult-right [symmetric]

lemma minus-mult-minus [simp]: $- a - b = a \cdot b$
by simp
lemma minus-mult-commute: $-a \cdot b = a \cdot -b$
  by simp

lemma right-diff-distrib [algebra-simps, field-simps]:
  $a \cdot (b - c) = a \cdot b - a \cdot c$
  using distrib-left [of $a\ b -\ c$] by simp

lemma left-diff-distrib [algebra-simps, field-simps]:
  $(a - b) \cdot c = a \cdot c - b \cdot c$
  using distrib-right [of $a -\ b\ c$] by simp

lemmas ring-distribs =
  distrib-left distrib-right left-diff-distrib right-diff-distrib

lemma eq-add-iff1:
  $a \cdot e + c = b \cdot e + d \iff (a - b) \cdot e + c = d$
  by (simp add: algebra-simps)

lemma eq-add-iff2:
  $a \cdot e + c = b \cdot e + d \iff c = (b - a) \cdot e + d$
  by (simp add: algebra-simps)

end

lemmas ring-distribs =
  distrib-left distrib-right left-diff-distrib right-diff-distrib

class comm-ring = comm-semiring + ab-group-add
begin

subclass ring ..

subclass comm-semiring-0-cancel ..

lemma square-diff-square-factored:
  $x \cdot x - y \cdot y = (x + y) \cdot (x - y)$
  by (simp add: algebra-simps)

end

class ring-1 = ring + zero-neq-one + monoid-mult
begin

subclass semiring-1-cancel ..

lemma square-diff-one-factored:
  $x \cdot x - 1 = (x + 1) \cdot (x - 1)$
  by (simp add: algebra-simps)

end
class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mult

begin

subclass ring-1..
subclass comm-semiring-1-cancel..

lemma dvd-minus-iff [simp]: x dvd − y ←→ x dvd y
proof
  assume x dvd − y
  then have x dvd − 1 * − y by (rule dvd-mult)
  then show x dvd y by simp
next
  assume x dvd y
  then have x dvd − 1 * y by (rule dvd-mult)
  then show x dvd − y by simp
qed

lemma minus-dvd-iff [simp]: − x dvd y ←→ x dvd y
proof
  assume − x dvd y
  then obtain k where y = − x * k..
  then have y = x * − k by simp
  then show x dvd y..
next
  assume x dvd y
  then obtain k where y = x * k..
  then have y = − x * − k by simp
  then show − x dvd y..
qed

lemma dvd-diff [simp]:
  x dvd y ⇒ x dvd z ⇒ x dvd (y − z)
  using dvd-add [of x y − z] by simp
end

class ring-no-zero-divisors = ring + no-zero-divisors
begin

lemma mult-eq-0-iff [simp]:
  shows a * b = 0 ←→ (a = 0 ∨ b = 0)
proof (cases a = 0 ∨ b = 0)
  case False then have a ≠ 0 and b ≠ 0 by auto
    then show ?thesis using no-zero-divisors by simp
next
  case True then show ?thesis by auto
qed
Cancellation of equalities with a common factor

\textbf{lemma} \texttt{mult-cancel-right} \ [simp]:
\[ a \cdot c = b \cdot c \iff c = 0 \lor a = b \]
\textbf{proof} –
\begin{align*}
\text{have } & (a \cdot c = b \cdot c) = ((a - b) \cdot c = 0) \\
\text{by } & (\text{simp add: algebra-simps}) \\
\text{thus } & \text{thesis by (simp add: disj-commute)}
\end{align*}
\texttt{qed}

\textbf{lemma} \texttt{mult-cancel-left} \ [simp]:
\[ c \cdot a = c \cdot b \iff c = 0 \lor a = b \]
\textbf{proof} –
\begin{align*}
\text{have } & (c \cdot a = c \cdot b) = (c \cdot (a - b) = 0) \\
\text{by } & (\text{simp add: algebra-simps}) \\
\text{thus } & \text{thesis by simp}
\end{align*}
\texttt{qed}

\textbf{lemma} \texttt{mult-left-cancel}: $c \neq 0 \impliedby (c \cdot a = c \cdot b) = (a = b)$
\texttt{by simp}

\textbf{lemma} \texttt{mult-right-cancel}: $c \neq 0 \impliedby (a \cdot c = b \cdot c) = (a = b)$
\texttt{by simp}

end

class \texttt{ring-1-no-zero-divisors} = \texttt{ring-1} + \texttt{ring-no-zero-divisors}
\begin{align*}
\textbf{lemma} \texttt{square-eq-1-iff}: \\
x \cdot x = 1 \iff x = 1 \lor x = -1
\end{align*}
\textbf{proof} –
\begin{align*}
\text{have } & (x - 1) \cdot (x + 1) = x \cdot x - 1 \\
\text{by } & (\text{simp add: algebra-simps}) \\
\text{hence } & x \cdot x = 1 \iff (x - 1) \cdot (x + 1) = 0 \\
\text{by } & \text{simp} \\
\text{thus } & \text{thesis} \\
\text{by } & (\text{simp add: eq-neg-iff-add-eq-0})
\end{align*}
\texttt{qed}

\textbf{lemma} \texttt{mult-cancel-right1} \ [simp]:
\[ c = b \cdot c \iff c = 0 \lor b = 1 \]
\texttt{by (insert \texttt{mult-cancel-right} \ [of \ 1 \ c \ b], force)}

\textbf{lemma} \texttt{mult-cancel-right2} \ [simp]:
\[ a \cdot c = c \iff c = 0 \lor a = 1 \]
\texttt{by (insert \texttt{mult-cancel-right} \ [of \ a \ c \ 1], simp)}

\textbf{lemma} \texttt{mult-cancel-left1} \ [simp]:
\[ c = c \cdot b \iff c = 0 \lor b = 1 \]
by (insert mult-cancel-left [of c 1 b], force)

lemma mult-cancel-left2 [simp]:
c * a = c ←→ c = 0 ∨ a = 1
by (insert mult-cancel-left [of c a 1], simp)
end

class idom = comm-ring-1 + no-zero-divisors
begin

subclass ring-1-no-zero-divisors ..

lemma square-eq-iff: a * a = b * b ←→ (a = b ∨ a = −b)
proof
assume a * a = b * b
then have (a − b) * (a + b) = 0
  by (simp add: algebra-simps)
then show a = b ∨ a = −b
  by (simp add: eq-neg-iff-add-eq-0)
next
assume a = b ∨ a = −b
then show a * a = b * b by auto
qed

lemma dvd-mult-cancel-right [simp]:
a * c dvd b * c ←→ c = 0 ∨ a dvd b
proof
have a * c dvd b * c ←→ (∃k. b * c = (a * k) * c)
  unfolding dvd-def by (simp add: ac-simps)
also have (∃k. b * c = (a * k) * c) ←→ c = 0 ∨ a dvd b
  unfolding dvd-def by simp
finally show ?thesis .
qed

lemma dvd-mult-cancel-left [simp]:
c * a dvd c * b ←→ c = 0 ∨ a dvd b
proof
have c * a dvd c * b ←→ (∃k. b * c = (a * k) * c)
  unfolding dvd-def by (simp add: ac-simps)
also have (∃k. b * c = (a * k) * c) ←→ c = 0 ∨ a dvd b
  unfolding dvd-def by simp
finally show ?thesis .
qed
end

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society
Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- Algebra I by van der Waerden, Springer.

```plaintext
class ordered-semiring = semiring + comm-monoid-add + ordered-ab-semigroup-add
+ assumes mult-left-mono: $a \leq b \implies 0 \leq c \implies c \cdot a \leq c \cdot b$
+ assumes mult-right-mono: $a \leq b \implies 0 \leq c \implies a \cdot c \leq b \cdot c$
begin

lemma mult-mono:
  $a \leq b \implies c \leq d \implies 0 \leq b \implies 0 \leq c \implies a \cdot c \leq b \cdot d$
apply (erule mult-right-mono [THEN order-trans], assumption)
apply (erule mult-left-mono, assumption)
done

lemma mult-mono':
  $a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a \cdot c \leq b \cdot d$
apply (rule mult-mono)
apply (fast intro: order-trans)+
done

class ordered-cancel-semiring = ordered-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

lemma mult-nonneg-nonneg[simp]: $0 \leq a \implies 0 \leq b \implies 0 \leq a \cdot b$
using mult-left-mono [of 0 b a] by simp

lemma mult-nonneg-nonpos: $0 \leq a \implies b \leq 0 \implies a \cdot b \leq 0$
using mult-left-mono [of b 0 a] by simp

lemma mult-nonpos-nonneg: $a \leq 0 \implies 0 \leq b \implies a \cdot b \leq 0$
using mult-right-mono [of a 0 b] by simp

Legacy - use mult-nonpos-nonneg

lemma mult-nonneg-nonpos2: $0 \leq a \implies b \leq 0 \implies b \cdot a \leq 0$
by (drule mult-right-mono [of b 0], auto)

lemma split-mult-neg-le: ($0 \leq a \land b \leq 0$) | ($a \leq 0 \land 0 \leq b$) \implies a \cdot b \leq 0
```

1979

- Partially Ordered Algebraic Systems, Pergamon Press 1963
by (auto simp add: mult-nonneg-nonneg mult-nonneg-nonneg2)

end

class linordered-semiring = ordered-semiring + linordered-cancel-ab-semigroup-add
begin
subclass ordered-cancel-semiring ..
subclass ordered-comm-monoid-add ..

lemma mult-left-less-imp-less:
  \( c \cdot a < c \cdot b \implies 0 \leq c \implies a < b \)
by (force simp add: mult-left-mono not-le [symmetric])

lemma mult-right-less-imp-less:
  \( a \cdot c < b \cdot c \implies 0 \leq c \implies a < b \)
by (force simp add: mult-right-mono not-le [symmetric])

end

class linordered-semiring-1 = linordered-semiring + semiring-1
begin

lemma convex-bound-le:
  assumes \( x \leq a \quad y \leq a \quad 0 \leq u \quad 0 \leq v \quad u + v = 1 \)
  shows \( u \cdot x + v \cdot y \leq a \)
proof
  from assms have \( u \cdot x + v \cdot y \leq u \cdot a + v \cdot a \)
  by (simp add: add-mono mult-left-mono)
  thus \(?thesis using assms unfolding distrib-right[ symmetric ] \) by simp
qed

end

class linordered-semiring-strict = semiring + comm-monoid-add + linordered-cancel-ab-semigroup-add +
  assumes mult-strict-left-mono: \( a < b \implies 0 < c \implies c \cdot a < c \cdot b \)
  assumes mult-strict-right-mono: \( a < b \implies 0 < c \implies a \cdot c < b \cdot c \)
begin
subclass semiring-0-cancel ..
subclass linordered-semiring
proof
  fix \( a \quad b \quad c :: 'a \)
  assume \( A: a \leq b \quad 0 \leq c \)
  from \( A \) show \( c \cdot a \leq c \cdot b \)
    unfolding le-less

end
using mult-strict-left-mono by (cases c = 0) auto
from A show a * c ≤ b * c
  unfolding le-less
  using mult-strict-right-mono by (cases c = 0) auto
qed

lemma mult-left-le-imp-le:
  c * a ≤ c * b ⇒ 0 < c ⇒ a ≤ b
by (force simp add: mult-strict-left-mono -not-less [symmetric])

lemma mult-right-le-imp-le:
  a * c ≤ b * c ⇒ 0 < c ⇒ a ≤ b
by (force simp add: mult-strict-right-mono not-less [symmetric])

lemma mult-pos-pos [simp]: 0 < a ⇒ 0 < b ⇒ 0 < a * b
using mult-strict-left-mono [of 0 b a] by simp

lemma mult-pos-neg: 0 < a ⇒ b < 0 ⇒ a * b < 0
using mult-strict-left-mono [of b 0 a] by simp

lemma mult-neg-pos: a < 0 ⇒ 0 < b ⇒ a * b < 0
using mult-strict-right-mono [of a 0 b] by simp

Legacy - use mult-neg-pos

lemma mult-pos-neg2: 0 < a ⇒ b < 0 ⇒ b * a < 0
by (drule mult-strict-right-mono [of b 0], auto)

lemma zero-less-mult-pos:
  0 < a * b ⇒ 0 < a ⇒ 0 < b
apply (cases b ≤ 0)
apply (auto simp add: le-less not-less)
apply (drule-tac mult-pos-neg [of a b])
apply (auto dest: less-not-sym)
done

lemma zero-less-mult-pos2:
  0 < b * a ⇒ 0 < a ⇒ 0 < b
apply (cases b ≤ 0)
apply (auto simp add: le-less not-less)
apply (drule-tac mult-pos-neg2 [of a b])
apply (auto dest: less-not-sym)
done

Strict monotonicity in both arguments

lemma mult-strict-mono:
  assumes a < b and c < d and 0 < b and 0 ≤ c
  shows a * c < b * d
using assms apply (cases c = 0)
apply (simp)
apply (erule mult-strict-right-mono \[\text{THEN less-trans}\])
apply (force simp add: le-less)
apply (erule mult-strict-left-mono, assumption)
done

This weaker variant has more natural premises

lemma mult-strict-mono':
assumes \(a < b\) and \(c < d\) and \(0 \leq a\) and \(0 \leq c\)
shows \(a * c < b * d\)
by (rule mult-strict-mono) (insert assms, auto)

lemma mult-less-le-imp-less:
assumes \(a < b\) and \(c \leq d\) and \(0 \leq a\) and \(0 < c\)
shows \(a * c < b * d\)
using assms apply (subgoal-tac a * c < b * c)
apply (erule less-le-trans)
apply (erule mult-strict-left-mono)
apply simp
apply (erule mult-strict-right-mono)
apply assumption
done

lemma mult-le-less-imp-less:
assumes \(a \leq b\) and \(c < d\) and \(0 < a\) and \(0 \leq c\)
shows \(a * c < b * d\)
using assms apply (subgoal-tac a * c < b * c)
apply (erule le-less-trans)
apply (erule mult-strict-left-mono)
apply simp
apply (erule mult-strict-right-mono)
apply simp
done

lemma mult-less-imp-less-left:
assumes less: \(c * a < c * b\) and nonneg: \(\theta \leq c\)
shows \(a < b\)
proof (rule ccontr)
  assume \(\neg a < b\)
  hence \(b \leq a\) by (simp add: linorder-not-less)
  hence \(c * b \leq c * a\) using nonneg by (rule mult-left-mono)
  with this and less show False by (simp add: not-less \[symmetric\])
qed

lemma mult-less-imp-less-right:
assumes less: \(a * c < b * c\) and nonneg: \(\theta \leq c\)
shows \(a < b\)
proof (rule ccontr)
  assume \(\neg a < b\)
  hence \(b \leq a\) by (simp add: linorder-not-less)
hence $b \cdot c \leq a \cdot c$ using nonneg by (rule mult-right-mono)
with this and less show False by (simp add: not-less [symmetric])
qed
end

class linordered-semiring-1-strict = linordered-semiring-strict + semiring-1
begin
subclass linordered-semiring-1 ..

lemma convex-bound-lt:
  assumes $x < a \ y < a \ 0 \leq u \ u + v = 1$
  shows $u \cdot x + v \cdot y < a$
proof –
  from assms have $u \cdot x + v \cdot y < u \cdot a + v \cdot a$
    by (cases u = 0)
      (auto intro: add-less-le-mono mult-strict-left-mono mult-left-mono)
  thus ?thesis using assms unfolding distrib-right[symmetric] by simp
qed

end
class ordered-comm-semiring = comm-semiring-0 + ordered-ab-semigroup-add +
  assumes comm-mult-left-mono: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c \cdot a \leq c \cdot b$
begin
subclass ordered-semiring
proof
  fix $a \ b \ c :: 'a$
  assume $a \leq b \ 0 \leq c$
  thus $c \cdot a \leq c \cdot b$ by (rule comm-mult-left-mono)
  thus $a \cdot c \leq b \cdot c$ by (simp only: mult.commute)
qed
end
class ordered-cancel-comm-semiring = ordered-comm-semiring + cancel-comm-monoid-add
begin
subclass comm-semiring-0-cancel ..
subclass ordered-comm-semiring ..
subclass ordered-cancel-semiring ..
end

class linordered-comm-semiring-strict = comm-semiring-0 + linordered-cancel-ab-semigroup-add +
  assumes comm-mult-strict-left-mono: $a < b \Longrightarrow 0 < c \Longrightarrow c \cdot a < c \cdot b$
begin

subclass linordered-semiring-strict
proof
  fix a b c :: 'a
  assume a < b 0 < c
  thus c * a < c * b by (rule comm-mult-strict-left-mono)
  thus a * c < b * c by (simp only: mult.commute)
qed

subclass ordered-cancel-comm-semiring
proof
  fix a b c :: 'a
  assume a ≤ b 0 ≤ c
  thus c * a ≤ c * b
  unfolding le-less using mult-strict-left-mono by (cases c = 0) auto
qed

end

class ordered-ring = ring + ordered-cancel-semiring
begin

subclass ordered-ab-group-add ..

lemma less-add-iff1:
  a * e + c < b * e + d ⟷ (a − b) * e + c < d
by (simp add: algebra-simps)

lemma less-add-iff2:
  a * e + c < b * e + d ⟷ c < (b − a) * e + d
by (simp add: algebra-simps)

lemma le-add-iff1:
  a * e + c ≤ b * e + d ⟷ (a − b) * e + c ≤ d
by (simp add: algebra-simps)

lemma le-add-iff2:
  a * e + c ≤ b * e + d ⟷ c ≤ (b − a) * e + d
by (simp add: algebra-simps)

lemma mult-left-mono-neg:
  b ≤ a ⟹ c ≤ 0 ⟹ c * a ≤ c * b
apply (erule mult-left-mono [of - - c])
apply simp-all
done

lemma mult-right-mono-neg:
\[ b \leq a \implies c \leq 0 \implies a \cdot c \leq b \cdot c \]

apply (drule mult-right-mono [of \( - - c \)])
apply simp-all
done

lemma mult-nonpos-nonpos: \( a \leq 0 \implies b \leq 0 \implies 0 \leq a \cdot b \)
using mult-right-mono-neg [of \( a 0 b \)] by simp

lemma split-mult-pos-le:
\[ (0 \leq a \land 0 \leq b) \lor (a \leq 0 \land b \leq 0) \implies 0 \leq a \cdot b \]
by (auto simp add: mult-nonpos-nonpos)

end

class linordered-ring = ring + linordered-semiring + linordered-ab-group-add + abs-if
begin
subclass ordered-ring ..
subclass ordered-ab-group-add-abs
proof
  fix \( a b \)
  show \( |a + b| \leq |a| + |b| \)
    by (auto simp add: abs-if not-le not-less algebra-simps simp del: add.commute
dest: add-neg-neg add-nonneg-nonneg)
qed (auto simp add: abs-if)

lemma zero-le-square [simp]: \( 0 \leq a \cdot a \)
using linear [of \( 0 a \)]
by (auto simp add: mult-nonpos-nonpos)

lemma not-square-less-zero [simp]: \( \neg (a \cdot a < 0) \)
by (simp add: not-less)
end

class linordered-ring-strict = ring + linordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin
subclass linordered-ring ..
lemma mult-strict-left-mono-neg: \( b < a \implies c < 0 \implies c \cdot a < c \cdot b \)
using mult-strict-left-mono [of \( b a - c \)] by simp

lemma mult-strict-right-mono-neg: \( b < a \implies c < 0 \implies a \cdot c < b \cdot c \)
using mult-strict-right-mono [of \( b a - c \)] by simp
lemma mult-neg-neg: \( a < 0 \implies b < 0 \implies 0 < a \times b \)

using mult-strict-right-mono-neg [of \( a \) \( b \)] by simp

subclass ring-no-zero-divisors

proof
  fix \( a \) \( b \)
  assume \( a \neq 0 \) then have \( A: a < 0 \lor 0 < a \) by (simp add: neq_iff)
  assume \( b \neq 0 \) then have \( B: b < 0 \lor 0 < b \) by (simp add: neq_iff)
  have \( a \times b < 0 \lor 0 < a \times b \)
    proof (cases \( a < 0 \))
      case True note \( A' = \) this
      show ?thesis proof (cases \( b < 0 \))
        case True with \( A' \) show ?thesis by (auto dest: mult-neg-neg)
      next
        case False with \( B \) have \( 0 < b \) by auto
        with \( A' \) show ?thesis by (auto dest: mult-strict-right-mono)
      qed
    next
    case False with \( A \) have \( 0 < a \) by auto
    show ?thesis proof (cases \( b < 0 \))
      case True with \( A' \) have \( 0 < b \) by auto
      with \( A' \) show ?thesis by (auto dest: mult-strict-right-mono-neg)
    qed
    qed
    then show \( a \times b \neq 0 \) by (simp add: neq_iff)
  qed

lemma zero-less-mult-iff: \( 0 < a \times b \iff 0 < a \land 0 < b \land a < 0 \land b < 0 \)
  by (cases \( a \) \( b \) \( 0 \) rule: linorder-cases[case-product linorder-cases])
     (auto simp add: mult-neg-neg not_less le_less dest: zero-less-mult-pos zero-less-mult-pos2)

lemma zero-le-mult-iff: \( 0 \leq a \times b \iff 0 \leq a \land 0 \leq b \land a \leq 0 \land b \leq 0 \)
  by (auto simp add: eq-commute [of \( 0 \)] le_less not_less zero-less-mult-iff)

lemma mult-less-0-iff:
  \( a \times b < 0 \iff 0 \leq a \land 0 \leq b \land a < 0 \land 0 < b \)
  apply (insert zero-less-mult-iff [of \(-a\) \( b\)])
  apply force
  done

lemma mult-le-0-iff:
  \( a \times b \leq 0 \iff 0 \leq a \land 0 \leq b \land a \leq 0 \land b \leq 0 \)
  apply (insert zero-le-mult-iff [of \(-a\) \( b\)])
  apply force
  done
Cancellation laws for $c \cdot a < c \cdot b$ and $a \cdot c < b \cdot c$, also with the relations $\leq$ and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

**lemma** mult-less-cancel-right-disj:

$a \cdot c < b \cdot c \quad \iff \quad 0 < c \land a < b \lor c < 0 \land b < a$

**apply** (cases $c = 0$)

**apply** (auto simp add: neq-iff mult-strict-right-mono

mult-strict-right-mono-neg)

**apply** (auto simp add: not-less

not-le [symmetric, of a*c]

not-le [symmetric, of a])

**apply** (erule-tac [!] notE)

**apply** (auto simp add: less-imp-le mult-left-mono

mult-left-mono-neg)

**done**

**lemma** mult-less-cancel-left-disj:

$c \cdot a < c \cdot b \quad \iff \quad 0 < c \land a < b \lor c < 0 \land b < a$

**apply** (cases $c = 0$)

**apply** (auto simp add: neq-iff mult-strict-left-mono

mult-strict-left-mono-neg)

**apply** (auto simp add: not-less

not-le [symmetric, of c*a]

not-le [symmetric, of a])

**apply** (erule-tac [!] notE)

**apply** (auto simp add: less-imp-le mult-left-mono

mult-left-mono-neg)

**done**

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

**lemma** mult-less-cancel-right:

$a \cdot c < b \cdot c \quad \iff \quad (0 \leq c \rightarrow a < b) \land (c \leq 0 \rightarrow b < a)$

**using** mult-less-cancel-right-disj [of a c b] **by** auto

**lemma** mult-less-cancel-left:

$c \cdot a < c \cdot b \quad \iff \quad (0 \leq c \rightarrow a < b) \land (c \leq 0 \rightarrow b < a)$

**using** mult-less-cancel-left-disj [of c a b] **by** auto

**lemma** mult-le-cancel-right:

$a \cdot c \leq b \cdot c \quad \iff \quad (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$

**by** (simp add: not-less [symmetric] mult-less-cancel-right-disj)

**lemma** mult-le-cancel-left:

$c \cdot a \leq c \cdot b \quad \iff \quad (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$
by (simp add: not-less [symmetric] mult-less-cancel-left-disj)

lemma mult-le-cancel-left-pos:
\[ 0 < c \implies c \cdot a \leq c \cdot b \iff a \leq b \]
by (auto simp: mult-le-cancel-left)

lemma mult-le-cancel-left-neg:
\[ c < 0 \implies c \cdot a \leq c \cdot b \iff b \leq a \]
by (auto simp: mult-le-cancel-left)

lemma mult-less-cancel-left-pos:
\[ 0 < c \implies c \cdot a < c \cdot b \iff a < b \]
by (auto simp: mult-less-cancel-left)

lemma mult-less-cancel-left-neg:
\[ c < 0 \implies c \cdot a < c \cdot b \iff b < a \]
by (auto simp: mult-less-cancel-left)

end

lemmas mult-sign-intros =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonpos-nonneg mult-nonpos-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

class ordered-comm-ring = comm-ring + ordered-comm-semiring
begin

subclass ordered-ring ..
subclass ordered-cancel-comm-semiring ..

end

class linordered-semidom = comm-semiring-1-cancel + linordered-comm-semiring-strict +

  assumes zero-less-one [simp]: \( 0 < 1 \)
begin

lemma pos-add-strict:
  shows \( 0 < a \implies b < c \implies b < a + c \)
  using add-strict-mono [of 0 a b c] by simp

lemma zero-le-one [simp]: \( 0 \leq 1 \)
by (rule zero-less-one [THEN less-imp-le])

lemma not-one-le-zero [simp]: \( \neg 1 \leq 0 \)
by (simp add: not-le)
lemma not-one-less-zero [simp]: \( \neg 1 < 0 \)
by (simp add: not-less)

lemma less-1-mult:
assumes \( 1 < m \) and \( 1 < n \)
shows \( 1 < m \cdot n \)
using assms mult-strict-mono [of \( m \) \( 1 \) \( n \)]
by (simp add: less-trans [OF zero-less-one])

end

class linordered-idom = comm-ring-1 +
linordered-comm-semiring-strict + ordered-ab-group-add +
abs-if + sgn-if

begin

subclass linordered-semiring-1-strict ..
subclass linordered-ring-strict..
subclass ordered-comm-ring ..
subclass idom ..
subclass linordered-semidom
proof
have \( 0 \leq 1 \cdot 1 \) by (rule zero-le-square)
thus \( 0 < 1 \) by (simp add: le-less)
qued

lemma linorder-neqE-linordered-idom:
assumes \( x \neq y \) obtains \( x < y \mid y < x \)
using assms by (rule neqE)

These cancellation simprules also produce two cases when the comparison
is a goal.

lemma mult-le-cancel-right1:
\( c \leq b \cdot c \iff (0 < c \rightarrow 1 \leq b) \land (c < 0 \rightarrow b \leq 1) \)
by (insert mult-le-cancel-right [of \( c \) \( b \)], simp)

lemma mult-le-cancel-right2:
\( a \cdot c \leq c \iff (0 < c \rightarrow a \leq 1) \land (c < 0 \rightarrow 1 \leq a) \)
by (insert mult-le-cancel-right [of \( a \) \( c \) \( 1 \)], simp)

lemma mult-le-cancel-left1:
\( c \leq c \cdot b \iff (0 < c \rightarrow 1 \leq b) \land (c < 0 \rightarrow b \leq 1) \)
by (insert mult-le-cancel-left [of \( c \) \( b \)], simp)

lemma mult-le-cancel-left2:
\( c \cdot a \leq c \iff (0 < c \rightarrow a \leq 1) \land (c < 0 \rightarrow 1 \leq a) \)
by (insert mult-le-cancel-left \([c a 1]\), simp)

lemma mult-less-cancel-right1:
\[ c < b \cdot c \iff (0 \leq c \rightarrow 1 < b) \land (c \leq 0 \rightarrow b < 1) \]
by (insert mult-less-cancel-right \([1 c b]\), simp)

lemma mult-less-cancel-right2:
\[ a \cdot c < c \iff (0 \leq c \rightarrow a < 1) \land (c \leq 0 \rightarrow 1 < a) \]
by (insert mult-less-cancel-right \([a c 1]\), simp)

lemma mult-less-cancel-left1:
\[ c < c \cdot b \iff (0 \leq c \rightarrow 1 < b) \land (c \leq 0 \rightarrow b < 1) \]
by (insert mult-less-cancel-left \([c 1 b]\), simp)

lemma mult-less-cancel-left2:
\[ c \cdot a < c \iff (0 \leq c \rightarrow a < 1) \land (c \leq 0 \rightarrow 1 < a) \]
by (insert mult-less-cancel-left \([c a 1]\), simp)

lemma sgn-sgn [simp]:
\[ sgn (sgn a) = sgn a \]
unfolding sgn-if by simp

lemma sgn-0-0:
\[ sgn a = 0 \iff a = 0 \]
unfolding sgn-if by simp

lemma sgn-1-pos:
\[ sgn a = 1 \iff a > 0 \]
unfolding sgn-if by simp

lemma sgn-1-neg:
\[ sgn a = -1 \iff a < 0 \]
unfolding sgn-if by auto

lemma sgn-pos [simp]:
\[ 0 < a \implies sgn a = 1 \]
unfolding sgn-1-pos .

lemma sgn-neg [simp]:
\[ a < 0 \implies sgn a = -1 \]
unfolding sgn-1-neg .

lemma sgn-times:
\[ sgn (a \cdot b) = sgn a \cdot sgn b \]
by (auto simp add: sgn-if zero-less-mult-iff)

lemma abs-sgn: \(|k| = k \cdot sgn k\)
unfolding sgn-if abs-if by auto
lemma sgn-greater [simp]:
\[ 0 < \text{sgn} \ a \iff 0 < a \]
unfolding sgn-if by auto

lemma sgn-less [simp]:
\[ \text{sgn} \ a < 0 \iff a < 0 \]
unfolding sgn-if by auto

lemma abs-dvd-iff [simp]:
\[ |m| \text{ dvd } k \iff m \text{ dvd } k \]
by (simp add: abs-if)

lemma dvd-abs-iff [simp]:
\[ m \text{ dvd } |k| \iff m \text{ dvd } k \]
by (simp add: abs-if)

lemma dvd-if-abs-eq:
\[ |l| = |k| \implies l \text{ dvd } k \]
by (subst abs-dvd-iff [symmetric]) simp

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of \((- ?a = ?a) = (?a = (0::'a)), (- ?a < ?a) = ((0::'a) < ?a), (- ?a \leq ?a) = ((0::'a) \leq ?a).\)

lemma equation-minus-iff-1 [simp, no-atp]:
\[ 1 = -a \iff a = -1 \]
by (fact equation-minus-iff)

lemma minus-equation-iff-1 [simp, no-atp]:
\[ -a = 1 \iff a = -1 \]
by (subst minus-equation-iff, auto)

lemma le-minus-iff-1 [simp, no-atp]:
\[ 1 \leq -b \iff b \leq -1 \]
by (fact le-minus-iff)

lemma minus-le-iff-1 [simp, no-atp]:
\[ -a \leq 1 \iff -1 \leq a \]
by (fact minus-le-iff)

lemma less-minus-iff-1 [simp, no-atp]:
\[ 1 < -b \iff b < -1 \]
by (fact less-minus-iff)

lemma minus-less-iff-1 [simp, no-atp]:
\[ -a < 1 \iff -1 < a \]
by (fact minus-less-iff)

end

Simprules for comparisons where common factors can be cancelled.

lemmas mult-compare-simps =
Reasoning about inequalities with division

context linordered-semidom
begin

lemma less-add-one: $a < a + 1$
proof
  have $a + 0 < a + 1$
  by (blast intro: zero-less-one add-strict-left-mono)
  thus \?thesis by simp
qed

lemma zero-less-two: $0 < 1 + 1$
by (blast intro: less-trans zero-less-one less-add-one)

end

context linordered-idom
begin

lemma mult-right-le-one-le:
$0 \leq x \implies 0 \leq y \implies y \leq 1 \implies x \cdot y \leq x$
by (auto simp add: mult-le-cancel-left2)

lemma mult-left-le-one-le:
$0 \leq x \implies 0 \leq y \implies y \leq 1 \implies y \cdot x \leq x$
by (auto simp add: mult-le-cancel-right2)

end

Absolute Value

context linordered-idom
begin

lemma mult-sgn-abs:
$\text{sgn } x \cdot |x| = x$

unfolding abs-if sgn-if by auto

lemma abs-one [simp]:
$|1| = 1$
by (simp add: abs-if)

end

class ordered-ring-abs = ordered-ring + ordered-ab-group-add-abs +
  assumes abs-eq-mult:
    \((0 \leq a \lor a \leq 0) \land (0 \leq b \lor b \leq 0) \implies |a \times b| = |a| \times |b|\)

context linordered-idom
begin

subclass ordered-ring-abs proof
  qed (auto simp add: abs-if not-less mult-less-0-iff)

lemma abs-mult:
  \(|a \times b| = |a| \times |b|\)
  by (rule abs-eq-mult) auto

lemma abs-mult-self:
  \(|a| \times |a| = a \times a|\)
  by (simp add: abs-if)

lemma abs-mult-less:
  \(|a| < c \implies |b| < d \implies |a| \times |b| < c \times d|\)
  proof –
    assume ac: \(|a| < c\)
    hence cpos: \(0 < c\) by (blast intro: le-less-trans abs-ge-zero)
    assume \(|b| < d\)
    thus \(?thesis\) by (simp add: ac cpos mult-strict-mono)
  qed

lemma abs-less-iff:
  \(|a| < b \iff a < b \land -a < b|\)
  by (simp add: abs-less-iff) (auto simp add: abs-if)

lemma abs-mult-pos:
  \(0 \leq x \implies |y| \times x = |y \times x|\)
  by (simp add: abs-mult)

lemma abs-diff-less-iff:
  \(|x - a| < r \iff a - r < x \land x < a + r|\)
  by (auto simp add: diff-less-eq ac-simps abs-less-iff)

end

code-identifier
code-module Rings → (SML) Arith and (OCaml) Arith and (Haskell) Arith
15 Fields: Fields

theory Fields
imports Rings
begin

15.1 Division rings

A division ring is like a field, but without the commutativity requirement.

class inverse =
  fixes inverse :: 'a ⇒ 'a
  and divide :: 'a ⇒ 'a ⇒ 'a (infixl '/')

Lemmas divide-simps move division to the outside and eliminates them on (in)equalities.

ML ⟨⟨
structure Divide-Simps = Named-Thms
{
  val name = @
  {binding divide-simps}
  val description = rewrite rules to eliminate divisions
}
⟩⟩

setup Divide-Simps.

class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]: a ≠ 0 ⇒ inverse a * a = 1
  assumes right-inverse [simp]: a ≠ 0 ⇒ a * inverse a = 1
  assumes divide-inverse: a / b = a * inverse b
begin

subclass ring-1-no-zero-divisors
proof
  fix a b :: 'a
  assume a: a ≠ 0 and b: b ≠ 0
  show a * b ≠ 0
  proof
    assume ab: a * b = 0
    hence 0 = inverse a * (a * b) * inverse b by simp
    also have ... = (inverse a * a) * (b * inverse b)
      by (simp only: mult.assoc)
    also have ... = 1 using a b by simp
    finally show False by simp
  qed

lemma nonzero-imp-inverse-nonzero:
  a ≠ 0 ⇒ inverse a ≠ 0
proof
  assume ianz: inverse a = 0
  assume a ≠ 0
  hence 1 = a * inverse a by simp
  also have ... = 0 by (simp add: ianz)
  finally have 1 = 0 .
  thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
  inverse a = 0 ⇒ a = 0
apply (rule classical)
apply (drule nonzero-imp-inverse-nonzero)
apply auto
done

lemma inverse-unique:
  assumes ab: a * b = 1
  shows inverse a = b
proof
  have a ≠ 0 using ab by (cases a = 0) simp-all
  moreover have inverse a * (a * b) = inverse a by (simp add: ab)
  ultimately show ?thesis by (simp add: mult.assoc [symmetric])
qed

lemma nonzero-inverse-minus-eq:
  a ≠ 0 ⇒ inverse (−a) = −inverse a
by (rule inverse-unique) simp

lemma nonzero-inverse-inverse-eq:
  a ≠ 0 ⇒ inverse (inverse a) = a
by (rule inverse-unique) simp

lemma nonzero-inverse-eq-imp-eq:
  assumes inverse a = inverse b and a ≠ 0 and b ≠ 0
  shows a = b
proof
  from inverse a = inverse b
  have inverse (inverse a) = inverse (inverse b) by (rule arg-cong)
  with ⟨a ≠ 0⟩ and ⟨b ≠ 0⟩ show a = b
  by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-1 [simp]: inverse 1 = 1
by (rule inverse-unique) simp

lemma nonzero-inverse-mult-distrib:
  assumes a ≠ 0 and b ≠ 0
  shows inverse (a * b) = inverse b * inverse a
proof –
  have a * (b * inverse b) * inverse a = 1 using assms by simp
  hence a * b * (inverse b * inverse a) = 1 by (simp only: mult.assoc)
  thus ?thesis by (rule inverse-unique)
qed

lemma division-ring-inverse-add:
  a ≠ 0 ⇒ b ≠ 0 ⇒ inverse a + inverse b = inverse a * (a + b) * inverse b
  by (simp add: algebra-simps)

lemma division-ring-inverse-diff:
  a ≠ 0 ⇒ b ≠ 0 ⇒ inverse a - inverse b = inverse a * (b - a) * inverse b
  by (simp add: algebra-simps)

lemma right-inverse-eq: b ≠ 0 ⇒ a / b = 1 ↔ a = b
  proof
    assume neq: b ≠ 0
    { hence a = (a / b) * b by (simp add: divide-inverse mult.assoc)
      also assume a / b = 1
      finally show a = b by simp
    next
      assume a = b
      with neq show a / b = 1 by (simp add: divide-inverse)
    }
  qed

lemma nonzero-inverse-eq-divide: a ≠ 0 ⇒ inverse a = 1 / a
  by (simp add: divide-inverse)

lemma divide-self [simp]: a ≠ 0 ⇒ a / a = 1
  by (simp add: divide-inverse)

lemma divide-zero-left [simp]: 0 / a = 0
  by (simp add: divide-inverse)

lemma inverse-eq-divide [field-simps, divide-simps]: inverse a = 1 / a
  by (simp add: divide-inverse)

lemma add-divide-distrib: (a+b) / c = a/c + b/c
  by (simp add: divide-inverse algebra-simps)

lemma divide-1 [simp]: a / 1 = a
  by (simp add: divide-inverse)

lemma times-divide-eq-right [simp]: a * (b / c) = (a * b) / c
  by (simp add: divide-inverse mult.assoc)

lemma minus-divide-left: - (a / b) = (-a) / b
by (simp add: divide-inverse)

lemma nonzero-minus-divide-right: \( b \neq 0 \implies -(a \div b) = a \div (-b) \)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma nonzero-minus-divide-divide: \( b \neq 0 \implies (-a) \div (-b) = a \div b \)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma divide-minus-left [simp]: \((-a) \div b = -(a \div b)\)
by (simp add: divide-inverse)

lemma diff-divide-distrib: \( (a - b) \div c = a \div c - b \div c \)
using add-divide-distrib [of \(a - b\) \(c\)] by simp

lemma nonzero-eq-divide-eq [field-simps]: \( c \neq 0 \implies a = b \div c \iff a \times c = b \)
proof –
  assume [simp]: \( c \neq 0 \)
  have \( a = b \div c \iff a \times c = (b \div c) \times c \) by simp
  also have \(... \iff a \times c = b \) by (simp add: divide-inverse mult.assoc)
  finally show \(?thesis\).
qed

lemma nonzero-divide-eq-eq [field-simps]: \( c \neq 0 \implies b \div c = a \iff b = a \times c \)
proof –
  assume [simp]: \( c \neq 0 \)
  have \( b \div c = a \iff (b \div c) \times c = a \times c \) by simp
  also have \(... \iff b = a \times c \) by (simp add: divide-inverse mult.assoc)
  finally show \(?thesis\).
qed

lemma nonzero-neg-divide-eq-eq [field-simps]: \( b \neq 0 \implies -(a \div b) = c \iff c \times b = -a \)
using nonzero-divide-eq-eq[of \(-a\) \(b\) \(c\)] by (simp add: divide-minus-left)

lemma nonzero-neg-divide-eq-eq2 [field-simps]: \( b \neq 0 \implies c = -(a \div b) \iff c \times b = -a \)
using nonzero-neg-divide-eq-eq[of \(b\) \(a\) \(c\)] by auto

lemma divide-eq-imp: \( c \neq 0 \implies b = a \times c \implies b \div c = a \)
by (simp add: divide-inverse mult.assoc)

lemma eq-divide-imp: \( c \neq 0 \implies a \times c = b \implies a = b \div c \)
by (drule sym) (simp add: divide-inverse mult.assoc)

lemma add-divide-eq-iff [field-simps]: \( z \neq 0 \implies x \div z + y \div z = (x \times z + y) \div z \)
by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma divide-add-eq-iff [field-simps]:
z \neq 0 \Rightarrow x / z + y = (x + y \cdot z) / z
by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma diff-divide-eq-iff [field-simps]:
z \neq 0 \Rightarrow x - y / z = (x \cdot z - y) / z
by (simp add: diff-divide-distrib nonzero-eq-divide-eq eq-diff-eq)

lemma minus-divide-add-eq-iff [field-simps]:
z \neq 0 \Rightarrow - (x / z) + y = (- x + y \cdot z) / z
by (simp add: add-divide-distrib diff-divide-eq-iff divide-minus-left)

lemma divide-diff-eq-iff [field-simps]:
z \neq 0 \Rightarrow x / z - y = (x - y \cdot z) / z
by (simp add: field-simps)

lemma minus-divide-diff-eq-iff [field-simps]:
z \neq 0 \Rightarrow - (x / z) - y = (- x - y \cdot z) / z
by (simp add: divide-diff-eq-iff [symmetric] divide-minus-left)

class division-ring-inverse-zero = division-ring +
  assumes inverse-zero [simp]: inverse 0 = 0
begin

lemma divide-zero [simp]:
a / 0 = 0
by (simp add: divide-inverse)

lemma divide-self-if [simp]:
a / a = (if a = 0 then 0 else 1)
by simp

lemma inverse-nonzero-iff-nonzero [simp]:
inverse a = 0 \iff a = 0
by rule (fact inverse-zero-imp-zero, simp)

lemma inverse-minus-eq [simp]:
inverse (- a) = - inverse a
proof cases
  assume a=0 thus ?thesis by simp
next
  assume a\neq 0
  thus ?thesis by (simp add: nonzero-inverse-minus-eq)
qed

lemma inverse-inverse-eq [simp]:
inverse (inverse a) = a
proof cases
\textbf{THEORY “Fields”}

\begin{verbatim}
assume \texttt{a=0} thus \texttt{thesis by simp}
next
assume \texttt{a\neq 0}
thus \texttt{thesis by \{simp add: nonzero-inverse-inverse-eq\}}
qed

lemma inverse-eq-imp-eq:
inverse a = inverse b \implies a = b
by \{drule arg-cong \[where \texttt{f=inverse}\], simp\}

lemma inverse-eq-iff-imp-eq: [simp]:
inverse a = inverse b \iff a = b
by \{force dest \texttt{inverse-eq-imp-eq}\}

lemma add-divide-eq-iff: [simp]:
\begin{align*}
a + b / z &= (if \, z = 0 \, then \, a \, else \, (a * z + b) / z) \\
a / z + b &= (if \, z = 0 \, then \, b \, else \, (a + b * z) / z) \\
-a / z + b &= (if \, z = 0 \, then \, b \, else \, (-a + b * z) / z) \\
a - b / z &= (if \, z = 0 \, then \, a \, else \, (a * z - b) / z) \\
a / z - b &= (if \, z = 0 \, then \, -a \, else \, (a - b * z) / z) \\
-a / z - b &= (if \, z = 0 \, then \, -b \, else \, (-a - b * z) / z)
\end{align*}
by \{simp-all add: add-divide-eq-iff divide-add-eq-iff diff-divide-eq-iff divide-diff-eq-iff minus-divide-diff-eq-iff\}

lemma divide-simps: [divide-simps]:
shows divide-eq-eq: \texttt{b / c = a} \iff (if \texttt{c \neq 0} then \texttt{b = a \ast c} else \texttt{a = 0})
and eq-divide-eq: \texttt{a = b / c} \iff (if \texttt{c \neq 0} then \texttt{a \ast c = b} else \texttt{a = 0})
and minus-divide-eq-eq: \texttt{- (b / c) = a} \iff (if \texttt{c \neq 0} then \texttt{- b = a \ast c} else \texttt{a = 0})
and eq-minus-divide-eq: \texttt{a = - (b / c)} \iff (if \texttt{c \neq 0} then \texttt{a \ast c = - b} else \texttt{a = 0})
by \{auto simp add: field-simps\}
end

15.2 Fields

class field = comm-ring-1 + inverse +
assumes field-inverse: \texttt{a \neq 0} \implies inverse \texttt{a \ast a = 1}
assumes field-divide-inverse: \texttt{a / b = a \ast inverse b}
begin
subclass division-ring
proof
fix \texttt{a :: 'a}
assume \texttt{a \neq 0}
thus inverse \texttt{a \ast a = 1} by \{rule field-inverse\}
thus \texttt{a \ast inverse a = 1} by \{simp only: mult.commute\}
next
\end{verbatim}
fix $a\ b :: 'a$
show $a / b = a \cdot inverse\ b$ by (rule field-divide-inverse)
qed

subclass idom ..

There is no slick version using division by zero.

lemma inverse-add:

\[
\begin{align*}
&\text{if } a \neq 0; \ b \neq 0 \text{ then } \\
&\text{inverse } a + \text{inverse } b = (a + b) \cdot \text{inverse } a \cdot \text{inverse } b \\
&\text{by (simp add: division-ring-inverse-add ac-simps)}
\end{align*}
\]

lemma nonzero-mult-divide-mult-cancel-left [simp]:
assumes [simp]: $b \neq 0$ and [simp]: $c \neq 0$
shows $(c \cdot a) / (c \cdot b) = a / b$
proof –
have $(c \cdot a) / (c \cdot b) = c \cdot a \cdot (\text{inverse } b \cdot \text{inverse } c)$
  by (simp add: divide-inverse nonzero-inverse-mult-distrib)
also have ... = $a \cdot \text{inverse } b \cdot (\text{inverse } c \cdot c)$
  by (simp only: ac-simps)
also have ... = $a \cdot \text{inverse } b$ by simp
finally show ?thesis by (simp add: divide-inverse)
qed

lemma nonzero-mult-divide-mult-cancel-right [simp]:
[b \neq 0; c \neq 0] \Rightarrow (a \cdot c) / (b \cdot c) = a / b
by (simp add: mult.commute [of - c])

lemma times-divide-eq-left [simp]: $(b / c) \cdot a = (b \cdot a) / c$
by (simp add: divide-inverse ac-simps)

It’s not obvious whether times-divide-eq should be simprules or not. Their
effect is to gather terms into one big fraction, like $a \cdot b \cdot c \div x \cdot y \cdot z$. The
rationale for that is unclear, but many proofs seem to need them.

tepend on times-divide-eq-right times-divide-eq-left

lemma add-frac-eq:
assumes $y \neq 0$ and $z \neq 0$
shows $x / (y + w) / z = (x \cdot z + w \cdot y) / (y \cdot z)$
proof –
have $x / (y + w) / z = (x \cdot z) / (y \cdot z) + (y \cdot w) / (y \cdot z)$
  using assms by simp
also have ... = $(x + y \cdot w) / (y \cdot z)$
  by (simp only: add-distribute)
finally show ?thesis
  by (simp only: mult.commute)
qed

Special Cancellation Simprules for Division

lemma nonzero-mult-divide-cancel-right [simp]:
THEORY “Fields” 309

\[ b \neq 0 \implies a * b / b = a \]

using \texttt{nonzero-mult-divide-mult-cancel-right [of 1 b a] by simp}

\textbf{lemma \textit{nonzero-mult-divide-cancel-left [simp]}}:
\[ a \neq 0 \implies a * b / a = b \]

using \texttt{nonzero-mult-divide-cancel-left [of 1 a b] by simp}

\textbf{lemma \textit{nonzero-divide-mult-cancel-right [simp]}}:
\[ [a \neq 0; b \neq 0] \implies b / (a * b) = 1 / a \]

using \texttt{nonzero-mult-divide-cancel-right [of 1 b a] by simp}

\textbf{lemma \textit{nonzero-divide-mult-cancel-left [simp]}}:
\[ [a \neq 0; b \neq 0] \implies a / (a * b) = 1 / b \]

using \texttt{nonzero-mult-divide-cancel-left [of a b 1] by simp}

\textbf{lemma \textit{nonzero-mult-divide-mult-cancel-right2 [simp]}}:
\[ [b \neq 0; c \neq 0] \implies (a * c) / (b + c) = a / b \]

using \texttt{nonzero-mult-divide-mult-cancel-left [of b c a] by (simp add: ac-simps)}

\textbf{lemma \textit{diff-frac-eq}}:
\[ y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z) \]

by \texttt{(simp add: field-simps)}

\textbf{lemma \textit{frac-eq-eq}}:
\[ y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y) \]

by \texttt{(simp add: field-simps)}

end

class field-inverse-zero = field +
  \textbf{assumes} field-inverse-zero: inverse 0 = 0

begin

subclass division-ring-inverse-zero \textbf{proof}
\texttt{qed (fact field-inverse-zero)}

This version builds in division by zero while also re-orienting the right-hand side.

\textbf{lemma \textit{inverse-mult-distrib [simp]}}:
\[ \text{inverse} (a * b) = \text{inverse} a * \text{inverse} b \]

\textbf{proof cases}
\begin{itemize}
  \item assume \( a \neq 0 \) \& \( b \neq 0 \)
  \item \textbf{thus} \( ?\text{thesis} \) \texttt{by (simp add: nonzero-inverse-mult-distrib ac-simps)}
\end{itemize}

\textbf{next}
\begin{itemize}
  \item assume \( \sim (a \neq 0 \text{ \& } b \neq 0) \)
\end{itemize}
thus ?thesis by force
qed

lemma inverse-divide [simp]:
inverse (a / b) = b / a
by (simp add: divide-inverse mult.commute)

Calculations with fractions

There is a whole bunch of simp-rules just for class field but none for class field and nonzero-divides because the latter are covered by a simproc.

lemma mult-divide-mult-cancel-left:
c ≠ 0 ⟹ (c * a) / (c * b) = a / b
apply (cases b = 0)
apply simp-all
done

lemma mult-divide-mult-cancel-right:
c ≠ 0 ⟹ (a * c) / (b * c) = a / b
apply (cases b = 0)
apply simp-all
done

lemma divide-divide-eq-right [simp]:
a / (b / c) = (a * c) / b
by (simp add: divide-inverse ac-simps)

lemma divide-divide-eq-left [simp]:
(a / b) / c = a / (b * c)
by (simp add: divide-inverse mult.assoc)

lemma divide-divide-times-eq:
(x / y) / (z / w) = (x * w) / (y * z)
by simp

Special Cancellation Simprules for Division

lemma mult-divide-mult-cancel-left-if [simp]:
shows (c * a) / (c * b) = (if c = 0 then 0 else a / b)
by (simp add: mult-divide-mult-cancel-left)

Division and Unary Minus

lemma minus-divide-right:
− (a / b) = a / − b
by (simp add: divide-inverse)

lemma divide-minus-right [simp]:
a / − b = − (a / b)
by (simp add: divide-inverse)
lemma minus-divide-divide:\n\[ (-a) / (-b) = a / b \]
apply (cases b=0, simp)
apply (simp add: nonzero-minus-divide-divide)
done

lemma inverse-eq-1-iff [simp]:
\[ \text{inverse } x = 1 \iff x = 1 \]
by (insert inverse-eq-iff-eq [of x 1], simp)

lemma divide-eq-0-iff [simp]:
\[ a / b = 0 \iff a = 0 \lor b = 0 \]
by (simp add: divide-inverse)

lemma divide-cancel-right [simp]:
\[ a / c = b / c \iff c = 0 \lor a = b \]
apply (cases c=0, simp)
apply (simp add: divide-inverse)
done

lemma divide-cancel-left [simp]:
\[ c / a = e / b \iff c = 0 \lor a = b \]
apply (cases c=0, simp)
apply (simp add: divide-inverse)
done

lemma divide-eq-1-iff [simp]:
\[ a / b = 1 \iff b \neq 0 \land a = b \]
apply (cases b=0, simp)
apply (simp add: right-inverse-eq)
done

lemma one-eq-divide-iff [simp]:
\[ 1 = a / b \iff b \neq 0 \land a = b \]
by (simp add: eq-commute [of 1])

lemma times-divide-times-eq:
\[ (x / y) * (z / w) = (x * z) / (y * w) \]
by simp

lemma add-frac-num:
\[ y \neq 0 \Rightarrow x / y + z = (x + z * y) / y \]
by (simp add: add-divide-distrib)

lemma add-num-frac:
\[ y \neq 0 \Rightarrow z + x / y = (x + z * y) / y \]
by (simp add: add-divide-distrib add.commute)

end
15.3 Ordered fields

class linordered-field = field + linordered-idom
begin

lemma positive-imp-inverse-positive:
  assumes a-gt-0: 0 < a
  shows 0 < inverse a
proof −
  have 0 < a * inverse a
    by (simp add: a-gt-0 [THEN less-imp-not-eq2])
  thus 0 < inverse a
    by (simp add: a-gt-0 [THEN less-not-sym] zero-less-mult-iff)
qed

lemma negative-imp-inverse-negative:
  a < 0 ⇒ inverse a < 0
by (insert positive-imp-inverse-positive [of −a],
  simp add: nonzero-inverse-minus-eq less-imp-not-eq)

lemma inverse-le-imp-le:
  assumes invle: inverse a ≤ inverse b and apos: 0 < a
  shows b ≤ a
proof (rule classical)
  assume ∼ b ≤ a
  hence a < b by (simp add: linorder-not-le)
  hence bpos: 0 < b by (blast intro: apos less-trans)
  hence a * inverse a ≤ a * inverse b
    by (simp add: apos invle less-imp-le mult-left-mono)
  hence (a * inverse a) * b ≤ (a * inverse b) * b
    by (simp add: bpos less-imp-le mult-right-mono)
  thus b ≤ a by (simp add: mult.assoc apos bpos less-imp-not-eq2)
qed

lemma inverse-positive-imp-positive:
  assumes inv-gt-0: 0 < inverse a and nz: a ≠ 0
  shows 0 < a
proof −
  have 0 < inverse (inverse a)
    using inv-gt-0 by (rule positive-imp-inverse-positive)
  thus 0 < a
    using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-negative-imp-negative:
  assumes inv-less-0: inverse a < 0 and nz: a ≠ 0
  shows a < 0
proof −
  have inverse (inverse a) < 0
    using inv-less-0 by (rule negative-imp-inverse-negative)
thus $a < 0$ using nz by (simp add: nonzero-inverse-inverse-eq)

qed

lemma linordered-field-no-lb:
$\forall x. \exists y. y < x$

proof
fix x::'a
have m1: $- (1::'a) < 0$ by simp
from add-strict-right-mono[OF m1, where c=x]
have $(- 1) + x < x$ by simp
thus $\exists y. y < x$ by blast

qed

lemma linordered-field-no-ub:
$\forall x. \exists y. y > x$

proof
fix x::'a
have m1: $(1::'a) > 0$ by simp
from add-strict-right-mono[OF m1, where c=x]
have $1 + x > x$ by simp
thus $\exists y. y > x$ by blast

qed

lemma less-imp-inverse-less:
assumes less: $a < b$ and apos: $0 < a$
shows inverse $b < inverse a$

proof (rule ccontr)
assume $\sim$ inverse $b < inverse a$
hence inverse $a \leq inverse b$ by simp
hence $\sim$ ($a < b$)
by (simp add: not-less inverse-le-imp-le [OF - apos])
thus False by (rule notE [OF - less])

qed

lemma inverse-less-imp-less:
inverse $a < inverse b$ $\implies$ $0 < a$ $\implies$ $b < a$

apply (simp add: less-le [of inverse $a$] less-le [of $b$])
apply (force dest!: inverse-le-imp-le nonzero-inverse-eq-imp-eq)
done

Both premises are essential. Consider -1 and 1.

lemma inverse-less-iff-less [simp]:
$0 < a$ $\implies$ $0 < b$ $\implies$ inverse $a < inverse b$ $\iff$ $b < a$
by (blast intro: less-imp-inverse-less dest: inverse-less-imp-less)

lemma le-imp-inverse-le:
$a \leq b$ $\implies$ $0 < a$ $\implies$ inverse $b \leq inverse a$
by (force simp add: le-less less-imp-inverse-less)
lemma inverse-le-iff-le [simp]:
\[ 0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a \]
by (blast intro: le-imp-inverse-le dest: inverse-le-imp-le)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma inverse-le-imp-le-neg:
\[ \text{inverse } a \leq \text{inverse } b \implies b < 0 \implies b \leq a \]
apply (rule classical)
apply (subgoal-tac a < 0)
prefer 2 apply force
apply (insert inverse-le-imp-le [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma less-imp-inverse-less-neg:
\[ a < b \implies b < 0 \implies \text{inverse } b < \text{inverse } a \]
apply (subgoal-tac a < 0)
prefer 2 apply (blast intro: less-trans)
apply (insert less-imp-inverse-less [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma inverse-less-imp-less-neg:
\[ \text{inverse } a < \text{inverse } b \implies b < 0 \implies b < a \]
apply (rule classical)
apply (subgoal-tac a < 0)
prefer 2 apply force
apply (insert inverse-less-imp-less [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma inverse-less-iff-less-neg [simp]:
\[ a < 0 \implies \text{inverse } a < \text{inverse } b \iff b < a \]
apply (insert inverse-less-iff-less [of \(-b\) \(-a\)])
apply (simp del: inverse-less-iff-less
add: nonzero-inverse-minus-eq)
done

lemma le-imp-inverse-le-neg:
\[ a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a \]
by (force simp add: le-less less-imp-inverse-less-neg)

lemma inverse-le-iff-le-neg [simp]:
\[ a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a \]
by (blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg)

lemma one-less-inverse:
\[0 < a \Rightarrow a < 1 \Rightarrow 1 < \text{inverse } a\]

using `less-imp-inverse-less` [of \(a \cdot 1\), unfolded inverse-1] .

**lemma** `one-le-inverse`:
\[0 < a \Rightarrow a \leq 1 \Rightarrow 1 \leq \text{inverse } a\]
using `le-imp-inverse-le` [of \(a \cdot 1\), unfolded inverse-1] .

**lemma** `pos-divide-eq` [field-simps]:
\[0 < c \Rightarrow a \leq b \/ c \iff a \cdot c \leq b\]

**proof** –
assume `less`: \(0 < c\)
hence \((a \leq b / c) = (a \cdot c \leq (b / c) \cdot c)\)
by (simp add: `mult-le-cancel-right` `less-not-sym` [OF `less`] del: `times-divide-eq`)
also have \(... = (a \cdot c \leq b)\)
by (simp add: `less-imp-not-eq2` [OF `less`] `divide-inverse` `mult.assoc`)
finally show \(?thesis\).

**qed**

**lemma** `neg-le-divide-eq` [field-simps]:
\[c < 0 \Rightarrow a \leq b \/ c \iff b \leq a \cdot c\]

**proof** –
assume `less`: \(c < 0\)
hence \((a \leq b / c) = ((b / c) \cdot c < a \cdot c)\)
by (simp add: `mult-le-cancel-right` `less-not-sym` [OF `less`] del: `times-divide-eq`)
also have \(... = (b \leq a \cdot c)\)
by (simp add: `less-imp-not-eq` [OF `less`] `divide-inverse` `mult.assoc`)
finally show \(?thesis\).

**qed**

**lemma** `pos-less-divide-eq` [field-simps]:
\[0 < c \Rightarrow b / c < a \iff b < a \cdot c\]

**proof** –
assume `less`: \(0 < c\)
hence \((a < b / c) = (a \cdot c < (b / c) \cdot c)\)
by (simp add: `mult-less-cancel-right` `less-not-sym` [OF `less`] del: `times-divide-eq`)
also have \(... = (a \cdot c < b)\)
by (simp add: `less-imp-not-eq2` [OF `less`] `divide-inverse` `mult.assoc`)
finally show \(?thesis\).

**qed**

**lemma** `neg-less-divide-eq` [field-simps]:
\[c < 0 \Rightarrow a < b / c \iff b < a \cdot c\]

**proof** –
assume `less`: \(c < 0\)
hence \((a < b / c) = ((b / c) \cdot c < a \cdot c)\)
by (simp add: `mult-less-cancel-right` `less-not-sym` [OF `less`] del: `times-divide-eq`)
also have \(... = (b < a \cdot c)\)
by (simp add: `less-imp-not-eq2` [OF `less`] `divide-inverse` `mult.assoc`)
finally show \(?thesis\).

**qed**

**lemma** `pos-divide-less-eq` [field-simps]:
\[0 < c \Rightarrow b / c < a \iff b < a \cdot c\]

**proof** –
assume less: $0 < c$

hence $(b/c < a) = ((b/c) * c < a * c)$

by (simp add: mult-less-cancel-right-disj less-not-sym [OF less] del: times-divide-eq)

also have ... $= (b < a * c)$

by (simp add: less-inv-not-eq2 [OF less] divide-inverse mult.assoc)

finally show ?thesis .

qed

lemma neg-divide-less-eq [field-simps]: $c < 0$ implies $b / c < a$ implies $a * c < b$

proof -

assume less: $c < 0$

hence $(b/c < a) = (a * c < (b/c) * c)$

by (simp add: mult-less-cancel-right-disj less-not-sym [OF less] del: times-divide-eq)

also have ... $= (a * c < b)$

by (simp add: less-inv-not-eq2 [OF less] divide-inverse mult.assoc)

finally show ?thesis .

qed

lemma pos-divide-le-eq [field-simps]: $0 < c$ implies $b / c \leq a$ implies $b \leq a * c$

proof -

assume less: $0 < c$

hence $(b/c \leq a) = ((b/c) * c \leq a * c)$

by (simp add: mult-le-cancel-right less-not-sym [OF less] del: times-divide-eq)

also have ... $= (b \leq a * c)$

by (simp add: less-inv-not-eq2 [OF less] divide-inverse mult.assoc)

finally show ?thesis .

qed

lemma neg-divide-le-eq [field-simps]: $c < 0$ implies $b / c \leq a$ implies $a * c \leq b$

proof -

assume less: $c < 0$

hence $(b/c \leq a) = (a * c \leq (b/c) * c)$

by (simp add: mult-le-cancel-right less-not-sym [OF less] del: times-divide-eq)

also have ... $= (a * c \leq b)$

by (simp add: less-inv-not-eq2 [OF less] divide-inverse mult.assoc)

finally show ?thesis .

qed

The following field-simps rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma pos-le-minus-divide-eq [field-simps]: $0 < c$ implies $b / c \leq a$ implies $a \leq - (b / c)$ implies $a * c \leq - b$

unfolding minus-divide-left by (rule pos-le-divide-eq)

lemma neg-le-minus-divide-eq [field-simps]: $c < 0$ implies $b / c \leq a$ implies $a \leq - (b / c)$ implies $a \leq - b$

unfolding minus-divide-left by (rule neg-le-divide-eq)

lemma pos-less-minus-divide-eq [field-simps]: $0 < c$ implies $a < - (b / c)$ implies $a * c < b$

unfolding minus-divide-left by (rule pos-less-divide-eq)
THEORY “Fields”

\[ c < - b \]

unfolding minus-divide-left by (rule pos-less-divide-eq)

lemma neg-less-minus-divide-eq [field-simps]:
\[ c < 0 \implies a < -(b / c) \iff - b \]

unfolding minus-divide-left by (rule neg-less-divide-eq)

lemma pos-minus-divide-less-eq [field-simps]:
\[ 0 < c \implies -(b / c) < a \iff - b \]

unfolding minus-divide-left by (rule pos-less-divide-eq)

lemma neg-minus-divide-less-eq [field-simps]:
\[ c < 0 \implies -(b / c) < a \iff a - c \]

unfolding minus-divide-left by (rule neg-less-divide-eq)

lemma pos-minus-divide-le-eq [field-simps]:
\[ 0 < c \implies -(b / c) \leq a \iff - b \]

unfolding minus-divide-left by (rule pos-less-divide-eq)

lemma neg-minus-divide-le-eq [field-simps]:
\[ c < 0 \implies -(b / c) \leq a \iff a - c \]

unfolding minus-divide-left by (rule neg-less-divide-eq)

lemma frac-less-eq:
\[ y \neq 0 \implies z \neq 0 \implies x / y < w / z \iff (x * z - w * y) / (y * z) < 0 \]

by (subst less-iff-diff-less-0) (simp add: diff-frac-eq)

lemma frac-le-eq:
\[ y \neq 0 \implies z \neq 0 \implies x / y \leq w / z \iff (x * z - w * y) / (y * z) \leq 0 \]

by (subst le-iff-diff-le-0) (simp add: diff-frac-eq)

Lemmas sign-simps is a first attempt to automate proofs of positivity/negativity needed for field-simps. Have not added sign-simps to field-simps because the former can lead to case explosions.

lemmas (in ~) sign-simps = algebra-simps zero-less-mult-iff mult-less-0-iff

lemmas (in ~) sign-simps = algebra-simps zero-less-mult-iff mult-less-0-iff

lemma divide-pos-pos[simp]:
\[ 0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x / y \]

by (simp add: field-simps)

lemma divide-nonneg-pos:
\[ 0 \leq x \Longrightarrow 0 < y \Longrightarrow 0 \leq x / y \]

by (simp add: field-simps)

lemma divide-neg-pos:
THEORY "Fields"

\[ x < 0 \implies 0 < y \implies x / y < 0 \]
by (simp add: field-simps)

lemma divide-nonpos-pos:
\[ x \leq 0 \implies 0 < y \implies x / y \leq 0 \]
by (simp add: field-simps)

lemma divide-pos-neg:
\[ 0 < x \implies y < 0 \implies x / y < 0 \]
by (simp add: field-simps)

lemma divide-nonneg-neg:
\[ 0 < x \implies y > 0 \implies x / y > 0 \]
by (simp add: field-simps)

lemma divide-neg-neg:
\[ x < 0 \implies y < 0 \implies 0 < x / y \]
by (simp add: field-simps)

lemma divide-nonpos-neg:
\[ x < 0 \implies y < 0 \implies 0 < x / y \]
by (simp add: field-simps)

lemma divide-strict-right-mono:
\[ [a < b; 0 < c] \implies a / c < b / c \]
by (simp add: less-imp-not-eq2 divide-inverse mult-strict-right-mono
positive-imp-inverse-positive)

lemma divide-strict-right-mono-neg:
\[ [b < a; c < 0] \implies a / c < b / c \]
apply (drule divide-strict-right-mono \[ of - - c \], simp)
apply (simp add: less-imp-not-eq nonzero-minus-divide-right \[ symmetric \])
done

The last premise ensures that \( a \) and \( b \) have the same sign

lemma divide-strict-left-mono:
\[ [b < a; 0 < c; 0 < a*b] \implies c / a < c / b \]
by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono)

lemma divide-left-mono:
\[ [b \leq a; 0 \leq c; 0 < a*b] \implies c / a \leq c / b \]
by (auto simp: field-simps zero-less-mult-iff mult-right-mono)

lemma divide-strict-left-mono-neg:
\[ [a < b; c < 0; 0 < a*b] \implies c / a < c / b \]
by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono-neg)

lemma mult-imp-div-pos-le: \( 0 < y \implies x \leq z * y \implies \)
\[ \frac{x}{y} \leq z \]
by (subst pos-divide-le-eq, assumption+)

**lemma** mult-imp-le-div-pos: \(0 < y \implies z \leq x\)
\[ z \leq x / y \]
by (simp add: field-simps)

**lemma** mult-imp-div-pos-less: \(0 < y \implies x < z \star y\)
\[ x / y < z \]
by (simp add: field-simps)

**lemma** mult-imp-less-div-pos: \(0 < y \implies z < x \star y\)
\[ z < x / y \]
by (simp add: field-simps)

**lemma** frac-le: \(0 \leq x \implies x \leq y \leq w \leq z \implies x / z \leq y / w\)
apply (rule mult-imp-div-pos-le)
apply simp
apply (subst times-divide-eq-left)
apply (rule mult-imp-le-div-pos, assumption)
apply (rule mult-mono)
apply simp-all
done

**lemma** frac-less: \(0 \leq x \implies x < y \leq w \leq z \implies x / z < y / w\)
apply (rule mult-imp-div-pos-less)
apply simp
apply (subst times-divide-eq-left)
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-less-le-imp-less)
apply simp-all
done

**lemma** frac-less2: \(0 < x \implies x < y \leq w < z \implies x / z < y / w\)
apply (rule mult-imp-div-pos-less)
apply simp-all
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-less-le-imp-less)
apply simp-all
done

**lemma** less-half-sum: \(a < b \implies a < (a + b) / (1 + 1)\)
by (simp add: field-simps zero-less-two)

**lemma** gt-half-sum: \(a < b \implies (a + b) / (1 + 1) < b\)
by (simp add: field-simps zero-less-two)
subclass unbounded-dense-linorder
proof
  fix $x\ y :: 'a$
  from less-add-one show $\exists y. \ x < y$ ..
  from less-add-one have $x + (-1) < (x + 1) + (-1)$ by (rule add-strict-right-mono)
  then have $x - 1 < x + 1 - 1$ by simp
  then have $x - 1 < x$ by (simp add: algebra-simps)
  then show $\exists y. \ y < x$ ..
  show $x < y \Rightarrow \exists z > x. \ z < y$ by (blast intro: less-half-sum gt-half-sum)
qed

lemma nonzero-abs-inverse:
  $a \neq 0 \Longrightarrow |\text{inverse } a| = \text{inverse } |a|$
apply (auto simp add: neq-iff abs-if nonzero-inverse-minus-eq
                  negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: less-asym)
done

lemma nonzero-abs-divide:
  $b \neq 0 \Longrightarrow |\text{a } / b| = |\text{a }| / |b|$
by (simp add: divide-inverse abs-mult nonzero-abs-inverse)

lemma field-le-epsilon:
  assumes $e: \forall e. \ 0 < e \Rightarrow x \leq y + e$
  shows $x \leq y$
proof (rule dense-le)
  fix $t$ assume $t < x$
  hence $0 < x - t$ by (simp add: less-diff-eq)
  from $e$ [OF this] have $x + 0 \leq x + (y - t)$ by (simp add: algebra-simps)
  then have $0 \leq y - t$ by (simp only: add-le-cancel-left)
  then show $t \leq y$ by (simp add: algebra-simps)
qed

end

class linordered-field-inverse-zero = linordered-field + field-inverse-zero
begin

lemma inverse-positive-iff-positive [simp]:
  $(0 < \text{inverse } a) = (0 < a)$
apply (cases a = 0, simp)
apply (blast intro: inverse-positive-imp-positive positive-imp-inverse-positive)
done

lemma inverse-negative-iff-negative [simp]:
  $(\text{inverse } a < 0) = (a < 0)$
apply (cases a = 0, simp)
apply (blast intro: inverse-negative-imp-negative negative-imp-inverse-negative)
THEORY “Fields” 321

done

lemma inverse-nonnegative-iff-nonnegative [simp]:
\[ 0 \leq \text{inverse } a \iff 0 \leq a \]
by (simp add: not-less [symmetric])

lemma inverse-nonpositive-iff-nonpositive [simp]:
\[ \text{inverse } a \leq 0 \iff a \leq 0 \]
by (simp add: not-less [symmetric])

lemma one-less-inverse-iff: \( 1 < \text{inverse } x \iff 0 < x \land x < 1 \)
using less-trans [of \( 1 x 0 \) for \( x \)]
by (cases \( x 0 \) rule: linorder-cases) (auto simp add: field-simps)

lemma one-le-inverse-iff: \( 1 \leq \text{inverse } x \iff 0 < x \land x \leq 1 \)
proof (cases \( x = 1 \))
  case True then show ?thesis by simp
next
  case False then have \( \text{inverse } x \neq 1 \) by simp
  then have \( 1 \neq \text{inverse } x \) by blast
  then have \( 1 \leq \text{inverse } x \iff 1 < \text{inverse } x \) by (simp add: le-less)
  with False show ?thesis by (auto simp add: one-less-inverse-iff)
qed

lemma inverse-less-1-iff: \( \text{inverse } x < 1 \iff x \leq 0 \lor 1 < x \)
by (simp add: not-le [symmetric] one-le-inverse-iff)

lemma inverse-le-1-iff: \( \text{inverse } x \leq 1 \iff x \leq 0 \lor 1 \leq x \)
by (simp add: not-less [symmetric] one-less-inverse-iff)

lemma [divide-simps]:
shows le-divide-eq: \( a \leq b / c \iff \text{if } 0 < c \text{ then } a * c \leq b \text{ else if } c < 0 \text{ then } b \leq a * c \text{ else } a \leq 0 \)
  and divide-le-eq: \( b / c \leq a \iff \text{if } 0 < c \text{ then } b \leq a * c \text{ else if } c < 0 \text{ then } a \leq c \text{ else } 0 \leq a \)
  and less-divide-eq: \( a < b / c \iff \text{if } 0 < c \text{ then } a \leq c \text{ else if } c < 0 \text{ then } b < a \leq c \text{ else } a < 0 \)
  and divide-less-eq: \( b / c < a \iff \text{if } 0 < c \text{ then } b < a \text{ else if } c < 0 \text{ then } a \leq c \text{ else } b = 0 \leq a \)
  and le-minus-divide-eq: \( a \leq - (b / c) \iff \text{if } 0 < c \text{ then } a * c \leq - b \text{ else if } c < 0 \text{ then } - b \leq a * c \text{ else } a \leq 0 \)
  and minus-divide-le-eq: \( -(b / c) \leq a \iff \text{if } 0 < c \text{ then } - b \leq a * c \text{ else if } c < 0 \text{ then } a * c \leq - b \text{ else } 0 \leq a \)
  and less-minus-divide-eq: \( a < - (b / c) \iff \text{if } 0 < c \text{ then } a * c < - b \text{ else if } c < 0 \text{ then } - b < a * c \text{ else } a < 0 \)
  and minus-divide-less-eq: \( -(b / c) < a \iff \text{if } 0 < c \text{ then } - b < a * c \text{ else if } c < 0 \text{ then } a * c < - b \text{ else } 0 < a \)
by (auto simp: field-simps not-less dest: antisym)

Division and Signs
lemma shows zero-less-divide-iff: $0 < a / b \iff 0 < a \land 0 < b \lor a < 0 \land b < 0$

and divide-less-0-iff: $a / b < 0 \iff 0 < a \land b < 0 \lor a < 0 \land b < 0$

and zero-le-divide-iff: $0 \leq a / b \iff 0 \leq a \land b \leq 0 \lor a \leq 0 \land b \leq 0$

and divide-le-0-iff: $a / b \leq 0 \iff 0 \leq a \land b \leq 0 \lor a \leq 0 \land b \leq 0$

by (auto simp add: divide-simps)

Division and the Number One

Simplify expressions equated with 1

lemma zero-eq-1-divide-iff [simp]: $0 = 1 / a \iff a = 0$

by (cases a = 0) (auto simp: field-simps)

lemma one-divide-eq-0-iff [simp]: $1 / a = 0 \iff a = 0$

using zero-eq-1-divide-iff [of a] by simp

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma zero-le-divide-1-iff [simp]:
$0 \leq 1 / a \iff 0 \leq a$

by (simp add: zero-le-divide-iff)

lemma zero-less-divide-1-iff [simp]:
$0 < 1 / a \iff 0 < a$

by (simp add: zero-less-divide-iff)

lemma divide-le-0-1-iff [simp]:
$1 / a \leq 0 \iff a \leq 0$

by (simp add: divide-le-0-iff)

lemma divide-less-0-1-iff [simp]:
$1 / a < 0 \iff a < 0$

by (simp add: divide-less-0-iff)

lemma divide-right-mono:
$[a \leq b; 0 \leq c] \Longrightarrow a/c \leq b/c$

by (force simp add: divide-strict-right-mono le-less)

lemma divide-right-mono-neg: $a \leq b$

$\Longrightarrow c \leq 0 \Longrightarrow b / c \leq a / c$

apply (drule divide-right-mono [of - - - c])

apply auto

done

lemma divide-left-mono-neg: $a \leq b$

$\Longrightarrow c \leq 0 \Longrightarrow 0 < a \times b \Longrightarrow c / a \leq c / b$

apply (drule divide-left-mono [of - - - c])

apply (auto simp add: mult.commute)

done
lemma inverse-le-iff: inverse a ≤ inverse b ⟷ (0 < a * b ⟷ b ≤ a) ∧ (a * b ≤ 0 ⟷ a ≤ b)
  by (cases a 0 b 0 rule: linorder-cases[case-product linorder-cases])
    (auto simp add: field-simps zero-less-mult-iff mult-le-0-iff)

lemma inverse-less-iff: inverse a < inverse b ⟷ (0 < a * b ⟷ b < a) ∧ (a * b ≤ 0 ⟷ a < b)
  by (subst less-le) (auto simp add: inverse-le-iff)

lemma divide-le-cancel: a / c ≤ b / c ⟷ (0 < c ⟷ a ≤ b) ∧ (c < 0 ⟷ b ≤ a)
  by (simp add: divide-inverse mult-le-cancel-right)

lemma divide-less-cancel: a / c < b / c ⟷ (0 < c ⟷ a < b) ∧ (c < 0 ⟷ b < a) ∧ c ≠ 0
  by (auto simp add: divide-inverse mult-less-cancel-right)

Simplify quotients that are compared with the value 1.

lemma le-divide-eq-1:
  (1 ≤ b / a) = (((0 < a & a ≤ b) & a < 0 & b ≤ a))
  by (auto simp add: le-divide-eq)

lemma divide-le-eq-1:
  (b / a ≤ 1) = (((0 < a & b ≤ a) & a < 0 & a ≤ b) & a=0)
  by (auto simp add: divide-le-eq)

lemma less-divide-eq-1:
  (1 < b / a) = (((0 < a & a < b) & a < 0 & a < b))
  by (auto simp add: less-divide-eq)

lemma divide-less-eq-1:
  (b / a < 1) = (((0 < a & b < a) & a < 0 & a < b) & a=0)
  by (auto simp add: divide-less-eq)

lemma divide-nonneg-nonneg [simp]:
  0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x / y
  by (auto simp add: divide-simps)

lemma divide-nonpos-nonpos:
  x ≤ 0 ⟹ y ≤ 0 ⟹ 0 ≤ x / y
  by (auto simp add: divide-simps)

lemma divide-nonneg-nonpos:
  0 ≤ x ⟹ y ≤ 0 ⟹ x / y ≤ 0
  by (auto simp add: divide-simps)

lemma divide-nonpos-nonneg:
  x ≤ 0 ⟹ 0 ≤ y ⟹ x / y ≤ 0
  by (auto simp add: divide-simps)
Conditional Simplification Rules: No Case Splits

**lemma** le-divide-eq-1-pos [simp]:
\[ 0 < a \implies (1 \leq b/a) = (a \leq b) \]
by (auto simp add: le-divide-eq)

**lemma** le-divide-eq-1-neg [simp]:
\[ a < 0 \implies (1 \leq b/a) = (b \leq a) \]
by (auto simp add: le-divide-eq)

**lemma** divide-le-eq-1-pos [simp]:
\[ 0 < a \implies (b/a \leq 1) = (b \leq a) \]
by (auto simp add: divide-le-eq)

**lemma** divide-le-eq-1-neg [simp]:
\[ a < 0 \implies (b/a \leq 1) = (a \leq b) \]
by (auto simp add: divide-le-eq)

**lemma** less-divide-eq-1-pos [simp]:
\[ 0 < a \implies (1 < b/a) = (a < b) \]
by (auto simp add: less-divide-eq)

**lemma** less-divide-eq-1-neg [simp]:
\[ a < 0 \implies (1 < b/a) = (b < a) \]
by (auto simp add: less-divide-eq)

**lemma** divide-less-eq-1-pos [simp]:
\[ 0 < a \implies (b/a < 1) = (b < a) \]
by (auto simp add: divide-less-eq)

**lemma** divide-less-eq-1-neg [simp]:
\[ a < 0 \implies b/a < 1 \iff a < b \]
by (auto simp add: divide-less-eq)

**lemma** eq-divide-eq-1 [simp]:
\[ (1 = b/a) = ((a \neq 0 \& a = b)) \]
by (auto simp add: eq-divide-eq)

**lemma** divide_eq-eq-1 [simp]:
\[ (b/a = 1) = ((a \neq 0 \& a = b)) \]
by (auto simp add: divide_eq-eq)

**lemma** abs-inverse [simp]:
\[ |inverse a| = inverse |a| \]
apply (cases a=0, simp)
apply (simp add: nonzero-abs-inverse)
done

**lemma** abs-divide [simp]:
theory Nat
imports Inductive Typedef Fun Fields
begin

ML-file ~/src/Tools/rat.ML
ML-file Tools/arith-data.ML
ML-file ~/src/Provers/Arith/fast-lin-arith.ML

16 Nat: Natural numbers

end
16.1 Type ind
typedecl ind

axiomatization Zero-Rep :: ind and Suc-Rep :: ind => ind where
— the axiom of infinity in 2 parts
Suc-Rep-inject:   Suc-Rep x = Suc-Rep y ==> x = y and

16.2 Type nat

Type definition
inductive Nat :: ind ⇒ bool where
Zero-RepI: Nat Zero-Rep
| Suc-RepI: Nat i ⇒ Nat (Suc-Rep i)

typedef nat = { n. Nat n}
morphisms Rep-Nat Abs-Nat
using Nat.Zero-RepI by auto

lemma Nat-Rep-Nat:
Nat (Rep-Nat n)
using Rep-Nat by simp

lemma Nat-Abs-Nat-inverse:
Nat n ⇒ Rep-Nat (Abs-Nat n) = n
using Abs-Nat-inverse by simp

lemma Nat-Abs-Nat-inject:
Nat n ⇒ Nat m ⇒ Abs-Nat n = Abs-Nat m ↔ n = m
using Abs-Nat-inject by simp

instantiation nat :: zero
begin

definition Zero-nat-def:
0 = Abs-Nat Zero-Rep

instance ..
end

definition Suc :: nat ⇒ nat where
Suc n = Abs-Nat (Suc-Rep (Rep-Nat n))

lemma Suc-not-Zero: Suc m ≠ 0
lemma Zero-not-Suc: \( 0 \neq \text{Suc} \, m \)
by (rule not-sym, rule Suc-not-Zero not-sym)

lemma Suc-Rep-inject': \( \text{Suc-Rep} \, x = \text{Suc-Rep} \, y \iff x = y \)
by (rule iffI, rule Suc-Rep-inject) simp-all

lemma nat-induct0:
  fixes \( n \)
  assumes \( P \, 0 \) and \( \forall n. \, P \, n \implies P \, (\text{Suc} \, n) \)
  shows \( P \, n \)
using assms
apply (unfold Zero-nat-def Suc-def)
apply (rule Rep-Nat-inverse [THEN subst]) — types force good instantiation
apply (erule Nat-Rep-Nat [THEN Nat.induct!])
apply (iprover elim: Nat-Abs-Nat-inverse [THEN subst])
done

free-constructors case-nat for
  \( 0 :: \text{nat} \)
| \( \text{Suc} \) \( \text{pred} \)
where
  \( \text{pred} \, (0 :: \text{nat}) = (0 :: \text{nat}) \)
apply atomize-elim
apply (rename-tac \( n \), induct-tac \( n \) rule: nat-induct0, auto)
apply (simp only: Suc-not-Zero)
done

— Avoid name clashes by prefixing the output of \texttt{rep-datatype} with \texttt{old}.
setup ⟪ Sign.mandatory-path old ⟪

rep-datatype \( 0 :: \text{nat} \text{Suc} \)
  apply (erule nat-induct0, assumption)
apply (rule nat.inject)
apply (rule nat.distinct(1))
done

setup ⟪ Sign.parent-path ⟪

— But erase the prefix for properties that are not generated by \texttt{free-constructors}.
setup ⟪ Sign.mandatory-path nat ⟪

declare
  old.nat.inject[iff del]
  old.nat.distinct(1)[simp del, induct-simp del]

lemmas induct = old.nat.induct
lemmas inducts = old.nat.inducts
lemmas rec = old.nat.rec
lemmas simps = nat.inject nat.distinct nat.case nat.rec

setup ⟨⟨ Sign.parent-path ⟩⟩

abbreviation rec-nat :: 'a ⇒ (nat ⇒ 'a ⇒ 'a) ⇒ nat ⇒ 'a where
  rec-nat ≡ old.rec-nat

declare nat.sel[code del]

hide-const (open) Nat.pred — hide everything related to the selector
hide-fact
  nat.case-eq-if
  nat.collapse
  nat.expand
  nat.sel
  nat.sel-exhaust
  nat.sel-split
  nat.sel-split-asm

lemma nat-exhaust [case-names 0 Suc, cases type: nat]:
  — for backward compatibility – names of variables differ
  \( y = 0 \Rightarrow P \) \( \Rightarrow (\forall n. y = Suc n \Rightarrow P) \Rightarrow P \)
by (rule old.nat.exhaust)

lemma nat-induct [case-names 0 Suc, induct type: nat]:
  — for backward compatibility – names of variables differ
  fixes n
  assumes P 0 and \( \forall n. P n \Rightarrow P (Suc n) \)
  shows P n
using assms by (rule nat.induct)

hide-fact
  nat-exhaust
  nat-induct0

Injectiveness and distinctness lemmas
lemma inj-Suc[simp]: inj-on Suc N
  by (simp add: inj-on-def)

lemma Suc-neq-Zero: Suc m = 0 \( \Rightarrow R \)
by (rule notE, rule Suc-not-Zero)

lemma Zero-neq-Suc: 0 = Suc m \( \Rightarrow R \)
by (rule Suc-neq-Zero, erule sym)

lemma Suc-inject: Suc x = Suc y \( \Rightarrow x = y \)
by (rule inj-Suc [THEN injD])
lemma n-not-Suc-n: n ≠ Suc n
by (induct n) simp-all

lemma Suc-n-not-n: Suc n ≠ n
by (rule not-sym, rule n-not-Suc-n)

A special form of induction for reasoning about m < n and m − n
lemma diff-induct: (!!x. P x 0) ==> (!!y. P 0 (Suc y)) ==>
    (!!x y. P x y ==> P (Suc x) (Suc y)) ==> P m n
apply (rule_tac x = m in spec)
apply (induct n)
prefer 2
apply (rule allI)
apply (induct-tac x, iprover+)
done

16.3 Arithmetic operators

instantiation nat :: comm-monoid-diff
begin

primrec plus-nat where
| add-0:  0 + n = (n::nat)
| add-Suc: Suc m + n = Suc (m + n)

lemma add-0-right [simp]: m + 0 = (m::nat)
by (induct m) simp-all

lemma add-Suc-right [simp]: m + Suc n = Suc (m + n)
by (induct m) simp-all

declare add-0 [code]

lemma add-Suc-shift [code]: Suc m + n = m + Suc n
by simp

primrec minus-nat where
| diff-0 [code]: m − 0 = (m::nat)
| diff-Suc: m − Suc n = (case m − n of 0 => 0 | Suc k => k)

declare diff-Suc [simp del]

lemma diff-0-eq-0 [simp, code]: θ − n = (0::nat)
by (induct n) (simp-all add: diff-Suc)

lemma diff-Suc-Suc [simp, code]: Suc m − Suc n = m − n
by (induct n) (simp-all add: diff-Suc)

instance proof
fix n m q :: nat
show (n + m) + q = n + (m + q) by (induct n) simp-all
show n + m = m + n by (induct n) simp-all
show 0 + n = n by simp
show n - 0 = n by simp
show 0 - n = 0 by simp
show (q + n) - (q + m) = n - m by (induct q) simp-all
show n - m - q = n - (m + q) by (induct q) (simp-all add: diff-Suc)
qed

definition One-nat-def [simp]: 1 = Suc 0

primrec times-nat where
  mult-0: 0 * n = (0::nat)
| mult-Suc: Suc m * n = n + (m * n)

lemma mult-0-right [simp]: (m::nat) * 0 = 0
  by (induct m) simp-all

lemma mult-Suc-right [simp]: m * Suc n = m + (m * n)
  by (induct m) (simp-all add: add.left-commute)

lemma add-mult-distrib: (m + n) * k = (m * k) + ((n * k)::nat)
  by (induct m) (simp-all add: add.assoc)

instance proof
  fix n m q :: nat
  show 0 ≠ (1::nat) unfolding One-nat-def by simp
  show 1 * n = n unfolding One-nat-def by simp
  show n * m = m * n by (induct n) simp-all
  show (n * m) * q = n * (m * q) by (induct n) (simp-all add: add-mult-distrib)
  show (n + m) * q = n * q + m * q by (rule add-mult-distrib)
  assume n + m = n + q thus m = q by (induct n) simp-all
qed

end

hide-fact (open) add-0 add-0-right diff-0

instantiation nat :: comm-semiring-1-cancel
begin


16.3.1 Addition

lemma nat-add-left-cancel:
  fixes k m n :: nat
shows $k + m = k + n \leftrightarrow m = n$
by (fact add-left-cancel)

lemma nat-add-right-cancel:
fixes $k \ m \ n :: \text{nat}$
shows $m + k = n + k \leftrightarrow m = n$
by (fact add-right-cancel)

Reasoning about $m + \theta = \theta$, etc.

lemma add-is-0 [iff]:
fixes $m \ n :: \text{nat}$
shows $(m + n = \text{Suc } 0) = (m = \text{Suc } 0 \& n = \text{Suc } 0)$
by (cases $m$) simp-all

lemma add-is-1:
$(m+n= \text{Suc } 0) = (m= \text{Suc } 0 \& n= \text{Suc } 0) \& (m=0 \& n=0)$
by (cases $m$) simp-all

lemma one-is-add:
$(\text{Suc } 0 = m + n) = (m = \text{Suc } 0 \& n = \text{Suc } 0) \or (m = 0 \& n = \text{Suc } 0)$
by (rule trans, rule eq-commute, rule add-is-1)

lemma add-eq-self-zero:
fixes $m \ n :: \text{nat}$
shows $m + n = m \Rightarrow n = 0$
by (induct $m$) simp-all

lemma inj-on-add-nat[simp]: $\text{inj-on } (\% n :: \text{nat}. \ n + k) \ N$
apply (induct $k$)
apply simp
apply (drule comp-inj-on[OF - inj-Suc])
apply (simp add:o-def)
done

lemma Suc-eq-plus1: $\text{Suc } n = n + 1$
unfolding One-nat-def by simp

lemma Suc-eq-plus1-left: $\text{Suc } n = 1 + n$
unfolding One-nat-def by simp

16.3.2 Difference

lemma diff-self-eq-0 [simp]: $(m :: \text{nat}) - m = 0$
by (fact diff-cancel)

lemma diff-diff-left: $(i :: \text{nat}) - j - k = i - (j + k)$
by (fact diff-diff-add)

lemma Suc-diff-diff [simp]: $(\text{Suc } m - n) - \text{Suc } k = m - n - k$
by \((simp\ add: \ diff\_diff\_left)\)

**lemma** **diff\_commute\:** \((i::nat) - j - k = i - k - j\)
by \((fact\ diff\_right\_commute)\)

**lemma** **diff\_add\_inverse\:** \((n + m) - n = (m::nat)\)
by \((fact\ add\_diff\_cancel\_left')\)

**lemma** **diff\_add\_inverse2\:** \((m + n) - n = (m::nat)\)
by \((fact\ add\_diff\_cancel\_right')\)

**lemma** **diff\_cancel\:** \((k + m) - (k + n) = m - (n::nat)\)
by \((fact\ comm\_monoid\_diff\_class.add\_diff\_cancel\_left)\)

**lemma** **diff\_cancel2\:** \((m + k) - (n + k) = m - (n::nat)\)
by \((fact\ add\_diff\_cancel\_right)\)

**lemma** **diff\_add\_0\:** \(n - (n + m) = (0::nat)\)
by \((fact\ diff\_add\_zero)\)

**lemma** **diff\_Suc\_1 \[simp\]:** \(Suc\ n - 1 = n\)
unfolding **One-nat-def** by simp

Difference distributes over multiplication

**lemma** **diff\_mult\_distrib\:** \((m::nat) - n) * k = (m * k) - (n * k)\)
by \((induct m n rule: \ diff\_induct)\) simp-all add: diff-cancel

**lemma** **diff\_mult\_distrib2\:** \(k * ((m::nat) - n) = (k * m) - (k * n)\)
by \((simp\ add: \ diff\_mult\_distrib\ \ mult\_commute\ [of\ k])\)
— NOT added as rewrites, since sometimes they are used from right-to-left

### 16.3.3 Multiplication

**lemma** **add\_mult\_distrib2\:** \(k * (m + n) = (k * m) + ((k * n)::nat)\)
by \((fact\ distrib\_left)\)

**lemma** **mult\_is\_0 \[simp\]:** \((m::nat) * n = 0) = (m=0 | n=0)\)
by \((induct m)\) auto

**lemmas** **nat\_distrib** =
\add-mul-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

**lemma** **mult\_eq\_1\_iff \[simp\]:** \((m * n = Suc\ 0) = (m = Suc\ 0 \ &\ n = Suc\ 0)\)
apply (induct m)
apply simp
apply (induct n)
apply auto
done
lemma one-eq-mult-iff [simp]: (Suc 0 = m * n) = (m = Suc 0 & n = Suc 0)
  apply (rule trans)
  apply (rule-tac [2] mult-eq-1-iff, fastforce)
done

lemma nat-mult-eq-1-iff [simp]: m * n = (1::nat) ←→ m = 1 & n = 1
  unfolding One-nat-def by (rule mult-eq-1-iff)

lemma nat-1-eq-mult-iff [simp]: (1::nat) = m * n ←→ m = 1 & n = 1
  unfolding One-nat-def by (rule one-eq-mult-iff)

lemma mult-cancel1 [simp]: (k * m = k * n) = (m = n | (k = (0::nat)))
  proof
  have k ≠ 0 ⇒ k * m = k * n ⇒ m = n
  proof (induct n arbitrary: m)
  case 0 then show m = 0 by simp
  next
  case (Suc n) then show m = Suc n
    by (cases m) (simp-all add: eq-commute [of 0])
  qed
  then show ?thesis by auto
  qed

lemma mult-cancel2 [simp]: (m * k = n * k) = (m = n | (k = (0::nat)))
  by (simp add: mult.commute)

lemma Suc-mult-cancel1: (Suc k * m = Suc k * n) = (m = n)
  by (subst mult-cancel1) simp

16.4 Orders on nat
16.4.1 Operation definition

instantiation nat :: linorder
begin

primrec less-eq-nat where
  (0::nat) ≤ n ←→ True |
  Suc m ≤ n ←→ (case n of 0 ⇒ False | Suc n ⇒ m ≤ n)

declare less-eq-nat.simps [simp del]
lemma le0 [iff]: 0 ≤ (n::nat) by (simp add: less-eq-nat.simps)
lemma [code]: (0::nat) ≤ n ←→ True by simp

definition less-nat where
  less-eq-Suc-le: n < m ←→ Suc n ≤ m

lemma Suc-le-mono [iff]: Suc n ≤ Suc m ←→ n ≤ m
  by (simp add: less-eq-nat.simps(2))
lemma Suc-le-eq [code]: Suc \( m \leq n \) \( \iff \) \( m < n \)
unfolding less-eq-Suc-le ..

lemma le-0-eq [iff]: \( (n::nat) \leq 0 \) \( \iff \) \( n = 0 \)
by (induct n) (simp-all add: less-eq-nat.simps(2))

lemma not-less0 [iff]: \( \neg n < (0::nat) \)
by (simp add: less-eq-Suc-le)

lemma less-nat-zero-code [code]: \( n < (0::nat) \) \( \iff \) False
by simp

lemma Suc-less-eq [iff]: Suc \( m < Suc n \) \( \iff \) \( m < n \)
by (simp add: less-eq-Suc-le)

lemma less-Suc-eq-le [code]: \( m < Suc n \) \( \iff \) \( m \leq n \)
by (simp add: less-eq-Suc-le)

lemma Suc-less-eq2: Suc \( n < m \) \( \iff \) \( \exists m'. m = Suc m' \land n < m' \)
by (cases m) auto

lemma le-SucI: \( m \leq n \Rightarrow m \leq Suc n \)
by (induct m arbitrary: n)
  (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma Suc-leD: Suc \( m \leq n \Rightarrow m \leq n \)
by (cases n) (auto intro: le-SucI)

lemma less-SucI: \( m < n \Rightarrow m < Suc n \)
by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD: Suc \( m < n \Rightarrow m < n \)
by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance
proof
  fix \( n m :: nat \)
  show \( n < m \) \( \iff \) \( n \leq m \land \neg m \leq n \)
  proof (induct n arbitrary: m)
    case \( 0 \) then show \(?case\) by (cases m) (simp-all add: less-eq-Suc-le)
    next
    case (Suc n) then show \(?case\) by (cases m) (simp-all add: less-eq-Suc-le)
    qed
  next
  fix \( n :: nat \) show \( n \leq n \) by (induct n) simp-all
  next
  fix \( n m :: nat \) assume \( n \leq m \) and \( m \leq n \)
  then show \( n = m \)
    by (induct n arbitrary: m)
THEORY “Nat”

(simp-all add: less-eq-nat.simps(2) split: nat.splits)

next
fix n m q :: nat assume n ≤ m and m ≤ q
then show n ≤ q
proof (induct n arbitrary: m q)
case 0 show ?case by simp
next
case (Suc n) then show ?case
by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)
qed

next
fix n m :: nat show n ≤ m ∨ m ≤ n
by (induct n arbitrary: m)
(simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed

end

instantiation nat :: order-bot
begin

definition bot-nat :: nat where
  bot-nat = 0

instance proof
  qed (simp add: bot-nat-def)
end

instance nat :: no-top
by default (auto intro: less-Suc-eq-le [THEN iffD2])

16.4.2 Introduction properties

lemma lessI [iff]: n < Suc n
  by (simp add: less-Suc-eq-le)

lemma zero-less-Suc [iff]: 0 < Suc n
  by (simp add: less-Suc-eq-le)

16.4.3 Elimination properties

lemma less-not-refl: ~ n < (n::nat)
  by (rule order-less-irrefl)

lemma less-not-refl2: n < m ==> m ≠ (n::nat)
  by (rule not-sym) (rule less-imp-neq)
lemma less-not-refl3: \((s::nat) < t ==> s \neq t\)
  by (rule less-imp-neq)

lemma less-irrefl-nat: \((n::nat) < n ==> R\)
  by (rule notE, rule less-not-refl)

lemma less-zeroE: \((n::nat) < 0 ==> R\)
  by (rule notE, rule not-less0)

lemma less-Suc-eq: \((m < Suc n) = (m < n | m = n)\)
  unfolding less-Suc-eq-le le-less ..

lemma less-Suc0 [iff]: \((n < Suc 0) = (n = 0)\)
  by (simp add: less-Suc-eq)

lemma less-one [iff]: \((n < (1::nat)) = (n = 0)\)
  unfolding One-nat-def by (rule less-Suc0)

lemma Suc-mono: \(m < n ==> Suc m < Suc n\)
  by simp

"Less than" is antisymmetric, sort of

lemma less-antisym: \([- n < m; n < Suc m] ==> m = n\)
  unfolding not-less less-Suc-eq-le by (rule antisym)

lemma nat-neq-iff: \((m::nat) \neq n) = (m < n | n < m)\)
  by (rule linorder-neq-iff)

lemma nat-less-cases: assumes major: \((m::nat) < n ==> P n m\)
  and eqCase: \(m = n ==> P n m\) and lessCase: \(n < m ==> P n m\)
  shows \(P n m\)
  apply (rule less-linear [THEN disjE])
  apply (erule_tac \[2\] disjE)
  apply (erule lessCase)
  apply (erule sym [THEN eqCase])
  apply (erule major)
  done

16.4.4 Inductive (?) properties

lemma Suc-lessI: \(m < n ==> Suc m \neq n ==> Suc m < n\)
  unfolding less-eq-Suc-le [of m] le-less by simp

lemma lessE:
  assumes major: \(i < k\)
  and p1: \(k = Suc i ==> P\) and p2: \(!j. i < j ==> k = Suc j ==> P\)
  shows \(P\)
  proof –
  from major have \(\exists j. i \leq j \land k = Suc j\)
unfolding less-eq-Suc-le by (induct k) simp-all
then have \((\exists j. i < j \land k = \text{Suc}\ j) \lor k = \text{Suc}\ i\)
  by (clarsimp simp add: less-le)
with p1 p2 simp show P by auto
qed

lemma less-SucE: assumes major: \(m < \text{Suc}\ n\)
  and less: \(m < n \Longrightarrow P\) and eq: \(m = n \Longrightarrow P\)
  shows P apply (rule major [THEN lessE])
  apply (rule eq, blast)
  apply (rule less, blast)
  done

lemma Suc-lessE: assumes major: \(\text{Suc}\ i < k\)
  and minor: \(!j. i < j \Longrightarrow k = \text{Suc}\ j \Longrightarrow P\)
  shows P apply (rule major [THEN lessE])
  apply (erule lessI [THEN minor])
  apply (erule Suc-lessD [THEN minor], assumption)
  done

lemma Suc-less-SucD: \(\text{Suc}\ m < \text{Suc}\ n\)
  by simp

Can be used with less-Suc-eq to get \(n = m \lor n < m\)

lemma not-less-eq: \(\neg m < n \iff n < \text{Suc}\ m\)
  unfolding not-less less-Suc-eq-le ..

lemma not-less-eq-eq: \(\neg m \leq n \iff \text{Suc}\ n \leq m\)
  unfolding not-le Suc-le-eq ..

Properties of "less than or equal"

lemma le-imp-less-Suc: \(m \leq n \Longrightarrow m < \text{Suc}\ n\)
  unfolding less-Suc-eq-le ..

lemma Suc-n-not-le-n: \(\neg \text{Suc}\ n \leq n\)
  unfolding not-le Suc-eq-le ..

lemma le-Suc-eq: \((m \leq \text{Suc}\ n) = (m \leq n \mid m = \text{Suc}\ n)\)
  by (simp add: less-Suc-eq-le [symmetric] less-Suc-eq)

lemma le-SucE: \(m \leq \text{Suc}\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = \text{Suc}\ n \Longrightarrow R)\)
==>
by (drule le-Suc-eq [THEN iffD1], iprover+)

lemma Suc-leI: m < n ==> Suc(m) ≤ n
  unfolding Suc-le-eq.

Stronger version of Suc-leD

lemma Suc-le-lessD: Suc m ≤ n ==> m < n
  unfolding Suc-le-eq.

lemma less-imp-le-nat: m < n ==> m ≤ (n::nat)
  unfolding less-eq-Suc-le by (rule Suc-leD)

For instance, (Suc m < Suc n) = (Suc m ≤ n) = (m < n)

lemmas le-simps = less-imp-le-nat less-Suc-eq-le Suc-le-eq

Equivalence of m ≤ n and m < n ∨ m = n

lemma less-or-eq-imp-le: m < n | m = n ==> m ≤ (n::nat)
  unfolding le-less.

lemma le-eq-less-or-eq: (m ≤ (n::nat)) = (m < n | m=n)
  by (rule le-less)

Useful with blast.

lemma eq-imp-le: (m::nat) = n ==> m ≤ n
  by auto

lemma le-refl: n ≤ (n::nat)
  by simp

lemma le-trans: [i ≤ j; j ≤ k] ==> i ≤ (k::nat)
  by (rule order-trans)

lemma le-antisym: [m ≤ n; n ≤ m] ==> m = (n::nat)
  by (rule antisym)

lemma nat-less-le: ((m::nat) < n) = (m ≤ n & m ≠ n)
  by (rule less-le)

lemma le-neeq-le: (m::nat) ≤ n ==> m ≠ n ==> m < n
  unfolding less-le ..

lemma nat-le-linear: (m::nat) ≤ n | n ≤ m
  by (rule linear)

lemmas linorder-neeqE-nat = linorder-neeqE [where 'a = nat]

lemma le-less-Suc-eq: m ≤ n ==> (n < Suc m) = (n = m)
unfolding  less-Suc-eq-le by auto

lemma  not-less-less-Suc-eq: ∼ n < m ==> (n < Suc m) = (n = m)
   unfolding  not-less by (rule le-less-Suc-eq)

lemmas  not-less-simps = not-less-less-Suc-eq le-less-Suc-eq

lemma  not0-implies-Suc: n ≠ 0 ==> ∃ m. n = Suc m
   by (cases n) simp-all

lemma  gr0-implies-Suc: n > 0 ==> ∃ m. n = Suc m
   by (cases n) simp-all

lemma  gr-implies-not0: fixes n :: nat shows m < n ==> (n ≠ 0)
   by (cases n) simp-all

This theorem is useful with blast

lemma  gr0I: ((n::nat) = 0 ==> False) ==> 0 < n
   by (rule neq0-conv[THEN iffD1], iprover)

lemma  gr0-conv-Suc: (0 < n) = (∃ m. n = Suc m)
   by (fast intro: not0-implies-Suc)

lemma  not-gr0 [iff]: !n::nat. (~ (0 < n)) = (n = 0)
   using  neq0-conv by blast

lemma  Suc-le-D: (Suc n ≤ m') ==> (∃ m. m' = Suc m)
   by (induct m') simp-all

Useful in certain inductive arguments

lemma  less-Suc-eq-0-disj: (m < Suc n) = (m = 0 | (∃ j. m = Suc j & j < n))
   by (cases m) simp-all

16.4.5  Monotonicity of Addition

lemma  Suc-pred [simp]: n>0 ==> Suc (n - Suc 0) = n
   by (simp add: diff-Suc split: nat.split)

lemma  Suc-diff-1 [simp]: 0 < n ==> Suc (n - 1) = n
   unfolding  One-nat-def by (rule Suc-pred)

lemma  nat-add-left-cancel-ge [simp]: (k + m ≤ k + n) = (m ≤ (n::nat))
   by (induct k) simp-all

lemma  nat-add-left-cancel-less [simp]: (k + m < k + n) = (m < (n::nat))
   by (induct k) simp-all
lemma \texttt{add-gr-0} [iff]: \(!m::\text{n}\at\. (m + n > 0) = (m>0 \mid n>0)\)
by (auto dest: gr0-implies-Suc)

strict, in 1st argument

lemma \texttt{add-less-mono1}: \(i < j ==> i + k < j + (k::\text{n})\)
by (induct \(k\)) simp-all

strict, in both arguments

lemma \texttt{add-less-mono}: \[[i < j; k < l] ==> i + k < j + (l::\text{n})\]
apply (rule add-less-mono1 [THEN less-trans], assumption+)
apply (induct \(j\), simp-all)
done

Deleted \texttt{less-natE}; use \texttt{less-imp-Suc-add RS exE}

lemma \texttt{less-imp-Suc-add}: \(m < n ==> (\exists k. n = \text{Suc} (m + k))\)
apply (induct \(n\))
apply (simp-all add: order-le-less)
apply (blast elim!: less-SucE intro!: Nat.add-0-right [symmetric] add-Suc-right [symmetric])
done

lemma \texttt{le-Suc-ex}: \((k::\text{n}) \leq l ==> (\exists n. l = k + n)\)
by (auto simp: less-Suc-eq-le [symmetric] dest: less-imp-Suc-add)

strict, in 1st argument; proof is by induction on \(k > 0\)

lemma \texttt{mult-less-mono2}: \((i::\text{n}) < j ==> 0<k ==> k * i < k * j\)
apply (auto simp: gr0-conv-Suc)
apply (induct-tac \(m\))
apply (simp-all add: add-less-mono)
done

The naturals form an ordered \texttt{comm-semiring-1-cancel}

instance \texttt{nat :: linordered-semidom}
proof
  show \(0 < (1::\text{n})\) by simp
  show \(\wedge m n q :: \text{n}. m \leq n ==> q + m \leq q + n\) by simp
  show \(\wedge m n q :: \text{n}. m < n ==> 0 < q ==> q * m < q * n\) by (simp add: mult-less-mono2)
qed

instance \texttt{nat :: no-zero-divisors}
proof
  fix \(a::\text{n}\) and \(b::\text{n}\) show \(a \sim 0 ==> b \sim 0 ==> a * b \sim 0\) by auto
qed

16.4.6 \texttt{min and max}

lemma \texttt{mono-Suc}: \(\text{mono Suc}\)
THEORY "Nat"

by (rule monoI) simp

lemma min-0L [simp]: min 0 n = (0::nat)
by (rule min-absorb1) simp

lemma min-0R [simp]: min n 0 = (0::nat)
by (rule min-absorb2) simp

lemma min-Suc-Suc [simp]: min (Suc m) (Suc n) = Suc (min m n)
by (simp add: mono-Suc min-of-mono)

lemma min-Suc1:
  min (Suc n) m = (case m of 0 => 0 | Suc m' => Suc(min m' n))
by (simp split: nat.split)

lemma min-Suc2:
  min m (Suc n) = (case m of 0 => 0 | Suc m' => Suc(min m' n))
by (simp split: nat.split)

lemma max-0L [simp]: max 0 n = (n::nat)
by (rule max-absorb2) simp

lemma max-0R [simp]: max n 0 = (n::nat)
by (rule max-absorb1) simp

lemma max-Suc-Suc [simp]: max (Suc m) (Suc n) = Suc(max m n)
by (simp add: mono-Suc max-of-mono)

lemma max-Suc1:
  max (Suc n) m = (case m of 0 => Suc n | Suc m' => Suc(max m' n))
by (simp split: nat.split)

lemma max-Suc2:
  max m (Suc n) = (case m of 0 => Suc n | Suc m' => Suc(max m' n))
by (simp split: nat.split)

lemma nat-mult-min-left:
  fixes m n q :: nat
  shows min m n * q = min (m * q) (n * q)
  by (simp add: min-def not-le) (auto dest: mult-right-le-imp-le mult-right-less-imp-less
  le-less-trans)

lemma nat-mult-min-right:
  fixes m n q :: nat
  shows m * min n q = min (m * n) (m * q)
  by (simp add: min-def not-le) (auto dest: mult-left-le-imp-le mult-left-less-imp-less
  le-less-trans)

lemma nat-add-max-left:
fixes \( m, n, q :: \text{nat} \)
shows \( \max m n + q = \max (m + q) (n + q) \)
by (simp add: max-def)

lemma nat-add-max-right:
fixes \( m, n, q :: \text{nat} \)
shows \( m + \max n q = \max (m + n) (m + q) \)
by (simp add: max-def)

lemma nat-mult-max-left:
fixes \( m, n, q :: \text{nat} \)
shows \( \max m n \cdot q = \max (m \cdot q) (n \cdot q) \)
by (simp add: max-def not-le (auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans))

lemma nat-mult-max-right:
fixes \( m, n, q :: \text{nat} \)
shows \( m \cdot \max n q = \max (m \cdot n) (m \cdot q) \)
by (simp add: max-def not-le (auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans))

16.4.7 Additional theorems about \( op \leq \)

Complete induction, aka course-of-values induction

instance nat :: wellorder proof
  fix \( P \) and \( n :: \text{nat} \)
  assume step: \( \forall n::\text{nat}. (\forall m, m < n \Rightarrow P m) \Rightarrow P n \)
  have \( \forall q. q \leq n \Rightarrow P q \)
  proof (induct n)
    case (0 n)
    have \( P 0 \) by (rule step) auto
    thus \( \{\text{case using 0 by auto} \)
  next
    case (Suc m n)
    then have \( n \leq m \lor n = \text{Suc} m \) by (simp add: le-Suc-eq)
    thus \( \{\text{case proof} \)
      assume \( n \leq m \) thus \( P n \) by (rule Suc(1))
  next
    assume \( n: n = \text{Suc} m \)
    show \( P n \)
      by (rule step) (rule Suc(1), simp add: le-simps)
  qed
  qed
  then show \( P n \) by auto
  qed

lemma Least-eq-0[simp]: \( P(0::\text{nat}) \Rightarrow \text{Least} P = 0 \)
by (rule Least-equality[OF le0])

lemma Least-Suc:
\[ \leq (\text{LEAST } n. P n) \] == (LEAST m. P(Suc m))
apply (cases n, auto)
apply (frule LeastI)
apply (erule-tac \[ \leq (\text{LEAST } x. P (\text{Suc } x)) \])
apply (erule-tac \[ \leq (\text{LEAST } x. P (\text{Suc } x)) \])
apply (auto)
done

lemma Least-Suc2:
\[ \leq \text{Least } P = \text{Suc } (\text{Least } Q) \] == (LEAST x. P (Suc x), auto)
apply (erule (1) Least-Suc [THEN ssubst])
apply (blast)
done

lemma ex-least-nat-le: \[ \neg P(0) \] \[ \Rightarrow \] \[ P(n::nat) \] \[ \Rightarrow \] \[ \exists k \leq n. (\forall i < k. \neg P i) \] \& \[ P(k) \]
apply (cases n)
apply blast
apply (erule exE)
apply (rename-tac k1)
apply (auto simp add: less-eq-Suc-le)
done

lemma ex-least-nat-less: \[ \neg P(0) \] \[ \Rightarrow \] \[ P(n::nat) \] \[ \Rightarrow \] \[ \exists k < n. (\forall i \leq k. \neg P i) \] \& \[ P(k+1) \]
unfolding One-nat-def
apply (cases n)
apply blast
apply (frule (1) ex-least-nat-le)
apply (erule exE)
apply simp
apply (rename-tac k1)
apply (rule_tac x=k1 in exI)
apply (auto simp add: less-eq-Suc-le)
done

lemma nat-less-induct:
assumes \[ \forall n. \forall m::nat. m < n \] \[ \Rightarrow \] \[ P m \] \[ \Rightarrow \] \[ P n \]
shows \[ P n \]
using assms less-induct by blast

lemma measure-induct-rule [case-names less]:
fixes f :: 'a \Rightarrow nat
assumes step: \( \forall x. (\forall y. f y < f x \Rightarrow P y) \Rightarrow P x \)
shows \[ P a \]
by (induct m\equiv f a arbitrary; a rule: less-induct) (auto intro: step)
old style induction rules:

**Lemma measure-induct:**
```plaintext
fixes f :: 'a ⇒ nat
shows (∀ x. ∀ y. f y < f x → P y ⇒ P x) ⇒ P a
by (rule measure-induct-rule [of f P a]) iprove
```

**Lemma full-nat-induct:**
```plaintext
assumes step: (!n. (ALL m. Suc m <= n --- > P m) ==> P n)
shows P n
by (rule less-induct) (auto intro: step simp:le-simps)
```

An induction rule for establishing binary relations

**Lemma less-Suc-induct:**
```plaintext
assumes less: i < j
and step: (!i. P i (Suc i))
and trans: (!i j k. i < j ==> j < k ==> P i j ==> P j k ==> P i k)
shows P i j
proof –
from less obtain k where j: j = Suc (i + k) by (auto dest: less-imp-Suc-add)
have P i (Suc (i + k))
proof (induct k)
case 0
  show ?case by (simp add: step)
next
case (Suc k)
  have 0 + i < Suc k + i by (rule add-less-mono1) simp
  hence i < Suc (i + k) by (simp add: add.commute)
  from trans[OF this lessI Suc step]
  show ?case by simp
qed
thus P i j by (simp add: j)
qed
```

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. \(P(n)\) is true for all \(n \in \mathbb{N}\) if

- case “0”: given \(n = 0\) prove \(P(n)\),
- case “smaller”: given \(n > 0\) and \(\neg P(n)\) prove there exists a smaller integer \(m\) such that \(\neg P(m)\).

A compact version without explicit base case:

**Lemma infinite-descent:**
```plaintext
[ !n::nat. \(\neg P n \Rightarrow \exists m < n. \neg P m \) ] \Rightarrow P n
by (induct n rule: less-induct) auto
```

**Lemma infinite-descent0 [case-names 0 smaller]:**
```plaintext
[ P 0; !n. n>0 \Rightarrow \neg P n \Rightarrow (\exists m::nat. m < n \land \neg P m) ] \Rightarrow P n
```
by \textit{(rule infinite-descent) (case-tac \texttt{n>0}, \texttt{auto})}

Infinite descent using a mapping to \(\mathbb{N}\): \(P(x)\) is true for all \(x \in D\) if there exists a \(V : D \rightarrow \mathbb{N}\) and

- case “0”: given \(V(x) = 0\) prove \(P(x)\),
- case “smaller”: given \(V(x) > 0\) and \(\neg P(x)\) prove there exists a \(y \in D\) such that \(V(y) < V(x)\) and \(\neg P(y)\).

NB: the proof also shows how to use the previous lemma.

\textbf{corollary} \textit{infinite-descent0-measure [case-names 0 smaller]:}
assumes \(A0\): \(\forall x. \ V x = (0::\text{nat}) \implies P x\)
and \(A1\): \(\forall x. \ V x > 0 \implies \neg P x \implies (\exists y. \ V y < V x \land \neg P y)\)
shows \(P x\)
proof –
obtain \(n\) where \(n = V x\) by \texttt{auto}
moreover have \(\forall x. \ V x = n \implies P x\)
proof \((\text{induct } n \text{ rule: } \text{infinite-descent0})\)
case 0 — i.e. \(V(x) = 0\)
with \(A0\) show \(P x\) by \texttt{auto}
next — now \(n > 0\) and \(P(x)\) does not hold for some \(x\) with \(V(x) = n\)
case (smaller \(n\))
then obtain \(x\) where \(\forall x. \ V x = n \land V x > 0 \land \neg P x\) by \texttt{auto}
with \(A1\) obtain \(y\) where \(V y < V x \land \neg P y\) by \texttt{auto}
with \(\forall x\) obtain \(m\) where \(m = V y \land m < n \land \neg P y\) by \texttt{auto}
then show \(?\text{case}\) by \texttt{auto}
qed
ultimately show \(P x\) by \texttt{auto}
qed

Again, without explicit base case:

\textbf{lemma} \textit{infinite-descent-measure}:
assumes \(\forall x. \ \neg P x \implies \exists y. \ (V::\text{a=>nat})\ y < V x \land \neg P y\) shows \(P x\)
proof –
from \(\text{assms}\) obtain \(n\) where \(n = V x\) by \texttt{auto}
moreover have \(\forall x. \ V x = n \implies P x\)
proof \((\text{induct } n \text{ rule: } \text{infinite-descent}, \texttt{auto})\)
fix \(x\) assume \(\neg P x\)
with \(\text{assms}\) show \(\exists m < V x. \ \exists y. \ V y = m \land \neg P y\) by \texttt{auto}
qed
ultimately show \(P x\) by \texttt{auto}
qed

A [clumsy] way of lifting \(<\) monotonicity to \(\le\) monotonicity

\textbf{lemma} \textit{less-mono-imp-le-mono}:
\[(\forall i j::\text{nat}. \ i < j \implies f i < f j; \ i \le j \implies f i \le ((f j)::\text{nat})]\]
by \textit{(simp add: order-le-less) (blast)}
THEORY "Nat"

non-strict, in 1st argument
lemma add-le-mono1: \(i \leq j \Rightarrow i + k \leq j + (k::nat)\)
by (rule add-right-mono)

non-strict, in both arguments
lemma add-le-mono: \([ i \leq j; \ k \leq l \] \Rightarrow i + k \leq j + (l::nat)\)
by (rule add-right-mono)

lemma le-add2: \(n \leq ((m + n)::nat)\)
by (insert add-right-mono [of 0 m n], simp)

lemma le-add1: \(n \leq ((n + m)::nat)\)
by (simp add: add.commute, rule le-add2)

lemma less-add-Suc1: \(i < Suc (i + m)\)
by (rule le-less-trans, rule le-add1, rule lessI)

lemma less-add-Suc2: \(i < Suc (m + i)\)
by (rule le-less-trans, rule le-add2, rule lessI)

lemma less-iff-Suc-add: \((m < n) = (\exists k. n = Suc (m + k))\)
by (iprover intro!: less-imp-Suc-add)

lemma trans-le-add1: \((i::nat) \leq j \Rightarrow i \leq j + m\)
by (rule le-trans, assumption, rule le-add1)

lemma trans-le-add2: \((i::nat) \leq j \Rightarrow i \leq m + j\)
by (rule le-trans, assumption, rule le-add2)

lemma trans-less-add1: \((i::nat) < j \Rightarrow i < j + m\)
by (rule less-le-trans, assumption, rule le-add1)

lemma trans-less-add2: \((i::nat) < j \Rightarrow i < m + j\)
by (rule less-le-trans, assumption, rule le-add2)

lemma add-lessD1: \(i + j < (k::nat) \Rightarrow i < k\)
apply (rule le-less-trans [of - i+j])
apply (simp-all add: le-add1)
done

lemma not-add-less1 [iff]: \(\neg (i + j < (i::nat))\)
apply (rule notI)
apply (drule add-lessD1)
apply (erule less-irrefl [THEN notE])
done

lemma not-add-less2 [iff]: \(\neg (j + i < (i::nat))\)
by (simp add: add.commute)
lemma add-leD1: \( m + k \leq n \implies m \leq (n :: \text{nat}) \)
apply (rule order-trans [of \(-m+k\)])
apply (simp-all add: le-add1)
done

lemma add-leD2: \( m + k \leq n \implies k \leq (n :: \text{nat}) \)
apply (simp add: add.commute)
apply (erule add-leD1)
done

lemma add-leE: \( m :: \text{nat} \) + \( k \leq n \implies \neg (m \leq n \implies k \leq n) \implies R \implies R \)
by (blast dest: add-leD1 add-leD2)

needs !! \( k \) for ac-simps to work

lemma less-add-eq-less: \( !k :: \text{nat} \). \( k < l \implies m + l = k + n \implies m < n \)
by (force simp del: add-Suc-right
simps add: less-iff-Suc-add add-Suc-right [symmetric] ac-simps)

16.4.8 More results about difference

Addition is the inverse of subtraction: if \( n \leq m \) then \( n + (m - n) = m \).

lemma add-diff-inverse: \( \sim m < n \implies n + (m - n) = (m :: \text{nat}) \)
by (induct m n rule: diff-induct) simp-all

lemma le-add-diff-inverse [simp]: \( n \leq m \implies n + (m - n) = (m :: \text{nat}) \)
by (simp add: add-diff-inverse linorder-not-less)

lemma le-add-diff-inverse2 [simp]: \( n \leq m \implies (m - n) + n = (m :: \text{nat}) \)
by (simp add: add.commute)

lemma Suc-diff-le: \( n \leq m \implies \text{Suc} \ m - n = \text{Suc} \ (m - n) \)
by (induct m n rule: diff-induct) simp-all

lemma diff-less-Suc: \( m - n < \text{Suc} \ m \)
apply (induct m n rule: diff-induct)
apply (erule-tac [3] less-SucE)
apply (simp-all add: less-Suc-eq)
done

lemma diff-le-self [simp]: \( m - n \leq (m :: \text{nat}) \)
by (induct m n rule: diff-induct) (simp-all add: le-SucI)

lemma le-iff-add: \( (m :: \text{nat}) \leq n \implies (\exists k. n = m + k) \)
by (auto simp: le-add1 dest!: le-add-diff-inverse sym [of \(-n\)])

instance nat :: ordered-cancel-comm-monoid-diff
proof
  show \( \forall m n :: \text{nat}. m \leq n \iff (\exists q. n = m + q) \) by (fact le-iff-add)
qed
lemma less-imp-diff-less: \((j::nat) < k \Longrightarrow j - n < k\)
by (rule le-less-trans, rule diff-le-self)

lemma diff-Suc-less [simp]: \(0 < n \Longrightarrow n - \text{Suc} \ i < n\)
by (cases n) (auto simp add: le-simps)

lemma diff-add-assoc: \(k \leq (j::nat) \Longrightarrow (i + j) - k = i + (j - k)\)
by (induct j k rule: diff-induct) simp-all

lemma diff-add-assoc2: \(k \leq (j::nat) \Longrightarrow (j + i) - k = (j - k) + i\)
by (simp add: add.commute diff-add-assoc)

lemma le-imp-diff-is-add: \(i \leq (j::nat) \Longrightarrow (j - i) = k \Longrightarrow j = k + i\)
by (auto simp add: diff-add-inverse2)

lemma le-imp-diff-is-add': \(0 < n - (m::nat) \Longrightarrow (m \leq n)\)
by (simp add: iffD2 dest: add-eq-self-zero)

lemma zero-less-diff [simp]: \(0 < n - m\)
by (induct m n rule: diff-induct)

lemma less-imp-add-positive:
  assumes \(i < j\)
  shows \(\exists k::nat. \ 0 < k \& i + k = j\)
proof
  from assms show \(0 < j - i \& i + (j - i) = j\)
  by (simp add: order-less-imp-le)
qed

a nice rewrite for bounded subtraction

lemma nat-minus-add-max:
  fixes \(n \ m::nat\)
  shows \(n - m + m = \text{max} \ n \ m\)
  by (simp add: max-def le-less dest: add-eq-self-zero)

lemma nat-diff-split:
  \(P(a - b::nat) = ((a < b \Longrightarrow P \ 0) \& \ (\text{ALL} \ d. \ a = b + d \longrightarrow P \ d))\)
  — elimination of \(-\) on \(\text{nat}\)
by (cases \(a < b\))
(auto simp add: diff-is-0-eq [THEN iffD2] diff-add-inverse
  not-less le-less dest!: add-eq-self-zero add-eq-self-zero[OF sym])

lemma nat-diff-split-asn:
  \(P(a - b::nat) = (\sim (a < b \& \sim P \ 0) \ | \ (EX \ d. \ a = b + d \& \sim P \ d)))\)
  — elimination of \(-\) on \(\text{nat}\) in assumptions
by (auto split: nat-diff-split)

lemma Suc-pred': 0 < n ==> n = Suc(n - 1)
  by simp

lemma add-eq-if: (m::nat) + n = (if m=0 then n else Suc ((m - 1) + n))
  unfolding One-nat-def by (cases m) simp-all

lemma mult-eq-if: (m::nat) * n = (if m=0 then 0 else Suc ((m - 1) * n))
  unfolding One-nat-def by (cases m) simp-all

lemma Suc-diff-eq-diff-pred: 0 < n == Suc m - n = m - (n - 1)
  unfolding One-nat-def by (cases n) simp-all

lemma diff-Suc-eq-diff-pred: m - Suc n = (m - 1) - n
  unfolding One-nat-def by (cases m) simp-all

lemma Let-Suc [simp]: Let (Suc n) f == f (Suc n)
  by (fact Let-def)

16.4.9 Monotonicity of Multiplication

lemma mult-le-mono1: i ≤ (j::nat) ==> i * k ≤ j * k
  by (simp add: mult-right-mono)

lemma mult-le-mono2: i ≤ (j::nat) ==> k * i ≤ k * j
  by (simp add: mult-left-mono)

≤ monotonicity, BOTH arguments

lemma mult-le-mono: i ≤ (j::nat) ==> k ≤ l ==> i * k ≤ j * l
  by (simp add: mult-mono)

lemma mult-less-mono1: (i::nat) < j ==> 0 < k ==> i * k < j * k
  by (simp add: mult-strict-right-mono)

Differs from the standard zero-less-mult-iff in that there are no negative numbers.

lemma nat-0-less-mult-iff [simp]: (0 < (m::nat) * n) = (0 < m & 0 < n)
  apply (induct m)
  apply simp
  apply (case-tac n)
  apply simp-all
  done

lemma one-le-mult-iff [simp]: (Suc 0 ≤ m * n) = (Suc 0 ≤ m & Suc 0 ≤ n)
  apply (induct m)
  apply simp
  apply (case-tac n)
  apply simp-all
done

lemma mult-less-cancel2 [simp]: \((m::nat) \ast k < n \ast k) = (0 < k & m < n)\)
  apply (safe intro: mult-less-mono1)
  apply (cases k, auto)
  apply (simp del: le-0-eq add linorder-not-le [symmetric])
  apply (blast intro: mult-le-mono1)
  done

lemma mult-less-cancel1 [simp]: \(k \ast (m::nat) < k \ast n) = (0 < k -- m < n)\)
  by (simp add: mult.commute [of k])

lemma mult-le-cancel1 [simp]: \(k \ast (m::nat) \leq k \ast n) = (0 < k -- m \leq n)\)
  by (simp add: linorder-not-less [symmetric], auto)

lemma mult-le-cancel2 [simp]: \((m::nat) \ast k \leq n \ast k) = (0 < k -- m \leq n)\)
  by (simp add: linorder-not-less [symmetric], auto)

lemma Suc-mult-less-cancel1: \((Suc k \ast m < Suc k \ast n) = (m < n)\)
  by (subst mult-less-cancel1)

lemma Suc-mult-le-cancel1: \((Suc k \ast m \leq Suc k \ast n) = (m \leq n)\)
  by (subst mult-le-cancel1)

lemma le-square: \(m \leq m \ast (m::nat)\)
  by (cases m) (auto intro: le-add1)

lemma le-cube: \((m::nat) \leq m \ast (m \ast m)\)
  by (cases m) (auto intro: le-add1)

Lemma for gcd

lemma mult-eq-self-implies-10: \((m::nat) = m \ast n =\> n = 1 | m = 0\)
  apply (drule sym)
  apply (rule disjCI)
  apply (rule nat-less-cases, erule-tac [2] -)
  apply (drule-tac [2] mult-less-mono2)
  apply (auto)
  done

lemma mono-times-nat:
  fixes n :: nat
  assumes n > 0
  shows mono (times n)
proof
  fix m q :: nat
  assume m \leq q
  with assms show n \ast m \leq n \ast q by simp
qed

the lattice order on nat
instantiation \textit{nat} :: distrib-lattice

begin

definition\lbrack\inf :: nat \Rightarrow nat \Rightarrow nat\rbrack = \text{\textit{min}}

definition\lbrack\sup :: nat \Rightarrow nat \Rightarrow nat\rbrack = \text{\textit{max}}

instance by intro-classes
\lbrack\text{\textit{auto simp add: inf-nat-def sup-nat-def max-def not-le min-def}}\rbrack
\lbrack\text{\textit{intro: order-less-imp-le antisym elim\!: order-trans order-less-trans}}\rbrack

end

16.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

consts \textit{compow} :: nat \Rightarrow \textit{'}a \Rightarrow \textit{'}a

abbreviation \textit{compower} :: \textit{'}a \Rightarrow nat \Rightarrow \textit{'}a \ (\text{\textit{infixr \text{\textquotedblleft} ^\_ 80\text{\textquotedblright}) where\lbrack f ^\_ n \equiv \text{\textit{compow} n f}}\rbrack

lemma \textit{funpow-Suc-right}:\lbrack f ^\_ \text{Suc} n = f ^\_ n \circ f\rbrack

proof (induct n)
\lbrack\text{\textit{case \textit{\theta} then show \textit{?case by simp}}\rbrack
\lbrack\text{\textit{next}}\rbrack
\lbrack\text{\textit{fix n}}\rbrack
\lbrack\text{\textit{assume f ^\_ \text{Suc} n = f ^\_ n \circ f}}\rbrack
then show \( f \circ \text{Suc} (\text{Suc} n) = f \circ \text{Suc} n \circ f \)
by (simp add: o-assoc)

qed

lemmas funpow-simps-right = funpow.simps(1) funpow-Suc-right

for code generation

definition funpow :: \( \text{nat} \to (\text{'a} \Rightarrow \text{'a}) \Rightarrow \text{'a} \Rightarrow \text{'a} \)
where
funpow-code-def [code-abbrev]: funpow = compow

hide-const (open) funpow

lemma funpow-add:
\( f \circ (m + n) = f \circ m \circ f \circ n \)
by (induct m) simp-all

lemma funpow-mult:
fixes \( f :: \text{'a} \Rightarrow \text{'a} \)
shows \( (f \circ m)^n = f \circ (m \times n) \)
by (induct n) (simp-all add: funpow-add)

lemma funpow-swap1:
\( f ((f \circ n) x) = (f \circ n) (f x) \)
proof -
  have \( f ((f \circ n) x) = (f \circ (n + 1)) x \) by simp
  also have \( \ldots = (f \circ n \circ f \circ 1) x \) by (simp only: funpow-add)
  also have \( \ldots = (f \circ n) (f x) \) by simp
  finally show \( ?\text{thesis} \).

qed

lemma comp-funpow:
fixes \( f :: \text{'a} \Rightarrow \text{'a} \)
shows \( \text{comp} f \circ n = \text{comp} (f \circ n) \)
by (induct n) simp-all

lemma Suc-funpow[simp]: Suc \( \circ n = (\text{op} + n) \)
by (induct n) simp-all

lemma id-funpow[simp]: id \( \circ n = id \)
by (induct n) simp-all

16.6 Kleene iteration

lemma Kleene-iter-lfpf:
assumes mono f and f p ≤ p shows \((f^\bot k) (\bot::'a::order-bot) \leq p\)

proof (induction k)
  case 0 show ?case by simp

next
  case Suc
  from monoD[of assms(1) Suc] assms(2)
  show ?case by simp

qed

lemma lfp-Kleene-iter: assumes mono f and \((f^\bot \text{Suc} k) \bot = (f^\bot k) \bot\)
shows lfp f = \((f^\bot k) \bot\)

proof (rule antisym)
  show \(lfp f \leq (f^\bot k) \bot\)
  proof (rule lfp-lowerbound)
    show \(f ((f^\bot k) \bot) \leq (f^\bot k) \bot\) using assms(2) by simp
  qed

next
  show \((f^\bot k) \bot \leq lfp f\)
    using Kleene-iter-lfpl[of assms(1)] lfp-unfold[of assms(1)] by simp

qed

16.7 Embedding of the Naturals into any semiring-1: of-nat

code begin

definition of-nat :: nat ⇒ 'a where
  of-nat n = \((\text{plus} \ 1 \ "^ n) \ 0\)

lemma of-nat-simps [simp]:
  shows of-nat-0: of-nat 0 = 0
    and of-nat-Suc: of-nat \((\text{Suc} \ m)\) = 1 + of-nat \(m\)
  by (simp-all add: of-nat-def)

lemma of-nat-1 [simp]: of-nat 1 = 1
  by (simp add: of-nat-def)

lemma of-nat-add [simp]: of-nat \((m + n)\) = of-nat \(m\) + of-nat \(n\)
  by (induct \(m\)) (simp-all add: ac-simps)

lemma of-nat-mult: of-nat \((m \ * \ n)\) = of-nat \(m\) \* of-nat \(n\)
  by (induct \(m\)) (simp-all add: ac-simps distrib-right)

primrec of-nat-aux :: \((\ 'a \Rightarrow \ 'a) \Rightarrow \text{nat} \Rightarrow \ 'a \Rightarrow \ 'a\) where
  of-nat-aux inc 0 i = i
  | of-nat-aux inc \((\text{Suc} \ n)\) i = of-nat-aux inc \(n\) (inc \(i\)) — tail recursive

lemma of-nat-code:
  of-nat n = of-nat-aux \((\lambda i. \ i + 1)\) \(n\) \(0\)
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
  have \( \bigwedge i. \text{of-nat-aux} (\lambda i. i + 1) n (i + 1) = \text{of-nat-aux} (\lambda i. i + 1) n i + 1 \)
  by (induct n) simp-all
  from this [of 0] have \( \text{of-nat-aux} (\lambda i. i + 1) n 1 = \text{of-nat-aux} (\lambda i. i + 1) n 0 + 1 \)
  by simp
  with Suc show ?case by (simp add: add.commute)
qed

declare of-nat-code [code]

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class semiring-char-0 = semiring-1 +
  assumes inj-of-nat: inj of-nat
begin

lemma of-nat-eq-iff [simp]: of-nat m = of-nat n \( \iff \) m = n
  by (auto intro: inj-of-nat injD)

Special cases where either operand is zero

lemma of-nat-0-eq-iff [simp]: 0 = of-nat n \( \iff \) 0 = n
  by (fact of-nat-eq-iff [of 0 n, unfolded of-nat])

lemma of-nat-eq-0-iff [simp]: of-nat m = 0 \( \iff \) m = 0
  by (fact of-nat-eq-0-iff [of m 0, unfolded of-nat])

end

context linordered-semidom
begin

lemma of-nat-0-le-iff [simp]: 0 \( \leq \) of-nat n
  by (induct n) simp-all

lemma of-nat-less-0-iff [simp]: \( \neg \) of-nat m < 0
  by (simp add: not-less)

lemma of-nat-less-iff [simp]: of-nat m < of-nat n \( \iff \) m < n
  by (induct m n rule: diff-induct, simp-all add: add-pos-nonneg)

lemma of-nat-le-iff [simp]: of-nat m \( \leq \) of-nat n \( \iff \) m \( \leq \) n
  by (simp add: not-less [symmetric] linorder-not-less [symmetric])
lemma less-imp-of-nat-less: \( m < n \implies \text{of-nat } m < \text{of-nat } n \)
by simp

lemma of-nat-less-imp-less: \( \text{of-nat } m < \text{of-nat } n \implies m < n \)
by simp

Every linordered-semidom has characteristic zero.

subclass semiring-char-0 proof
qed (auto intro!: injI simp add: eq-iff)

Special cases where either operand is zero

lemma of-nat-le-0-iff [simp]: \( \text{of-nat } m \leq 0 \iff m = 0 \)
by (rule of-nat-le-iff [of \- 0, simplified])

lemma of-nat-0-less-iff [simp]: \( 0 < \text{of-nat } n \iff 0 < n \)
by (rule of-nat-less-iff [of 0, simplified])

end

class ring_1

begin

lemma of-nat-diff: \( n \leq m \implies \text{of-nat } (m - n) = \text{of-nat } m - \text{of-nat } n \)
by (simp add: algebra-simps of-nat-add [symmetric])

end

class linordered_idom

begin

lemma abs-of-nat [simp]: \( |\text{of-nat } n| = \text{of-nat } n \)
unfolding abs-if by auto

end

lemma of-nat-id [simp]: \( \text{of-nat } n = n \)
by (induct n) simp-all

lemma of-nat-eq-id [simp]: \( \text{of-nat} = \text{id} \)
by (auto simp add: fun-eq-iff)

16.8 The Set of Natural Numbers

begin

definition Nats :: 'a set where
Nats = range of-nat
notation (xsymbols)
Nats (\mathbb{N})

lemma of-nat-in-Nats [simp]: of-nat n \in \mathbb{N}
by (simp add: Nats-def)

lemma Nats-0 [simp]: 0 \in \mathbb{N}
apply (simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-0 [symmetric])
done

lemma Nats-1 [simp]: 1 \in \mathbb{N}
apply (simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-1 [symmetric])
done

lemma Nats-add [simp]: a \in \mathbb{N} \Rightarrow b \in \mathbb{N} \Rightarrow a + b \in \mathbb{N}
apply (auto simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-add [symmetric])
done

lemma Nats-mult [simp]: a \in \mathbb{N} \Rightarrow b \in \mathbb{N} \Rightarrow a \ast b \in \mathbb{N}
apply (auto simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-mult [symmetric])
done

lemma Nats-cases [cases set: Nats]:
assumes x \in \mathbb{N}
obtains (of-nat) n where x = of-nat n
unfolding Nats-def
proof -
from (x \in \mathbb{N}) have x \in range of-nat unfolding Nats-def .
then obtain n where x = of-nat n ..
then show thesis ..
qed

lemma Nats-induct [case-names of-nat, induct set: Nats]:
x \in \mathbb{N} \Rightarrow (\forall n. P (of-nat n)) \Rightarrow P x
by (rule Nats-cases) auto

end
16.9 Further Arithmetic Facts Concerning the Natural Numbers

lemma subst-equals:
assumes 1: \( t = s \) and 2: \( u = t \)
shows \( u = s \)
using 2 1 by (rule trans)

setup Arith-Data.setup

ML-file Tools/nat-arith.ML

simproc-setup nateq-cancel-sums
\[
\langle \langle \text{fn } \phi \text{ => try o Nat-Arith.cancel-eq-conv} \rangle \rangle
\]

simproc-setup natless-cancel-sums
\[
\langle \langle \text{fn } \phi \text{ => try o Nat-Arith.cancel-less-conv} \rangle \rangle
\]

simproc-setup natle-cancel-sums
\[
\langle \langle \text{fn } \phi \text{ => try o Nat-Arith.cancel-le-conv} \rangle \rangle
\]

simproc-setup natdiff-cancel-sums
\[
\langle \langle \text{fn } \phi \text{ => try o Nat-Arith.cancel-diff-conv} \rangle \rangle
\]

ML-file Tools/lin-arith.ML
setup \langle \langle \text{Lin-Arith.global-setup} \rangle \rangle
declaration \langle \langle \text{K Lin-Arith.setup} \rangle \rangle

simproc-setup fast-arith-nat \langle \langle \text{fn } \phi \text{ => try o Lin-Arith.simproc ss (term-of ct)} \rangle \rangle

lemmas [arith-split] = nat-diff-split split-min split-max

context order
begin

lemma lift-Suc-mono-le:
assumes mono: \( \forall n. f n \leq f (\text{Suc } n) \) and \( n \leq n' \)
shows \( f n \leq f n' \)
proof (cases \( n < n' \))
case True
then show \( \vdash \phi \)
by (induct \( n \) \( n' \) rule: \text{less-Suc-induct \{consumes 1\}}) (auto intro: mono)
qed (insert \( n \leq n', \text{auto} \) — trivial for \( n = n' \)}
lemma lift-Suc-antimono-le:
  assumes mono: \( \forall n. f n \leq f (Suc n) \) and \( n \leq n' \)
  shows \( f n \leq f n' \)
proof (cases \( n < n' \))
  case True
  then show \(?thesis\)
  by (induct \( n \) \( n' \) rule: less-Suc-induct [consumes 1]) (auto intro: mono)
qed (insert \( (n \leq n', auto) \) — trivial for \( n = n' \))

lemma lift-Suc-mono-less:
  assumes mono: \( \forall n. f n < f (Suc n) \) and \( n < n' \)
  shows \( f n < f n' \)
using \( (n < n') \)
by (induct \( n \) \( n' \) rule: less-Suc-induct [consumes 1]) (auto intro: mono)

lemma lift-Suc-mono-less-iff:
\[ (\forall n. f n < f (Suc n)) \implies f n < f m \iff n < m \]
by (blast intro: less-asym' lift-Suc-mono-less [of f]
  dest: linorder-not-less[THEN iffD1] le-eq-less-or-eq [THEN iffD1])
end

lemma mono-iff-le-Suc:
mono f \iff (\forall n. f n \leq f (Suc n))
unfolding mono-def by (auto intro: lift-Suc-mono-le [of f])

lemma antimono-iff-le-Suc:
antimono f \iff (\forall n. f (Suc n) \leq f n)
unfolding antimono-def by (auto intro: lift-Suc-antimono-le [of f])

lemma mono-nat-linear-lb:
fixes f :: nat \Rightarrow nat
assumes \( \forall m n. m < n \implies f m < f n \)
shows \( f m + k \leq f (m + k) \)
proof (induct k)
  case 0 then show ?case by simp
next
  case (Suc k)
  then have Suc \( (f m + k) \leq Suc (f (m + k)) \) by simp
  also from assms \[ of m + k \ Suc (m + k) \] have Suc \( (f (m + k)) \leq f (Suc (m + k)) \)
    by (simp add: Suc-le-eq)
  finally show ?case by simp
qed

Subtraction laws, mostly by Clemens Ballarin

lemma diff-less-mono: \[ \| a < (b::nat); c \leq a \| \implies a - c < b - c \]
by arith
lemma less-diff-conv: \((i < j - k) = (i + k < (j::nat))\)
by arith

lemma less-diff-conv2:
  fixes \(j \ k \ i :: nat\)
  assumes \(k \leq j\)
  shows \((j - k < i \leftrightarrow j < i + k)\)
  using assms by arith

lemma le-diff-conv: \((j - k \leq (i::nat)) = (j \leq i + k)\)
by arith

lemma le-diff-conv2: \(k \leq j \Rightarrow (i \leq j - k) = (i + k \leq (j::nat))\)
by (fact le-diff-conv2) — FIXME delete

lemma diff-diff-cancel [simp]: \(i \leq (n::nat) \Rightarrow n - (n - i) = i\)
by arith

lemma le-add-diff: \(k \leq (n::nat) \Rightarrow m \leq n + m - k\)
by (fact le-add-diff) — FIXME delete

lemma diff-less [simp]: \(!!m::nat. \[ \theta < n; \theta < m \] \Rightarrow m - n < m\)
by arith

Simplification of relational expressions involving subtraction

lemma diff-diff-eq: \[ \| k \leq m; k \leq (n::nat) \| \Rightarrow (m - k = n - k) = (m - n)\]
by (simp split add: nat-diff-split)

hide-fact (open) diff-diff-eq

lemma eq-diff-iff: \[ \| k \leq m; k \leq (n::nat) \| \Rightarrow (m - k = n - k) = (m = n)\]
by (auto split add: nat-diff-split)

lemma less-diff-iff: \[ \| k \leq m; k \leq (n::nat) \| \Rightarrow (m - k < n - k) = (m < n)\]
by (auto split add: nat-diff-split)

lemma le-diff-iff: \[ \| k \leq m; k \leq (n::nat) \| \Rightarrow (m - k \leq n - k) = (m \leq n)\]
by (auto split add: nat-diff-split)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma diff-le-mono: \(m \leq (n::nat) \Rightarrow (m - l) \leq (n - l)\)
by (simp split add: nat-diff-split)

lemma diff-le-mono2: \(m \leq (n::nat) \Rightarrow (l - n) \leq (l - m)\)
by (simp split add: nat-diff-split)

lemma diff-less-mono2: \[ m < (n::nat); m < l \| \Rightarrow (l - n) < (l - m)\]
THEORY “Nat”

by (simp split add: nat-diff-split)

lemma diffs0-imp-equal: !m::nat. [| m-n = 0; n-m = 0 |] ==> m=n
by (simp split add: nat-diff-split)

lemma min-diff: min (m - (i::nat)) (n-i) = min m n - i
by auto

lemma inj-on-diff-nat:
assumes k-le-n: \forall n \in N. k \leq (n::nat)
shows inj-on (λn. n - k) N
proof (rule inj-onI)
  fix x y
  assume a: x \in N y \in N x - k = y - k
  with k-le-n have x - k + k = y - k + k by auto
  with a k-le-n show x = y by auto
qed

Rewriting to pull differences out

lemma diff-diff-right [simp]: k \leq j --- i - (j - k) = i + (k::nat) - j
by arith

lemma diff-Suc-diff-eq1 [simp]: k \leq j ==> m - Suc (j - k) = m + k - Suc j
by arith

lemma diff-Suc-diff-eq2 [simp]: k \leq j ==> Suc (j - k) - m = Suc j - (k + m)
by arith

lemma Suc-diff-Suc: n < m \implies Suc (m - Suc n) = m - n
by simp

lemmas add-diff-assoc = diff-add-assoc [symmetric]
lemmas add-diff-assoc2 = diff-add-assoc2[symmetric]
declare diff-diff-left [simp] add-diff-assoc [simp] add-diff-assoc2[simp]

At present we prove no analogue of not-less-Least or Least-Suc, since there appears to be no need.

Lemmas for ex/Factorization

lemma one-less-mult: [| Suc 0 < n; Suc 0 < m |] ==> Suc 0 < m*n
by (cases m) auto

lemma n-less-m-mult-n: [| Suc 0 < n; Suc 0 < m |] ==> n<m*n
by (cases m) auto

lemma n-less-n-mult-m: [| Suc 0 < n; Suc 0 < m |] ==> n<n*m
by (cases m) auto

Specialized induction principles that work ”backwards”:
lemma inc-induct[consumes 1, case-names base step]:
assumes less: \( i \leq j \)
assumes base: \( P j \)
assumes step: \( \forall n. i \leq n \implies n < j \implies P(n) \implies P n \)
shows \( P i \)
using less step
proof (induct \( d \equiv j - i \) arbitrary: \( i \))
case (0 \( i \))
hence \( i = j \) by simp
with base show ?case by simp
next
case (Suc \( d \) \( n \))
hence \( n \leq n < j \) \( P (Suc n) \)
by simp-all
then show \( P n \) by fact
qed

lemma strict-inc-induct[consumes 1, case-names base step]:
assumes less: \( i < j \)
assumes base: \( !i. j = Suc i ==> P i \)
assumes step: \( !i. [i < j; P (Suc i)] \implies P i \)
shows \( P i \)
using less
proof (induct \( d \equiv j - i - 1 \) arbitrary: \( i \))
case (0 \( i \))
with \( \langle i < j \rangle \) have \( j = Suc i \) by simp
with base show ?case by simp
next
case (Suc \( d \) \( i \))
hence \( i < j \) \( P (Suc i) \)
by simp-all
thus \( P i \) by (rule step)
qed

lemma zero-induct-lemma: \( P k \implies (\forall n. P (Suc n) \implies P n) \implies P (k - i) \)
using inc-induct[of \( k - i \) \( P \), simplified] by blast

lemma zero-induct: \( P k \implies (\forall n. P (Suc n) \implies P n) \implies P 0 \)
using inc-induct[of 0 \( P \)] by blast

Further induction rule similar to \( [\forall i \leq \forall j; \forall P \forall j; \forall n. [\forall i \leq n; n < \forall j; \forall P (Suc n)] \implies P n] \implies P 0 \)

lemma dec-induct[consumes 1, case-names base step]:
\( i \leq j \implies P i \implies (\forall n. i \leq n \implies n < j \implies P n \implies P (Suc n)) \implies P j \)
by (induct \( j \) arbitrary: \( i \)) (auto simp: le-Suc-eq)

16.10 The divides relation on nat

lemma dvd-1-left [iff]: \( Suc 0 \) dvd \( k \)
unfolding dvd-def by simp

lemma dvd-1-iff-1 [simp]: \((m \text{ dvd } \text{Suc } 0) \iff (m = \text{Suc } 0)\)
by (simp add: dvd-def)

lemma nat-dvd-1-iff-1 [simp]: \(m \text{ dvd } (1::nat) \iff m = 1\)
by (simp add: dvd-def)

lemma dvd-antisym: \([| m \text{ dvd } n; n \text{ dvd } m |] \implies m = (n::nat)\)
unfolding dvd-def
by (force dest: mult-eq-self-implies-10 simp add: mult.assoc)

op dvd is a partial order

interpretation dvd: order op dvd \(\forall m \cdot n \text{ :: nat. } n \text{ dvd } m \land \neg m \text{ dvd } n\)
proof qed (auto intro: dvd-refl dvd-trans dvd-antisym)

lemma dvd-diff-nat [simp]: \([| k \text{ dvd } m; k \text{ dvd } n |] \implies k \text{ dvd } (m-n ::nat)\)

lemma dvd-diffD: \([| k \text{ dvd } m-n; k \text{ dvd } m; n \leq m |] \implies k \text{ dvd } n::nat\)
applie (erule linorder-not-less THEN dvdI2, THEN add-diff-inverse, THEN subst))
applie (blast intro: dvd-add)
done

lemma dvd-diffD1: \([| k \text{ dvd } m-n; k \text{ dvd } m; n \leq m |] \implies k \text{ dvd } n::nat\)
by (drule_tac m = m in dvd-diff-nat, auto)

lemma dvd-reduce: \((k \text{ dvd } n + k) = (k \text{ dvd } (n::nat))\)
applie (rule iffI)
applie (erule-tac [2] dvd-add)
applie (rule-tac [2] dvd-refl)
applie (subgoal-tac n = (n+k) -k)
prefer 2 applie simp
applie (erule sssubst)
applie (erule dvd-diff-nat)
applie (rule dvd-refl)
done

lemma dvd-mult-cancel: \(!!(k::nat. [| k \text{ dvd } k*n; 0<k |] \implies m \text{ dvd } n)\)
unfolding dvd-def
applie (erule exE)
applie (simp add: ac-simps)
done

lemma dvd-mult-cancel1: \(0\text{<c } m \implies (m*n \text{ dvd } m) = (n = (1::nat))\)
applie auto
applie (subgoal-tac m*n dvd m*1)
apply (drule dvd-mult-cancel, auto)
done

lemma dvd-mult-cancel2: \( \theta < m \implies (n \cdot m \text{ dvd } m) = (n = (1::nat)) \)
apply (subst mult.commute)
apply (erule dvd-mult-cancel1)
done

lemma dvd-imp-le: \([k \text{ dvd } n; \ 0 < n \implies k \leq (n::nat)]\)
by (auto elim!: dvdE) (auto simp add: gr0_conv_Suc)

lemma nat-dvd-not-less:
fixes m n :: nat
shows \( 0 < m \implies m \not< n \implies \neg n \text{ dvd } m \)
by (auto elim!: dvdE) (auto simp add: gr0_conv_Suc)

lemma dvd-plusE:
fixes m n q :: nat
assumes m dvd n + q m dvd n
obtains m dvd q
proof (cases m = 0)
case True with assms that show thesis by simp
next
case False then have m > 0 by simp
from assms obtain r s where n = m \cdot r \and n + q = m \cdot s by (blast elim: dvdE)
then have *: m \cdot r + q = m \cdot s by simp
show thesis proof (cases r \leq s)
case False then have s < r by (simp add: not-le)
with * have m \cdot r + q - m \cdot s = m \cdot s - m \cdot s by simp
then have m \cdot r + q - m \cdot r = 0 by simp
with \( m > 0 \cdot (s < r) \) have m \cdot r - m \cdot s + q = 0 by (unfold less-le-not-le)
auto
then have m \cdot (r - s) + q = 0 by auto
then have m \cdot (r - s) = 0 by simp
then have m = 0 \or r - s = 0 by simp
with \( s < r \) have m = 0 by (simp add: less-le-not-le)
with \( m > 0 \) show thesis by auto
next
case True with * have m \cdot r + q - m \cdot r = m \cdot s - m \cdot r by simp
with \( m > 0 \cdot (r \leq s) \) have m \cdot r - m \cdot r + q = m \cdot s - m \cdot r by simp
then have q = m \cdot (s - r) by (simp add: diff-mult-distrib2)
with assms that show thesis by (auto intro: dvdI)
qed
qed

lemma dvd-plus-eq-right:
fixes m n q :: nat
assumes m dvd n
shows \( m \text{ dvd } n + q \iff m \text{ dvd } q \)

using assms by (auto elim: dvd-plusE)

**lemma** dvd-plus-eq-left:

fixes \( m \ n q :: \text{n} \)

assumes \( m \text{ dvd } q \)

shows \( m \text{ dvd } n + q \iff m \text{ dvd } n \)

using assms by (simp add: dvd-plus-eq-right add.commute [of n])

**lemma** less-eq-dvd-minus:

fixes \( m \ n :: \text{n} \)

assumes \( m \leq n \)

shows \( m \text{ dvd } n \iff m \text{ dvd } n - m \)

proof –

from assms have \( n = m + (n - m) \) by simp

then obtain \( q \) where \( n = m + q \) ..

then show ?thesis by (simp add: dvd-reduce add.commute [of m])

qed

**lemma** dvd-minus-self:

fixes \( m \ n :: \text{n} \)

shows \( m \text{ dvd } n - m \iff n < m \lor m \text{ dvd } n \)

by (cases \( n < m \)) (auto elim!: dvdE simp add: not-less le-imp-diff-is-add)

**lemma** dvd-minus-add:

fixes \( m \ n q r :: \text{n} \)

assumes \( q \leq n \quad q \leq r \cdot m \)

shows \( m \text{ dvd } n - q \iff m \text{ dvd } n + (r \cdot m - q) \)

proof –

have \( m \text{ dvd } n - q \iff m \text{ dvd } r \cdot m + (n - q) \)

by (auto elim: dvd-plusE)

also from assms have \( \ldots \iff m \text{ dvd } r \cdot m + n - q \) by simp

also from assms have \( \ldots \iff m \text{ dvd } (r \cdot m - q) + n \) by simp

also have \( \ldots \iff m \text{ dvd } n + (r \cdot m - q) \) by (simp add: add.commute)

finally show ?thesis .

qed

16.11 aliases

**lemma** nat-mult-1: \((1::\text{n}) \ast n = n\)

by (rule mult-1-left)

**lemma** nat-mult-1-right: \(n \ast (1::\text{n}) = n\)

by (rule mult-1-right)

16.12 size of a datatype value

**class** size =

fixes \( \text{size :: ('a \Rightarrow \text{n})} \) — see further theory Wellfounded
16.13 code module namespace

code-identifier
  code-module Nat \rightarrow (SML) Arith and (OCaml) Arith and (Haskell) Arith

hide-const (open) of-nat-aux

end

17 Finite-Set: Finite sets

theory Finite-Set
imports Product-Type Sum-Type Nat
begin

17.1 Predicate for finite sets

inductive finite :: 'a set \Rightarrow bool
where
  emptyI [simp, intro!]: finite {}
| insertI [simp, intro!]: finite A \Rightarrow finite (insert a A)

simproc-setup finite-Collect (finite (Collect P)) = ⟨⟨ K Set-Comprehension-Pointfree.simproc ⟩⟩

declare [[simproc del: finite-Collect]]

lemma finite-induct [case-names empty insert, induct set: finite]:
— Discharging x \notin F entails extra work.
assumes finite F
assumes P {}
and insert: \forall x F. finite F \Rightarrow x \notin F \Rightarrow P F \Rightarrow P (insert x F)
shows P F
using (finite F)
proof induct
  show P {} by fact
  fix x F assume F: finite F and P: P F
  show P (insert x F)
  proof cases
    assume x \in F
    hence insert x F = F by (rule insert-absorb)
    with P show \?thesis by (simp only:)
  next
    assume x \notin F
    from F this P show \?thesis by (rule insert)
  qed
qed

lemma infinite-finite-induct [case-names infinite empty insert]:
assumes infinite: $\forall A. \neg\text{finite } A \rightarrow P A$
assumes empty: $P \{\}$
assumes insert: $\forall x F. \text{finite } F \rightarrow x \notin F \rightarrow P F \rightarrow P(\text{insert } x F)$
shows $P A$
proof (cases finite $A$)
  case False with infinite show $\text{thesis}$.
next
case True then show $\text{thesis}$ by (induct $A$) (fact empty insert)+
qed

17.1.1 Choice principles

lemma ex-new-if-finite: — does not depend on def of finite at all
  assumes $\neg\text{finite } (\text{UNIV :: 'a set}) \text{ and finite } A$
  shows $\exists a::'a. a \notin A$
proof —
  from assms have $A \neq \text{UNIV}$ by blast
  then show $\text{thesis}$ by blast
qed

A finite choice principle. Does not need the SOME choice operator.

lemma finite-set-choice:
  finite $A \rightarrow \forall x A. \exists y. P x y \rightarrow \exists f. \forall x A. P x (f x)$
proof (induct rule: finite-induct)
case empty then show $\text{case}$ by simp
next
case (insert a $A$)
then obtain $f b$ where $f: \forall x A. P x (f x)$ and $ab: P a b$ by auto
show $\text{case}$ (is $\exists f. ?P f$)
proof
  show $?P(\%x. \text{ if } x = a \text{ then } b \text{ else } f x)$ using $ab$ by auto
qed
qed

17.1.2 Finite sets are the images of initial segments of natural numbers

lemma finite-imp-nat-seg-image-inj-on:
  assumes finite $A$
  shows $\exists (n::\text{nat}) f. A = f \upharpoonright \{i. i < n\} \land \text{inj-on } f \{i. i < n\}$
using assms
proof induct
  case empty
  show $\text{case}$
  proof
    show $\exists f. \{\} = f \upharpoonright \{i::\text{nat. } i < 0\} \land \text{inj-on } f \{i. i < 0\}$ by simp
  qed
next
case (insert a $A$)
have notinA: a \notin A by fact
from insert.hyps obtain n f
  where A = f \{i::nat. i < n\} inj-on f \{i. i < n\} by blast
hence insert a A = f(n:=a) \{i. i < Suc n\}
  inj-on (f(n:=a)) \{i. i < Suc n\} using notinA
  by (auto simp add: image-def Ball-def inj-on-def less-Suc-eq)
thus ?case by blast
qed

lemma nat-seg-image-imp-finite:
  A = f \{i::nat. i < n\} =\Rightarrow finite A
proof (induct n arbitrary: A)
case 0 thus ?case by simp
next
case (Suc n)
let ?B = f \{i. i < n\}
have finB: finite ?B by (rule Suc.hyps [OF refl])
show ?case
proof cases
  assume \\exists k<n. f n = f k
  hence A = ?B using Suc.prems by (auto simp:less-Suc-eq)
  thus ?thesis using finB by simp
next
  assume \\neg(\exists k<n. f n = f k)
  hence A = insert (f n) ?B using Suc.prems by (auto simp:less-Suc-eq)
  thus ?thesis using finB by simp
qed

lemma finite-conv-nat-seg-image:
  finite A \iff (\exists (n::nat) f. A = f \{i::nat. i < n\})
by (blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-imp-inj-to-nat-seg:
  assumes finite A
  shows \exists n::nat. f \{i. i < n\} \land inj-on f A
proof
  from finite-imp-nat-seg-image-inj-on[OF finite A]
  obtain f and n::nat where bij: bij-betw f \{i. i<n\} A
    by (auto simp: bij-betw-def)
  let \?f = the-inv-into \{i. i<n\} f
  have inj-on \?f A \& \?f \' A = \{i. i<n\}
    by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
  thus ?thesis by blast
qed

lemma finite-Collect-less-nat [iff]:
  finite \{n::nat. n < k\}
by (fastforce simp: finite-conv-nat-seg-image)
lemma finite-Collect-le-nat [iff]:
finite \{n::nat. n \leq k\}
by (simp add: le-eq-less-or-eq Collect-disj-eq)

17.1.3 Finiteness and common set operations

lemma rev-finite-subset:
finite B =⇒ A ⊆ B =⇒ finite A
proof (induct arbitrary: A rule: finite-induct)
case empty
then show ?case by simp
next
case (insert x F A)
have A: A ⊆ insert x F and r: A - {x} ⊆ F =⇒ finite (A - {x}) by fact+
show finite A
proof cases
assume x: x ∈ A
with A have A - {x} ⊆ F by (simp add: subset-insert-iff)
with r have finite (A - {x}) .
hence finite (insert x (A - {x})) ..
also have insert x (A - {x}) = A using x by (rule insert-Diff)
finally show ?thesis .
next
show A ⊆ F =⇒ ?thesis by fact
assume x /∈ A
with A show A ⊆ F by (simp add: subset-insert-iff)
qed
qed

lemma finite-subset:
A ⊆ B =⇒ finite B =⇒ finite A
by (rule rev-finite-subset)

lemma finite-UnI:
assumes finite F and finite G
shows finite (F ∪ G)
using assms by induct simp-all

lemma finite-Un [iff]:
finite (F ∪ G) =⇒ finite F ∧ finite G
by (blast intro: finite-UnI finite-subset [of - F ∪ G])

lemma finite-insert [simp]: finite (insert a A) =⇒ finite A
proof -
have finite \{a\} ∧ finite A =⇒ finite A by simp
then have finite \{a\} ∪ A =⇒ finite A by (simp only: finite-Un)
then show ?thesis by simp
qed
lemma finite-Int [simp, intro]:  
finite F ∨ finite G ⇒ finite (F ∩ G)  
by (blast intro: finite-subset)

lemma finite-Collect-conjI [simp, intro]:  
finite \{ x. P x \} ∨ finite \{ x. Q x \} ⇒ finite \{ x. P x ∧ Q x \}  
by (simp add: Collect-conj-eq)

lemma finite-Collect-disjI [simp]:  
finite \{ x. P x ∨ Q x \} ←→ finite \{ x. P x \} ∧ finite \{ x. Q x \}  
by (simp add: Collect-disj-eq)

lemma finite-Diff [simp, intro]:  
finite A ⇒ finite (A − B)  
by (rule finite-subset, rule Diff-subset)

lemma finite-Diff2 [simp]:  
assumes finite B  
shows finite (A − insert a B) ←→ finite (A − B)  
proof -  
  have finite A ←→ finite((A − B) ∪ (A ∩ B)) by (simp add: Un-Diff-Int)  
  also have ... ←→ finite (A − B) using finite B by simp  
  finally show ?thesis ..

qed

lemma finite-Diff-insert [iff]:  
finite (A − insert a B) ←→ finite (A − B)  
proof -  
  have finite (A − B) ←→ finite (A − B − {a}) by simp  
  moreover have A − insert a B = A − B − {a} by auto  
  ultimately show ?thesis by simp

qed

lemma finite-compl[simp]:  
finite (A :: 'a set) ⇒ finite (− A) ←→ finite (UNIV :: 'a set)  
by (simp add: Compl-eq-Diff-UNIV)

lemma finite-Collect-not[simp]:  
finite \{ x :: 'a. P x \} ⇒ finite (− P x) ←→ finite (UNIV :: 'a set)  
by (simp add: Collect-neg-eq)

lemma finite-Union [simp, intro]:  
finite A ⇒ (∩M. M ∈ A ⇒ finite M) ⇒ finite(∪ A)  
by (induct rule: finite-induct) simp-all

lemma finite-UN-I [intro]:  
finite A ⇒ (∩a. a ∈ A ⇒ finite (B a)) ⇒ finite (∪a∈A. B a)  
by (induct rule: finite-induct) simp-all
lemma finite-UN [simp]:
finite A ==> finite (UNION A B) <-> (\forall x\in A. finite (B x))
by (blast intro: finite-subset)

lemma finite-Inter [intro]:
\exists A\in M. finite A ==> finite (\INTER M)
by (blast intro: Inter-lower finite-subset)

lemma finite-INT [intro]:
\exists x\in A. finite (A x) ==> finite (\INTER x\in A. A x)
by (blast intro: INT-lower finite-subset)

lemma finite-imageI [simp, intro]:
finite F = ==> finite (h ' F)
by (induct rule: finite-induct) simp-all

lemma finite-image-set [simp]:
finite \{ x. P x \} = ==> finite \{ f x | x. P x \}
by (simp add: image-Collect [symmetric])

lemma finite-imageD:
assumes finite (f ' A) and inj-on f A
shows finite A
using assms
proof (induct f ' A arbitrary: A)
case empty then show ?case by simp
next
case (insert x B)
then have B-A: insert x B = f ' A by simp
then obtain y where x = f y and y \in A by blast
from B-A \{ x \notin B \} have B = f ' A - \{ x \} by blast
with B-A \{ x \notin B \} have B = f ' (A - \{ y \}) by (simp add: inj-on-image-set-diff)
moreover from (inj-on f A) have inj-on f (A - \{ y \}) by (rule inj-on-diff)
ultimately have finite (A - \{ y \}) by (rule insert.hyps)
then show finite A by simp
qed

lemma finite-surj:
finite A ==> B \subseteq f ' A ==> finite B
by (erule finite-subset) (rule finite-imageI)

lemma finite-range-imageI:
finite (range g) ==> finite (range (\lambda x. f (g x)))
by (drule finite-imageI) (simp add: range-composition)

lemma finite-subset-image:
assumes finite B
shows $B \subseteq f \cdot A \Rightarrow \exists C \subseteq A. \text{finite } C \land B = f \cdot C$

using assms

proof induct
  case empty then show ?case by simp
next
  case insert then show ?case
    by (clarsimp simp del: image-insert simp add: image-insert [symmetric])
    blast

qed

lemma finite-vimage-IntI:
  finite $F \Rightarrow \text{inj-on } h A \Rightarrow \text{finite } (h -\cdot F \cap A)$
apply (induct rule: finite-induct)
apply simp-all
apply (subst vimage-insert)
apply (simp add: finite-subset [OF inj-on-vimage-singleton]
  Int-Un-distrib2)
done

lemma finite-vimageI:
  finite $F \Rightarrow \text{inj } h \Rightarrow \text{finite } (h -\cdot F)$
using finite-vimage-IntI[of $F$ $h$ UNIV]
by auto

lemma finite-vimageD:
  assumes fin: finite $(h -\cdot F)$ and surj: surj $h$
  shows finite $F$
proof
  have finite $(h -\cdot (h -\cdot F))$ using fin by (rule finite-vimageI)
  also have $h -\cdot (h -\cdot F) = F$ using surj by (rule surj-image-vimage-eq)
  finally show finite $F$.

qed

lemma finite-vimage-iff: bij $h \Rightarrow \text{finite } (h -\cdot F) \iff \text{finite } F$
  unfolding bij-def by (auto elim: finite-vimageD finite-vimageI)

lemma finite-Collect-bex [simp]:
  assumes finite $A$
  shows finite $(\{ x. \exists y \in A. Q x y \} \leftrightarrow (\forall y \in A. \text{finite } \{ x. Q x y \})$
proof
  have $(\{ x. \exists y \in A. Q x y \} = (\bigcup y \in A. \{ x. Q x y \})$ by auto
  with assms show ?thesis by simp

qed

lemma finite-Collect-bounded-ex [simp]:
  assumes finite $(\{ y. P y \})$
  shows finite $(\{ x. \exists y. P y \land Q x y \} \leftrightarrow (\forall y. P y \rightarrow \text{finite } \{ x. Q x y \})$
proof
  have $(\{ x. \exists y. P y \land Q x y \} = (\bigcup y \in \{ y. P y \}. \{ x. Q x y \})$ by auto
  with assms show ?thesis by simp

qed
lemma finite-Plus:
finite A → finite B → finite (A <+> B)
by (simp add: Plus-def)

lemma finite-PlusD:
fixes A :: 'a set and B :: 'b set
assumes fin: finite (A <+> B)
shows finite A ∧ finite B

proof
  have Inl ' A ⊆ A <+> B by auto
  then have finite (Inl ' A :: ('a + 'b) set) using fin by (rule finite-subset)
  then show finite A by (rule finite-imageD) (auto intro: inj-onI)

next
  have Inr ' B ⊆ A <+> B by auto
  then have finite (Inr ' B :: ('a + 'b) set) using fin by (rule finite-subset)
  then show finite B by (rule finite-imageD) (auto intro: inj-onI)
qed

lemma finite-Plus-iff [simp]:
finite (A <+> B) ←→ finite A ∧ finite B
by (auto intro: finite-PlusD finite-Plus)

lemma finite-Plus-UNIV-iff [simp, intro]:
finite (UNIV :: ('a + 'b) set) ←→ finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set)
by (subst UNIV-Plus-UNIV [symmetric]) (rule finite-Plus-iff)

lemma finite-SigmaI [simp, intro]:
finite A ⇒ (∀a. a ∈ A ⇒ finite (B a)) == finite (SIGMA a:A. B a)
by (unfold Sigma-def) blast

lemma finite-SigmaI2:
assumes finite {x ∈ A. B x ≠ {}}
and ∀a. a ∈ A ⇒ finite (B a)
shows finite (SIGMA A B)

proof
  from assms have finite (Sigma {x ∈ A. B x ≠ {}} B) by auto
  also have Sigma {x:A. B x ≠ {}} B = Sigma A B by auto
  finally show ?thesis .
qed

lemma finite-cartesian-product:
finite A → finite B → finite (A × B)
by (rule finite-SigmaI)

lemma finite-Prod-UNIV:
finite (UNIV :: 'a set) → finite (UNIV :: 'b set) → finite (UNIV :: ('a × 'b) set)
by  (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product)

lemma finite-cartesian-productD1:
  assumes finite (A × B) and B ≠ {}
  shows finite A
proof –
  from assms obtain n f where A × B = f ∘ {i :: nat. i < n}
    by (auto simp add: finite-conv-nat-seg-image)
  then have fst ∘ (A × B) = fst ∘ f ∘ {i :: nat. i < n} by simp
  with ⟨B ≠ {}⟩ have A = (fst ∘ f) ∘ {i :: nat. i < n}
    by (simp add: image-comp)
  then have ∃ n f. A = f ∘ {i :: nat. i < n} by blast
  then show ?thesis
    by (auto simp add: finite-conv-nat-seg-image)
qed

lemma finite-cartesian-productD2:
  assumes finite (A × B) and A ≠ {}
  shows finite B
proof –
  from assms obtain n f where A × B = f ∘ {i :: nat. i < n}
    by (auto simp add: finite-conv-nat-seg-image)
  then have snd ∘ (A × B) = snd ∘ f ∘ {i :: nat. i < n} by simp
  with ⟨A ≠ {}⟩ have B = (snd ∘ f) ∘ {i :: nat. i < n}
    by (simp add: image-comp)
  then have ∃ n f. B = f ∘ {i :: nat. i < n} by blast
  then show ?thesis
    by (auto simp add: finite-conv-nat-seg-image)
qed

lemma finite-cartesian-product-iff:
  finite (A × B) ←→ (A = {} ∨ B = {} ∨ (finite A ∧ finite B))
by (auto dest: finite-cartesian-productD1 finite-cartesian-productD2 finite-cartesian-product)

lemma finite-prod:
  finite (UNIV :: ('a × 'b) set) ←→ finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set)
using finite-cartesian-product-iff[of UNIV UNIV] by simp

lemma finite-Pow-iff [iff]:
  finite (Pow A) ←→ finite A
proof
  assume finite (Pow A)
  then have finite (%x. {x}) ∘ A by (blast intro: finite-subset)
  then show finite A by (rule finite-imageD [unfolded inj-on-def]) simp
next
  assume finite A
  then show finite (Pow A)
    by induct (simp-all add: Pow-insert)
corollary finite-Coll-subsets [simp, intro]:
finite A \Rightarrow finite \{B. B \subseteq A\}
by (simp add: Pow-def [symmetric])

lemma finite-set: finite (UNIV :: 'a set set) \iff finite (UNIV :: 'a set)
by (simp only: finite-Pow-iff Pow-UNIV [symmetric])

lemma finite-UnionD: finite(\bigcup A) =\Rightarrow finite A
by (blast intro: finite-subset [OF subset-Pow-Union])

lemma finite-set-of-finite-funs: assumes finite A finite B
shows finite {f. \forall x. (x \in A \rightarrow f x \in B) \land (x /\in A \rightarrow f x = d)}
(is finite ?S)
proof -
let ?F = \lambda f. \{(a,b). a \in A \land b = f a\}
have ?F ' ?S \subseteq Pow(A \times B) by auto
from finite-subset [OF this] assms have 1: finite (?F ' ?S) by simp
have 2: inj-on ?F ?S
  by (fastforce simp add: inj-on-def set-eq-iff fun-eq-iff)
show ?thesis by (rule \finite-imageD[OF 1 2])
qed

17.1.4 Further induction rules on finite sets

lemma finite-ne-induct [case-names singleton insert, consumes 2]:
assumes finite F and F \neq {}
assumes \A x. P \{x\}
  and \A x F. finite F \Rightarrow F \neq {} \Rightarrow x /\in F \Rightarrow P F \Rightarrow P (insert x F)
shows P F
using assms
proof induct
  case empty then show ?case by simp
next
  case (insert x F) then show ?case by cases auto
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
assumes finite F and F \subseteq A
assumes empty: P \{\}
  and insert: \A a. finite F \Rightarrow a \in A \Rightarrow a /\in F \Rightarrow P F \Rightarrow P (insert a F)
shows P F
using (finite F). \forall F \subseteq A;
proof induct
  show P \{\} by fact
next
  fix x F
  assume finite F and x /\in F and
  P. F \subseteq A \Rightarrow P F and i: insert x F \subseteq A
show $P \ (\text{insert } x \ F)$
proof (rule insert)
  from $i$ show $x \in A$ by blast
from $i$ have $F \subseteq A$ by blast
  with $P$ show $P \ F$ .
show $\text{finite } F$ by fact
show $x \not\in F$ by fact
qed

lemma $\text{finite-empty-induct}$:
  assumes $\text{finite } A$
  assumes $P \ A$
  and remove: $\forall a \in A. \text{finite } A \Rightarrow a \in A \Rightarrow P \ A \Rightarrow P \ (A - \{a\})$
  shows $P \ \{\}$
proof -
  have $\forall B. \ B \subseteq A \Rightarrow P \ (A - B)$
proof -
  fix $B :: \ 'a \ set$
  assume $B \subseteq A$
  with $(\text{finite } A)$ have $\text{finite } B$ by (rule rev-finite-subset)
  from $\text{this } (B \subseteq A)$ show $P \ (A - B)$
  proof induct
    case empty
    from $\langle P \ A \rangle$ show $?\text{case by simp}$
  next
    case $(\text{insert } b \ B)$
    have $P \ (A - B - \{b\})$
    proof (rule remove)
      from $(\text{finite } A)$ show $\text{finite } (A - B)$ by induct auto
      from $\text{insert}$ show $b \in A - B$ by simp
      from $\text{insert}$ show $P \ (A - B)$ by simp
    qed
    also have $A - B - \{b\} = A - \text{insert } b \ B$ by (rule Diff-insert [symmetric])
    finally show $?\text{case .}$
  qed
  qed
  qed
  then have $P \ (A - A)$ by blast
  then show $\text{thesis by simp}$
  qed

17.2 Class finite

class finite =
  assumes finite-UNIV: $\text{finite } (UNIV :: \ 'a \ set)$
begin

lemma finite [simp]: $\text{finite } (A :: \ 'a \ set)$
  by (rule subset-UNIV finite-UNIV finite-subset)+
lemma finite-code [code]: finite (A :: 'a set) ⟷ True
  by simp
end

instance prod :: (finite, finite) finite
  by default (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product finite)

lemma inj-graph: inj (%f. {(x, y). y = f x})
  by (rule inj-onI, auto simp add: set-eq-iff fun-eq-iff)

instance fun :: (finite, finite) finite
proof
  show finite (UNIV :: ('a ⇒ 'b) set)
  proof (rule finite-imageD)
    let ?graph = %f.'a⇒'b. {(x, y). y = f x}
    have range ?graph ⊆ Pow UNIV by simp
    moreover have finite (Pow (UNIV :: ('a * 'b) set))
      by (simp only: finite-Pow-iff finite)
    ultimately show finite (range ?graph)
      by (rule finite-subset)
    show inj ?graph by (rule inj-graph)
  qed
qed

instance bool :: finite
  by default (simp add: UNIV-bool)

instance set :: (finite) finite
  by default (simp only: Pow-UNIV [symmetric] finite-Pow-iff finite)

instance unit :: finite
  by default (simp add: UNIV-unit)

instance sum :: (finite, finite) finite
  by default (simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite)

17.3 A basic fold functional for finite sets

The intended behaviour is fold f z \{x_1, ..., x_n\} = f x_1 (f x_n z) if f is “left-commutative”:
locale comp-fun-commute =
  fixes f :: 'a ⇒ 'b ⇒ 'b
  assumes comp-fun-commute: f y o f x = f x o f y
begin

lemma fun-left-comm: f y (f x z) = f x (f y z)
using comp-fun-commute by (simp add: fun-eq-iff)

lemma commute-left-comp:
  \( f \circ (f \circ g) = f \circ (f \circ g) \)
by (simp add: o-assoc comp-fun-commute)
end

inductive fold-graph :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool
for f :: 'a ⇒ 'b ⇒ 'b and z :: 'b where
  emptyI [intro]: fold-graph f z {} z
  insertI [intro]: \( x \notin A \imp \text{fold-graph } f z A y \)
  \( \implies \text{fold-graph } f z (\text{insert } x A) (f x y) \)

inductive-cases empty-fold-graphE [elim!]: fold-graph f z {} x

definition fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b where
fold f z A = (if finite A then (THE y. fold-graph f z A y) else z)

A tempting alternative for the definiens is if finite A then THE y. fold-graph
f z A y else e. It allows the removal of finiteness assumptions from the
theorems fold-comm, fold-reindex and fold-distrib. The proofs become ugly.
It is not worth the effort. (???)

lemma finite-imp-fold-graph: finite A ⇒ ∃x. fold-graph f z A x
by (induct rule: finite-induct) auto

17.3.1 From fold-graph to fold

context comp-fun-commute
begin

lemma fold-graph-finite:
  assumes fold-graph f z A y
  shows finite A
  using assms by (induct simp-all)

lemma fold-graph-insertE-aux:
  fold-graph f z A y ⇒ a ∈ A ⇒ ∃y'. y = f a y' ∧ fold-graph f z (A - {a}) y'
proof (induct set: fold-graph)
  case (insertI x A y) show ?case
  proof (cases x = a)
    assume x = a with insertI show ?case by auto
  next
    assume x ≠ a
    then obtain y' where y: y = f a y' and y': fold-graph f z (A - {a}) y'
    using insertI by auto
    have f x y = f a (f x y')
    unfolding y by (rule fun-left-comm)
    moreover have fold-graph f z (insert x A - {a}) (f x y')
using $y'$ and ($x \neq a$ and ($x \notin A$)
by (simp add: insert-Diff-if fold-graph.insertI)
ultimately show \texttt{?case} by fast
qed

\textbf{lemma} fold-graph-insertE:
\begin{enumerate}
  \item assumes \texttt{fold-graph f z (insert x A) v and}\n  \item obtains \texttt{y where}\n  \item using \texttt{assms by (auto dest: fold-graph-insertE-aux [OF - insertI1])}
\end{enumerate}

\textbf{lemma} fold-graph-determ:
\begin{enumerate}
  \item assumes \texttt{finite A}\n  \item shows \texttt{fold-graph f z A (fold f z A)}
\end{enumerate}
\begin{proof}
\begin{enumerate}
  \item from \texttt{assms have }\exists x. \texttt{fold-graph f z A x by (rule finite-imp-fold-graph)}
  \item moreover note \texttt{fold-graph-determ}
  \item ultimately have \exists! x. \texttt{fold-graph f z A x by (rule ex-ex1I)}
  \item then have \texttt{fold-graph f z A (The (fold-graph f z A)) by (rule the1')}\n  \item with \texttt{assms show }\texttt{?thesis by (simp add: fold-def)}
\end{enumerate}
\end{proof}

\textbf{lemmas} (in –) fold-infinite [simp]:
\begin{enumerate}
  \item assumes \texttt{finite A}\n  \item shows \texttt{fold f z A = z}
\end{enumerate}
\begin{proof}
\begin{enumerate}
  \item from \texttt{assms by (auto simp add: fold-def)}
\end{enumerate}
\end{proof}

\textbf{lemmas} (in –) fold-empty [simp]:
\begin{enumerate}
  \item \texttt{fold f z \{} = z\n  \item by (auto simp add: fold-def)}
\end{enumerate}

The various recursion equations for \texttt{fold}:
\textbf{lemma} fold-insert [simp]:
assumes finite A and x \notin A
shows fold f z (insert x A) = f x (fold f z A)
proof (rule fold-equality)
  fix z
  from \langle finite A \rangle have fold-graph f z A (fold f z A) by (rule fold-graph-fold)
  with \langle x \notin A \rangle have fold-graph f z (insert x A) (f x (fold f z A)) by (rule fold-graph.insertI)
  then show fold-graph f z (insert x A) (f x (fold f z A)) by simp
qed

declare (in −) empty-fold-graphE [rule del] fold-graph.intros [rule del]
— No more proofs involve these.

lemma fold-fun-left-comm:
  finite A \implies f x (fold f z A) = fold f (f x z) A
proof (induct rule: finite-induct)
  case empty then show ?case by simp
next
  case (insert y A) then show ?case
    by (simp add: fun-left-comm [of x])
qed

lemma fold-insert2:
  finite A \implies f x (fold f z A) = fold f (f x z) A
by (simp add: fold-fun-left-comm)

lemma fold-rec:
  assumes finite A and x \in A
  shows fold f z A = f x (fold f z (A − {x}))
proof
  have A: A = insert x (A − {x}) using \langle x \in A \rangle by blast
  then have fold f z A = fold f z (insert x (A − {x})) by simp
  also have \ldots = f x (fold f z (A − {x}))
    by (rule fold-insert) (simp add: finite A)+
  finally show ?thesis .
qed

lemma fold-insert-remove:
  assumes finite A
  shows fold f z (insert x A) = f x (fold f z (A − {x}))
proof
  from \langle finite A \rangle have finite (insert x A) by auto
  moreover have x \in insert x A by auto
  ultimately have fold f z (insert x A) = f x (fold f z (insert x A − {x}))
    by (rule fold-rec)
  then show ?thesis by simp
qed

lemma fold-set-union-disj:
assumes finite A finite B A ∩ B = {}
shows Finite-Set.fold f z (A ∪ B) = Finite-Set.fold f (Finite-Set.fold f z A) B
using assms(2,1,3) by induction simp-all
end

Other properties of fold:

lemma fold-image:
assumes inj-on g A
shows fold f z (g ' A) = fold (f ∘ g) z A
proof (cases finite A)
case False with assms show ?thesis by (auto dest: finite-imageD simp add: fold-def)
next
case True
have fold-graph f z (g ' A) = fold-graph (f ∘ g) z A
proof
fix w
show fold-graph f z (g ' A) w ←→ fold-graph (f ∘ g) z A w (is ?P ←→ ?Q)
proof
assume ?P then show ?Q using assms
proof (induct g ' A w arbitrary: A)
case emptyI then show ?case by (auto intro: fold-graph.emptyI)
next
case (insertI x A r)
from inj-on g B ⟨x /∈ A⟩ ⟨insert x A = image g B⟩ obtain x' A' where x' /∈ A' and [simp]: B = insert x' A' x = g x' A = g ' A'
by (rule inj-img-insertE)
from insertI.prems have fold-graph f z (g ' A) r
by (auto intro: insertI.hyps)
with ⟨x' /∈ A'⟩ have fold-graph (f ∘ g) z (insert x' A') ((f ∘ g) x' r)
by (rule fold-graph.insertI)
then show ?case by simp
qed
next
assume ?Q then show ?P using assms
proof induction
next
case (insertI x A r)
from ⟨x /∈ A⟩ insertI.prems have g x /∈ g ' A by auto
moreover from insertI have fold-graph f z (g ' A) r by simp
ultimately have fold-graph f z (insert (g x) (g ' A)) (f (g x) r)
by (rule fold-graph.insertI)
then show ?case by simp
qed
qed
qed
with True assms show ?thesis by (auto simp add: fold-def)
lemma fold-cong:
  assumes comp-fun-commute f comp-fun-commute g
  assumes finite A and cong: ∀x. x ∈ A ⇒ f x = g x
  and s = t and A = B
  shows fold f s A = fold g t B
proof –
  have fold f s A = fold g s A
    using ⟨finite A⟩ cong proof (induct A)
    case empty then show ?case by simp
  next
    case (insert x A)
    interpret f: comp-fun-commute f by (fact ⟨comp-fun-commute f⟩)
    interpret g: comp-fun-commute g by (fact ⟨comp-fun-commute g⟩)
    from insert show ?case by simp
  qed
  with assms show ?thesis by simp
qed

A simplified version for idempotent functions:
locale comp-fun-idem = comp-fun-commute +
  assumes comp-fun-idem: f x ◦ f x = f x
begin

lemma fun-left-idem: f x (f x z) = f x z
  using comp-fun-idem by (simp add: fun-eq-iff)

lemma fold-insert-idem:
  assumes fin: finite A
  shows fold f z (insert x A) = f x (fold f z A)
proof cases
  assume x ∈ A
  then obtain B where A = insert x B and x ∉ B by (rule set-insert)
  then show ?thesis using assms by (simp add: comp-fun-idem fun-left-idem)
next
  assume x ∉ A then show ?thesis using assms by simp
qed

declare fold-insert [simp del] fold-insert-idem [simp]

lemma fold-insert-idem2:
  finite A ⇒ fold f z (insert x A) = fold f (f x z) A
  by (simp add: fold-fun-left-comm)
end
17.3.2 Liftings to \textit{comp-fun-commute} etc.

\textbf{lemma (in \textit{comp-fun-commute}) \textit{comp-comp-fun-commute}:}
\textit{comp-fun-commute} \((f \circ g)\)

\textbf{proof}
\textbf{qed (simp-all add: comp-fun-commute)}

\textbf{lemma (in \textit{comp-fun-idem}) \textit{comp-comp-fun-idem}:}
\textit{comp-fun-idem} \((f \circ g)\)
\textit{by (rule \textit{comp-fun-idem.intro}, \textit{rule \textit{comp-comp-fun-commute}}, \textit{unfold-locales})}
\textbf{(simp-all add: comp-fun-idem)}

\textbf{lemma (in \textit{comp-fun-commute}) \textit{comp-fun-commute-funpow}:}
\textit{comp-fun-commute} \((\lambda x. f x^{g} x)\)

\textbf{proof}
\textit{fix} \textit{y x}
\textit{show} \(f y^{g} y \circ f x^{g} x = f x^{g} x \circ f y^{g} y\)
\textbf{proof (cases} \textit{x = y)}
\textit{case True} then show \(\text{thesis by simp}\)
\textbf{next}
\textit{case False} show \(\text{thesis}\)
\textbf{proof (induct} \textit{g x arbitrary:} \textit{g)}
\textit{case} \textit{0 then show} \(\text{?case by simp}\)
\textbf{next}
\textit{case} \textit{(Suc} \textit{n g)}
\textit{have} \textit{hyp1:} \(f y^{g} y \circ f x^{g} x = f x^{g} x \circ f y^{g} y\)
\textbf{proof (induct} \textit{g y arbitrary:} \textit{g)}
\textit{case} \textit{0 then show} \(\text{?case by simp}\)
\textbf{next}
\textit{case} \textit{(Suc} \textit{n g)}
\textit{def} \textit{h \equiv \lambda z. g z - 1} with \textit{Suc} have \textit{n = h y by simp}
\textit{with} \textit{Suc} have \textit{hyp:} \(f y^{g} h y \circ f x^{g} x = f x^{g} x \circ f y^{g} h y\)
\textit{by auto}
\textbf{from} \textit{Suc} \textit{h-def have} \textit{g y = Suc (h y) by simp then show} \(\text{?case by (simp add: comp-assoc hyp)}\)
\textbf{(simp add: o-assoc comp-fun-commute)}
\textbf{qed}

\textit{def} \textit{h \equiv \lambda z. if} \textit{z = x} then \textit{g x - 1} \textit{else} \textit{g z} with \textit{Suc} have \textit{n = h x by simp}
\textit{with} \textit{Suc} have \textit{hyp2:} \(f y^{g} h y \circ f x^{g} h x = f x^{g} h x \circ f y^{g} h y\)
\textit{by auto}
\textit{with} \textit{False} \textit{h-def have} \textit{hyp2:} \(f y^{g} g y \circ f x^{g} h x = f x^{g} h x \circ f y^{g} g y\)
\textit{by simp}
\textbf{from} \textit{Suc} \textit{h-def have} \textit{g x = Suc (h x) by simp then show} \(\text{?case by (simp del: funpow.simps add: funpow-Suc-right o-assoc hyp2)}\)
\textbf{(simp add: comp-assoc hyp1)}
\textbf{qed}
\textbf{qed}
17.3.3 Expressing set operations via fold

**lemma** comp-fun-commute-const:
  comp-fun-commute ($\lambda$. $f$)
**proof**
qed rule

**lemma** comp-fun-idem-insert:
  comp-fun-idem insert
**proof**
qed auto

**lemma** comp-fun-idem-remove:
  comp-fun-idem Set.remove
**proof**
qed auto

**lemma** (in semilattice-inf) comp-fun-idem-inf:
  comp-fun-idem inf
**proof**
qed (auto simp add: inf-left-commute)

**lemma** (in semilattice-sup) comp-fun-idem-sup:
  comp-fun-idem sup
**proof**
qed (auto simp add: sup-left-commute)

**lemma** union-fold-insert:
  assumes finite $A$
  shows $A \cup B = \text{fold insert } B \ A$
**proof**
  interpret comp-fun-idem insert by (fact comp-fun-idem-insert)
  from (finite $A$) show ?thesis by (induct $A$ arbitrary: $B$) simp-all
qed

**lemma** minus-fold-remove:
  assumes finite $A$
  shows $B - A = \text{fold Set.remove } B \ A$
**proof**
  interpret comp-fun-idem Set.remove by (fact comp-fun-idem-remove)
  from (finite $A$) have fold Set.remove $B \ A = B - A$ by (induct $A$ arbitrary: $B$)
  auto
  then show ?thesis ..
qed

**lemma** comp-fun-commute-filter-fold:
  comp-fun-commute ($\lambda x \ A'. \text{ if } P \ x \ \text{then } \text{Set.insert } x \ A' \ \text{else } A'$)
proof 
  interpret comp-fun-idem Set.insert by (fact comp-fun-idem-insert) 
  show ?thesis by default (auto simp: fun-eq-iff) 
qed 

lemma Set-filter-fold: 
  assumes finite A 
  shows Set.filter P A = fold (λx A'. if P x then Set.insert x A' else A') {} A 
  using assms 
  by (induct A) (auto simp add: Set.filter-def comp-fun-commute.fold-insert[OF comp-fun-commute-filter-fold]) 

lemma inter-set-filter: 
  assumes finite B 
  shows A ∩ B = Set.filter (λx. x ∈ A) B 
  using assms 
  by (induct B) (auto simp: Set.filter-def) 

lemma image-fold-insert: 
  assumes finite A 
  shows image f A = fold (λk A. Set.insert (f k) A) {} A 
  using assms 
  proof 
    interpret comp-fun-commute λk A. Set.insert (f k) A by default auto 
    show ?thesis using assms by (induct A) auto 
  qed 

lemma Ball-fold: 
  assumes finite A 
  shows Ball A P = fold (λk s. s ∧ P k) True A 
  using assms 
  proof 
    interpret comp-fun-commute λk s. s ∧ P k by default auto 
    show ?thesis using assms by (induct A) auto 
  qed 

lemma Bex-fold: 
  assumes finite A 
  shows Bex A P = fold (λk s. s ∨ P k) False A 
  using assms 
  proof 
    interpret comp-fun-commute λk s. s ∨ P k by default auto 
    show ?thesis using assms by (induct A) auto 
  qed 

lemma comp-fun-commute-Pow-fold: 
  comp-fun-commute (λx A. A ∪ Set.insert x A) 
  by (clarsimp simp: fun-eq-iff comp-fun-commute-def) blast
lemma Pow-fold:
assumes finite A
shows Pow A = fold (λx A. A ∪ Set.insert x ' A) {{}} A
using assms
proof –
interpret comp-fun-commute λx A. A ∪ Set.insert x ' A by (rule comp-fun-commute-Pow-fold)
show ?thesis using assms by (induct A) (auto simp: Pow-insert)
qed

lemma fold-union-pair:
assumes finite B
shows (∪ y ∈ B. {(x, y)}) ∪ A = fold (λy. Set.insert (x, y)) A B
proof –
interpret comp-fun-commute λy. Set.insert (x, y) by default auto
show ?thesis using assms by (induct B arbitrary: A) simp-all
qed

lemma comp-fun-commute-product-fold:
assumes finite B
shows comp-fun-commute (λx z. fold (λy. Set.insert (x, y)) z B)
by default (auto simp: fold-union-pair[symmetric] assms)

lemma product-fold:
assumes finite A
assumes finite B
shows A × B = fold (λx z. fold (λy. Set.insert (x, y)) z B) {{}} A
using assms unfolding Sigma-def
by (induct A)
(simp-all add: comp-fun-commute.fold-insert[OF comp-fun-commute-product-fold]
fold-union-pair)

context complete-lattice
begin

lemma inf-Inf-fold-inf:
assumes finite A
shows inf (Inf A) B = fold inf B A
proof –
interpret comp-fun-idem inf by (fact comp-fun-idem-inf)
from ⟨finite A⟩ fold-fun-left-comm show ?thesis by (induct A arbitrary: B)
(simp-all add: inf-commute fun-eq-iff)
qed

lemma sup-Sup-fold-sup:
assumes finite A
shows sup (Sup A) B = fold sup B A
proof –
interpret comp-fun-idem sup by (fact comp-fun-idem-sup)
from ⟨finite A⟩ fold-fun-left-comm show ?thesis by (induct A arbitrary: B) (simp-all add: sup-commute fun-eq-iff) qed

lemma Inf-fold-inf:
  assumes finite A
  shows Inf A = fold inf top A
  using assms inf-Inf-fold-inf [of A top] by (simp add: inf-absorb2)

lemma Sup-fold-sup:
  assumes finite A
  shows Sup A = fold sup bot A
  using assms sup-Sup-fold-sup [of A bot] by (simp add: sup-absorb2)

lemma inf-INF-fold-inf:
  assumes finite A
  shows inf B (INFIMUM A f) = fold (inf ∘ f) B A (is inf = ?fold)
  proof (rule sgm)
    interpret comp-fun-idem inf by (fact comp-fun-idem-inf)
    interpret comp-fun-idem inf ∘ f by (fact comp-comp-fun-idem)
    from ⟨finite A⟩ show ?fold = ?inf
      by (induct A arbitrary: B) (simp-all add: inf-left-commute)
  qed

lemma sup-SUP-fold-sup:
  assumes finite A
  shows sup B (SUPREMUM A f) = fold (sup ∘ f) B A (is sup = ?fold)
  proof (rule sgm)
    interpret comp-fun-idem sup by (fact comp-fun-idem-sup)
    interpret comp-fun-idem sup ∘ f by (fact comp-comp-fun-idem)
    from ⟨finite A⟩ show ?fold = ?sup
      by (induct A arbitrary: B) (simp-all add: sup-left-commute)
  qed

lemma INF-fold-inf:
  assumes finite A
  shows INFIMUM A f = fold (inf ∘ f) top A
  using assms inf-INF-fold-inf [of A top] by simp

lemma SUP-fold-sup:
  assumes finite A
  shows SUPREMUM A f = fold (sup ∘ f) bot A
  using assms sup-SUP-fold-sup [of A bot] by simp

end
17.4 Locales as mini-packages for fold operations

17.4.1 The natural case

locale folding =
  fixes f :: 'a ⇒ 'b ⇒ 'b
  fixes z :: 'b
  assumes comp-fun-commute: f y ∘ f x = f x ∘ f y

begin

interpretation fold?: comp-fun-commute f
  by default (insert comp-fun-commute, simp add: fun-eq-iff)

definition F :: 'a set ⇒ 'b where
  eq-fold: F A = fold f z A

lemma empty [simp]:
  F { } = z
  by (simp add: eq-fold)

lemma infinite [simp]:
  ¬ finite A ⇒ F A = z
  by (simp add: eq-fold)

lemma insert [simp]:
  assumes finite A and x /∈ A
  shows F (insert x A) = f x (F A)
  proof –
    from fold-insert assms
    have fold f z (insert x A) = f x (fold f z A) by simp
    with ⟨finite A⟩ show ?thesis by (simp add: eq-fold fun-eq-iff)
  qed

lemma remove:
  assumes finite A and x ∈ A
  shows F A = f x (F (A − {x}))
  proof –
    from ⟨x ∈ A ⟩ obtain B where A: A = insert x B and x /∈ B
    by (auto dest: mk-disjoint-insert)
    moreover from ⟨finite A ⟩ A have finite B by simp
    ultimately show ?thesis by simp
  qed

lemma insert-remove:
  assumes finite A
  shows F (insert x A) = f x (F (A − {x}))
  using assms by (cases x ∈ A) (simp-all add: remove insert-absorb)

end
17.4.2 With idempotency

locale folding-idem = folding +
  assumes comp-fun-idem: \( f \circ f = f \) begin
  declare insert [simp del]

interpretation fold?: comp-fun-idem \( f \)
  by default (insert comp-fun-commute comp-fun-idem, simp add: fun-eq-iff)

lemma insert-idem [simp]:
  assumes finite \( A \)
  shows \( F (\text{insert } x \ A) = f x (F A) \)
  proof -
    from fold-insert-idem assms have fold \( f \ z (\text{insert } x \ A) = f x (\text{fold } f \ z \ A) \) by simp
    with \( \text{finite } A \) show \( \text{thesis} \) by (simp add: eq-fold fun-eq-iff)
  qed

end

17.5 Finite cardinality

The traditional definition \( \text{card } A \equiv \text{LEAST } n. \exists f. A = \{ f \ i \ | i. i < n \} \) is ugly to work with. But now that we have fold things are easy:

definition card :: \('a set \Rightarrow \text{nat}\) where
  card = folding.F \( \lambda \ -. \ Suc \) 0

interpretation card!: folding \( \lambda \ -. \ Suc \) 0 where
  folding.F \( \lambda \ -. \ Suc \) 0 = card
  proof -
    show folding \( \lambda \ -. \ Suc \) by default rule
      then interpret card!: folding \( \lambda \ -. \ Suc \) 0 .
      from card-def show folding.F \( \lambda \ -. \ Suc \) 0 = card by rule
  qed

lemma card-infinite: 
  \( \neg \text{finite } A \Rightarrow \text{card } A = 0 \)
  by (fact card.infinite)

lemma card-empty: 
  card \( \{\} \) = 0
  by (fact card.empty)

lemma card-insert-disjoint: 
  finite \( A \Rightarrow x \notin A \Rightarrow \text{card } (\text{insert } x \ A) = \text{Suc } (\text{card } A) \)
  by (fact card.insert)
lemma card-insert-if:
finite A \implies card (insert x A) = (if x \in A then card A else Suc (card A))
by auto (simp add: card.insert-remove card.remove)

lemma card-ge-0-finite:
card A > 0 \implies finite A
by (rule ccontr) simp

lemma card-0-eq [simp]:
finite A \implies card A = 0 \iff A = \{} 
by (auto dest: mk-disjoint-insert)

lemma finite-UNIV-card-ge-0:
finite (UNIV :: 'a set) \implies card (UNIV :: 'a set) > 0
by (rule ccontr) simp

lemma card-eq-0-iff:
card A = 0 \iff A = \{} \lor \neg finite A
by auto

lemma card-gt-0-iff:
0 < card A \iff A \neq \{} \land finite A
by (simp add: neq0-conv [symmetric] card-eq-0-iff)

lemma card-Suc-Diff1:
finite A \implies x \in A \implies Suc (card (A - \{x\})) = card A
apply (rule-tac t = A in insert-Diff [THEN subst], assumption)
apply (simp del:insert-Diff-single)
done

lemma card-Diff-singleton:
finite A \implies x \in A \implies card (A - \{x\}) = card A - 1
by (simp add: card-Suc-Diff1 [symmetric])

lemma card-Diff-singleton-if:
finite A \implies card (A - \{x\}) = (if x \in A then card A - 1 else card A)
by (simp add: card-Diff-singleton)

lemma card-Diff-insert [simp]:
assumes finite A and a \in A and a \notin B
shows card (A - insert a B) = card (A - B) - 1
proof
  have A - insert a B = (A - B) - \{a\} using assms by blast
  then show thesis using assms by(simp add: card-Diff-singleton)
qed

lemma card-insert: finite A \implies card (insert x A) = Suc (card (A - \{x\}))
by (fact card.insert-remove)
lemma card-insert-le: finite A ==> card A <= card (insert x A)
by (simp add: card-insert-if)

lemma card-Collect-less-nat[simp]: card{i::nat. i < n} = n
by (induct n) (simp-all add:less-Suc-eq Collect-disj-eq)

lemma card-Collect-le-nat[simp]: card{i::nat. i <= n} = Suc n
using card-Collect-less-nat[of Suc n] by(simp add: less-Suc-eq-le)

lemma card-mono:
  assumes finite B and A <= B
  shows card A <= card B
proof -
  from assms have finite A by (auto intro: finite-subset)
  then show thesis using assms proof (induct A arbitrary: B)
    case empty then show ?case by simp
  next
    case (insert x A)
    then have x : B by simp
    from insert have A <= B - {x} and finite (B - {x}) by auto
    with insert.hyps have card A <= card (B - {x}) by auto
    with (finite A) (x /∈ A) (finite B) (x ∈ B) show ?case by simp
  qed
qed

lemma card-seteq: finite B ==> (!!A. A <= B ==> card B <= card A ==> A = B)
apply (induct rule: finite-induct)
apply simp
apply clarify
apply (subgoal-tac finite A & A - {x} <= F)
prefer 2 apply (blast intro: finite-subset, atomize)
apply (drule-tac x = A - {x} in spec)
apply (simp add: card-Diff-singleton-if split add: split-if-asm)
apply (case-tac card A, auto)
done

lemma psubset-card-mono: finite B ==> A < B ==> card A < card B
apply (simp add: psubset-eq linorder-not-le [symmetric])
apply (blast dest: card-seteq)
done

lemma card-Un-Int:
  assumes finite A and finite B
  shows card A + card B = card (A ∪ B) + card (A ∩ B)
using assms proof (induct A)
  case empty then show ?case by simp
next
  case (insert x A) then show ?case
    by (auto simp add: insert-absorb Int-insert-left)
qed

lemma card-Un-disjoint:
  assumes finite A and finite B
  assumes A ∩ B = {}
  shows card (A ∪ B) = card A + card B
  using assms card-Un-Int [of A B] by simp

lemma card-Diff-subset:
  assumes finite B and B ⊆ A
  shows card (A − B) = card A − card B
proof (cases finite A)
  case False with assms show ?thesis by simp
next
  case True with assms show ?thesis by (induct B arbitrary: A) simp-all
qed

lemma card-Diff-subset-Int:
  assumes AB: finite (A ∩ B) shows card (A − B) = card A − card (A ∩ B)
proof
  have A − B = A − A ∩ B by auto
  thus ?thesis
    by (simp add: card-Diff-subset AB)
qed

lemma diff-card-le-card-Diff:
  assumes finite B shows card A − card B ≤ card (A − B)
proof
  have card A − card B ≤ card A − card (A ∩ B)
    using card_mono[OF assms Int_lower2, of A] by arith
  also have ... = card (A − B) using assms by (simp add: card-Diff-subset-Int)
  finally show ?thesis.
qed

lemma card-Diff1-less: finite A ==> x: A ==> card (A − {x}) < card A
apply (rule Suc-less-SucD)
apply (simp add: card-Suc-Diff1 del:card-Diff-insert)
done

lemma card-Diff2-less:
  finite A ==> x: A ==> y: A ==> card (A − {x} − {y}) < card A
apply (case_tac x = y)
apply (simp add: card-Diff1-less del:card-Diff-insert)
apply (rule less_trans)
prefer 2 apply (auto intro!: card-Diff1-less simp del:card-Diff-insert)
done
lemma card-Diff1-le: finite A ==> card (A - {x}) < card A
apply (case-tac x : A)
apply (simp-all add: card-Diff1-less less-imp-le)
done

lemma card-psubset: finite B ==> A ⊆ B ==> card A < card B ==> A < B
by (erule psubsetI, blast)

lemma card-le-inj:
assumes fA: finite A
and fB: finite B
and c: card A ≤ card B
shows ∃ f. f ' A ⊆ B ∧ inj-on f A
using fA fB c
proof (induct arbitrary: B rule: finite-induct)
case empty
then show ?case by simp
next
case (insert x s t)
then show ?case
proof (induct rule: finite-induct[of insert.prems(1)])
case 1
then show ?case by simp
next
next
from 2.prems(1,2,5) 2.hyps(1,2) have cst: card s ≤ card t
by simp
from 2.prems(3) [OF 2.hyps(1) cst]
obtain f where f ' s ⊆ t inj-on f s
by blast
with 2.prems(2) 2.hyps(2) show ?case
apply
apply (rule exI[where x = λz. if z = x then y else f z])
apply (auto simp add: inj-on-def)
done
qed

lemma card-subset-eq:
assumes fB: finite B
and AB: A ⊆ B
and c: card A = card B
shows A = B
proof
from fB AB have fA: finite A
by (auto intro: finite-subset)
from fA fB have fBA: finite (B - A)
by auto
have e: \( A \cap (B - A) = \{\} \)
  by blast
have eq: \( A \cup (B - A) = B \)
  using \( AB \) by blast
from \( \text{card-Un-disjoint}[OF fA fBA e, unfolded eq e] \) have \( \text{card} (B - A) = 0 \)
  by arith
then have \( B - A = \{\} \)
  unfolding \( \text{card-eq-0-iff} \) using \( fA fB \) by simp
with \( AB \) show \( A = B \)
  by blast
qed

lemma \( \text{insert-partition}: \)
[ \[ x \notin F; \forall c1 \in \text{insert} x F. \forall c2 \in \text{insert} x F. c1 \neq c2 \rightarrow c1 \cap c2 = \{\} \] ]
\( \Rightarrow x \cap \bigcup F = \{\} \)
by auto

lemma \( \text{finite-psubset-induct}[\text{consumes 1}, \text{case-names psubset}]: \)
  assumes fin: \( \text{finite} A \)
  and major: \( \forall A. \text{finite} A \Rightarrow (\forall B. B \subseteq A \Rightarrow P B) \Rightarrow P A \)
  shows \( P A \)
using \( \text{fin} \)
proof (induct \( A \) taking: \( \text{card} \) rule: measure-induct-rule)
  case \( \text{less} A \)
  have fin: \( \text{finite} A \) by fact
  have \( \text{ih}: \forall B. [\[ \text{card} B < \text{card} A; \text{finite} B \] \Rightarrow P B] \) by fact
  \{ fix \( B \)
    assume asm: \( B \subseteq A \)
    from asm have \( \text{card} B < \text{card} A \) using \( \text{psubset-card-mono} \) \( \text{fin} \) by blast
    moreover
    from asm have \( B \subseteq A \) by auto
    then have \( \text{finite} B \) using \( \text{fin} \) \( \text{finite-subset} \) by blast
    ultimately
    have \( P B \) using \( \text{ih} \) by simp
  \}
  with \( \text{fin} \) show \( P A \) using major by blast
qed

lemma \( \text{finite-induct-select}[\text{consumes 1}, \text{case-names empty select}]: \)
  assumes \( \text{finite} S \)
  assumes \( P \{\} \)
  assumes select: \( \forall T. T \subseteq S \Rightarrow P T \Rightarrow \exists s \in S - T. P (\text{insert} s T) \)
  shows \( P S \)
proof -
  have \( 0 \leq \text{card} S \) by simp
  then have \( \exists T \subseteq S. \text{card} T = \text{card} S \land P T \)
  proof (induct rule: \( \text{dec-induct} \))
    case base with \( P \{\} \); show \( ?\text{case} \)
    by (intro \( \text{exI} \)[of - \{\}]); auto
next
  case (step n)
  then obtain T where T ⊆ S card T = n P T
    by auto
  with ⟨n < card S⟩ have T ⊂ S P T
    by auto
  with ⟨finite S⟩ obtain s where s ∈ S s ∉ T
    by (auto dest: finite-subset)
  with ⟨finite S⟩ show ?case
    by (intro exI[of - insert s T]) (auto dest: finite-subset)
qed

main cardinality theorem

lemma card-partition [rule-format]:
  finite C ==>
  finite (∪ C) ==>
  (∀ c ∈ C. card c = k) ==>
  (∀ c1 ∈ C. ∀ c2 ∈ C. c1 ≠ c2 ==> c1 ∩ c2 = {}) ==>
  k * card(C) = card (∪ C)
apply (erule finite-induct, simp)
apply (simp add: card-Un-disjoint insert-partition
  finite-subset [of - (∪ (insert x F))])
done

lemma card-eq-UNIV-imp-eq-UNIV:
  assumes fin: finite (UNIV :: 'a set)
  and card: card A = card (UNIV :: 'a set)
  shows A = (UNIV :: 'a set)
proof
  show A ⊆ UNIV by simp
  show UNIV ⊆ A
    proof
      fix x
      show x ∈ A
        proof (rule ccontr)
          assume x ∉ A
          then have A ⊂ UNIV by auto
          with fin have card A < card (UNIV :: 'a set) by (fact psubset-card-mono)
          with card show False by simp
        qed
    qed
qed

The form of a finite set of given cardinality

lemma card-eq-SucD:
  assumes card A = Suc k
shows $\exists b. A = \text{insert } b B \land b \notin B \land \text{card } B = k \land (k=0 \rightarrow B=\{\})$

proof –
  have fin: finite A using assms by (auto intro: ccontr)
  moreover have card A $\neq 0$ using assms by auto
  ultimately obtain b where b: $b \in A$ by auto
  show ?thesis
    proof (intro exI conjI)
      show A = insert b (A-{b}) using b by blast
      show b $\notin$ A-{b} by blast
      show card (A-{b}) = k and k = 0 $\rightarrow$ A-{b} = {} using assms b fin
    qed
  qed

lemma card-Suc-eq:
  $(\exists b. A = \text{insert } b B \land b \notin B \land \text{card } B = k \land (k=0 \rightarrow B=\{\})))$
  apply (auto elim!: card-eq-SucD)
  apply (subst card.insert)
  apply (auto simp add: intro: ccontr)
  done

lemma card-le-Suc-iff: finite A $\Longrightarrow$
  Suc n $\leq$ card A $\Longleftarrow$ $(\exists a. A = \text{insert } a B \land a \notin B \land n \leq \text{card } B \land \text{finite } B)$
  by (fastforce simp: card-Suc-eq less-eq-nat.simps(2) insert-eq-iff
    dest: subset-singletonD split: nat.splits if-splits)

lemma finite-fun-UNIVD2:
  assumes fin: finite (UNIV :: 'a $\Rightarrow$ 'b set)
  shows finite (UNIV :: 'b set)
  proof –
    from fin have $\forall$ arbitrary. finite (range ($\lambda f :: 'a \Rightarrow 'b. f$ arbitrary))
      by (rule finite-imageI)
    moreover have $\forall$ arbitrary. UNIV = range ($\lambda f :: 'a \Rightarrow 'b. f$ arbitrary)
      by (rule UNIV-eq-I) auto
    ultimately show finite (UNIV :: 'b set) by simp
  qed

lemma card-UNIV-unit [simp]: card (UNIV :: unit set) = 1
  unfolding UNIV-unit by simp

lemma infinite-arbitrarily-large:
  assumes $\neg$ finite A
  shows $\exists B. \text{finite } B \land \text{card } B = n \land B \subseteq A$
  proof (induction n)
    case 0 show ?case by (intro exI[of - {}]) auto
  next
    case (Suc n)
    then guess B .. note B = this
with \( \neg \text{finite } A \) have \( A \neq B \) by auto
with \( B \) have \( B \subseteq A \) by auto
hence \( \exists x. \ x \in A - B \) by (elim psubset-imp-ex-mem)
then guess \( x \) .. note \( x = \) this
with \( B \) have finite (insert \( x \) \( B \)) \( \land \) card (insert \( x \) \( B \)) = Suc \( n \) \( \land \) insert \( x \) \( B \) \( \subseteq A \)
by auto
thus \( \exists B. \ \text{finite } B \land \text{card } B = \text{Suc } n \land B \subseteq A \)
qed

17.5.1 Cardinality of image

lemma card-image-le: finite \( A \) \( \implies \) card (\( f \) \( \cdot \) \( A \)) \( \leq \) card \( A \)
by (induct rule: finite-induct) (simp-all add: le-SucI card-insert-if)

lemma card-image:
assumes inj-on \( f \) \( A \)
shows card (\( f \) \( \cdot \) \( A \)) = card \( A \)
proof (cases finite \( A \))
case True then show \(?\)thesis using assms by (induct \( A \)) simp-all
next
case False then have \( \neg \) finite (\( f \) \( \cdot \) \( A \)) using assms by (auto dest: finite-imageD)
with False show \(?\)thesis by simp
qed

lemma bij-betw-same-card:
bij-betw \( f \) \( A \) \( B \) \( \implies \) \( \text{card } A \) = \( \text{card } B \)
by (auto simp: bij-betw-def)

lemma endo-inj-surj: finite \( A \) \( \implies \) \( f \) \( \cdot \) \( A \) \( \subseteq \) \( A \) \( \implies \) inj-on \( f \) \( A \) \( \implies \) \( f \) \( \cdot \) \( A \) = \( A \)
by (simp add: card-seteq card-image)

lemma eq-card-imp-inj-on:
asumes finite \( A \) card(\( f \) \( \cdot \) \( A \)) = card \( A \) shows inj-on \( f \) \( A \)
using assms
proof (induct rule:finite-induct)
case empty show \(?\)case by simp
next
case (insert \( x \) \( A \))
then show \(?\)case using card-image-le [of \( A \) \( f \)]
by (simp add: card-insert-if split: if-splits)
qed

lemma inj-on-iff-eq-card: finite \( A \) \( \implies \) inj-on \( f \) \( A \) \( \iff \) card(\( f \) \( \cdot \) \( A \)) = card \( A \)
by (blast intro: card-image eq-card-imp-inj-on)

lemma card-inj-on-le:
asumes inj-on \( f \) \( A \) \( f \) \( \cdot \) \( A \) \( \subseteq \) \( B \) finite \( B \) shows card \( A \) \( \leq \) card \( B \)
proof
have finite \( A \) using assms
by (blast intro: finite-imageD dest: finite-subset)
then show ?thesis using assms
  by (force intro: card-mono simp: card-image [symmetric])
qed

lemma card-bij-eq:
  [| inj-on f A; f ' A ⊆ B; inj-on g B; g ' B ⊆ A; finite A; finite B |] ==> card A = card B
by (auto intro: le-antisym card-inj-on-le)

lemma bij-betw-finite:
  assumes bij-betw f A B
  shows finite A <-> finite B
using assms unfolding bij-betw-def
using finite-imageD [of f A] by auto

lemma inj-on-finite:
  assumes inj-on f A f ' A ⊆ B finite B
  shows finite A
using assms finite-imageD finite-subset by blast

17.5.2 Pigeonhole Principles

lemma pigeonhole: card A > card(f ' A) ==> ~ inj-on f A
by (auto dest: card-image less-irrefl-nat)

lemma pigeonhole-infinite:
  assumes ~ finite A and finite(f'A)
  shows EX a0:A. ~finite{a:A. f a = f a0}
proof -
  have finite(f'A) ==> ~ finite A ==> EX a0:A. ~finite{a:A. f a = f a0}
  proof (induct f'A arbitrary: A rule: finite-induct)
    case empty
    thus ?case by simp
next
  case (insert b F)
  show ?case
  proof cases
    assume finite{a:A. f a = b}
    hence ~ finite(A - {a:A. f a = b}) using (~ finite A) by simp
    also have A - {a:A. f a = b} = {a:A. f a ≠ b} by blast
    finally have ~ finite({a:A. f a ≠ b}) .
    from insert(3)[OF - this]
    show ?thesis using insert(2,4) by simp (blast intro: rev-finite-subset)
next
  assume 1: ~finite{a:A. f a = b}
  hence {a ∈ A. f a = b} ≠ {} by force
  thus ?thesis using 1 by blast
qed
qed
from this[OF assms(2,1)] show ?thesis .
THEORY “Finite-Set”

qed

lemma pigeonhole-infinite-rel:
assumes ∼ finite A and finite B and ALL a:A. EX b:B. R a b
shows EX b:B. ∼ finite{a:A. R a b}
proof –
let ?F = %a. {b:B. R a b}
from finite-Pow-iff[THEN iffD2, OF finite B]
have finite(?F ‘ A) by(blast intro: rev-finite-subset)
from pigeonhole-infinite[where f = ?F, OF assms(1) this]
obtain a0 where a0∈A and 1: ∼ finite {a∈A. ?F a = ?F a0} ..
obtain b0 where b0 : B and R a0 b0 using ⟨a0:A⟩ assms(3) by blast
{ assume finite{a:A. R a b0}
then have finite {a∈A. ?F a = ?F a0}
using ⟨b0 : B⟩ ⟨R a0 b0⟩ by(blast intro: rev-finite-subset)
}
with 1 ⟨b0 : B⟩ show ?thesis by blast
qed

17.5.3 Cardinality of sums

lemma card-Plus:
assumes finite A and finite B
shows card (A <+> B) = card A + card B
proof –
have Inl‘A ∩ Inr‘B = {} by fast
with assms show ?thesis
unfolding Plus-def
by (simp add: card-Un-disjoint card-image)
qed

lemma card-Plus-conv-if:
   card (A <+> B) = (if finite A ∧ finite B then card A + card B else 0)
by (auto simp add: card-Plus)

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma dvd-partition:
assumes f: finite (∪ C) and ∀ c∈C. k dvd c ∀ c1∈C. ∀ c2∈C. c1 ≠ c2 −→
c1 ∩ c2 = {}
shows k dvd card (∪ C)
proof –
have finite C
by (rule finite-UnionD [OF f])
then show ?thesis using assms
proof (induct rule: finite-induct)
case empty show ?case by simp
next
case (insert c C)
then show ?case
apply simp
apply \( \text{subst \ card-Un-disjoint} \)
apply \( \text{(auto simp add: disjoint-eq-subset-Compl)} \)
done
qed
qed

17.5.4 Relating injectivity and surjectivity

lemma \( \text{finite-surj-inj} \): assumes finite \( A \subseteq f \cdot A \) shows inj-on \( f \cdot A \)
proof
  have \( f \cdot A = A \)
  by \( \text{(rule card-seteq [THEN sym]) (auto simp add: assms card-image-le)} \)
  then show ?thesis using assms
  by \( \text{(simp add: eq-card-imp-inj-on)} \)
qed

lemma \( \text{finite-UNIV-surj-inj} \): fixes \( f \) shows finite \( \text{UNIV} \) = \( \Rightarrow \) surj \( f \) = \( \Rightarrow \) inj \( f \)
by \( \text{(blast intro: finite-surj-inj subset-UNIV)} \)

lemma \( \text{finite-UNIV-inj-surj} \): fixes \( f \) shows finite \( \text{UNIV} \) = \( \Rightarrow \) inj \( f \) = \( \Rightarrow \) surj \( f \)
by \( \text{(fastforce simp:surj-def dest:\! \text{endo-inj-surj}} \)

corollary \( \text{infinite-UNIV-nat [iff]} \):
\( \neg \) finite \( \text{UNIV} \) = nat \( \) set
proof
  assume finite \( \text{UNIV} \) = nat \( \) set
  with finite-UNIV-inj-surj \[\text{of Suc} \]
  show False by simp \( \text{blast dest:\ Suc-neq-Zero surjD} \)
qed

lemma \( \text{infinite-UNIV-char-0} \):
\( \neg \) finite \( \text{UNIV} \) = \( \text{\'a::semiring-char-0 set} \)
proof
  assume finite \( \text{UNIV} \) = \( \text{\'a} \)
  with subset-UNIV have finite \( \text{range of-nat} \) = \( \text{\'a set} \)
  by \( \text{(rule finite-subset)} \)
  moreover have inj \( \text{of-nat} \) = nat \( \Rightarrow \) \( \text{\'a} \)
  by \( \text{(simp add: inj-on-def)} \)
  ultimately have finite \( \text{UNIV} \) = nat \( \) set
  by \( \text{(rule finite-imageD)} \)
  then show False
  by simp
qed

hide-const \( \text{(open) Finite-Set.fold} \)
18 Groups-Big: Big sum and product over finite (non-empty) sets

theory Groups-Big
imports Finite-Set
begin

18.1 Generic monoid operation over a set

no-notation times (infixl * 70)
no-notation Groups.one (1)

locale comm-monoid-set = comm-monoid
begin

interpretation comp-fun-commute f
  by default (simp add: fun-eq-iff left-commute)

interpretation comp?: comp-fun-commute f ◦ g
  by (fact comp-comp-fun-commute)

definition F :: ('b ⇒ 'a ⇒ 'a)
  where eq-fold: F g A = Finite-Set.fold (f ◦ g) 1 A

lemma infinite [simp]:
  ¬ finite A ⇒ F g A = 1
  by (simp add: eq-fold)

lemma empty [simp]:
  F g {} = 1
  by (simp add: eq-fold)

lemma insert [simp]:
  assumes finite A and x ∉ A
  shows F g (insert x A) = g x * F g A
  using assms by (simp add: eq-fold)

lemma remove:
  assumes finite A and x ∈ A
  shows F g A = g x * F g (A - {x})
proof
  from ⟨x ∈ A⟩ obtain B where A: A = insert x B and x ∉ B
    by (auto dest: mk-disjoint-insert)
  moreover from (finite A) A have finite B by simp
  ultimately show ?thesis by simp
lemma insert-remove:
assumes finite A
shows \( F \ g \ (\text{insert} \ x \ A) = g \ x \ast F \ g \ (A - \{x\}) \)
using assms by (cases \( x \in A \)) (simp-all add: remove-insert-absorb)

lemma neutral:
assumes \( \forall x \in A. \ g \ x = 1 \)
shows \( F \ g \ A = 1 \)
using assms by (induct A rule: infinite-finite-induct) simp-all

lemma neutral-const [simp]:
\( F \ (\lambda\ -. \ 1) \ A = 1 \)
by (simp add: neutral)

lemma union-inter:
assumes finite A and finite B
shows \( F \ g \ (A \cup B) \ast F \ g \ (A \cap B) = F \ g \ A \ast F \ g \ B \)
— The reversed orientation looks more natural, but LOOPS as a simprule!
using assms proof (induct A)
case empty then show ?case by simp
next
case (insert \( x \ A \)) then show ?case
  by (auto simp add: insert-absorb Int-insert-left commute \( \text{of - g x} \) assoc left-commute)
qed

corollary union-inter-neutral:
assumes finite A and finite B
and I0: \( \forall x \in A \cap B. \ g \ x = 1 \)
shows \( F \ g \ (A \cup B) = F \ g \ A \ast F \ g \ B \)
using assms by (simp add: union-inter [symmetric] neutral)

corollary union-disjoint:
assumes finite A and finite B
assumes \( A \cap B = \{\} \)
shows \( F \ g \ (A \cup B) = F \ g \ A \ast F \ g \ B \)
using assms by (simp add: union-inter-neutral)

lemma union-diff2:
assumes finite A and finite B
shows \( F \ g \ (A \cup B) = F \ g \ (A - B) \ast F \ g \ (B - A) \ast F \ g \ (A \cap B) \)
proof
  have \( A \cup B = A - B \cup (B - A) \cup A \cap B \)
  by auto
  with assms show ?thesis by simp (subst union-disjoint, auto)+
qed
lemma **subset-diff**:
assumes \( B \subseteq A \) and finite \( A \)
shows \( F \ g \ A = F \ g \ (A - B) * F \ g \ B \)
proof –
  from assms have finite \((A - B)\) by auto
moreover from assms have finite \( B \) by (rule finite-subset)
moreover from assms have \((A - B) \cap B = \{\}\) by auto
ultimately have \( F \ g \ (A - B \cup B) = F \ g \ (A - B) * F \ g \ B \) by (rule union-disjoint)
moreover from assms have \( A \cup B = A \) by auto
ultimately show \(?thesis by simp \)
qed

lemma **setdiff-irrelevant**:
assumes finite \( A \)
shows \( F \ g \ (A - \{x. \ g x = z\}) = F \ g \ A \)
using assms by (induct \( A \)) (simp-all add: insert-Diff-if)

lemma **not-neutral-contains-not-neutral**:
assumes \( F \ g \ A \neq z \)
obtains \( a \) where \( a \in A \) and \( g \ a \neq z \)
proof –
  from assms have \( \exists a \in A. \ g \ a \neq z \)
proof (induct \( A \)) (simp-all)
  case (insert \( a \) \( A \))
  then show \(?case by simp \)
qed simp-all
with that show \( \text{thesis by blast} \)
qed

lemma **reindex**:
assumes inj-on \( h \ A \)
shows \( F \ g \ (h \ A) = F \ (g \circ h) \ A \)
proof (cases finite \( A \))
  case True
  with assms show \(?thesis by (simp add: eq-fold fold-image comp-assoc) \)
next
  case False with assms have \( \neg \) finite \((h \ A)\) by (blast dest: finite-imageD)
with False show \(?thesis by simp \)
qed

lemma **cong**:
assumes \( A = B \)
assumes \( g\)-h: \( \forall x. \ x \in B \implies g \ x = h \ x \)
shows \( F \ g \ A = F \ h \ B \)
using \( g\)-h unfolding \((A = B)\)
by (induct B rule: infinite-finite-induct) auto

lemma **strong-cong** [cong]:
assumes $A = B \land x, x \in B \Rightarrow g x = h x$
shows $F (\lambda x. g x) A = F (\lambda x. h x) B$
by (rule cong) (insert assms, simp-all add: simp-implies-def)

lemma reindex-cong:
assumes inj-on $l$ $B$
assumes $A = l \cdot B$
assumes $\forall x. x \in B \Rightarrow g (l x) = h x$
shows $F g A = F h B$
using assms by (simp add: reindex)

lemma UNION-disjoint:
assumes finite $I$ and $\forall i \in I. \ finite (A i)$
and $\forall i \in I. \forall j \in I. \ i \neq j \rightarrow A i \cap A j = \{}$
shows $F g (\bigcup I A) = F (\lambda x. F g (A x)) I$
apply (insert assms)
apply (induct rule: finite-induct)
apply simp
apply atomize
apply (subgoal-tac $\forall i \in Fa. \ x \neq i$)
prefer 2 apply blast
apply (subgoal-tac $A x \ Int \ UNION Fa A = \{}$)
prefer 2 apply blast
apply (simp add: union-disjoint)
done

lemma Union-disjoint:
assumes $\forall A \in C. \ finite A \ \forall A \in C. \forall B \in C. \ A \neq B \rightarrow A \cap B = \{}$
shows $F g (\bigcup C) = (F \circ F) g C$
proof cases
assume finite $C$
from UNION-disjoint [OF this assms]
show ?thesis by simp
qed (auto dest: finite-UnionD intro: infinite)

lemma distrib:
$F (\lambda x. g x * h x) A = F g A * F h A$
using assms by (induct $A$ rule: infinite-finite-induct) (simp-all add: assoc commute left-commute)

lemma Sigma:
finite $A \Rightarrow \forall x \in A. \ finite (B x) \ \Rightarrow F (\lambda x. F (g x) (B x)) A = F (\text{split } g)$
(SIGMA $x:A. B x$)
apply (subst Sigma-def)
apply (subst UNION-disjoint, assumption, simp)
apply blast
apply (rule cong)
apply rule
apply (simp add: fun-eq-iff)
apply (subst UNION-disjoint, simp, simp)
apply blast
apply (simp add: comp-def)
done

lemma related:
  assumes Re: R 1 1
  and Rop: ∀x1 y1 x2 y2. R x1 x2 ∧ R y1 y2 → R (x1 * y1) (x2 * y2)
  and fS: finite S and Rfg: ∀x ∈ S. R (h x) (g x)
  shows R (F h S) (F g S)
using fS by (rule finite-subset-induct) (insert assms, auto)

lemma mono-neutral-cong-left:
  assumes finite T and S ⊆ T and ∀i ∈ T − S. h i = 1
  and ∀x. x ∈ S ⇒ g x = h x shows F g S = F h T
proof-
  have eq: T = S ∪ (T − S) using (S ⊆ T) by blast
  have d: S ∩ (T − S) = {} using (S ⊆ T) by blast
  from (finite T; S ⊆ T) have f: finite S finite (T − S)
  by (auto intro: finite-subset)
  show ?thesis using assms(4)
  by (simp add: union-disjoint [OF f d, unfolded eq [symmetric]] neutral [OF assms(3)])
qed

lemma mono-neutral-cong-right:
  [ finite T; S ⊆ T; ∀i ∈ T − S. g i = 1; ∀x. x ∈ S ⇒ g x = h x ]
  ⊨ F g T = F g S
  by (auto intro: mono-neutral-cong-left [symmetric])

lemma mono-neutral-left:
  [ finite T; S ⊆ T; ∀i ∈ T − S. g i = 1 ] ⊨ F g S = F g T
  by (blast intro: mono-neutral-cong-left)

lemma mono-neutral-right:
  [ finite T; S ⊆ T; ∀i ∈ T − S. g i = 1 ] ⊨ F g T = F g S
  by (blast intro: mono-neutral-left [symmetric])

lemma reindex-bij-betw: bij-betw h S T ⇒ F (λx. g (h x)) S = F g T
by (auto simp: bij-betw-def reindex)

lemma reindex-bij-witness:
  assumes witness:
    ∀a. a ∈ S ⇒ i (j a) = a
    ∀a. a ∈ S ⇒ j a ∈ T
    ∀b. b ∈ T ⇒ j (i b) = b
    ∀b. b ∈ T ⇒ i b ∈ S
  assumes eq:
    ∀a. a ∈ S ⇒ h (j a) = g a
shows $F\ g\ S = F\ h\ T$

proof
  have bij-betw j S T
    using bij-betw-byWitness[where A=S and f=j and f'=i and A'=T] witness
  moreover have $F\ g\ S = F\ (\lambda\ x.\ h\ (j\ x))\ S$
    by (intro cong) (auto simp: eq)
  ultimately show ?thesis
    by (simp add: reindex-bij-betw)
qed

lemma reindex-bij-betw-not-neutral:
  assumes fin: finite S' finite T'
  assumes bij: bij-betw h $(S - S')$ $(T - T')$
  assumes nn:
    $\forall a.\ a\in S' \implies g\ (h\ a) = z$
    $\forall b.\ b\in T' \implies g\ b = z$
  shows $F\ (\lambda\ x.\ g\ (h\ x))\ S = F\ g\ T$

proof
  have [simp]: finite S $\iff$ finite T
    using bij-betw-finite [OF bij] by auto
  also have ...
    by (intro mono-neutral-cong-right) auto
  also have ...
    by (rule reindex-bij-betw)
  also have ...
    using nn (finite S) by (intro mono-neutral-cong-left) auto
  finally show ?thesis
    by (simp add: reindex-bij-betw)
qed

lemma reindex-nontrivial:
  assumes fin: finite A
  and nz: $\forall x\ y.\ x\in A \implies y\in A \implies x \neq y \implies h\ x = h\ y \implies g\ (h\ x) = 1$
  shows $F\ g\ (h\ A) = F\ (g\circ h)\ A$

proof (subst reindex-bij-betw-not-neutral [symmetric])
  show bij-betw h $(A - \{x\in A.\ (g\circ h)\ x = 1\})$ $(h\ - A - h\ - \{x\in A.\ (g\circ h)\ x = 1\})$
    using nz by (auto intro!: inj-onI simp: bij-betw-def)
qed (insert [finite A], auto)

lemma reindex-bij-witness-not-neutral:
  assumes fin: finite S' finite T'
  assumes witness:
    $\forall a.\ a\in S - S' \implies i\ (j\ a) = a$
\[ a \in S - S' \implies f a \in T - T' \]
\[ b \in T - T' \implies j (i b) = b \]
\[ i b \in S - S' \]
\[ a \in S' \implies g a = z \]
\[ b \in T' \implies h b = z \]

**lemma delta:**

**assumes nn:**
\[ a \in S \implies h (j a) = g a \]
**shows** \( F g S = F h T \)

**proof**

- **have bij:** \( \text{bij-betw } j (S - (S' \cap S)) (T - (T' \cap T)) \)**
  - **using witness by** \( \text{intro bij-betw-byWitness[where } f' = i] \) auto
- **have F-eq:** \( F g S = F (\lambda x. h (j x)) S \)**
  - **by** \( \text{intro cong} \) \( \text{auto simp: eq} \)
- **show ?thesis**
  - **unfolding F-eq using** \( \text{fin nn eq} \)
  - **by** \( \text{intro reindex-bij-betw-not-neutral[OF - - bij]} \) auto

**qed**

**lemma delta':**

**assumes fS:** \( \text{finite } S \)
**shows** \( F (\lambda k. \text{if } a = k \text{ then } b k \text{ else } 1) S = (\text{if } a \in S \text{ then } b a \text{ else } 1) \)

**proof**

- **let \( ?f = (\lambda k. \text{if } k = a \text{ then } b k \text{ else } 1) \)**
  - **{ assume a: } a \notin S **
    - **hence \( \forall k \in S. \text{if } k = 1 \text{ by } simp \)**
    - **hence ?thesis using a by simp }**
  - **moreover**
    - **{ assume a: } a \in S **
    - **let \( ?A = S - \{a\} \)**
    - **let \( ?B = \{a\} \)**
    - **have eq: \( S = ?A \cup ?B \) using a by blast**
    - **have dj: \( ?A \cap ?B = \{\} \) by simp**
    - **from fS have fAB: \( \text{finite } ?A \text{ finite } ?B \) by auto**
    - **have \( F ?f S = F ?f ?A \) by simp**
    - **using union-disjoint [OF fAB dj, of ?f, unfolded eq [symmetric]]**
    - **then have ?thesis using a by simp }**
  - ultimately **show ?thesis by blast**

**qed**

**lemma If-cases:**

**fixes P :: 'b \Rightarrow \text{bool} and g h :: 'b \Rightarrow 'a**
**assumes fA:** \( \text{finite } A \)
shows $F \left( \lambda x. \text{if } P x \text{ then } h x \text{ else } g x \right) A =$

$F h (A \cap \{x. P x\}) * F g (A \cap \{-x. P x\})$

proof

have a: $A = A \cap \{x. P x\} \cup A \cap \{-x. P x\}$

$(A \cap \{x. P x\}) \cap (A \cap \{-x. P x\}) = \{\}$

by blast+

from fA

have f: finite $(A \cap \{x. P x\})$ finite $(A \cap \{-x. P x\})$ by auto

let $?g = \lambda x. \text{if } P x \text{ then } h x \text{ else } g x$

from union-disjoint [OF f a(2), af $?g$ a(1)]

show $?\text{thesis}$

by (subst (1 2 cong) simp-all)

qed

**lemma** cartesian-product:

$F \left( \lambda x. F (g x) B \right) A = F (\text{split } g) (A <\leftrightarrow> B)$

apply (rule sym)

apply (cases finite A)

apply (cases finite B)

apply (simp add: Sigma)

apply (cases A=$\{\}$, simp)

apply simp

apply (auto intro: infinite dest: finite-cartesian-productD2)

apply (cases B=$\{\}$) apply (auto intro: infinite dest: finite-cartesian-productD1)

done

**lemma** inter-restrict:

assumes finite A

shows $F g (A \cap B) = F (\lambda x. \text{if } x \in B \text{ then } g x \text{ else } 1) A$

proof

let $?g = \lambda x. \text{if } x \in A \cap B \text{ then } g x \text{ else } 1$

have $\forall i \in A - A \cap B. (\text{if } i \in A \cap B \text{ then } g i \text{ else } 1) = 1$

by simp

moreover have $A \cap B \subseteq A$ by blast

ultimately have $F ?g (A \cap B) = F ?g A$ using (finite A)

by (intro mono-neutral-left) auto

then show $?\text{thesis}$ by simp

qed

**lemma** inter-filter:

finite A $\implies F g \{x \in A. P x\} = F (\lambda x. \text{if } P x \text{ then } g x \text{ else } 1) A$

by (simp add: inter-restrict [symmetric, of A \{x. P x\} g, simplified mem-Collect-eq] Int-def)

**lemma** Union-comp:

assumes $\forall A \in B. \text{finite } A$ and $\bigwedge A1 A2 x. A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2$

$\implies g x = 1$

shows $F g (\bigcup B) = (F \circ F) g B$
using assms proof (induct B rule: infinite-finite-induct)

  case (infinite A)
  then have ¬ finite (∪ A) by (blast dest: finite-UnionD)
  with infinite show ?case by simp

next

  case empty then show ?case by simp

next

  case (insert A B)
  then have finite A finite B finite (∪ B) A ⊈ B
    and H: F g (∪ B) = (F o F) g B by auto
  then have F g (A ∪ ∪ B) = F g A * F g (∪ B)
    by (simp add: union-inter-neutral)
  with ⟨finite B⟩ ⟨A /∈ B⟩
    show ?case
      by (simp add: H)
  qed

lemma commute:
  F (λx. F (g x) i) A = F (λi. F (λi. g i j) A) B

unfolding cartesian-product
  by (rule reindex-bij-witness [where i = λ(i, j). (j, i)] and j = λ(i, j). (j, i))
  auto

lemma commute-restrict:
  finite A → finite B
    → F (λx. F (g x) i) A = F (λy. F (λx. g x y) i) A

proof
  have A ⊔ B = Inl ' A ⊔ Inr ' B by auto
  moreover from fin have finite (Inl ' A :: ('b + 'c) set) finite (Inr ' B :: ('b + 'c) set)
    by (auto intro: inj-onI)
  ultimately show ?thesis
    using fin
      by (simp add: union-disjoint reindex)
  qed

end

notation times (infixl * 70)
notation Groups.one (1)
18.2 Generalized summation over a set

context comm-monoid-add

begin

definition setsum :: ('b ⇒ 'a) ⇒ 'b set ⇒ 'a where
  setsum = comm-monoid-set.F plus 0

sublocale setsum!: comm-monoid-set plus 0
  where
  comm-monoid-set.F plus 0 = setsum

proof
  show comm-monoid-set plus 0 ..
  then interpret setsum!: comm-monoid-set plus 0
  from setsum-def show comm-monoid-set.F plus 0 = setsum by rule
qed

abbreviation
  Setsum (∑ – [1000] 999) where
  ∑ A ≡ setsum (%x. x) A

end

Now: lot’s of fancy syntax. First, setsum (λx. e) A is written \( \sum x \in A \cdot e \).

syntax
  -setsum :: pttrn => 'a set => 'b => 'b::comm-monoid-add ((3SUM : -) [0, 51, 10] 10)
syntax (xsymbols)
  -setsum :: pttrn => 'a set => 'b => 'b::comm-monoid-add ((3∑ : -) [0, 51, 10] 10)
syntax (HTML output)
  -setsum :: pttrn => 'a set => 'b => 'b::comm-monoid-add ((3∑ : -) [0, 51, 10] 10)

translations — Beware of argument permutation!
  SUM i:A. b == CONST setsum (%i. b) A
  \( \sum i \in A \cdot b \) == CONST setsum (%i. b) A

Instead of \( \sum x \in \{x. P\} \cdot e \) we introduce the shorter \( \sum x|P\cdot e \).

syntax
  -qsetsum :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3SUM - | -) [0,0,10] 10)
syntax (xsymbols)
  -qsetsum :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∑ - | -) [0,0,10] 10)
syntax (HTML output)
  -qsetsum :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∑ - | -) [0,0,10] 10)

translations
  SUM x|P. t => CONST setsum (%x. t) {x. P}
  \( \sum x|P\cdot t \) => CONST setsum (%x. t) {x. P}
print-translation ⟨⟨
let
  fun setsum-tr'[Abs (x, Tx, t), Const (@{const-syntax Collect}, _) $ Abs (y, Ty, P')] =
    if x <> y then raise Match
    else
      let
        val x' = Syntax-Trans.mark-bound-body (x, Tx);
        val t' = subst-bound (x', t);
        val P' = subst-bound (x', P);
      in
        Syntax.const @{
          syntax-const -qsetsum}
        $ Syntax-Trans.mark-bound-abs
        (x, Tx) $ P' $ t'
      end
    in
      case setsum-tr' - = raise Match;
    end
⟩⟩

TODO generalization candidates

lemma setsum-image-gen:
  assumes fS: finite S
  shows setsum g S = setsum (λy. setsum g {x. x ∈ S ∧ f x = y}) (f ' S)
proof−
  { fix x assume x ∈ S then have {y. y ∈ f'S ∧ f x = y} = {f x} by auto }
  hence setsum g S = setsum (λx. setsum (λy. g x) {y. y ∈ f'S ∧ f x = y}) S
    by simp
  also have ... = setsum (λy. setsum g {x. x ∈ S ∧ f x = y}) (f ' S)
    by (rule setsum.commute-restrict [OF fS finite-imageI [OF fS]])
finally show ?thesis .
qed

18.2.1 Properties in more restricted classes of structures

lemma setsum-Un: finite A ==> finite B ==> (setsum f (A Un B) :: 'a :: ab-group-add) =
  setsum f A + setsum f B - setsum f (A Int B)
by (subst setsum.union-inter [symmetric], auto simp add: algebra-simps)

lemma setsum-Un2:
  assumes finite (A ∪ B)
  shows setsum f (A ∪ B) = setsum f (A - B) + setsum f (B - A) + setsum f (A ∩ B)
proof −
  have A ∪ B = A - B ∪ (B - A) ∪ A ∩ B
    by auto
  with assms show ?thesis by simp (subst setsum.union-disjoint, auto)+
qed
lemma setsum-diff1: finite A →
(setsum f (A − {a})) :: ('a::ab-group-add) =
(if a:A then setsum f A − f a else setsum f A)
by (erule finite-induct) (auto simp add: insert-Diff-if)

lemma setsum-diff:
assumes le: finite A B ⊆ A
shows setsum f (A − B) = setsum f A − ((setsum f B)::('a::ab-group-add))
proof –
  from le have finiteB: finite B using finite-subset by auto
  show ?thesis using finiteB le
  proof
      case empty
      thus ?case by auto
    next
      case (insert x F)
      thus ?case using le finiteB
        by (simp add: Diff-insert[where a=x and B=F] setsum-diff1 insert-absorb)
  qed
qed

lemma setsum-mono:
assumes le: ∀i. i∈K ⇒ f i :: ('a::comm-monoid-add, ordered-ab-semigroup-add))
shows (∑ i∈K. f i) ≤ (∑ i∈K. g i)
proof (cases finite K)
  case True
  thus ?thesis using le
  proof
    case empty
    thus ?case by simp
  next
    case insert
    thus ?case using add-mono by fastforce
  qed
next
  case False then show ?thesis by simp
qed

lemma setsum-strict-mono:
fixes f :: 'a ⇒ 'b::{ordered-cancel-ab-semigroup-add,comm-monoid-add}
assumes finite A A ≠ {}
  and ∀x. x:A ⇒ f x < g x
shows setsum f A < setsum g A
using assms
proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case insert thus ?case by (auto simp: add-strict-mono)
qed
lemma setsum-strict-mono-ex1:
fixes f :: 'a ⇒ 'b::{comm-monoid-add, ordered-cancel-ab-semigroup-add}
assumes finite A and ALL x:A. f x ≤ g x and EX a:A. f a < g a
shows setsum f A < setsum g A
proof
  from assms(3) obtain a where a: a:A f a < g a by blast
  have setsum f A = setsum f ((A−{a}) ∪ {a})
    by(simp add:insert-absorb[OF (a:A)])
  also have ... = setsum f (A−{a}) + setsum f {a}
    by(finite A) by(subst setsum.union-disjoint) auto
  also have setsum f (A−{a}) ≤ setsum g (A−{a})
    by(rule setsum-mono)(simp add: assms(2))
  also have setsum f {a} < setsum g {a} using a by simp
  also have setsum g (A − {a}) + setsum g {a} = setsum g((A−{a}) ∪ {a})
    by(finite A) by(subst setsum.union-disjoint[symmetric]) auto
  also have ... = setsum g A by(simp add:insert-absorb[OF (a:A)])
  finally show ?thesis by (auto simp add: add-right-mono add-strict-left-mono)
qed

lemma setsum-negf:
  setsum (%x. − (f x)::'a::ab-group-add) A = − setsum f A
proof (cases finite A)
  case True thus ?thesis by (induct set: finite) auto
next
case False thus ?thesis by simp
qed

lemma setsum-subtractf:
  setsum (%x. (f x):'a::ab-group-add) − g x) A =
  setsum f A − setsum g A
  using setsum.distrib[of f − g A] by (simp add: setsum-negf)

lemma setsum-nonneg:
  assumes nn: ∀x∈A. (0::'a::{ordered-ab-semigroup-add,comm-monoid-add}) ≤ f x
  shows 0 ≤ setsum f A
proof (cases finite A)
  case True thus ?thesis using nn
  proof induct
    case empty then show ?case by simp
  next
case insert x F
  then have 0 + 0 ≤ f x + setsum f F by (blast intro: add-mono)
    with insert show ?case by simp
  qed
next
case False thus ?thesis by simp
qed
lemma setsum-nonpos:
assumes np: \( \forall x \in A. \, f x \leq (0::'a::{\text{ordered-ab-semigroup-add, comm-monoid-add}}) \)
shows setsum f A \leq 0
proof (cases finite A)
case True thus \(?\text{thesis}\) using np
proof induct
  case empty then show \(?\text{case}\) by simp
next
  case (insert x F)
  then have \( f x + \text{setsum} f F \leq 0 + 0 \) by (blast intro: add-mono)
  with \( \text{insert}\) show \(?\text{case}\) by simp
qed
next
case False thus \(?\text{thesis}\) by simp
qed

lemma setsum-nonneg-leq-bound:
fixes \( f :: 'a \Rightarrow 'b::{\text{ordered-ab-group-add}} \)
assumes finite s \( \land \. i : i \in s \Rightarrow f i \geq 0 \) \( (\sum i \in s. f i) = B \) \( i \in s \)
shows \( f i \leq B \)
proof
  have \( 0 \leq (\sum i \in s - \{i\}. f i) \) and \( \leq f i \)
    using assms by (auto intro!: \text{setsum-nonneg})
  moreover
  have \( (\sum i \in s - \{i\}. f i) + f i = B \)
    using assms by (simp add: \text{setsum-diff1})
  ultimately show \(?\text{thesis}\) by auto
qed

lemma setsum-nonneg-0:
fixes \( f :: 'a \Rightarrow 'b::{\text{ordered-comm-monoid-add}} \)
assumes \( \text{fin}\) \( \land \. f i \geq 0 \) \( (\sum i \in s. f i) = 0 \) \( i : i \in s \)
shows \( f i = 0 \)
using \text{setsum-nonneg-leq-bound}[OF assms] \( \text{pos}[OF i] \) by auto

lemma setsum-mono2:
fixes \( f :: 'a \Rightarrow 'b :: \text{ordered-comm-monoid-add} \)
assumes \( \text{fin}\) \( \land \. f i \leq 0 \) \( (\sum i \in s. f i) \leq 0 \) \( i : i \in s \)
shows \( \text{setsum} f A \leq \text{setsum} f B \)
proof
  have \( \text{setsum} f A \leq \text{setsum} f A + \text{setsum} f \ (B-A) \)
    by (simp add: \text{add-increasing2}[OF \text{setsum-nonneg}] \ nn \text{Ball-def})
  also have \( \ldots . \) = \( \text{setsum} f \ (A \cup (B-A)) \) using \( \text{fin}\) \( \text{finite-subset}[OF \text{sub} \text{fin}] \)
    by (simp add: \text{setsum.anion-disjoint del:Un-Diff-cancel})
  also have \( A \cup (B-A) = B \) using \( \text{sub}\) by \text{blast}
  finally show \(?\text{thesis}\)
qed
lemma setsum-le-included:
fixes f :: 'a ⇒ 'b::ordered-comm-monoid-add
assumes finite s finite t
and ∀ y∈t. 0 ≤ g y (∀ x∈s. i y = x ∧ f x ≤ g y)
shows setsum f s ≤ setsum g t
proof –
have setsum f s ≤ setsum (λ y. setsum g {x. x∈t ∧ i x = y}) s
proof (rule setsum-mono)
  fix y assume y ∈ s
  with assms obtain z where z: z ∈ t y = i z f y ≤ g z by auto
  with assms show f y ≤ setsum g {x∈t. i x = y} (i 'A y ≤ ?B y)
  using order-trans[of ?A (i z) setsum g {z} ?B (i z), intro]
  by (auto intro!: setsum-mono2)
qed
also have ... ≤ setsum (λ y. setsum g {x. x∈t ∧ i x = y}) (i 't)
  using assms(2-4) by (auto intro!: setsum-mono2 setsum-nonneg)
also have ... ≤ setsum g t
  using assms by (auto simp: setsum-image-gen[symmetric])
finally show ?thesis .
qed

lemma setsum-mono3: finite B ==⇒ A ⊆ B ==⇒
ALL x: B - A.
  0 <= ((f x)::'a::{comm-monoid-add,ordered-ab-semigroup-add}) ==⇒
  setsum f A <= setsum f B
apply (subgoal_tac setsum f B = setsum f A + setsum f (B - A))
apply (erule ssubst)
apply (subgoal_tac setsum f A + 0 <= setsum f A + setsum f (B - A))
apply simp
apply (rule add-left-mono)
apply (erule setsum-nonneg)
apply (subst setsum.union-disjoint [THEN sym])
apply (erule finite-subset, assumption)
apply (rule finite-subset)
pref 2
apply assumption
apply (auto simp add: sup-absorb2)
done

lemma setsum-right-distrib:
fixes f :: 'a ⇒ 'b::semiring-0
shows r * setsum f A = setsum (%n. r * f n) A
proof (cases finite A)
  case True
  thus ?thesis
proof induct
  case empty thus ?case by simp
next
case (insert x A) thus ?case by (simp add: distrib-left)
qed

next
case False thus ?thesis by simp
qed

lemma setsum-left-distrib:
setsum f A * (r::'a::semiring-0) = (∑ n∈A. f n * r)
proof (cases finite A)
case True
then show ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert x A) thus ?case by (simp add: distrib-right)
qed

next
case False thus ?thesis by simp
qed

lemma setsum-divide-distrib:
setsum f A / (r::'a::field) = (∑ n∈A. f n / r)
proof (cases finite A)
case True
then show ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert x A) thus ?case by (simp add: add-divide-distrib)
qed

next
case False thus ?thesis by simp
qed

lemma setsum-\text{abs}[iff]:
fixes f :: 'a => ('b::ordered-ab-group-add-abs)
shows abs (setsum f A) ≤ setsum (%i. abs(f i)) A
proof (cases finite A)
case True
thus ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert x A)
  thus ?case by (auto intro: abs-triangle-ineq order-truns)
qed

next
case False thus ?thesis by simp
qed
lemma setsum-abs-ge-zero[iff]:
  fixes f :: 'a => ('b::ordered-ab-group-add-abs)
  shows 0 ≤ setsum (%i. abs(f i)) A
proof (cases finite A)
  case True
  thus ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert x A) thus ?case by auto
qed
next
  case False thus ?thesis by simp
qed

lemma abs-setsum-abs[simp]:
  fixes f :: 'a => ('b::ordered-ab-group-add-abs)
  shows abs (∑a∈A. abs(f a)) = (∑a∈A. abs(f a))
proof (cases finite A)
  case True
  thus ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert a A)
  hence ∑a∈insert a A. |f a| = |f a| + (∑a∈A. |f a|) by simp
  also have ... = |f a| + (∑a∈A. |f a|) using insert by simp
  also have ... = |f a| + (∑a∈A. |f a|) by (simp del: abs-of-nonneg)
  also have ... = (∑a∈insert a A. |f a|) using insert by simp
  finally show ?case .
qed
next
  case False thus ?thesis by simp
qed

lemma setsum-diff1-ring: assumes finite A a ∈ A
  shows setsum f (A - {a}) = setsum f A - (f a::'a::ring)
unfolding setsum.remove [OF assms] by auto

lemma setsum-product:
  fixes f :: 'a => ('b::semiring-0)
  shows setsum f A * setsum g B = (∑i∈A. ∑j∈B. f i * g j)
by (simp add: setsum-right-distrib setsum-left-distrib) (rule setsum.commute)

lemma setsum-mult-setsum-if-inj:
  fixes f :: 'a => ('b::semiring-0)
  shows inj-on (%(a,b). f a * g b) (A × B) =>
setsum f A * setsum g B = setsum id \{ f a * g b | a b : A & B \} 
by(auto simp: setsum-product setsum.cartesian-product intro!: setsum.reindex-cong[symmetric])

lemma setsum-SucD: setsum f A = Suc n ==> EX a:A. 0 < f a 
apply (case-tac finite A) 
prefer 2 apply simp 
apply (erule rev-mp) 
apply (erule finite-induct, auto) 
done

lemma setsum-eq-0-iff [simp]:
finite F ==> (setsum f F = 0) = (ALL a:A. f a = (0::nat)) 
by (induct set: finite) auto

lemma setsum-eq-Suc0-iff: finite A ==> 
setsum f A = Suc 0 ==> (EX a:A. f a = Suc 0 & (ALL b:A. a\neq b --> f b = 0)) 
apply(erule finite-induct) 
apply (auto simp add: add-is-1) 
done

lemmas setsum-eq-1-iff = setsum-eq-Suc0-iff [simplified One-nat-def [symmetric]]

lemma setsum-Un-nat: finite A == finite B ==>
(setsum f (A Un B) :: nat) = setsum f A + setsum f B - setsum f (A Int B) 
— For the natural numbers, we have subtraction. 
by (subst setsum.union-inter [symmetric], auto simp add: algebra-simps)

lemma setsum-diff1-nat: (setsum f (A - {x}) :: nat) = 
(if a:A then setsum f A - f a else setsum f A) 
apply (case-tac finite A) 
prefer 2 apply simp 
apply (erule finite-induct) 
apply (auto simp add: insert-Diff-if) 
apply (erule finite-induct, auto) 
done

lemma setsum-diff-nat:
assumes finite B and B \subseteq A 
shows (setsum f (A - B) :: nat) = (setsum f A) - (setsum f B) 
using assms 
proof induct 
show setsum f (A - {}) = (setsum f A) - (setsum f {}) by simp 
next 
fix F x assume finF: finite F and xnotinF: x \notin F 
and xFinA: insert x F \subseteq A 
and IH: F \subseteq A ==> setsum f (A - F) = setsum f A - setsum f F 
from xnotinF xFinA have xinAF: x \in (A - F) by simp 
from xinAF have A: setsum f ((A - F) - \{x\}) = setsum f (A - F) - f x
by (simp add: setsum-diff1-nat)
from xFinA have F ⊆ A by simp
with IH have setsum f (A − F) = setsum f A − setsum f F by simp
with A have B: setsum f ((A − F) − {x}) = setsum f A − setsum f F − f x by simp
from xnotinF have A − insert x F = (A − F) − {x} by auto
with B have C: setsum f ((A − insert x F) − {x}) = setsum f A − setsum f F − f x by simp
thus setsum f (A − insert x F) = setsum f A − setsum f (insert x F) by simp qed

lemma setsum-comp-morphism:
assumes h 0 = 0 and ⋀ x y. h(x + y) = h x + h y
shows setsum (h ∘ g) A = h(setsum g A)
proof (cases finite A)
case False then show ?thesis by (simp add: assms)
next
case True then show ?thesis by (induct A) (simp-all add: assms)
qed

18.2.2 Cardinality as special case of setsum

lemma card-eq-setsum:
card A = setsum (λx. 1) A
proof –
  have plus ∘ (λ_. Suc 0) = (λ_. Suc)
    by (simp add: fun-eq-iff)
  then have Finite-Set.fold (plus ∘ (λ_. Suc 0)) = Finite-Set.fold (λ_. Suc)
    by (rule arg-cong)
  then have Finite-Set.fold (plus ∘ (λ_. Suc 0)) 0 A = Finite-Set.fold (λ_. Suc) 0 A
    by (blast intro: fun-cong)
  then show ?thesis by (simp add: card.eq-fold setsum.eq-fold)
qed

lemma setsum-constant [simp]:
(∑ x ∈ A. y) = of-nat (card A) * y
apply (cases finite A)
apply (erule finite-induct)
apply (auto simp add: algebra-simps)
done

lemma setsum-bounded:
assumes le: ⋀ i. i ∈ A ⇒ f i ≤ (K::'a::{semiring-1, ordered-ab-semigroup-add})
shows setsum f A ≤ of-nat (card A) * K
proof (cases finite A)
case True
  thus \( ?\text{thesis using le setsum-mono[where } K=\text{A and } g = \%x. \text{K}] \) by simp
next
  case False thus \( ?\text{thesis by simp} \)
qed

lemma card-UN-disjoint:
  assumes finite I and \( \forall i\in I. \text{finite } (A i) \)
  and \( \forall i\in I. \forall j\in I. i \neq j \rightarrow A i \cap A j = \{\} \)
  shows card \( (\text{UNION } I \ A) = (\sum i\in I. \text{card}(A i)) \)
proof –
  have \( (\sum i\in I. \text{card } (A i)) = (\sum i\in I. \sum x\in A i. \text{1}) \) by simp
  with assms show \( ?\text{thesis by simp add: card-eq-setsum } \text{setsum.UNION-disjoint del: setsum-constant} \)
qed

lemma card-Union-disjoint:
  finite C ==> (ALL A\in C. finite A) ==>
  (ALL A\in C. ALL B\in C. A \neq B ==> A \cap B = \{\})
  ==>
  card \( (\text{Union } C) = \text{setsum } \text{card } C \)
apply \( \text{frule card-UN-disjoint [of C id]} \)
apply simp-all
done

lemma setsum-multicount-gen:
  assumes finite s finite t \( \forall j\in t. (\text{card } \{i\in s. R i j\} = k j) \)
  shows setsum \( (\lambda i. (\text{card } \{j\in t. R i j\})) s = \text{setsum } k t \) (is \( ?l = ?r \))
proof –
  have \( ?l = \text{setsum } (\lambda i. \text{setsum } (\lambda x.1) \{j\in t. R i j\}) s \) by auto
  also have \( \ldots = ?r \) unfolding setsum.commute-restrict \([OF assms(1-2)]\)
    using assms(3) by auto
  finally show \( ?\text{thesis} \).
qed

lemma setsum-multicount:
  assumes finite S finite T \( \forall j\in T. (\text{card } \{i\in S. R i j\} = k) \)
  shows setsum \( (\lambda i. \text{card } \{j\in T. R i j\}) S = k \ast \text{card } T \) (is \( ?l = ?r \))
proof –
  have \( ?l = \text{setsum } (\lambda i. k) T \) by \( \text{rule setsum-multicount-gen} \) (auto simp: assms)
  also have \( \ldots = ?r \) by \( \text{simp add: mult.commute} \)
  finally show \( ?\text{thesis by auto} \)
qed

18.2.3 Cardinality of products

lemma card-SigmaI [simp]:
  \[ \text{finite } A; \text{ALL } a\in A. \text{finite } (B a) \] 
  \( \rightarrow \text{card } (\Sigma x\in A. B x) = (\sum a\in A. \text{card } (B a)) \)
by (simp add: card-eq-setsum setsum.Sigma del: setsum-constant)
lemma card-cartesian-product: \( \text{card}(A \times B) = \text{card}(A) \times \text{card}(B) \)
by (cases finite A \& finite B)
(auto simp add: card-eq-0-iff dest: finite-cartesian-productD1 finite-cartesian-productD2)

lemma card-cartesian-product-singleton: \( \text{card}(\{x\} \times A) = \text{card}(A) \)
by (simp add: card-cartesian-product)

18.3 Generalized product over a set
context comm-monoid-mult
begin

definition setprod :: \( 'b \Rightarrow 'a \Rightarrow 'b \) set \Rightarrow 'a
where
setprod = comm-monoid-set.F times 1

sublocale setprod!: comm-monoid-set times 1
where
comm-monoid-set.F times 1 = setprod
proof –
  show comm-monoid-set times 1 ..
  then interpret setprod!: comm-monoid-set times 1 .
  from setprod-def show comm-monoid-set.F times 1 = setprod by rule
qed

abbreviation Setprod (\( \prod \cdot [1000] 999 \)) where
\( \prod A \equiv \text{setprod} (\lambda x. x) A \)
end

syntax
-setprod :: ptttrn => 'a set => 'b => 'b::comm-monoid-mult ((3PROD ::- ::-)
[0, 51, 10] 10)
syntax (xsymbols)
-setprod :: ptttrn => 'a set => 'b => 'b::comm-monoid-mult ((3\prod \cdot \cdot ::- ::-) [0, 51, 10] 10)
syntax (HTML output)
-setprod :: ptttrn => 'a set => 'b => 'b::comm-monoid-mult ((3\prod \cdot \cdot ::- ::-) [0, 51, 10] 10)

translations — Beware of argument permutation!
\( PROD i. A. b \equiv \text{CONST setprod} (%i. b) A \)
\( \prod i \in A. b \equiv \text{CONST setprod} (%i. b) A \)

Instead of \( \prod x\in\{x. P\}. e \) we introduce the shorter \( \prod x | P. e \).
syntex
-ssetprod : pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3PROD - |. -. /) [0,0,10] 10)
syntex (symbols)
-ssetprod : pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - |. -. /) [0,0,10] 10)
syntex (HTML output)
-ssetprod : pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - |. -. /) [0,0,10] 10)

translations
PROD x [P. t] = CONST setprod (%x. t) {x. P}
Π x[P. t] = CONST setprod (%x. t) {x. P}

18.3.1 Properties in more restricted classes of structures

lemma setprod-zero:
finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
apply (induct set: finite, force, clarsimp)
apply (erule disjE, auto)
done

lemma setprod-zero-iff[simp]: finite A ==> (setprod f A = (0::'a::{comm-semiring-1,no-zero-divisors})) =
(EX x: A. f x = 0)
by (erule finite-induct, auto simp:no-zero-divisors)

lemma setprod-Un: finite A ==> finite B ==> (ALL x: A Int B. f x ≠ 0) ==> (setprod f (A Un B) :: 'a::{field}) =
(setprod f A * setprod f B / setprod f (A Int B)
by (subst setprod.union-inter [symmetric], auto)

lemma setprod-nonneg [rule-format]: (ALL x: A. (0::'a::linordered-semidom) ≤ f x) ==> 0 ≤ setprod f A
by (cases finite A, induct set: finite, simp-all)

lemma setprod-pos [rule-format]: (ALL x: A. (0::'a::linordered-semidom) < f x)
--- 0 < setprod f A
by (cases finite A, induct set: finite, simp-all)

lemma setprod-diff1: finite A ==> f a ≠ 0 ==> (setprod f (A - {a}) :: 'a::{field}) =
(if a:A then setprod f A / f a else setprod f A)
by (erule finite-induct) (auto simp add: insert-Diff-if)

lemma setprod-inversef:
fixes f :: 'b ⇒ 'a::field-inverse-zero
shows finite A ==> setprod (inverse o f) A = inverse (setprod f A)
by (erule finite-induct) auto

lemma setprod-dividef:
fixes f :: 'b ⇒ 'a::field-inverse-zero
shows finite A
  ==> setprod (%x. f x / g x) A = setprod f A / setprod g A
apply (subgoal-tac
  setprod (%x. f x / g x) A = setprod (%x. f x * (inverse o g) x) A)
apply (erule ssubst)
apply (subst divide-inverse)
apply (subst setprod. distrib)
apply (subst setprod-inversef, assumption+, rule refl)
apply (rule setprod.cong, rule refl)
apply (subst divide-inverse, auto)
done

lemma setprod-dvd-setprod [rule-format]:
  (ALL x : A. f x dvd g x) ==> setprod f A dvd setprod g A
apply (cases finite A)
apply (induct set: finite)
apply (auto simp add: dvd-def)
apply (rule-tac x = k * ka in exI)
apply (simp add: algebra-simps)
done

lemma setprod-dvd-setprod-subset:
  finite B ==> A <= B ==> setprod f A dvd setprod f B
apply (subgoal-tac setprod f B = setprod f A * setprod f (B - A))
apply (unfold dvd-def, blast)
apply (subst setprod.union-disjoint [symmetric])
apply (auto elim: finite-subset intro: setprod.cong)
done

lemma setprod-dvd-setprod-subset2:
  finite B ==> A <= B ==> ALL x : A. (f x::'a::comm-semiring-1) dvd g x ==> setprod f A dvd setprod g B
apply (rule dvd-trans)
apply (rule setprod-dvd-setprod, erule (1) bspec)
apply (erule (1) setprod-dvd-setprod-subset)
done

lemma dvd-setprod: finite A ==> i:A ==> (f i::'a::comm-semiring-1) dvd setprod f A
by (induct set: finite) (auto intro: dvd-mult)

lemma dvd-setsum [rule-format]: (ALL i : A. d dvd f i) ==> (d::'a::comm-semiring-1) dvd (SUM x : A. f x)
apply (cases finite A)
apply (induct set: finite)
apply auto
done

lemma setprod-mono:
fixes \( f :: 'a \Rightarrow 'b :: \text{linordered-semidom} \)
assumes \( \forall i \in A. \ 0 \leq f i \land f i \leq g i \)
shows \( \text{setprod } f A \leq \text{setprod } g A \)
proof (cases finite \( A \))
case True
hence \( \text{thesis setprod } f A \geq 0 \) using subset_refl[OF \( A \)]
proof (induct \( A \) rule: finite-subset-induct)
case (insert \( a \) \( F \))
thus \( \text{setprod } f (\text{insert } a F) \leq \text{setprod } g (\text{insert } a F) \)
unfolding setprod.insert[OF insert(1,3)]
using assms[rule-format,OF insert(2)] insert
by (auto intro: mult_mono)
qed auto
thus \( \text{thesis} \) by simp
qed auto

lemma abs-setprod:
fixes \( f :: 'a \Rightarrow 'b :: \{\text{linordered-field}, \text{abs}\} \)
shows \( \text{abs } (\text{setprod } f A) = \text{setprod } (\lambda x. \text{abs } (f x)) A \)
proof (cases finite \( A \))
case True thus \( \text{thesis} \)
by (induct (auto simp add: field_simps abs_mult))
qed auto

lemma setprod-eq-1-iff [simp]:
finite \( F \) \implies \( \text{setprod } f F = 1 \) \iff \( (\forall i : F. f a = (1 :: \text{nat})) \)
by (induct set: finite) auto

lemma setprod-pos-nat:
finite \( S \) \implies \( \text{setprod } f S > 0 \)
using setprod-zero-iff by(simp del:neq0_conv add:neq0_conv[symmetric])

lemma setprod-pos-nat-iff[simp]:
finite \( S \) \implies \( \text{setprod } f S > 0 \) \iff \( (\forall x : S. f x > (0 :: \text{nat})) \)
using setprod-zero-iff by(simp del:neq0_conv add:neq0_conv[symmetric])

lemma (in ordered-comm-monoid-add) setsum-pos:
finite \( I \) \implies \( I \neq \{\} \implies (\forall i : I. f i \leq 0 \implies 0 < f i) \implies 0 < \text{setsum } f I \)
by (induct \( I \) rule: finite-ne-induct) (auto intro: add-pos-pos)

end

19 Relation: Relations – as sets of pairs, and binary predicates

theory Relation
imports Finite-Set
begin
A preliminary: classical rules for reasoning on predicates

19.1 Fundamental

19.1.1 Relations as sets of pairs

type-synonym 'a rel = ('a * 'a) set

lemma subrelI: — Version of subsetI for binary relations
19.1.2 Conversions between set and predicate relations

**Lemma** pred-equals-eq: — Version of lfp-induct for binary relations

\[
\forall a, b \in \text{lfp } f \implies \text{mona } f \implies \forall (x, y). \ (x, y) \in f \implies P \ x \ y \implies P \ a \ b \\
\text{using lfp-induct-set \ of \ } (a, b) \ f \ \text{case-prod } P \ \text{by auto}
\]

**Lemma** pred-equals-eq2 [pred-set-conv]: \((\lambda x. \ x \in R) = (\lambda x. \ x \in S) \iff R = S\)

**Lemma** pred-equals-eq3 [pred-set-conv]: \((\lambda x. \ (x, y) \in R) = (\lambda x. \ (x, y) \in S) \iff R = S\)

**Lemma** pred-subset-eq [pred-set-conv]: \((\lambda x. \ x \in R) \subseteq (\lambda x. \ x \in S) \iff R \subseteq S\)

**Lemma** pred-subset-eq2 [pred-set-conv]: \((\lambda x. \ (x, y) \in R) \subseteq (\lambda x. \ (x, y) \in S) \iff R \subseteq S\)

**Lemma** bot-empty-eq [pred-set-conv]: \(\bot = (\lambda x. \ x \in \{\})\)

**Lemma** bot-empty-eq2 [pred-set-conv]: \(\bot = (\lambda x. \ (x, y) \in \{\})\)

**Lemma** top-empty-eq [pred-set-conv]: \(\top = (\lambda x. \ x \in \text{UNIV})\)

**Lemma** top-empty-eq2 [pred-set-conv]: \(\top = (\lambda x. \ (x, y) \in \text{UNIV})\)

**Lemma** inf-Int-eq [pred-set-conv]: \((\lambda x. \ x \in R) \cap (\lambda x. \ x \in S) = (\lambda x. \ x \in R \cap S)\)

**Lemma** inf-Int-eq2 [pred-set-conv]: \((\lambda x. \ (x, y) \in R) \cap (\lambda x. \ (x, y) \in S) = (\lambda x. \ (x, y) \in R \cap S)\)

**Lemma** sup-Un-eq [pred-set-conv]: \((\lambda x. \ x \in R) \cup (\lambda x. \ x \in S) = (\lambda x. \ x \in R \cup S)\)

**Lemma** sup-Un-eq2 [pred-set-conv]: \((\lambda x. \ (x, y) \in R) \cup (\lambda x. \ (x, y) \in S) = (\lambda x. \ (x, y) \in R \cup S)\)
lemma INF-INT-eq [pred-set-conv]: \((\prod i \in S. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcap i \in S. r i))\)
by (simp add: fun-eq-iff)

lemma INF-INT-eq2 [pred-set-conv]: \((\prod i \in S. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcap i \in S. r i))\)
by (simp add: fun-eq-iff)

lemma SUP-UN-eq [pred-set-conv]: \((\bigcup i \in S. (\lambda x. x \in r i)) = (\lambda x. x \in (\bigcup i \in S. r i))\)
by (simp add: fun-eq-iff)

lemma SUP-UN-eq2 [pred-set-conv]: \((\bigcup i \in S. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in (\bigcup i \in S. r i))\)
by (simp add: fun-eq-iff)

lemma Inf-INT-eq [pred-set-conv]: \(\bigcap S = (\lambda x. x \in \text{INTER} S \text{ Collect})\)
by (simp add: fun-eq-iff)

lemma INF-Int-eq [pred-set-conv]: \((\prod i \in S. (\lambda x. x \in r i)) = (\lambda x. x \in \bigcap S)\)
by (simp add: fun-eq-iff)

lemma Inf-INT-eq2 [pred-set-conv]: \((\prod i \in S. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in \bigcap S)\)
by (simp add: fun-eq-iff)

lemma Sup-SUP-eq [pred-set-conv]: \(\bigcup S = (\lambda x. x \in \text{UNION} S \text{ Collect})\)
by (simp add: fun-eq-iff)

lemma SUP-Sup-eq [pred-set-conv]: \((\bigcup i \in S. (\lambda x. x \in r i)) = (\lambda x. x \in \bigcup S)\)
by (simp add: fun-eq-iff)

lemma Sup-SUP-eq2 [pred-set-conv]: \((\bigcup i \in S. (\lambda x y. (x, y) \in r i)) = (\lambda x y. (x, y) \in \bigcup S)\)
by (simp add: fun-eq-iff)

19.2 Properties of relations

19.2.1 Reflexivity

definition refl-on :: 'a set ⇒ 'a rel ⇒ bool
where
\[ \text{refl-on } A \ r \iff r \subseteq A \times A \land (\forall x \in A. (x, x) \in r) \]

abbreviation \text{refl} :: 'a rel \Rightarrow bool
where
\[ \text{refl} \equiv \text{refl-on UNIV} \]

definition \text{reflp} :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where
\[ \text{reflp } r \iff (\forall x. r x x) \]

lemma \text{reflp-refl-eq [pred-set-cone]}:
\[ \text{reflp (}\lambda x y. (x, y) \in r) \iff \text{refl } r \]
by (simp add: refl-on-def reflp-def)

lemma \text{refl-onI}:
\[ r \subseteq A \times A \Longrightarrow (\forall x. x : A \Rightarrow (x, x) : r) \Longrightarrow \text{refl-on } A \ r \]
by (unfold refl-on-def) (iprover intro: ballI)

lemma \text{refl-onD}:
\[ \text{refl-on } A \ r \Longrightarrow a : A \Rightarrow (a, a) : r \]
by (unfold refl-on-def) blast

lemma \text{refl-onD1}:
\[ \text{refl-on } A \ r \Longrightarrow (x, y) : r \Rightarrow x : A \]
by (unfold refl-on-def) blast

lemma \text{refl-onD2}:
\[ \text{refl-on } A \ r \Longrightarrow (x, y) : r \Rightarrow y : A \]
by (unfold refl-on-def) blast

lemma \text{reflpI}:
\[ (\forall x. r x x) \Rightarrow \text{reflp } r \]
by (auto intro: refl-onD simp add: reflp-def)

lemma \text{reflpE}:
assumes \text{reflp } r
obtains \text{r x x}
using \text{assms} by (auto dest: refl-onD simp add: reflp-def)

lemma \text{reflpD}:
assumes \text{reflp } r
shows \text{r x x}
using \text{assms} by (auto elim: reflpE)

lemma \text{refl-on-Int}:
\[ \text{refl-on } A \ r \Longrightarrow \text{refl-on } B \ s \Longrightarrow \text{refl-on } (A \cap B) \ (r \cap s) \]
by (unfold refl-on-def) blast

lemma \text{reflp-inf}:
\[ \text{reflp } r \Longrightarrow \text{reflp } s \Longrightarrow \text{reflp } (r \cap s) \]
by (auto intro: reflpI elim: reflpE)

lemma \text{refl-on-Un}:
\[ \text{refl-on } A \ r \Longrightarrow \text{refl-on } B \ s \Longrightarrow \text{refl-on } (A \cup B) \ (r \cup s) \]
THEORY "Relation"

by (unfold refl-on-def) blast

lemma reflp-sup:
reflp r \Longrightarrow reflp s \Longrightarrow reflp (r \sqcup s)
by (auto intro: reflpI elim: reflpE)

lemma refl-on-INTER:
\( \forall x : S. \ \text{refl-on} \ (A \ x) \ (r \ x) \Longrightarrow \text{refl-on} \ (INTER \ S \ A) \ (INTER \ S \ r) \)
by (unfold refl-on-def) fast

lemma refl-on-UNION:
\( \forall x : S. \ \text{refl-on} \ (A \ x) \ (r \ x) \Longrightarrow \text{refl-on} \ (UNION \ S \ A) \ (UNION \ S \ r) \)
by (unfold refl-on-def) blast

lemma refl-on-empty [simp]: refl-on \{\} \{\}
by (simp add: refl-on-def)

lemma refl-on-def' [nitpick-unfold, code]:
refl-on A r \iff \( \forall a. \ (a, a) \not\in r \) \[19.2.2 Irreflexivity\]
definition irrefl :: \('a \ rel \Rightarrow bool\)
where
irrefl r \iff \( \forall a. \ (a, a) \not\in r \) \[19.2.2 Irreflexivity\]
definition irreflp :: \('a \Rightarrow \to \ 'a \Rightarrow bool\)
where
irreflp R \iff \( \forall a. \ \neg R \ a \ a \) \[19.2.2 Irreflexivity\]
lemma irreflp-irrefl-eq [pred-set-conv]:
irreflp \( \lambda a \ b. \ (a, b) \in R \) \iff irrefl R
by (simp add: irrefl-def irreflp-def)
lemma irreflI:
\( \forall a. \ (a, a) \not\in R \) \Longrightarrow irrefl R
by (simp add: irrefl-def)
lemma irreflpI:
\( \forall a. \ \neg R \ a \ a \) \Longrightarrow irreflp R
by (fact irreflI [to-pred])
lemma irrefl-distinct [code]:
irrefl r \iff \( \forall (a, b) \in r. \ a \neq b \)
by (auto simp add: irrefl-def)

19.2.3 Asymmetry
inductive asym :: \('a \ rel \Rightarrow bool\)
where

\texttt{asymI: irrefl } R \implies (\forall a. b. (a, b) \in R \implies (b, a) \notin R) \implies asym\ R

\textbf{inductive} \texttt{asymp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool}

\texttt{where}

\texttt{asympI: irreflp } R \implies (\forall a b. R a b \Rightarrow \neg R b a) \implies asymp\ R

\textbf{lemma} \texttt{asymp-asym-eq [pred-set-conv]:}

\texttt{asymp (\lambda a b. (a, b) \in R) \leftrightarrow asym\ R}

\texttt{by (auto intro: asymI asympI elim: asym elim cases asymp cases simp add: irreflp-irrefl-eq)}

### 19.2.4 Symmetry

\textbf{definition} \texttt{sym :: 'a rel \Rightarrow bool}

\texttt{where}

\texttt{sym } r \iff (\forall x y. (x, y) \in r \implies (y, x) \in r)

\textbf{definition} \texttt{symp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool}

\texttt{where}

\texttt{symp } r \iff (\forall x y. r x y \implies r y x)

\textbf{lemma} \texttt{symp-sym-eq [pred-set-conv]:}

\texttt{symp (\lambda x y. (x, y) \in r) \leftrightarrow sym\ r}

\texttt{by (simp add: sym-def symp-def)}

\textbf{lemma} \texttt{symI:}

\texttt{(\forall a b. (a, b) \in r \implies (b, a) \in r) \implies sym\ r}

\texttt{by (unfold sym-def) sprover}

\textbf{lemma} \texttt{sympI:}

\texttt{(\forall a b. r a b \implies r b a) \implies symp\ r}

\texttt{by (fact symI [to-pred])}

\textbf{lemma} \texttt{symE:}

\texttt{assumes sym\ r \ and \ (b, a) \in r}

\texttt{obtains (a, b) \in r}

\texttt{using assms by (simp add: sym-def)}

\textbf{lemma} \texttt{sympE:}

\texttt{assumes symp\ r \ and \ r b a}

\texttt{obtains r a b}

\texttt{using assms by (rule symE [to-pred])}

\textbf{lemma} \texttt{symD:}

\texttt{assumes sym\ r \ and \ (b, a) \in r}

\texttt{shows (a, b) \in r}

\texttt{using assms by (rule symE)}

\textbf{lemma} \texttt{sympD:}
assumes symp r and r b a
shows r a b
using assms by (rule symD [to-pred])

lemma sym-Int:
  \( \text{sym } r \implies \text{sym } s \implies \text{sym } (r \cap s) \)
by (fast intro: symI elim: symE)

lemma symp-inf:
  symp r \implies symp s \implies symp (r \cap s)
by (fact sym-Int [to-pred])

lemma sym-Un:
  sym r \implies sym s \implies sym (r \cup s)
by (fast intro: symI elim: symE)

lemma symp-sup:
  symp r \implies symp s \implies symp (r \cup s)
by (fact sym-Un [to-pred])

lemma sym-INTER:
  \( \forall x \in S. \text{sym } (r x) \implies \text{sym } (\INTER S r) \)
by (fast intro: symI elim: symE)

lemma symp-INF:
  \( \forall x \in S. \text{symp } (r x) \implies \text{symp } (\INFIMUM S r) \)
by (fact sym-INTER [to-pred])

lemma sym-UNION:
  \( \forall x \in S. \text{sym } (r x) \implies \text{sym } (\UNION S r) \)
by (fast intro: symI elim: symE)

lemma symp-SUP:
  \( \forall x \in S. \text{symp } (r x) \implies \text{symp } (\SUPREMUM S r) \)
by (fact sym-UNION [to-pred])

19.2.5 Antisymmetry

definition antisym :: 'a rel \Rightarrow bool
where
  antisym r \iff (\forall x y. (x, y) \in r \implies (y, x) \in r \implies x = y)

abbreviation antisymP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool
where
  antisymP r \equiv antisym \{ (x, y). r x y \}

lemma antisymI:
  (! x y. (x, y) : r \Rightarrow (y, x) : r \Rightarrow x = y) \Rightarrow antisym r
by (unfold antisym-def) iprover
lemma antisymD: antisym r ==> (a, b) : r ==> (b, a) : r ==> a = b
by (unfold antisym-def) iprover

lemma antisym-subset: r ⊆ s ==> antisym s ==> antisym r
by (unfold antisym-def) blast

lemma antisym-empty [simp]: antisym {}
by (unfold antisym-def) blast

19.2.6 Transitivity

definition trans :: 'a rel ⇒ bool
where
trans r ≜ (∀ x y z. (x, y) ∈ r → (y, z) ∈ r → (x, z) ∈ r)

definition transp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
where
transp r ≜ (∀ x y z. r x y → r y z → r x z)

lemma transp-trans-ev [pred-set-conv]:
transp (λ x y. (x, y) ∈ r) ≜ trans r
by (simp add: trans-def transp-def)

abbreviation transP :: ('a ⇒ 'a ⇒ bool) ⇒ bool
where — FIXME drop
transP r ≜ trans { (x, y). r x y }

lemma transI:
(∀ x y z. (x, y) ∈ r → (y, z) ∈ r → (x, z) ∈ r) → trans r
by (unfold trans-def) iprover

lemma transpI:
(∀ x y z. (x, y) ∈ r → (y, z) ∈ r → (x, z) ∈ r) → transp r
by (fact transI [to-pred])

lemma transE:
assumes trans r and (x, y) ∈ r and (y, z) ∈ r
obtains (x, z) ∈ r
using assms by (unfold trans-def) iprover

lemma transpE:
assumes transp r and r x y and r y z
obtains r x z
using assms by (rule transE [to-pred])

lemma transD:
assumes trans r and (x, y) ∈ r and (y, z) ∈ r
shows (x, z) ∈ r
using assms by (rule transE)

lemma transD:
  assumes transp r and r x y and r y z
  shows r x z
  using assms by (rule transD [to-pred])

lemma trans-Int:
  trans r =⇒ trans s =⇒ trans (r ∩ s)
  by (fast intro: transI elim: transE)

lemma transp-inf:
  transp r =⇒ transp s =⇒ transp (r ⊓ s)
  by (fact trans-Int [to-pred])

lemma trans-INTER:
∀ x ∈ S. trans (r x) =⇒ trans (INTER S r)
  by (fast intro: transI elim: transD)

lemma trans-join [code]:
  trans r =⇒ (∀ (x, y1) ∈ r. ∀ (y2, z) ∈ r. y1 = y2 → (x, z) ∈ r)
  by (auto simp add: trans-def)

lemma transp-trans:
  transp r =⇒ trans {(x, y). r x y}
  by (simp add: transp-def)

19.2.7 Totality

definition total-on :: 'a set ⇒ 'a rel ⇒ bool
where
  total-on A r =⇒ (∀ x ∈ A. ∀ y ∈ A. x ≠ y → (x, y) ∈ r ∨ (y, x) ∈ r)

abbreviation total ≡ total-on UNIV

lemma total-on-empty [simp]: total-on {} r
  by (simp add: total-on-def)

19.2.8 Single valued relations

definition single-valued :: ('a × 'b) set ⇒ bool
where
  single-valued r =⇒ (∀ x y. (x, y) ∈ r → (∀ z. (x, z) ∈ r → y = z))

abbreviation single-valuedP :: ('a ⇒ 'b ⇒ bool) ⇒ bool where
  single-valuedP r ≡ single-valued {(x, y). r x y}

lemma single-valuedI:
THEORY “Relation”

\[ \forall x \, y. \, (x, y) : r \rightarrow (\forall z. \, (x, z) : r \rightarrow y = z) \rightarrow \text{single-valued } r \]
by (unfold single-valued-def)

lemma single-valuedD:
\[ \text{single-valued } r \rightarrow (x, y) : r \rightarrow (x, z) : r \rightarrow y = z \]
by (simp add: single-valued-def)

lemma single-valued-empty[simp]: single-valued \{\}
by (simp add: single-valued-def)

lemma single-valued-subset:
\[ r \subseteq s \rightarrow \text{single-valued } s \rightarrow \text{single-valued } r \]
by (unfold single-valued-def) blast

19.3 Relation operations

19.3.1 The identity relation

definition Id :: 'a rel
where
\[ \{p. \exists x. \, p = (x, x)\} \]

lemma IdI [intro]: \((a, a) : Id\)
by (simp add: Id-def)

lemma IdE [elim!]: \(p : Id \rightarrow (!x. \, p = (x, x) \rightarrow P) \rightarrow P\)
by (unfold Id-def) (iprover elim: CollectE)

lemma pair-in-Id-conv [iff]: \((a, b) : Id = (a = b)\)
by (unfold Id-def) blast

lemma refl-Id: refl Id
by (simp add: refl-on-def)

lemma antisym-Id: antisym Id
— A strange result, since Id is also symmetric.
by (simp add: antisym-def)

lemma sym-Id: sym Id
by (simp add: sym-def)

lemma trans-Id: trans Id
by (simp add: trans-def)

lemma single-valued-Id [simp]: single-valued Id
by (unfold single-valued-def) blast

lemma irrefl-diff-Id [simp]: irrefl \((r - Id)\)
by (simp add:irrefl-def)
THEORY "Relation"

lemma trans-diff-Id: trans r \implies antisym r \implies trans (r - Id)
unfolding antisym-def trans-def by blast

lemma total-on-diff-Id [simp]: total-on A (r - Id) = total-on A r
by (simp add: total-on-def)

19.3.2 Diagonal: identity over a set
definition Id-on :: 'a set \Rightarrow 'a rel
where
Id-on A = (\bigcup x\in A. \{(x, x)\})

lemma Id-on-empty [simp]: Id-on {} = {}
by (simp add: Id-on-def)

lemma Id-on-eqI [intro!]: a = b \implies a : A \implies (a, b) : Id-on A
by (rule Id-on-eqI) (rule refl)

lemma Id-onI [intro!]: a : A \implies (a, a) : Id-on A
by (rule Id-on-eqI) (rule refl)

lemma Id-onE [elim!]:
c : Id-on A \implies (!x. x : A \implies c = (x, x) \implies P) \implies P
— The general elimination rule.
by (unfold Id-on-def) (iprover elim: UN-E singletonE)

lemma Id-on-iff: ((x, y) : Id-on A) = (x = y & x : A)
by blast

lemma Id-on-def' [nitpick-unfold]:
Id-on \{x. A x\} = Collect (\lambda(x, y). x = y \land A x)
by auto

lemma Id-on-subset-Times: Id-on A \subseteq A \times A
by blast

lemma refl-on-Id-on: refl-on A (Id-on A)
by (rule refl-onI [OF Id-on-subset-Times Id-onI])

lemma antisym-Id-on [simp]: antisym (Id-on A)
by (unfold antisym-def) blast

lemma sym-Id-on [simp]: sym (Id-on A)
by (rule symI) clarify

lemma trans-Id-on [simp]: trans (Id-on A)
by (fast intro: transI elim: transD)

lemma single-valued-Id-on [simp]: single-valued (Id-on A)
by \((\text{unfold single-valued-def})\) blast

### 19.3.3 Composition

**inductive-set** \(\text{relcomp} :: ('a \times 'b) \set \Rightarrow ('b \times 'c) \set \Rightarrow ('a \times 'c) \set\) (**infixr** \(O\) 75)

for \(r :: ('a \times 'b) \set\) and \(s :: ('b \times 'c) \set\)

where

\(\text{relcompI \[intro\]}: (a, b) \in r \Rightarrow (b, c) \in s \Rightarrow (a, c) \in r O s\)

**notation** \(\text{relcompp} \ (\text{infixr} \ O O 75)\)

**lemmas** \(\text{relcomppI} = \text{relcompp\_intros}\)

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.

**inductive-cases** \(\text{relcompEpair}\): \((a, c) \in r O s\)

**inductive-cases** \(\text{relcomppE \[elim\]}: (r OO s) a c\)

**lemma** \(\text{relcomppE \[elim\]}: \{x z : \in r O s \Rightarrow (\forall x y z. xz = (x, z) \Rightarrow (x, y) \in r \Rightarrow (y, z) \in s \Rightarrow P) \Rightarrow P\)

by \((\text{cases} xz) \ (\text{simp, erule relcomppEpair, iprover})\)

**lemma** \(\text{R-O-Id \[simp\]}: R O Id = R\)

by \(\text{fast}\)

**lemma** \(\text{Id-O-R \[simp\]}: Id O R = R\)

by \(\text{fast}\)

**lemma** \(\text{relcomp-empty1 \[simp\]}: \emptyset O R = \emptyset\)

by \(\text{blast}\)

**lemma** \(\text{relcompp-bot1 \[simp\]}: \bot O R = \bot\)

by \((\text{fact relcompp-empty1})\)

**lemma** \(\text{relcompp-empty2 \[simp\]}: R O \emptyset = \emptyset\)

by \(\text{blast}\)

**lemma** \(\text{relcompp-bot2 \[simp\]}: R O \bot = \bot\)

by \((\text{fact relcompp-empty2})\)

**lemma** \(\text{O-assoc}: (R O S) O T = R O (S O T)\)
by blast

lemma relcompp-assoc:
(r OO s) OO t = r OO (s OO t)
by (fact O-assoc [to-pred])

lemma trans-O-subset:
trans r → r O r ⊆ r
by (unfold trans-def) blast

lemma transp-relcompp-less-eq:
transp r → r OO r \leq r
by (fact trans-O-subset [to-pred])

lemma relcomp-mono:
r' \subseteq r \Rightarrow s' \subseteq s \Rightarrow r' O s' \subseteq r O s
by blast

lemma relcompp-mono:
r' \leq r \Rightarrow s' \leq s \Rightarrow r' OO s' \leq r OO s
by (fact relcomp- mono [to-pred])

lemma relcomp-subset-Sigma:
r \subseteq A \times B \Rightarrow s \subseteq B \times C \Rightarrow r O s \subseteq A \times C
by blast

lemma relcomp-distrib [simp]:
R O (S \cup T) = (R O S) \cup (R O T)
by auto

lemma relcomp-distrib [simp]:
R OO (S \cup T) = R OO S \cup R OO T
by (fact relcomp-distrib [to-pred])

lemma relcomp-distrib2 [simp]:
(S \cup T) O R = (S O R) \cup (T O R)
by auto

lemma relcomp-distrib2 [simp]:
(S \cup T) OO R = S OO R \cup T OO R
by (fact relcomp-distrib2 [to-pred])

lemma relcomp-UNION-distrib:
s O UNION I r = (\bigcup i \in I. s O r i)
by auto
lemma relcomp-UNION-distrib2:
  UNION I r O s = (⋃i ∈ I. r i O s)
by auto

lemma single-valued-relcomp:
  single-valued r ⇒ single-valued s ⇒ single-valued (r O s)
by (unfold single-valued-def) blast

lemma relcomp-unfold:
  r O s = { (x, z). ∃y. (x, y) ∈ r ∧ (y, z) ∈ s }
by (auto simp add: set-eq-iff)

lemma eq-OO: op= OO R = R
by blast

19.3.4 Converse

inductive-set converse :: ('a × 'b) set ⇒ ('b × 'a) set ((·⁻¹) [1000] 999)
  for r :: ('a × 'b) set
where
  (a, b) ∈ r ⇒ (b, a) ∈ r⁻¹

notation (xsymbols)
  converse ((·⁻¹) [1000] 999)

notation
  converse (·⁻¹) [1000] 1000)

notation (xsymbols)
  conversep ((·⁻¹⁻¹) [1000] 1000)

lemma converseI [sym]:
  (a, b) ∈ r ⇒ (b, a) ∈ r⁻¹
by (fact converse.intros)

lemma conversepI :
  r a b ⇒ r⁻¹⁻¹ b a
by (fact conversep.intros)

lemma converseD [sym]:
  (a, b) ∈ r⁻¹ ⇒ (b, a) ∈ r
by (erule converse.cases) iprover

lemma conversepD :
  r⁻¹⁻¹ b a ⇒ r a b
by (fact converseD [to-pred])
lemma converseE [elim!]:
— More general than converseD, as it “splits” the member of the relation.
\[yx \in r^{-1} \Longrightarrow (\forall x y. \ yx = (y, x) \Longrightarrow (x, y) \in r \Longrightarrow P) \Longrightarrow P\]
by (cases \(yx\)) (simp, erule converse_cases, iprover)

lemmas converseE [elim!] = conversep_cases

lemma converse-iff [iff]:
\[(a, b) \in r^{-1} \iff (b, a) \in r\]
by (auto intro: converseI)

lemma conversep-iff [iff]:
\[r^{-1}^{-1} a b = r b a\]
by (fact converse-iff [to-pred])

lemma converse-converse [simp]:
\[(r^{-1})^{-1} = r\]
by (simp add: set-eq_iff)

lemma conversep-conversep [simp]:
\[(r^{-1}^{-1})^{-1} = r\]
by (fact converse-converse [to-pred])

lemma converse-empty [simp]: \[
\{\}^{-1} = \{}
\]
by auto

lemma converse-UNIV [simp]: \[UNIV^{-1} = UNIV\]
by auto

lemma converse-relcomp: \[(r O s)^{-1} = s^{-1} O r^{-1}\]
by blast

lemma converse-relcompp: \[(r OO s)^{-1} = s^{-1} OO r^{-1}\]
by (iprover intro: order-antisym conversepI relcomppI
elim: relcomppE dest: conversepD)

lemma converse-Int: \[(r \cap s)^{-1} = r^{-1} \cap s^{-1}\]
by blast

lemma converse-meet: \[(r \cap s)^{-1} = r^{-1} \cap s^{-1}\]
by (simp add: inf-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-Un: \[(r \cup s)^{-1} = r^{-1} \cup s^{-1}\]
by blast

lemma converse-join: \[(r \sqcup s)^{-1} = r^{-1} \sqcup s^{-1}\]
by (simp add: sup-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-INTER: \[\operatorname{INTER} S r^{-1} = (\operatorname{INT} x:S. \ (x \cdot r)^{-1}\)\]
by fast

lemma converse-UNION: $(\text{UNION } S \ r) \ ^{-1} = (\text{UN } x:S. \ (r \ x) \ ^{-1})$
by blast

lemma converse-mono[simp]: $r^{-1} \subseteq s ^{-1} \iff r \subseteq s$
by auto

lemma conversep-mono[simp]: $r^{--1} \leq s ^{--1} \iff r \leq s$
by (fact converse-mono[to-pred])

lemma converse-inject[simp]: $r^{-1} = s ^{-1} \iff r = s$
by auto

lemma conversep-inject[simp]: $r^{--1} = s ^{--1} \iff r = s$
by (fact converse-inject[to-pred])

lemma converse-subset-swap: $r \subseteq s ^{-1} = (r ^{-1} \subseteq s)$
by auto

lemma conversep-le-swap: $r \leq s ^{--1} = (r ^{--1} \leq s)$
by (fact converse-subset-swap[to-pred])

lemma converse-Id [simp]: $\text{Id}^{-1} = \text{Id}$
by blast

lemma converse-Id-on [simp]: $(\text{Id-on } A) ^{-1} = \text{Id-on } A$
by blast

lemma refl-on-converse [simp]: refl-on A (converse r) = refl-on A r
by (unfold refl-on-def) auto

lemma sym-converse [simp]: sym (converse r) = sym r
by (unfold sym-def) blast

lemma antisym-converse [simp]: antisym (converse r) = antisym r
by (unfold antisym-def) blast

lemma trans-converse [simp]: trans (converse r) = trans r
by (unfold trans-def) blast

lemma sym-conv-converse-eq: sym r = $(r^{-1} \ = \ r)$
by (unfold sym-def) fast

lemma sym-Un-converse: sym $(r \cup r^{-1})$
by (unfold sym-def) blast

lemma sym-Int-converse: sym $(r \cap r^{-1})$
by (unfold sym-def) blast
lemma total-on-converse [simp]: total-on A (r⁻¹) = total-on A r
  by (auto simp: total-on-def)

lemma finite-converse [iff]: finite (r⁻¹) = finite r
  unfolding converse-def conversep-iff using [[simproc add: finite-Collect]]
  by (auto elim: finite-imageD simp: inj-on-def)

lemma conversep-noteq [simp]: (op ≠)⁻¹ = op ≠
  by (auto simp add: fun-eq-iff)

lemma conversep-eq [simp]: (op =)⁻¹ = op =
  by (auto simp add: fun-eq-iff)

lemma converse-unfold [code]:
  r⁻¹ = {(y, x). (x, y) ∈ r}
  by (simp add: set-eq-iff)

19.3.5 Domain, range and field

inductive-set Domain :: (′a × ′b) set ⇒ ′a set
  for r :: (′a × ′b) set
  where
    DomainI [intro]: (a, b) ∈ r =⇒ a ∈ Domain r

abbreviation (input) DomainP ≡ Domainp

lemmas DomainPI = Domainp.DominI

inductive-cases DomainE [elim!]: a ∈ Domain r

inductive-set Range :: (′a × ′b) set ⇒ ′b set
  for r :: (′a × ′b) set
  where
    RangeI [intro]: (a, b) ∈ r =⇒ b ∈ Range r

abbreviation (input) RangeP ≡ Rangep

lemmas RangePI = Rangep.RangeI

inductive-cases RangeE [elim!]: b ∈ Range r
inductive-cases RangepE [elim!]: Rangep r b

definition Field :: ′a rel ⇒ ′a set
  where
    Field r = Domain r ∪ Range r

lemma Domain-fst [code]:
Domain \( r = \text{fst} \cdot r \)
by force

lemma Range-snd [code]:
Range \( r = \text{snd} \cdot r \)
by force

lemma fst-eq-Domain:
\( \text{fst} \cdot R = \text{Domain} R \)
by force

lemma snd-eq-Range:
\( \text{snd} \cdot R = \text{Range} R \)
by force

lemma Domain-empty [simp]: Domain \( \{\} = \{\} \)
by auto

lemma Range-empty [simp]: Range \( \{\} = \{\} \)
by auto

lemma Field-empty [simp]: Field \( \{\} = \{\} \)
by (simp add: Field-def)

lemma Domain-empty-iff:
\( \text{Domain} \ r = \{\} \iff r = \{\} \)
by auto

lemma Range-empty-iff:
\( \text{Range} \ r = \{\} \iff r = \{\} \)
by auto

lemma Domain-insert [simp]: Domain (insert \( \{\) \( (a, b) r\) = insert \( a \) (Domain \( r\)
by blast

lemma Range-insert [simp]: Range (insert \( \{\) \( (a, b) r\) = insert \( b \) (Range \( r\)
by blast

lemma Field-insert [simp]: Field (insert \( \{\) \( (a, b) r\) = \( a, b \) \cup Field \( r\)
by (auto simp add: Field-def)

lemma Domain-iff:
\( a \in \text{Domain} \ r \iff (\exists y. (a, y) \in r) \)
by blast

lemma Range-iff:
\( a \in \text{Range} \ r \iff (\exists y. (y, a) \in r) \)
by blast

lemma Domain-Id [simp]: Domain Id = UNIV
by blast

lemma Range-Id [simp]: Range Id = UNIV
by blast
lemma Domain-Id-on [simp]: Domain (Id-on A) = A
  by blast

lemma Range-Id-on [simp]: Range (Id-on A) = A
  by blast

lemma Domain-Un-eq: Domain (A ∪ B) = Domain A ∪ Domain B
  by blast

lemma Range-Un-eq: Range (A ∪ B) = Range A ∪ Range B
  by blast

lemma Field-Un [simp]: Field (r ∪ s) = Field r ∪ Field s
  by (auto simp: Field-def)

lemma Domain-Int-subset: Domain (A ∩ B) ⊆ Domain A ∩ Domain B
  by blast

lemma Range-Int-subset: Range (A ∩ B) ⊆ Range A ∩ Range B
  by blast

lemma Domain-Diff-subset: Domain A − Domain B ⊆ Domain (A − B)
  by blast

lemma Range-Diff-subset: Range A − Range B ⊆ Range (A − B)
  by blast

lemma Domain-Union: Domain (∪ S) = (∪ A∈S. Domain A)
  by blast

lemma Range-Union: Range (∪ S) = (∪ A∈S. Range A)
  by blast

lemma Field-Union [simp]: Field (∪ R) = ∪ (Field ' R)
  by (auto simp: Field-def)

lemma Domain-converse [simp]: Domain (r⁻¹) = Range r
  by auto

lemma Range-converse [simp]: Range (r⁻¹) = Domain r
  by blast

lemma Field-converse [simp]: Field (r⁻¹) = Field r
  by (auto simp: Field-def)

lemma Domain-Collect-split [simp]: Domain {((x, y). P x y)} = {x. EX y. P x y}
  by auto

lemma Range-Collect-split [simp]: Range {((x, y). P x y)} = {y. EX x. P x y}
by auto

lemma finite-Domain: finite $r \implies$ finite (Domain $r$)
by (induct set: finite) auto

lemma finite-Range: finite $r \implies$ finite (Range $r$)
by (induct set: finite) auto

lemma finite-Field: finite $r \implies$ finite (Field $r$)
by (simp add: Field-def finite-Domain finite-Range)

lemma Domain-mono: $r \subseteq s \implies$ Domain $r \subseteq$ Domain $s$
by blast

lemma Range-mono: $r \subseteq s \implies$ Range $r \subseteq$ Range $s$
by blast

lemma mono-Field: $r \subseteq s \implies$ Field $r \subseteq$ Field $s$
by (auto simp: Field-def Domain-def Range-def)

lemma Domain-unfold:
Domain $r = \{ x. \exists y. (x, y) \in r \}$
by blast

19.3.6 Image of a set under a relation

definition Image :: "('a × 'b) set ⇒ 'a set ⇒ 'b set (infixr "" 90)" where
r "" s = { y. \exists x \in s. (x, y) \in r }

lemma Image-iff: (b : r""A) = (EX x:A. (x, b) : r)
by (simp add: Image-def)

lemma Image-singleton: r""{a} = { b. (a, b) : r}
by (simp add: Image-def)

lemma Image-singleton-iff [iff]: (b : r""{a}) = ((a, b) : r)
by (rule Image-iff [THEN trans]) simp

lemma ImageI [intro]: (a, b) : r ==⇒ a : A ==⇒ b : r""A
by (unfold Image-def) blast

lemma ImageE [elim!]:
b : r "" A ==⇒ (!!x. (x, b) : r ==⇒ x : A ==⇒ P) ==⇒ P
by (unfold Image-def) (iprover elim!: CollectE bexE)

lemma rev-ImageI: a : A ==⇒ (a, b) : r ==⇒ b : r "" A
— This version’s more effective when we already have the required a
by blast
lemma Image-empty [simp]:  
  \( R'' \{ \} = \{ \} \)
  by blast

lemma Image-Id [simp]:  
  \( \text{Id} \{ A = A \) 
  by blast

lemma Image-Id-on [simp]:  
  \( \text{Id-on} A \{ B = A \cap B \) 
  by blast

lemma Image-Int-subset:  
  \( R'' (A \cap B) \subseteq R'' A \cap R'' B \) 
  by blast

lemma Image-Int-eq:  
  \( \text{single-valued} (\text{converse } R) \implies R'' (A \cap B) = R'' A \cap R'' B \) 
  by (simp add: single-valued-def, blast)

lemma Image-Un:  
  \( R'' (A \cup B) = R'' A \cup R'' B \) 
  by blast

lemma Un-Image:  
  \( (R \cup S)'' A = R'' A \cup S'' A \) 
  by blast

lemma Image-subset:  
  \( r \subseteq A \times B \implies r'' C \subseteq B \) 
  by (iprover intro!: subsetI elim!: ImageE dest: subsetD SigmaD2)

lemma Image-eq-UN:  
  \( r'' B = (\bigcup y \in B. r'' \{ y \}) \) 
  — NOT suitable for rewriting 
  by blast

lemma Image-mono:  
  \( r' \subseteq r 
  \implies A' \subseteq A 
  \implies (r'' A') \subseteq (r'' A) \) 
  by blast

lemma Image-UN:  
  \( (r'' (\text{UNION } A B)) = (\bigcup x \in A. r'' (B x)) \) 
  by blast

lemma UN-Image:  
  \( (\bigcup i \in I. X i)'' S = (\bigcup i \in I. X i'' S) \) 
  by auto

lemma Image-INT-subset:  
  \( (r'' \text{INTER } A B) \subseteq (\bigcap x \in A. r'' (B x)) \) 
  by blast

Converse inclusion requires some assumptions

lemma Image-INT-eq:
  \[[\text{single-valued} (r^{-1}); A \neq \{ \} \] 
  \implies r'' \text{INTER } A B = (\bigcap x \in A. r'' (B x)) \)

apply (rule equalityI)
apply (rule Image-INT-subset)
apply (simp add: single-valued-def, blast)
done
lemma Image-subset-eq: \( (r''A \subseteq B) = (A \subseteq -(r^-1)''(-B)) \)
by blast

lemma Image-Collect-split [simp]: \( \{ (x, y). P x y \}''A = \{ y. \text{EX} x:A. P x y \} \)
by auto

lemma Sigma-Image: \((SIGMA \ x:A. B \ x)''X = (\bigcup x \in X \cap A. B \ x)\)
by auto

lemma relcomp-Image: \((X \ O \ Y)''Z = Y ''(X '' Z)\)
by auto

19.3.7 Inverse image

definition inv-image :: 'b rel \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a rel
where
inv-image r f = \{(x, y). (f x, f y) \in r\}

definition inv-imagep :: ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool
where
inv-imagep r f = (\lambda x y. r (f x) (f y))

lemma [pred-set-conv]: inv-imagep (\lambda x y. (x, y) \in r) f = (\lambda x y. (x, y) \in inv-image r f)
by (simp add: inv-image-def inv-imagep-def)

lemma sym-inv-image: sym r \Rightarrow sym (inv-image r f)
by (unfold sym-def inv-image-def) blast

lemma trans-inv-image: trans r \Rightarrow trans (inv-image r f)
apply (unfold trans-def inv-image-def)
apply blast
done

lemma in-inv-image[simp]: \((x,y) : inv-image r f) = ((f x, f y) : r)
by (auto simp:inv-image-def)

lemma converse-inv-image[simp]: (inv-image R f)''-1 = inv-image (R''-1) f
unfolding inv-image-def converse-unfold by auto

lemma in-inv-imagep [simp]: inv-imagep r f x y = r (f x) (f y)
by (simp add: inv-imagep-def)

19.3.8 Powerset

definition Powp :: ('a \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool
where
Powp A = (\lambda B. \forall x \in B. A x)
lemma Powp-Pow-eq [pred-set-conv]: Powp (λx. x ∈ A) = (λx. x ∈ Pow A)
by (auto simp add: Powp-def fun-eq-iff)

lemmas Powp-mono [mono] = Pow-mono [to-pred]

19.3.9 Expressing relation operations via Finite-Set.fold

lemma Id-on-fold:
assumes finite A
shows Id-on A = Finite-Set.fold (λx. Set.insert (Pair x x)) {} A
proof –
interpret comp-fun-commute λx. Set.insert (Pair x x) by default auto
show ?thesis using assms unfolding Id-on-def by (induct A) simp-all
qed

lemma comp-fun-commute-Image-fold:
  comp-fun-commute (λ(x,y) A. if x ∈ S then Set.insert y A else A)
proof –
interpret comp-fun-idem Set.insert by (fact comp-fun-idem-insert)
show ?thesis by default (auto simp add: fun-eq-iff comp-fun-commute split:prod.split)
qed

lemma Image-fold:
assumes finite R
shows R " S = Finite-Set.fold (λ(x,y) A. if x ∈ S then Set.insert y A else A) {} R
proof –
interpret comp-fun-commute (λ(x,y) A. if x ∈ S then Set.insert y A else A)
  by (rule comp-fun-commute-Image-fold)
have *: ∀x F. Set.insert x F " S = (if fst x ∈ S then Set.insert (snd x) (F " S) else (F " S))
  by (force intro: rev-ImageI)
show ?thesis using assms by (induct R) (auto simp: *)
qed

lemma insert-relcomp-union-fold:
assumes finite S
shows {x} O S ∪ X = Finite-Set.fold (λ(w,z) A'. if snd x = w then Set.insert (fst x,z) A' else A') X S
proof –
interpret comp-fun-commute λ(w,z) A'. if snd x = w then Set.insert (fst x,z) A' else A'
proof –
interpret comp-fun-idem Set.insert by (fact comp-fun-idem-insert)
  show comp-fun-commute (λ(w,z) A'. if snd x = w then Set.insert (fst x,z) A' else A')
THEORY “Equiv-Relations”

by default (auto simp add: fun-eq_iff split:prod.split)
qed

have *: \{x\} O S = \{(x', z). x' = \text{fst} x \land (\text{snd} x, z) \in S\} by (auto simp: relcomp-unfold intro!: exI)

show ?thesis unfolding *
using \{finite S\} by (induct S) (auto split: prod.split)
qed

lemma insert-relcomp-fold:
  assumes finite S
  shows Set.insert x R O S = Finite-Set.fold (\lambda(w,z) A'. if \text{snd} x = w then Set.insert (\text{fst} x, z) A' else A') (R O S) S
proof –
  have Set.insert x R O S = (\{x\} O S) \cup (R O S) by auto
  then show ?thesis by (auto simp: insert-relcomp-union-fold[OF assms])
qed

lemma comp-fun-commute-relcomp-fold:
  assumes finite S
  shows comp-fun-commute (\lambda(x,y) A. Finite-Set.fold (\lambda(w,z) A'. if y = w then Set.insert (x, z) A' else A') A S)
proof –
  have *: \forall a b A.
    Finite-Set.fold (\lambda(w, z) A'. if b = w then Set.insert (a, z) A' else A') A S = \{(a,b)\} O S \cup A
    by (auto simp: insert-relcomp-union-fold[OF assms] cong: if-cong)
  show ?thesis by default (auto simp: *)
qed

lemma relcomp-fold:
  assumes finite R
  assumes finite S
  shows R O S = Finite-Set.fold (\lambda(x,y) A. Finite-Set.fold (\lambda(w,z) A'. if y = w then Set.insert (x, z) A' else A') A S) \{\}\ R
  using assms by (induct R)
  (auto simp: comp-fun-commute_fold-insert comp-fun-commute-relcomp-fold insert-relcomp-fold cong: if-cong)

end

20 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory Equiv-Relations
imports Groups-Big Relation
begin
20.1 Equivalence relations – set version

**Definition** `equiv :: 'a set ⇒ ('a × 'a) set ⇒ bool` where
\[ \text{equiv } A \ r \leftarrow\rightarrow \text{refl-on } A \ r \land \text{sym } r \land \text{trans } r \]

**Lemma** `equivI`:
\[ \text{refl-on } A \ r \implies \text{sym } r \implies \text{trans } r \implies \text{equiv } A \ r \]
by `(simp add: equiv-def)

**Lemma** `equivE`:
- **Assumes** `equiv A r`
- **Obtains** `refl-on A r` and `sym r` and `trans r`
using **Assms** by `(simp add: equiv-def)

Suppes, Theorem 70: \( r \) is an equiv relation iff \( r^{-1} \ O r = r \).
First half: `equiv A r ==> r^{-1} O r = r`.

**Lemma** `sym-trans-comp-subset`:
\[ \text{sym } r \implies \text{trans } r \implies r^{-1} O r \subseteq r \]
by `(unfold trans-def sym-def converse-unfold) blast`

**Lemma** `refl-on-comp-subset`:
\[ \text{refl-on } A \ r \implies r \subseteq r^{-1} O r \]
by `(unfold refl-on-def) blast`

**Lemma** `equiv-comp-eq`:
\[ \text{equiv } A \ r \implies r^{-1} O r = \text{equiv } A \ r \]
apply `(unfold equiv-def)`
apply clarify
apply `(rule equalityI)`
apply `(iprover intro: sym-trans-comp-subset refl-on-comp-subset)`
done

Second half.

**Lemma** `comp-equivI`:
\[ r^{-1} O r = r \implies \text{Domain } r = A \implies \text{equiv } A \ r \]
apply `(unfold equiv-def refl-on-def sym-def trans-def)`
apply `(erule equalityE)`
apply `(subgoal-tac ∀ x y. (x, y) ∈ r −→ (y, x) ∈ r)`
apply fast
apply fast
done

20.2 Equivalence classes

**Lemma** `equiv-class-subset`:
\[ \text{equiv } A \ r \implies (a, b) ∈ r \implies r'^{-1}\{a\} \subseteq r'^{-1}\{b\} \]
— lemma for the next result
by `(unfold equiv-def trans-def sym-def) blast`

**Theorem** `equiv-class-eq`:
\[ \text{equiv } A \ r \implies (a, b) ∈ r \implies r'^{-1}\{a\} = r'^{-1}\{b\} \]
apply `(assumption | rule equalityI equiv-class-subset)`+
apply (unfold equiv-def sym-def)
apply blast
done

lemma equiv-class-self: equiv A r ==> a ∈ A ==> a ∈ r``{a}
  by (unfold equiv-def refl-on-def) blast

lemma subset-equiv-class:
  equiv A r ==> r``{b} ⊆ r``{a} ==> b ∈ A ==> (a, b) ∈ r
  — lemma for the next result
  by (unfold equiv-def refl-on-def) blast

lemma eq-equiv-class:
  r``{a} = r``{b} ==> equiv A r ==> b ∈ A ==> (a, b) ∈ r
  by (iprover intro: equalityD2 subset-equiv-class)

lemma equiv-type:
  equiv A r ==> r ⊆ A × A
  by (unfold equiv-def refl-on-def) blast

theorem equiv-class-eq-iff:
  equiv A r ==> ((x, y) ∈ r) = (r``{x} = r``{y} & x ∈ A & y ∈ A)
  by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

theorem eq-equiv-class-iff:
  equiv A r ==> x ∈ A ==> y ∈ A ==> (r``{x} = r``{y}) = ((x, y) ∈ r)
  by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

20.3 Quotients

definition quotient :: 'a set ⇒ ('a × 'a) set ⇒ 'a set set (infixl '/'/ 90)
  where
  A//r = (∪ x ∈ A. {r``{x}}) — set of equiv classes

lemma quotientI: x ∈ A ==> r``{x} ∈ A//r
  by (unfold quotient-def) blast

lemma quotientE:
  X ∈ A//r ==> (∀x. X = r``{x} ==> x ∈ A ==> P) ==> P
  by (unfold quotient-def) blast

lemma Union-quotient: equiv A r ==> Union (A//r) = A
  by (unfold equiv-def refl-on-def quotient-def) blast

lemma quotient-disj:
  equiv A r ==> X ∈ A//r ==> Y ∈ A//r ==> X = Y | (X ∩ Y = {})
  apply (unfold quotient-def)
apply clarify
apply (rule equiv-class-eq)
  apply assumption
apply (unfold equiv-def trans-def sym-def)
apply blast
done

lemma quotient-eqI:
  \[
  \begin{align*}
  &\equiv A r; X \in A/r; Y \in A/r; x \in X; y \in Y; (x,y) \in r\] \implies X = Y \\
  &apply (clarify elim!: quotientE)
  &apply (rule equiv-class-eq, assumption)
  &apply (unfold equiv-def sym-def trans-def, blast)
  &done
\]

lemma quotient-eq-iff:
  \[
  \begin{align*}
  &\equiv A r; X \in A/r; Y \in A/r; x \in X; y \in Y\] \implies (X = Y) = ((x,y) \in r) \\
  &apply (rule iffI)
  &prefer 2 apply (blast del: equalityI intro: quotient-eqI)
  &apply (clarify elim!: quotientE)
  &apply (unfold equiv-def sym-def trans-def, blast)
  &done
\]

lemma eq-equiv-class-iff2:
  \[
  \begin{align*}
  &equiv A r; x \in A; y \in A\] \implies (\{x\}/r = \{y\}/r) = ((x,y) : r) \\
  &by(simp add:quotient-def eq-equiv-class-iff)
\]

lemma quotient-empty [simp]: \{\}/r = {}
by(simp add: quotient-def)

lemma quotient-is-empty [iff]: (\{\}/r = {\}) = (A = {\})
by(simp add: quotient-def)

lemma quotient-is-empty2 [iff]: ({\}) = \{\}/r = (A = {\})
by(simp add: quotient-def)

lemma singleton-quotient: \{x\}/r = \{r'' \{x\}\}
by(simp add: quotient-def)

lemma quotient-diff1:
  \[
  \begin{align*}
  &inj-on (\%a. \{a\}/r) A; a \in A\] \implies (A - \{a\}/r = A/r - \{a\}/r \\
  &apply(simp add:quotient-def inj-on-def)
  &apply blast
  &done
\]
20.4 Defining unary operations upon equivalence classes

A congruence-preserving function

definition congruent :: (′a × ′a) set ⇒ (′a ⇒ ′b) ⇒ bool where
congruent r f ←→ (∀ (y, z) ∈ r. f y = f z)

lemma congruentI:
(∀ y z. (y, z) ∈ r ⇒ f y = f z) ⇒ congruent r f
by (auto simp add: congruent-def)

lemma congruentD:
congruent r f ⇒ (y, z) ∈ r ⇒ f y = f z
by (auto simp add: congruent-def)

abbreviation RESPECTS :: (′a ⇒ ′b) ⇒ (′a × ′a) set ⇒ bool
(infixr respects 80) where
f respects r == congruent r f

lemma UN-constant-eq: a ∈ A ==> ∀ y ∈ A. f y = c ==> (∪ y ∈ A. f(y))=c
— lemma required to prove UN-equiv-class
by auto

lemma UN-equiv-class:
equiv A r ==> f respects r ==> a ∈ A
==> (∪ x ∈ r``{a}. f x) = f a
— Conversion rule
apply (rule equiv-class-self [THEN UN-constant-eq], assumption+)
apply (unfold equiv-def congruent-def sym-def)
apply (blast del: equalityI)
done

lemma UN-equiv-class-type:
equiv A r ==> f respects r ==> X ∈ A//r ==> (!x. x ∈ A ==> f x ∈ B) ==> (∪ x ∈ X. f x) ∈ B
apply (unfold quotient-def)
apply clarify
apply (subst UN-equiv-class)
apply auto
done

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be !!y. y ∈ A ==> f y ∈ B.

lemma UN-equiv-class-inject:
equiv A r ==> f respects r ==> (∪ x ∈ X. f x) = (∪ y ∈ Y. f y) ==> X ∈ A//r ==> Y ∈ A//r
==> (!x y. x ∈ A ==> y ∈ A ==> f x = f y ==> (x, y) ∈ r)
==> X = Y
apply (unfold quotient-def)
apply clarify
apply (rule equiv-class-eq)
apply assumption
apply (subgoal-tac f x = f xa)
apply blast
apply (erule box-equals)
apply (assumption | rule UN-equiv-class)+
done

20.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

definition congruent2 :: ('a × 'a) set ⇒ ('b × 'b) set ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ bool
where
congruent2 r1 r2 f = (∀ (y1, z1) ∈ r1. ∀ (y2, z2) ∈ r2. f y1 y2 = f z1 z2)

lemma congruent2I':: assumes (∀ y1 z1 y2 z2. (y1, z1) ∈ r1 =⇒ (y2, z2) ∈ r2 =⇒ f y1 y2 = f z1 z2)
shows congruent2 r1 r2 f
using assms by (auto simp add: congruent2-def)

lemma congruent2D:: congruent2 r1 r2 f =⇒ (∀ y1 z1 y2 z2. (y1, z1) ∈ r1 =⇒ (y2, z2) ∈ r2 =⇒ f y1 y2 = f z1 z2)
using assms by (auto simp add: congruent2-def)

Abbreviation for the common case where the relations are identical

abbreviation RESPECTS2:: '[a => 'a, ('a * 'a) set] => bool
(infixr respects2 80) where
  f respects2 r = congruent2 r r f

lemma congruent2-implies-congrent:
equiv A r1 =⇒ congruent2 r1 r2 f =⇒ a ∈ A =⇒ congruent r2 (f a)
by (unfold congruent-def congruent2-def equiv-def refl-on-def) blast

lemma congruent2-implies-congrent-UN:
equiv A1 r1 =⇒ equiv A2 r2 =⇒ congruent2 r1 r2 f =⇒ a ∈ A2 =⇒ congruent r1 (λx. ∪x ∈ r2. {a} · f x)
apply (unfold congruent-def)
apply clarify
apply (rule equiv-type [THEN subsetD, THEN SigmaE2], assumption+)
apply (simp add: UN-equiv-class congruent2-implies-congrent)
apply (unfold congruent2-def equiv-def refl-on-def)
apply (blast del: equalityI)
done

lemma UN-equiv-class2:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a1 ∈ A1 ==> a2 ∈ A2
==> (∪ x1 ∈ r1"{a1}. ∪ x2 ∈ r2"{a2}. f x1 x2) = f a1 a2
by (simp add: UN-equiv-class congruent2-implies-congruent
congruent2-implies-congruent-UN)

lemma UN-equiv-class-type2:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 ∈ A1/\r1 ==> X2 ∈ A2/\r2
==> (!!y1 x2. x1 ∈ A1 ==> x2 ∈ A2 ==> f x1 x2 ∈ B)
==> (∪ x1 ∈ X1. ∪ x2 ∈ X2. f x1 x2) ∈ B
apply (unfold quotient-def)
apply clarify
apply (blast intro: UN-equiv-class-type congruent2-implies-congruent-UN con stuck quotientI)
done

lemma UN-UN-split-split-eq:
(∪(x1, x2) ∈ X. ∪(y1, y2) ∈ Y. A x1 x2 y1 y2) =
(∪ x ∈ X. ∪ y ∈ Y. (λ(x1, x2). (λ(y1, y2). A x1 x2 y1 y2) y) x)
— Allows a natural expression of binary operators,
— without explicit calls to split
by auto

lemma congruent2I:
equiv A1 r1 ==> equiv A2 r2
==> (!!y z w. w ∈ A2 ==> (y,z) ∈ r1 ==> f y w = f z w)
==> (!!y z w. w ∈ A1 ==> (y,z) ∈ r2 ==> f w y = f w z)
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— much simpler than the direct proof.
apply (unfold congruent2-def equiv-def refl-on-def)
apply clarify
apply (blast intro: trans)
done

lemma congruent2-commuteI:
assumes equivA: equiv A r
and commute: (!!y z. y ∈ A ==> z ∈ A ==> f y z = f z y)
and cong: (!!y z w. w ∈ A ==> (y,z) ∈ r ==> f w y = f w z)
shows f respects2 r
apply (rule congruent2I [OF equivA equivA])
apply (rule commute [THEN trans])
apply (rule_tac [3] commute [THEN trans, symmetric])
apply (rule_tac [5] sym)
apply (rule cong | assumption |
erule equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2])+
done
20.6 Quotients and finiteness

Suggested by Florian Kammüller

**lemma finite-quotient:** \( \text{finite } A \implies r \subseteq A \times A \implies \text{finite } (A//r) \)
- recall \( \text{equiv } ?A \implies \text{finite } ?r \subseteq ?A \times ?A \)
  apply (rule finite-subset)
  apply (erule-tac [2] finite-Pow-iff [THEN iffD2])
  apply (unfold quotient-def)
  apply blast
  done

**lemma finite-equiv-class:**
\( \text{finite } A \implies r \subseteq A \times A \implies X \in A//r \implies \text{finite } X \)
  apply (rule finite-subset)
  prefer 2 apply assumption
  apply blast
  done

**lemma equiv-imp-dvd-card:**
\( \text{finite } A \implies \text{equiv } A r \implies \forall X \in A//r. k \text{ dvd card } X \implies \text{card } A \)
  apply (rule Union-quotient [THEN subst [where \( P=\lambda A. k \text{ dvd card } A \)]]
  apply assumption
  apply (rule dvd-partition)
  prefer 3 apply (blast dest: quotient-disj)
  apply (simp-all add: Union-quotient equiv-type)
  done

**lemma card-quotient-disjoint:**
\([ \text{finite } A; \text{inj-on } (\lambda x. \{x\}//r) A \] \implies \text{card}(A//r) = \text{card } A \)
  apply (simp add: quotient-def)
  apply (subst card-UN-disjoint)
  apply assumption
  apply simp
  apply (fastforce simp add: inj-on-def)
  apply simp
  done

20.7 Projection

definition proj where \( \text{proj } r x = r'' \{x\} \)

**lemma proj-preserves:**
\( x \in A \implies \text{proj } r x \in A//r \)
  unfolding proj-def by (rule quotientI)

**lemma proj-in iff:**
assumes \( \text{equiv } A r \)
shows \( (\text{proj } r \ x \in A//r) = (x \in A) \)

apply (rule iffI, auto simp add: proj-preserves)

unfolding proj-def quotient-def proof clarsimp

fix \( y \) assume \( y \in A \) and \( r " \{x\} = r " \{y\} \)

moreover have \( y \in r " \{y\} \) using assms unfolding equiv-def refl-on-def

by blast

ultimately have \( (x,y) \in r \) by blast

thus \( x \in A \) using assms unfolding equiv-def refl-on-def

qed

lemma proj-iff:

\[
\equiv A r; \{x,y\} \subseteq A \implies (\text{proj } r \ x = \text{proj } r \ y) = ((x,y) \in r)
\]

by (simp add: proj-def eq-equiv-class-iff)

lemma proj-image: \( (\text{proj } r) \setminus A = A//r \)

unfolding proj-def abs-def quotient-def by blast

lemma in-quotient-imp-non-empty:

\[
\equiv A r; X \in A//r \implies X \neq \{}
\]

unfolding quotient-def using equiv-class-self by fast

lemma in-quotient-imp-in-rel:

\[
\equiv A r; X \in A//r; \{x,y\} \subseteq X \implies (x,y) \in r
\]

using quotient-eq-iff [THEN iffD1] by fastforce

lemma in-quotient-imp-closed:

\[
\equiv A r; X \in A//r; x \in X; (x,y) \in r \implies y \in X
\]

unfolding quotient-def equiv-def trans-def by blast

lemma in-quotient-imp-subset:

\[
\equiv A r; X \in A//r \implies X \subseteq A
\]

using assms in-quotient-imp-in-rel equiv-type by fastforce

20.8 Equivalence relations – predicate version

Partial equivalences

definition part-equivp :: \( ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool \) where

\[
\text{part-equivp } R \leftrightarrow (\exists x. \ R x x) \land (\forall x y. \ R x y \leftrightarrow R x x \land R y y \land R x = R y)
\]

— John-Harrison-style characterization

lemma part-equivpI:

\[
(\exists x. \ R x x) \Rightarrow \text{symmp } R \Rightarrow \text{transp } R \Rightarrow \text{part-equivp } R
\]

by (auto simp add: part-equivp-def) (auto elim: symmpE transpE)

lemma part-equivpE:

assumes part-equivp R

obtains \( x \) where \( R x x \) and \( \text{symmp } R \) and \( \text{transp } R \)
proof –
from assms have 1: $\exists x. R x x$
and 2: $\forall x y. R x y \iff R x x \land R y y \land R x = R y$
by (unfold part-equivp-def) blast+
from 1 obtain $x$ where $R x x$ ..
moreover have symp $R$
proof (rule sympI)
  fix $x y$
  assume $R x y$
  with 2 [of $x y$] show $R y x$ by auto
qed
moreover have transp $R$
proof (rule transpI)
  fix $x y z$
  assume $R x y$ and $R y z$
  with 2 [of $x y$] 2 [of $y z$] show $R x z$ by auto
qed
ultimately show thesis by (rule that)
qed

lemma part-equivp-refl-symp-transp:
  part-equivp $R$ $\iff$ ($\exists x. R x x$) $\land$ symp $R$ $\land$ transp $R$
by (auto intro: part-equivpI elim: part-equivpE)

lemma part-equivp-symp:
  part-equivp $R$ $\implies$ $R x y$ $\implies$ $R y x$
by (erule part-equivpE, erule sympE)

lemma part-equivp-transp:
  part-equivp $R$ $\implies$ $R x y$ $\implies$ $R y z$ $\implies$ $R x z$
by (erule part-equivpE, erule transpE)

lemma part-equivp-typedef:
  part-equivp $R$ $\implies$ $\exists d. d \in \{ c. \exists x. R x x \land c = \text{Collect} \ (R x) \}$
by (auto elim: part-equivpE)

Total equivalences

definition equivp :: '"a ⇒ 'a ⇒ bool ⇒ bool where
  equivp $R$ $\iff$ ($\forall x y. R x y = (R x = R y)$) — John-Harrison-style characterization

lemma equivpI:
  reflp $R$ $\implies$ symp $R$ $\implies$ transp $R$ $\implies$ equivp $R$
by (auto elim: reflpE sympE transpE simp add: equivp-def)

lemma equivpE:
  assumes equivp $R$
  obtains reflp $R$ and symp $R$ and transp $R$
using assms by (auto intro!: that reflpI sympI transpI simp add: equivp-def)
THEORY "Datatype"

**lemma** equivp-implies-part-equivp:
equivp R \Rightarrow part-equivp R
by (auto intro: part-equivpI elim: equivpE reflpE)

**lemma** equivp-equiv:
equiv UNIV A \iff equivp (\lambda x y. (x, y) \in A)
by (auto intro!: equivI equivpI [to-set] elim!: equivE equivpE [to-set])

**lemma** equivp-reflp-symp-transp:
shows equivp R \iff reflp R \land symp R \land transp R
by (auto intro: equivpI elim: equivpE)

**lemma** identity-equivp:
equivp (op =)
by (auto intro: equivpI reflpI sympI transpI)

**lemma** equivp-reflp:
equivp R \Rightarrow R x x
by (erule equivpE, erule reflpE)

**lemma** equivp-symp:
equivp R \Rightarrow R x y \Rightarrow R y x
by (erule equivpE, erule sympE)

**lemma** equivp-transp:
equivp R \Rightarrow R x y \Rightarrow R y z \Rightarrow R x z
by (erule equivpE, erule transpE)

hide-const (open) proj

end

21 Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory Datatype
imports Product-Type Sum-Type Nat
keywords datatype :: thy-decl
begin

21.1 The datatype universe

definition Node = {p. EX f x k. p = (f :: nat => 'b + nat, x ::'a + nat) & f k = Inr 0}

typedef ('a, 'b) node = Node :: ((nat => 'b + nat) * ('a + nat)) set
morphisms Rep-Node Abs-Node
Datatypes will be represented by sets of type \( \text{node} \)

**type-synonym**
- \( \text{item} = ('a, \text{unit}) \text{ node set} \)
- \( ('a, 'b) \text{ dtree} = ('a, 'b) \text{ node set} \)

**consts**
- \( \text{Push} :: \begin{array}{c} ('b + \text{nat}), \text{nat} \Rightarrow ('b + \text{nat}) \Rightarrow \text{nat} \Rightarrow ('b + \text{nat}) \end{array} \)
- \( \text{Push-Node} :: \begin{array}{c} ('b + \text{nat}), ('a, 'b) \text{ node} \Rightarrow ('a, 'b) \text{ node} \end{array} \)
- \( \text{ntrunc} :: \begin{array}{c} \text{nat}, ('a, 'b) \text{ dtree} \Rightarrow ('a, 'b) \text{ dtree} \end{array} \)
- \( \text{uprod} :: \begin{array}{c} ('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree set} \Rightarrow ('a, 'b) \text{ dtree set} \end{array} \)
- \( \text{usum} :: \begin{array}{c} ('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree set} \Rightarrow ('a, 'b) \text{ dtree set} \end{array} \)
- \( \text{Split} :: \begin{array}{c} ('a, 'b) \text{ dtree}, ('a, 'b) \text{ dtree} \Rightarrow 'c \end{array} \)
- \( \text{Case} :: \begin{array}{c} ('a, 'b) \text{ dtree} \Rightarrow 'c, ('a, 'b) \text{ dtree} \Rightarrow 'c \end{array} \)
- \( \text{dprod} :: \begin{array}{c} ('a, 'b) \text{ dtree} \ast ('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree} \ast ('a, 'b) \text{ dtree set} \Rightarrow ('a, 'b) \text{ dtree set} \end{array} \)
- \( \text{dsum} :: \begin{array}{c} ('a, 'b) \text{ dtree} \ast ('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree} \ast ('a, 'b) \text{ dtree set} \Rightarrow ('a, 'b) \text{ dtree set} \end{array} \)

**defs**

**Push-Node-def:** \( \text{Push-Node} :: = (\%n x. \text{Abs-Node} (\text{apfst} (\text{Push} n)) (\text{Rep-Node} x)) \)

**Push-def:** \( \text{Push} :: = (\%b h. \text{case-nat} b h) \)

\( \text{Atom-def:} \quad \text{Atom} :: = (\%x. \{\text{Abs-Node}(\%k. \text{Inr} 0, x)\}) \)

\( \text{Scons-def:} \quad \text{Scons} M N :: = (\text{Push-Node} (\text{Inr} 1 \ast M) \text{ Un} (\text{Push-Node} (\text{Inr} (\text{Suc} 1)) \ast N) \)
THEORY “Datatype”

Leaf-def: Leaf == Atom o Inl
Numb-def: Numb == Atom o Inr

In0-def: In0(M) == Scons (Numb 0) M
In1-def: In1(M) == Scons (Numb 1) M

Lim-def: Lim f == Union {z. ? x. z = Push-Node (Inl x) \< f x\}

ndepth-def: ndepth(n) == (%(f,x). LEAST k. f k = Inr 0) (Rep-Node n)
ntrunc-def: ntrunc k N == {n. n:N & ndepth(n)<k}

uprod-def: uprod A B == UN x:A. UN y:B. { Scons x y }
usum-def: usum A B == In0'A Un In1'B

Split-def: Split c M == THE u. EX x y. M = Scons x y & u = c x y

Case-def: Case c d M == THE u. (EX x . M = In0(x) & u = c(x)) |
        (EX y . M = In1(y) & u = d(y))

dprod-def: dprod r s == UN (x,x'):r. UN (y,y'):s. {(Scons x y, Scons x' y')} 
dsum-def: dsum r s == (UN (x,x'):r. {(In0(x),In0(x'))}) Un
        (UN (y,y'):s. {(In1(y),In1(y'))})

lemma apfst-convE:
    \[ q = apfst \sim p; \forall x y. \sim p = (x,y); \sim q = (f(x),y) \implies R \]
by (force simp add: apfst-def)

lemma Push-inject1: Push i f = Push j g ===> i=j
apply (simp add: Push-def fun-eq-iff)
apply (drule_tac x=0 in spec, simp)
done

lemma Push-inject2: Push i f = Push j g ===> f=g
apply (auto simp add: Push-def fun-eq-iff)
apply (drule-tac x=Suc x in spec, simp)
done

lemma Push-inject:
  \[ \text{Push } i f = \text{Push } j g ; \quad i=j ; \quad f=g \implies P \implies P \]
by (blast dest: Push-inject1 Push-inject2)

lemma Push-neq-K0: \text{Push} (\text{Inr} (\text{Suc} k)) f = (\%z. \text{Inr} 0) \implies P
by (auto simp add: Push-def fun-eq-iff split: nat.split_asm)


lemma Node-K0-I: (%k. Inr 0, a) : Node
by (simp add: Node-def)

lemma Node-Push-I: \text{p} : \text{Node} \implies \text{apfst} (\text{Push } i) \text{p} : \text{Node}
apply (simp add: Node-def Push-def)
apply (fast intro!: apfst-conv nat.case(2)[THEN trans])
done

21.2  Freeness: Distinctness of Constructors

lemma Scons-not-Atom [iff]: \text{Scons} M N \neq \text{Atom}(a)
unfolding Atom-def Scons-def Push-Node-def One-nat-def
  elim!: apfst-convE sym [THEN Push-neq-K0])

lemmas Atom-not-Scons [iff] = Scons-not-Atom [THEN not-sym]

lemma inj-Atom: inj(Atom)
apply (simp add: Atom-def)
apply (blast intro!: injI Node-K0-I dest!: Abs-Node-inj)
done
lemmas Atom-inject = inj-Atom [THEN injD]

lemma Atom-Atom-eq [iff]: (\text{Atom}(a)=\text{Atom}(b)) = (a=b)
by (blast dest!: Atom-inject)

lemma inj-Leaf: inj(Leaf)
apply (simp add: Leaf-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inl-inject])
done

lemmas Leaf-inject [dest] = inj-Leaf [THEN injD]

lemma inj-Numb: inj(Numb)
apply (simp add: Numb-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inr-inject])
done

lemmas Numb-inject [dest] = inj-Numb [THEN injD]

lemma Push-Node-inject:
[[ Push-Node i m = Push-Node j n; ] [ i=j; m=n ]] ==> P
[[ ] ] ==> P
apply (simp add: Push-Node-def)
apply (erule Abs-Node-inj [THEN apfst-convE])
apply (rule Rep-Node [THEN Node-Push-I])+
apply (erule sym [THEN apfst-convE])
apply (blast intro: Rep-Node-inject [THEN iffD1] trans sym elim!: Push-inject)
done

lemma Scons-inject-lemma1: Scons M N <= Scons M' N' ==> M<=M'
unfolding Scons-def One-nat-def
by (blast dest!: Push-Node-inject)

lemma Scons-inject-lemma2: Scons M N <= Scons M' N' ==> N<=N'
unfolding Scons-def One-nat-def
by (blast dest!: Push-Node-inject)

lemma Scons-inject1: Scons M N = Scons M' N' ==> M=M'
apply (erule equalityE)
apply (iprover intro: equalityI Scons-inject-lemma1)
done

lemma Scons-inject2: Scons M N = Scons M' N' ==> N=N'
apply (erule equalityE)
apply (iprover intro: equalityI Scons-inject-lemma2)
done

lemma Scons-inject:
\[
\begin{align*}
& || \text{Scons } M \ N = \text{Scons } M' \ N' ; || M=M' ; N=N' || \implies P || \implies P \\
& \text{by (iprover dest: Scons-inject1 Scons-inject2)}
\end{align*}
\]

**lemma** \text{Scons-Scons-eq} [iff]: \( (\text{Scons } M \ N = \text{Scons } M' \ N') = (M=M' \& N=N') \)

**by** (blast elim!: Scons-inject)

**lemma** \text{Scons-not-Leaf} [iff]: \( \text{Scons } M \ N \neq \text{Leaf}(a) \)

**unfolding** \text{Leaf-def o-def} **by** (rule \text{Scons-not-Atom})

**lemmas** \text{Leaf-not-Scons} [iff] = \text{Scons-not-Leaf} [THEN not-sym]

**lemma** \text{Scons-not-Numb} [iff]: \( \text{Scons } M \ N \neq \text{Numb}(k) \)

**unfolding** \text{Numb-def o-def} **by** (rule \text{Scons-not-Atom})

**lemmas** \text{Numb-not-Scons} [iff] = \text{Scons-not-Numb} [THEN not-sym]

**lemma** \text{Leaf-not-Numb} [iff]: \( \text{Leaf}(a) \neq \text{Numb}(k) \)

**by** (simp add: \text{Leaf-def Numb-def})

**lemmas** \text{Numb-not-Leaf} [iff] = \text{Leaf-not-Numb} [THEN not-sym]

**lemma** \text{ndepth-K0}: \( \text{ndepth } (\text{Abs-Node}(\%k. \text{Inr} \ 0, \ x)) = 0 \)

**by** (simp add: \text{ndepth-def Node-K0-I} [THEN \text{Abs-Node-inverse}] \text{Least-equality})

**lemma** \text{ndepth-Push-Node-aux}:

\[
\begin{align*}
& \text{case-nat } (\text{Inr } (\text{Suc } i)) \ f k = \text{Inr} \ 0 \implies \text{Suc}(\text{LEAST } x. f \ x = \text{Inr} \ 0) \leq k \\
& \text{apply (induct-tac } k, \text{auto}) \\
& \text{apply (erule Least-le)} \\
& \text{done}
\end{align*}
\]

**lemma** \text{ndepth-Push-Node}:

\[
\begin{align*}
& \text{ndepth } (\text{Push-Node } (\text{Inr } (\text{Suc } i)) \ n) = \text{Suc}(\text{ndepth}(n)) \\
& \text{apply (insert Rep-Node [of } n, \text{unfolded Node-def])] \\
& \text{apply (auto simp add: ndepth-def Push-Node-def} \\
& \text{Rep-Node [THEN Node-Push-I, THEN Abs-Node-inverse])} \\
& \text{apply (rule Least-equality)} \\
& \text{apply (auto simp add: Push-def ndepth-Push-Node-aux)}
\end{align*}
\]
apply (erule LeastI)
done

lemma ntrunc-0 [simp]: ntrunc 0 M = {}
by (simp add: ntrunc-def)

lemma ntrunc-Atom [simp]: ntrunc (Suc k) (Atom a) = Atom(a)
by (auto simp add: Atom-def ntrunc-def ndepth-K0)

unfolding Leaf-def o-def by (rule ntrunc-Atom)

lemma ntrunc-Numb [simp]:
  ntrunc (Suc k) (Numb i) = Numb(i)
unfolding Numb-def o-def by (rule ntrunc-Atom)

lemma ntrunc-Scons [simp]:
  ntrunc (Suc k) (Scons M N) = Scons (ntrunc k M) (ntrunc k N)
unfolding Scons-def ntrunc-def One-nat-def
by (auto simp add: ndepth-Push-Node)

lemma ntrunc-one-In0 [simp]: ntrunc (Suc 0) (In0 M) = {}
apply (simp add: In0-def)
apply (simp add: Scons-def)
done

lemma ntrunc-In0 [simp]: ntrunc (Suc(Suc k)) (In0 M) = In0 (ntrunc (Suc k) M)
by (simp add: In0-def)

lemma ntrunc-one-In1 [simp]: ntrunc (Suc 0) (In1 M) = {}
apply (simp add: In1-def)
apply (simp add: Scons-def)
done

lemma ntrunc-In1 [simp]: ntrunc (Suc(Suc k)) (In1 M) = In1 (ntrunc (Suc k) M)
by (simp add: In1-def)

21.3 Set Constructions

by (simp add: uprod-def)
lemma uprodE [elim!]:
 \[ c : \text{uprod} \ A \ B; \]
 \[
 !x. y. \quad \| x:A; \quad y:B; \quad c = \text{Scons} \ x \ y \| \implies P \\
 \| \implies P \\
\]
by (auto simp add: uprod-def)

lemma uprodE2: \[ \| \text{Scons} \ M \ N : \text{uprod} \ A \ B; \quad \| M:A; \quad N:B \| \implies P \| \implies P \\
\]
by (auto simp add: uprod-def)

lemma usum-In0I [intro]: \[ M:A \implies \text{In0}(M) : \text{usum} \ A \ B \]
by (simp add: usum-def)

lemma usum-In1I [intro]: \[ N:B \implies \text{In1}(N) : \text{usum} \ A \ B \]
by (simp add: usum-def)

lemma usumE [elim!]:
 \[ u : \text{usum} \ A \ B; \]
 \[
 !x. \quad \| x:A; \quad u = \text{In0}(x) \| \implies P; \\
 !y. \quad \| y:B; \quad u = \text{In1}(y) \| \implies P \\
\| \implies P \\
\]
by (auto simp add: usum-def)

lemma In0-not-In1 [iff]: \[ \text{In0}(M) \neq \text{In1}(N) \]
unfolding In0-def In1-def One-nat-def by auto

lemmas In1-not-In0 [iff] = In0-not-In1 \[ \text{THEN} \ \text{not-sym} \]

lemma In0-inject: \[ \text{In0}(M) = \text{In0}(N) \implies M=N \]
by (simp add: In0-def)

lemma In1-inject: \[ \text{In1}(M) = \text{In1}(N) \implies M=N \]
by (simp add: In1-def)

lemma In0-eq [iff]: \( \text{In0} \ M = \text{In0} \ N \) \( = (M=N) \)
by (blast dest!: In0-inject)

lemma In1-eq [iff]: \( \text{In1} \ M = \text{In1} \ N \) \( = (M=N) \)
by (blast dest!: In1-inject)
lemma inj-In0: inj In0
by (blast intro: inj-onI)

lemma inj-In1: inj In1
by (blast intro: inj-onI)

lemma Lim-inject: Lim f = Lim g ==> f = g
apply (simp add: Lim-def)
apply (rule ext)
apply (blast elim!: Push-Node-inject)
done

lemma ntrunc-subsetI: ntrunc k M <= M
by (auto simp add: ntrunc-def)

lemma ntrunc-subsetD: (!!!k. ntrunc k M <= N) ==> M <= N
by (auto simp add: ntrunc-def)

lemma ntrunc-equality: (!!k. ntrunc k M = ntrunc k N) ==> M = N
apply (rule equalityI)
apply (rule-tac [!] ntrunc-subsetD)
apply (rule-tac [!] ntrunc-subsetI [THEN [2] subset-trans], auto)
done

lemma ntrunc-o-equality:
  [| !!k. (ntrunc(k) o h1) = (ntrunc(k) o h2) || |] ==> h1 = h2
apply (rule ntrunc-equality [THEN ext!])
apply (simp add: fun-eq-iff)
done

lemma uprod-mono: [| A <= A'; B <= B' || |] ==> uprod A B <= uprod A' B'
by (simp add: uprod-def, blast)

lemma usum-mono: [| A <= A'; B <= B' || |] ==> usum A B <= usum A' B'
by (simp add: usum-def, blast)

lemma Scons-mono: [| M <= M'; N <= N' || |] ==> Scons M N <= Scons M' N'
by (simp add: Scons-def, blast)
**THEORY “Datatype”**

**lemma** `In0-mono`: \( M \leq N \Rightarrow In0(M) \leq In0(N) \)

**by** (simp add: `In0-def` `Scons-mono`)

**lemma** `In1-mono`: \( M \leq N \Rightarrow In1(M) \leq In1(N) \)

**by** (simp add: `In1-def` `Scons-mono`)

**lemma** `Split [simp]`: \( \text{Split } c \ (Scons \ M \ N) = c \ M \ N \)

**by** (simp add: `Split-def`)

**lemma** `Case-In0 [simp]`: \( \text{Case } c \ d \ (In0 \ M) = c(M) \)

**by** (simp add: `Case-def`)

**lemma** `Case-In1 [simp]`: \( \text{Case } c \ d \ (In1 \ N) = d(N) \)

**by** (simp add: `Case-def`)

**lemma** `ntrunc-UN1`: \( \text{ntrunc } k \ (\text{UN } x. f(x)) = (\text{UN } x. \text{ntrunc } k (f x)) \)

**by** (simp add: `ntrunc-def`, blast)

**lemma** `Scons-UN1-x`: \( \text{Scons } (\text{UN } x. f x) M = (\text{UN } x. \text{Scons } (f x) M) \)

**by** (simp add: `Scons-def`, blast)

**lemma** `Scons-UN1-y`: \( \text{Scons } M (\text{UN } x. f x) = (\text{UN } x. \text{Scons } M (f x)) \)

**by** (simp add: `Scons-def`, blast)

**lemma** `In0-UN1`: \( \text{In0}(\text{UN } x. f(x)) = (\text{UN } x. \text{In0}(f(x))) \)

**by** (simp add: `In0-def Scons-UN1-y`)

**lemma** `In1-UN1`: \( \text{In1}(\text{UN } x. f(x)) = (\text{UN } x. \text{In1}(f(x))) \)

**by** (simp add: `In1-def Scons-UN1-y`)

**lemma** `dprodI [intro!]`:

\[
\begin{align*}
  & (M,M') : r; (N,N') : s \quad \Rightarrow \quad (\text{Scons } M, \text{Scons } M') : \text{dprod } r \ s \\
\end{align*}
\]

**by** (auto simp add: `dprod-def`)

**lemma** `dprodE [elim!]`:

\[
\begin{align*}
  & c : \text{dprod } r \ s; \\
  & \exists x \ y \ x' \ y'. \ (x,x') : r; (y,y') : s; \\
  & c = (\text{Scons } x \ y, \text{Scons } x' \ y') \quad \Rightarrow \quad P
\end{align*}
\]
lemma dsum-In0I [intro]: (M, M') : r ==> (In0(M), In0(M')) : dsum r s
by (auto simp add: dsum-def)

lemma dsum-In1I [intro]: (N, N') : s ==> (In1(N), In1(N')) : dsum r s
by (auto simp add: dsum-def)

lemma dsumE [elim!]:
[[ w : dsum r s; 
   ![x x']. ![ (x, x') : r; w = (In0(x), In0(x')) ]] ===> P; 
   ![y y']. ![ (y, y') : s; w = (In1(y), In1(y')) ]] ===> P 
by (auto simp add: dsum-def)

lemma dprod-mono: ![ r<=r'; s<=s' ] ===> dprod r s <= dprod r' s'
by blast

lemma dsum-mono: ![ r<=r'; s<=s' ] ===> dsum r s <= dsum r' s'
by blast

lemma dprod-Sigma: (dprod (A <> B) (C <> D)) <= (uprod A C) <> (uprod B D)
by blast

lemmas dprod-subset-Sigma = subset-trans [OF dprod-mono dprod-Sigma]

lemma dprod-subset-Sigma2:
(dprod (Sigma A B) (Sigma C D)) <=
Sigma (uprod A C) (Split (%x y. uprod (B x) (D y)))
by auto

lemma dsum-Sigma: (dsum (A <> B) (C <> D)) <= (usum A C) <> (usum B D)
by blast

lemmas dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma]

hides popular names
22 Transitive-Closure: Reflexive and Transitive closure of a relation

theory Transitive-Closure
imports Relation
begin

ML-file ~~/src/Provers/trancl.ML

rtrancl is reflexive/transitive closure, trancl is transitive closure, reflcl is reflexive closure.

These postfix operators have maximum priority, forcing their operands to be atomic.

inductive-set
rtrancl :: ('a × 'a) set ⇒ ('a × 'a) set ((¬*) [1000] 999)
for r :: ('a × 'a) set
where
rtrancl-refl [intro!, Pure.intro!, simp]: (a, a) : r
| rtrancl-into-rtrancl [Pure.intro]: (a, b) : r∗==⇒ (b, c) : r ==⇒ (a, c) : r

inductive-set
trancl :: ('a × 'a) set ⇒ ('a × 'a) set ((¬+) [1000] 999)
for r :: ('a × 'a) set
where
r-into-trancl [intro, Pure.intro]: (a, b) : r==⇒ (a, b) : r+
| trancl-into-trancl [Pure.intro]: (a, b) : r+==⇒ (b, c) : r==⇒ (a, c) : r+

declare rtrancl-def [nitpick-unfold del]
| rtranclp-def [nitpick-unfold del]
| trancl-def [nitpick-unfold del]
| tranclp-def [nitpick-unfold del]

otation
| rtranclp ((¬**)) [1000] 1000) and
| tranclp ((¬++) [1000] 1000)
**Theoretical Closure**

**Abbreviation**

```
reflect :: ('a => 'a => bool) => 'a => 'a => bool     ((\=) [1000] 1000)
```

**Where**

```
r\= = \equiv \sup r \op =
```

**Abbreviation**

```
reflect :: ('a x 'a) set => ('a x 'a) set     ((\=) [1000] 999)
```

**Notation** (xsymbols)

```
\reflect (\op)? \equiv \sup r \op = \reflect (\op)
```

**Notation** (HTML output)

```
\reflect ((\op)? [1000] 1000) and
\reflect ((\op)? [1000] 1000) and
\reflect ((\op)? [1000] 999) and
\reflect ((\op)? [1000] 999) and
```

---

### 22.1 Reflexive closure

**Lemma** refl-refcl[simp]: refl(r\=)

**By** (simp add:refl-on-def)

**Lemma** antisym-refcl[simp]: antisym(r\=) = antisym r

**By** (simp add:antisym-def)

**Lemma** trans-refclII[simp]: trans \(r => trans(r\=)

**Unfolding** trans-def by blast

**Lemma** reflclp-idemp [simp]: \((P\=)\= = P\=\= =

**By** blast

---

### 22.2 Reflexive-transitive closure

**Lemma** reflcl-set-eq [pred-set-conv]: \((sup (\lambda x y. (x, y) \in r) \op =) = (\lambda x y. (x, y) \in r \cup Id)

**By** (auto simp add: fun-eq-iff)

**Lemma** r-into-rtrancl [intro]: !\!p. p \in r => p \in r\*

**Apply** (simp only: split-tupled-all)

**Apply** (erule rtrancl-refl [THEN rtrancl-into-rtrancl])
done

lemma r-into-rtranclp [intro]: \( r \ x \ y \Rightarrow r^{**} \ x \ y \)
— rtrancl of r contains r
by (erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl])

lemma rtranclp-mono: \( r \leq s \Rightarrow r^{**} \leq s^{**} \)
— monotonicity of rtrancl
apply (erule rtranclp.induct)
apply (erule rtranclp.induct)
apply (rule_tac [2] rtranclp.rtrancl-into-rtrancl, blast+)
done

lemma rtrancl-mono = rtranclp-mono [to-set]

theorem rtranclp-induct [consumes 1, case-names base step, induct set: rtranclp]:
assumes a: \( r^{**} \ a \ b \)
and cases: \( P \ a \ ) || \( r^{**} \ a \ y ; r y z ; P y || \Rightarrow P z \)
shows \( P b \) using a
by (induct x≡a b) (rule cases)+

lemma rtranclp-induct [induct set: rtrancl] = rtranclp-induct [to-set]

lemmas rtranclp-induct2 =
rtranclp-induct[of - (ax,ay) (bx,by), split-rule, consumes 1, case-names refl step]
lemmas rtranclp-induct2 =
rtranclp-induct[of (ax,ay) (bx,by), split-format (complete), consumes 1, case-names refl step]

lemma refl-rtrancl: refl (r^{*})
by (unfold refl-on-def) fast

Transitivity of transitive closure.

lemma trans-rtrancl: trans (r^{*})
proof (rule rtrancl)
fix \( x \ y \ z \)
assume \( (x, y) \in r^{*} \)
assume \( (y, z) \in r^{*} \)
then show \( (x, z) \in r^{*} \)
proof induct
  case base
  show \( (x, y) \in r^{*} \) by fact
next
  case (step u v)
  from \( (x, u) \in r^{*} \) and \( (u, v) \in r \)
  show \( (x, v) \in r^{*} \)
qed
lemmas rtrancl-trans = trans-rtrancl [THEN transD]

lemma rtranclp-trans:
  assumes xy: r^** x y
  and yz: r^** y z
  shows r^** x z using yz xy
  by (erule asmD)

lemma rtranclp-trans:
  assumes xy: r^** x y
  and yz: r^** y z
  shows r^** x z using yz xy
  by (erule asmD)

lemma rtranclE [cases set: rtrancl]:
  assumes major: (a::'a, b) : r^*
  obtains
    (base) a = b
    | (step) y where (a, y) : r^* and (y, b) : r
     — elimination of rtrancl — by induction on a special formula
  apply (erule rtranclp-elim)
  apply blast
  done

lemma rtranclp-idemp [simp]: (r^**)^** = r^**
  by (rule rtranclp-refl)

lemmas rtrancl-idemp = rtranclp-idemp [to-set]

More r^* equations and inclusions.

lemma rtrancp-idemp [simp]: (r^**)^** = r^**
  by (erule rtranclp-refl)

lemmas rtrancl-idemp = rtranclp-idemp [to-set]

lemma rtrancl-idemp-self-comp [simp]: R^* O R^* = R^*
  by (auto simp only: rtranclp-trans)

prefer 2 apply blast
done
done

lemma rtrancl-subset-rtrancl: \( r \subseteq s^* \Rightarrow r^* \subseteq s^* \)
apply (drule rtrancl-mono)
apply simp
done

lemma rtranclp-subset: \( R \leq S \Rightarrow S \leq R^{**} \Rightarrow \sup R S = R^{**} \)
apply (drule rtranclp-mono)
apply (drule rtranclp-mono)
apply simp
done

lemmas rtrancl-subset = rtranclp-subset [to-set]

lemma rtranclp-sup-rtranclp: \( \sup (R^**) (S^**) = (\sup R S)^{**} \)
by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

lemmas rtrancl-Un-rtrancl = rtranclp-sup-rtranclp [to-set]

lemma rtranclp-reflclp [simp]: \( R^* = R^{**} \)
by (blast intro!: rtranclp-subset)

lemmas rtrancl-reflcl = rtranclp-reflclp [to-set]

lemma rtrancl-r-diff-Id: \( (r - Id)^* = r^* \)
apply (rule sym)
apply (rule rtrancl-subset, blast, clarify)
apply (rename-tac a b)
apply (case-tac a = b)
apply blast
apply blast
done

lemma rtranclp-r-diff-Id: \( \inf r op \sim = \) \( R^{**} = r^{**} \)
apply (rule sym)
apply (rule rtranclp-subset)
apply blast+
done

theorem rtranclp-converseD:
assumes \( r : (r^{-1})^{**} x y \)
shows \( r^{**} y x \)
proof -
  from r show \( \)thesis
  by induct (iprover intro: rtranclp-trans dest!: conversepD)+
  qed

lemmas rtrancl-converseD = rtranclp-converseD [to-set]
theorem rtranclp-converseI:
  assumes \( r^{\ast\ast} y x \)
  shows \( (r^{-\ast} \ast)^{\ast\ast} x y \)
  using assms
by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse: \((r^{-\ast})^\ast = (r^{\ast})^{-\ast}\)
by (fast dest!: rtrancl-converseD intro!: rtrancl-converseD)

lemma sym-rtrancl: sym r == sym (r^{\ast})
by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct [consumes 1, case-names base step]:
  assumes major: \( r^{\ast\ast} a b \)
  and cases: \( P b \) \( y z \) \( r y z \) \( r^{\ast\ast} z b \); \( P z \) \( y \)
  shows \( P a \)
using rtranclp-converseI [OF major]
by induct (iprover intro: cases dest!: converse-rtranclp-induct)

lemmas converse-rtrancl-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of \((ax,ay)\) \((bx,by)\), split-rule,
  consumes 1, case-names refl step]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of \((ax,ay)\) \((bx,by)\), split-format (complete),
  consumes 1, case-names refl step]

lemma converse-rtranclpE [consumes 1, case-names base step]:
  assumes major: \( r^{\ast\ast} x z \)
  and cases: x=x \( y \) \( r x y \) \( r^{\ast\ast} y z \) \( y \)
  shows \( P \)
apply (subgoal-tac x = z | (EX y. r x y & r^{\ast\ast} y z))
apply (rule-tac [2] major [THEN converse-rtranclp-induct])
pref 2 apply iprover
pref 2 apply iprover
apply (erule asm-rl exE disjE conjE cases)+
done

lemmas converse-rtranclE = converse-rtranclpE [to-set]

lemmas converse-rtranclpE2 = converse-rtranclpE [of \((xa,xb)\) \((xa,zb)\), split-rule]

lemmas converse-rtranclE2 = converse-rtranclE [of \((xa,xb)\) \((za,zb)\), split-rule]
lemma r-comp-rtrancl-eq: \( r O r^* = r^* O r \)
by (blast elim: rtranclE converse-rtranclE
   intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl)

lemma rtrancl-unfold: \( r^* = Id Un r^* O r \)
by (auto intro: rtrancl-into-rtrancl elim: rtranclE)

lemma rtrancl-Un-separatorE:
\[(a,b) : (P \cup Q)^* \Rightarrow \forall x y. (a,x) : P^* \rightarrow (x,y) : Q \rightarrow x=y \Rightarrow (a,b) : P^*\]
apply (induct rule: rtrancl.induct)
apply blast
apply (blast intro: rtrancl-trans)
done

lemma rtrancl-Un-separator-converseE:
\[(a,b) : (P \cup Q)^* \Rightarrow \forall x y. (x,b) : P^* \rightarrow (y,x) : Q \rightarrow y=x \Rightarrow (a,b) : P^*\]
apply (induct rule: converse-rtrancl-induct)
apply blast
apply (blast intro: rtrancl-trans)
done

lemma Image-closed-trancl:
assumes \( r \subseteq X \)
shows \( r^+ \subseteq X = X \)
proof -
  from assms have **: \{y. \exists x \in X. (x, y) \in r\} \subseteq X by auto
  have \( \forall x y. (y, x) \in r^+ \Rightarrow y \in X \Rightarrow x \in X \)
  proof -
    fix x y
    assume *: \( y \in X \)
    assume \( (y, x) \in r^* \)
    then show \( x \in X \)
    proof
      case base show ?case by (fact *)
    next
      case step with ** show ?case by auto
    qed
  qed
  then show ?thesis by auto
qed

22.3 Transitive closure

lemma trancl-mono: \( \forall p. p \in r^+ \Rightarrow r \subseteq s \Rightarrow p \in s^+ \)
apply (simp add: split-tupled-all)
apply (erule trancl.induct)
apply (iprover dest: subsetD)
done
lemma r-into-trancl': \( \forall p. \ p : r \implies p : r^++ \)
by (simp only: split-tupled-all) (erule r-into-trancl)

Conversions between trancl and rtrancl.

lemma tranclp-into-rtranclp: \( r^++ a b \implies r^{++} a b \)
by (erule tranclp.induct) iprover+

lemmas trancl-into-rtrancl = tranclp-into-rtranclp [to-set]

lemma rtranclp-into-tranclp1: assumes \( r \) shows \( c \).
by induct iprover+

lemmas rtranclp-into-rtranclp1 = rtranclp-into-tranclp1 [to-set]

lemma rtranclp-into-tranclp2: \( \mid r a b; r^{++} b c \mid \implies r^{++} a c \)
— intro rule from \( r \) and rtrancl
apply (erule rtranclp.cases)
apply iprover
apply (rule rtranclp-trans [THEN rtranclp-into-tranclp1])
apply (simp | rule r-into-rtranclp)+ done

lemmas rtranclp-into-tranclp2 = rtranclp-into-tranclp2 [to-set]

Nice induction rule for trancl

lemma tranclp-induct [consumes 1, case-names base step, induct pred: tranclp]:
assumes \( a: r^++ a b \)
and cases: \( \forall y. r a y \implies P y \)
\( \forall y z. r^{++} a y \implies r y z \implies P y \implies P z \)
shows \( P b \) using \( a \)
by (induct \( x \equiv a b \)) (iprover intro: cases)+

lemmas tranclp-induct = tranclp-induct [to-set]

lemmas tranclp-induct2 = tranclp-induct [of \( \langle ax, ay \rangle \langle bx, by \rangle \), split-rule, consumes 1, case-names base step]

lemmas tranclp-induct2 = tranclp-induct [of \( \langle ax, ay \rangle \langle bx, by \rangle \), split-format (complete), consumes 1, case-names base step]

lemma tranclp-trans-induct:
assumes \( r^{++} x y \)
and cases: \( \forall x y. r x y \implies P x y \)
\( \forall x y z. \mid r^{++} x y; P x y; r^{++} y z; P y z \mid \implies P x z \)
shows \( P x y \)
— Another induction rule for trancl, incorporating transitivity
by (iprover intro: major \[THEN\ tranclp-induct\] cases)

lemmas trancl-trans-induct = tranclp-trans-induct \[to-set\]

lemma tranclE \[cases set: trancl\]:
  assumes \((a, b) : r^+\)
  obtains
    \[\text{(base)} (a, b) : r\]
    \[| \text{(step)} c \text{ where } (a, c) : r^+ \text{ and } (c, b) : r\]
  using assms by cases simp-all

lemma trancl-Int-subset: \[| r \subseteq s; (r^+ \cap s) O r \subseteq s | \] \[==>\] \(r^+ \subseteq s\)
  apply ((rule subsetI)
  apply (rule-tac p = x in PairE)
  apply clarify
  apply (erule trancl-induct)
  apply auto
  done

lemma trancl-unfold: \(r^+ = r \cup r^+ \cup r\)
  by (auto intro: trancl-into-trancl elim: tranclE)

Transitivity of \(r^+\)

lemma trans-trancl \[simp\]: \(\text{trans } (r^+)\)
  proof (rule transI)
    fix \(x\ y\ z\)
    assume \((x, y) \in r^+\)
    assume \((y, z) \in r^+\)
    then show \((x, z) \in r^+\)
      proof induct
        case (base u)
        from \((x, y) \in r^+\) and \((y, u) \in r\)
        show \((x, u) \in r^+\) ..
      next
        case (step u v)
        from \((x, u) \in r^+\) and \((u, v) \in r\)
        show \((x, v) \in r^+\) ..
      qed
    qed

lemmas trancl-trans = trans-trancl \[THEN\ transD\]

lemma tranclp-trans:
  assumes xy: \(r^{++} x\ y\)
  and yz: \(r^{++} y\ z\)
  shows \(r^{++} x\ z\) using yz xy
  by induct iprover+

lemma trancl-id \[simp\]: \(\text{trans } r \Longrightarrow r^+ = r\)
apply auto
apply (erule trancl-induct)
apply assumption
apply (unfold trans-def)
apply blast
done

lemma rtranclp-tranclp-tranclp:
assumes r** x y
shows !!z. r+++ y z ==> r+++ x z using assms
by induct (iprover intro: tranclp-trans)+

lemmas rtrancl-trancl-trancl = rtranclp-tranclp-tranclp [to-set]

lemma tranclp-into-trancl2: r a b ==> r+++ b c ==> r+++ a c
by (erule tranclp-trans [OF tranclp.r-into-trancl])

lemmas trancl-into-trancl2 = tranclp-into-trancl2 [to-set]

lemma tranclp-converseI: (r-1)+++1 x y ==> (r-1)+ +1 x y
apply (erule tranclp-induct)
apply (iprover intro: conversepI tranclp-trans)+
done

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD: (r+-1)+ +1 x y ==> (r-1)+ +1 x y
apply (rule conversepI)
apply (erule tranclp-induct)
apply (iprover dest: conversepD intro: tranclp-trans)+
done

lemmas trancl-converseD = tranclp-converseD [to-set]

lemma tranclp-converse: (r-1)+ +1 = (r-1)+ +1
by (fastforce simp add: fun-eq-iff
intro!: tranclp-converseI dest!: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl: sym r ==> sym (r+)
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

lemma converse-tranclp-induct [consumes 1, case_names base step]:
assumes major: r+++ a b
and cases: !!y. r y b ==> P(y)
!!y z. r y z; r+++ z b; P(z) [] ==> P(y)
shows P a
apply (rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major])
apply (rule cases)
apply (erule conversepD)
apply (blast intro: assms dest!: tranclp-converseD)
done

lemmas converse-trancl-induct = converse-tranclp-induct [to-set]

lemma tranclD: \( R^{++} x y \Rightarrow \exists z. \ R z z \land R^{**} z y \)
apply (erule converse-tranclp-induct)
apply auto
apply (blast intro: rtranclp-trans)
done

lemmas tranclD = tranclpD [to-set]

lemma converse-tranclpE:
assumes major: \( \text{trancl} \ r \ x \ z \)
assumes base: \( \forall \ x \ z \Rightarrow \ P \)
assumes step: \( \forall \ y. \ [\forall r \ y \ z; \ \text{trancl} \ r \ y \ z] \Rightarrow \ P \)
shows \( P \)
proof
from tranclpD[OF major]
obtain y where \( r x y \) and \( rtranclp r y z \) by iprover
from this(2) show \( P \)
proof (cases rule: rtranclp.cases)
  case rtrancl-refl
  with \( r x y \) base show \( P \) by iprover
next
  case rtrancl-into-rtrancl
  from this have \( r x y \) by (iprover intro: rtranclp-into-trancl)
  with \( r x y \) step show \( P \) by iprover
qed

lemmas converse-tranclE = converse-tranclpE [to-set]

lemma tranclD2:
\( (x, y) \in R^+ \Rightarrow \exists z. \ (x, z) \in R^* \land (z, y) \in R \)
by (blast elim: tranclE intro: trancl-into-rtrancl)

lemma irrefl-tranclI: \( r^{-1} \cap r^{**} = \{\} \Rightarrow (x, x) \notin r^+ \)
by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma irrefl-trancl-rD: \( \forall X. \ (x, y) \notin r^+ \Rightarrow (x, y) \in r \Rightarrow x \neq y \)
by (blast dest: r-into-trancl)

lemma trancl-subset-Sigma-aux:
\( (a, b) \in r^* \implies r \subseteq A \times A \implies a = b \lor a \in A \)

**Lemma** transcl-subset-Sigma: \( r \subseteq A \times A \implies r^+ \subseteq A \times A \)

**Proof**
- **apply** \( \text{rule subsetI} \)
- **apply** \( \text{simpl only: split-tupled-all} \)
- **apply** \( \text{erule tranclE} \)
- **apply** \( \text{blast dest!: trancl-into-rtrancl trancl-subset-Sigma-aux} \)
- **done**

**Lemma** reflclp-tranclp [simp]: \( (r^++)^* = r^{**} \)

**Proof**
- **apply** \( \text{safe intro!: order-antisym} \)
- **apply** \( \text{erule tranclp-into-rtranclp} \)
- **apply** \( \text{blast elim: rtranclp.cases dest: rtranclp-into-tranclp1} \)
- **done**

**Lemmas** reflcl-trancl [simp] = reflclp-tranclp [to-set]

**Lemma** trancl-reflcl [simp]: \( (r^=)^+ = r^* \)

**Proof**
- **apply** \( \text{safe} \)
- **apply** \( \text{erule trancl-into-rtrancl, simp} \)
- **apply** \( \text{erule rtranclE, safe} \)
- **apply** \( \text{erule rtrancl-reflcl [THEN equalityD2, THEN subsetD], fast} \)
- **done**

**Lemma** rtrancl-trancl-reflcl [code]: \( r^{**} = (r^+)^* \)

**Proof**
- **by simp**

**Lemma** trancl-empty [simp]: \( \{\}^+ = \{\} \)

**Proof**
- **by** \( \text{auto elim: trancl-induct} \)

**Lemma** rtrancl-empty [simp]: \( \{\}^* = 1d \)

**Proof**
- **by** \( \text{rule subst [OF reflcl-trancl] simp} \)

**Lemma** rtranclpD: \( R^{**} a b \implies a = b \lor a \neq b \land R^{+++} a b \)

**Proof**
- **by** \( \text{force simp add: reflclp-tranclp [symmetric] simp del: reflclp-tranclp} \)

**Lemmas** tranclD = rtranclpD [to-set]

**Lemma** rtrancl-eq-or-trancl:
\( (x,y) \in R^* = (x=y \lor x\neq y \land (x,y) \in R^+) \)

**Proof**
- **by** \( \text{fast elim: trancl-into-rtrancl dest: tranclD} \)

**Lemma** trancl-unfold-right: \( r^{++} = r^* O r \)

**Proof**
- **by** \( \text{auto dest: tranclD2 intro: trancl-into-trancl1} \)

**Lemma** trancl-unfold-left: \( r^+ = r O r^* \)
 Theory "Transitive-Closure"

by (auto dest: tranclD intro: rtrancl-into-trancl2)

lemma trancl-insert:
(insert (y, x) r) \^+ = r \^+ \cup \{(a, b). (a, y) \in r \^* \land (x, b) \in r \^*)\}
— primitive recursion for trancl over finite relations
apply (rule equalityI)
apply (rule subsetI)
apply (simp only: split-tupled-all)
apply (erule trancl-induct, blast)
apply (rule subsetI)
apply (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl trancl-trans)
apply (rule subsetI)
apply (blast intro: trancl-mono rtrancl-mono)
apply (rule rtrancl-trancl-trancl rtrancl-into-trancl2)
done

lemma trancl-insert2:
(insert (a, b) r) \^+ = r \^+ \cup \{(x, y). ((x, a) \in r \^* \lor x = a) \land ((b, y) \in r \^* \lor y = b)\}
by (auto simp add: trancl-insert rtrancl-eq-or-trancl)

lemma rtrancl-insert:
(insert (a, b) r) \^* = r \^* \cup \{(x, y). (x, a) : r \^* \land (b, y) \in r \^*)\}
using trancl-insert[of a b r]
by (simp add: rtrancl-trancl-reflcl del: reflcl-trancl) blast

Simplifying nested closures

lemma rtrancl-trancl-absorb[simp]: (R \^*) \^+ = R \^*
by (simp add: trans-rtrancl)

lemma trancl-rtrancl-absorb[simp]: (R \^+) \^* = R \^*
by (subst reflcl-trancl[ symmetric]) simp

lemma rtrancl-reflcl-absorb[simp]: (R \^*) \^* = R \^*
by auto

Domain and Range

lemma Domain-rtrancl [simp]: Domain (R \^*) = UNIV
by blast

lemma Range-rtrancl [simp]: Range (R \^*) = UNIV
by blast

lemma rtrancl-Un-subset: (R \^* \cup S \^*) \subseteq (R \cup S) \^*
by (rule rtrancl-Un-rtrancl [ THEN subst]) fast

lemma in-rtrancl-UnI: x \in R \^* \lor x \in S \^* \Longrightarrow x \in (R \cup S) \^*
by (blast intro: subsetD [OF rtrancl-Un-subset])

lemma trancl-domain [simp]: Domain (r \^+) = Domain r
by (unfold Domain-unfold) (blast dest: tranclD)
**THEORY “Transitive-Closure”**

lemma trancl-range [simp]: \( \text{Range } (r^+) = \text{Range } r \)
unfolding Domain-converse [symmetric] by (simp add: trancl-converse [symmetric])

lemma Not-Domain-rtrancl:
\( x \sim: \text{Domain } R \Longrightarrow ((x, y) : R^*) = (x = y) \)
apply auto
apply (erule rev-mp)
apply (erule rtrancl-induct)
apply auto
done

lemma trancl-subset-Field2: \( r^+ \subseteq \text{Field } r \times \text{Field } r \)
apply clarify
apply (erule trancl-induct)
apply (auto simp add: Field-def)
done

lemma finite-trancl[simp]: finite \( (r^+) \) = finite \( r \)
apply auto
prefer 2
apply (rule trancl-subset-Field2 [THEN finite-subset])
apply (rule finite-SigmaI)
prefer 3
apply (blast intro: r-into-trancl' finite-subset)
apply (auto simp add: finite-Field)
done

More about converse \( rtrancl \) and \( trancl \), should be merged with main body.

lemma single-valued-confluent:
\[ \text{single-valued } r; (x,y) \in r^*; (x,z) \in r^* \Rightarrow (y,z) \in r^* \lor (z,y) \in r^* \]
apply (erule rtrancl-induct)
apply simp
apply (erule disjE)
apply (blast elim: converse-rtranclE dest:single-valuedD)
apply (blast intro:rtrancl-trans)
done

lemma r-r-into-trancl: \((a, b) \in R \Longrightarrow (b, c) \in R \Longrightarrow (a, c) \in R^+ \)
by (fast intro: trancl-trans)

lemma trancl-into-trancl [rule-format]:
\((a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a,c) \in r^+ \)
apply (erule trancl-induct)
apply (fast intro: r-r-into-trancl)
apply (fast intro: r-r-into-trancl trancl-trans)
done
lemma tranclp-rtranclp-tranclp:
  \( r^{++} a b \implies r^{**} b c \implies r^{++} a c \)
  apply (drule tranclpD)
  apply (elim exE conjE)
  apply (drule rtranclp-trans, assumption)
  apply (drule rtranclp-into-tranclp2, assumption, assumption)
done

lemmas trancl-rtrancl-trancl = tranclp-rtranclp-tranclp [to-set]

lemmas transitive-closure-trans [trans] =
  r-r-into-trancl trancl-trans
  trancl.trancl-into-trans trancl-into-trancl
  rtrancl.rtrancl-into-trancl converse-rtrancl-into-rtrancl
  rtrancl.trancl-trancl trancl-rtrancl-trancl

lemmas transitive-closurep-trans [trans] =
  tranclp-trans rtranclp-trans
  tranclp.tranclp-into-trancl tranclp-into-tranclp
  rtranclp.rtranclp-into-rtrancl converse-rtranclp-into-rtranclp
  rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare trancl-into-rtrancl [elim]

22.4 The power operation on relations

\( R ^{\circ n} = R O ... O R \), the n-fold composition of \( R \)

overloading
  relpow :: nat \to (\times a) set \to (\times a) set
  relpowp :: nat \to (\times a) set \to (\times a) set \to bool

begin
  primrec relpow :: nat \to (\times a) set \to (\times a) set
    where
    relpow 0 R = Id
    | relpow (Suc n) R = (R ^n O R)

  primrec relpowp :: nat \to (\times a) set \to (\times a) set \to bool
    where
    relpowp 0 R = HOL.eq
    | relpowp (Suc n) R = (R ^n OO R)
  end

lemma relpowp-relpow-eq [pred-set-conv]:
  fixes R :: 'a rel
  shows \( (\lambda x. (x, y) \in R) ^n = (\lambda x. (x, y) \in R) ^n \)
  by (induct n) (simp-all add: relcompp-relcomp-eq)

for code generation

definition relpow :: nat \to (\times a) set \to (\times a) set
    where
relpow-code-def [code-abbrev]: relpow = compow

definition relpowp :: nat ⇒ (′a ⇒ ′a ⇒ bool) ⇒ (′a ⇒ ′a ⇒ bool) where
relpowp-code-def [code-abbrev]: relpowp = compow

lemma [code]:
relpow (Suc n) R = (relpow n R) O R
relpow 0 R = Id
by (simp-all add: relpow-code-def)

lemma [code]:
relpowp (Suc n) R = (R ^^ n) OO R
relpowp 0 R = HOL.eq
by (simp-all add: relpowp-code-def)

hide-const (open) relpow
hide-const (open) relpowp

lemma relpow-1 [simp]:
fixes R :: (′a × ′a) set
shows R ^^ 1 = R
by simp

lemma relpowp-1 [simp]:
fixes P :: ′a ⇒ ′a ⇒ bool
shows P ^^ 1 = P
by (fact relpow-1 [to-pred])

lemma relpow-0-I:
(x, x) ∈ R ^^ 0
by simp

lemma relpowp-0-I:
(P ^^ 0) x x
by (fact relpowp-0-I [to-pred])

lemma relpow-Suc-I:
(x, y) ∈ R ^^ n ⇒ (y, z) ∈ R ⇒ (x, z) ∈ R ^^ Suc n
by auto

lemma relpowp-Suc-I:
(P ^^ n) x y ⇒ P y z ⇒ (P ^^ Suc n) x z
by (fact relpowp-Suc-I [to-pred])

lemma relpowp-Suc-I2:
(x, y) ∈ R ⇒ (y, z) ∈ R ^^ n ⇒ (x, z) ∈ R ^^ Suc n
by (induct n arbitrary: z) (simp, fastforce)

lemma relpowp-Suc-I2:
lemma relpow-0-E:
\((x, y) \in R \Rightarrow (x = y \Rightarrow P) \Rightarrow P\)
by simp

lemma relpowp-0-E:
\((P \Rightarrow 0) x y \Rightarrow (x = y \Rightarrow Q) \Rightarrow Q\)
by (fact relpow-0-E [to-pred])

lemma relpowp-Suc-E:
\((P \Rightarrow \text{Suc } n) x z \Rightarrow (\forall y. (P \Rightarrow n) x y \Rightarrow P y z) \Rightarrow Q) \Rightarrow Q\)
by (fact relpow-Suc-E2 [to-pred])

lemma relpowp-Suc-D2:
\((P \Rightarrow \text{Suc } n) x z \Rightarrow (P \Rightarrow n x z) \Rightarrow (\exists y. (P \Rightarrow n) y z) \Rightarrow Q)\)
by (fact relpowp-Suc-D2 [to-pred])

lemma relpowp-Suc-D2:
\((P \Rightarrow \text{Suc } n) x z \Rightarrow (P \Rightarrow n x z) y z\)
by (fact relpow-Suc-D2 [to-pred])

lemma relpow-Suc-E2:
\((x, z) \in R \Rightarrow (\forall y. (P \Rightarrow n) x y \Rightarrow P) \Rightarrow P\)
by (blast dest: relpow-Suc-D2)

lemma relpow-Suc-E2:
\((P \Rightarrow \text{Suc } n) x z \Rightarrow (\forall y. (P \Rightarrow n) y z) \Rightarrow Q) \Rightarrow Q\)
by (fact relpow-Suc-E2 [to-pred])
lemma relpow-Suc-D2′:
\(\forall x y z. (x, y) \in R \wedge (y, z) \in R \rightarrow (\exists w. (x, w) \in R \wedge (w, z) \in R)\)
by (induct n) (simp-all, blast)

lemma relpowp-Suc-D2′:
\(\forall x y z. (P \wedge (P \wedge (P \wedge z)) \rightarrow (\exists w. P w z \wedge (P \wedge w))\)
by (fact relpow-Suc-D2′ [to-pred])

lemma relpow-E2:
\((x, z) \in R \rightarrow (n = 0 \rightarrow x = z \rightarrow P)\)
\(\rightarrow (\forall y m. n = Suc m \rightarrow (x, y) \in R \rightarrow (y, z) \in R \rightarrow P)\)
\(\rightarrow P\)
apply (cases n, simp)
apply (rename-tac nat)
apply (cut-tac n = nat and \(R = R\) in relpow-Suc-D2′, simp, blast)
done

lemma relpowp-E2:
\(P \wedge (P \wedge m) \rightarrow Q\)
\(\rightarrow (\forall y m. n = Suc m \rightarrow P x y \rightarrow (P \wedge m) y z \rightarrow Q)\)
\(\rightarrow Q\)
by (fact relpow-E2 [to-pred])

lemma relpow-add: \(R \wedge (m + n) = R \wedge m O R \wedge n\)
by (induct n) auto

lemma relpowp-add: \(P \wedge (m + n) = P \wedge m OO P \wedge n\)
by (fact relpow-add [to-pred])

lemma relpow-commute: \(R O R \wedge n = R \wedge n O R\)
by (induct n) (simp, simp add: O-assoc [symmetric])

lemma relpow-commute: \(P OO P \wedge n = P \wedge n OO P\)
by (fact relpow-commute [to-pred])

lemma relpow-empty:
\(\emptyset < n \rightarrow (\emptyset : (\emptyset \times \emptyset) \text{ set}) \wedge n = \emptyset\)
by (cases n) auto

lemma relpow-bot:
\(\emptyset < n \rightarrow (\bot : \text{ 'a \Rightarrow 'a \Rightarrow bool}) \wedge n = \bot\)
by (fact relpow-empty [to-pred])

lemma rtrancl-imp-UN-relpow:
assumes \(p \in R^*\)
shows \(p \in (\bigcup n. R \wedge n)\)
proof (cases p)
  case (Pair x y)

with assms have \((x, y) \in R^*\) by simp
then have \((x, y) \in (\bigcup n. R ^\circ n)\) proof induct
  case base show ?case by (blast intro: relpow-0-I)
next
  case step then show ?case by (blast intro: relpow-Suc-I)
qed
with Pair show ?thesis by simp
qed

lemma rtranclp-imp-Sup-relpowp:
  assumes \((P ^\circ\circ) x y\)
  shows \((\bigcup n. P ^\circ n) x y\)
  using assms and rtrancl-imp-UN-relpow [to-pred] by blast

lemma relpow-imp-rtrancl:
  assumes \(p \in R ^\circ n\)
  shows \(p \in R^*\)
  proof (cases p)
    case \((Pair x y)\)
    with assms have \((x, y) \in R ^\circ n\) by simp
    then have \((x, y) \in R^*\) proof (induct n arbitrary: x y)
      case 0 then show ?case by simp
    next
      case Suc then show ?case by (blast elim: relpow-Suc-E intro: rtrancl-into-rtrancl)
    qed
  with Pair show ?thesis by simp
qed

lemma relpowp-imp-rtranclp:
  assumes \((P ^\circ n) x y\)
  shows \((P ^\circ\circ) x y\)
  using rtrancl-is-UN-relpow [to-pred] by auto

lemma rtrancl-power:
  \(p \in R^* \iff (\exists n. p \in R ^\circ n)\)
  by (simp add: rtrancl-is-UN-relpow)

lemma rtranclp-power:
  \((P ^\circ\circ) x y \iff (\exists n. (P ^\circ n) x y)\)
  by (simp add: rtranclp-is-Sup-relpowp)
lemma trancl-power:
\( p \in R^* \iff (\exists n > 0. \; p \in R^n) \)
apply (cases p)
apply simp
apply (rule iffI)
apply (drule tranclD2)
apply (clarsimp simp: rtrancl-is-UN-relpow)
apply (rule_tac x=Suc n in exI)
apply (clarsimp simp: relcomp-unfold)
apply fastforce
apply clarsimp
apply (case_tac n, simp)
apply (clarsimp)
apply (drule relpow-imp-rtrancl)
apply (drule rtrancl-into-trancl1)
apply auto
done

lemma tranclp-power:
\( (P^{++}) x y \iff (\exists n > 0. \; (P^n) x y) \)
using trancl-power[to-pred, of P (x, y)] by simp

lemma rtrancl-imp-relpow:
\( p \in R^* \Rightarrow (\exists n. \; p \in R^n) \)
by (auto dest: rtrancl-imp-UN-relpow)

lemma rtranclp-imp-relpowp:
\( (P^{**}) x y \Rightarrow (\exists n. \; (P^n) x y) \)
by (auto dest: rtranclp-imp-Sup-relpowp)

By Sternagel/Thiemann:

lemma relpow-fun-conv:
\( ((a,b) \in R^n) = (\exists f. f 0 = a \land f n = b \land (\forall i < n. \; (f i, f(Suc i)) \in R)) \)
proof (induct n arbitrary: b)
case 0 show ?case by auto
next
case (Suc n)
show ?case
proof (simp add: relcomp-unfold Suc)
  show \( (\exists y. (\exists f, f 0 = a \land f n = y \land (\forall i < n. \; (f i, f(Suc i)) \in R)) \land (y, b) \in R) \)
  = \( (\exists f, f 0 = a \land f(Suc n) = b \land (\forall i < Suc n. \; (f i, f(Suc i)) \in R)) \)
(is ?l = ?r)
proof
  assume ?l
  then obtain c f where 1: \( f 0 = a \land f n = c \land i < n \Rightarrow (f i, f(Suc i)) \in R \)
  \( (c, b) \in R \) by auto
  let ?g = \( \lambda m. \; \text{if} \; m = Suc n \; \text{then} \; b \; \text{else} \; f m \)
  show ?r by (rule exI[of - ?g], simp add: 1)
next
assume ?r
then obtain f where 1: f 0 = a b = f (Suc n) \( \forall i. i < Suc n \implies (f i, f (Suc i)) \in R \) by auto
show ?l by (rule exI[of - f n], rule conjI, rule exI[of - f], insert 1, auto)
qed
qed

lemma relpow-fun-conv:
  \((P \multimap n) x y \iff (\exists f. f 0 = x \land f n = y \land (\forall i<n. P (f i, f (Suc i))))\)
by (fact relpow-fun-conv [to-pred])

lemma relpow-finite-bounded1:
assumes finite \((\sim k \subseteq (UN n:\{n. 0 < n \& n <= card R\}. R \multimap n)) (is - \subseteq ?r)\)
shows \(R \multimap k \subseteq (UN n:\{n. 0 < n \& n <= card R\}. R \multimap n)\) (is - \subseteq ?r)
proof -
{ fix a b k
  have \((a, b) : R \multimap (Suc k) \implies EX n. 0 < n \& n <= card R \& (a, b) : R \multimap n\)
  proof (induct k arbitrary: b)
    case 0
    hence \(R \neq \{\}\ \) by auto
    with card-0-eq[OF finite R] have \(card R >= Suc 0\) by auto
    thus ?case using 0 by force
  next
    case (Suc k)
    then obtain a' where \((a, a') : R \multimap (Suc k) \land (a', b) : R \multimap b\)
    from Suc(1)[OF \((a, a') : R \multimap (Suc k)\)]
    obtain n where \(n \leq card R \land (a, a') \in R \multimap n\) by auto
    have \((a, b) : R \multimap (Suc n)\) using \((a, a') \in R \multimap n \land (a', b) \in R\) by auto
    { assume \(n < card R\)
      hence ?case using \((a, b) : R \multimap (Suc n)\) Suc-leI[OF \(\{n. card R < n\}\)] by blast
    } moreover
    { assume \(n = card R\)
      from \((a, b) \in R \multimap (Suc n)\) [unfolded relpow-fun-conv]
      obtain f where \(f 0 = a \land f (Suc n) = b\)
      and steps: \(\forall i. i <= n \implies (f i, f (Suc i)) \in R \land by auto\)
      let ?p = \(\%i. (f i, f (Suc i))\)
      let ?N = \(\{i. i < n\}\)
      have \(?p \multimap ?N \subseteq R\) using steps by auto
      from card-mono[OF assms(1) this]
      have card(?p \multimap ?N) <= card R.
      also have \(\ldots < card ?N\) using \(n = card R\) by simp
    } finally have \(~ inj-on ?p \multimap ?N\) by (rule pigeonhole)
    then obtain i j where \(i <= n \land j <= n \land ij: i \neq j \land ij\)
    pij: ?p i = ?p j by (auto simp; inj-on-def)
    let \(\hat{i} = min i j\) and \(\hat{j} = max i j\)
    have \(i: \hat{i} <= n \land j: \hat{j} <= n \land pij: \hat{i} <= n \land pij: \hat{i} \neq \hat{j}\)
    and \(ij: \hat{i} < \hat{j}\)
    using \(i j ij pij\) unfolding min-def max-def by auto
from $i$ $j$ $p_{ij}$ $ij$ obtain $i$ $j$ where $i$ $\leq$ $n$ and $j$ $\leq$ $n$ and $ij$ $i$ $<$ $j$

and $p_{ij}$: $?p$ $i$ $=$ $?p$ $j$ by blast

let $?g$ = $\lambda$ $l$. if $l$ $\leq$ $i$ then $f$ $l$ else $f$ $(l + (j - i))$

let $?n$ = $\text{Suc}(n - (j - i))$

have abl: $(a, b) \in R : (a, b) \leq (\text{card } R)$ unfolding $\text{relpow-fun-conv}$

proof (rule ext1[of $?g$], intro conjI impI allI)

show $?g$ $?n$ $=$ $b$ using $\langle f(Suc \ n) = b \rangle$

next

fix $k$ assume $k$ $<$ $?n$

show $(?g$ $k$, $?g$ $(\text{Suc } k)) \in R$

proof (cases $k$ $<$ $i$)

case True

with $i$ have $k$ $\leq$ $n$ by auto

from steps[OF this] show ?thesis using True by simp

next

case False

hence $i$ $\leq$ $k$ by auto

show ?thesis

proof (cases $k$ $=$ $i$)

case True

thus ?thesis using $ij$ $p_{ij}$ steps[OF $i$] by simp

next

case False

with $i$ $\leq$ $k$ have $i$ $<$ $k$ by auto

hence small: $k + (j - i) <= n$ using $(k$ $<$ $?n$) by arith

show ?thesis using steps[OF small] $(i$ $<$ $k)$ by auto

qed

qed (simp add: $(f \ 0 = a)$)

moreover have $?n$ $\leq$ $n$ using $ij$ $ij$ by arith

ultimately have ?case using $n$ $=$ $\text{card } R$ by blast

}

ultimately show ?case using gr0-implies-Suc[OF $(k > 0)$] by auto

qed

lemma relpow-finite-bounded:

assumes finite $(R :: ('a*'a)\text{set})$

shows $R ^ {\ast} k$ $\subseteq$ $(\text{UN } n: \{n. \ n <= \text{card } R\}. R ^ {\ast} n)$

apply(cases $k$)

apply force

using relpow-finite-bounded1[OF assms, of $k$] by auto

lemma rtrancl-finite-eq-relpow:

finite $R$ $\implies$ $R ^ {\ast\ast}$ $=$ $(\text{UN } n: \{n. \ n <= \text{card } R\}. R ^ {\ast\ast} n)$

by (fastforce simp: rtrancl-power dest: relpow-finite-bounded)
lemma trancl-finite-eq-relpow:
finite R \implies R^+ = (\bigcup n : \{n. 0 < n \& n \leq \text{card } R\}. R^{\leq n})
apply(auto simp add: trancl-power)
apply(auto dest: relpow-finite-bounded1)
done

lemma finite-relcomp[simp,intro]:
assumes finite R and finite S
shows finite(R O S)
proof
  have R O S = (\bigcup (x,y) : R. \bigcup (%(u,v). if u=y then \{(x,v)\} else \{\}) ^{S})
    by(force simp add: split-def)
  thus ?thesis using assms by(clarsimp)
qed

lemma finite-relpow[simp,intro]:
  assumes finite(R :: ('a * 'a) set) shows n>0 \implies finite(R ^^ n)
apply(induct n)
apply(simp)
apply(case_tac n)
apply(simp-all add: assms)
done

lemma single-valued-relpow:
  fixes R :: ('a * 'a) set
  shows single-valued R \implies single-valued (R ^^ n)
apply(induct n arbitrary: R)
apply(simp-all)
apply(rule single-valuedI)
apply(fast dest: single-valuedD elim: relpow-Suc-E)
done

22.5 Bounded transitive closure

definition ntrancl :: nat \Rightarrow ('a * 'a) set \Rightarrow ('a * 'a) set
where
  ntrancl n R = (\bigcup i \in \{i. 0 < i \& i \leq \text{Suc } n\}. R ^^ i)

lemma ntrancl-Zero [simp, code]:
  ntrancl 0 R = R
proof
  show R \subseteq ntrancl 0 R
    unfolding ntrancl-def by fastforce
next
  { fix i have 0 < i \& i \leq \text{Suc } 0 \longleftrightarrow i = 1 by auto }
from this show ntrancl 0 R \leq R
    unfolding ntrancl-def by auto
lemma ntrancl-Suc [simp]:
  \( ntrancl \ (\text{Suc} \ n) \ R = ntrancl \ n \ R \ O \ (\text{Id} \cup R) \)
proof
{ 
  fix \( a \ b \)
  assume \((a, b) \in ntrancl \ (\text{Suc} \ n) \ R \)
  from this obtain \( i \) where \( 0 < i \ i \leq \text{Suc} \ (\text{Suc} \ n) \) \( (a, b) \in R ^ \circ i \)
  unfolding ntrancl-def by auto 
  have \((a, b) \in ntrancl \ n \ R \ O \ (\text{Id} \cup R) \)
  proof (cases \( i = 1 \) )
    case True
    from this \((a, b) \in R ^ \circ i \) show ?thesis
    unfolding ntrancl-def by auto 
    next 
    case False
    from this \(0 < i \) obtain \( j \) where \( i = \text{Suc} \ j \ 0 < j \)
    by (cases \( i \) ) auto 
    from this \((a, b) \in R ^ \circ i \) obtain \( c \) where \( c1: \ (a, c) \in R ^ \circ j \) and \( c2: (c, b) \in R \)
    by auto 
    from \( c1 \ j \ i \leq \text{Suc} \ (\text{Suc} \ n) \) have \((a, c) \in ntrancl \ n \ R \)
    unfolding ntrancl-def by fastforce
    from this \( c2 \) show ?thesis by fastforce 
  qed 
} from this show \( ntrancl \ (\text{Suc} \ n) \ R \subseteq ntrancl \ n \ R \ O \ (\text{Id} \cup R) \)
  by auto
next 
  show \( ntrancl \ n \ R \ O \ (\text{Id} \cup R) \subseteq ntrancl \ (\text{Suc} \ n) \ R \)
  unfolding ntrancl-def by fastforce 
qed

lemma [code]:
  \( ntrancl \ (\text{Suc} \ n) \ r = (\text{let} \ r' = ntrancl \ n \ r \ in \ r' \ Un \ r' \ O \ r) \)
unfolding Let-def by auto

lemma finite-trancl-ntranl:
  finite \( R \Longrightarrow \text{trancl} \ R = ntrancl \ (\text{card} \ R - 1) \ R \)
  by (cases \( \text{card} \ R \) ) (auto simp add: trancl-finite-eq-relpow relpow-empty ntrancl-def)

22.6 Acyclic relations

definition acyclic :: ('a * 'a) set => bool where
  acyclic \( r \equiv (\forall x. (x, x) \sim r ^ \circ) \)
abbreviation acyclicP :: ('a => 'a => bool) => bool where
  acyclicP \( r \equiv \text{acyclic} \ \{(x, y). \ r \ x \ y\} \)
**THEORY “Transitive-Closure”**

```plaintext

lemma acyclic-irrefl [code]:
acyclic r ←→ irrefl (r^+)
by (simp add: acyclic-def irrefl-def)

lemma acyclicI: ALL x. (x, x) ~: r^+ ==> acyclic r
by (simp add: acyclic-def)

lemma (in order) acyclicI-order:
assumes *: \( \forall a b. (a, b) \in r \Rightarrow f b < f a \)
shows acyclic r
proof –
  { fix a b assume (a, b) ∈ r^+
    then have f b < f a
      by induct (auto intro: * less-trans) }
then show ?thesis
  by (auto intro!: acyclicI)
qed

lemma acyclic-insert [iff]:
acyclic(insert (y,x) r) = (acyclic r & (x,y) ~: r^*)
apply (simp add: acyclic-def trancl-insert)
apply (blast intro: rtrancl-trans)
done

lemma acyclic-converse [iff]: acyclic(r^−1) = acyclic r
by (simp add: acyclic-def trancl-converse)

lemmas acyclicP-converse [iff] = acyclic-converse [to-pred]

lemma acyclic-impl-antisym-rtrancl: acyclic r ==> antisym(r^*)
apply (simp add: acyclic-def antisym-def)
apply (blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl)
done

lemma acyclic-subset: [[ acyclic s; r <= s ]] ==> acyclic r
apply (simp add: acyclic-def)
apply (blast intro: trancl-mono)
done

22.7 Setup of transitivity reasoner

ML ⟨⟨
structure Trancl-Tac = Trancl-Tac
{
  val r-into-trancl = @{thm trancl.r-into-trancl};
```
THEORY “Transitive-Closure”

val trancl-trans = @ {thm trancl-trans};
val rtrancl-refl = @ {thm rtrancl.rtrancl-refl};
val r-into-rtrancl = @ {thm r r-into-rtrancl};
val trancl-into-rtrancl = @ {thm trancl-into-rtrancl};
val rtrancl-trancl-trancl = @ {thm rtrancl-trancl-trancl};
val trancl-rtrancl-trancl = @ {thm trancl-rtrancl-trancl};
val rtrancl-trans = @ {thm rtrancl-trans};

fun decomp (@ {const Trueprop} $ t) =
  let fun dec (Const (@ {const-name Set}.member, -$) $ a $ b) $ rel =
    let fun decr (Const (@ {const-name rtrancl}, -) $ r) = (r, r *)
        | decr (Const (@ {const-name trancl}, -) $ r) = (r, r +)
        | decr r = (r, r);
    in SOME (a, b, rel, r) end
    in SOME (a, b, rel, r) end
  | dec - = NONE
  in dec t end
  | decomp - = NONE;

structure Tranclp-Tac = Trancl-Tac
{
  val r-into-trancl = @ {thm tranclp.r-into-trancl};
  val trancl-trans = @ {thm tranclp.trancl-trans};
  val rtrancl-refl = @ {thm rtranclp.rtrancl-refl};
  val r-into-rtrancl = @ {thm r r-into-rtranclp};
  val trancl-into-rtrancl = @ {thm tranclp-into-rtranclp};
  val rtrancl-trancl-trancl = @ {thm rtranclp-tranclp-tranclp};
  val trancl-rtrancl-trancl = @ {thm tranclp-rtranclp-tranclp};
val rtrancl-trans = @ {thm rtranclp-trans};

fun decomp (@ {const Trueprop} $ t) =
  let fun dec (rel $ a $ b) =
    let fun decr (Const (@ {const-name rtranclp}, -) $ r) = (r, r *)
        | decr (Const (@ {const-name tranclp}, -) $ r) = (r, r +)
        | decr r = (r, r);
    in SOME (a, b, rel, r) end
    in SOME (a, b, rel, r) end
  | dec - = NONE
  in dec t end
  | decomp - = NONE;

setup ⟨⟨
  map-theory-simpset (fn ctxt => ctxt
    addSolver (mk-solver Trancl Trancl-Tac.trancl-tac)
    addSolver (mk-solver Rtrancl Trancl-Tac.rtrancl-tac)
)
Optional methods.

**method-setup trancl** =

\[ \langle \langle \text{Scan.succeed (SIMPLE-METHOD'} o \text{Trancl-Tac trancl-tac) } \rangle \rangle \]

**method-setup rtrancl** =

\[ \langle \langle \text{Scan.succeed (SIMPLE-METHOD'} o \text{Tranclp-Tac rtrancl-tac) } \rangle \rangle \]

**method-setup tranclp** =

\[ \langle \langle \text{Scan.succeed (SIMPLE-METHOD'} o \text{Tranclp-Tac trancl-tac) } \rangle \rangle \]

**method-setup rtranclp** =

\[ \langle \langle \text{Scan.succeed (SIMPLE-METHOD'} o \text{Tranclp-Tac rtrancl-tac) } \rangle \rangle \]

end

23 Wellfounded: Well-founded Recursion

theory Wellfounded
imports Transitive-Closure
begin

23.1 Basic Definitions

definition wf :: \((\alpha \times \alpha)\) set \Rightarrow bool where

\[
\text{wf } r \leftarrow (\forall \, P \cdot (\forall x \cdot \forall y \cdot (y, x) : r \Rightarrow P(y) \Rightarrow P(x) \Rightarrow (\forall x \cdot P(x))))
\]

definition wfp :: \(('a => 'a) \Rightarrow bool where

\[
\text{wfp } r \leftarrow \text{wfp } \{ (x, y) \cdot r x y \}
\]

lemma wfp-wf-eq [pred-set-conv]: \text{wfp } (\lambda x y. (x, y) \in r) = \text{wf } r

by (simp add: wfp-def)

lemma wfUNIVI:

\[(\forall \, P \cdot (\forall x \cdot (\forall y \cdot (y, x) \in r : r \Rightarrow P(y)) \Rightarrow P(x) \Rightarrow (\forall x \cdot P(x))))\]

unfolding wf-def by blast

lemmas wfPUNIVI = wfUNIVI [to-pred]

Restriction to domain A and range B. If r is well-founded over their intersection, then \text{wf } r

lemma wfI:

\[ \| r \subseteq A \leftrightarrow B; \]
THEORY “Wellfounded”

\[ \forall x. P. \forall y. (y, x : r \longrightarrow P y) \longrightarrow P x; \ x : A; x : B \ | \Longrightarrow P x | \]

===> \textit{wf} r

unfolding \textit{wf-def} by blast

\textbf{lemma} \textit{wf-induct}:

\[
\begin{align*}
\forall x. [\forall y. (y, x : r \longrightarrow P y) \longrightarrow P x] & \Longrightarrow P(a) \\
\end{align*}
\]

unfolding \textit{wf-def} by blast

\textbf{lemmas} \textit{wfP-induct} = \textit{wf-induct} [to-pred]

\textbf{lemmas} \textit{wf-induct-rule} = \textit{wf-induct} [rule-format, consumes 1, case-names less, induct set: \textit{wf}]

\textbf{lemmas} \textit{wfP-induct-rule} = \textit{wf-induct-rule} [to-pred, induct set: \textit{wfP}]

\textbf{lemma} \textit{wf-not-sym}:

\[\text{assumes} \ \textit{wf} \ r \ (a, x) \in r \ \Rightarrow (x, a) \notin r \ \text{by} \ (\text{drule \textit{wf-not-sym}[OF \ \textit{assms}])} \]

\textbf{lemma} \textit{wf-not-refl} [simp]:

\[\text{assumes} \ \textit{wf} \ r \Rightarrow (a, a) \sim r \ \text{by} \ (\text{blast \ elim: \textit{wf-asym}}) \]

\textbf{lemma} \textit{wf-irrefl}:

\[\text{assumes} \ \textit{wf} \ r \ \text{obtains} \ (a, a) \notin r \ \text{by} \ (\text{drule \textit{wf-not-refl}[OF \ \textit{assms}])} \]

\textbf{lemma} \textit{wf-wellorderI}:

\[\text{assumes} \ \textit{wf} \ \xi \ (x::a::ord, y) \ x < y \ \text{by} \ (\text{drule \textit{lin}[OF \ \textit{assms}])} \]

\textbf{lemma} \textit{(in wellorder) wf}:

\[\text{wf} \ \{\{x, y\}, x < y\} \ \text{unfolding \textit{wf-def} by (blast intro: less-induct)} \]

\textbf{23.2} Basic Results

Point-free characterization of well-foundedness

\textbf{lemma} \textit{wfE-pf}:

\[\text{assumes} \ \textit{wf} \ r \ \text{by} \ (\text{drule \textit{less-induct}[OF \ \textit{assms}])} \]

\textbf{assumes} \[\textit{a} : A \subseteq A \ \text{shows} \ A = \{\} \]

\textbf{23.2} Basic Results

Point-free characterization of well-foundedness

\textbf{lemma} \textit{wfE-pf}:

\[\text{assumes} \ \textit{wf} \ r \ \text{assumes} \ \textit{a} : A \subseteq A \ \text{shows} \ A = \{\} \]

\textbf{23.2} Basic Results

Point-free characterization of well-foundedness

\textbf{lemma} \textit{wfE-pf}:

\[\text{assumes} \ \textit{wf} \ r \ \text{assumes} \ \textit{a} : A \subseteq A \ \text{shows} \ A = \{\} \]

\textbf{23.2} Basic Results

Point-free characterization of well-foundedness

\textbf{lemma} \textit{wfE-pf}:

\[\text{assumes} \ \textit{wf} \ r \ \text{assumes} \ \textit{a} : A \subseteq A \ \text{shows} \ A = \{\} \]
proof
{ fix x
  from wf have x /∈ A
proof induct
  fix x assume \( \text{\( y, (y, x) \in R \Rightarrow y \not\in A \)} \)
  then have x /∈ R " " A by blast
  with a show x /∈ A by blast
  qed
} thus ?thesis by auto
qed

lemma wfI-pf:
  assumes a: \( \bigwedge A. A \subseteq R \implies A = \{\} \)
  shows wf R
proof (rule wfUNIVI)
  fix P :: 'a ⇒ bool and x
  let ?A = \{x. \neg P x\}
  assume \( \forall x. (\forall y. (y, x) \in R \implies P y) \implies P x \)
  then have ?A \subseteq R " " ?A by blast
  with a show P x by blast
qed

Minimal-element characterization of well-foundedness

lemma wfE-min:
  assumes wf: wf R and Q: x \in Q
  obtains z where z \in Q \bigwedge y. (y, z) \in R \implies y \not\in Q
  using Q wfE-pf[OF wf, of Q] by blast

lemma wfI-min:
  assumes a: \( \bigwedge x Q. x \in Q \implies \exists z \in Q. \forall y. (y, z) \in R \implies y \not\in Q \)
  shows wf R
proof (rule wfI-pf)
  fix A assume b: A \subseteq R " " A
  { fix x assume x \in A
    from a[OF this] b have False by blast
  }
  thus A = {} by blast
qed

lemma wf-eq-minimal: wf r = (\( \forall Q x. x \in Q \implies (\exists z \in Q. \forall y. (y, z) \in r \implies y \not\in Q) \))
apply auto
apply (erule wfE-min, assumption, blast)
apply (rule wfI-min, auto)
done

lemmas wfP-eq-minimal = wf-eq-minimal [to-pred]

Well-foundedness of transitive closure
lemma wf-trancl:
  assumes wf r
  shows wf (r⁺⁺)
proof -
  { fix P and x
    assume induct-step: !!x. (!!y. (y, x) : r⁺⁺ ==> P y) ==> P x
    have P x
    proof (rule induct-step)
      fix y assume (y, x) : r⁺⁺
      with (wf r) show P y
      proof (induct x arbitrary: y)
        case (less x)
        note hyp = (∀x′ y′. (x′, x) : r ==> (y′, x′) : r⁺⁺ ==> P y′)
        from (y, x) : r⁺⁺ show P y
        proof cases
          case base
          show P y
          proof (rule induct-step)
            fix y' assume (y', y) : r⁺⁺
            with (y, x) : r show P y' by (rule hyp [of y y'])
          qed
          next
          case step
          then obtain x' where (x', x) : r and (y, x') : r⁺⁺ by simp
          then show P y by (rule hyp [of x' y])
        qed
      qed
    qed
  } then show thesis unfolding wf-def by blast
qed

lemmas wfP-trancl = wf-trancl [to-pred]

lemma wf-converse-trancl: wf (r⁻⁻) ==> wf ((r⁺⁺)⁻⁻)
apply (subst trancl-converse [symmetric])
apply (erule wf-trancl)
done

Well-foundedness of subsets

lemma wf-subset: [| wf(r); p<=r |] ==> wf(p)
apply (simp [no-asn-use] add: wf-eq-minimal)
apply fast
done

lemmas wfP-subset = wf-subset [to-pred]

Well-foundedness of the empty relation

lemma wf-empty [iff]: wf {}
by \((simp\ add:\ wf-def)\)

**lemma** \(_{\text{wfP-empty \ [iff]}}\):

\(\text{wfP} \ (\lambda x \, y. \ \text{False})\)

**proof** –

- **have** \(\text{wfP bot} \ \text{by} \ (\text{fact \, wf-empty \ [to-pred \, bot-empty-\text{eq2}])}\)
- **then show** \(?thesis \ \text{by} \ (simp\ add:\ bot-fun-def)\)

qed

**lemma** \(_{\text{wf-Int1}}\): \(\text{wf \, r} \Longrightarrow \text{wf \, (r \, \text{Int} \, r')}\)

apply \((\text{erule \, wf-subset})\)
apply \((\text{rule \, Int-lower1})\)
done

**lemma** \(_{\text{wf-Int2}}\): \(\text{wf \, r} \Longrightarrow \text{wf \, (r' \, \text{Int} \, r)}\)

apply \((\text{erule \, wf-subset})\)
apply \((\text{rule \, Int-lower2})\)
done

Exponentiation

**lemma** \(_{\text{wf-exp}}\):

- **assumes** \(\text{wf} \ (R \ ^n)\)
- **shows** \(\text{wf} \ R\)

**proof** \((\text{rule \, wfI-pf})\)

- **fix** \(A\)
- **assume** \(A \subseteq R ^ n\)
- **then have** \(A \subseteq (R ^ n) ^ n\)
- **by** \((\text{induct \, n})\)
- **with** \(\text{wf} \ (R ^ n)\)
- **show** \(A = \{\}\)

qed

Well-foundedness of insert

**lemma** \(_{\text{wf-insert \ [iff]}}\): \(\text{wf(insert \ (y,x) \ r) = (wf(r) \ & \ (x,y) \ \\sim) \ r^*})\)

apply \((\text{rule \, iffI})\)
apply \((\text{blast \, elim: \, wf-trancl \ [THEN \, wf-irrefl]}\)
intro: \text{trancI-into-trancl1 \, wf-subset}
\text{rtrancI-\text{mono} \ [THEN \ [2] \, rev-subsetD])}\)
apply \((\text{simp \, add: \, wf-eq-minimal, \, safe})\)
apply \((\text{rule \, allE, \, assumption, \, erule \, impE, \, blast})\)
apply \((\text{erule \, bexE})\)
apply \((\text{rename-tac \ a, \, case-tac \ a = x})\)
prefer 2
apply blast
apply \((\text{case-tac \ y:Q})\)
prefer 2 apply blast
apply \((\text{rule-tac \ x = \{z. \, z:Q \ & \ (z,y) : r^*\ \} \ in \ allE})\)
apply assumption
apply \((\text{rule-tac \ V = ALL \, Q. \ (EX \, x. \, x : Q) \ --\ \ ?P \ Q \ in \ thin-rl})\)
— essential for speed

Blast with new substOccur fails
apply (fast intro: converse-rtrancl-into-rtrancl)
done

Well-foundedness of image

lemma wf-map-prod-image: \[ \forall \text{wf } r; \text{inj } f \quad \Rightarrow \quad \text{wf } (\text{map-prod } f f' r) \]
apply (simp only: wf-eq-minimal, clarify)
apply (case-tac EX p. f p : Q)
apply (erule-tac x = {p. f p : Q} in allE)
apply (fast dest: inj-onD, blast)
done

23.3 Well-Foundedness Results for Unions

lemma wf-union-compatible:
  assumes \( \text{wf } R \) \( \text{wf } S \)
  assumes \( R \circ S \subseteq R \)
  shows \( \text{wf } (R \cup S) \)
proof (rule wfI-min)
  fix \( x : 'a \) and \( Q \)
  let \( ?Q' = \{ x \in Q. \forall y. (y, x) \in R \rightarrow y \notin Q \} \)
  assume \( x \in Q \)
  obtain \( a \) where \( a \in ?Q' \)
    by (rule wfE-min \[ OF \langle \text{wf } R \rangle \langle x \in Q \rangle \] blast)
  with \( \langle \text{wf } S \rangle \)
  obtain \( z \) where \( z \in ?Q' \) and \( z\text{min}: \ \land y. (y, z) \in S \rightarrow y \notin ?Q' \) by (erule
  \text{wfE-min})
  \{
  fix \( y \) assume \( (y, z) \in S \)
  then have \( y \notin ?Q' \) by (rule zmin)
  have \( y \notin Q \)
  proof
    assume \( y \in Q \)
    with \( \langle y \notin ?Q' \rangle \)
    obtain \( w \) where \( (w, y) \in R \) and \( w \in Q \) by auto
    from \( \langle (w, y) \in R \rangle \langle (y, z) \in S \rangle \) have \( (w, z) \in R \circ S \) by (rule relcompI)
    with \( \langle R \circ S \subseteq R \rangle \) have \( (w, z) \in R \).
    with \( \langle z \in ?Q' \rangle \) have \( w \notin Q \) by blast
    with \( \langle w \in Q \rangle \) show False by contradiction
  qed
  \}
  with \( \langle z \in ?Q' \rangle \) show \( \exists z \in Q. \forall y. (y, z) \in R \cup S \rightarrow y \notin Q \) by blast
qed

Well-foundedness of indexed union with disjoint domains and ranges

lemma wf-UN: \[ \forall i. \text{wf}(r i); \]
  \[ \forall i. \text{All } j. i \sim j \quad \Rightarrow \quad \text{Domain}(r i) \text{ Int Range}(r j) = \{ \} \]
  \[ \Rightarrow \quad \text{wf}(\text{UN } i. r i) \]
apply (simp only: wf-eq-minimal, clarify)
apply (rename-tac A a, \(\text{case-tac EX i: EX a:A. EX b:A. (b,a) : r i}\))
prefer 2
apply force
apply clarify
apply (drule bspec, assumption)
apply (erule-tac \(\{a. a:A \& (EX b:A. (b,a) : r i)\} \text{ in allE}\))
apply (blast elim:: allE)
done

lemma \(\text{wfP-SUP}:\) 
\(\forall i. \text{wfP} (r i) \Rightarrow \forall i j. r i \neq r j \rightarrow \inf (\text{DomainP} (r i)) (\text{RangeP} (r j)) = \bot\) 
\Rightarrow \text{wfP} (\text{SUPREMUM UNIV r})
apply (rule wf-UN[to-pred])
apply simp-all
done

lemma \(\text{wf-Union}:\) 
\[\forall R. \text{wf r}; \forall R. \forall s. \text{r} \sim s \rightarrow \text{Domain r Int Range s} = \{\}\] 
\[\Rightarrow \text{wf} (\text{Union R})\]
using wf-UN[of R \(\lambda i. i\)] by simp

lemma \(\text{wf-Un}:\) 
\[\forall r; \text{wf s}; \text{Domain r Int Range s} = \{\}\ \Rightarrow \text{wf} (r \text{ Un s})\]
using wf-union-compatible[of s r]
by (auto simp: Un-ac)

lemma \(\text{wf-union-merge}:\) 
\(\text{wf} (R \cup S) = \text{wf} (R O R \cup S O R \cup S)\) (is \(\text{wf ?A = wf ?B}\))
proof
assume \(\text{wf ?A}\)
with \(\text{wf-trancl} \) have \(\text{wfT: wf (\text{?A}^+)}\).
moreover have \(\text{?B} \subseteq \text{?A}^+\)
  by (subst trancl-unfold, subst trancl-unfold) blast
ultimately show \(\text{wf ?B}\) by (rule wf-subset)
next
assume \(\text{wf ?B}\)
show \(\text{wf ?A}\)
proof (rule wfI-min)
fix \(Q :: 'a \text{ set and x}\)
assume \(x \in Q\)

with \(\text{wf ?B}\)
obtain \(z\ where z \in Q\ and \(\text{\(\land\ y. (y, z) \in ?B \Rightarrow y \notin Q\)}\)
  by (erule \(\text{wfE-min}\))
then have \(A1: \(\land\ y. (y, z) \in R O R \Rightarrow y \notin Q\)\)
  and \(A2: \(\land\ y. (y, z) \in S O R \Rightarrow y \notin Q\)\)
and A3: \( \forall y. (y, z) \in S \implies y \notin Q \)

by auto

show \( \exists z \in Q. \forall y. (y, z) \in ?A \implies y \notin Q \)

proof (cases \( \forall y. (y, z) \in R \implies y \notin Q \))

case True

with \( (z \in Q) \) A3 show ?thesis by blast

next

case False

then obtain \( z' \) where \( z' \in Q \) \( (z', z) \in R \)

by blast

have \( \forall y. (y, z') \in ?A \implies y \notin Q \)

proof (intro allI impI)

fix \( y \) assume \( (y, z') \in ?A \)

then show \( y \notin Q \)

proof

assume \( (y, z') \in R \)

then have \( (y, z) \in R \cup R \) using \( (z', z) \in R \).

with A1 show \( y \notin Q \).

next

assume \( (y, z') \in S \)

then have \( (y, z) \in S \cup R \) using \( (z', z) \in R \).

with A2 show \( y \notin Q \).

qed

with \( (z' \in Q) \) show ?thesis ..

qed

qed

lemma wf-comp-self: \( \text{wf } R = \text{wf } (R \cup R) \) — special case

by (rule wf-union-merge [where \( S = \emptyset \), simplified])

23.4 Acyclic relations

lemma wf-acyclic: \( \text{wf } r \implies \text{acyclic } r \)

apply (simp add: acyclic-def)

apply (blast elim: wf-trancl [THEN wf-irrefl])

done

lemmas wfP-acyclicP = wf-acyclic [to-pred]

Wellfoundedness of finite acyclic relations

lemma finite-acyclic-wf [rule-format]: \( \text{finite } r \implies \text{acyclic } r \implies \text{wf } r \)

apply (erule finite-induct, blast)

apply (simp (no-asmsimp) only: split-tupled-all)

apply simp

done
THEORY “Wellfounded”

lemma finite-acyclic-wf-converse: \[\text{finite } r \implies \text{acyclic } r\]
apply (erule finite-converse [THEN iffD2, THEN finite-acyclic-wf])
apply (erule acyclic-converse [THEN iffD2])
done

lemma wf-iff-acyclic-if-finite: finite r ==> \text{acyclic } r
by (blast intro: finite-acyclic-wf wf-acyclic)

23.5 \text{n} \text{at} \text{is} \text{well-founded}

lemma less-nat-rel: \(\text{op} < = (\lambda m \ n. \ n = \text{Suc } m)^{++}\)
proof (rule ext, rule ext, rule iffI)
  fix n m :: nat
  assume m < n
  then show \((\lambda m \ n. \ n = \text{Suc } m)^{++} \ m \ n\)
  proof (induct n)
    case 0 then show \(?case\) by auto
  next
    case (Suc n) then show \(?case\)
    by (auto simp add: less-Suc-eq-le reflexive le-less)
  qed
next
  fix n m :: nat
  assume \((\lambda m \ n. \ n = \text{Suc } m)^{++} \ m \ n\)
  then show m < n
  by (induct n)
  (simp-all add: less-Suc-eq-le reflexive le-less)
qed

definition pred-nat :: \((\text{nat} \times \text{nat}) \text{ set where}\)
pred-nat = \{(m, n). \ n = \text{Suc } m\}

definition less-than :: \((\text{nat} \times \text{nat}) \text{ set where}\)
less-than = pred-nat ^+

lemma less-eq: \((m, n) \in \text{pred-nat}^{++} \iff m < n\)
unfolding less-nat-rel pred-nat-def trancl-def by simp

lemma pred-nat-trancl-eq-le:
  \((m, n) \in \text{pred-nat}^{*} \iff m \leq n\)
unfolding less-eq rtrancl-eq-or-trancl by auto

lemma wf-pred-nat: \text{wf pred-nat}
apply (unfold wf-def pred-nat-def, clarify)
apply (induct-tac x, blast+)
done
lemma wf-less-than \(\text{iff}\): \(\text{wf less-than}\)
by (simp add: less-than-def wf-pred-nat \(\text{THEN} \text{wf-trancl}\))

lemma trans-less-than \(\text{iff}\): \(\text{trans less-than}\)
by (simp add: less-than-def)

lemma less-than-iff \(\text{iff}\):
\[(x, y) : \text{less-than} \equiv (x < y)\]
by (simp add: less-than-def less-eq)

lemma wf-less: \(\text{wf} (x, y :: \text{nat} \Rightarrow x < y)\)
using wf-less-than by (simp add: less-than-def less-eq [symmetric])

23.6 Accessible Part

Inductive definition of the accessible part \(\text{acc} r\) of a relation; see also [?].

inductive-set
\(\text{acc} :: (\text{'}a \ast \text{'}a) \set \Rightarrow \text{'}a \set\)
for \(r :: (\text{'}a \ast \text{'}a) \set\)
where
\(\text{accI}: (!!(y, (y, x) : r \Rightarrow y : \text{acc} r) \Rightarrow x : \text{acc} r)\)

abbreviation
\(\text{termip} :: (\text{'}a \Rightarrow \text{'}a \Rightarrow \text{bool}) \Rightarrow \text{'}a \Rightarrow \text{bool}\)
where
\(\text{termip} r \equiv \text{accp} (r^{-1})\)

abbreviation
\(\text{termi} :: (\text{'}a \ast \text{'}a) \set \Rightarrow \text{'}a \set\)
where
\(\text{termi} r \equiv \text{acc} (r^{-1})\)

lemmas accpI = accp.accI

lemma accp-eq-acc [code]:
\(\text{accp} r = (\lambda x. x \in \text{Wellfounded.acc} \{(x, y). r x y\})\)
by (simp add: acc-def)

Induction rules
theorem accp-induct:
assumes major: \(\text{accp} r a\)
assumes hyp: \(!!x. \text{accp} r x \Rightarrow \forall y. y x \Rightarrow P y \Rightarrow P x\)
shows \(P a\)
apply (rule major [THEN accp.induct])
apply (rule hyp)
apply (rule accp.accI)
apply fast
apply fast
done

theorems accp-induct-rule = accp-induct [rule-format, induct set: accp]
THEORY “Wellfounded”

theorem accp-downward: accp r b ==> r a b ==> accp r a
  apply (erule accp.cases)
  apply fast
  done

lemma not-accp-down:
  assumes na: ~ accp R x
  obtains z where R z x and ~ accp R z
proof
  assume a: \forall z. [R z x; ~ accp R z] ==> thesis
  show thesis
  proof (cases \forall z. R z x ==> accp R z)
    case True
    hence \forall z. R z x ==> accp R z by auto
    hence accp R x
    by (rule accp.accI)
    with na show thesis ..
  next
    case False then obtain z where R z x and ~ accp R z
      by auto
    with a show thesis .
  qed
  qed

lemma accp-downwards-aux: r** b a ==> accp r a ==> accp r b
  apply (erule rtranclp-induct)
  apply blast
  apply (blast dest: accp-downward)
  done

theorem accp-downwards: accp r a ==> r** b a ==> accp r b
  apply (blast dest: accp-downwards-aux)
  done

theorem accp-wfPI: \forall x. accp r x ==> wfP r
  apply (rule wfPUNIVI)
  apply (rule-tac P=P in accp-induct)
  apply blast
  apply blast
  done

theorem accp-wfPD: wfP r ==> accp r x
  apply (erule wfP-induct-rule)
  apply (rule accp.accI)
  apply blast
  done

theorem wfP-accp-iff: wfP r = (\forall x. accp r x)
apply (blast intro: accp-wfPI dest: accp-wfPD)
done

Smaller relations have bigger accessible parts:

lemma accp-subset:
  assumes sub: $R_1 \leq R_2$
  shows accp $R_2 \leq$ accp $R_1$
proof (rule predicate1I)
  fix $x$
  assume accp $R_2 x$
  then show accp $R_1 x$
  proof (induct $x$)
    fix $x$
    assume ih: $\forall y. R_2 y x \Longrightarrow$ accp $R_1 y$
    with sub show accp $R_1 x$
    by (blast intro: accp.accI)
  qed
  qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma accp-subset-induct:
  assumes subset: $D \leq$ accp $R$
  and del: $\forall x z. [D x; R z x] \Longrightarrow D z$
  and $D x$
  and istep: $\forall x. [D x; (\forall z. R z x \Longrightarrow P z)] \Longrightarrow P x$
  shows $P x$
proof
  from subset and $(D x)$
  have accp $R x$
  then show $P x$ using $(D x)$
  proof (induct $x$)
    fix $x$
    assume $D x$
    and $\forall y. R y x \Longrightarrow D y \Longrightarrow P y$
    with del and istep show $P x$ by blast
  qed
  qed

Set versions of the above theorems

lemmas acc-induct = accp-induct [to-set]
lemmas acc-induct-rule = accp-induct [rule-format, induct set: acc]
lemmas acc-downward = accp-downward [to-set]
lemmas not-acc-down = not-accp-down [to-set]
lemmas acc-downwards-aux = accp-downwards-aux [to-set]
lemmas acc-downwards = accp-downwards [to-set]

lemmas acc-wfI = accp-wfPI [to-set]

lemmas acc-wfD = accp-wfPD [to-set]

lemmas wf-acc-iff = wfP-accp-iff [to-set]

lemmas acc-subset = accp-subset [to-set]

lemmas acc-subset-induct = accp-subset-induct [to-set]

23.7 Tools for building wellfounded relations

Inverse Image

lemma wf-inv-image [simp.intro]: wf(r) ===> wf(inv-image r (f::'a=>>'b))
apply (simp (no-asm-use) add: inv-image-def wf-eq-minimal)
apply clarify
apply (subgoal_tac EX (w::'b) . w : {w. EX (x::'a) . x: Q & (f x = w) })
prefer 2 apply (blast del: allE)
apply (erule allE)
apply (erule (1) notE impE)
apply blast
done

Measure functions into nat

definition measure :: ('a => nat) => ('a * 'a)set
where measure = inv-image less-than

lemma in-measure[simp, code-unfold]: ((x,y) : measure f) = (f x < f y)
  by (simp add:measure-def)

lemma wf-measure [iff]: wf (measure f)
apply (unfold measure-def)
apply (rule wf-less-than [THEN wf-inv-image])
done

lemma wf-if-measure: fixes f :: 'a => nat
shows (!x. P x ==> f(g x) < f x) ==> wf {(y,x). P x & y = g x}
apply(insert wf-measure[of f])
apply(simp only: measure-def inv-image-def less-than-def less-eq)
apply(erule wf-subset)
apply auto
done

Lexicographic combinations

definition lex-prod :: ('a * 'a) set => ('b * 'b) set => (('a * 'b) x ('a * 'b)) set
(infixr "<*lex*>" 80) where
ra \lex* \subset rb = \{((a, b), (a', b')). (a, a') \in ra \lor a = a' \land (b, b') \in rb\}

**lemma** wf-lex-prod [intro!]: [\[ \text{wf(ra); wf(rb) } \] =\=\=> \text{wf(ra \lex* \subset rb)}

**apply** (unfold \text{wf-def lex-prod-def})

**apply** (rule allI, rule \text{impI})

**apply** (simp (no-asm-use) only: split-paired-All)

**apply** (drule spec, erule mp)

**apply** (rule allI, rule \text{impI})

**apply** (drule spec, erule mp, blast)

**done**

**lemma** \text{in-lex-prod[simp]}:

((((a,b),(a',b')): r \lex* \subset s) = (((a,a'): r \lor (a = a' \land (b, b'): s))

**by** (auto simp: \text{lex-prod-def})

\text{op \lex* \subset preserves transitivity}

**lemma** trans-lex-prod [intro!]:

[\[ \text{trans R1; trans R2 } \] =\=\=> \text{trans (R1 \lex* \subset R2)}

**by** (unfold \text{trans-def lex-prod-def}, blast)

lexicographic combinations with measure functions

**definition** mlex-prod :: ('a \Rightarrow \text{nat}) \Rightarrow ('a \times 'a) set \Rightarrow ('a \times 'a) set (\text{infixr \lex* mlex 80})

where

f \lex* mlex R = inv-image (less-than \lex* \subset R) (%x. (f x, x))

**lemma** wf-mlex: \text{wf R =\=\=> wf (f \lex* mlex R)}

**unfolding** mlex-prod-def

**by** auto

**lemma** mlex-less: f x < f y \Longrightarrow (x, y) \in f \lex* mlex R

**unfolding** mlex-prod-def **by** simp

**lemma** mlex-leq: f x \leq f y \Longrightarrow (x, y) \in R \Longrightarrow (x, y) \in f \lex* mlex R

**unfolding** mlex-prod-def **by** auto

proper subset relation on finite sets

**definition** finite-psubset :: ('a set \times 'a set) set

where finite-psubset = \{(A,B). A < B \& finite B\}

**lemma** wf-finite-psubset[simp]: \text{wf (finite-psubset)}

**apply** (unfold finite-psubset-def)

**apply** (rule \text{wf-measure [THEN wf-subset]})

**apply** (simp add: measure-def inv-image-def less-than-def less-eq)

**apply** (fast elim!: psubset-card-mono)

**done**

**lemma** trans-finite-psubset: trans finite-psubset

**by** (simp add: finite-psubset-def less-le trans-def, blast)
THEORY “Wellfounded”

lemma in-finite-psubset[simp]: \((A, B) \in \text{finite-psubset} = (A < B \& \text{finite } B)\)

unfolding finite-psubset-def by auto

max- and min-extension of order to finite sets

inductive-set max-ext :: \((\times \times )\) set \Rightarrow \((\times \times )\) set

for R :: \(\times \times \) set

where

max-extI[intro]: finite X \Rightarrow finite Y \Rightarrow Y \neq \{} \Rightarrow (\\& x. x \in X \Rightarrow \exists y Y. (x, y) \in R) \Rightarrow (X, Y) \in \text{max-ext } R

lemma max-ext-wf:

assumes \(\text{wf: wf } r\)

shows \(\text{wf: (max-ext r)}\)

proof (rule acc-wfI, intro allI)

fix M show M \in acc (max-ext r) (is - \in ?W)

proof cases

assume finite M

thus ?thesis

proof (induct M)

show {} \in ?W

by (rule accI) (auto elim: max-ext.cases)

next

fix M a assume M \in ?W finite M

with uf show insert a M \in ?W

proof (induct arbitrary: M)

fix M a

assume M \in ?W and [intro]: finite M

assume hyp: \(\\_ b. (b, a) \in r \Rightarrow M \in ?W \Rightarrow \text{finite } M \Rightarrow \text{insert } b M \in ?W\)

\{\}

fix N M :: \(\times\) set

assume finite N finite M

then

have \[[M \in ?W ; (\\& y. y \in N \Rightarrow (y, a) \in r)] \Rightarrow N \cup M \in ?W\]

by (induct N arbitrary: M) (auto simp: hyp)

note add-less = this

show insert a M \in ?W

proof (rule accI)

fix N assume Nless: \((N, \text{insert } a M) \in \text{max-ext } r\)

hence asm1: \(\\& x. x \in N \Rightarrow (x, a) \in r \vee (\exists y M. (x, y) \in r)\)

by (auto elim!: max-ext.cases)

let \(?N1 = \{ n \in N. (n, a) \in r \}\)

let \(?N2 = \{ n \in N. (n, a) \notin r \}\)

have N: \(?N1 \cup \?N2 = N\) by (rule set-eqI) auto

from Nless have finite N by (auto elim: max-ext.cases)
then have \( \text{finites: finite ?N1 finite ?N2} \) by auto

have \( ?N2 \in ?W \)
proof cases
assume [simp]: \( M = {} \)
have \( Mw: {} \in ?W \) by (rule accI) (auto elim: max-ext.cases)

from \( \text{asm1} \) have \( ?N2 = {} \) by auto
with \( Mw \) show \( ?N2 \in ?W \) by (simp only:)
next
assume \( M \neq {} \)
from \( \text{asm1 finites have N2: (} ?N2, M \text{) \in max-ext r} \)
by (rule-tac max-extI [OF - - \( M \neq {} \)]) auto

with \( \langle M \in ?W \rangle \) show \( ?N2 \in ?W \) by (rule acc-downward)
qed
with \( \text{finites have } ?N1 \cup ?N2 \in ?W \)
by (rule add-less) simp
then show \( N \in ?W \) by (simp only: \( N \))
qed
qed
next
assume [simp]: \( \neg \text{ finite } M \)
show \( \text{thesis} \)
by (rule accI) (auto elim: max-ext.cases)
qed
qed

lemma \( \text{max-ext-additive}: \)
\( (A, B) \in \text{max-ext } R \implies (C, D) \in \text{max-ext } R \implies \)
\( (A \cup C, B \cup D) \in \text{max-ext } R \)
by (force elim!: max-ext.cases)


definition \( \text{min-ext :: (} 'a \times 'a \text{) set } \implies (\text{'}a\text{ set } \times \text{'}a\text{ set}) \text{ set where} \)
\( \text{min-ext } r = \{ (X, Y) \mid X \neq {} \and (\forall y \in Y. (\exists x \in X. (x, y) \in r)) \} \)

lemma \( \text{min-ext-wf}: \)
assumes \( \text{wf } r \)
shows \( \text{wf } (\text{min-ext } r) \)
proof (rule wfI-min)
fix \( Q :: \text{'}a\text{ set set} \)
fix \( x \)
assume nonempty: \( x \in Q \)
show \( \exists m \in Q. (\forall n. (n, m) \in \text{min-ext } r \implies n \notin Q) \)
proof cases
assume \( Q = \{ \} \) thus \( \text{thesis} \) by (simp add: min-ext-def)
next
assume $Q \neq \{\emptyset\}$
with nonempty
obtain $e \cdot x$ where $x \in Q \quad e \in x$ by force
then have $e \cdot U : e \in \bigcup Q$ by auto
with $(\text{wf } r)$
obtain $z$ where $z : z \in \bigcup Q \land y. \quad (y, z) \in r \implies y \notin \bigcup Q$
by (erule $\text{wfE-min}$)
from $z$ obtain $m$ where $m \in Q \quad z \in m$ by auto
from $m \in Q$
show ?thesis
proof (rule, intro bexI allI impI)
fix $n$
assume smaller: $(n, m) \in \text{min-ext } r$
with $(z \in m)$ obtain $y$ where $y : y \in n \quad (y, z) \in r$ by (auto simp: \text{min-ext-def})
then show $n \notin Q$ using $z(2)$ by auto
qed
qed

Bounded increase must terminate:

lemma $\text{wf-bounded-measure}$:
fixes $ub :: 'a \Rightarrow \text{nat}$ and $f :: 'a \Rightarrow \text{nat}$
assumes $!!a. (b, a) : r \implies ub b \leq ub a \land ub a \geq f b \land f b > f a$
shows $\text{wf } r$
apply (rule $\text{wf-subset}$[OF $\text{wf-measure}$[of $\%a. \quad ub a - f a$]])
apply (auto dest: assms)
done

lemma $\text{wf-bounded-set}$:
fixes $ub :: 'a \Rightarrow 'b \text{ set}$ and $f :: 'a \Rightarrow 'b \text{ set}$
assumes $!!a. (b, a) : r \implies \quad \text{finite}(ub a) \land ub b \subseteq ub a \land ub a \supseteq f b \land f b \supset f a$
shows $\text{wf } r$
apply (rule $\text{wf-bounded-measure}$[of $r$ $\%a. \quad \text{card}(ub a)$ $\%a. \quad \text{card}(f a)$])
apply (erule assms)
apply (blast intro: \text{card-mono finite-subset psubset-card-mono dest: psubset-eq}[THEN iffD2])
done

hide-const (open) acc accp

end

24 Wfrec: Well-Founded Recursion Combinator

theory Wfrec
imports Wellfounded
begin
inductive
wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool
for R :: ('a * 'a) set
and F :: ('a => 'b) => 'a => 'b
where
wfrecI: ALL z. (z, x) : R --> wfrec-rel R F z (g z) =>
wfrec-rel R F x (F g x)

definition
cut :: ('a => 'b) => ('a * 'a) set => 'a => 'b
where
cut f r x == (% y. if (y, x):r then f y else undefined)
definition
adm-wf :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => bool
where
adm-wf R F == ALL f g x. (ALL z. (z, x) : R --> f z = g z) --> F f x = F g x
definition
wfrec :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b
where
wfrec R F == %x. THE y. wfrec-rel R (%f x. F (cut f R x) x) x y

lemma cuts-eq: (cut f r x = cut g r x) = (ALL y. (y, x):r --> f(y)=g(y))
by (simp add: fun-eq-iff cut-def)

lemma cut-apply: (x,a):r ==>((cut f r a)(x) = f(x))
by (simp add: cut-def)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma wfrec-unique: [adm-wf R F; wf R] =>> EX! y. wfrec-rel R F x y
apply (simp add: adm-wf-def)
apply (erule-tac a=x in wf-induct)
apply (rule exI)
apply (rule-tac g = %x. THE y. wfrec-rel R F x y in wfrec-rel.wfrecI)
apply (fast dest!: theI)
apply (erule wfrec-rel.cases, simp)
apply (erule allE, erule allE, erule allE, erule mp)
apply (blast intro: the-equality [symmetric])
done

lemma adm-lemma: adm-wf R (%f x. F (cut f R x) x)
apply (simp add: adm-wf-def)
apply (intro strip)
apply (rule cuts-eq [THEN iffD2, THEN subst], assumption)
apply (rule refl)
done

lemma wfrec: wf(r) =>> wfrec r H a = H (cut (wfrec r H) r a) a
apply (simp add: wfrec-def)
apply (rule adm-lemma [THEN wfrec-unique, THEN the1-equality], assumption)
apply (rule wfrec-rel wfrecI)
apply (intro strip)
apply (erule adm-lemma [THEN wfrec-unique, THEN the1'])
done

* This form avoids giant explosions in proofs. NOTE USE OF ==

lemma def-wfrec: \[ | f == wfrec r H; \ q f (r) | == f(a) = H (cut f r a) a \]
apply auto
apply (blast intro: wfrec)
done

24.1 Wellfoundedness of same-fst

definition
same-fst :: \((a => bool) => \('a => ('b * 'b) set) => (('as*'b)*('as*'b)) set\)
where
  same-fst P R == \{((x',y'),(x,y)) . x'=x & P x & (y',y) : R x\}
— For rec-def declarations where the first n parameters stay unchanged in the recursive call.

lemma same-fstI [intro!]:
  [| P x; (y',y) : R x |] ==\> ((x,y'),(x,y)) : same-fst P R
by (simp add: same-fst-def)

lemma wf-same-fst:
  assumes prem: (!!x. P x ==> wf(R x))
  shows wf(same-fst P R)
apply (simp cong del: imp-cong add: wf-def same-fst-def)
apply (intro strip)
apply (rename-tac a b)
apply (case-tac wf (R a))
apply (erule-tac a = b in wf-induct, blast)
apply (blast intro: prem)
done

end

25 Order-Relation: Orders as Relations

theory Order-Relation
imports Wfrec
begin

25.1 Orders on a set

definition preorder-on A r \equiv refl-on A r \land trans r
**THEORY “Order-Relation”**

**definition** partial-order-on $A \ r \equiv preorder-on \ A \ r \land antisym \ r$

**definition** linear-order-on $A \ r \equiv partial-order-on \ A \ r \land total-on \ A \ r$

**definition** strict-linear-order-on $A \ r \equiv trans \ r \land irrefl \ r \land total-on \ A \ r$

**definition** well-order-on $A \ r \equiv linear-order-on \ A \ r \land wf(r - Id)$

**lemmas** order-on-defs =

preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

**lemma** preorder-on-empty[simp]: preorder-on {} {} by (simp add:preorder-on-def trans-def)

**lemma** partial-order-on-empty[simp]: partial-order-on {} {} by (simp add:partial-order-on-def)

**lemma** linear-order-on-empty[simp]: linear-order-on {} {} by (simp add:linear-order-on-def)

**lemma** well-order-on-empty[simp]: well-order-on {} {} by (simp add:well-order-on-def)

**lemma** preorder-on-converse[simp]: preorder-on $A (r^{-}1) = preorder-on \ A \ r$
by (simp add:preorder-on-def)

**lemma** partial-order-on-converse[simp]:
partial-order-on $A (r^{-}1) = partial-order-on \ A \ r$
by (simp add: partial-order-on-def)

**lemma** linear-order-on-converse[simp]:
linear-order-on $A (r^{-}1) = linear-order-on \ A \ r$
by (simp add: linear-order-on-def)

**lemma** strict-linear-order-on-diff-Id:
linear-order-on $A \ r \Rightarrow strict-linear-order-on \ A \ (r - Id)$
by (simp add: order-on-defs trans-diff-Id)

### 25.2 Orders on the field

**abbreviation** Refl $r \equiv refl-on \ (Field \ r) \ r$

**abbreviation** Preorder $r \equiv preorder-on \ (Field \ r) \ r$

**abbreviation** Partial-order $r \equiv partial-order-on \ (Field \ r) \ r$
THEORY “Order-Relation”

abbreviation Total r ≡ total-on (Field r) r

abbreviation Linear-order r ≡ linear-order-on (Field r) r

abbreviation Well-order r ≡ well-order-on (Field r) r

lemma subset-Image-Image-iff:
  [ Preorder r; A ⊆ Field r; B ⊆ Field r ] ⇒
  r " A ⊆ r " B ⇔ (∀ a∈A.∃ b∈B. (b,a):r)
unfolding preorder-on-def refl-on-def Image-def
apply (simp add: subset-eq)
unfolding trans-def by fast

lemma subset-Image1-Image1-iff:
  [ Preorder r; a : Field r; b : Field r ] ⇒ r " {a} ⊆ r " {b} ⇔ (b,a):r
by(simp add:subset-Image-Image-iff)

lemma Refl-antisym-eq-Image1-Image1-iff:
  assumes r: Refl r and as: antisym r and abf: a ∈ Field r b ∈ Field r
  shows r " {a} = r " {b} ⇔ a = b
proof
  assume r " {a} = r " {b}
  hence c: ∃ x. (a, x) ∈ r ⇔ (b, x) ∈ r by (simp add: set-eq-iff)
  have (a, a) ∈ r (b, b) ∈ r using r abf by (simp-all add: refl-on-def)
  hence (a, b) ∈ r (b, a) ∈ r using e[of a] e[of b] by simp-all
  thus a = b using as[unfolded antisym-def] by blast
qed fast

lemma Partial-order-eq-Image1-Image1-iff:
  [Partial-order r; a: Field r; b: Field r ] ⇒ r " {a} = r " {b} ⇔ a = b
by(auto simp:order-on-defs Refl-antisym-eq-Image1-Image1-iff)

lemma Total-Id-Field:
  assumes TOT: Total r and NID: ∼ (r <= Id)
  shows Field r = Field(r − Id)
using mono-Field[of r − Id] Diff-subset[of r Id]
proof(auto)
  have r ≠ {} using NID by fast
  then obtain b and c where b ≠ c ∧ (b,c) ∈ r using NID by auto
  hence 1: b ≠ c ∧ {b,c} ≤ Field r by (auto simp: Field-def)
  fix a assume *: a ∈ Field r
  obtain d where 2: d ∈ Field r and 3: d ≠ a
  using * 1 by auto
  hence (a,d) ∈ r ∨ (d,a) ∈ r using * TOT
  by (simp add: total-on-def)
  thus a ∈ Field(r − Id) using 3 unfolding Field-def by blast
25.3 Orders on a type

abbreviation strict-linear-order ≡ strict-linear-order-on UNIV

abbreviation linear-order ≡ linear-order-on UNIV

abbreviation well-order ≡ well-order-on UNIV

25.4 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

25.4.1 Auxiliaries

lemma refl-on-domain:

\[ \text{refl-on } A \ r \ (a,b) \in r \implies a \in A \land b \in A \]

by (auto simp add: refl-on-def)

corollary well-order-on-domain:

\[ \text{well-order-on } A \ r \ (a,b) \in r \implies a \in A \land b \in A \]

by (auto simp add: refl-on-domain order-on-defs)

lemma well-order-on-Field:

\[ \text{well-order-on } A \ r \implies A = \text{Field } r \]

by (auto simp add: refl-on-def Field-def order-on-defs)

lemma well-order-on-Well-order:

\[ \text{well-order-on } A \ r \implies A = \text{Field } r \land \text{Well-order } r \]

using well-order-on-Field by auto

lemma Total-subset-Id:

assumes TOT: Total r and SUB: r ≤ Id

shows \( r = \{\} \cup (\exists a. \ r = \{(a,a)\}) \)

proof −

{assume \( r \neq \{\} \)

then obtain \( a, b \) where \( 1: (a,b) \in r \) by fast

hence \( a = b \) using SUB by blast

hence \( 2: (a,a) \in r \) using \( 1 \) by simp

{fix \( c, d \) assume \( (c,d) \in r \)

hence \( \{a,c,d\} \subseteq \text{Field } r \) using \( 1 \) unfolding Field-def by blast

hence \( ((a,c) \in r \lor (c,a) \in r \lor a = c) \land \\
(\lor (a,d) \in r \lor (d,a) \in r \lor a = d) \)

using TOT unfolding total-on-def by blast

hence \( a = c \land a = d \) using SUB by blast

} }
hence \( r \leq \{ (a,a) \} \) by auto
with 2 have \( \exists a. \ r = \{ (a,a) \} \) by blast

thus \(?thesis by blast
qed

lemma Linear-order-in-diff-Id:
assumes LI: Linear-order r and
IN1: a \in Field r and IN2: b \in Field r
shows \((a,b) \in r \Rightarrow (b,a) \notin r - Id\)
using assms unfolding order-on-defs total-on-def antisym-def Id-def refl-on-def
by force

25.4.2 The upper and lower bounds operators

Here we define upper ("above") and lower ("below") bounds operators. We think of \( r \) as a non-strict relation. The suffix "S" at the names of some operators indicates that the bounds are strict – e.g., \( \text{under}S \ a \) is the set of all strict lower bounds of \( a \) (w.r.t. \( r \)). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition under::'a rel \Rightarrow 'a \Rightarrow 'a set
where under r a \equiv \{ b. \ (b,a) \in r \}

definition underS::'a rel \Rightarrow 'a \Rightarrow 'a set
where underS r a \equiv \{ b. \ b \neq a \land (b,a) \in r \}

definition Under::'a rel \Rightarrow 'a set \Rightarrow 'a set
where Under r A \equiv \{ b \in Field r. \ \forall a \in A. \ (b,a) \in r \}

definition UnderS::'a rel \Rightarrow 'a set \Rightarrow 'a set
where UnderS r A \equiv \{ b \in Field r. \ \forall a \in A. \ b \neq a \land (b,a) \in r \}

definition above::'a rel \Rightarrow 'a \Rightarrow 'a set
where above r a \equiv \{ b. \ (a,b) \in r \}

definition aboveS::'a rel \Rightarrow 'a \Rightarrow 'a set
where aboveS r a \equiv \{ b. \ b \neq a \land (a,b) \in r \}

definition Above::'a rel \Rightarrow 'a set \Rightarrow 'a set
where Above r A \equiv \{ b \in Field r. \ \forall a \in A. \ (a,b) \in r \}

definition AboveS::'a rel \Rightarrow 'a set \Rightarrow 'a set
where AboveS r A \equiv \{ b \in Field r. \ \forall a \in A. \ b \neq a \land (a,b) \in r \}

definition ofilter :: 'a rel \Rightarrow 'a set \Rightarrow bool
where ofilter r A \equiv (A \leq Field r) \land (\forall a \in A. \ \text{under} r a \leq A)
Note: In the definitions of \textit{Above}[S] and \textit{Under}[S], we bounded comprehension by Field \(r\) in order to properly cover the case of \(A\) being empty.

\textbf{lemma} \textit{underS-subset-under}: \(\text{under}r \ a \subseteq \text{under} \ r \ a\)
\textbf{by} (auto simp add: underS-def under-def)

\textbf{lemma} \textit{underS-notIn}: \(a \notin \text{under}S \ r \ a\)
\textbf{by} (simp add: underS-def)

\textbf{lemma} \textit{Refl-under-underS}:
\textbf{assumes} \(\text{Refl} \ r \ a \in \text{Field} \ r\)
\textbf{shows} \(\text{under}r \ a = \text{under}S \ r \ a \cup \{a\}\)
\textbf{unfolding} under-def underS-def
\textbf{using} asms refl-on-def[of - r] by fastforce

\textbf{lemma} \textit{underS-empty}: \(a \notin \text{Field} \ r \Rightarrow \text{under}S \ r \ a = {}\)
\textbf{by} (auto simp: Field-def underS-def)

\textbf{lemma} \textit{under-Field}: \(\text{under}r \ a \leq \text{Field} \ r\)
\textbf{by} (unfold under-def Field-def, auto)

\textbf{lemma} \textit{underS-Field}: \(\text{under}S \ r \ a \leq \text{Field} \ r\)
\textbf{by} (unfold underS-def Field-def, auto)

\textbf{lemma} \textit{underS-Field2}:
\(a \in \text{Field} \ r \Rightarrow \text{under}S \ r \ a < \text{Field} \ r\)
\textbf{using} \textit{underS-notIn} underS-Field \textbf{by} fast

\textbf{lemma} \textit{underS-Field3}:
\(\text{Field} \ r \neq \{\} \Rightarrow \text{under}S \ r \ a < \text{Field} \ r\)
\textbf{by} (cases \(a \in \text{Field} \ r\), simp add: underS-Field2, auto simp add: underS-empty)

\textbf{lemma} \textit{AboveS-Field}: \(\text{Above}S \ r \ A \leq \text{Field} \ r\)
\textbf{by} (unfold AboveS-def Field-def, auto)

\textbf{lemma} \textit{under-incr}:
\textbf{assumes} \(\text{TRANS}: \text{trans} \ r \ \text{and} \ \text{REL}: (a,b) \in r\)
\textbf{shows} \(\text{under}r \ a \leq \text{under} \ r \ b\)
\textbf{proof} (unfold under-def, auto)
\textbf{fix} \(x\) \textbf{assume} \((x,a) \in r\)
\textbf{with} \textit{REL TRANS} trans-def[of \(r\)]
show \((x,b) \in r\) by blast
qed

\textbf{lemma underS-incr:}
\textbf{assumes} \(\text{TRANS:}\ trans\ r\ \text{and}\ \text{ANTISYM:}\ \text{antisym}\ r\ \text{and}\ \text{REL:}\ (a,b) \in r\)
\textbf{shows} \(\text{underS}\ r\ a \leq \text{underS}\ r\ b\)
\textbf{proof}(unfold\ underS-def,\ auto)
\quad \text{assume} \(\ast\): \(b \neq a\) \text{ and} \(\ast\ast\): \((b,a) \in r\)
\quad \text{with} \ \text{ANTISYM}\ \text{antisym-def[of}\ r} \ \text{REL}
\quad \text{show} \ False\ by\ blast
\textbf{next}
\quad \text{fix} \ x\ \text{assume} \(x \neq a\) \(\text{(x,a)} \in r\)
\quad \text{with} \ \text{REL} \ \text{TRANS} \ \text{trans-def[of}\ r}
\quad \text{show} \ (x,b) \in r\ by\ blast
\textbf{qed}

\textbf{lemma underS-incl-iff:}
\textbf{assumes} \(\text{LO:}\ \text{Linear-order}\ r\ \text{and}\ \text{INa:}\ a \in \text{Field}\ r\ \text{and}\ \text{INb:}\ b \in \text{Field}\ r\)
\textbf{shows} \((\text{underS}\ r\ a \leq \text{underS}\ r\ b) = ((a,b) \in r)\)
\textbf{proof}
\quad \text{assume} \((a,b) \in r\)
\quad \text{thus} \(\text{underS}\ r\ a \leq \text{underS}\ r\ b\ \text{using}\ LO\)
\quad \text{by} \ (\text{simp add: order-on-defs underS-incr})
\textbf{next}
\quad \text{assume} \(\ast\): \(\text{underS}\ r\ a \leq \text{underS}\ r\ b\)
\quad \text{\{assume} \(a = b\)
\quad \text{\quad hence} \((a,b) \in r\ \text{using}\ \text{assms}\)
\quad \text{\quad by} \ (\text{simp add: order-on-defs refl-on-def})\}
\quad \text{moreover}
\quad \text{\{assume} \(a \neq b\ \land\ (b,a) \in r\)
\quad \text{\quad hence} \(b \in \text{underS}\ r\ a\ \text{unfolding}\ \text{underS-def}\ \text{by}\ blast\)
\quad \text{\quad hence} \(b \in \text{underS}\ r\ b\ \text{using}\ \ast\ \text{by}\ blast\)
\quad \text{\quad hence} \(False\ \text{by}\ (\text{simp add: underS-notIn})\}
\quad \text{ultimately}
\quad \text{\{show} \((a,b) \in r\ \text{using}\ \text{assms}\)
\quad \text{\\quad order-on-defs[of Field r r] total-on-def[of Field r r] by blast}\)
\textbf{qed}

\textbf{25.5 Variations on Well-Founded Relations}

This subsection contains some variations of the results from \textit{Wellfounded}:

\begin{itemize}
  \item means for slightly more direct definitions by well-founded recursion;
  \item variations of well-founded induction;
\end{itemize}
THEORY “Order-Relation” 519

- means for proving a linear order to be a well-order.

25.5.1 Well-founded recursion via genuine fixpoints

lemma wfrec-fixpoint:
fixes r :: ('a * 'a) set and
H :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b
assumes WF: wf r and ADM: adm-wf r H
shows wfrec r H = H (wfrec r H)
proof (rule ext)
fix x
have wfrec r H x = H (cut (wfrec r H) r x) x
using wfrec[of r H] WF by simp
also have ∨ y. (y, x) : r ⇒ (cut (wfrec r H) r x) y = (wfrec r H) y
by (auto simp add: cut-apply)
hence H (cut (wfrec r H) r x) x = H (wfrec r H) x
using ADM adm-wf-def[of r H] by auto
}
finally show wfrec r H x = H (wfrec r H) x.
qed

25.5.2 Characterizations of well-foundedness

A transitive relation is well-founded iff it is “locally” well-founded, i.e., iff its restriction to the lower bounds of any element is well-founded.

lemma trans-wf-iff:
assumes trans r
shows wf r = (∀ a. wf (r Int (r^−1''{a} × r^−1''{a})))
proof –
obtain R where R-def: R = (λ a. r Int (r^−1''{a} × r^−1''{a})) by blast
{assume *: wf r
{fix a
have wf(R a)
using * R-def wf-subset[of r R a] by auto
}
}
moreover
{assume *: ∀ a. wf(R a)
have wf r
proof (unfold wf-def, clarify)
fix phi a
assume **: ∀ a. (∀ b. (b, a) ∈ r ⇒ phi b) ⇒ phi a
obtain chi where chi-def: chi = (λ b. (b, a) ∈ r ⇒ phi b) by blast
with * have wf (R a) by auto
hence (∀ b. (∀ c. (c, b) ∈ R a ⇒ chi c) ⇒ chi b) ⇒ (∀ b. chi b)
unfolding wf-def by blast
}
moreover
have \( \forall b. (\forall c. (c,b) \in R \rightarrow \chi c) \rightarrow \chi b \)
proof(auto simp add: chi-def R-def)

fix \( b \)
assume 1: \( (b,a) \in r \) and 2: \( \forall c. (c, b) \in r \land (c, a) \in r \rightarrow \phi c \)
hence \( \forall c. (c, b) \in r \rightarrow \phi c \)
using assms trans-def [of r] by blast
thus \( \phi b \) using ** by blast

qed

ultimately have \( \forall b. \chi b \)
by (rule mp)
with ** chi-def show \( \phi a \) by blast

qed

The next lemma is a variation of \( \text{wf-eq-minimal} \) from Wellfounded, allowing one to assume the set included in the field.

lemma \( \text{wf-eq-minimal2} \):
\[ \text{wf}\ r = (\forall A. A \subseteq \text{Field}\ r \land A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. \neg (a',a) \in r)) \]

proof-
let \( \phi = \lambda A. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. \neg (a',a) \in r) \)

have \( \text{wf}\ r = (\forall A. \phi A) \)
by (auto simp: ex-in-conv THEN sym, erule wfE-min, assumption, blast)

also have \( (\forall A. \phi A) = (\forall B \subseteq \text{Field}\ r. \phi B) \)
proof
assume \( \forall A. \phi A \)
thus \( \forall B \subseteq \text{Field}\ r. \phi B \) by simp

next
assume \( \ast: \forall B \subseteq \text{Field}\ r. \phi B \)
show \( \forall A. \phi A \)
proof(clarify)
fix A::’a set assume \( \ast: A \neq \{\} \)

obtain B where B-def: \( B = A \text{ Int (Field } r) \) by blast

show \( \exists a \in A. \forall a' \in A. (a',a) \notin r \)
proof(cases B = {})
assume Case1: \( B = \{\} \)

obtain a where 1: \( a \in A \land a \notin \text{Field } r \)
using Case1 unfolding B-def by blast

hence \( \forall a' \in A. (a',a) \notin r \) using 1 unfolding Field-def by blast

thus \( \text{thesis} \) using 1 by blast

next
assume Case2: \( B \neq \{\} \) have 1: \( B \subseteq \text{Field } r \) unfolding B-def by blast

obtain a where 2: \( a \in B \land (\forall a' \in B. (a',a) \notin r) \)
using Case2 1 * by blast

have \( \forall a' \in A. (a',a) \notin r \)
proof(clarify)
fix \( a' \) assume \( a' \in A \) and \(*: (a',a) \in r \)
hence \( a' \in B \) unfolding \( B\)-def Field-def by blast
thus \( \text{False using 2 ** by blast} \)
qed
thus \(?thesis using 2 unfolding B-def by blast \)
qed
qed
finally show \(?thesis by blast \)
qed

25.5.3 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is a well-order in a way which is more standard than via well-foundedness of the strict version of the relation.

lemma \( \text{Linear-order-wf-diff-Id} \):
assumes \( LI: \text{Linear-order r} \)
shows \( \text{wf}(r - \text{Id}) = (\forall A \in \text{Field r}. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r)) \)
proof (cases \( r \leq \text{Id} \))
assume \( \text{Case1: } r \leq \text{Id} \)
hence temp: \( r - \text{Id} = \{\} \) by blast
hence \( \text{wf}(r - \text{Id}) \) by (simp add: temp)
moreover
\{\text{fix } A \text{ assume } *: A \leq \text{Field r and } **: A \neq \{\} \}
\text{obtain } a \text{ where 1: } r = \{\} \lor r = \{(a,a)\} \text{ using LI}
\text{unfolding order-on-defs using Case1 Total-subset-Id by auto}
hence \( A = \{a\} \land r = \{(a,a)\} \) using ** unfolding Field-def by blast
hence \( \exists a \in A. \forall a' \in A. (a,a') \in r \) using 1 by blast
\}
ultimately show \(?thesis by blast \)
next
assume \( \text{Case2: } \neg r \leq \text{Id} \)
hence 1: \( \text{Field } r = \text{Field}(r - \text{Id}) \) using Total-Id-Field LI
\text{unfolding order-on-defs by blast}
show \(?thesis \)
proof
assume *: \( \text{wf}(r - \text{Id}) \)
show \( \forall A \leq \text{Field r}. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r) \)
proof (clarify)
fix \( A \) assume **: \( A \leq \text{Field r} \) and ***: \( A \neq \{\} \)
hence \( \exists a \in A. \forall a' \in A. (a',a) \notin r - \text{Id} \)
using 1 * unfolding wf-eq-minimal2 by simp
moreover have \( \forall a \in A. \forall a' \in A. (a,a') \in r = (a',a) \notin r - \text{Id} \)
using Linear-order-in-diff-Id[of r] ** LI by blast
ultimately show \( \exists a \in A. \forall a' \in A. (a,a') \in r \) by blast
qed
next
assume *: \( \forall A \leq \text{Field r}. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r) \)
show $\text{wf}(r - \text{Id})$
proof(unfold $\text{wf-eq-minimal2}$, clarify)
fix $A$
assume **: $A \subseteq \text{Field}(r - \text{Id})$ and ***: $A \neq \{\}$
hence $\exists a \in A. \forall a' \in A. (a,a') \in r$
using $1 \ast$ by simp
moreover have $\forall a \in A. \forall a' \in A. ((a,a') \in r) = ((a',a) \notin r - \text{Id})$
using $\text{Linear-order-in-diff-Id}[of r]$ ** $\text{LI mono-Field}[of r - \text{Id}]$ by blast
ultimately show $\exists a \in A. \forall a' \in A. (a',a) \notin r - \text{Id}$ by blast
qed
qed

corollary $\text{Linear-order-Well-order-iff}$:
assumes $\text{Linear-order } r$
shows $\text{Well-order } r = (\forall A \subseteq \text{Field } r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r))$
using assms unfolding $\text{well-order-on-def}$ using $\text{Linear-order-uf-diff-Id}[of r]$ by blast

end

26 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice

theory Hilbert-Choice
imports Nat Wellfounded
keywords specification :: thy-goal
begin

26.1 Hilbert’s epsilon

axiomatization $\text{Eps} :: (\:'a \Rightarrow \text{bool}) \Rightarrow \:'a$ where
someI: $P \Rightarrow \exists x. P (\text{Eps } P)$

syntax (epsilon)
$\text{-Eps} :: [\text{pttrn}, \text{bool}] \Rightarrow \:'a$ $(\exists a.\:-\:- [0, 10] 10)$
syntax (HOL)
$\text{-Eps} :: [\text{pttrn}, \text{bool}] \Rightarrow \:'a$ $(\exists a.\ :-\ :- [0, 10] 10)$
syntax
$\text{-Eps} :: [\text{pttrn}, \text{bool}] \Rightarrow \:'a$ $(\exists a.\ :-\ :- [0, 10] 10)$
translations
$\text{SOME } x. \; P = = \text{CONST Eps (\%x. } P)$

print-translation $\langle$
$\langle@\{\text{const-syntax Eps}\}, \text{fn } \Rightarrow \text{fn } [\text{Abs abs}] \Rightarrow$
let val $(x, t) = \text{Syntax-Trans.atomic-abs-tr' abs}$
in $\text{Syntax.const } @\{\text{syntax-const -Eps}\} x$ $\$$ t end$\rangle$ — to avoid eta-contraction of body
definition \textit{inv-into} :: 'a set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a) where
\textit{inv-into} \ A \ f \ = \ \%x. \ \text{SOME} \ y. \ A \ & \ f \ y \ = \ x

abbreviation \textit{inv} :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a) where
\textit{inv} \ = \ \textit{inv-into} \ \text{UNIV}

26.2 Hilbert’s Epsilon-operator

Easier to apply than \textit{someI} if the witness comes from an existential formula

\begin{itemize}
  \item lemma \textit{someI-ex} [elim?]: \exists \ x. \ P \ x \Longrightarrow \ P \ (\text{SOME} \ x. \ P \ x)
  \item apply (erule exE)
  \item apply (erule someI)
  \item done
\end{itemize}

Easier to apply than \textit{someI} because the conclusion has only one occurrence of \(P\).

\begin{itemize}
  \item lemma \textit{someI2}: \[ \exists \ a. \ P \ a; \ \exists \ x. \ P \ x \Longrightarrow \ Q \ x \ \Longrightarrow \ Q \ (\text{SOME} \ x. \ P \ x) \]
  \item by (blast intro: someI)
\end{itemize}

Easier to apply than \textit{someI2} if the witness comes from an existential formula

\begin{itemize}
  \item lemma \textit{someI2-ex}: \[ \exists \ a. \ P \ a; \ \exists \ x. \ P \ x \Longrightarrow \ Q \ x \ \Longrightarrow \ Q \ (\text{SOME} \ x. \ P \ x) \]
  \item by (blast intro: someI2)
\end{itemize}

\begin{itemize}
  \item lemma \textit{some-equality} [intro]:
    \[ \exists \ a. \ P \ a; \ \exists \ x. \ P \ x \Longrightarrow \ x=a \ \Longrightarrow \ (\text{SOME} \ x. \ P \ x) = a \]
  \item by (blast intro: someI2)
\end{itemize}

\begin{itemize}
  \item lemma \textit{some1-equality}: \[ \exists \ x. \ P \ a; \ P \ a \ \Longrightarrow \ (\text{SOME} \ x. \ P \ x) = a \]
  \item by blast
\end{itemize}

\begin{itemize}
  \item lemma \textit{some-eq-ex}: \( P \ (\text{SOME} \ x. \ P \ x) = (\exists \ x. \ P \ x) \)
  \item by (blast intro: someI)
\end{itemize}

\begin{itemize}
  \item lemma \textit{some-eq-trivial} [simp]: \( (\text{SOME} \ y. \ y=x) = x \)
  \item apply (rule some-equality)
  \item apply (rule refl, assumption)
  \item done
\end{itemize}

\begin{itemize}
  \item lemma \textit{some-sym-eq-trivial} [simp]: \( (\text{SOME} \ y. \ x=y) = x \)
  \item apply (rule some-equality)
  \item apply (rule refl)
  \item apply (erule sym)
  \item done
\end{itemize}

26.3 Axiom of Choice, Proved Using the Description Operator

\begin{itemize}
  \item lemma \textit{choice}: \( \forall \ x. \ \exists \ y. \ Q \ x \ y \Longrightarrow \ \exists \ f. \ \forall \ x. \ Q \ x \ (f \ x) \)
\end{itemize}
THEORY "Hilbert-Choice"

by (fast elim: someI)

lemma bchoice: \( \forall x \in S. \exists y. Q x y \implies \exists f. \forall x \in S. Q x (f x) \)
by (fast elim: someI)

lemma choice-iff: \( (\forall x. \exists y. Q x y) \iff (\exists f. \forall x. Q x (f x)) \)
by (fast elim: someI)

lemma choice-iff': \( (\forall x \in S. \exists y. Q x y) \iff (\exists f. \forall x \in S. Q x (f x)) \)
by (fast elim: someI)

lemma bchoice-iff: \( (\forall x \in S. \exists y. Q x y) \iff (\exists f. \forall x \in S. Q x (f x)) \)
by (fast elim: someI)

lemma dependent-nat-choice:
assumes 1: \( \exists x. P 0 x \) and
2: \( \forall x n. P n x \implies \exists y. P (Suc n) y \land Q n x y \)
shows \( \exists f. \forall n. P n (f n) \land Q n (f (Suc n)) \)
proof (intro exI allI conjI)
fix n def f \( \equiv \) rec-nat (SOME x. P 0 x) (\( \lambda n x. \) SOME y. P (Suc n) y \land Q n x y)
have P 0 (f 0) \land n. P n (f n) \implies P (Suc n) (f (Suc n)) \land Q n (f n) (f (Suc n))
using someI-ex[of 1] someI-ex[of 2] by (simp-all add: f-def)
then show P n (f n) Q n (f n) (f (Suc n))
by (induct n) auto
qed

26.4 Function Inverse

lemma inv-def: inv f = (%y. SOME x. f x = y)
by(simp add: inv-into-def)

lemma inv-into-into: x : f ` A ==\=> inv-into A f x : A
apply (simp add: inv-into-def)
apply (fast intro: someI2)
done

lemma inv-id [simp]: inv id = id
by (simp add: inv-into-def id-def)

lemma inv-into-f-f [simp]:
| inj-on f A; x : A | ==\=> inv-into A f (f x) = x
apply (simp add: inv-into-def inj-on-def)
apply (blast intro: someI2)
done

lemma \textit{inv-f-f}: \textit{inj f} \implies \textit{inv f (f x) = x}
by simp

lemma \textit{f-inv-into-f}: \textit{y : f'A} \implies \textit{f (inv-into A f y) = y}
apply (simp add: inv-into-def)
apply (fast intro: someI2)
done

lemma \textit{inv-into-f-eq}: \textit{| inj-on f A; x : A; f x = y |} \implies \textit{inv-into A f y = x}
apply (erule subst)
apply (fast intro: inv-into-f-f)
done

lemma \textit{inv-f-eq}: \textit{| inj f; f x = y |} \implies \textit{inv f y = x}
by (simp add: inv-into-f-eq)

lemma \textit{inj-transfer}:
assumes \textit{injf}: \textit{inj f} and \textit{minor}: \textit{!! y. y \in \text{range(f)} \implies P(f y)}
shows \textit{P x}
proof –
  have \textit{f x \in \text{range f}} by auto
  hence \textit{P(inv f (f x))} by (rule minor)
  thus \textit{P x} by (simp add: inv-into-f-f [OF injf])
qed

lemma \textit{inj-iff}: \textit{(inj f) = (inv f o f = id)}
apply (simp add: o-def fun-eq-iff)
apply (blast intro: inj-on-inverseI inv-into-f-f)
done

lemma \textit{inv-o-cancel}[simp]: \textit{inj f \implies inv f o f = id}
by (simp add: inj-iff)

lemma \textit{o-inv-o-cancel}[simp]: \textit{inj f \implies g o inv f o f = g}
by (simp add: comp-assoc)

lemma \textit{inv-into-image-cancel}[simp]:
\textit{inj-on f A \implies S \subseteq A \implies inv-into A f \circ f \circ S = S}
by (fastforce simp: image-def)

lemma \textit{inj-imp-surj-inv}: \textit{inj f \implies surj (inv f)}
by (blast intro: surjI inv-into-f-f)
lemma surj-f-inv: surj f ==> \( f inv f y = y \)
by (simp add: f-inv-into-f)

lemma inv-into-injective:
  assumes eq: inv-into A f x = inv-into A f y
  and x: x: fA
  and y: y: fA
  shows \( x=y \)
proof
  have \( f (inv-into A f x) = f (inv-into A f y) \) using eq by simp
  thus ?thesis by (simp add: f-inv-into-f x y)
qed

lemma inj-on-inv-into: B <= f'A ==> inj-on (inv-into A f) B
by (blast intro: inj-onI dest: inv-into-injective injD)

lemma bij-betw-inv-into: bij-betw f A B ==> bij-betw (inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-inv-into)

lemma surj-imp-inj-inv: surj f ==> inj (inv f)
by (simp add: inj-on-inv-into)

lemma surj-iff: (surj f) = (f o inv f = id)
by (auto intro: surjI simp: surj-imp-inv-into[fun-eq-iff[where 'b='a]])

lemma surj-iff-all: surj f if \( \forall x. f (g x) = x \)
  unfolding surj-iff by (simp add: o-def fun-eq-iff)

lemma surj-imp-inv-eq: surj f; \( \forall x. g (f x) = x \) \( \Longrightarrow \) inv f = g
apply (rule ext)
apply (erule_tac x = inv f x in spec)
apply (simp add: surj-imp-inv-f)
done

lemma bij-imp-bij-inv: bij f ==> \( f inv f \)
by (simp add: bij-imp-surj-imp-inv-imp-inv)

lemma inv-equality: \( \forall x. g (f x) = x \); \( \forall y. f (g y) = y \) \( \Longrightarrow \) inv f = g
apply (rule ext)
apply (auto simp add: inv-into-def)
done

lemma inv-inv-eq: \( \forall x. g (f x) = f \)
apply (rule inv-equality)
apply (auto simp add: bij-def surj-f-inv-f)
done
lemma inv-into-comp: 
| inj-on f (g ' A); inj-on g A; x : f ' g ' A | ==>
| inv-into A (f o g) x = (inv-into A g o inv-into (g ' A) f) x |
apply (rule inv-into-f-eq)
apply (fast intro: comp-inj-on)
apply (simp add: inv-into-into)
done

lemma o-inv-distrib: | bij f; bij g | ==>
| inv (f o g) = inv g o inv f |
apply (rule inv-equality)
apply (auto simp add: bij-def surj-f-inv-f)
done

lemma image-surj-f-inv-f: surj f ==>
| f ' (inv f ' A) = A |
apply (simp add: image-eq-UN surj-f-inv-f)
done

lemma image-inv-f-f: inj f ==>
| inv f ' (f ' A) = A |
by (simp add: image-eq-UN)

lemma inv-image-comp: inj f ==>
| inv f ' (f ' X) = X |
by (fact image-inv-f-f)

lemma bij-image-Collect-eq: bij f ==>
| f ' Collect P = {y. P (inv f y)} |
apply auto
apply (force simp add: bij-is-surj)
apply (blast intro: bij-is-surj [THEN surj-f-inv-f, symmetric])
done

lemma bij-vimage-eq-inv-image: bij f ==>
| f −' A = inv f ' A |
apply (auto simp add: bij-is-inj [THEN inv-into-f-f, symmetric])
done

lemma finite-fun-UNIVD1: 
assumes fin: finite (UNIV :: ('a ⇒ 'b) set)
and card: card (UNIV :: 'b set) ≠ Suc 0
shows finite (UNIV :: 'a set)
proof –
from fin have finb: finite (UNIV :: 'b set) by (rule finite-fun-UNIVD2)
with card have card (UNIV :: 'b set) ≥ Suc (Suc 0)
  by (cases card (UNIV :: 'b set)) (auto simp add: card-eq-0-iff)
then obtain n where card (UNIV :: 'b set) = Suc (Suc n) n = card (UNIV :: 'b set) − Suc (Suc 0) by auto
then obtain b1 b2 where b1b2: (b1 :: 'b) ≠ (b2 :: 'b) by (auto simp add: card-Suc-eq)
from fin have finite (range (∃f :: 'a ⇒ 'b. inv f b1)) by (rule finite-imageI)
moreover have UNIV = range (∃f :: 'a ⇒ 'b. inv f b1)
proof (rule UNIV-eq-I)
fix \( x :: 'a \)

from \( b1 \cdot b2 \) have \( x = \text{inv} (\lambda y. \text{if } y = x \text{ then } b1 \text{ else } b2) \cdot b1 \) by (simp add: inv-into-def)

thus \( x \in \text{range} (\lambda f.'a \Rightarrow 'b. \text{inv} f \cdot b1) \) by blast

qed

ultimately show \( \text{finite} (\text{UNIV} :: 'a \text{ set}) \) by simp

qed

Every infinite set contains a countable subset. More precisely we show that a set \( S \) is infinite if and only if there exists an injective function from the naturals into \( S \).

The “only if” direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set \( S \). The idea is to construct a sequence of non-empty and infinite subsets of \( S \) obtained by successively removing elements of \( S \).

**lemma infinite-countable-subset:**

assumes \( \neg \text{finite} (S :: 'a \text{ set}) \)

shows \( \exists f. \text{inj} (f :: \text{nat} \Rightarrow 'a) \land \text{range } f \subseteq S \)

— Courtesy of Stephan Merz

**proof**

\[
\text{def} \quad \text{Sseq} \equiv \text{rec-nat } S (\lambda n. T - \{ \text{SOME } e. e \in T \})
\]

\[
\text{def} \quad \text{pick} \equiv \lambda n. (\text{SOME } e. e \in \text{Sseq } n)
\]

\[
\{ \text{fix } n \text{ have } \text{Sseq } n \subseteq S \neg \text{finite} (\text{Sseq } n) \text{ by (induct } n) \text{ (auto simp add: Sseq-def inf)} \}
\]

moreover then have \( \ast \cdot \forall n. \text{pick } n \in \text{Sseq } n \)

unfolding \( \text{pick-def} \) by (subst (asm) finite.simps) (auto simp add: ex-in-conv intro: someI-ex)

ultimately have \( \text{range } \text{pick} \subseteq S \) by auto

moreover

\[
\{ \text{fix } n m \text{ have } \text{pick } n \notin \text{Sseq } (n + \text{Suc } m) \text{ by (induct } m) \text{ (auto simp add: Sseq-def pick-def)} \}
\]

\[
\text{with } \ast \text{ have } \text{pick } n \neq \text{pick } (n + \text{Suc } m) \text{ by auto}
\]

then have \( \text{inj } \text{pick} \) by (intro linorder-injI) (auto simp add: less-iff-Suc-add)

ultimately show \( \text{thesis} \) by blast

qed

**lemma infinite-iff-countable-subset:**

\( \neg \text{finite } S \leftrightarrow (\exists f. \text{inj} (f :: \text{nat} \Rightarrow 'a) \land \text{range } f \subseteq S) \)

— Courtesy of Stephan Merz

using finite-imageD finite-subset infinite-UNIV-char-0 infinite-countable-subset

by auto

**lemma image-inv-into-cancel:**

assumes \( \text{SURJ: } f' \cdot A' \text{ and } \text{SUB: } B' \subseteq A' \)

shows \( f' ((\text{inv-into } A \cdot f) \cdot B') = B' \)

using assms
proof (auto simp add: f-inv-into-f)
let \( ?f' = (\text{inv-into } A f) \)
fix \( a' \) assume \( *: a' \in B' \)
then have \( a' \in A' \) using SUB by auto
then have \( a' = f (\?f' a') \)
using SURJ by (auto simp add: f-inv-into-f)
then show \( a' \in f' (\?f' \cdot B') \) using \( * \) by blast
qed

lemma inv-into-inv-into-eq:
assumes bij-betw f A A'
shows inv-into A' (inv-into A f) a = f a
proof
let \( \?f' = \text{inv-into } A f \)
let \( \?f'' = \text{inv-into } A' \?f' \)
have 1: bij-betw \( ?f' \) A A'
using assms by (auto simp add: bij-betw-inv-into)
obtain a' where 2: \( a' \in A' \) and 3: \( ?f' a' = a \)
using 1 (\( a \in A \)) unfolding bij-betw-def by force
hence \( ?f'' a' = a' \)
using \( \langle a \in A \rangle \) 1 3 by (auto simp add: f-inv-into-f bij-betw-def)
moreover have \( f a = a' \) using assms 2 3
by (auto simp add: bij-betw-def)
ultimately show \( ?f'' a' = f a \) by simp
qed

lemma inj-on-iff-surj:
assumes \( A \neq \{\} \)
shows \( (\exists f. \text{inj-on } f A \land f \cdot A \leq A') \iff (\exists g. g \cdot A' = A) \)
proof safe
fix \( f \) assume INJ: \( \text{inj-on } f A \) and INCL: \( f \cdot A \leq A' \)
let \( \?phi = \lambda a. a \in A \land f a = a' \) let \( \?csi = \lambda a. a \in A \)
let \( ?g = \lambda a'. if \ a' \in f \cdot A \ then (\text{SOME } a. \?phi a' a) \ else (\text{SOME } a. \?csi a) \)
have \( ?g \cdot A' = A \)
proof
show \( ?g \cdot A' \leq A \)
proof clarify
fix \( a' \) assume \( *: a' \in A' \)
show \( ?g a' \in A \)
proof cases
assume Case1: \( a' \in f \cdot A \)
then obtain a where \( \?phi a' a \) by blast
hence \( \?phi a' (\text{SOME } a. \?phi a' a) \) using someI[of \( \?phi a' a \)] by blast
with Case1 show \( \theta h e s i s \) by auto
next
assume Case2: \( a' \notin f \cdot A \)
hence \( \?csi (\text{SOME } a. \?csi a) \) using assms someI-ex[of \( \?csi \)] by blast
with Case2 show \( \theta h e s i s \) by auto
qed
qed
next
show $A \subseteq \exists g \cdot A'$
proof
\begin{itemize}
\item[\{ fix $a$ assume $\ast$: $a \in A$ ]
\item[let $?b = \text{SOME} \ a a$. $\phi (f \ a) \ a$ ]
\item[have $\phi (f \ a) \ a$ using $\ast$ by auto ]
\item[hence $I$: $\phi (f \ a) \ ?b$ using someI[of $\phi (f \ a) \ a$] by blast ]
\item[hence $g(f \ a) = ?b$ using $\ast$ by auto ]
\item[moreover have $a = ?b$ using $\text{INJ} \ \ast$ by simp ]
\item[ultimately have $g(f \ a) = ?b$ using $\ast$ by auto ]
\item[\}]
\item[thus $\ast$thesis by force ]
\end{itemize}
qed

next
fix $g$
let $?f = \text{inv-into} \ A' \ g$

have inj-on $?f \ (g \ ' \ A')$
\begin{itemize}
\item[(auto simp add: inj-on-inv-into)]
\item[moreover]
\begin{itemize}
\item[\{ fix $a \ ' \ a'$ assume $\ast$: $a' \in A'$ ]
\item[let $\phi = \lambda b \ '. \ b' \in A' \land g \ b' = g \ a'$ ]
\item[have $\phi \ a'$ using $\ast$ by auto ]
\item[hence $\phi(\text{SOME} \ b' \ '. \ \phi \ b')$ using someI[of $\phi$] by blast ]
\item[hence $?f(g \ a') \in A'$ unfolding inv-into-def by auto ]
\item[\}]
\item[ultimately show $\exists f. \ \text{inj-on} \ f \ (g \ ' \ A') \land f \ ' \ g \ ' \ A' \subseteq A' \ by \ auto ]
\item[auto]
\end{itemize}
\item[auto]
\end{itemize}
qed

lemma $\text{Ex-inj-on-UNION-Sigma}$:
$\exists f. \ (\text{inj-on} \ f \ (\bigcup \ i \in I. \ A \ i)) \land \ f \ ' \ (\bigcup \ i \in I. \ A \ i) \subseteq (\text{SIGMA} \ i : I. \ A \ i))$

proof
\begin{itemize}
\item[let $\phi = \lambda a \ i. \ i \in I \land a \in A \ i$ ]
\item[let $\phi = \lambda a. \ \text{SOME} \ i. \ \phi \ a$ ]
\item[let $\phi = \lambda a. \ (?sm \ a, \ a)$ ]
\item[have inj-on $?f \ (\bigcup \ i \in I. \ A \ i)$ unfolding inj-on-def by auto ]
\item[moreover]
\begin{itemize}
\item[\{ fix $i \ a$ assume $i \in I$ and $a \in A \ i$ ]
\item[hence $?sm \ a \in I \land a \in A(?sm \ a)$ using someI[of $\phi \ a \ i$] by auto ]
\item[\}]
\item[ultimately]
\begin{itemize}
\item[show inj-on $?f \ (\bigcup \ i \in I. \ A \ i) \land \ ?f \ i \ (\bigcup \ i \in I. \ A \ i) \subseteq (\text{SIGMA} \ i : I. \ A \ i)$] by auto
\item[auto]
\end{itemize}
\item[auto]
\end{itemize}
\end{itemize}

lemma $\text{inv-unique-comp}$:
assumes \( fg \colon f \circ g = id \)
and \( gf \colon g \circ f = id \)
shows \( \text{inv} f = g \)
using \( fg \, gf \, \text{inv-equalit}[\text{of} \, g \, f] \) by (auto simp add: fun-eq-iff)

### 26.5 The Cantor-Bernstein Theorem

**lemma** Cantor-Bernstein-aux:

shows \( \exists A' h. A' \leq A \wedge \)
\( (\forall a \in A'. \, a \notin g'(B - f' A')) \wedge \)
\( (\forall a \in A'. \, h \, a = f \, a) \wedge \)
\( (\forall a \in A - A'. \, h \, a \in B - (f' A') \wedge a = g(h \, a)) \)

**proof**

obtain \( H \) where \( H \text{-def}: H = (\lambda A'. \, A - (g'(B - (f' A')))) \) by blast
have \( 0 \colon \text{mono} \, H \) unfolding mono-def H-def by blast
then obtain \( A' \) where \( 1 \colon H \, A' = A' \) \text{using} lfp-unfold by blast
hence \( 2 \colon A' = A - (g'(B - (f' A')) \) unfolding H-def by simp
hence \( 3 \colon A' \leq A \) by blast
have \( 4 \colon \forall a \in A'. \, a \notin g'(B - f' A') \)
using \( 2 \) by blast
have \( 5 \colon \forall a \in A - A'. \, \exists b \in B - (f' A'). \, a = g \, b \)
using \( 2 \) by blast

obtain \( h \) where \( h \text{-def}: \)
\( h = (\lambda a. \, \text{if} \, a \in A' \text{ then } f \, a \text{ else } \text{SOME} \, b. \, b \in B - (f' A') \wedge a = g \, b) \) by blast
hence \( \forall a \in A'. \, h \, a = f \, a \) by auto
moreover
have \( \forall a \in A - A'. \, h \, a \in B - (f' A') \wedge a = g(h \, a) \)
proof
  fix \( a \) assume \( \ast: a \in A - A' \)
let \( ?\phi = \lambda b. \, b \in B - (f' A') \wedge a = g \, b \)
have \( h \, a = (\text{SOME} \, b. \, ?\phi \, b) \) using h-def \( \ast \) by auto
moreover have \( \exists b. \, ?\phi \, b \) using \( 5 \) \( \ast \) by auto
ultimately show \( ?\phi \, (h \, a) \) using someI-ex[\text{of} \, ?\phi] by auto
qed
ultimately show \( ?\text{thesis} \) using \( 3 \, 4 \) by blast
qed

**theorem** Cantor-Bernstein:

assumes \( \text{INJ1: inj-on} \, f \, A \) \text{ and } \( \text{SUB1:} \, f' A \leq B \) \text{ and } \( \text{INJ2:} \, \text{inj-on} \, g \, B \) \text{ and } \( \text{SUB2:} \, g' B \leq A \)
shows \( \exists h. \, \text{bij-betw} \, h \, A \, B \)

**proof**

obtain \( A' \) and \( h \) where \( 0 \colon A' \leq A \)
1. \( \forall a \in A'. \, a \notin g'(B - f' A') \) \text{ and }
2. \( \forall a \in A'. \, h \, a = f \, a \) \text{ and }
3. \( \forall a \in A - A'. \, h \, a \in B - (f' A') \wedge a = g(h \, a) \)
using Cantor-Bernstein-aux[\text{of} \, A \, g \, B \, f] by blast
have \( \text{inj-on} \, h \, A \)
proof (intro inj-onI)
fix a1 a2
assume 4: a1 ∈ A and 5: a2 ∈ A and 6: h a1 = h a2
show a1 = a2
proof(cases a1 ∈ A')
assume Case1: a1 ∈ A'
show ?thesis
proof(cases a2 ∈ A')
assume Case11: a2 ∈ A'
hence f a1 = f a2 using Case1 2 6 by auto
thus ?thesis using INJ1 Case1 Case11 0
unfolding inj-on-def by blast
next
assume Case12: a2 ∉ A'
hence False using 3 5 2 6 Case1 by force
thus ?thesis by simp
qed
next
assume Case2: a1 ∉ A'
show ?thesis
proof(cases a2 ∈ A')
assume Case21: a2 ∈ A'
hence False using 3 4 2 6 Case2 by auto
thus ?thesis by simp
next
assume Case22: a2 ∉ A'
hence a1 = g(h a1) ∧ a2 = g(h a2) using Case2 4 5 3 by auto
thus ?thesis using 6 by simp
qed
qed

moreover
have h' A = B
proof safe
fix a assume a ∈ A
thus h a ∈ B using SUB1 2 3 by (cases a ∈ A') auto
next
fix b assume *: b ∈ B
show b ∈ h' A
proof(cases b ∈ f' A')
assume Case1: b ∈ f' A'
then obtain a where a ∈ A' ∧ b = f a by blast
thus ?thesis using 2 0 by force
next
assume Case2: b ∉ f' A'
hence g b ∉ A' using 1 * by auto
hence 4: g b ∈ A − A' using * SUB2 by auto
hence h(g b) ∈ B ∧ g(h(g b)) = g b
using 3 by auto

hence \( h(g \, b) = b \) using \(*\) INJ2 unfolding inj-on-def by auto

thus \( ?thesis \) using 4 by force

qed

ultimately show \( ?thesis \) unfolding bij-betw-def by auto

qed

26.6 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the case-prod Operator

Looping simprule

lemma split-paired-Eps: \( (\text{SOME } x. \, P \, x) = (\text{SOME } (a,b). \, P(a,b)) \)

by simp

lemma Eps-split: \( \text{Eps } (\text{split } P) = (\text{SOME } xy. \, P \, (\text{fst } xy) \, (\text{snd } xy)) \)

by (simp add: split-def)

lemma Eps-split-eq [simp]: \( (\text{Eps } \, P, \, x \, y) = (x, y) \)

by blast

A relation is wellfounded iff it has no infinite descending chain

lemma wf-iff-no-infinite-down-chain:

\( \text{wf } r = (\neg \exists f. \neg \forall i. \, (f(Suc \, i), \, f \, i) \in r) \)

apply (simp only: wf-eq-minimal)

apply (rule iffI)

apply (rule notI)

apply (erule exE)

apply (erule_tac x = \{ w. \exists i. \, w=f \, i \} in allE, blast)

apply (erule contrapos-np, simp, clarify)

apply (erule_tac \% n. \text{rec-nat } x \, (\% i. \text{z}:Q \, \text{z,y} :r) \, n \in Q)

apply (rule_tac x = \text{rec-nat } x \, (\% i. \text{z}:Q \, \text{z,y} :r) \, \text{in } exI)

apply (rule allI, simp)

apply (rule someI2-ex, blast, blast)

apply (rule allI)

apply (induct_tac n, simp-all)

apply (rule someI2-ex, blast+)

done

lemma wf-no-infinite-down-chainE:

assumes \( \text{wf } r \) obtains \( k \) where \( (f \, \text{Suc } k), \, f \, k) \notin r \)

using \( \text{wf } r \) \text{wf-iff-no-infinite-down-chain[of } r … by blast

A dynamically-scoped fact for TFL

lemma tfl-some: \( \forall P \, x. \, P \, x \, \rightarrow \, P \, (\text{Eps } P) \)

by (blast intro: someI)
26.7 Least value operator

definition
LeastM :: [a => 'b::ord, 'a => bool] => 'a where
LeastM m P == SOME x. P x & (∀ y. P y ---> m x <= m y)

syntax
-LeastM :: [pttrn, 'a => 'b::ord, bool] => 'a (LEAST - WRT -. - [0, 4, 10])

translations
LEAST x WRT m. P == CONST LeastM m (%x. P)

lemma LeastMI2:
P x ==> (!y. P y ==> m x <= m y)
==> (!x. P x ==> ∀ y. P y ---> m x <= m y ==> Q x)
==> Q (LeastM m P)
apply (simp add: LeastM-def)
apply (rule someI2-ex, blast, blast)
done

lemma LeastM-equality:
P k ==>(∀ x y. ((x,y):r^=) = (y,x):r^=) ==> P k
==> (∃ x. P x & (∀ y. P y ---> (m x,m y):r^=))
apply (drule wf-trancl [THENwf-eq-minimal [THEN iffD1]])
apply (drule-tac x = m'Collect P in spec, force)
done

lemma ex-has-least-nat:
P k ==>(∀ x. P x & (∀ y. P y ---> m x <= (m y::nat)))
apply (simp only: pred-nat-trancl-eq-le [symmetric])
apply (rule wf-pred-nat [THEN wf-linord-ex-has-least])
apply (simp add: less-eq linorder-not-le pred-nat-trancl-eq-le, assumption)
done

lemma LeastM-nat-lemma:
P k ==>(∃ x. P (LeastM m P) & (∀ y. P y ---> m (LeastM m P) <= (m y::nat)))
apply (simp add: LeastM-def)
apply (rule someI-ex)
apply (erule ex-has-least-nat)
done

lemmas LeastM-natI = LeastM-nat-lemma [THEN conjunct1]
lemma LeastM-nat-le: \[ P \ x \Rightarrow m \ (\text{LeastM} \ m \ P) \leq (m \ x :: \text{nat}) \]

by (rule LeastM-nat-lemma [THEN conjunct2, THEN spec, THEN mp], assumption, assumption)

26.8 Greatest value operator

definition
\[
\text{GreatestM} :: [\text{a} \Rightarrow \text{b} :: \text{ord}, \text{a} \Rightarrow \text{bool}] \Rightarrow \text{a} \ \text{where} \\
\text{GreatestM} \ m \ P \Rightarrow \text{SOME} \ x. \ P \ x \ & \ (\forall y. \ P \ y \Rightarrow m \ y < m \ x)
\]

definition
\[
\text{Greatest} :: (\text{a} :: \text{ord} \Rightarrow \text{bool}) \Rightarrow \text{a} \ \text{(binder GREATEST 10) where} \\
\text{Greatest} \Rightarrow \text{GreatestM} (\% \ x. \ x)
\]

syntax
\[
-\text{GreatestM} :: [\text{pttrn}, \text{a} \Rightarrow \text{b} :: \text{ord}, \text{bool}] \Rightarrow \text{a} \\
\text{(GREATEST - WRT - [0, 4, 10] 10)}
\]

translations
\[
\text{GREATEST} \ x \ \text{WRT} \ m. \ P \Rightarrow \text{CONST} \ \text{GreatestM} \ m \ (\% \ x. \ P)
\]

lemma GreatestMI2:
\[
P \ x \Rightarrow (!y. \ P \ y \Rightarrow m \ y \leq m \ x) \\
\Rightarrow (!x. \ P \ x \Rightarrow \forall y. \ P \ y \Rightarrow m \ y \leq m \ x \Rightarrow Q \ x)
\]

apply (simp add: GreatestM-def)
apply (rule someI2-ex, blast, blast)
done

lemma GreatestM-equality:
\[
P \ k \Rightarrow (!x. \ P \ x \Rightarrow m \ x \leq m \ k) \\
\Rightarrow m \ (\text{GREATEST} \ x \ \text{WRT} \ m. \ P \ x) = (m \ k :: \text{a} :: \text{order})
\]

apply (rule-tac m = m in GreatestMI2, assumption, blast)
apply (blast intro!: order-antisym)
done

lemma Greatest-equality:
\[
P \ (k :: \text{a} :: \text{order}) \Rightarrow (!x. \ P \ x \Rightarrow x \leq k) \Rightarrow (\text{GREATEST} \ x. \ P \ x) = k
\]

apply (simp add: Greatest-def)
apply (erule GreatestM-equality, blast)
done

lemma ex-has-greatest-nat-lemma:
\[
P \ k \Rightarrow \forall x. \ P \ x \Rightarrow (\exists y. \ P \ y \ & \sim ((m \ y :: \text{nat}) \leq m \ x)) \\
\Rightarrow \exists y. \ P \ y \ & \sim (m \ y \ < m \ k + n)
\]

apply (induct n, force)
apply (force simp add: le-Suc-eq)
done

lemma ex-has-greatest-nat:
THEORY “Hilbert-Choice”

\[ P \mathbin{\Rightarrow} \forall y. P y \mathbin{\Rightarrow} m y < b \]
\[ \mathbin{\Rightarrow} \exists x. P x \& (\forall y. P y \mathbin{\Rightarrow} (m y::\text{nat}) <= m x) \]
apply (rule ccontr)
apply (cut-tac P = P and n = b - m k in ex-has-greatest-nat-lemma)
apply (subgoal-tac [3] m k <= b, auto)
done

lemma GreatestM-nat-lemma:
\[ P \mathbin{\Rightarrow} \forall y. P y \mathbin{\Rightarrow} m y < b \]
\[ \mathbin{\Rightarrow} (m x::\text{nat}) <= m (GreatestM m P) \]
apply (simp add: GreatestM-def)
apply (rule someI-ex)
apply (erule ex-has-greatest-nat, assumption)
done

lemmas GreatestM-natI = GreatestM-nat-lemma [THEN conjunct1]

lemma GreatestM-nat-le:
\[ P x \mathbin{\Rightarrow} \forall y. P y \mathbin{\Rightarrow} m y < b \]
\[ \mathbin{\Rightarrow} (m x::\text{nat}) <= m (GreatestM m P) \]
apply (blast dest: GreatestM-nat-lemma [THEN conjunct2, THEN spec, of P])
done

Specialization to GREATEST.

lemma GreatestI: \[ P \mathbin{\Rightarrow} \forall y. P y \mathbin{\Rightarrow} y < b \mathbin{\Rightarrow} P \mathbin{\Rightarrow} \] (GREATEST x. P x)
apply (simp add: Greatest-def)
apply (rule GreatestM-natI, auto)
done

lemma Greatest-le:
\[ P x \mathbin{\Rightarrow} \forall y. P y \mathbin{\Rightarrow} y < b \mathbin{\Rightarrow} \] (GREATEST x. P x)
apply (simp add: Greatest-def)
apply (rule GreatestM-nat-le, auto)
done

26.9 An aside: bounded accessible part

Finite monotone eventually stable sequences

lemma finite-mono-remains-stable.implies-strict-prefix:
fixes f :: nat ⇒ 'a::order
assumes S: finite (range f) mono f and eq: \( \forall n. f n = f (\text{Suc } n) \mathbin{\Rightarrow} f (\text{Suc } n) = f (\text{Suc } n))
shows \( \exists N. (\forall n \leq N. \forall m \leq N. m < n \mathbin{\Rightarrow} f m < f n) \& (\forall n \geq N. f N = f n) \)
using assms
proof -
have \( \exists n. f n = f (\text{Suc } n) \)
proof (rule ccontr)
  assume ¬ ?thesis
  then have ∀ n. f n ≠ f (Suc n) by auto
  then have ∀ n. f n < f (Suc n)
    using (mono f) by (auto simp: le-less mono-iff-le-Suc)
  with lift-Suc-mono-less-iff[of f]
  have *: ∀ n m. n < m ⇒ f n < f m by auto
  have inj f
  proof (intro injI)
    fix x y
    assume f x = f y
    then show x = y by (cases x y rule: linorder-cases) (auto dest: *)
  qed
  with ⟨finite (range f)⟩
  have finite (∪ n. f n)
    by (rule finite-imageD)
  then show False by simp
  qed
then obtain n where n: f n = f (Suc n)
  def N ≡ LEAST n. f n = f (Suc n)
  have N: f N = f (Suc N)
    unfolding N-def using n by (rule LeastI)
  show ?thesis
  proof (intro exI[of - N] conjI allI impI)
    fix n assume N ≤ n
    then have ∀ m. N ≤ m ⇒ m ≤ n ⇒ f m = f N
      proof (induct rule: dec-induct)
        case (step n) then show ?case
          using eq[rule-format, of n - 1] N
          by (cases i) (auto simp add: le-Suc-eq)
      qed simp
    from this[of n] (N ≤ n) show f N = f n by auto
  next
    fix m n :: nat assume m < n n ≤ N
    then show f m < f n
      proof (induct rule: less-Suc-induct[consumes 1])
        case (1 i)
        then have i < N by simp
        then have f i ≠ f (Suc i)
          unfolding N-def by (rule not-less-Least)
        with (mono f) show ?case by (simp add: mono-iff-le-Suc less-le)
      qed
  qed
  qed
lemma finite-mono-strict-prefix-implies-finite-fixpoint:
  fixes f :: nat ⇒ 'a set
  assumes S: ∀ i. f i ⊆ S finite S
    and inj: ∃ N. (∀ n ≤ N. ∀ m ≤ N. m < n −→ f m ⊂ f n) ∧ (∀ n ≥ N. f N = f n)
  shows f (card S) = (∪ n. f n)
proof
from inj obtain N where inj: (∀ n≤N. ∀ m≤N. m < n → f m ⊂ f n) and
eq: (∀ n≥N. f N = f n) by auto

(1) fix i have i ≤ N ⇒ i ≤ card (f i)
proof (induct i)
  case 0 then show case by simp
next
  case (Suc i)
  with inj [rule-format, of Suc i i]
  have (f i) ⊂ (f (Suc i)) by auto
  moreover have finite (f (Suc i)) using S by (rule finite-subset)
  ultimately have card (f i) < card (f (Suc i)) by (intro psubset-card-mono)
  with Suc show case using inj by auto
  qed

then have N ≤ card (f N) by simp
also have ... ≤ card S using S by (intro card-mono)
finally have f (card S) = f N using eq by auto
then show ?thesis using eq inj [rule-format, of N]
  apply auto
  apply (case-tac n < N)
  apply (auto simp: not_less)
  done
qed

26.10 More on injections, bijections, and inverses

lemma infinite-imp-bij-betw:
assumes INF: ¬ finite A
shows ∃ h. bij-betw h A (A − {a})
proof (cases a ∈ A)
  assume Case1: a ∉ A hence A − {a} = A by blast
  thus ?thesis using bij-betw-id [of A] by auto
next
  assume Case2: a ∈ A
find-theorems ¬ finite -
  have ¬ finite (A − {a}) using INF by auto
  with infinite-iff-countable-subset [of A − {a}] obtain f: nat ⇒ 'a
  where 1: inj f and 2: f · UNIV ≤ A − {a} by blast
  obtain g where g-def: g = (λ n. if n = 0 then a else f (Suc n)) by blast
  obtain A' where A'-def: A' = g · UNIV by blast
  have temp: ∀ y. f y ≠ a using 2 by blast
  have 3: inj-on g UNIV ∧ g · UNIV ≤ A ∧ a ∈ g · UNIV
proof (auto simp add: Case2 g-def, unfold inj-on-def, intro ballI impI,
  case-tac x = 0, auto simp add: 2)
  fix y assume a = (if y = 0 then a else f (Suc y))
  thus y = 0 using temp by (case-tac y = 0, auto)
next
fix \( x \) \( y \)
assume \( f \) \((\text{Suc } x) = (\text{if } y = 0 \text{ then } a \text{ else } f \text{ (Suc } y))\)
thus \( x = y \) using \( 1 \) \( \text{temp} \) unfolding inj-on-def \( \text{by} \) \( (\text{case-tac } y = 0, \text{auto}) \)
next
fix \( n \) show \( f \) \((\text{Suc } n) \in A \) using \( 2 \) by blast
qed

hence \( 4: \text{bij-betw} \ g \ \text{UNIV} \ A' \land a \in A' \land A' \leq A \)
using inj-on-imp-bij-betw[\( \text{of } g \)] unfolding A'-def \( \text{by} \) \( \text{auto} \)
hence \( 5: \text{bij-betw} \ (\text{inv } g) \ A' \ \text{UNIV} \)
by \( (\text{auto } \text{simp add: bij-betw-inv-into}) \)

obtain \( n \) where \( g \ n = a \) using \( 3 \) by \( \text{auto} \)
hence \( 6: \text{bij-betw} \ g \ (\text{UNIV} - \{n\}) \ (A' - \{a\}) \)
using \( 3 \ 4 \) unfolding A'-def
by clarify \( (\text{rule bij-betw-subset, auto simp: image-set-diff}) \)

obtain \( v \) where \( v\)-def: \( v = (\lambda m. \text{if } m < n \text{ then } m \text{ else } \text{Suc } m) \) by \( \text{blast} \)

proof\((\text{unfold bij-betw-def inj-on-def, intro conjI, clarify})\)
fix \( m1 \) \( m2 \) assume \( v \ m1 = v \ m2 \)
thus \( m1 = m2 \)
by\((\text{case-tac } m1 < n, \text{case-tac } m2 < n, \text{auto } \text{simp add: inj-on-def } v\)-def, case-tac m2 < n, \( \text{auto} \))
next
show \( v \) ' \( \text{UNIV} = \text{UNIV} - \{n\} \)
proof\((\text{auto } \text{simp add: } v\)-def\)
fix \( m \) assume \( * : m \neq n \) \( \text{and} \ ** : m \notin \text{Suc} \ ' \ \{m\'. \sim m' < n\} \)
{assume \( n \leq m \) \( \text{with} \) \( \text{have} \ 71: \text{Suc } n \leq m \) \( \text{by} \) \( \text{auto} \)
then obtain \( m' \) \( \text{where} \ 72: m = \text{Suc } m' \) \( \text{using} \ \text{Suc-le-D} \) \( \text{by} \) \( \text{auto} \)
with \( 71 \) \( \text{have} \ n \leq m' \) \( \text{by} \) \( \text{auto} \)
with \( 72 \ ** \) \( \text{have} \ False \) \( \text{by} \) \( \text{auto} \)
}
thus \( m < n \) \( \text{by force} \)
qed
qed

obtain \( h' \) \( \text{where} \ h'-\text{def}: h' = g \circ v \circ (\text{inv } g) \) \( \text{by} \) \( \text{blast} \)
hence \( 8: \text{bij-betw} \ h' \ A' \ (A' - \{a\}) \) \( \text{using} \ 5 \ 7 \ 6 \)
by \( (\text{auto } \text{simp add: bij-betw-trans}) \)

obtain \( h \) \( \text{where} \ h\)-def: \( h = (\lambda b. \text{if } b \in A' \text{ then } h' \ b \text{ else } b) \) \( \text{by} \) \( \text{blast} \)
have \( \forall b \in A'. \ h \ b = h' \ b \) \( \text{unfolding } h\)-def \( \text{by} \) \( \text{auto} \)
hence \( \text{bij-betw} \ h \ A' \ (A' - \{a\}) \) \( \text{using} \ 8 \ \text{bij-betw-cong}[\text{of } A' \ h] \) \( \text{by} \) \( \text{auto} \)
moreover
\{\text{have } \forall b \in A - A', \ h \ b = b \ \text{unfolding } h\)-def \( \text{by} \) \( \text{auto} \)
hence \( \text{bij-betw} \ h \ (A - A') \ (A - A') \)
using \( \text{bij-betw-cong}[\text{of } A - A' \ h \ \text{id}] \) \( \text{bij-betw-Id}[\text{of } A - A'] \) \( \text{by} \) \( \text{auto} \)
\}
moreover
have \((A' \text{ Int } (A - A')) = \{\} \land A' \cup (A - A') = A) \land \((A' - \{a\}) \text{ Int } (A - A') = \{\} \land (A' - \{a\}) \cup (A - A') = A - \{a\})
use 4 by blast
ultimately have bij-betw h A (A - \{a\})
using bij-betw-combine[of h A' A' - \{a\} A - A' A - A'] by simp
thus ?thesis by blast
qed

lemma infinite-imp-bij-betw2:
assumes INF: \(\neg\) finite A
shows \(\exists h.\ bij-betw h A (A \cup \{a\})\)
proof (cases a \(\in\) A)
assume Case1: a \(\in\) A hence A \(\cup\) \{a\} = A by blast
thus ?thesis using bij-betw-id[of A] by auto
next
let \(?A' = A \cup \{a\}\)
assume Case2: a \(\notin\) A hence A = \(?A' - \{a\}\) by blast
moreover have \(\neg\) finite \(?A'\) using INF by auto
ultimately obtain f where bij-betw f \(?A' A\)
using infinite-imp-bij-betw[of \(?A' a\)] by auto
hence bij-betw(inv-into \(?A' f\) A \(?A'\) using bij-betw-inv-into by blast
thus ?thesis by auto
qed

lemma bij-betw-inv-into-left:
assumes BIJ: bij-betw f A A' and IN: a \(\in\) A
shows (inv-into A f) (f a) = a
using assms unfolding bij-betw-def
by clarify (rule inv-into-f-f)

lemma bij-betw-inv-into-right:
assumes bij-betw f A A' a' \(\in\) A'
s show s (inv-into A f a') = a'
using assms unfolding bij-betw-def using f-inv-into-f by force

lemma bij-betw-inv-into-subset:
assumes BIJ: bij-betw f A A' and
\(\text{SUB: } B \leq A \land IM: f ' B = B'\)
shows bij-betw (inv-into A f) B' B
using assms unfolding bij-betw-def
by (auto intro: inj-on-inv-into)

26.11 Specification package – Hilbertized version

lemma exE-some: \(| Ex P ; c == Eps P | \Longrightarrow P c\)
by (simp only: someE-ex)

ML-file Tools/choice-specification.ML
THEORY "Zorn"

end

27 Zorn: Zorn’s Lemma

theory Zorn
imports Order-Relation Hilbert-Choice
begin

27.1 Zorn’s Lemma for the Subset Relation

27.1.1 Results that do not require an order

Let $P$ be a binary predicate on the set $A$.

locale pred-on =
  fixes $A$ :: 'a set
  and $P$ :: 'a ⇒ 'a ⇒ bool (infix ⊏ 50)

begin

abbreviation Peq :: 'a ⇒ 'a ⇒ bool (infix ⊑ 50)
  where $x$ ⊑ $y$ ≡ $P == x y$

A chain is a totally ordered subset of $A$.

definition chain :: 'a set ⇒ bool where
  chain C ←→ C ⊆ A ∧ (∀ x y. x ∈ C ∧ y ∈ C. x ⊑ y ∨ y ⊑ x)

We call a chain that is a proper superset of some set $X$, but not necessarily a chain itself, a superchain of $X$.

abbreviation superchain :: 'a set ⇒ 'a set ⇒ bool (infix <c 50)
  where $X <c C$ ≡ chain C ∧ $X ⊂ C$

A maximal chain is a chain that does not have a superchain.

definition maxchain :: 'a set ⇒ bool where
  maxchain C ←→ chain C ∧ ¬ (∃ S. C <c S)

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

definition suc :: 'a set ⇒ 'a set where
  suc C = (if ¬ chain C ∨ maxchain C then C else (SOME D. C <c D))

lemma chainI [Pure.intro?]:
  [C ⊆ A; ∀ x y. [x ∈ C; y ∈ C] ⇒ x ⊑ y ∨ y ⊑ x] ⇒ chain C
  unfolding chain-def by blast

lemma chain-total:
  chain C ⇒ x ∈ C ⇒ y ∈ C ⇒ x ⊆ y ∨ y ⊆ x
  by (simp add: chain-def)
lemma not-chain-suc [simp]: \( \neg \) chain \( X \Longrightarrow \) suc \( X = X \)
by (simp add: suc-def)

lemma maxchain-suc [simp]: maxchain \( X \Longrightarrow \) suc \( X = X \)
by (simp add: suc-def)

lemma suc-subset: \( X \subseteq \) suc \( X \)
by (auto simp: suc-def maxchain-def intro: someI2)

lemma chain-empty [simp]: chain \{\}\nby (auto simp: chain-def)

lemma not-maxchain-Some:
chain \( C \) \( \Longrightarrow \neg \) maxchain \( C \) \( \Longrightarrow \) suc \( C \neq C \)
using not-maxchain-Some by (auto simp: suc-def)

lemma subset-suc:
assumes \( X \subseteq Y \) shows \( X \subseteq \) suc \( Y \)
using assms by (rule subset-trans) (rule suc-subset)

We build a set \( C \) that is closed under applications of suc and contains the union of all its subsets.

inductive-set suc-Union-closed \( (C) \) where
suc: \( X \in C \Longrightarrow suc \ X \in C \mid \)
Union [unfolded Pow-iff]: \( X \in Pow \ C \Longrightarrow X \in C \)

Since the empty set as well as the set itself is a subset of every set, \( C \) contains at least \( \{\} \in C \) and \( \bigcup C \in C \).

lemma
suc-Union-closed-empty: \( \{\} \in C \) and
suc-Union-closed-Union: \( \bigcup C \in C \)
using Union [of \{\}] and Union [of \( C \)] by simp+

Thus closure under suc will hit a maximal chain eventually, as is shown below.

lemma suc-Union-closed-induct [consumes 1, case-names suc Union, induct pred: suc-Union-closed]:
assumes \( X \in C \)
and \( \bigwedge X. [X \in C; Q \ X] \Longrightarrow Q \ (suc \ X) \)
and \( \bigwedge X. [X \subseteq C; \forall x \in X. Q \ x] \Longrightarrow Q \ (\bigcup X) \)
shows \( Q \ X \)
using assms by (induct) blast+

lemma suc-Union-closed-cases [consumes 1, case-names suc Union,
cases pred; suc-Union-closed]:
assumes $X \in C$
and $\forall Y. \left[ X = \text{suc} \ Y; Y \in C \right] \Rightarrow Q$
and $\forall Y. \left[ X = \bigcup Y; Y \subseteq C \right] \Rightarrow Q$
shows $Q$
using assms by (cases) simp+

On chains, suc yields a chain.

lemma chain-suc:
assumes chain $X$ shows chain (suc $X$)
using assms
by (cases ¬ chain $X$ ∨ maxchain $X$)
(force simp: suc-def dest: not-maxchain-Some)+

lemma chain-sucD:
assumes chain $X$ shows suc $X$ ⊆ $A$ ∧ chain (suc $X$)
proof (induct)
case (suc $X$)
with ∗ show ?case by (blast del: subsetI intro: subset-suc)
qed blast

lemma suc-Union-closed-total′:
assumes $X \in C$ and $Y \in C$
and ∗: $\forall Z. \left[ Z \in C \Rightarrow Z \subseteq Y \Rightarrow Z = Y \lor \text{suc} Z \subseteq Y \right]$
shows $X \subseteq Y \lor \text{suc} Y \subseteq X$
using ⟨$X \in C$⟩
proof (induct)
case (suc $X$)
with ∗ show ?case by (blast del: subsetI intro: subset-suc)
qed blast

lemma suc-Union-closed-subsetD:
assumes $Y \subseteq X$ and $X \in C$ and $Y \in C$
shows $X = Y \lor \text{suc} Y \subseteq X$
using assms(2−, 1)
proof (induct arbitrary: $Y$)
case (suc $X$)
note ∗ = ($\forall Y. \left[ Y \in C; Y \subseteq X \right] \Rightarrow X = Y \lor \text{suc} Y \subseteq X$)
with suc-Union-closed-total′[OF ⟨$Y \in C$⟩ ⟨$X \in C$⟩]
have $Y \subseteq X \lor \text{suc} X \subseteq Y$ by blast
then show ?case
proof
assume $Y \subseteq X$
with ∗ and ⟨$Y \in C$⟩ have $X = Y \lor \text{suc} Y \subseteq X$ by blast
then show ?thesis
proof
assume $X = Y$ then show ?thesis by simp
next
THEORY “Zorn”

```
assume suc Y ⊆ X
then have suc Y ⊆ suc X by (rule subset-suc)
then show ?thesis by simp
qed
next
assume suc X ⊆ Y
with ⟨Y ⊆ suc X⟩ show ?thesis by blast
qed
next
case (Union X)
show ?case
proof (rule ccontr)
assume ¬ ?thesis
with ⟨Y ⊆ ∪X⟩ obtain x y z
where ¬ suc Y ⊆ ∪X
and x ∈ X and y ∈ x and y ⊈ Y
and z ∈ suc Y and ∀x∈X. z ⊈ x by blast
with ⟨X ⊆ C⟩ have x ∈ C by blast
from Union and ⟨x ∈ X⟩
have *: ∃y. [y ∈ C; y ⊆ x] ===> x = y ∨ suc y ⊆ x by blast
with suc-Union-closed-total′ [OF ⟨Y ∈ C; ⟨x ∈ C⟩] have Y ⊆ x ∨ suc x ⊆ Y by blast
then show False
proof
assume Y ⊆ x
with * [OF ⟨Y ∈ C⟩] have x = Y ∨ suc Y ⊆ x by blast
then show False
proof
assume x = Y with ⟨y ∈ x⟩ and ⟨y ⊈ Y⟩ show False by blast
next
assume suc Y ⊆ x
with ⟨x ∈ X⟩ have suc Y ⊆ ∪X by blast
with ⟨¬ suc Y ⊆ ∪X⟩ show False by contradiction
qed
next
assume suc x ⊆ Y
moreover from suc-subset and ⟨y ∈ x⟩ have y ∈ suc x by blast
ultimately show False using ⟨y ⊈ Y⟩ by blast
qed
qed
qed
```

The elements of C are totally ordered by the subset relation.

```
lemma suc-Union-closed-total:
assumes X ∈ C and Y ∈ C
shows X ⊆ Y ∨ Y ⊆ X
proof (cases ∀Z∈C. Z ⊆ Y ===> Z = Y ∨ suc Z ⊆ Y)
case True
with suc-Union-closed-total′ [OF assms]
```
have }X \subseteq Y \lor suc \ Y \subseteq X\text{ by } blast
then show } \neg \text{thesis using suc-subset [of } Y\text{] by } blast
next
case } False
then obtain }Z \text{ where } Z \in \mathcal{C} \text{ and } Z \subseteq Y \text{ and } Z \neq Y \text{ and } \neg \text{suc } Z \subseteq Y\text{ by } blast
with suc-Union-closed-subsetD and } Y \in \mathcal{C}\text{ show } \neg \text{thesis by } blast
qed

Once we hit a fixed point w.r.t. suc, all other elements of } \mathcal{C}\text{ are subsets of this fixed point.

lemma suc-Union-closed-suc:
\text{assumes } X \in \mathcal{C} \text{ and } Y \in \mathcal{C} \text{ and } suc \ Y = Y
\text{shows } X \subseteq Y
\text{using } \langle \text{X } \in \mathcal{C}\rangle
\text{proof (induct)}
case } (suc \ X)
with } Y \in \mathcal{C}\text{ and suc-Union-closed-subsetD}
\text{have } X = Y \lor suc \ X \subseteq Y\text{ by } blast
then show } \neg \text{case by } (\text{auto simp: (suc } Y = Y))
qed blast

lemma eq-suc-Union:
\text{assumes } X \in \mathcal{C}
\text{shows suc } X = X \iff X = \bigcup \mathcal{C}
\text{proof}
\text{assume suc } X = X
\text{with suc-Union-closed-suc [of suc-Union-closed-Union } \langle X \in \mathcal{C}\rangle]\text{[OF suc-Union-closed-Union]\[X \in \mathcal{C}\]}\text{]
\text{have } \bigcup \mathcal{C} \subseteq X\text{ .}
\text{with } \langle X \in \mathcal{C}\rangle \text{ show } X = \bigcup \mathcal{C}\text{ by } blast
next
from } \langle X \in \mathcal{C}\rangle \text{ have suc } X \in \mathcal{C}\text{ by } (\text{rule suc})
then have } suc \ X \subseteq \bigcup \mathcal{C}\text{ by } blast
moreover assume } X = \bigcup \mathcal{C}\text{ ultimately have suc } X \subseteq X\text{ by simp}
moreover have } X \subseteq suc \ X\text{ by } (\text{rule suc-subset})
ultimately show suc } X = X\text{ .}
qed

lemma suc-in-carrier:
\text{assumes } X \subseteq A
\text{shows suc } X \subseteq A
\text{using } assms
\text{by } (\text{cases } \neg \text{chain } X \lor \text{maxchain } X)
\text{ (auto dest: chain-sucD)}

lemma suc-Union-closed-in-carrier:
\text{assumes } X \in \mathcal{C}
\text{shows } X \subseteq A
using assms
by (induct) (auto dest: suc-in-carrier)

All elements of $C$ are chains.

**lemma** suc-Union-closed-chain:
assumes $X \in C$
shows $\text{chain } X$
using assms
proof (induct)
case (suc $X$) then show ?case using not-maxchain-Some by (simp add: suc-def)
next
case (Union $X$) then have $\bigcup X \subseteq A$ by (auto dest: suc-Union-closed-in-carrier)
moreover have $\forall x \in \bigcup X. \forall y \in \bigcup X. x \subseteq y \lor y \subseteq x$
proof (intro ballI)
fix $x$ $y$
assume $x \in \bigcup X$ and $y \in \bigcup X$
then obtain $u$ $v$ where $x \in u$ and $u \in X$ and $y \in v$ and $v \in X$ by blast
with Union have $u \in C$ and $v \in C$ and $\text{chain } u$ and $\text{chain } v$ by blast+
with suc-Union-closed-total have $u \subseteq v \lor v \subseteq u$ by blast
then show $x \subseteq y \lor y \subseteq x$
proof
assume $u \subseteq v$
from $\langle \text{chain } v \rangle$ show ?thesis
proof (rule chain-total)
  show $y \in v$ by fact
  show $x \in v$ using $\langle u \subseteq v \rangle$ and $\langle x \in u \rangle$ by blast
qed
next
assume $v \subseteq u$
from $\langle \text{chain } u \rangle$ show ?thesis
proof (rule chain-total)
  show $x \in u$ by fact
  show $y \in u$ using $\langle v \subseteq u \rangle$ and $\langle y \in v \rangle$ by blast
qed
qed
qed
ultimately show ?case unfolding chain-def ..
qed

### 27.1.2 Hausdorff’s Maximum Principle

There exists a maximal totally ordered subset of $A$. (Note that we do not require $A$ to be partially ordered.)

**theorem** Hausdorff: $\exists C. \text{maxchain } C$
proof
  let $?M = \bigcup C$
  have maxchain $?M"
proof (rule ccontr)
  assume ¬ maxchain ?M
  then have suc ?M ≠ ?M
    using suc-not-equals and
    suc-Union-closed-chain [OF suc-Union-closed-Union] by simp
  moreover have suc ?M = ?M
    using eq-suc-Union [OF suc-Union-closed-Union] by simp
  ultimately show False by contradiction
qed
then show ?thesis by blast
qed

Make notation C available again.

no-notation suc-Union-closed (C)

lemma chain-extend:
  chain C ⟹ z ∈ A ⟹ ∀ x ∈ C. x ⊆ z ⟹ chain ( {z} ∪ C)
  unfolding chain-def by blast

lemma maxchain-imp-chain:
  maxchain C ⟹ chain C
  by (simp add: maxchain-def)

end

Hide constant pred-on.suc-Union-closed, which was just needed for the proof
of Hausforff’s maximum principle.

hide-const pred-on.suc-Union-closed

lemma chain-mono:
  assumes ∀ x y. [ x ∈ A; y ∈ A; P x y] ⟹ Q x y
  and pred-on.chain A P C
  shows pred-on.chain A Q C
  using assms unfolding pred-on.chain-def by blast

27.1.3 Results for the proper subset relation

interpretation subset: pred-on A op ⊂ for A.

lemma subset-maxchain-max:
  assumes subset.maxchain A C and X ∈ A and ∪ C ⊆ X
  shows ∪ C = X
  proof (rule ccontr)
    let ☰C = {X} ∪ C
    from subset.maxchain A C have subset.chain A C
      and *: ∀ S. subset.chain A S ⟹ ¬ C ⊆ S
      by (auto simp: subset.maxchain-def)
    moreover have ∀ x ∈ C. x ⊆ X using ∪ C ⊆ X by auto
    ultimately have subset.chain A ☰C
  qed


using \texttt{subset.chain-extend [of }A \texttt{ }C \texttt{] and }\langle X \in A \rangle \texttt{ by auto}

moreover assume \( \ast \colon \bigcup C \neq X \)

moreover from \( \ast \) have \( C \subseteq \{ \textit{?} C \} \) using \( \bigcup C \subseteq X \) by auto

ultimately show \( \textit{False} \) using \( * \) by blast

\textbf{27.1.4 Zorn’s lemma}

If every chain has an upper bound, then there is a maximal set.

\textbf{lemma subset-Zorn:}

\begin{itemize}
  \item \texttt{assumes } \( \forall \langle C \rangle \texttt{. subset.chain }A \texttt{ }C \texttt{. }\bigcup U \subseteq A \texttt{. }\forall X \in C \texttt{. }X \subseteq U \)
  \item \texttt{shows } \( \exists M \in A \texttt{. }\forall X \in A \texttt{. }M \subseteq X \rightarrow X = M \)
\end{itemize}

\textbf{proof –}

\begin{itemize}
  \item from \texttt{subset.Hausdorff [of }A \texttt{]} \texttt{obtain } M \texttt{ where } \texttt{subset.maxchain }A \texttt{ }M \texttt{..}
  \item then have \texttt{subset.chain }A \texttt{ }M \texttt{ by (rule subset.maxchain-imp-chain)}
  \item with \texttt{assms }\texttt{obtain } Y \texttt{ where } Y \texttt{ in } A \texttt{ and } \forall X \in M \texttt{. }X \subseteq Y \texttt{ by blast}
  \item moreover have \( \forall X \in A \texttt{. }Y \subseteq X \rightarrow Y = X \)
  \item proof (\texttt{intro ballI impI})
    \begin{itemize}
      \item fix \( X \)
      \item assume \( X \in A \texttt{ and } Y \subseteq X \)
      \item show \( Y = X \)
      \item proof (\texttt{rule ccontr})
        \begin{itemize}
          \item assume \( Y \neq X \)
          \item with \( \langle Y \subseteq X \rangle \) have \( \neg X \subseteq Y \) by blast
        \end{itemize}
        \begin{itemize}
          \item from \texttt{subset.chain-extend [OF }\langle \texttt{subset.chain }A \texttt{ }M \texttt{) [OF }\langle X \in A \rangle \texttt{ and } \forall X \in M \texttt{. }X \subseteq Y \rangle \) \texttt{by auto}
          \item have \texttt{subset.chain }A \texttt{ }\langle \{ X \} \cup M \rangle \texttt{ using } \langle Y \subseteq X \rangle \texttt{ by auto}
          \item moreover have \( M \subseteq \{ X \} \cup M \texttt{ using } \forall X \in M \texttt{. }X \subseteq Y \) \texttt{ and } \( \neg X \subseteq Y \)
        \end{itemize}
        \begin{itemize}
          \item by \texttt{auto}
        \end{itemize}
        \begin{itemize}
          \item ultimately show \( \textit{False} \) using \( \texttt{subset.maxchain }A \texttt{ }M \texttt{) by (auto simp: subset.maxchain-def)}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}

qed

\textbf{Alternative version of Zorn’s lemma for the subset relation.}

\textbf{lemma subset-Zorn’:}

\begin{itemize}
  \item \texttt{assumes } \( \forall \langle C \rangle \texttt{. subset.chain }A \texttt{ }C \texttt{. }\bigcup C \subseteq A \)
  \item \texttt{shows } \( \exists M \in A \texttt{. }\forall X \in A \texttt{. }M \subseteq X \rightarrow X = M \)
\end{itemize}

\textbf{proof –}

\begin{itemize}
  \item from \texttt{subset.Hausdorff [of }A \texttt{]} \texttt{obtain } M \texttt{ where } \texttt{subset.maxchain }A \texttt{ }M \texttt{..}
  \item then have \texttt{subset.chain }A \texttt{ }M \texttt{ by (rule subset.maxchain-imp-chain)}
  \item with \texttt{assms }\texttt{have } \bigcup M \subseteq A \texttt{.}
  \item moreover have \( \forall Z \in A \texttt{. }\bigcup M \subseteq Z \rightarrow \bigcup M = Z \)
  \item proof (\texttt{intro ballII impI})
    \begin{itemize}
      \item fix \( Z \)
      \item assume \( Z \in A \texttt{ and } \bigcup M \subseteq Z \)
      \item with \texttt{subset-maxchain-max [OF }\langle \texttt{subset.maxchain }A \texttt{ }M \texttt{) [OF }\langle X \in A \rangle \texttt{..} \) 
    \end{itemize}
  \end{itemize}

}\textbf{qed}
show $\bigcup M = Z$.

qed

ultimately show thesis by blast

qed

27.2 Zorn’s Lemma for Partial Orders

Relate old to new definitions.

definition chain-subset :: 'a set set ⇒ bool (chain<_>) where
  chain<_> C ←→ (∀ A∈C. ∀ B∈C. A ⊆ B ∨ B ⊆ A)

definition chains :: 'a set set ⇒ 'a set set set where
  chains A = {C. C ⊆ A ∧ chain<_> C}

definition Chains :: ('a × 'a) set ⇒ 'a set set where
  Chains r = {C. ∀ a∈C. ∀ b∈C. (a, b) ∈ r ∨ (b, a) ∈ r}

lemma chains-extend:
  \[ c ∈ chains S; z ∈ S; ∀ x ∈ c. x ⊆ (z :: 'a set) \implies \{ z \} \subseteq c \] Un c ∈ chains S
  by (unfold chains-def chain-subset-def) blast

lemma mono-Chains: r ⊆ s ==> Chains r ⊆ Chains s
  unfolding Chains-def by blast

lemma chain-subset-alt-def: chain<_> C = subset.chain UNIV C
  unfolding chain-subset-def subset.chain-def by fast

lemma chains-alt-def: chains A = {C. subset.chain A C}
  by (simp add: chains-def chain-subset-alt-def subset.chain-def)

lemma Chains-subset:
  Chains r ⊆ {C. pred-on.chain UNIV (λ x y. (x, y) ∈ r) C}
  by (force simp: Chains-def pred-on.chain-def)

lemma Chains-subset':
  assumes refl r
  shows \{ C. pred-on.chain UNIV (λ x y. (x, y) ∈ r) C \} ⊆ Chains r
  using assms
  by (auto simp: Chains-def pred-on.chain-def refl-on-def)

lemma Chains-alt-def:
  assumes refl r
  shows Chains r = \{ C. pred-on.chain UNIV (λ x y. (x, y) ∈ r) C \}
  using assms Chains-subset Chains-subset' by blast

lemma Zorn-Lemma:
  ∀ C ∈ chains A. \bigcup C ∈ A → ∃ M ∈ A. ∀ X ∈ A. M ⊆ X → X = M
  using subset-Zorn' [of A] by (force simp: chains-alt-def)
lemma Zorn-Lemma2:
\[ \forall C \in \text{chains } A. \exists U \in A. \forall X \in C. X \subseteq U \implies \exists M \in A. \forall X \in A. M \subseteq X \implies X = M \]
using subset-Zorn [of A] by (auto simp: chains-alt-def)

Various other lemmas

lemma chainsD:\[ \forall c \in \text{chains } S; \quad x \in c; \quad y \in c \quad \implies \quad x \subseteq y \mid y \subseteq x \]
by (unfold chains-def chain-subset-def) blast

lemma chainsD2:\[ \forall c :: \text{a set set}. \quad e \in \text{chains } S \implies e \subseteq S \]
by (unfold chains-def) blast

lemma Zorns-po-lemma:
assumes po: Partial-order r
and u: \forall C \in \text{Chains } r. \exists u \in \text{Field } r. \forall a \in C. (a, u) \in r
shows \exists m \in \text{Field } r. \forall a \in \text{Field } r. (m, a) \in r \implies a = m
proof
have Preorder r using po by (simp add: partial-order-on-def)
— Mirror r in the set of subsets below (wrt r) elements of A
let \( \exists B = \%x. r^{-1} \{x\} \)
let \(?S = \exists B ' \text{ Field } r \)
\{ fix C assume 1: \( C \subseteq \exists S \) and 2: \( \forall A \subseteq C. \forall B \subseteq C. A \subseteq B \lor B \subseteq A \)
let \( \exists A = \{x \in \text{Field } r. \exists M \in C. M = \exists B x\} \)
have \( C = \exists B ' \exists A \) using 1 by (auto simp: image-def)
have \( \exists A \in \text{Chains } r \)
proof (simp add: Chains-def, intro allI impl, elim conjE)
fix a b
assume a \in Field r and \( \exists B a \in C \) and \( b \in \text{Field } r \) and \( \exists B b \in C \)
thus \( \exists B a \subseteq \exists B b \lor \exists B b \subseteq \exists B a \) using 2 by auto
using (Preorder r) and (a \in Field r) and (b \in Field r)
by (simp add:subset-Image1-Image1-iff)
qed
then obtain u where ua: \( u \in \text{Field } r \forall a \in \exists A. (a, u) \in r \) using u by auto
have \( \forall A \subseteq C. A \subseteq r^{-1} \{u\} \) (is \(?P u\) u)
proof auto
fix a B assume aB: \( B \in C a \in B \)
with 1 obtain x where x \in Field r and \( B = r^{-1} \{x\} \) by auto
thus (a, u) \in r using ua and aB and (Preorder r)
unfolding preorder-on-def refl-on-def by simp (fast dest: transD)
qed
then have \( \exists u \in \text{Field } r. \exists P u \) using \( u \in \text{Field } r \) by blast
\}
then have \( \forall C \in \text{chains } \exists S. \exists U \subseteq \exists S. \forall A \subseteq C. A \subseteq U \)
by (auto simp: chains-def chain-subset-def)
from Zorn-Lemma2 [OF this]
obtain m B where m \in Field r and \( B = r^{-1} \{m\} \)
and \( \forall x \in \text{Field } r. B \subseteq r^{-1} \{x\} \to r^{-1} \{x\} = B \)
THEORY "Zorn"

by auto 

hence \( \forall \, a \in \text{Field } r \) \( (m, a) \in r \rightarrow a = m \)

using po and \((\text{Preorder } r)\) and \((m \in \text{Field } r)\)

by (auto simp: subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff)

thus ?thesis using \((m \in \text{Field } r)\) by blast 

qed

27.3 The Well Ordering Theorem

definition \( \text{init-seg-of } :: (\{a \times a\} \times \{a \times a\}) \times (\{a \times a\}) \rightarrow\)

\( (\{a \times a\} \times \{a \times a\}) \rightarrow \) where

\( \text{init-seg-of } = \{ (r, s). \, r \subseteq s \land (\forall \, a \, b \, c. \, (a, b) \in s \land (b, c) \in r \rightarrow (a, b) \in r)\} \)

abbreviation \( \text{initialSegmentOf } :: (\{a \times a\} \rightarrow (\{a \times a\} \rightarrow \) bool where

\( r \) \( \text{initial-segment-of } s \equiv (r, s) \in \text{init-seg-of } \)

lemma refl-on-init-seg-of \[ simp \]: \( r \) \( \text{initial-segment-of } r \)

by (simp add: \text{init-seg-of-def})

lemma trans-init-seg-of:

\( r \) \( \text{initial-segment-of } s \rightarrow s \) \( \text{initial-segment-of } t \rightarrow r \) \( \text{initial-segment-of } t \)

by (simp (no-asn-use) add: \text{init-seg-of-def}) blast

lemma antisym-init-seg-of:

\( r \) \( \text{initial-segment-of } s \rightarrow s \) \( \text{initial-segment-of } r \rightarrow r = s \)

unfolding \text{init-seg-of-def} by safe

lemma Chains-init-seg-of-Union:

\( R \in \text{Chains init-seg-of } \rightarrow r \in R \rightarrow \) \( r \) \( \text{initial-segment-of } \bigcup R \)

by (auto simp: \text{init-seg-of-def Ball-def Chains-def}) blast

lemma chain-subset-trans-Union:

assumes \( \text{chain} \subseteq R \rightarrow r \in R. \) \( \text{trans } r \)

shows \( \text{trans } (\bigcup R) \)

proof (intro transI, elim UnionE)

fix \( S1 \, S2 :: \{a \times a\} \) \( \text{rel } x \, y \, z :: \{a \)

assume \( S1 \in R \, S2 \in R \)

with assms(1) have \( S1 \subseteq S2 \lor S2 \subseteq S1 \) unfolding chain-subset-def by blast

moreover assume \( (x, y) \in S1 \) \( (y, z) \in S2 \)

ultimately have \( ((x, y) \in S1 \land (y, z) \in S1) \lor ((x, y) \in S2 \land (y, z) \in S2) \) by blast

with \( S1 \in R \) \( S2 \in R \) assms(2) show \( (x, z) \in \bigcup R \) by (auto elim: transE)

qed

lemma chain-subset-antisym-Union:

assumes \( \text{chain} \subseteq R \rightarrow r \in R. \) \( \text{antisym } r \)

shows \( \text{antisym } (\bigcup R) \)
proof (intro antisymI, elim UnionE)
fix S1 S2 :: 'a rel and x y :: 'a
assume S1 ∈ R, S2 ∈ R
with assms(1) have S1 ⊆ S2 ∨ S2 ⊆ S1 unfolding chain-subset-def by blast
moreover assume ⟨x, y⟩ ∈ S1 (y, x) ∈ S2
ultimately have ((x, y) ∈ S1 ∧ (y, x) ∈ S2) ∨ ((x, y) ∈ S2 ∧ (y, x) ∈ S2)
by blast
with (S1 ∈ R) (S2 ∈ R) assms(2) show x = y unfolding antisym-def by auto
qed

lemma chain-subset-Total-Union:
assumes chain_⊆ R and ∀ r∈R. Total r
shows Total (⋃ R)
proof (simp add: total-on-def Ball-def, auto del: disjCI)
fix r s a b assume A: r ∈ R s ∈ R a ∈ Field r b ∈ Field s a ≠ b
from ⟨chain_⊆ R, and r∈R and s∈R have r ⊆ s ∨ s ⊆ r⟩
by (auto simp add: chain-subset-def)
thus (∃ r∈R. (a, b) ∈ r) ∨ (∃ r∈R. (b, a) ∈ r)
proof
assume r ⊆ s hence (a, b) ∈ s ∨ (b, a) ∈ s using assms(2) A mono-Field[of r s]
by (auto simp add: total-on-def)
thus ?thesis using s ∈ R by blast
next
assume s ⊆ r hence (a, b) ∈ r ∨ (b, a) ∈ r using assms(2) A mono-Field[of s r]
by (fastforce simp add: total-on-def)
thus ?thesis using r ∈ R by blast
qed

lemma wf-Union-wf-init-segs:
assumes R ∈ Chains init-seg-of and ∀ r∈R. wf r
shows wf (⋃ R)
proof (simp add: wf-iff-no-infinite-down-chain, rule ccontr, auto)
fix f assume 1: ∀ i. ∃ r∈R. (f (Suc i), f i) ∈ r
then obtain r where r ∈ R and (f (Suc 0), f 0) ∈ r by auto
{ fix i have (f (Suc i), f i) ∈ r
  proof (induct i)
  case 0 show ?case by fact
  next
  case (Suc i)
  then obtain s where s: s ∈ R (f (Suc i), f (Suc i)) ∈ s
  using 1 by auto
  then have s initial-segment-of r ∨ r initial-segment-of s
  using assms(1) (r ∈ R) by (simp add: Chains-def)
  with Suc s show ?case by (simp add: init-seg-of-def) blast
  qed
  }
Thus False using assms(2) and \( r \in R \)
by (simp add: wf_iff_no_infinite_down_chain) blast
qed

lemma initial-segment-of-Diff:
p initial-segment-of q \( \implies \) p \(-\) s initial-segment-of q \(-\) s
unfolding init-seg-of-def by blast

lemma Chains-init-seg-DiffI:
\( R \in \text{Chains init-seg-of} \implies \{ r \(-\) s \mid r, r \in R \} \in \text{Chains init-seg-of} \)
unfolding Chains-def by (blast intro: initial-segment-of-Diff)

theorem well-ordering: \( \exists r::\alpha \ \text{rel}. \ \text{Well-order } r \land \text{Field } r = \text{UNIV} \)
proof
— The initial segment relation on well-orders:
let \( ?WO = \{ r::\alpha \ \text{rel}. \ \text{Well-order } r \} \)
def I \( \equiv \) init-seg-of \( \cap \) ?WO \times ?WO
have I-init: \( I \subseteq \text{init-seg-of} \) by (auto simp: I-def)
hence subch: \( \bigwedge R. R \in \text{Chains } I \implies \text{chain } \subseteq R \)
unfolding init-seg-of-def chain-subset-def Chains-def by blast
have Chains-wo: \( \bigwedge R r. R \in \text{Chains } I \implies r \in R \implies \text{Well-order } r \)
by (simp add: Chains-def I-def)
hence 0: \( \text{Partial-order } I \)
by (auto simp: partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-on-def elim !: trans-init-seg-of)
— I-chains have upper bounds in ?WO wrt I: their Union
\{ fix R assume R \in Chains I \}
hence Ris: \( R \in \text{Chains init-seg-of} \) using mono-Chains [OF I-init] by blast
have subch: \( \bigwedge R. R \subseteq \text{Chains } I \) I-init
by (auto simp: init-seg-of-def chain-subset-def Chains-def)
have \( \forall r \in R. \ \text{Refl } r \land \forall r \in R. \ \text{trans } r \land \forall r \in R. \ \text{antisym } r \)
and \( \forall r \in R. \ \text{Total } r \land \forall r \in R. \ \text{wf } (r - \text{Id}) \)
using Chains-wo [OF \( R \in \text{Chains } I \)] by (simp-all add: order-on-defs)
have \( \text{Refl } (\bigcup R) \) using \( \forall r \in R. \ \text{Refl } r \) unfolding refl-on-def by fastforce
moreover have \( \text{trans } (\bigcup R) \)
by (rule chain-subset-trans-Union [OF subch \( \forall r \in R. \ \text{trans } r \)])[OF Ris]
moreover have \( \text{antisym } (\bigcup R) \)
by (rule chain-subset-antisym-Union [OF subch \( \forall r \in R. \ \text{antisym } r \)])[OF Ris]
moreover have \( \text{Total } (\bigcup R) \)
by (rule chain-subset-Total-Union [OF subch \( \forall r \in R. \ \text{Total } r \)])[OF Ris]
moreover have \( \text{wf } (\bigcup R - \text{Id}) \)
proof
have \( (\bigcup R - \text{Id}) = \bigcup \{ r - \text{Id} \mid r, r \in R \} \) by blast
with \( \forall r \in R. \ \text{wf } (r - \text{Id}) \) and wf-Union-wf-init-segs [OF Chains-init-seg-DiffI]
[OF Ris] show \( ?\text{thesis} \) by fastforce
qed
ultimately have Well-order (\( \bigcup R \)) by (simp add: order-on-defs)
moreover have \( \forall r \in R. \ r \) initial-segment-of \( \bigcup R \) using Ris
by (simp add: Chains-init-seg-of-Union)
ultimately have \( \bigcup R \in \?WO \land (\forall r \in R. \ (r, \bigcup R) \in I) \)
using mono-Chains [OF I-init] Chains-wo[of R] and \( :R \in \text{Chains I} \)
unfolding I-def by blast

\}

hence \( \forall R \in \text{Chains I}. \ \exists u \in \text{Field I}. \ \forall r \in R. \ (r, u) \in I \) by (subst FI) blast
— Zorn’s Lemma yields a maximal well-order \( m \):
then obtain \( m::\'a \) rel where Well-order \( m \) and
max: \( \forall r. \ \text{Well-order } r \land (m, r) \in I \rightarrow r = m \)
using Zorns-po-lemma [OF 0 1] unfolding FI by fastforce
— Now show by contradiction that \( m \) covers the whole type:
\{ fix \( x::\'a \) assume \( x \notin \text{Field } m \)
— We assume that \( x \) is not covered and extend \( m \) at the top with \( x \)
have \( m \neq \{} \)
proof
assume \( m = \{} \)
moreover have Well-order \( \{(x, x)\} \)
by (simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def)
ultimately show False using max
by (auto simp: I-def init-seg-of-def simp del: Field-insert)
qed

— The extension of \( m \) by \( x \):
let \( ?s = \{(a, x) \mid a. \ a \in \text{Field } m\} \)
let \( ?m = \text{insert } (x, x) \text{ m } \cup ?s \)
have \( \text{Fm: Field } ?m = \text{insert } x \text{ (Field m)} \)
by (auto simp: Field-def)
have Refl \( m \text{ and trans } m \text{ and antisym } m \text{ and Total } m \text{ and wf } (m - \text{Id}) \)
using (Well-order m) by (simp-all add: order-on-defs)
— We show that the extension is a well-order
have Refl ?m using (Refl m) Fm unfolding refl-on-def by blast
moreover have trans ?m using (trans m) and \( (x \notin \text{Field m}) \)
unfolding trans-def Field-def by blast
moreover have antisym ?m using (antisym m) and \( (x \notin \text{Field m}) \)
unfolding antisym-def Field-def by blast
moreover have Total ?m using (Total m) and Fm by (auto simp: total-on-def)
moreover have \( \text{wf } (?m \text{ - Id}) \)
proof —
have \( \text{wf } ?s \text{ using } (x \notin \text{Field m}) \) unfolding wf-eq-minimal Field-def
by (auto simp: Bex-def)
thus \( \text{thesis using } \text{wf } (m - \text{Id}) \text{ and } (x \notin \text{Field m}) \)
wf-subset [OF \( (\text{wf } ?s) \text{ Diff-subset} \)]
unfolding Un-Diff Field-def by (auto intro: wf-Un)
qed
ultimately have Well-order \( \forall m \) by (simp add: order-on-defs)

— We show that the extension is above \( m \)

moreover have \( (m, \forall m) \in I \) using (Well-order \( \forall m \)) and (Well-order \( m \)) and 
\( \langle x \notin Field \ m \rangle \)

by (fastforce simp: I-def init-seg-of-def Field-def)

ultimately

— This contradicts maximality of \( m \):

have False using max and \( \langle x \notin Field \ m \rangle \) unfolding Field-def by blast
}

hence Field \( m = UNIV \) by auto

with (Well-order \( m \)) show \( ?thesis \) by blast

qed

corollary well-order-on: \( \exists \langle \forall x :: a \ rel. \ well-order-on \ A r \rangle \)

proof —

obtain \( \langle \forall x :: a \ rel. \ well-order-on \ A r \rangle \) using well-ordering [where \( 'a = 'a \)] by blast

let \( \forall r = \{ \langle x, y \rangle. \ x \in A \land y \in A \land (x, y) \in r \} \)

have 1: Field \( \forall r = A \) using \( \forall \) univ

by (fastforce simp: Field-def order-on-defs refl-on-def)

have Refl \( \forall r \) and trans \( \forall r \) and antisym \( \forall r \) and Total \( \forall r \) and wf \( (r - \Id) \)

using (Well-order \( \forall r \)) by (simp-all add: order-on-defs)

have Refl \( \forall r \) using (Refl \( \forall r \)) by (auto simp: refl-on-def 1 univ)

moreover have trans \( \forall r \) using (trans \( r \))

unfolding trans-def by blast

moreover have antisym \( \forall r \) using (antisym \( \forall r \))

unfolding antisym-def by blast

moreover have Total \( \forall r \) using (Total \( r \)) by (simp add: total-on-def 1 univ)

moreover have wf \( (\forall r - \Id) \) by (rule wf-subset [OF \( (\forall r - \Id) \)]) blast

ultimately have Well-order \( \forall r \) by (simp add: order-on-defs)

with 1 show \( ?thesis \) by auto

qed

end

28 BNF-Wellorder-Relation: Well-Order Relations as Needed by Bounded Natural Functors

theory BNF-Wellorder-Relation
imports Order-Relation
begin

In this section, we develop basic concepts and results pertaining to well-order relations. Note that we consider well-order relations as non-strict relations, i.e., as containing the diagonals of their fields.

locale wo-rel =
  fixes \( r :: 'a \ rel \)

assumes WELL: Well-order \( r \)
begin

The following context encompasses all this section. In other words, for the whole section, we consider a fixed well-order relation $r$.

**abbreviation** under where $\text{under} \equiv \text{Order-Relation.under } r$

**abbreviation** underS where $\text{underS} \equiv \text{Order-Relation.underS } r$

**abbreviation** Under where $\text{Under} \equiv \text{Order-Relation.Under } r$

**abbreviation** UnderS where $\text{UnderS} \equiv \text{Order-Relation.UnderS } r$

**abbreviation** above where $\text{above} \equiv \text{Order-Relation.above } r$

**abbreviation** aboveS where $\text{aboveS} \equiv \text{Order-Relation.aboveS } r$

**abbreviation** ofilter where $\text{ofilter} \equiv \text{Order-Relation.ofilter } r$

**lemmas** ofilter-def $= \text{Order-Relation.ofilter-def}[of r]$

### 28.1 Auxiliaries

**lemma** REFL: $\text{Refl } r$

using WELL order-on-defs[of - r] by auto

**lemma** TRANS: $\text{trans } r$

using WELL order-on-defs[of - r] by auto

**lemma** ANTISYM: $\text{antisym } r$

using WELL order-on-defs[of - r] by auto

**lemma** TOTAL: $\text{Total } r$

using WELL order-on-defs[of - r] by auto

**lemma** TOTALS: $\forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \lor (b,a) \in r$

using REFL TOTAL refl-on-def[of - r] total-on-def[of - r] by force

**lemma** LIN: $\text{Linear-order } r$

using WELL well-order-on-def[of - r] by auto

**lemma** WF: $\text{wf } (r - \text{Id})$

using WELL well-order-on-def[of - r] by auto

**lemma** cases-Total:

$\bigwedge \phi a b. \left[\{a,b\} \leq \text{Field } r; ((a,b) \in r \implies \phi a b); ((b,a) \in r \implies \phi a b)\right] \implies \phi a b$

using TOTALS by auto

**lemma** cases-Total3:

$\bigwedge \phi a b. \left[\{a,b\} \leq \text{Field } r; ((a,b) \in r - \text{Id} \lor (b,a) \in r - \text{Id} \implies \phi a b); (a = b \implies \phi a b)\right] \implies \phi a b$

using TOTALS by auto
28.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to non-strict well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

**Lemma** well-order-induct:

**Assumes** IND: \( \forall x. \forall y. y \neq x \land (y, x) \in r \rightarrow y \not\rightarrow P y \rightarrow P x \)

**Shows** \( P a \)

**Proof**
- **Have** \( \forall x. \forall y. (y, x) \in r - Id \rightarrow y \not\rightarrow P y \rightarrow P x \)
- **Using** IND by blast
- **Thus** \( P a \) **Using** WF wf-induct[of \( r - Id \) \( P a \)] by blast

**Qed**

**Definition** worec :: \( ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \)

**Where**

\( \text{worec } F \equiv \text{wfrec} \ (r - Id) \ F \)

**Definition** adm-wo :: \( ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \)

**Where**

\( \text{adm-wo } H \equiv \forall f g x. (\forall y \in \text{underS } x. f y = g y) \rightarrow H f x = H g x \)

**Lemma** worec-fixpoint:

**Assumes** ADM: adm-wo H

**Shows** \( \text{worec } H = H \ (\text{worec } H) \)

**Proof**
- **Let** \( ?rS = r - Id \)
- **Have** adm-wf (r - Id) H
- **Unfolding** adm-wf-def
- **Using** ADM adm-wo-def[of H] underS-def[of r] by auto
- **Hence** \( \text{wfrec } ?rS H = H \ (\text{wfrec } ?rS H) \)
- **Using** WF wfrec-firpoint[of \( ?rS H \)] by simp
- **Thus** \(?thesis unfolding worec-def\).

**Qed**

28.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor of a set, and not of an element (the latter is of course a particular case). Also, we define the maximum of two elements, \( \text{max}2 \), and the minimum of a set, \( \text{minim} \) – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator \( \text{isMinim} \). Then, supremum and
successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as “initial segments”.

**definition** max2 :: \( 'a \Rightarrow 'a \Rightarrow 'a \)

**where**

\[
\text{max2} \ a \ b \equiv \text{if } (a,b) \in r \text{ then } b \text{ else } a
\]

**definition** isMinim :: \( 'a \text{ set} \Rightarrow 'a \Rightarrow \text{bool} \)

**where**

\[
\text{isMinim} \ A \ b \equiv b \in A \land (\forall a \in A. (b,a) \in r)
\]

**definition** minim :: \( 'a \text{ set} \Rightarrow 'a \)

**where**

\[
\text{minim} \ A \equiv \text{THE } b. \text{isMinim} \ A \ b
\]

**definition** supr :: \( 'a \text{ set} \Rightarrow 'a \)

**where**

\[
\text{supr} \ A \equiv \text{minim} \ (\text{Above} \ A)
\]

**definition** suc :: \( 'a \text{ set} \Rightarrow 'a \)

**where**

\[
\text{suc} \ A \equiv \text{minim} \ (\text{AboveS} \ A)
\]

### 28.3.1 Properties of max2

**lemma** max2-greater-among:

**assumes** \( a \in \text{Field} \ r \text{ and } b \in \text{Field} \ r \)

**shows** \( (a, \text{max2} \ a \ b) \in r \land (b, \text{max2} \ a \ b) \in r \land \text{max2} \ a \ b \in \{a,b\} \)

**proof**

\[
\begin{align*}
\{&\text{assume } (a,b) \in r \\
&\text{hence } \text{thesis using } \text{max2-def assms REFL refl-on-def} \\
&\text{by (auto simp add: refl-on-def)}
\} \\
&\text{moreover} \\
&\{\text{assume } a = b \\
&\text{hence } (a,b) \in r \text{ using REFL assms} \\
&\text{by (auto simp add: refl-on-def)}
\} \\
&\text{moreover} \\
&\{\text{assume } *: a \neq b \land (b,a) \in r \\
&\text{hence } (a,b) \notin r \text{ using ANTISYM} \\
&\text{by (auto simp add: antisym-def)} \\
&\text{hence } \text{thesis using } * \text{ max2-def assms REFL refl-on-def} \\
&\text{by (auto simp add: refl-on-def)}
\}
\]

**ultimately show** \( \text{thesis using assms TOTAL total-on-def[of Field r r]} \text{ by blast} \)

qed

**lemma** max2-greater:

**assumes** \( a \in \text{Field} \ r \text{ and } b \in \text{Field} \ r \)

**shows** \( (a, \text{max2} \ a \ b) \in r \land (b, \text{max2} \ a \ b) \in r \)

**using** assms by (auto simp add: max2-greater-among)
lemma max2-among:
assumes $a \in \text{Field } r$ and $b \in \text{Field } r$
shows $\text{max2 } a \ b \in \{a, b\}$
using assms max2-greater-among[of $a \ b$] by simp

lemma max2-equals1:
assumes $a \in \text{Field } r$ and $b \in \text{Field } r$
shows $(\text{max2 } a \ b = a) = ((a, b) \in r)$
using assms ANTISYM unfolding antisym-def using TOTALS
by (auto simp add: max2-def max2-among)

lemma max2-equals2:
assumes $a \in \text{Field } r$ and $b \in \text{Field } r$
shows $(\text{max2 } a \ b = b) = ((a, b) \in r)$
using assms ANTISYM unfolding antisym-def using TOTALS
unfolding max2-def by auto

28.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma isMinim-unique:
assumes MINIM: $\text{isMinim } B \ a$ and MINIM’: $\text{isMinim } B \ a'$
shows $a = a'$
proof –
\{have $a \in B$
  using MINIM isMinim-def by simp
  hence $(a', a) \in r$
  using MINIM’ isMinim-def by simp
\}
moreover
\{have $a' \in B$
  using MINIM’ isMinim-def by simp
  hence $(a, a') \in r$
  using MINIM isMinim-def by simp
\}
ultimately
show ?thesis using ANTISYM antisym-def[of $r$] by blast
qed

lemma Well-order-isMinim-exists:
assumes SUB: $B \leq \text{Field } r$ and NE: $B \neq \{\}$
shows $\exists b. \text{isMinim } B \ b$
proof –
from spec[OF WF[unfolded wf-eq-minimal[of $r - Id$]], of $B$] NE obtain $b$ where
\*: $b \in B \land (\forall b'. b' \neq b \land (b', b) \in r \rightarrow b' \notin B)$ by auto
show ?thesis
proof (simp add: isMinim-def, rule exI[of - $b$], auto)
  show $b \in B$ using * by simp
next
  fix \( b' \) assume \( As \): \( b' \in B \)
  hence **: \( b \in \text{Field } r \land b' \in \text{Field } r \) using \( As \) SUB * by auto

  from \( As \) * have \( b' = b \lor (b',b) \notin r \) by auto
  moreover
  \{ assume \( b' = b \)
    hence \( (b,b') \in r \)
    using ** REFL by (auto simp add: refl-on-def) \}
  moreover
  \{ assume \( b' \neq b \land (b',b) \notin r \)
    hence \( (b,b') \in r \)
    using ** TOTAL by (auto simp add: total-on-def) \}
  ultimately show \( (b,b') \in r \) by blast
qed

lemma minim-isMinim:
assumes SUB: \( B \leq \text{Field } r \) and NE: \( B \neq \{\} \)
shows isMinim \( B \) (minim \( B \))
proof
  let \( \phi = (\lambda b. \text{isMinim } B \ b) \)
  from assms Well-order-isMinim-exists
  obtain \( b \) where *: \( \phi b \) by blast
  moreover
  have \( \bigwedge b', \phi b' \implies b' = b \)
  using isMinim-unique * by auto
  ultimately show ?thesis
  unfolding minim-def using theI[of \phi b] by blast
qed

28.3.3 Properties of minim

lemma minim-in:
assumes \( B \leq \text{Field } r \) and \( B \neq \{\} \)
shows minim \( B \in B \)
proof
  from minim-isMinim[of \( B \)] assms
  have isMinim \( B \) (minim \( B \)) by simp
  thus ?thesis by (simp add: isMinim-def)
qed

lemma minim-inField:
assumes \( B \leq \text{Field } r \) and \( B \neq \{\} \)
shows minim \( B \in \text{Field } r \)
proof
  have minim \( B \in \text{Field } r \) using assms by (simp add: minim-in)
thus \(?thesis\) using assms by blast

\textbf{lemma} \textit{minim-least}:
\begin{description}
\item[assumes] \text{SUB}: \(B \subseteq \text{Field } r\) and \text{IN}: \(b \in B\)
\item[shows] \((\text{minim } B, b) \in r\)
\end{description}
\begin{proof}
\begin{enumerate}
\item from \text{minim-isMinim[of } B\text{] assms}
\item have \text{isMinim } B (\text{minim } B) \text{ by auto}
\end{enumerate}
thus \(?thesis\) by (auto simp add: isMinim-def IN)
\end{proof}

\textbf{lemma} \textit{equals-minim}:
\begin{description}
\item[assumes] \text{SUB}: \(B \subseteq \text{Field } r\) and \text{IN}: \(a \in B\) and
\item[LEAST] \(\bigwedge b. \, b \in B \Rightarrow (a, b) \in r\)
\item[shows] \(a = \text{minim } B\)
\end{description}
\begin{proof}
\begin{enumerate}
\item from \text{minim-isMinim[of } B\text{] assms}
\item have \text{isMinim } B (\text{minim } B) \text{ by auto}
\item moreover have \text{isMinim } B \text{ a using IN LEAST isMinim-def by auto}
\item ultimately show \(?thesis\) using \text{isMinim-unique by auto}
\end{enumerate}
\end{proof}

\textit{28.3.4 Properties of successor}

\textbf{lemma} \textit{suc-AboveS}:
\begin{description}
\item[assumes] \text{SUB}: \(B \subseteq \text{Field } r\) and \text{ABOVES}: \(\text{AboveS } B \neq \{\}\)
\item[shows] \(\text{suc } B \in \text{AboveS } B\)
\end{description}
\begin{proof}(unfold suc-def)
\begin{enumerate}
\item have \text{AboveS } B \subseteq \text{Field } r
\item using \text{AboveS-Field[of } r\text{] by auto}
\item thus \text{minim } (\text{AboveS } B) \in \text{AboveS } B
\item using \text{assms by (simp add: minim-in)}
\end{enumerate}
\end{proof}

\textbf{lemma} \textit{suc-greater}:
\begin{description}
\item[assumes] \text{SUB}: \(B \subseteq \text{Field } r\) and \text{ABOVES}: \(\text{AboveS } B \neq \{\}\) and 
\item[IN]: \(b \in B\)
\item[shows] \(\text{suc } B \neq b \land (b, \text{suc } B) \in r\)
\end{description}
\begin{proof}
\begin{enumerate}
\item from \text{assms suc-AboveS}
\item have \text{suc } B \in \text{AboveS } B \text{ by simp}
\item with \text{IN AboveS-def[of } r\text{]} show \(?thesis\) by simp
\end{enumerate}
\end{proof}

\textbf{lemma} \textit{suc-least-AboveS}:
\begin{description}
\item[assumes] \text{ABOVES}: \(a \in \text{AboveS } B\)
\item[shows] \((\text{suc } B, a) \in r\)
\end{description}
proof (unfold suc-def)
  have AboveS B ≤ Field r
  using AboveS-Field[of r] by auto
  thus (minim (AboveS B), a) ∈ r
  using assms minim-least by simp
qed

lemma suc-inField:
  assumes B ≤ Field r and AboveS B ≠ {}
  shows suc B ∈ Field r
proof -
  have suc B ∈ AboveS B using suc-AboveS assms by simp
  thus ?thesis
  using assms AboveS-Field[of r] by auto
qed

lemma equals-suc-AboveS:
  assumes SUB: B ≤ Field r and ABV: a ∈ AboveS B and
           MINIM: ⋀ a'. a' ∈ AboveS B =⇒ (a, a') ∈ r
  shows a = suc B
proof (unfold suc-def)
  have AboveS B ≤ Field r
  using AboveS-Field[of r B] by auto
  thus a = minim (AboveS B)
  using assms equals-minim
  by simp
qed

lemma suc-underS:
  assumes IN: a ∈ Field r
  shows a = suc (underS a)
proof -
  have underS a ≤ Field r
  using underS-Field[of r] by auto
  moreover
  have a ∈ AboveS (underS a)
  using in-AboveS-underS IN by fast
  moreover
  have ∀ a' ∈ AboveS (underS a). (a, a') ∈ r
  proof (clarify)
    fix a'
    assume #: a' ∈ AboveS (underS a)
    hence **: a' ∈ Field r
    using AboveS-Field by fast
    {assume (a, a') / ∈ r
     hence a' = a ∨ (a', a) ∈ r
     using TOTAL IN ** by (auto simp add: total-on-def)
     moreover
     {assume a' = a

THEORY “BNF-Wellorder-Relation”
hence \((a,a') \in r\)
using \(\text{REFL IN \(\ast\ast\)}\) by \((\text{auto simp add: refl-on-def})\)
}
moreover
\{assume \(a' \neq a \land (a',a) \in r\)
hence \(a' \in \text{underS \(a\)}\)
unfolding \(\text{underS-def by simp}\)
hence \(a' \notin \text{AboveS (underS \(a\)}\)
using \(\text{AboveS-disjoint by fast}\)
with \(*\) have \(\text{False by simp}\)
\}
ultimately have \((a,a') \in r\) by \(\text{blast}\)
}
thus \((a, a') \in r\) by \(\text{blast}\)
qed
ultimately show \(?\text{thesis}\)
using \(\text{equals-suc-AboveS by auto}\)
qed

28.3.5 Properties of order filters

lemma \(\text{under-ofilter}\):
ofilter \((\text{under \(a\)}\)
proof\((\text{unfold ofilter-def under-def, auto simp add: Field-def})\)
fix \(aa x\)
assume \((aa,a) \in r (x,aa) \in r\)
thus \((x,a) \in r\)
using \(\text{TRANS trans-def[of r] by blast}\)
qed

lemma \(\text{underS-ofilter}\):
ofilter \((\text{underS \(a\)}\)
proof\((\text{unfold ofilter-def underS-def under-def, auto simp add: Field-def})\)
fix \(aa x\)
assume \((a, aa) \in r (aa, a) \in r\) and \(\text{DIFF: \(aa \neq a\)}\)
thus \(\text{False}\)
using \(\text{ANTISYM antisym-def[of r] by blast}\)
next
fix \(aa x\)
assume \((aa,a) \in r aa \neq a (x,aa) \in r\)
thus \((x,a) \in r\)
using \(\text{TRANS trans-def[of r] by blast}\)
qed

lemma \(\text{Field-ofilter}\):
ofilter \((\text{Field \(r\)}\)
by\((\text{unfold ofilter-def under-def, auto simp add: Field-def})\)

lemma \(\text{ofilter-underS-Field}\):
ofilter \(A = ((\exists a \in \text{Field \(r\)}. A = \text{underS \(a\)} \land (A = \text{Field \(r\})))\)
proof
  assume (\exists a \in Field r. A = underS a) \lor A = Field r
  thus ofilter A
  by (auto simp: underS-ofilter Field-ofilter)
next
  assume \(*\): ofilter A
  let ?One = (\exists a \in Field r. A = underS a)
  let ?Two = (A = Field r)
  show ?One \lor ?Two
  proof (cases ?Two, simp)
    let ?B = (Field r) - A
    let ?a = minim ?B
    assume A \neq Field r
    moreover have A \leq Field r using \(*\) ofilter-def by simp
    ultimately have 1: ?B \neq {} by blast
    hence 2: ?a \in Field r using minim-inField[of ?B] by blast
    have 3: ?a \in ?B using minim-in[of ?B] 1 by blast
    hence 4: ?a \notin A by blast
    have 5: A \leq Field r using \(*\) ofilter-def by auto
    moreover have A = underS ?a
    proof
      show A \leq underS ?a
      proof (unfold underS-def, auto simp add: 4)
        fix x assume \(*\): x \in A
        hence 11: x \in Field r using 5 by auto
        have 12: x \neq ?a using 4 \(*\) by auto
        have 13: under x \leq A using \(*\) ofilter-def \(*\) by auto
        {assume (x,?a) \notin r
         hence (?a,x) \in r
         using TOTAL total-on-def[of Field r r]
         2 4 11 12 by auto
         hence ?a \in under x using under-def[of r] by auto
         hence ?a \in A using \(*\) 13 by blast
         with 4 have False by simp
        }
        thus (x,?a) \in r by blast
      qed
    qed
    next
    show underS ?a \leq A
    proof (unfold underS-def, auto)
      fix x
      assume \(*\): x \neq ?a and \(*\): (x,?a) \in r
      hence 11: x \in Field r using Field-def by fastforce
      {assume x \notin A
       hence x \notin ?B using 11 by auto
       hence (?a,x) \in r using \(*\) minim-least[of ?B x] by blast
       hence False
      }
      qed
    qed
  qed
qed
using ANTSYM antisym-def[of r] ** *** by auto
}
thus $x \in A$ by blast
qed
qed
ultimately have ?One using 2 by blast
thus ?thesis by simp
qed
qed

lemma ofilter-UNION:
($\bigwedge i. i \in I \Rightarrow ofilter(A i)) \Rightarrow ofilter (\bigcup i \in I. A i)$
unfolding ofilter-def by blast

lemma ofilter-under-UNION:
assumes ofilter A
shows $A = (\bigcup a \in A. \under a)$
proof
have $\forall a \in A. \under a \leq A$
using assms ofilter-def by auto
thus $(\bigcup a \in A. \under a) \leq A$ by blast
next
have $\forall a \in A. a \in \under a$
using REFL Refl-under-in[of r] assms ofilter-def[of A] by blast
thus $A \leq (\bigcup a \in A. \under a)$ by blast
qed

28.3.6 Other properties

lemma ofilter-linord:
assumes OF1: ofilter A and OF2: ofilter B
shows $A \leq B \lor B \leq A$
proof(cases $A = Field r$)
assume Case1: $A = Field r$
hence $B \leq A$ using OF2 ofilter-def by auto
thus ?thesis by simp
next
assume Case2: $A \neq Field r$
with ofilter-underS-Field OF1 obtain a where
1: $a \in Field r \land A = underS a$ by auto
show ?thesis
proof(cases $B = Field r$)
assume Case21: $B = Field r$
hence $A \leq B$ using OF1 ofilter-def by auto
thus ?thesis by simp
next
assume Case22: $B \neq Field r$
with ofilter-underS-Field OF2 obtain b where
2: $b \in Field r \land B = underS b$ by auto
have \( a = b \lor (a,b) \in r \lor (b,a) \in r \)
using \( I \ 2 \ TOTAL \ total-on-def[of - r] \) by auto
moreover
\{ assume \( a = b \) with \( I \ 2 \) have \( \text{thesis by auto} \) \}
moreover
\{ assume \( (a,b) \in r \) with \( \text{underS-incr[of r]} \) \( TRANS \ \text{ANTISYM} \ I \ 2 \) have \( A \leq B \) by auto
hence \( \text{thesis by auto} \) \}
moreover
\{ assume \( (b,a) \in r \) with \( \text{underS-incr[of r]} \) \( TRANS \ \text{ANTISYM} \ I \ 2 \) have \( B \leq A \) by auto
hence \( \text{thesis by auto} \) \}
ultimately show \( \text{thesis by blast} \) qed

lemma ofilter-AboveS-Field:
assumes ofilter \( A \)
shows \( A \cup (\text{AboveS} \ A) = \text{Field} \ r \)
proof
show \( A \cup (\text{AboveS} \ A) \leq \text{Field} \ r \)
using \( \text{assms ofilter-def AboveS-Field[of r]} \) by auto
next
\{ fix \( x \) assume \( *: x \in \text{Field} \ r \) and \( **: x \notin A \)
\{ fix \( y \) assume \( ***: y \in A \)
with \( ** \) have \( I: y \neq x \) by auto
\{ assume \( (y,x) \notin r \) moreover
have \( y \in \text{Field} \ r \) using \( \text{assms ofilter-def *** by auto} \)
ultimately have \( (x,y) \in r \)
using \( I * TOTAL \ total-on-def[of - r] \) by auto
with \( *** \) assms ofilter-def under-def[of r] have \( x \in A \) by auto
with \( ** \) have False by contradiction \}
hence \( (y,x) \in r \) by blast
with \( I \) have \( y \neq x \land (y,x) \in r \) by auto \}
with \( * \) have \( x \in \text{AboveS} \ A \) unfolding \( \text{AboveS-def} \) by auto \}
thus \( \text{Field} \ r \leq A \cup (\text{AboveS} \ A) \) by blast qed

lemma suc-ofilter-in:
assumes \( \text{OF}: \text{ofilter} \ A \) and \( \text{ABOVE-NE}: \text{AboveS} \ A \neq \{\} \) and
REL: \((b, \text{suc } A) \in r\) and DIFF: \(b \neq \text{suc } A\)

shows \(b \in A\)

proof –

have \(*\): \(\text{suc } A \in \text{Field } r \land b \in \text{Field } r\)
using WELL REL well-order-on-domain[of Field r] by auto
{assume \(*\): \(b \notin A\)
  hence \(b \in \text{AboveS } A\)
  using OF * ofilter-AboveS-Field by auto
  hence \((\text{suc } A, b) \in r\)
  using suc-least-AboveS by auto
  hence False using REL DIFF ANTISYM *
  by (auto simp add: antisym-def)
}
thus \(!?\text{thesis}\) by blast
qed

end

end

29 BNF-Wellorder-Embedding: Well-Order Embeddings as Needed by Bounded Natural Functors

theory BNF-Wellorder-Embedding
imports Hilbert-Choice BNF-Wellorder-Relation
begin

In this section, we introduce well-order embeddings and isomorphisms and prove their basic properties. The notion of embedding is considered from the point of view of the theory of ordinals, and therefore requires the source to be injected as an initial segment (i.e., order filter) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

29.1 Auxiliaries

lemma UNION-inj-on-ofilter:
assumes WELL: Well-order \(r\) and
\(\text{OF}: \bigwedge i. i \in I \implies \text{wo-rel.ofilter } r \ (A \ i)\) and
\(\text{INJ}: \bigwedge i. i \in I \implies \text{inj-on } f \ (A \ i)\)
shows inj-on \(f \ (\bigcup i. i \in I. A \ i)\)
proof–

have \(\text{wo-rel } r\) using WELL by (simp add: \text{wo-rel-def})
  hence \(\bigwedge i. j. \ll i; j \in I\] \implies A \ i \subseteq A \ j \lor A \ j \subseteq A \ i\)
  using wo-rel.ofilter-linord[of \(r\) \ OF by blast
with WELL INJ show ?thesis
  by (auto simp add: inj-on-UNION-chain)
qed

lemma under-underS-bij-betw:
  assumes WELL: Well-order r and WELL': Well-order r' and
    IN: a ∈ Field r and IN': f a ∈ Field r' and
    BIJ: bij-betw f (underS r a) (underS r' (f a))
  shows bij-betw f (under r a) (under r' (f a))
proof –
  have a /∈ underS r a ∧ f a /∈ underS r' (f a)
    unfolding underS-def by auto
  moreover
  { have Refl r ∧ Refl r' using WELL WELL'
      by (auto simp add: order-on-defs)
      hence under r a = underS r a ∪ {a} ∧
        under r' (f a) = underS r' (f a) ∪ {f a}
        using IN IN' by(auto simp add: Refl-under-underS)
  }
  ultimately show ?thesis
    using BIJ notIn-Un-bij-betw[of a underS r a f underS r' (f a)] by auto
qed

29.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator embed), asking that, for any element in the source, the function should be a bijection between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma embed-iff-compat-inj-on-ofilter.) A strict embedding (operator embedS) is a non-bijective embedding and an isomorphism (operator iso) is a bijective embedding.

definition embed :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
where
  embed r r' f ≡ ∀ a ∈ Field r. bij-betw f (under r a) (under r' (f a))

lemmas embed-defs = embed-def embed-def[abs-def]

Strict embeddings:
definition embedS :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
where
  embedS r r' f ≡ embed r r' f ∧ ¬ bij-betw f (Field r) (Field r')

lemmas embedS-defs = embedS-def embedS-def[abs-def]
definition iso :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
where
iso r r' f ≡ embed r r' f ∧ bij-betw f (Field r) (Field r')

lemmas iso-defs = iso-def iso-def[abs-def]

definition compat :: 'a rel ⇒ 'a' rel ⇒ ('a ⇒ 'a') ⇒ bool
where
compat r r' f ≡ ∀ a b. (a, b) ∈ r → (f a, f b) ∈ r'

lemma compat-wf:
assumes CMP: compat r r' f and WF: wf r'
shows wf r
proof
  have r ≤ inv-image r' f
    unfolding inv-image-def using CMP
    by (auto simp add: compat-def)
  with WF show ?thesis
    using wf-inv-image[of r' f] wf-subset[of inv-image r' f] by auto
qed

lemma id-embed: embed r r id
by(auto simp add: id-def embed-def bij-betw-def)

lemma id-iso: iso r r id
by(auto simp add: id-def embed-def iso-def bij-betw-def)

lemma embed-in-Field:
assumes WELL: Well-order r and
  EMB: embed r r' f and IN: a ∈ Field r
shows f a ∈ Field r'
proof
  have Well: wo-rel r
    using WELL by (auto simp add: wo-rel-def)
  hence 1: Refl r
    by (auto simp add: wo-rel.REFL)
  hence a ∈ under r a using IN Refl-under-in by fastforce
  hence f a ∈ under r' (f a)
  using EMB IN by (auto simp add: embed-def bij-betw-def)
  thus ?thesis unfolding Field-def
    by (auto simp: under-def)
qed

lemma comp-embed:
assumes WELL: Well-order r and
  EMB: embed r r' f and EMB': embed r' r'' f'
shows embed r r'' (f' o f)
proof(unfold embed-def, auto)
fix \( a \) assume \( *: a \in \text{Field } r \)
hence bij-betw \( f \) (under \( r \) \( a \)) (under \( r' \) \( (f \ a) \))
using embed-def[\( \text{of } r \)] EMB by auto
moreover
\{ have \( f \ a \in \text{Field } r' \)
  using EMB WELL \( * \) by (auto simp add: embed-in-Field)
  hence bij-betw \( f' \) (under \( r' \) \( (f \ a) \)) (under \( r'' \) \( (f' \ (f \ a)) \))
  using embed-def[\( \text{of } r' \)] EMB' by auto \}
ultimately
show bij-betw \( (f' \circ f) \) (under \( r \) \( a \)) (under \( r'' \) \( (f' \ (f \ a)) \))
by (auto simp add: bij-betw-trans)
qed

lemma comp-iso:
assumes WELL: Well-order \( r \) and
  \( \text{EMB}: \text{iso } r \ \ r' \ \ f \) and \( \text{EMB'}: \text{iso } r' \ \ r'' \ \ f' \)
shows iso \( r \ \ r'' \ (f' \circ f) \)
using assms unfolding iso-def
by (auto simp add: comp-embed bij-betw-trans)

That \( \text{embedS} \) is also preserved by function composition shall be proved only later.

lemma embed-Field:
\[ \left[ \text{Well-order } r; \text{embed } r \ \ r' \ \ f \right] \implies f'(\text{Field } r) \leq \text{Field } r' \]
by (auto simp add: embed-in-Field)

lemma embed-preserved-ofilter:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
  \( \text{EMB}: \text{embed } r \ \ r' \ \ f \) and \( \text{OF}: \text{wo-rel.ofilter } r \ A \)
shows \( \text{wo-rel.ofilter } r' \ (f' \ A) \)
proof –

from WELL have \( \text{Well: } \text{wo-rel } r \ \ \text{unfolding } \text{wo-rel-def} \).
from WELL' have \( \text{Well': } \text{wo-rel } r' \ \ \text{unfolding } \text{wo-rel-def} \).
from OF have \( 0: A \leq \text{Field } r \) by (auto simp add: Well \text{wo-rel.ofilter-def})

show \( \text{thesis} \) using \( \text{Well': } \text{WELL EMB } 0 \ \ \text{embed-Field[of } r \ \ r' \ f] \)
proof
(unfold wo-rel.ofilter-def, auto simp add: image-def)
  fix \( a \ b' \)
  assume \( *: a \in A \) and \( **: b' \in \text{under } r' \ (f \ a) \)
  hence \( a \in \text{Field } r \) using \( \emptyset \) by auto
  hence bij-betw \( f \) (under \( r \) \( a \)) (under \( r' \) \( (f \ a) \))
  using * EMB by (auto simp add: embed-def)
  hence \( f'(\text{under } r \ a) = \text{under } r' \ (f \ a) \)
  by (simp add: bij-betw-def)
  with \( ** \) image-def[\( \text{of } f \ \text{under } r \ a \)] obtain \( b \) where
  \( 1: b \in \text{under } r \ a \ \land \ b' = f \ b \) by blast
  hence \( b \in A \) using Well * OF
lemmas [global]
  embed-Field-ofilter:
  assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
E: embed \( r \) \( r' \) \( f \)
  shows wo-rel.ofilter \( r' \) \((f\of Field \( r \))\)
proof (\unf[compat-def, clarify)\)
  fix \( a \) \( b \)
  assume *: \((a, b) \in r\) \( \)
  hence \( 1: b \in Field \( r \) \) using Field-def[of \( r \)] by blast
  have \( a \in \under r b \) \( \)
  using * under-def[of \( r \)] by simp
  hence \( f a \in \under r' (f b) \) \( \)
  using E embed-def[of \( r \) \( r' \) \( f \)]
  bij-betw-def[of \( f \) under \( r \) \( b \) under \( r' \) \((f \ b)\)]
  image-def[of \( f \) under \( r \) \( b \)] \( 1 \) by auto
  thus \((f a, f b) \in r' \) \( \)
  by (auto simp add: under-def)
qed

lemma [global]
  embed-compat:
  assumes E: embed \( r \) \( r' \) \( f \)
  shows compat \( r \) \( r' \) \( f \)
proof (\unf, clarify)\)
  fix \( a \) \( b \)
  assume *: \((a, b) \in r\) \( \)
  hence \( 1: b \in Field \( r \) \) using Field-def[of \( r \)] by blast
  have \( a \in \under r b \) \( \)
  using * under-def[of \( r \)] by simp
  hence \( f a \in \under r' (f b) \) \( \)
  using E embed-def[of \( r \) \( r' \) \( f \)]
  bij-betw-def[of \( f \) under \( r \) \( b \) under \( r' \) \((f \ b)\)]
  image-def[of \( f \) under \( r \) \( b \)] \( 1 \) by auto
  thus \((f a, f b) \in r' \) \( \)
  by (auto simp add: under-def)
qed

lemma [global]
  embed-inj-on:
  assumes WELL: Well-order \( r \) and E: embed \( r \) \( r' \) \( f \)
  shows inj-on \( f \) (Field \( r \))
proof (\unf[inj-on-def, clarify)\)
  from WELL have Well: wo-rel \( r \) unrolling wo-rel-def.
  with wo-rel.TOTAL[of \( r \)]
  have Total: Total \( r \) by simp
  from Well wo-rel.REFL[of \( r \)]
  have Refl: Refl \( r \) by simp

  fix \( a \) \( b \)
  assume *: \( a \in Field \( r \) \) and **: \( b \in Field \( r \) \) and
  ***: \( f a = f b \)
  hence \( 1: a \in Field \( r \) \land b \in Field \( r \) \)
  unfolding Field-def by auto
{assume \((a,b) \in r\)
  hence \(a \in \text{under } r b \land b \in \text{under } r b\)
  using \(\text{Refl by (auto simp add: under-def refl-on-def)}\)
  hence \(a = b\)
  using \(\text{EMB 1 \(\cdot\\cdot\cdot\)}\)
  by (auto simp add: embed-def bij-betw-def inj-on-def)
}
moreover
{assume \((b,a) \in r\)
  hence \(a \in \text{under } r a \land b \in \text{under } r a\)
  using \(\text{Refl by (auto simp add: under-def refl-on-def)}\)
  hence \(a = b\)
  using \(\text{EMB 1 \(\cdot\cdot\cdot\)}\)
  by (auto simp add: embed-def bij-betw-def inj-on-def)
}
ultimately
show \(a = b\)
using \(\text{Total 1 \(\cdot\cdot\cdot\)}\)
by (auto simp add: total-on-def)
qed

lemma \(\text{embed-underS:}\)
assumes \(\text{WELL: Well-order } r \text{ \ and WELL': Well-order } r' \text{ \ and } EMB: \text{embed } r r' f \text{ \ and IN: } a \in \text{Field } r\)
shows \(\text{bij-betw } f \ (\text{underS } r a) \ (\text{underS } r' (f a))\)
proof
  have \(\text{bij-betw } f \ (\text{under } r a) \ (\text{under } r' (f a))\)
  using assms by (auto simp add: embed-def)
moreover
{have \(f a \in \text{Field } r' \text{ \ using assms embed-Field[of } r r' f\} \text{ \ by auto}\)
  hence \(\text{under } r a = \text{underS } r a \cup \{a\} \land\)
  \(\text{under } r' (f a) = \text{underS } r' (f a) \cup \{f a\}\)
  using assms by (auto simp add: order-on-defs Refl-under-underS)
}
moreover
{have \(a \notin \text{underS } r a \land f a \notin \text{underS } r' (f a)\)
  unfolding underS-def by blast
}
ultimately show \(?\text{thesis}\)
by (auto simp add: notIn-Un-bij-betw3)
qed

lemma \(\text{embed-iff-compat-inj-on-ofilter:}\)
assumes \(\text{WELL: Well-order } r \text{ \ and WELL': Well-order } r'\)
shows \(\text{embed } r r' f = (\text{compat } r r' f \wedge \text{inj-on } f \ (\text{Field } r) \wedge \text{wo-rel.ofilter } r' (f'(\text{Field } r)))\)
using assms
proof(auto simp add: embed-compat embed-inj-on embed-Field-ofilter, unfold embed-def, auto)
fix \(a\)
assume \( \ast: \text{inj-on } f \ (\text{Field } r) \) and
\[
\ast\ast: \text{compat } r \ r' \ f \ \text{and}
\ast\ast\ast: \text{wo-rel ofilter } r' \ (f'(\text{Field } r)) \ \text{and}
\ast\ast\ast\ast: a \in \text{Field } r
\]

have \( \text{Well: } \text{wo-rel } r \)
using \( \text{WELL } \text{wo-rel-def}[\text{of } r] \) by simp
hence \( \text{Refl: } \text{Refl } r \)
using \( \text{wo-rel.REFL}[\text{of } r] \) by simp
have \( \text{Total: } \text{Total } r \)
using \( \text{WELL wo-rel-def}[\text{of } r] \) by simp
hence \( \text{Refl: } \text{Refl } r' \)
using \( \text{wo-rel.REFL}[\text{of } r] \) by simp
have \( \text{Total: } \text{Total } r' \)
using \( \text{WELL wo-rel-def}[\text{of } r'] \) by simp
hence \( \text{Antisym: } \text{antisym } r \)
using \( \text{wo-rel.ANTISYM}[\text{of } r'] \) by simp
have \( (a, a) \in r \)
using \( \text{Refl: } \text{Refl } r' \)
refl-on-def[\text{of } r] by auto
hence \( (f a, f a) \in r' \)
using \( \text{compat-def} \) by (auto simp add :)
hence \( f a \in \text{Field } r' \)
unfolding \( \text{Field-def} \) by auto
have \( f a \in f'(\text{Field } r) \)
using \( \text{Refl: } \text{Refl } r' \)
refl-on-def[\text{of } r] by auto
hence \( 2: \text{under } r' \ (f a) \leq f'(\text{Field } r) \)
using \( \text{Well': } \text{wo-rel.ofilter-def}[\text{of } r' \ f'(\text{Field } r)] \) by fastforce

show \( \text{bij-betw } f \ (\text{under } r \ a \ (\text{under } r' \ (f a))) \)
proof\(\text{(unfold bij-betw-def, auto)}\)
show \( \text{inj-on } f \ (\text{under } r \ a) \) by (rule subset-inj-on[OF \ast \text{under-Field}])
next
fix \( b \) assume \( b \in \text{under } r \ a \)
thus \( f b \in \text{under } r' \ (f a) \)
unfolding \( \text{under-def} \) using \( \ast \)
by (auto simp add: \ast)
next
fix \( b' \) assume \( \ast\ast\ast\ast: b' \in \text{under } r' \ (f a) \)
hence \( b' \in f'(\text{Field } r) \)
using \( \ast\ast\ast\ast \) by auto
with \( \text{Field-def}[\text{of } r] \) obtain \( b \) where
\( 3: b \in \text{Field } r \) and \( 4: b' = f b \) by auto
have \( (b, a): r \)
proof
\{ assume \( (a, b) \in r \)
with \( \ast\ast\ast\ast \) have \( (f a, b') : r' \)
by (auto simp add: \ast)
with \( \ast\ast\ast\ast \) have \( f a = b' \)
by (auto simp add: \ast)
with \( 3 \ast\ast\ast\ast \) have \( a = b \)
by (auto simp add: inj-on-def)
}
moreover
{ assume a = b
  hence (b,a) ∈ r using Refl **** 3
  by (auto simp add: refl-on-def)
}
ultimately
show thesis using Total **** 3 by (fastforce simp add: total-on-def)
qed
with ⟦ show b' ∈ f'(under r a)
  unfolding under-def by auto
qed
qed

lemma inv-into-ofilter-embed:
assumes WELL: Well-order r and OF: wo-rel.ofilter r A and
  BIJ: ∀ b ∈ A. bij-betw f (under r b) (under r' (f b)) and
  IMAGE: f ' A = Field r'
shows embed r' r (inv-into A f)
proof −

  have Well: wo-rel r
  using WELL wo-rel-def[of r] by simp
  have Refl: Refl r
  using Well wo-rel.REFL[of r] by simp
  have Total: Total r
  using Well wo-rel.TOTAL[of r] by simp

have 1: bij-betw f A (Field r')
proof (unfold bij-betw-def inj-on-def, auto simp add: IMAGE)
  fix b1 b2
  assume *: b1 ∈ A and **: b2 ∈ A and
    ***: f b1 = f b2
  have 11: b1 ∈ Field r ∧ b2 ∈ Field r
    using * ** Well OF by (auto simp add: wo-rel.ofilter-def)
moreover
{ assume (b1,b2) ∈ r
  hence b1 ∈ under r b2 ∧ b2 ∈ under r b2
  unfolding under-def using 11 Refl
  by (auto simp add: refl-on-def)
  hence b1 = b2 using BIJ * ** ***
  by (simp add: bij-betw-def inj-on-def)
}
moreover
{ assume (b2,b1) ∈ r
  hence b1 ∈ under r b2 ∧ b2 ∈ under r b1
  unfolding under-def using 11 Refl
  by (auto simp add: refl-on-def)
hence \( b_1 = b_2 \) using \( BIJ \) by (simp add: bij-betw-def inj-on-def)
}
ultimately
show \( b_1 = b_2 \)
using Total by (auto simp add: total-on-def)
qed

let \( ?f' = (\text{inv-into} \ A \ f) \)

have 2: \( \forall b \in A. \ bij-betw ?f' \ (\under r \ (f \ b)) \ (\under r \ b) \)
proof (clarify)
  fix \( b \) assume \(*: b \in A \)
  hence \( \under r \ b \leq A \)
  using Well OF by (auto simp add: wo-rel-def)
moreover
have \( f' \ (\under r \ b) = \under r' \ (f \ b) \)
using \( * \) BIJ by (auto simp add: bij-betw-def)
ultimately
show \( bij-betw ?f' \ (\under r' \ (f \ b)) \ (\under r \ b) \)
using 1 by (auto simp add: bij-betw-inv-into-subset)
qed

have 3: \( \forall b' \in \text{Field} \ r'. \ bij-betw ?f' \ (\under r' \ b') \ (\under r \ (?f' \ b')) \)
proof (clarify)
  fix \( b' \) assume \(*: b' \in \text{Field} \ r' \)
  have \( b' = f \ (?f' \ b') \) using \( * \) 1
  by (auto simp add: bij-betw-inv-into-right)
moreover
{obtain \( b \) where \( 31: b \in A \) and \( f \ b = b' \) using IMAGE by force
  hence \( ?f' \ b' = b \) using \( 1 \) by (auto simp add: bij-betw-inv-into-left)
with \( 31 \) have \( ?f' \ b' \in A \) by auto
}
ultimately
show \( bij-betw ?f' \ (\under r' \ b') \ (\under r \ (?f' \ b')) \)
using 2 by auto
qed

thus \( \texttt{thesis} \) unfolding embed-def .

qed

lemma inv-into-underS-embed:
assumes WELL: Well-order \( r \) and
  \( BIJ: \forall b \in \text{under} \ S \ r \ a. \ bij-betw f \ (\under r \ b) \ (\under r' \ (f \ b)) \) and
  \( IN: a \in \text{Field} \ r \) and
  IMAGE: \( f' \ (\under S \ r \ a) = \text{Field} \ r' \)
shows \( \text{embed} \ r' \ r \ (\text{inv-into} \ (\under S \ r \ a) \ f) \)
using assms
by (auto simp add: wo-rel-def wo-rel.underS-ofilter inv-into-ofilter-embed)
lemma `inv-into-Field-embed`:
assumes `WELL: Well-order r and EMB: embed r r' f` and
`IMAGE: Field r' \leq f' (Field r)`
shows `embed r' r (inv-into (Field r) f)`
proof –
  have `(\forall b \in Field r. bij-betw f \under{r} (under r' (f b)))`
  using `EMB by (auto simp add: embed-def)`
moreover
  have `f' (Field r) \leq Field r'`
  using `EMB WELL by (auto simp add: embed-Field)`
ultimately
  show `?thesis using assms by (auto simp add: wo-rel-def wo-rel(Field-ofilter inv-into-ofilter-embed))`
qed

lemma `inv-into-Field-embed-bij-betw`:
assumes `WELL: Well-order r and EMB: embed r r' f` and
`BIJ: bij-betw f (Field r) (Field r')`
shows `embed r' r (inv-into (Field r) f)`
proof –
  have `Field r' \leq f' (Field r)`
  using `BIJ by (auto simp add: bij-betw-def)`
thus `?thesis using assms by (auto simp add: inv-into-Field-embed)`
qed

29.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of of this fact, stated in theorem `wellorders-totally-ordered`:

Fix the well-orders `r::'a rel` and `r'::'a' rel`. Attempt to define an embedding `f::'a \Rightarrow 'a'` from `r` to `r'` in the natural way by well-order recursion ("hoping" that `Field r` turns out to be smaller than `Field r'`), but also record, at the recursive step, in a function `g::'a \Rightarrow bool`, the extra information of whether `Field r'` gets exhausted or not.

If `Field r'` does not get exhausted, then `Field r` is indeed smaller and `f` is the desired embedding from `r` to `r'` (lemma `wellorders-totally-ordered-aux`). Otherwise, it means that `Field r'` is the smaller one, and the inverse of (the "good" segment of) `f` is the desired embedding from `r'` to `r` (lemma `wellorders-totally-ordered-aux2`).

lemma `wellorders-totally-ordered-aux`:
fixes `r::'a rel` and `r'::'a' rel` and
`f :: 'a \Rightarrow 'a'` and \(a::'a\)
assumes `WELL: Well-order r and WELL': Well-order r' and IN: a \in Field r` and
`IH: \forall b \in \under{S} r a. bij-betw f (under r b) (under r' (f b))` and
NOT: \( f' (\text{under}_S r a) \neq \text{Field } r' \) and SUC: \( f a = \text{wo-rel.suc } r' (f (\text{under}_S r a)) \)
shows bij-betw \( f (\text{under}_S r a) (\text{under } r' (f a)) \)

proof –

have Well: wo-rel \( r \) using WELL unfolding wo-rel-def .
hence Refl: Refl \( r \) using wo-rel.REFL[of r] by auto
have Trans: trans \( r \) using Well wo-rel.TRANS[of r] by auto
have Well': wo-rel \( r' \) using WELL' unfolding wo-rel-def .
have OF: wo-rel.ofilter \( r \) (under\( S r a \))
by (auto simp add: Well wo-rel.underS-ofilter)
hence UN: under\( S r a = (\bigcup b \in \text{under}_S r a . \text{under } r b) \)
using Well wo-rel.ofilter-under-UNION[of r under\( S r a \)] by blast

\{ fix \( b \) assume \( \ast: b \in \text{under}_S r a \)
  hence 0\( \ast: (b,a) \in r \wedge b \neq a \) unfolding under\( S\)-def by auto
  have t1\( \ast: b \in \text{Field } r \)
  using \( \ast\) under\( S\)-Field[of r a] by auto
  have t2\( \ast: f (\text{under}_r b) = \text{under } r' (f b) \)
  using IH \( \ast\) by (auto simp add: bij-betw-def)
  hence t3\( \ast: \text{wo-rel.ofilter } r' (f (\text{under}_r b)) \)
  using Well' by (auto simp add: wo-rel.under-ofilter)
  have \( f (\text{under}_r b) \leq \text{Field } r' \)
  using t2 \( \ast\) by (auto simp add: under-Field)
  moreover
  have \( b \in \text{under } r b \)
  using t1 \( \ast\) (auto simp add: Refl Refl-under-in)
  ultimately
  have t4\( \ast: \text{Field } r' \)
  have t5\( \ast: f b \in \text{Field } r' \)
  using t2 t3 t4 \( \ast\) by auto
\}

hence bFact:
\( \forall b \in \text{under}_S r a . f (\text{under}_r b) = \text{under } r' (f b) \) \wedge
  wo-rel.ofilter \( r' (f (\text{under}_r b)) \) \wedge
  \( f b \in \text{Field } r' \) by blast

have subField: \( f (\text{under}_S r a) \leq \text{Field } r' \)
using bFact by blast

have OF': wo-rel.ofilter \( r' (f (\text{under}_S r a)) \)
proof –
  have \( f (\text{under}_S r a) = f' (\bigcup b \in \text{under}_S r a . \text{under } r b) \)
  using UN by auto
  also have \( . . . = (\bigcup b \in \text{under}_S r a . f' (\text{under } r b)) \) by blast
  also have \( . . . = (\bigcup b \in \text{under}_S r a . (\text{under } r' (f b))) \)
  using bFact by auto
finally
have \( f'(\text{under} S r a) = (\bigcup b \in \text{under} S r a. (\text{under} r' (f b))) \).
thus \( \square \text{thesis} \)
using \( \text{Well}^t \text{bFact} \)
\( \text{wo-rel.ofilter\text{-}\text{UNION}}[\text{of} r' \text{under} S r a \lambda b. \text{under} r' (f b)] \) by fastforce
qed

have \( f'(\text{under} S r a) \cup \text{Above} S r' (f'(\text{under} S r a)) = \text{Field} r' \)
using \( \text{Well}^t \text{OF}' \) by (auto simp add: \( \text{wo-rel.ofilter\text{-}\text{Above}\text{S}\text{\text{-}\text{Field}}} \))
hence \( \text{NE}: \text{Above} S r' (f'(\text{under} S r a)) \neq \{\} \)
using \( \text{subField} \text{NOT} \) by blast

have \( \text{INCL1}: f'(\text{under} S r a) \leq \text{under} S r' (f a) \)
proof (auto)
  fix \( b \) assume \( *: b \in \text{under} S r a \)
  have \( f b \neq f a \wedge (f b, f a) \in r' \)
  using \( \text{subField} \text{Well}^t \text{SUC NE} * \)
  \( \text{wo-rel.suc\text{-}\text{greater}}[\text{of} r' f'(\text{under} S r a) f b] \) by force
  thus \( f b \in \text{under} S r' (f a) \)
  unfolding \( \text{under}\text{-}\text{def} \) by simp
qed

have \( \text{INCL2}: \text{under} S r' (f a) \leq f'(\text{under} S r a) \)
proof
  fix \( b' \) assume \( b' \in \text{under} S r' (f a) \)
  hence \( b' \neq f a \wedge (b', f a) \in r' \)
  unfolding \( \text{under}\text{-}\text{def} \) by simp
  thus \( b' \in f'(\text{under} S r a) \)
  using \( \text{Well}^t \text{SUC NE OF}' \)
  \( \text{wo-rel.suc\text{-}\text{ofilter\text{-}\text{in}}}[\text{of} r' f' \text{\text{-}\text{under} S r a b}] \) by auto
qed

have \( \text{INJ}: \text{inj\text{-}\text{on} f} (\text{under} S r a) \)
proof−
  have \( \forall b \in \text{under} S r a. \text{inj\text{-}\text{on} f} (\text{under} r b) \)
  using \( \text{IH} \) by (auto simp add: bij-betw-def)
  moreover
  have \( \forall b. \text{wo-rel.ofilter} r (\text{under} r b) \)
  using \( \text{Well} \) by (auto simp add: \( \text{wo-rel.under-ofilter} \))
  ultimately show \( \square \text{thesis} \)
  using WELL \text{bFact} \text{UN}
  \( \text{UNION\text{-}\text{inj\text{-}\text{on-ofilter}}}[\text{of} r \text{\text{-}\text{under} S r a b} \lambda b. \text{under} r b f] \)
  by auto
qed

have \( \text{BIJ}: \text{bij\text{-}\text{betw f} (under} S r a) (\text{under} S r' (f a)) \)
unfolding \( \text{bij\text{-}\text{betw-def} \)\)
using \( \text{INJ INCL1 INCL2} \) by auto
have \( f \in \text{Field } r' \)
using Well' subField NE SUC
by (auto simp add: wo-rel.suc-inField)
thus \(?\text{thesis}\)
using WELL WELL' IN BIJ under-underS-bij-betw[of r r' a f]
by auto
qed

lemma wellorders-totally-ordered-aux2:
fixes \( r :: \text{a rel} \) and \( r' :: \text{a' rel} \) and
\( f :: \text{a' a} \) and \( g :: \text{bool a a} \) and \( a :: \text{a} \)
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and

\[
\begin{align*}
\text{MAIN1:} & \quad \forall a. (\text{False } \notin g'(\text{under } r a) \land f'(\text{under } r a) \neq \text{Field } r') \\
& \quad \quad \quad \quad \rightarrow f a = \text{wo-rel.suc } r' (f'(\text{under } r a)) \land g a = \text{True} \\
& \quad \quad \quad \quad \quad \land \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{False } \notin (g'(\text{under } r a)) \land f'(\text{under } r a) \neq \text{Field } r') \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow g a = \text{False}
\end{align*}
\]

\[
\text{MAIN2:} \quad \forall a. a \in \text{Field } r \land \text{False } \notin g'(\text{under } r a) \quad \rightarrow \\
\text{bij-betw } f (\text{under } r a) (\text{under } r' (f a)) \quad \text{and}
\]
Case: \( a \in \text{Field } r \land \text{False } \notin g'(\text{under } r a) \)
shows \( \exists f', \text{ embed } r' r f' \)

proof

have Well: wo-rel \( r \) using WELL unfolding wo-rel-def .
hence Refl: Refl \( r \) using wo-rel.REFL[of r] by auto
have Trans: trans \( r \) using Well wo-rel.TRANS[of r] by auto
have Antisym: antisym \( r \) using Well wo-rel.ANTISYM[of r] by auto
have Well': wo-rel \( r' \) using WELL' unfolding wo-rel-def .

have 0: under \( r a = \text{under } r a \cup \{a\} \)
using Refl Case by(auto simp add: Refl-under-underS)

have 1: \( g a = \text{False} \)
proof
  \{assume \( g a \neq \text{False} \)
  with 0 Case have \( \text{False } \in g'(\text{under } r a) \) by blast
  with MAIN1 have \( g a = \text{False} \) by blast\}
thus \(?\text{thesis}\) by blast
qed

let \(?A = \{a \in \text{Field } r. \ g a = \text{False}\}\)
let \(?a = (\text{wo-rel.minim } r \ ?A)\)

have 2: \(?A \neq \{\} \land \ ?A \leq \text{Field } r \) using Case 1 by blast

have 3: \( \text{False } \notin g'(\text{under } r \ ?a) \)
proof
  assume \( \text{False } \in g'(\text{under } r \ ?a) \)
  then obtain \( b \) where \( b \in \text{under } r \ ?a \) and 31: \( g b = \text{False} \) by auto
  hence 32: \( (b,\ ?a) \in r \land b \neq \ ?a \)
  by (auto simp add: underS-def)
hence $b \in \text{Field } r$ unfolding Field-def by auto

with 31 have $b \in ?A$ by auto

hence $(?a,b) \in r$ using wo-rel.minim-least 2 Well by fastforce

with 32 Antisym show False

by (auto simp add: antisym-def)

qed

have temp: $?a \in ?A$

using Well 2 wo-rel.minim-in[of r $?A] by auto

hence 4: $?a \in \text{Field } r$ by auto

have 5: $g \ ?a = \text{False}$ using temp by blast

have 6: $f'(\text{under } S \ ?a) = \text{Field } r'$

using MAIN1[of $?a] 3 5 by blast

have 7: $\forall b \in \text{under } S \ ?a. \bij-betw f (\text{under } r \ b) (\text{under } r' (f \ b))$

proof

fix $b$ assume as: $b \in \text{under } S \ ?a$

moreover

have wo-rel.ofilter r (\text{under } S \ ?a)

using Well by (auto simp add: wo-rel.underS-ofilter)

ultimately

have False $\notin g'(\text{under } r \ b)$ using 3 Well by (subst (asm) wo-rel.ofilter-def)

fast+

moreover have $b \in \text{Field } r$

unfolding Field-def using as by (auto simp add: underS-def)

ultimately

show bij-betw f (\text{under } r \ b) (\text{under } r' (f \ b))

using MAIN2 by auto

qed

have $\text{embed } r' \ r \ (\text{inv-into } (\text{under } S \ ?a) \ f)$

using WELL WELL' 7 4 6 inv-into-underS-embed[of r $?a f r' \ f'] by auto

thus $\text{thesis}$

unfolding embed-def by blast

qed

theorem wellorders-totally-ordered:

fixes $r :: a \ rel$ and $r' :: a' \ rel$

assumes WELL: Well-order $r$ and WELL': Well-order $r'$

shows $(\exists f. \text{embed } r \ r' \ f) \lor (\exists f'. \text{embed } r' \ r \ f')$

proof

have $\text{Well } \ : \ \text{wo-rel } r \ \text{using } \text{WELL unfolding wo-rel-def }$

hence $\text{Refl } r \ \text{using } \text{wo-rel.REFL[of } r \text{] by auto}$

have $\text{Trans } \ : \ \text{trans } r \ \text{using } \text{Well wo-rel.TRANS[of } r \text{] by auto}$

have $\text{Well' } : \ \text{wo-rel } r' \ \text{using } \text{WELL' unfolding wo-rel-def }$
obtain \( H \) where \( H \text{-def: } H = \)
(\( \lambda h \ a. \) if \( \text{False} \notin (\text{snd} \ o \ h)\text{'}(\text{underS} \ r \ a) \land (\text{fst} \ o \ h)\text{'}(\text{underS} \ r \ a) \neq \text{Field} \ r' \)
then \( \text{wo-rel.suc} \ r' ((\text{fst} \ o \ h)\text{'}(\text{underS} \ r \ a)) \), \( \text{True} \)
else \((\text{undefined}, \text{False}))\) by blast
have \( \text{Adm: wo-rel.adm-wo} \ r \ H \)
using \( \text{Well} \)
proof\((\text{unfold wo-rel.adm-wo-def, clarify})\)
fix \( h1::'a = 'a' * \text{bool} \) and \( h2::'a = 'a' * \text{bool} \) and \( x \)
assume \( \forall y \in \text{underS} \ r \ x. \ h1 \ y = h2 \ y \)
 pricey \( \forall y \in \text{underS} \ r \ x. \ (\text{fst} \ o \ h1) \ y = (\text{fst} \ o \ h2) \ y \land \)
\((\text{snd} \ o \ h1) \ y = (\text{snd} \ o \ h2) \ y \) by auto
hence \((\text{fst} \ o \ h1)\text{'}(\text{underS} \ r \ x) = (\text{fst} \ o \ h2)\text{'}(\text{underS} \ r \ x) \land \)
\((\text{snd} \ o \ h1)\text{'}(\text{underS} \ r \ x) = (\text{snd} \ o \ h2)\text{'}(\text{underS} \ r \ x) \)
by \((\text{auto simp add: image-def})\)
thus \( H \ h1 \ x = H \ h2 \ x \) by \((\text{simp add: H-def del: not-False-in-image-Ball})\)
qed

obtain \( h::'a = 'a' * \text{bool} \) and \( f::'a = 'a' \) and \( g::'a = \text{bool} \)
where \( h\text{-def: } h = \text{wo-rel.worec} \ r \ H \) and
\( f\text{-def: } f = \text{fst} \ o \ h \) and \( g\text{-def: } g = \text{snd} \ o \ h \) by blast

obtain \( \text{test where test-def: } \)
\( \text{test} = (\lambda a. \text{False} \notin (g'(\text{underS} \ r \ a)) \land f'(\text{underS} \ r \ a) \neq \text{Field} \ r') \) by blast

have \( \ast: \bigwedge a. \ h \ a = H \ h \ a \)
using \( \text{Adm Well wo-rel.worec-fixpoint[of r H]} \) by \((\text{simp add: h-def})\)

have \( \text{Main1: } \) \( \bigwedge a. \ 
(\text{test} \ a \longrightarrow f \ a = \text{wo-rel.suc} \ r' ((f'(\text{underS} \ r \ a)) \land g \ a = \text{True}) \land \)
(\( \neg \text{test} \ a \longrightarrow g \ a = \text{False} \) \)

proof
- fix \( a \) show \( \text{(test} \ a \longrightarrow f \ a = \text{wo-rel.suc} \ r' ((f'(\text{underS} \ r \ a)) \land g \ a = \text{True}) \land \)
(\( \neg \text{test} \ a \longrightarrow g \ a = \text{False} \) \)
using \( \ast[\text{of } a] \) test-def f-def g-def H-def by auto

qed

let \( \phi = \lambda a. \ a \in \text{Field} \ r \land \text{False} \notin g'(\text{under} \ r \ a) \longrightarrow \)
\( \text{bij-betw} \ f \ (\text{under} \ r \ a) \ (\text{under} \ r' \ (f \ b)) \)

have \( \text{Main2: } \bigwedge a. \ \phi \ a \)
proof
- fix \( a \) show \( \phi \ a \)
proof\((\text{rule wo-rel.well-order-induct[of r } \phi \text{]}\),
\( \text{simp only: Well, clarify})\)

fix \( a \)
assum \( \ast: \forall b. b \neq a \land (b,a) \in r \longrightarrow \phi \ b \) and

\( \ast: \text{False} \notin g'(\text{under} \ r \ a) \)

have \( 1: \forall b \in \text{underS} \ r \ a. \ \text{bij-betw} \ f \ (\text{under} \ r \ b) \ (\text{under} \ r' \ (f \ b)) \)
proof\((\text{clarify})\)

fix \( b \) assume \( \ast: b \in \text{underS} \ r \ a \)

hence \( 0: \ (b,a) \in r \land b \neq a \) unfolding underS-def by auto
moreover have \( b \in \text{Field } r \)
using \( *** \text{ underS-Field[of } r a \text{]} \) by auto
moreover have \( \text{False} \notin g'(\text{under } r b) \)
using \( \emptyset *** \text{ Trans under-incr[of } r b a \text{]} \) by auto
ultimately show \( \text{bij-betw } f (\text{under } r b) (\text{under } r' (f b)) \)
using \( \text{IH} \) by auto
qed

have \( 21: \text{False} \notin g'(\text{under } r a) \)
using \( ** \text{ underS-subset-under[of } r a \text{]} \) by auto
have \( 22: g'(\text{under } r a) \leq \{ \text{True} \} \) using \( ** \) by auto
moreover have \( 23: a \in \text{under } r a \)
using \( \text{Refl} * \) by \((\text{auto simp add: Refl-under-in})\)
ultimately have \( 24: g a = \text{True} \) by blast
have \( 2: f'(\text{under } S r a) \neq \text{Field } r' \)
proof
  assume \( f'(\text{under } S r a) = \text{Field } r' \)
  hence \( g a = \text{False} \) using \( \text{Main1 test-def} \) by blast
  with \( 24 \) show \( \text{False} \) using \( ** \) by blast
qed

have \( 3: f a = \text{wo-rel.suc } r' (f'(\text{under } S r a)) \)
using \( 21 \ 2 \text{ Main1 test-def} \) by blast

show \( \text{bij-betw } f (\text{under } r a) (\text{under } r' (f a)) \)
using \( \text{WELL WELL'} 1 \ 2 \ 3 * \text{ wellorders-totally-ordered-aux[of } r r' a f \text{]} \) by auto
qed

let \( ?\chi = (\lambda a. \ a \in \text{Field } r \land \text{False} \in g'(\text{under } r a)) \)
show \( ?\text{thesis} \)
proof
\( \text{cases } \exists a. ?\chi a \)
  assume \( \neg (\exists a. ?\chi a) \)
  hence \( \forall a \in \text{Field } r. \ \text{bij-betw } f (\text{under } r a) (\text{under } r' (f a)) \)
  using \( \text{Main2} \) by blast
  thus \( ?\text{thesis unfolding embed-def} \) by blast
next
  assume \( \exists a. ?\chi a \)
  then obtain \( a \) where \( ?\chi a \) by blast
  hence \( \exists f', \text{ embed } r' r f' \)
  using \( \text{wellorders-totally-ordered-aux2[of } r r' g f a \text{]} \)
  \( \text{WELL WELL'} \text{ Main1 Main2 test-def} \) by fast
  thus \( ?\text{thesis} \) by blast
qed

qed


29.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is at most one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

lemma embed-determined:
assumes WELL: Well-order r and WELL': Well-order r' and
\[ \text{EMB}: \text{embed } r r' f \text{ and } \text{IN}: a \in \text{Field } r \]
sows \( f a = \text{wo-rel.suc } r' (f(\text{underS } r a)) \)
proof–
\begin{itemize}
  \item have \( \text{bij-betw } f (\text{underS } r a) (\text{underS } r' (f a)) \)
  \item using assms by (auto simp add: embed-underS)
  \item hence \( f(\text{underS } r a) = \text{underS } r' (f a) \)
  \item by (auto simp add: bij-betw-def)
\end{itemize}
moreover
\begin{itemize}
  \item have \( f a \in \text{Field } r' \) using IN
  \item using EMB WELL embed-Field[of r r' f] by auto
  \item hence \( f a = \text{wo-rel.suc } r' (\text{underS } r' (f a)) \)
  \item using WELL' by (auto simp add: wo-rel-def wo-rel.suc-underS)
\end{itemize}
ultimately show ?thesis by simp
qed

lemma embed-unique:
assumes WELL: Well-order r and WELL': Well-order r' and
\[ \text{EMB}: \text{embed } r r' f \text{ and } \text{EMB'}: \text{embed } r' r g \]
sows \( a \in \text{Field } r \rightarrow f a = g a \)
proof(rule wo-rel.well-order-induct[of r], auto simp add: WELL wo-rel-def)
fix a
assume IH: \( \forall b. b \neq a \land (b,a): r \rightarrow b \in \text{Field } r \rightarrow f b = g b \) and
\[ a: a \in \text{Field } r \]
hence \( \forall b \in \text{underS } r a. f b = g b \)
unfolding underS-def by (auto simp add: Field-def)
hence \( f(\text{underS } r a) = g'(\text{underS } r a) \) by force
thus \( f a = g a \)
using assms * embed-determined[of r r' f a] embed-determined[of r r' g a] by auto
qed

lemma embed-bothWays-inverse:
assumes WELL: Well-order r and WELL': Well-order r' and
\[ \text{EMB}: \text{embed } r r' f \text{ and } \text{EMB'}: \text{embed } r' r f' \]
sows \( \forall a \in \text{Field } r. f'(f a) = a \) \land (\forall a' \in \text{Field } r'. f(f' a') = a')
proof–
\begin{itemize}
  \item have embed r r (f' a f) using assms
  \item by(auto simp add: comp-embed)
\end{itemize}
moreover have embed \( r \rightarrow id \) using assms
by (auto simp add: id-embed)
ultimately have \( \forall a \in \mathrm{Field}\ r. \ f'(f\ a) = a \)
using assms embed-unique[of \( r, r', f \) o f id] id-def by auto
moreover
\{ have embed \( r' \rightarrow r' \) (f o f') using assms 
by(auto simp add: comp-embed)
moreover have embed \( r' \rightarrow id \) using assms 
by (auto simp add: id-embed)
ultimately have \( \forall a' \in \mathrm{Field}\ r'. \ f(f' a') = a' \)
using assms embed-unique[of \( r', r', f \) o f id] id-def by auto 
\}
ultimately show \( \mathrm{thesis} \) by blast
qed

lemma embed-bothWays-bij-betw:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
EMB: embed \( r \rightarrow r' \) f and EMB': embed \( r' \rightarrow r \) g
shows bij-betw f (\( \mathrm{Field}\ r \)) (\( \mathrm{Field}\ r' \))
proof –
let \( \mathcal{A} = \mathrm{Field}\ r \) let \( \mathcal{A}' = \mathrm{Field}\ r' \)
have embed \( r \rightarrow (g\ o\ f) \) \( \land \) embed \( r' \rightarrow (f\ o\ g) \)
using assms by (auto simp add: comp-embed)
\hence \( \forall a \in \mathcal{A}. \ g(f\ a) = a \) \( \land \) \( \forall a' \in \mathcal{A}'. \ f(g\ a') = a' \)
using WELL id-embed[of \( r \)] embed-unique[of \( r\ r\ g\ o\ f\ id \)]
\( \land \) WELL' id-embed[of \( r' \)] embed-unique[of \( r'\ r'\ f\ o\ g\ id \)]
id-def by auto
have 2: \( \forall a \in \mathcal{A}. \ f a \in \mathcal{A}' \) \( \land \) \( \forall a' \in \mathcal{A}'. \ g a' \in \mathcal{A} \)
using assms embed-Field[of \( r \rightarrow r' \) f] embed-Field[of \( r' \rightarrow r \) g] by blast

show \( \mathrm{thesis} \)
proof(unfold bij-betw-def inj-on-def, auto simp add: 2)
fix a b assume *: \( a \in \mathcal{A} \) \( \land \) \( b \in \mathcal{A} \) and **: \( f\ a = f\ b \)
have a = g(f a) \( \land \) b = g(f b) using * 1 by auto
with ** show a = b by auto
next
fix a' assume *: \( a' \in \mathcal{A}' \)
\hence g a' \in \mathcal{A} \( \land \) f(g a') = a' using 1 2 by auto
thus a' \in \mathcal{A}' \( \implies \) \( \mathcal{A} \) by force
qed

qed

lemma embed-bothWays-iso:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
EMB: embed \( r \rightarrow r' \) f and EMB': embed \( r' \rightarrow r \) g
shows iso \( r \rightarrow r' \)
unfolding iso-def using assms by (auto simp add: embed-bothWays-bij-betw)
29.5 More properties of embeddings, strict embeddings and isomorphisms

lemma embed-bothWays-Field-bij-betw:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
EMB: embed \( r \to r' \) and EMB': embed \( r' \to r \)
shows bij-betw \( f \) (Field \( r \)) (Field \( r' \))

proof –
  have \((\forall a \in \text{Field } r. \, f' (f a) = a) \land (\forall a' \in \text{Field } r'. \, f(f' a') = a')\)
  using assms by (auto simp add: embed-bothWays-inverse)
  moreover
  have \(f' (\text{Field } r) \leq \text{Field } r' \land f' \) (Field \( r' \)) \(\leq \) Field \( r \)
  using assms by (auto simp add: embed-Field)
  ultimately
  show ?thesis using bij-betw-byWitness[of Field \( r \) \( f' \) Field \( r' \)] by auto
qed

lemma embedS-comp-embed:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and WELL'': Well-order \( r'' \)
  and EMB: embed \( r \to r' \) \( f \) and EMB': embed \( r' \to r'' \) \( f' \)
shows embedS \( r \to r'' \) (\( f' \) o \( f \))

proof –
  let \(?g = (f' \circ f)\) let \(?h = \text{inv-into} (\text{Field } r) \ ?g\)
  have 1: embed \( r \to r' \) \( f \) \(\land \neg \) (bij-betw \( f \) (Field \( r \)) (Field \( r' \)))
  using EMB by (auto simp add: embedS-def)
  hence 2: embed \( r \to r'' \) \(?g\)
  using WELL EMB' comp-embed[of \( r' \) \( f \) \( r'' \) \( f' \)] by auto
  moreover
  {assume bij-betw \(?g = (\text{Field } r) \ (\text{Field } r'')\)
   hence embed \( r'' \to r' \) \(?h \) using 2 WELL
   by (auto simp add: inv-into-Field-embed-bij-betw)
   hence embed \( r'' \to r' \) \(?h \circ f\) using WELL' EMB'
   by (auto simp add: comp-embed)
   hence bij-betw \( f \) (Field \( r' \)) (Field \( r'' \)) using WELL WELL' 1
   by (auto simp add: embed-bothWays-Field-bij-betw)
   with 1 have False by blast
  }
  ultimately show ?thesis unfolding embedS-def by auto
qed

lemma embed-comp-embedS:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and WELL'': Well-order \( r'' \)
  and EMB: embed \( r \to r' \) \( f \) and EMB': embedS \( r' \to r'' \) \( f' \)
shows embedS \( r \to r'' \) (\( f' \) o \( f \))

proof –
  let \(?g = (f' \circ f)\) let \(?h = \text{inv-into} (\text{Field } r) \ ?g\)
  have 1: embed \( r'' \to r' \) \( f' \) \(\land \neg \) (bij-betw \( f' \) (Field \( r' \)) (Field \( r'' \)))
  using EMB' by (auto simp add: embedS-def)
hence 2: embed r r'' ?g
using WELL EMB comp-embed[of r r' f r'' f'] by auto
moreover
{assume bij-betw ?g (Field r) (Field r'')
 hence embed r'' r ?h using 2 WELL
 by (auto simp add: inv-into-Field-embed-bij-betw)
 hence embed r'' r' (f o ?h) using WELL'' EMB
 by (auto simp add: comp-embed)
 hence bij-betw f' (Field r') (Field r'') using WELL' WELL'' 1
 by (auto simp add: embed-bothWays-Field-bij-betw)
 with 1 have False by blast
}
ultimately show ?thesis unfolding embedS-def by auto
qed

lemma embed-comp-iso:
assumes WELL: Well-order r and WELL': Well-order r' and WELL'': Well-order r''
 and EMB: embed r r' f and EMB': iso r' r'' f'
shows embed r r'' (f' o f)
using assms unfolding iso-def
by (auto simp add: comp-embed)

lemma iso-comp-embed:
assumes WELL: Well-order r and WELL': Well-order r' and WELL'': Well-order r''
 and EMB: iso r r' f and EMB': embed r' r'' f'
shows embed r r'' (f' o f)
using assms unfolding iso-def
by (auto simp add: comp-embed)

lemma embedS-comp-iso:
assumes WELL: Well-order r and WELL': Well-order r' and WELL'': Well-order r''
 and EMB: embedS r r' f and EMB': iso r' r'' f'
shows embedS r r'' (f' o f)
using assms unfolding iso-def
by (auto simp add: embedS-comp-embed)

lemma iso-comp-embedS:
assumes WELL: Well-order r and WELL': Well-order r' and WELL'': Well-order r''
 and EMB: iso r r' f and EMB': embedS r' r'' f'
shows embedS r r'' (f' o f)
using assms unfolding iso-def using embed-comp-embedS
by (auto simp add: embed-comp-embedS)

lemma embedS-Field:
assumes WELL: Well-order r and EMB: embedS r r' f
shows \( f' \cdot (\text{Field } r) < \text{Field } r' \)

proof –
- have \( f'(\text{Field } r) \leq \text{Field } r' \) using assms
  by (auto simp add: embed-Field embedS-def)
moreover
  \{ have inj-on \( f \cdot (\text{Field } r) \) using assms
    by (auto simp add: embedS-def embed-inj-on)
    hence \( f'(\text{Field } r) \neq \text{Field } r' \) using EMB
    by (auto simp add: embedS-def bij-betw-def)
  \}
ultimately show \( \text{thesis} \) by blast
qed

lemma embedS-iff:
assumes WELL: \( \text{Well-order } r \) and ISO: \( \text{embed } r \to r' \)
shows embedS \( r \times r' \cdot f = (f' \cdot (\text{Field } r) < \text{Field } r') \)
proof
- assume \( \text{embedS } r \times r' \cdot f \)
  thus \( f' \cdot \text{Field } r \subseteq \text{Field } r' \)
  using WELL by (auto simp add: embedS-Field)
next
- assume \( f' \cdot \text{Field } r \subseteq \text{Field } r' \)
  hence \( \neg \text{bij-betw } f \cdot \text{Field } r \times \text{Field } r' \)
  unfolding bij-betw-def by blast
  thus \( \text{embedS } r \times r' \cdot f \) unfolding embedS-def
  using ISO by auto
qed

lemma iso-Field:
iso \( r \times r' \cdot f \implies f' \cdot (\text{Field } r) = \text{Field } r' \)
using assms by (auto simp add: iso-def bij-betw-def)

lemma iso-iff:
assumes \( \text{Well-order } r \)
shows iso \( r \times r' \cdot f = (\text{embed } r \to r' \cdot f \land f' \cdot (\text{Field } r) = \text{Field } r') \)
proof
- assume \( \text{iso } r \times r' \cdot f \)
  thus \( \text{embed } r \to r' \cdot f \land f' \cdot (\text{Field } r) = \text{Field } r' \)
  by (auto simp add: iso-Field iso-def)
next
- assume \( * \cdot \text{embed } r \to r' \cdot f \land f' \cdot \text{Field } r = \text{Field } r' \)
  hence inj-on \( f \cdot (\text{Field } r) \) using assms by (auto simp add: embed-inj-on)
  with \( * \) have \( \text{bij-betw } f \cdot (\text{Field } r) \times (\text{Field } r') \)
  unfolding bij-betw-def by simp
  with \( * \) show iso \( r \times r' \cdot f \) unfolding iso-def by auto
qed

lemma iso-iff2:
assumes \( \text{Well-order } r \)
shows iso r r' f = (bij-betw f (Field r) (Field r')) ∧
(∀ a ∈ Field r. ∀ b ∈ Field r.
((a, b) ∈ r) = ((f a, f b) ∈ r'))

using assms

proof(auto simp add: iso-def)

fix a b
assume embed r r' f
hence compat r r' f using embed-compat[of r] by auto
moreover assume (a, b) ∈ r
ultimately show (f a, f b) ∈ r' using compat-def[of r] by auto

next
let ?f' = inv-into (Field r) f
assume embed r r' f and 1: bij-betw f (Field r) (Field r')
hence embed r' r ?f' using assms
by (auto simp add: inv-into-Field-embed-bij-betw)

next
assume *: bij-betw f (Field r) (Field r') and
**: ∀ a ∈ Field r. ∀ b ∈ Field r. ((a, b) ∈ r) = ((f a, f b) ∈ r')

have 1: ∃ a. under r a ≤ Field r ∧ under r'(f a) ≤ Field r'
by (auto simp add: under-Field)

have 2: inj-on f (Field r) using * by (auto simp add: bij-betw-def)
{fix a assume ***: a ∈ Field r
have bij-betw f (under r a) (under r' (f a))
proof(unfold bij-betw-def, auto)
show inj-on f (under r a) using 1 2 subset-inj-on by blast

next
fix b assume b ∈ under r a
hence a ∈ Field r ∧ b ∈ Field r ∧ (b, a) ∈ r
unfolding under-def by (auto simp add: Field-def Range-def Domain-def)
with 1 ** show f b ∈ under r' (f a)
unfolding under-def by auto

next
fix b' assume b' ∈ under r' (f a)
hence 3: (b', f a) ∈ r' unfolding under-def by simp
hence b' ∈ Field r' unfolding Field-def by auto
with * obtain b where b ∈ Field r ∧ f b = b'
unfolding bij-betw-def by force
with 3 ***
show b' ∈ f⁻¹ (under r a) unfolding under-def by blast
qed
}
thus embed r r' f unfolding embed-def using * by auto

qed
lemma iso-iff3:
assumes WELL: Well-order r and WELL': Well-order r'
shows iso r r' f = (bij-betw f (Field r) (Field r') ∧ compat r r' f)
proof
  assume iso r r' f
  thus bij-betw f (Field r) (Field r') ∧ compat r r' f
unfolding compat-def using WELL by (auto simp add: iso-iff2 Field-def)
next
  have Well: wo-rel r ∧ wo-rel r' using WELL WELL'
  by (auto simp add: wo-rel-def)
  assume *: bij-betw f (Field r) (Field r') ∧ compat r r' f
  thus iso r r' f
unfolding compat-def using assms
proof(auto simp add: iso-iff2)
  fix a b assume **: a ∈ Field r b ∈ Field r and
  ***: (f a, f b) ∈ r'
  {assume (b,a): r using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
    hence (f b, f a) ∈ r' using * unfolding compat-def by auto
    hence f a = f b
    using Well *** wo-rel.ANTISYM[of r'] antisym-def[of r'] by blast
    hence a = b using * unfolding bij-betw-def inj-on-def by auto
    hence (a,b) ∈ r using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
  }
  thus (a,b) ∈ r
  using Well ** wo-rel.TOTAL[of r] total-on-def[of - r] by blast
qed
qed
end

30 BNF-Constructions-on-Wellorders: Constructions on Wellorders as Needed by Bounded Natural Functors

theory BNF-Constructions-on-Wellorders
imports BNF-Wellorder-Embedding
begin

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations ordLeq, of being embedded (abbreviated ≤o), ordLess, of being strictly embedded (abbreviated <o), and ordIso, of being isomorphic (abbreviated =o). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section is that <o is well-founded.
30.1 Restriction to a set

abbreviation \( \text{Restr} :: 'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ rel} \)

where \( \text{Restr} \ r \ A \equiv r \ \text{Int} \ (A \times A) \)

lemma Restr-subset:
\( A \leq B \Rightarrow \text{Restr} (\text{Restr} \ r \ B) \ A = \text{Restr} \ r \ A \)
by blast

lemma Restr-Field: \( \text{Restr} \ r \ (\text{Field} \ r) = r \)
unfolding Field-def by auto

lemma Refl-Restr:
\( \text{Refl} \ r \Rightarrow \text{Refl}(\text{Restr} \ r \ A) \)
unfolding refl-on-def Field-def by auto

lemma antisym-Restr:
\( \text{antisym} \ r \Rightarrow \text{antisym}(\text{Restr} \ r \ A) \)
unfolding antisym-def Field-def by auto

lemma Total-Restr:
\( \text{Total} \ r \Rightarrow \text{Total}(\text{Restr} \ r \ A) \)
unfolding total-on-def Field-def by auto

lemma trans-Restr:
\( \text{trans} \ r \Rightarrow \text{trans}(\text{Restr} \ r \ A) \)
unfolding trans-def Field-def by blast

lemma Preorder-Restr:
\( \text{Preorder} \ r \Rightarrow \text{Preorder}(\text{Restr} \ r \ A) \)
unfolding preorder-on-def by (simp add: Refl-Restr trans-Restr)

lemma Partial-order-Restr:
\( \text{Partial-order} \ r \Rightarrow \text{Partial-order}(\text{Restr} \ r \ A) \)
unfolding partial-order-on-def by (simp add: Preorder-Restr antisym-Restr)

lemma Linear-order-Restr:
\( \text{Linear-order} \ r \Rightarrow \text{Linear-order}(\text{Restr} \ r \ A) \)
unfolding linear-order-on-def by (simp add: Partial-order-Restr Total-Restr)

lemma Well-order-Restr:
assumes Well-order \( r \)
shows Well-order(\text{Restr} \ r \ A)
proof
have \( \text{Restr} \ r \ A - \text{Id} \leq r - \text{Id} \) using Restr-subset by blast
hence \( \text{wf}(\text{Restr} \ r \ A - \text{Id}) \) using assms
using well-order-on-def wf-subset by blast
thus \(?thesis \) using assms unfolding well-order-on-def by (simp add: Linear-order-Restr)
qed
lemma Field-Restr-subset: Field(Restr r A) ≤ A
  by (auto simp add: Field-def)

lemma Refl-Field-Restr:
  Refl r ==> Field(Restr r A) = (Field r) Int A
  unfolding refl-on-def Field-def by blast

lemma Refl-Field-Restr2:
  [Refl r; A ≤ Field r] ==> Field(Restr r A) = A
  by (auto simp add: Refl-Field-Restr)

lemma well-order-on-Restr:
  assumes WELL: Well-order r and SUB: A ≤ Field r
  shows well-order-on A (Restr r A)
  using assms
  using Well-order-Restr[of r A] Refl-Field-Restr2[of r A]
  order-on-defs[of Field r r] by auto

30.2 Order filters versus restrictions and embeddings

lemma Field-Restr-ofilter:
  [[Well-order r; wo-rel.ofilter r A]] ==> Field(Restr r A) = A
  by (auto simp add: wo-rel-def wo-rel.ofilter-def wo-rel.REFL Refl-Field-Restr2)

lemma ofilter-Restr-under:
  assumes WELL: Well-order r and OF: wo-rel.ofilter r A and IN: a ∈ A
  shows under (Restr r A) a = under r a
  using assms wo-rel-def
  proof (auto simp add: wo-rel.ofilter-def under-def)
    fix b assume *: a ∈ A and (b,a) ∈ r
    hence b ∈ under r a ∧ a ∈ Field r
    unfolding under-def using Field-def by fastforce
    thus b ∈ A using * assms by (auto simp add: wo-rel-def wo-rel.ofilter-def)
  qed

lemma ofilter-embed:
  assumes Well-order r
  shows wo-rel.ofilter r A = (A ≤ Field r ∧ embed (Restr r A) r id)
  proof
    assume *: wo-rel.ofilter r A
    show A ≤ Field r ∧ embed (Restr r A) r id
      proof (unfold embed-def, auto)
        fix a assume a ∈ A thus a ∈ Field r using assms *
        by (auto simp add: wo-rel-def wo-rel.ofilter-def)
      next
        fix a assume a ∈ Field (Restr r A)
        thus bij-betw id (under (Restr r A) a) (under r a) using assms *
        by (simp add: ofilter-Restr-under Field-Restr-ofilter)
    qed
next

assume ∗: A ≤ Field r ∧ embed (Restr r A) r id
hence Field(REstr r A) ≤ Field r
using assms embed-Field[of Restr r A r id] id-def
Well-order-Restr[of r] by auto

{ fix a assume a ∈ A
  hence a ∈ Field(REstr r A) using ∗ assms
  by (simp add: order-on-defs Refl-Field-Restr2)
hence bij-betw id (under (Restr r A) a) (under r a)
using ∗ unfolding embed-def by auto
hence under r a ≤ under (Restr r A) a
unfolding bij-betw-def by auto
also have ... ≤ Field(REstr r A) by (simp add: under-Field)
also have ... ≤ A by (simp add: Field-Restr-subset)
finally have under r a ≤ A .
}
thus wo-rel.ofilter r A using assms ∗ by (simp add: wo-rel-def wo-rel.ofilter-def)
qed

lemma ofilter-Restr-Int:
assumes WELL: Well-order r and OFA: wo-rel.ofilter r A
shows wo-rel.ofilter (Restr r B) (A Int B)
proof −
  let ?rB = Restr r B
  have WELL: wo-rel r unfolding wo-rel-def using WELL .
hence Refl: Refl r by (simp add: wo-rel.REFL)
hence Field: Field ?rB = Field r Int B
using Refl-Field-Restr by blast
have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
by (simp add: Well-order-Restr wo-rel-def)

show ?thesis using WellB assms
proof(auto simp add: wo-rel.ofilter-def under-def)
  fix a assume a ∈ A and ∗: a ∈ B
  hence a ∈ Field r using OFA Well by (auto simp add: wo-rel.ofilter-def)
  with ∗ show a ∈ Field ?rB using Field by auto
next
  fix a b assume a ∈ A and (b,a) ∈ r
  thus b ∈ A using Well OFA by (auto simp add: wo-rel.ofilter-def under-def)
qed
qed

lemma ofilter-Restr-subset:
assumes WELL: Well-order r and OFA: wo-rel.ofilter r A and SUB: A ≤ B
shows wo-rel.ofilter (Restr r B) A
proof −
  have A Int B = A using SUB by blast
  thus ?thesis using assms ofilter-Restr-Int[of r A B] by auto
qed
lemma ofilter-subset-embed:
assumes WELL: Well-order r and
OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows \((A \leq B) = (\text{embed} \ (\text{Restr} r A) \ (\text{Restr} r B) \ \text{id})\)
proof
let \(\tau r A = \text{Restr} r A\) let \(\tau r B = \text{Restr} r B\)
have Well: wo-rel r unfolding wo-rel-def using WELL .
hence Refl: Refl r by (simp add: wo-rel.REFL)
hence FieldA: Field \(\tau r A = \text{Field} \ r \ \text{Int} A\)
using Refl-Field-Restr by blast
have FieldB: Field \(\tau r B = \text{Field} r \ \text{Int} B\)
using Refl-Field-Restr by blast
have WellA: wo-rel \(\tau r A \land \text{Well-order} \ \tau r A\) using WELL
by (simp add: Well-order-Restr wo-rel-def)
have WellB: wo-rel \(\tau r B \land \text{Well-order} \ \tau r B\) using WELL
by (simp add: Well-order-Restr wo-rel-def)
show \(\text{thesis}\)
proof
assume \(*\): \(A \leq B\)
hence wo-rel.ofilter (\(\text{Restr} r B\)) A using assms
by (simp add: ofilter-Restr-subset)
hence embed (\(\text{Restr} \ \tau r B\)) (\(\text{Restr} r B\)) \(\text{id}\)
using WellB ofilter-embed[of \(\tau r B\) \(\tau r A\)] by auto
thus embed (\(\text{Restr} r A\)) (\(\text{Restr} r B\)) \(\text{id}\)
using \(*)\) by (simp add: Restr-subset)
next
assume \(*\): embed (\(\text{Restr} r A\)) (\(\text{Restr} r B\)) \(\text{id}\)
{fix a assume \(**\): \(a \in A\)
hence a \(\in\) Field r using Well OFA by (auto simp add: wo-rel.ofilter-def)
with \(**\) FieldA have a \(\in\) Field \(\tau r A\) by auto
hence a \(\in\) Field \(\tau r B\) using \(*\) WellA embed-Field[of \(\tau r A\) \(\tau r B\) \(\text{id}\)] by auto
hence a \(\in\) B using FieldB by auto
}
thus \(A \leq B\) by blast
qed
qed

lemma ofilter-subset-embedS-iso:
assumes WELL: Well-order r and
OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows \(((A < B) = (\text{embedS} \ (\text{Restr} r A) \ (\text{Restr} r B) \ \text{id})) \land \ ((A = B) = (\text{iso} \ (\text{Restr} r A) \ (\text{Restr} r B) \ \text{id}))\)
proof
let \(\tau r A = \text{Restr} r A\) let \(\tau r B = \text{Restr} r B\)
have Well: wo-rel r unfolding wo-rel-def using WELL .
hence Refl: Refl r by (simp add: wo-rel.REFL)
hence Field \(\tau r A = \text{Field} r \ \text{Int} A\)
using \texttt{Refl-Field-Restr} by \texttt{blast}

hence \texttt{FieldA}: Field \(?rA = A\) using \texttt{OFA Well}

by (\texttt{auto simp add: wo-rel.ofilter-def})

have \texttt{Field \(?rB = Field r Int B\)}

using \texttt{Refl-Field-Restr} by \texttt{blast}

hence \texttt{FieldB}: Field \(?rB = B\) using \texttt{OFB Well}

by (\texttt{auto simp add: wo-rel.ofilter-def})

show \(?thesis unfolding embedS-def iso-def\)

using assms ofilter-subset-embed[of \(A B\)]

\texttt{FieldA FieldB bij-betw-id-iff[of \(A B\)] by auto}

qed

\textbf{lemma} ofilter-subset-embedS:

\texttt{assumes WELL: Well-order \(r\) and}

\texttt{OFA: wo-rel.ofilter \(r A\) and OFB: wo-rel.ofilter \(r B\)}

\texttt{shows \((A < B) = embedS (Restr r A) (Restr r B) id\)}

using assms by (\texttt{simp add: ofilter-subset-embedS-iso})

\textbf{lemma} embed-implies-iso-Restr:

\texttt{assumes WELL: Well-order \(r\) and WELL': Well-order \(r'\) and}

\texttt{EMB: embed \(r' f\)}

\texttt{shows iso \(r' (Restr r (f' (Field r'))) f\)}

\textbf{proof—}

\texttt{let \(?A' = Field r'\)}

\texttt{let \(?r'' = Restr r (f' \(?A')\)}

\texttt{have \(0:\) Well-order \(?r''\) using WELL Well-order-Restr by blast}

\texttt{have \(1:\) wo-rel.ofilter \(r (f' \(?A')\) using assms embed-Field-ofilter by blast}

\texttt{hence Field \(?r'' = f' (Field r')\) using WELL Field-Restr-ofilter by blast}

\texttt{hence bij-betw \(f \?A' (Field \?r'')\)}

\texttt{using EMB embed-inj-on WELL' unfolding bij-betw-def by blast}

moreover

\{ \texttt{have \(\forall a b. (a,b) \in r' \rightarrow a \in Field r' \land b \in Field r'\)}

\texttt{unfolding Field-def by auto}

\texttt{hence compat \(r' \?r'' f\)}

\texttt{using assms embed-iff-compat-inj-on-ofilter}

\texttt{unfolding compat-def by blast}

\}

ultimately show \(?thesis using WELL' 0 iso-iff3 by blast\)

qed

\textbf{30.3 The strict inclusion on proper ofilters is well-founded}

\textbf{definition} ofilterIncl :: \('a rel \Rightarrow 'a set rel\)

\textbf{where}

\texttt{ofilterIncl \(r \equiv \{(A,B), wo-rel.ofilter \(r A \land A \neq Field r \land\)}

\texttt{wo-rel.ofilter \(r B \land B \neq Field r \land A < B\}\}}

\textbf{lemma} wf-ofilterIncl:
assumes WELL: Well-order r
shows \( \text{wf}(\text{oFilterIncl } r) \)
proof –
  have Well: wo-rel r using WELL by (simp add: wo-rel-def)
  hence Lo: Linear-order r by (simp add: wo-LIN)
  let \( ?h = (\lambda A. \text{wo-rel } r A) \)
  let \( ?rS = r - \text{Id} \)
  have \( \text{wf } ?rS \) using WELL by (simp add: order-on-defs)
  moreover have compat (oFilterIncl r) ?rS ?h
proof (unfold compat-def oFilterIncl-def,
  intro allI impI, simp, elim conjE)
fix A B
assumes *: wo-rel.ofilter r A A \( \neq \) Field r and
  **: wo-rel.ofilter r B B \( \neq \) Field r and ***: A < B
then obtain a and b where 0: a \( \in \) Field r \( \land \) b \( \in \) Field r and
  1: A = underS r a \( \land \) B = underS r b
using Well by (auto simp add: wo-rel.ofilter-underS-Field)
  hence a \( \neq \) b using *** by auto
  moreover have \( (a,b) \in r \) using 0 1 Lo ***
  by (auto simp add: underS-incl-iff)
  moreover have a = wo-rel.suc r A \( \land \) b = wo-rel.suc r B
  using Well 0 1 by (simp add: wo-rel.suc-underS)
  ultimately show (wo-rel.suc r A, wo-rel.suc r B) \( \in \) r \( \land \) wo-rel.suc r A \( \neq \) wo-rel.suc r B
  by simp
qed
ultimately show \( \text{wf } (\text{oFilterIncl } r) \) by (simp add: compat-wf)
qed

30.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- \( \text{ordLeq} \), of being embedded (abbreviated \( \leq o \));
- \( \text{ordLess} \), of being strictly embedded (abbreviated \( < o \));
- \( \text{ordIso} \), of being isomorphic (abbreviated \( = o \)).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio \( \leq, <, = \) associated to a total order on a set.

definition \( \text{ordLeq} :: (\text{'a rel } \times \text{'a rel}) \Rightarrow \text{set} \)
where \( \text{ordLeq} = \{ (r,r'). \text{ Well-order } r \land \text{ Well-order } r' \land (\exists f. \text{ embed } r r' f) \} \)
abbreviation ordLeq2 :: 'a rel ⇒ 'a rel ⇒ bool (infix <= o 50)
where r <= o r' ≡ (r,r') ∈ ordLeq

abbreviation ordLeq3 :: 'a rel ⇒ 'a rel ⇒ bool (infix <= o 50)
where r <= o r' ≡ r <= o r'

definition ordLess :: (('a rel o ('a rel)) set
where ordLess = {(r,r'). Well-order r ∧ Well-order r' ∧ (∃ f. embedS r r' f)}

abbreviation ordLess2 :: 'a rel ⇒ 'a rel ⇒ bool (infix <= o 50)
where r <= o r' ≡ (r,r') ∈ ordLess

definition ordIso :: (('a rel o ('a rel)) set
where ordIso = {(r,r'). Well-order r ∧ Well-order r' ∧ (∃ f. iso r r' f)}

abbreviation ordIso2 :: 'a rel ⇒ 'a rel ⇒ bool (infix <= o 50)
where r = o r' ≡ (r,r') ∈ ordIso

lemmas ordRels-def = ordLeq-def ordLess-def ordIso-def

lemma ordLeq-Well-order-simp:
assumes r <= o r'
shows Well-order r ∧ Well-order r'
using assms unfolding ordLeq-def by simp

Notice that the relations <= o, < o, = o connect well-orders on potentially
distinct types. However, some of the lemmas below, including the next one,
restrict implicitly the type of these relations to (('a rel o ('a rel)) set , i.e.,
to 'a rel rel.

lemma ordLeq-reflexive:
Well-order r ⇒ r <= o r
unfolding ordLeq-def using id-embed[of r] by blast

lemma ordLeq-transitive[trans]:
assumes *: r <= o r' and **: r' <= o r''
shows r <= o r''
proof −
  obtain f and f'
  where 1: Well-order r ∧ Well-order r' ∧ Well-order r'' and
  embed r r' f and embed r' r'' f'
  using * ** unfolding ordLeq-def by blast
  hence embed r r'' (f' o f)
  using comp-embed[of r r' f r'' f'] by auto
  thus r <= o r'' unfolding ordLeq-def using 1 by auto
qed

lemma ordLeq-total:
THEORY “BNF-Constructions-on-Wellorders”

\[
\text{Well-order } r, \text{ Well-order } r' \implies r \leq o r' \lor r' \leq o r
\]

unfolding ordLeq-def using wellorders-totally-ordered by blast

lemma ordIso-reflexive:

Well-order \( r \implies r = o r \)

unfolding ordIso-def using id-iso[of \( r \)] by blast

lemma ordIso-transitive[trans]:

assumes \(*: \) \( r = o r' \) and \(*\!\!\!: \) \( r' = o r'' \)

shows \( r = o r'' \)

proof –

obtain \( f \) and \( f' \)

where 1: Well-order \( r \) \& Well-order \( r' \) \& Well-order \( r'' \) and

iso \( r \: r' \: f \) and 3: iso \( r' \: r'' \: f' \)

using \(*\!\!\!\!*\) unfolding ordIso-def by auto

hence iso \( r \: r'' \: (f' \circ o f) \)

using comp-iso[of \( r \: r' \: f \: r'' \: f' \)] by auto

thus \( r = o r'' \) unfolding ordIso-def using \( f \) by auto

qed

lemma ordIso-symmetric:

assumes \(*: \) \( r = o r' \)

shows \( r' = o r \)

proof –

obtain \( f \) where 1: Well-order \( r \) \& Well-order \( r' \) and

2: embed \( r \: r' \: f \) \& bij-betw \( f \) (Field \( r \)) (Field \( r' \))

using \(*\) by (auto simp add: ordIso-def iso-def)

let \(?f'\!\!\!: = \) inv-into (Field \( r \)) \( f \)

have embed \( r \: r' \: ?f' \) \& bij-betw ?f' (Field \( r' \)) (Field \( r \))

using 1 2 by (simp add: bij-betw-inv-into inv-into-Field-embed-bij-betw)

thus \( r' = o r \) unfolding ordIso-def using \( f \) by (auto simp add: iso-def)

qed

lemma ordLeq-ordLess-trans[trans]:

assumes \( r \leq o r' \) and \( r' < o r'' \)

shows \( r < o r'' \)

proof –

have Well-order \( r \) \& Well-order \( r'' \)

using assms unfolding ordLeq-def ordLess-def by auto

thus \(?\text{thesis}\) using assms unfolding ordLeq-def ordLess-def

using embed-comp-embedS by blast

qed

lemma ordLess-ordLeq-trans[trans]:

assumes \( r < o r' \) and \( r' \leq o r'' \)

shows \( r < o r'' \)

proof –

have Well-order \( r \) \& Well-order \( r'' \)

using assms unfolding ordLeq-def ordLess-def by auto
thus \[\text{thesis using asms unfolding ordLeq-def ordLess-def}\]
using embedS-comp-embed by blast
qed

lemma ordLeq-ordIso-trans[trans]:
assumes \[r \leq o \ r' \text{ and } r' = o \ r''\]
shows \[r \leq o \ r''\]
proof –
  have \[\text{Well-order } r \land \text{Well-order } r''\]
  using asms unfolding ordLeq-def ordIso-def by auto
  thus \[\text{thesis using asms unfolding ordLeq-def ordIso-def}\]
  using embed-comp-iso by blast
qed

lemma ordIso-ordLeq-trans[trans]:
assumes \[r = o \ r' \text{ and } r' \leq o \ r''\]
shows \[r \leq o \ r''\]
proof –
  have \[\text{Well-order } r \land \text{Well-order } r''\]
  using asms unfolding ordLeq-def ordIso-def by auto
  thus \[\text{thesis using asms unfolding ordLeq-def ordIso-def}\]
  using iso-comp-embed by blast
qed

lemma ordLess-ordIso-trans[trans]:
assumes \[r < o \ r' \text{ and } r' = o \ r''\]
shows \[r < o \ r''\]
proof –
  have \[\text{Well-order } r \land \text{Well-order } r''\]
  using asms unfolding ordLeq-def ordIso-def by auto
  thus \[\text{thesis using asms unfolding ordLeq-def ordIso-def}\]
  using embed-comp-iso by blast
qed

lemma ordIso-ordLess-trans[trans]:
assumes \[r = o \ r' \text{ and } r' < o \ r''\]
shows \[r < o \ r''\]
proof –
  have \[\text{Well-order } r \land \text{Well-order } r''\]
  using asms unfolding ordLess-def ordIso-def by auto
  thus \[\text{thesis using asms unfolding ordLeq-def ordIso-def}\]
  using embed-comp-iso by blast
qed

lemma ordLess-not-embed:
assumes \[r < o \ r'\]
shows \[\neg (\exists f'.\text{ embed } r' \ r f')\]
proof –
obtain f where 1: \[\text{Well-order } r \land \text{Well-order } r' \text{ and } 2: \text{ embed } r' \ r f'\] and
3: \( \neg \text{bij-betw } f \text{ (Field } r \text{)} \text{ (Field } r' \text{)} \)

using assms unfolding ordLess-def by (auto simp add: embedS-def)

{fix } f'\{ assume } \ast:\text{ embed } r' \text{ } r \text{ } f' \n
hence \( \text{bij-betw } f \text{ (Field } r \text{)} \text{ (Field } r' \text{)} \) using 1 2

by (simp add: embed-bothWays-Field-bij-betw)

with \( \ast \) have False by contradiction

}\{ thus ?thesis by blast \}

qed

lemma ordLess-Field:

assumes OL: \( r1 <_o r2 \) and EMB: embed \( r1 \) \( r2 \) \( f \)

shows \( \neg (f' \text{ (Field } r1 \text{)} = \text{ Field } r2 \)

proof–

let \( ?A1 = \text{ Field } r1 \) let \( ?A2 = \text{ Field } r2 \)

obtain g where

\( \theta: \text{ Well-order } r1 \land \text{ Well-order } r2 \) and

\( 1: \text{ embed } r1 \text{ } r2 \text{ } g \land \neg (\text{bij-betw } g \text{ } ?A1 \text{ } ?A2) \)

using OL unfolding ordLess-def by (auto simp add: embedS-def)

hence \( \forall a \in ?A1. \text{ } f a = g a \)

using \( \theta \) EMB embed-unique[of \( r1 \)] by auto

hence \( \neg (\text{bij-betw } f \text{ } ?A1 \text{ } ?A2) \)

using \( 1 \) bij-betw-cong[of \( ?A1 \)] by blast

moreover

have \( \text{inj-on } f \text{ } ?A1 \) using EMB \( \theta \) by (simp add: embed-inj-on)

ultimately show ?thesis by (simp add: bij-betw-def)

qed

lemma ordLess-iff:

\( r \triangleleft_o r' = (\text{Well-order } r \land \text{ Well-order } r' \land \neg (\exists f'. \text{ embed } r' \text{ } r \text{ } f')) \)

proof

assume \( \ast: r \triangleleft_o r' \)

hence \( \neg (\exists f'. \text{ embed } r' \text{ } r \text{ } f') \) using ordLess-not-embed[of \( r \text{ } r' \)] by simp

with \( \ast \) show \( \text{Well-order } r \land \text{ Well-order } r' \land \neg (\exists f'. \text{ embed } r' \text{ } r \text{ } f') \)

unfolding ordLess-def by auto

next

assume \( \ast: \text{ Well-order } r \land \text{ Well-order } r' \land \neg (\exists f'. \text{ embed } r' \text{ } r \text{ } f') \)

then obtain \( f \) where

\( 1: \text{ embed } r \text{ } r' \) \( f \)

using wellorders-totally-ordered[of \( r \text{ } r' \)] by blast

moreover

{assume \( \text{bij-betw } f \text{ (Field } r \text{)} \text{ (Field } r' \text{)} \)

with \( \ast \) have \( \text{embed } r' \text{ } r \text{ } (\text{inv-into } \text{ (Field } r \text{)} \text{ } f) \)

using inv-into-Field-embed-bij-betw[of \( r \text{ } r' \text{ } f] \) by auto

with \( \ast \) have False by blast

}

ultimately show \( (r,r') \in \text{ordLess} \)

unfolding ordLess-def using \( \ast \) by (fastforce simp add: embedS-def)

qed
lemma `ordLess-irreflexive`: \( \neg r < o r \)
proof
  assume \( r < o r \)
  hence \( \text{Well-order } r \land \neg(\exists f. \text{ embed } r f) \)
  unfolding `ordLess-iff` ..
  moreover have \( \text{ embed } r r \text{id using id-embed[of } r \) .
  ultimately show \( \text{False by blast} \)
qed

lemma `ordLeq-iff-ordLess-or-ordIso`:
\( r \leq o r' = (r < o r' \lor r = o r') \)
unfolding `ordRels-def embedS-defs iso-defs` by blast

lemma `ordIso-iff-ordLeq`:
\( (r= o r') = (r \leq o r' \land r' \leq o r) \)
proof
  assume \( r = o r' \)
  then obtain \( f \) where \( 1: \text{Well-order } r \land \text{Well-order } r' \land \text{embed } r r' f \land \text{bij-betw } f (\text{Field } r) (\text{Field } r') \)
  unfolding `ordIso-def iso-defs` by auto
  hence \( \text{iso } r r' f \land \text{embed } r' r (\text{inv-into } (\text{Field } r) f) \)
  by (simp add: inv-into-Field-embed-bij-betw)
  thus \( r \leq o r' \land r' \leq o r \)
  unfolding `ordLeq-def` using \( 1 \) by auto
next
  assume \( r \leq o r' \land r' \leq o r \)
  then obtain \( f \) and \( g \) where \( 1: \text{Well-order } r \land \text{Well-order } r' \land \text{embed } r r' f \land \text{embed } r' r g \)
  unfolding `ordLeq-def` by auto
  hence \( \text{iso } r r' f \) by (auto simp add: embed-bothWays-iso)
  thus \( r = o r' \) unfolding `ordIso-def` using \( 1 \) by auto
qed

lemma `not-ordLess-ordLeq`:
\( r < o r' \Rightarrow \neg r' \leq o r \)
using `ordLess-ordLeq-trans` `ordLess-irreflexive` by blast

lemma `ordLess-or-ordLeq`:
assumes \( \text{WELL}: \text{Well-order } r \) and \( \text{WELL'}: \text{Well-order } r' \)
shows \( r < o r' \lor r' \leq o r \)
proof
  have \( r \leq o r' \lor r' \leq o r \)
  using assms by (simp add: ordLeq-total)
  moreover
  {assume \( \neg r < o r' \land r \leq o r' \)
    hence \( r = o r' \) using `ordLeq-iff-ordLess-or-ordIso` by blast
    hence \( r' \leq o r \) using ordIso-symmetric `ordIso-iff-ordLeq` by blast
  }
  ultimately show \( \text{?thesis by blast} \)
qed

lemma not-ordLess-ordIso :  
\( r <_o r' \implies \neg r =_o r' \)  
using assms ordLess-ordIso-trans ordIso-symmetric ordLess-irreflexive by blast

lemma not-ordLess-iff-ordLess :  
assumes WELL: Well-order r and WELL': Well-order r'  
shows (\neg r' \leq_o r) = (r <_o r')  
using assms not-ordLess-ordLeq ordLess-or-ordLeq by blast

lemma ordLess-transitive[trans]:  
[\[ r <_o r'; r' < r'' \] \implies r <_o r'' ]  
using assms ordLess-or-ordLeq-trans ordLeq-iff-ordLess-or-ordIso by blast

corollary ordLess-trans: trans ordLess  
unfolding trans-def using ordLess-transitive by blast

lemmas ordIso-equivalence = ordIso-transitive ordIso-reflexive ordIso-symmetric

lemma ordIso-imp-ordLeq:  
\( r =_o r' \implies r \leq o r' \)  
using ordIso-iff-ordLeq by blast

lemma ordLess-imp-ordLeq:  
\( r <_o r' \implies r \leq o r' \)  
using ordLeq-iff-ordLess-or-ordIso by blast

lemma ofilter-subset-ordLeq:  
assumes WELL: Well-order r and  
OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B  
shows (A \leq B) = (Restr r A \leq_o Restr r B)  
proof  
assume A \leq B  
thus Restr r A \leq_o Restr r B  
unfolding ordLeq-def using assms  
Well-order-Restr Well-order-Restr ofilter-subset-embed by blast

next  
assume *: Restr r A \leq_o Restr r B  
then obtain f where embed (Restr r A) (Restr r B) f  
unfolding ordLeq-def by blast  
{assume B < A  
hence Restr r B <_o Restr r A  
unfolding ordLess-def using assms

proof
Well-order-Restr Well-order-Restr ofilter-subset-embed by blast
hence False using * not-ordLess-ordLeq by blast
}

thus \(A \leq B\) using OFA OFB WELL
wo-rel-def[of r] wo-rel.ofilter-linord[of r A B] by blast
qed

lemma ofilter-subset-ordLess:
assumes WELL: Well-order r and
OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows \((A < B) = (\text{Restr } r A < o \text{ Restr } r B)\)
proof −
let \(?rA = \text{Restr } r A\) let \(?rB = \text{Restr } r B\)
have 1: Well-order \(?rA\) ∧ Well-order \(?rB\)
using WELL Well-order-Restr by blast
have \((A < B) = (\neg B \leq A)\) using assms
wo-rel-def wo-rel.ofilter-linord[of r A B] by blast
also have \(\ldots = (\neg \text{Restr } r B \leq o \text{ Restr } r A)\)
using assms ofilter-subset-ordLeq by blast
also have \(\ldots = (\text{Restr } r A < o \text{ Restr } r B)\)
using 1 not-ordLeq-iff-ordLess by blast
finally show \(?thesis\).
qed

lemma ofilter-ordLess:
\[
\text{[WELL: Well-order r \ and \ OFA: wo-rel.ofilter r A \ and \ OFB: wo-rel.ofilter r B]}
\Rightarrow (A < Field r) = (\text{Restr } r A < o r)
\]
by (simp add: ofilter-subset-ordLess wo-rel.Field-ofilter
wo-rel-def Restr-Field)

corollary underS-Restr-ordLess:
assumes Well-order r and Field r ≠ \{\}
shows Restr r (underS r a) < o r
proof −
have underS r a < Field r using assms
by (simp add: underS-Field3)
thus \(?thesis\) using assms
by (simp add: ofilter-ordLess wo-rel.underS-ofilter wo-rel-def)
qed

lemma embed-ordLess-ofilterIncl:
assumes OL12: \(r1 < o r2\) and OL23: \(r2 < o r3\) and
EMB13: embed r1 r3 f13 and EMB23: embed r2 r3 f23
shows \((f13' (\text{Field } r1), f23' (\text{Field } r2)) \in (\text{ofilterIncl } r3)\)
proof −
have OL13: \(r1 < o r3\)
using OL12 OL23 using ordLess-transitive by auto
let \(?A1 = \text{Field } r1\) let \(?A2 = \text{Field } r2\) let \(?A3 = \text{Field } r3\)
obtain \(f12 g23\) where
\begin{verbatim}
0: Well-order r1 \land Well-order r2 \land Well-order r3 and
1: embed r1 r2 f12 \land \neg (bij-betw f12 ?A1 ?A2) and
2: embed r2 r3 g23 \land \neg (bij-betw g23 ?A2 ?A3)

using OL12 OL23 by (auto simp add: ordLess-def embedS-def)

hence \forall a \in ?A2. f12 a \neq g23 a

using EMB23 embed-unique[of r2 r3] by blast

hence 3: \neg (bij-betw f23 ?A2 ?A3)

using 2 bij-betw-cong[of \{r2 r3\}] by blast

have 4: wo-rel.ofilter r2 (f12 \circ \{?A1\}) \land f12 \circ \{?A1\} \neq ?A2

using 0 1 OL12 by (simp add: embed-field-ofilter ordLess-Field)

have 5: wo-rel.ofilter r3 (f23 \circ \{?A2\}) \land f23 \circ \{?A2\} \neq ?A3

using 0 EMB23 OL23 by (simp add: embed-field-ofilter ordLess-Field)

have 6: wo-rel.ofilter r3 (f13 \circ \{?A1\}) \land f13 \circ \{?A1\} \neq ?A3

using 0 EMB13 OL13 by (simp add: embed-field-ofilter ordLess-Field)

have f12 \circ \{?A1\} < f13 \circ \{?A1\}

using 0 4 by (auto simp add: wo-rel-def wo-rel.ofilter-def)

moreover have inj-on f23 ?A2

using EMB23 0 by (simp add: wo-rel-def embed-inj-on)

ultimately have f23 \circ (f12 \circ \{?A1\}) = f13 \circ \{?A1\}

moreover

\{ have embed r1 r3 (f23 \circ f12)

using 1 EMB23 0 by (auto simp add: comp-embed)

hence \forall a \in ?A1. f23(f12 a) = f13 a

using EMB13 0 embed-unique[of r1 r3 f23 o f12 f13] by auto

hence f23 \circ (f12 \circ \{?A1\}) = f13 \circ \{?A1\} by force

\}

ultimately have f13 \circ \{?A1\} < f23 \circ \{?A2\} by simp

with 5 6 show ?thesis

unfolding ofilterIncl-def by auto

qed

lemma ordLess-iff-ordIso-Restr:

assumes WELL: Well-order r and WELL': Well-order r'

shows (r' <o r) = (\exists a \in Field r. r' =o Restr r (underS r a))

proof(auto)

fix a assume *: a \in Field r and **: r' =o Restr r (underS r a)

hence Restr r (underS r a) <o r using WELL underS-Restr-ordLess[of r] by blast

thus r' <o r using ** ordIso-ordLess-trans by blast

next

assume r' <o r

then obtain f where 1: Well-order r \land Well-order r' and

2: embed r' r f \land f \circ (Field r') \neq Field r

unfolding ordLess-def embedS-def[abs-def] bij-betw-def using embed-inj-on by
\end{verbatim}
THEORY "BNF-Constructions-on-Wellorders"

blast
hence wo-rel.ofilter r (f' (Field r')) using embed-Field-ofilter by blast
then obtain a where 3: a ∈ Field r and 4: underS r a = f' (Field r')
using 1 2 by (auto simp add: wo-rel.ofilter-underS-Field wo-rel-def)
have iso r' (Restr r (f' (Field r')))) using blast
then obtain a where 3:
 a ∈ Field r and 4: underS r a = f' (Field r')
using 1 2 by (auto simp add: wo-rel.ofilter-underS-Field wo-rel-def)

moreover have Well-order (Restr r (f' (Field r'))))
using WELL Well-order-Restr by blast
ultimately have r' = Restr r (underS r a) using 4 by auto
thus ∃a ∈ Field r. r' = Restr r (underS r a) using 3 by auto
qed

lemma internalize-ordLess:
(r' < o r) = (∃p. Field p < Field r r' = o p ∧ p < o r)
proof
assume *: r' < o r

hence 0: Well-order r ∧ Well-order r' unfolding ordLess-def by auto
with * obtain a where 1: a ∈ Field r and 2: r' = o Restr r (underS r a)
using ordLess-iff-ordIso-Restr by blast
let ?p = Restr r (underS r a)
have wo-rel.ofilter r (underS r a) using 0
by (simp add: wo-rel-def wo-rel.underS-ofilter)

hence Field ?p = underS r a using 0 Field-Restr-ofilter by blast
hence Field ?p < Field r using underS-Field2 1 by fast
moreover have ?p < o r using underS-Restr-ordLess[of r a] 0 1 by blast
ultimately
show ∃p. Field p < Field r r' = o p ∧ p < o r using 2 by blast
next
assume ∃p. Field p < Field r r' = o p ∧ p < o r
thus r' < o r using ordIso-ordLess-trans by blast
qed

lemma internalize-ordLeq:
(r' ≤ o r) = (∃p. Field p ≤ Field r r' = o p ∧ p ≤ o r)
proof
assume *: r' ≤ o r

moreover {assume r' < o r
then obtain p where Field p < Field r r' = o p ∧ p < o r
using internalize-ordLess[of r' r] by blast
hence ∃p. Field p < Field r r' = o p ∧ p < o r
using ordLeq-iff-ordLess-or-ordIso by blast
}
moreover
have r ≤ o r using * ordLeq-def ordLeq-reflexive by blast
ultimately show ∃p. Field p ≤ Field r r' = o p ∧ p ≤ o r
using ordLeq-iff-ordLess-or-ordIso by blast
next
  assume \( \exists p. \text{Field } p \leq \text{Field } r \land r' =_o p \land p \leq_o r \)
  thus \( r' \leq_o r \) using ordIso-ordLeq-trans by blast
qed

lemma ordLeq-iffLess-Restr:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \)
shows \( (r \leq_o r') = (\forall a \in \text{Field } r. \text{Restr } r \ (\text{underS } r \ a) <_o r') \)
proof (auto)
  assume \( \ast: r \leq_o r' \)
  fix \( a \) assume \( a \in \text{Field } r \)
  hence \( \text{Restr } r \ (\text{underS } r \ a) <_o r \)
  using WELL underS-Restr-ordLess[of \( r \)] by blast
  thus \( \text{Restr } r \ (\text{underS } r \ a) <_o r' \)
  using \( \ast \) ordLess-ordLeq-trans by blast
next
  assume \( \ast: \forall a \in \text{Field } r. \text{Restr } r \ (\text{underS } r \ a) <_o r' \)
  { assume \( r' <_o r \)
    then obtain \( h \) where \( h \in \text{Field } r' \land h \ (\text{Field } r') \leq \text{Field } r \)
    unfolding ordLeq-def using assms embed-inj-on embed-Field by blast
    hence \( \text{False} \) using \( \ast \) not-ordLess-ordIso ordIso-symmetric by blast
  }
  thus \( r \leq_o r' \) using ordLess-or-ordLeq assms by blast
qed

lemma finite-ordLess-infinite:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and
FIN: finite \( \text{Field } r \) and INF: \( \neg \)finite \( \text{Field } r' \)
shows \( r <_o r' \)
proof −
  { assume \( r' \leq_o r \)
    then obtain \( h \) where \( \text{inj-on } h \ (\text{Field } r') \land h \ (\text{Field } r') \leq \text{Field } r \)
    unfolding ordLeq-def using assms embed-inj-on embed-Field by blast
    hence \( \text{False} \) using finite-imageD finite-subset FIN INF by blast
  }
  thus \( \ast \)thesis using WELL WELL' ordLess-or-ordLeq assms by blast
qed

lemma finite-well-order-on-ordIso:
assumes FIN: finite \( A \) and
WELL: well-order-on \( A \) \( r \) and WELL': well-order-on \( A \) \( r' \)
shows \( r =_o r' \)
proof −
  have \( \theta: \text{Well-order } r \land \text{Well-order } r' \land \text{Field } r = A \land \text{Field } r' = A \)
  using assms well-order-on-Well-order by blast
  moreover
  have \( \forall r. \text{well-order-on } A \ r \land \text{well-order-on } A \ r' \land r \leq_o r' \rightarrow r =_o r' \)
  proof (clarify)
fix \( r \) \( r' \) assume \(*\): well-order-on \( A \) \( r \) and \(**\): well-order-on \( A \) \( r' \)

have \( 2: \) Well-order \( r \) \( \land \) Well-order \( r' \) \( \land \) Field \( r \) = \( A \) \( \land \) Field \( r' \) = \( A \)

using \(*\)** well-order-on-Well-order by blast

assume \( r \leq_o r' \)

then obtain \( f \) where \( 1: \) embed \( r \) \( r' \) \( f \) and inj-on \( f \) \( A \) \( \land \) \( f' \) \( A \) \( \leq \) \( A \)

unfolding ordLeq-def using \( \phi \) embed-inj-on embed-Field by blast

hence bij-betw \( f \) \( A \) \( A \)

unfolding bij-betw-def using \( \phi \) FIN endo-inj-surj by blast

thus \( r = o r' \) unfolding ordIso-def iso-def \( \phi \)

[abs-def] using \( 1 \) \( 2 \) by auto

qed

ultimately show \( \phi \) thesis using assms ordLeq-total ordIso-symmetric by blast

qed

30.5 \(<_o \) is well-founded

Of course, it only makes sense to state that the \(<_o \) is well-founded on the restricted type \( ' a \) rel rel. We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to \(<_o \) versus strict inclusion; and we already know that strict inclusion of order filters is well-founded.

definition ord-to-filter :: \( ' a \) rel \( \Rightarrow \) \( ' a \) rel \( \Rightarrow \) \( ' a \) set

where ord-to-filter \( r0 \) \( r \)

\( \equiv ( \text{SOME } f. \text{ embed } r \ r0 \ f) \ ' ( \text{Field } r) \)

lemma ord-to-filter-compat:

compat (ordLess Int (ordLess \( ^{-1} \)) \( \times \) \( \times \) \( \mathcal{O} \)\( \{ r0 \} \times \mathcal{O} \)\( \{ r0 \} \))

(ofilterIncl \( r0 \))

(ord-to-filter \( r0 \))

proof (unfold compat-def ord-to-filter-def, clarify)

fix \( r1 :: ' a \) rel and \( r2 :: ' a \) rel

let \( ?A1 = \text{Field } r1 \)

let \( ?A2 = \text{Field } r2 \)

let \( ?A0 = \text{Field } r0 \)

let \( ?phi10 = \lambda \ f10. \ \text{embed } r1 \ r0 \ f10 \)

let \( ?f10 = \text{SOME } f. \ ?phi10 f \)

let \( ?phi20 = \lambda \ f20. \ \text{embed } r2 \ r0 \ f20 \)

let \( ?f20 = \text{SOME } f. \ ?phi20 f \)

assume \(*\): \( r1 <_o r0 \) \( r2 <_o r0 \) \( \land \) \( **\): \( r1 <_o r2 \)

hence \( ( \exists f. \ ?phi10 f) \land ( \exists f. \ ?phi20 f) \)

by (auto simp add: ordLess-def embedS-def)

hence \( ?phi10 \ ?f10 \land ?phi20 \ ?f20 \) by (auto simp add: someI-ex)

thus \( ( ?f10 \cdot ?A1, \ ?f20 \cdot \ ?A2) \in \text{ofilterIncl } r0 \)

using \(*\)** by (simp add: embed-ordLess-ofilterIncl)

qed

theorem wf-ordLess: wf ordLess

proof –

{fix \( r0 :: (\ ' a \times ' a ) \) set

let \( ?ordLess = \text{ordLess}:: (\ ' d \ rel \ ' d \ rel ) \) set

let \( ?R = ?ordLess \ \text{Int} ( \ ?ordLess \ ^{-1} \ \mathcal{O} \ \times \ ?ordLess \ ^{-1} \ \mathcal{O} \) \)

{assume Case1: Well-order \( r0 \)
hence \( \text{wf } ?R \)

using \( \text{wf-ofilterIncl[of r0]} \)

\( \text{compat-wf[of } ?R \text{ ofilterIncl r0 ord-to-filter r0]} \)

\( \text{ord-to-filter-compat[of r0] by auto} \)

}\n
moreover

\{ assume \( \text{Case2: } \neg \text{ Well-order r0} \)

hence \( ?R = \{\} \) unfolding \( \text{ordLess-def} \) by auto

hence \( \text{wf } ?R \) using \( \text{wf-empty} \) by simp

}\n
ultimately have \( \text{wf } ?R \) by blast

}\n
thus \( ?\text{thesis} \) by (simp add: \( \text{trans-wf-iff ordLess-trans} \))

qed

corollary \( \text{exists-minim-Well-order:} \)

assumes \( \text{NE: } R \neq \{\} \) and \( \text{WELL: } \forall r \in R. \text{ Well-order } r \)

shows \( \exists r \in R. \forall r' \in R. r \leq o r' \)

proof –

obtain \( r \) where \( r \in R \land (\forall r' \in R. \neg r' < o r) \)

using \( \text{NE spec[OF spec[OF subst[OF wf-eq-minimal, of } %x. x, OF wf-ordLess]], of - R]} \)

\( \text{equals0I[of R] by blast} \)

with \( \text{not-ordLeq-iff-ordLess WELL show } ?\text{thesis} \) by blast

qed

30.6 Copy via direct images

The direct image operator is the dual of the inverse image operator \( \text{inv-image} \)
from \( \text{Relation.thy}. \) It is useful for transporting a well-order between different

 types.

definition \( \text{dir-image :: } 'a \text{ rel } \Rightarrow (',a \Rightarrow 'a') \Rightarrow 'a' \text{ rel} \)

where

\( \text{dir-image r f} = \{ (f a, f b) | a b. (a,b) \in r \} \)

lemma \( \text{dir-image-Field:} \)

\( \text{Field(dir-image r f) = f • (Field r)} \)

unfolding \( \text{dir-image-def Field-def Range-def Domain-def by fast} \)

lemma \( \text{dir-image-minus-Id:} \)

\( \text{inj-on f (Field r) } \Rightarrow \text{ (dir-image r f) - Id = dir-image (r - Id) f} \)

unfolding \( \text{inj-on-def Field-def dir-image-def by auto} \)

lemma \( \text{Refl-dir-image:} \)

assumes \( \text{Refl r} \)

shows \( \text{Refl(dir-image r f)} \)

proof –

\{ fix a' b' \}

\( \text{assume } (a',b') \in \text{ dir-image r f} \)
then obtain \( a, b \) where
\[
1: a' = f a \land b' = f b \land (a, b) \in r
\]
unfolding \textit{dir-image-def} by blast
hence \( a \in \text{Field } r \land b \in \text{Field } r \) using \textit{Field-def} by fastforce
hence \((a, a) \in r \land (b, b) \in r\) using \textit{assms} by (simp add: \textit{refl-on-def})
with \( 1 \) have \((a', a') \in \text{dir-image } r f \land (b', b') \in \text{dir-image } r f\)
unfolding \textit{dir-image-def} by auto

\}
thus \( ?\text{thesis} \)
by (unfold \textit{refl-on-def Field-def Domain-def Range-def}, auto)

\]
lemma Partial-order-dir-image:
[[Partial-order r; inj-on f (Field r)]] \implies Partial-order (dir-image r f)
by (simp add: partial-order-on-def Preorder-dir-image antisym-dir-image)

lemma Total-dir-image:
assumes TOT: Total r and INJ: inj-on f (Field r)
shows Total(dir-image r f)
proof(unfold total-on-def, intro ballI impI)
  fix a' b'
  assume a' \in Field (dir-image r f) b' \in Field (dir-image r f)
  then obtain a and b where I: a \in Field r \land b \in Field r \land f a = a' \land f b = b'
  unfolding dir-image-Field[of r f] by blast
moreover assume a' \neq b'
ultimately have a \neq b using INJ unfolding inj-on-def by auto
hence (a,b) \in r \lor (b,a) \in r using 1 TOT unfolding total-on-def by auto
thus (a',b') \in dir-image r f \lor (b',a') \in dir-image r f
using 1 unfolding dir-image-def by auto
qed

lemma Linear-order-dir-image:
[[Linear-order r; inj-on f (Field r)]] \implies Linear-order (dir-image r f)
by (simp add: linear-order-on-def Partial-order-dir-image Total-dir-image)

lemma wf-dir-image:
assumes WF: wf r and INJ: inj-on f (Field r)
shows wf(dir-image r f)
proof(unfold wf-eq-minimal2, intro allI impI, elim conjE)
  fix A':='b set
  assume SUB: A' \subseteq Field(dir-image r f) and NE: A' \neq {}
  obtain A where A-def: A = {a \in Field r. \ f a \in A'} by blast
  have A \neq {} \land A \subseteq Field r using A-def SUB NE by (auto simp: dir-image-Field)
  then obtain a where I: a \in A \land (\forall b \in A. (b,a) \notin r) using spec[OF WF[unfolded wf-eq-minimal2], of A] by blast
  have \forall b' \in A'. (b',f a) \notin dir-image r f
  proof(clarify)
    fix b'\ assume *: b' \in A' and **: (b',f a) \in dir-image r f
    obtain b1 a1 where 2: b' = f b1 \land f a = f a1 and
      3: (b1,a1) \in r \land \{a1,b1\} \subseteq Field r
    using ** unfolding dir-image-def Field-def by blast
    hence a = a1 using 1 A-def INJ unfolding inj-on-def by auto
    hence b1 \in A \land (b1,a) \in r using 2 3 A-def * by auto
    with I show False by auto
  qed
  thus \exists a'\in A'. \forall b'\in A'. (b', a') \notin dir-image r f
  using A-def I by blast
  qed

lemma Well-order-dir-image:
[[Well-order r; inj-on f (Field r)]] \implies Well-order (dir-image r f)
30.7 Bounded square

This construction essentially defines, for an order relation $r$, a lexicographic order $bsqr$ on $(\text{Field } r) \times (\text{Field } r)$, applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.
The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The essential property required there (and which is ensured by this construction) is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

**definition** bsqr :: 'a rel => ('a * 'a)rel
where
bsqr r = {((a1,a2),(b1,b2)).
          \{a1,a2,b1,b2\} <= Field r \land
          (a1 = b1 \land a2 = b2 ∨
            wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) \in r - Id \land
            wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 \land (a1,b1) \in r - Id \land
            wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) \in r
            - Id
          }\)

**lemma** Field-bsqr:
Field (bsqr r) = Field r × Field r
**proof**
  show Field (bsqr r) ≤ Field r × Field r
  proof
    {fix a1 a2 assume (a1,a2) ∈ Field (bsqr r)
      moreover
      have \land b1 b2. ((a1,a2),(b1,b2)) ∈ bsqr r \lor ((b1,b2),(a1,a2)) ∈ bsqr r \Longrightarrow
        a1 \in Field r \land a2 \in Field r unfolding bsqr-def by auto
      ultimately have a1 ∈ Field r \land a2 ∈ Field r unfolding Field-def by auto
    }
  thus ?thesis unfolding Field-def by force
  qed
next
  show Field r × Field r ≤ Field (bsqr r)
  proof(auto)
    fix a1 a2 assume a1 \in Field r and a2 \in Field r
    hence ((a1,a2),(a1,a2)) \in bsqr r unfolding bsqr-def by blast
    thus (a1,a2) \in Field (bsqr r) unfolding Field-def by auto
  qed
  qed

**lemma** bsqr-Refl: Refl(bsqr r)
by(unfold refl-on-def Field-bsqr, auto simp add: bsqr-def)

**lemma** bsqr-Trans:
assumes Well-order r
shows trans (bsqr r)
**proof**(unfold trans-def, auto)
  have Well: wo-rel r using assms wo-rel-def by auto
hence Trans: trans r using wo-rel.TRANS by auto
have Anti: antisym r using wo-rel.ANTISYM Well by auto
hence TransS: trans(r−Id) using Trans by (simp add: trans-diff-Id)

fix a1 a2 b1 b2 c1 c2
assume *: ((a1,a2),(b1,b2)) ∈ bsqr r and **: ((b1,b2),(c1,c2)) ∈ bsqr r
hence 0: {a1,a2,b1,b2,c1,c2} ≤ Field r unfolding bsqr-def by auto
have 1: a1 = b1 ∧ a2 = b2 ∨ (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r − Id ∨
        wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id ∨
        wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) ∈ r − Id
using * unfolding bsqr-def by auto
have 2: b1 = c1 ∧ b2 = c2 ∨ (wo-rel.max2 r b1 b2, wo-rel.max2 r c1 c2) ∈ r − Id ∨
        wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2 ∧ (b1,c1) ∈ r − Id ∨
        wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2 ∧ b1 = c1 ∧ (b2,c2) ∈ r − Id
using ** unfolding bsqr-def by auto
show ((a1,a2),(c1,c2)) ∈ bsqr r
proof−
{assume Case1: a1 = b1 ∧ a2 = b2
  hence ?thesis using ** by simp }
moreover
{assume Case2: (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r − Id
  assume Case21: b1 = c1 ∧ b2 = c2
  hence ?thesis using * by simp }
moreover
{assume Case22: (wo-rel.max2 r b1 b2, wo-rel.max2 r c1 c2) ∈ r − Id
  hence (wo-rel.max2 r a1 a2, wo-rel.max2 r c1 c2) ∈ r − Id
  using Case2 TransS trans-def[of r − Id] by blast
  hence ?thesis using 0 unfolding bsqr-def by auto }
moreover
{assume Case23-4: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2
  hence ?thesis using Case2 0 unfolding bsqr-def by auto }
ultimately have ?thesis using 0 2 by auto
moreover
{assume Case3: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id
  assume Case31: b1 = c1 ∧ b2 = c2
  hence ?thesis using * by simp }
moreover
{assume Case32: (wo-rel.max2 r b1 b2, wo-rel.max2 r c1 c2) ∈ r − Id
  assume Case321: b1 = c1 ∧ b2 = c2
  hence ?thesis using ** by simp }
moreover
{assume Case33-4: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2
  hence ?thesis using Case3 0 unfolding bsqr-def by auto }
ultimately have ?thesis using 0 2 by auto
moreover
{assume Case4: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id
  assume Case41: b1 = c1 ∧ b2 = c2
  hence ?thesis using * by simp }
hence \( \text{thesis using } \text{Case3 } \text{0 unfolding } \text{bsqr-def} \text{ by auto} \)

moreover

{\{\text{assume Case33: } \text{wo-rel.max2 } r \text{ b1 b2 } = \text{wo-rel.max2 } r \text{ c1 c2 } \land (b1,c1) \in r \}
  - \text{Id}
  \text{hence } (a1,c1) \in r - \text{Id}
  \text{using Case3 TransS trans-def[of r - Id] by blast}
  \text{hence } \text{thesis using } \text{Case3 Case33 } \text{0 unfolding } \text{bsqr-def} \text{ by auto} \}

ultimately have \( \text{thesis using } 0 \text{ 2 by auto} \)

moreover

{\{\text{assume Case4: } \text{wo-rel.max2 } r \text{ a1 a2 } = \text{wo-rel.max2 } r \text{ b1 b2 } \land a1 = b1 \land (a2,b2) \in r - \text{Id}
  \text{assume Case41: } b1 = c1 \land b2 = c2
  \text{hence } \text{thesis using } * \text{ by simp} \}

moreover

{\{\text{assume Case42: } (\text{wo-rel.max2 } r \text{ b1 b2 } , \text{wo-rel.max2 } r \text{ c1 c2}) \in r - \text{Id}
  \text{hence } \text{thesis using } \text{Case4 } \text{0 unfolding } \text{bsqr-def} \text{ by force} \}

ultimately have \( \text{thesis using } 0 \text{ 2 by auto} \)

ultimately show \( \text{thesis using } 0 \text{ 1 by auto} \)

qed

lemma bsqr-antisym:
assumes Well-order r
shows antisym (bsqr r)
proof(\text{unfold antisym-def, clarify})

have Well: wo-rel r using assms wo-rel-def by auto
hence Trans: \(\text{trans } r\) using wo-rel.TRANS by auto

have Anti: \(\text{antisym } r\) using wo-rel.antisym Well by auto

hence TransS: \(\text{trans}(r - Id)\) using Trans by (simp add: trans-diff-Id)

hence IrrS: \(\forall a b. \neg((a,b) \in r - Id \land (b,a) \in r - Id)\)

using Anti trans-def[of r - Id] antisym-def[of r - Id] by blast

fix \(a1\ a2\ b1\ b2\)

assume \(*\): \(((a1,a2),(b1,b2)) \in \text{bsqr } r\) and \(**\): \(((b1,b2),(a1,a2)) \in \text{bsqr } r\)

hence 0: \(\{a1,a2,b1,b2\} \leq \text{Field } r\) unfolding bsqr-def by auto

have 1: \(a1 = b1 \land a2 = b2 \lor (\text{wo-rel.max2 } r a1 a2, \text{wo-rel.max2 } r b1 b2) \in r - Id\)

- \(Id \lor \) wo-rel.max2 \(r a1 a2 = \text{wo-rel.max2 } r b1 b2 \land (a1,b1) \in r - Id \lor \) wo-rel.max2 \(r a1 a2 = \text{wo-rel.max2 } r b1 b2 \land a1 = b1 \land (a2,b2) \in r - Id\)

Id using \(*\) unfolding bsqr-def by auto

have 2: \(b1 = a1 \land b2 = a2 \lor (\text{wo-rel.max2 } r b1 b2, \text{wo-rel.max2 } r a1 a2) \in r - Id\)

- \(Id \lor \) wo-rel.max2 \(r b1 b2 = \text{wo-rel.max2 } r a1 a2 \land (b1,a1) \in r - Id \lor \) wo-rel.max2 \(r b1 b2 = \text{wo-rel.max2 } r a1 a2 \land b1 = a1 \land (b2,a2) \in r - Id\)

Id using \(**\) unfolding bsqr-def by auto

show \(a1 = b1 \land a2 = b2\)

proof

{ assume Case1: \((\text{wo-rel.max2 } r a1 a2, \text{wo-rel.max2 } r b1 b2) \in r - Id\)

  { assume Case11: \((\text{wo-rel.max2 } r b1 b2, \text{wo-rel.max2 } r a1 a2) \in r - Id\)
    hence False using Case1 IrrS by blast
  }

  moreover
  { assume Case12-3: \(\text{wo-rel.max2 } r b1 b2 = \text{wo-rel.max2 } r a1 a2\)
    hence False using Case1 by auto
  }
}

ultimately have ?thesis using 0 2 by auto

} moreover

{ assume Case2: \((\text{wo-rel.max2 } r a1 a2 = \text{wo-rel.max2 } r b1 b2 \land (a1,b1) \in r - Id\)

  { assume Case21: \((\text{wo-rel.max2 } r b1 b2, \text{wo-rel.max2 } r a1 a2) \in r - Id\)
    hence False using Case2 by auto
  }

  moreover
  { assume Case22: \((b1,a1) \in r - Id\)
    hence False using Case2 IrrS by blast
  }

  moreover
  { assume Case23: \(b1 = a1\)
    hence False using Case2 by auto
  }
}

ultimately have ?thesis using 0 2 by auto
moreover
{assume Case3: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) ∈ r − Id
  moreover
  {assume Case31: (wo-rel.max2 r b1 b2, wo-rel.max2 r a1 a2) ∈ r − Id
    hence False using Case3 by auto
  }
  moreover
  {assume Case32: (b1,a1) ∈ r − Id
    hence False using Case3 by auto
  }
  moreover
  {assume Case33: (b2,a2) ∈ r − Id
    hence False using Case3 IrrS by blast
  }
  ultimately have ?thesis using 0 2 by auto
}
moreover
{assume Case31: (wo-rel.max2 r b1 b2, wo-rel.max2 r a1 a2) ∈ r − Id
  hence False using Case3 by auto
}
moreover
{assume Case32: (b1,a1) ∈ r − Id
  hence False using Case3 by auto
}
moreover
{assume Case33: (b2,a2) ∈ r − Id
  hence False using Case3 IrrS by blast
}
ultimately have ?thesis using 0 2 by auto
}
ultimately show ?thesis using 0 1 by blast
qed
qed

lemma bsqr-Total:
assumes Well-order r
sows Total(bsqr r)
proof -
  have Well: wo-rel r using assms wo-rel-def by auto
  hence Total: ∀ a ∈ Field r. ∀ b ∈ Field r. (a,b) ∈ r ∨ (b,a) ∈ r
    using wo-rel.TOTALS by auto
    {fix a1 a2 b1 b2 assume {(a1,a2), (b1,b2)} ≤ Field(bsqr r)
      hence 0: a1 ∈ Field r ∧ a2 ∈ Field r ∧ b1 ∈ Field r ∧ b2 ∈ Field r
        using Field-bsqr by blast
      have ((a1,a2) = (b1,b2) ∨ ((a1,a2),(b1,b2)) ∈ bsqr r ∨ ((b1,b2),(a1,a2)) ∈ bsqr r)
        proof(rule wo-rel.cases-Total[of r a1 a2], clarsimp simp add: Well, simp add: 0)
        assume Case1: (a1,a2) ∈ r
          hence 1: wo-rel.max2 r a1 a2 = a2
            using Well 0 by (simp add: wo-rel.max2-equals2)
        show ?thesis
          proof(rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add: 0)
            assume Case11: (b1,b2) ∈ r
              hence 2: wo-rel.max2 r b1 b2 = b2
                using Well 0 by (simp add: wo-rel.max2-equals2)
            show ?thesis
              proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
add: 0
assume Case11: (a2,b2) ∈ r − Id ∨ (b2,a2) ∈ r − Id
thus ?thesis using 0 1 2 unfolding bsqr-def by auto
next
assume Case112: a2 = b2
show ?thesis

proof
(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
assume Case1121: (a1,b1) ∈ r − Id ∨ (b1,a1) ∈ r − Id
thus ?thesis using 0 1 2 Case112 unfolding bsqr-def by auto
next
assume Case1122: a1 = b1
thus ?thesis using Case112 by auto
qed
qed
next
assume Case12: (b2,b1) ∈ r
hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)
show ?thesis

proof
(rule wo-rel.cases-Total3[of r a2 b1], clarsimp simp add: Well, simp
add: 0)
assume Case121: (a2,b1) ∈ r − Id ∨ (b1,a2) ∈ r − Id
thus ?thesis using 0 1 3 unfolding bsqr-def by auto
next
assume Case122: a2 = b1
show ?thesis

proof
(rule wo-rel.cases-Total3[of r a1 b2], clarsimp simp add: Well, simp
add: 0)
assume Case1221: (a1,b1) ∈ r − Id ∨ (b1,a1) ∈ r − Id
thus ?thesis using 0 1 3 Case122 unfolding bsqr-def by auto
next
assume Case1222: a1 = b1
show ?thesis

proof
(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
assume Case12221: (a2,b2) ∈ r − Id ∨ (b2,a2) ∈ r − Id
thus ?thesis using 0 1 3 Case122 Case1222 unfolding bsqr-def by auto
next
assume Case12222: a2 = b2
thus ?thesis using Case122 Case1222 by auto
qed
qed
qed
next
assume Case2: (a2,a1) ∈ r
hence 1: wo-rel.max2 r a1 a2 = a1 using Well 0 by (simp add: wo-rel.max2-equals1)
show ?thesis
proof (rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add: 0)
  assume Case21: \((b1, b2) \in r\)
  hence 2: wo-rel.max2 r b1 b2 = b2 using Well 0 by (simp add: wo-rel.max2-equals2)
  show ?thesis
    proof (rule wo-rel.cases-Total3[of r a1 b2], clarsimp simp add: Well, simp add: 0)
      assume Case211: \((a1, b2) \in r\)
      thus ?thesis using 0 1 2 unfolding bsqr-def by auto
    next
      assume Case212: \(a1 = b2\)
      show ?thesis
        proof (rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
          assume Case2121: \((a2, b2) \in r\)
          thus ?thesis using 0 1 2 Case212 unfolding bsqr-def by auto
        next
          assume Case2122: \(a2 = b2\)
          thus ?thesis using Case2122 Case212 by auto
        qed
      qed
    next
      assume Case22: \((b2, b1) \in r\)
      hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)
      show ?thesis
        proof (rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
          assume Case221: \((a1, b1) \in r\)
          thus ?thesis using 0 1 3 unfolding bsqr-def by auto
        next
          assume Case222: \(a1 = b1\)
          show ?thesis
            proof (rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
              assume Case2221: \((a2, b2) \in r\)
              thus ?thesis using 0 1 3 Case222 unfolding bsqr-def by auto
            next
              assume Case2222: \(a2 = b2\)
              thus ?thesis using Case222 by auto
            qed
        qed
    qed
next
assume Case22: \((b2, b1) \in r\)
  hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)
  show ?thesis
    proof (rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
      assume Case221: \((a1, b1) \in r\)
      thus ?thesis using 0 1 3 unfolding bsqr-def by auto
    next
      assume Case222: \(a1 = b1\)
      show ?thesis
        proof (rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
          assume Case2221: \((a2, b2) \in r\)
          thus ?thesis using 0 1 3 Case222 unfolding bsqr-def by auto
        next
          assume Case2222: \(a2 = b2\)
          thus ?thesis using Case222 by auto
        qed
        qed
thus thesis unfolding total-on-def by fast qed

lemma bsqr-Linear-order:
assumes Well-order r
shows Linear-order(bsqr r)
unfolding order-on-defs
using assms bsqr-Refl bsqr-Trans bsqr-antisym bsqr-Total by blast

lemma bsqr-Well-order:
assumes Well-order r
shows Well-order(bsqr r)
using assms proof (simp add: bsqr-Linear-order Linear-order-Well-order-iff, intro allI impI)
have 0: ∀ A ≤ Field r. A ≠ {} → (∃ a ∈ A. ∀ a' ∈ A. (a,a') ∈ r)
using assms well-order-on-def Linear-order-Well-order-iff by blast
fix D assume *: D ≤ Field (bsqr r) and **: D ≠ {}
hence 1: D ≤ Field r × Field r unfolding Field-bsqr by simp
obtain M where M-def: M = {wo-rel.max2 r a1 a2| a1 a2. (a1,a2) ∈ D} by blast
have M ≠ {} using 1 M-def ** by auto
moreover have M ≤ Field r unfolding M-def
using 1 assms wo-rel-def[of r] wo-rel.max2-among[of r] by fastforce
ultimately obtain m where m-min: m ∈ M ∧ (∀ a ∈ M. (m,a) ∈ r)
using 0 by blast
obtain A1 where A1-def: A1 = {a1. ∃ a2. (a1,a2) ∈ D ∧ wo-rel.max2 r a1 a2 = m} by blast
have A1 ≤ Field r unfolding A1-def using 1 by auto
moreover have A1 ≠ {} unfolding A1-def using m-min unfolding M-def by blast
ultimately obtain a1 where a1-min: a1 ∈ A1 ∧ (∀ a ∈ A1. (a1,a) ∈ r)
using 0 by blast
obtain A2 where A2-def: A2 = {a2. (a1,a2) ∈ D ∧ wo-rel.max2 r a1 a2 = m} by blast
have A2 ≤ Field r unfolding A2-def using 1 by auto
moreover have A2 ≠ {} unfolding A2-def
using m-min a1-min unfolding A1-def M-def by blast
ultimately obtain a2 where a2-min: a2 ∈ A2 ∧ (∀ a ∈ A2. (a2,a) ∈ r)
using 0 by blast
have 2: wo-rel.max2 r a1 a2 = m
using $a1\text{-min}$ $a2\text{-min}$ unfolding $A1\text{-def}$ $A2\text{-def}$ by auto
have $3: (a1, a2) \in D$ using $a2\text{-min}$ unfolding $A2\text{-def}$ by auto

moreover
{fix $b1$ $b2$ assume $$: (b1, b2) \in D$
  hence $4: \{a1, a2, b1, b2\} \leq \text{Field}(r)$ using $1, 3$ by blast
have $5: (\text{wo-rel}\text{-max2} \ r \ a1 \ a2, \text{wo-rel}\text{-max2} \ r \ b1 \ b2) \in r$
using $*** \text{a1-min a2-min m-min unfolding} \ A1\text{-def} \ A2\text{-def} \ M\text{-def}$ by auto
have $((a1, a2), (b1, b2)) \in \text{bsqr}(r)$
proof\(\text{cases} \ \text{wo-rel}\text{-max2} \ r \ a1 \ a2 = \text{wo-rel}\text{-max2} \ r \ b1 \ b2\)$
  assume Case1: $\text{wo-rel}\text{-max2} \ r \ a1 \ a2 \neq \text{wo-rel}\text{-max2} \ r \ b1 \ b2$
  thus $?\text{thesis}$ unfolding $\text{bsqr-def}$ using $4, 5$ by auto
next
  assume Case2: $\text{wo-rel}\text{-max2} \ r \ a1 \ a2 = \text{wo-rel}\text{-max2} \ r \ b1 \ b2$
  hence $b1 \in A1$ unfolding $A1\text{-def}$ using $2$ *** by auto
  hence $6: (a1, b1) \in r$ using $a1\text{-min}$ by auto
  show $?\text{thesis}$
  proof\(\text{cases} \ a1 = b1\)$
    assume Case21: $a1 \neq b1$
    thus $?\text{thesis}$ unfolding $\text{bsqr-def}$ using $4$ Case2 6 by auto
next
  assume Case22: $a1 = b1$
  hence $b2 \in A2$ unfolding $A2\text{-def}$ using $2$ *** Case2 by auto
  hence $7: (a2, b2) \in r$ using $a2\text{-min}$ by auto
  thus $?\text{thesis}$ unfolding $\text{bsqr-def}$ using $4$ 7 Case2 Case22 by auto
qed
qed
}

ultimately show $\exists d \in D. \ \forall d' \in D. \ (d, d') \in \text{bsqr}(r)$ by fastforce
qed

lemma $\text{bsqr-max2}$:
assumes WELL: Well-order $r$ and LEQ: $((a1, a2), (b1, b2)) \in \text{bsqr}(r)$
shows $(\text{wo-rel}\text{-max2} \ r \ a1 \ a2, \text{wo-rel}\text{-max2} \ r \ b1 \ b2) \in r$
proof –
  have $\{(a1, a2), (b1, b2)\} \leq \text{Field}(\text{bsqr}(r))$
  using LEQ unfolding $\text{Field-def}$ by auto
  hence $\{a1, a2, b1, b2\} \leq \text{Field}(r)$ unfolding $\text{Field-bsqr}$ by auto
  hence $\{\text{wo-rel}\text{-max2} \ r \ a1 \ a2, \text{wo-rel}\text{-max2} \ r \ b1 \ b2\} \leq \text{Field}(r)$
  using WELL wo-rel-def[of $r$] wo-rel-def-among[of $r$] by fastforce
moreover have $\{\text{wo-rel}\text{-max2} \ r \ a1 \ a2, \text{wo-rel}\text{-max2} \ r \ b1 \ b2\} \in r \ \forall \ \text{wo-rel}\text{-max2} \ r \ a1 \ a2 = \text{wo-rel}\text{-max2} \ r \ b1 \ b2$
  using LEQ unfolding $\text{bsqr-def}$ by auto
  ultimately show $?\text{thesis}$ using WELL unfolding order-on-defs refl-on-def by auto
qed

lemma $\text{bsqr-ofilter}$:
assumes WELL: Well-order r and
OF: wo-rel.ofilter (bsqr r) D and SUB: D < Field r × Field r and
NE: ~ (∃a. Field r = under r a)

shows ∃A. wo-rel.ofilter r A ∧ A < Field r ∧ D ≤ A × A

proof= let ?r' = bsqr r
  have Well: wo-rel using WELL wo-rel-def by blast
  hence Trans: trans r using wo-rel.TRANS by blast
  have Well': Well-order ?r' ∧ wo-rel ?r'
    using WELL bsqr-Well-order wo-rel-def by blast
  have D < Field ?r' unfolding Field-bsqr using SUB
    with OF obtain a1 and a2 where
    (a1,a2) ∈ Field ?r' and 1: D = underS ?r' (a1,a2)
    using Well' wo-rel.ofilter-underS-Field[of ?r' D] by auto
    hence 2: {a1,a2} ≤ Field r unfolding Field-bsqr by auto
    let ?m = wo-rel.max2 r a1 a2
    have D ≤ (under r ?m) × (under r ?m)
      proof(unfold 1)
      {fix b1 b2
        let ?n = wo-rel.max2 r b1 b2
        assume (b1,b2) ∈ underS ?r' (a1,a2)
        hence 3: ((b1,b2),(a1,a2)) ∈ ?r'
        unfolding underS-def by blast
        hence (?n,?m) ∈ r using WELL by (simp add: bsqr-max2)
        moreover
        {have (b1,b2) ∈ Field ?r' using 3 unfolding Field-bsqr by auto
          hence (b1,b2) ≤ Field r unfolding Field-bsqr by auto
          hence (b1,?n) ∈ r ∧ (b2,?n) ∈ r
            using Well by (simp add: wo-rel.max2-greater)
        }
        ultimately have (b1,?m) ∈ r ∧ (b2,?m) ∈ r
          using Trans trans-def[of r] by blast
        hence (b1,b2) ∈ (under r ?m) × (under r ?m) unfolding under-def by simp
        thus underS ?r' (a1,a2) ≤ (under r ?m) × (under r ?m) by auto
      qed
      moreover have wo-rel.ofilter r (under r ?m)
        using Well by (simp add: wo-rel.under.ofilter)
      moreover have under r ?m < Field r
        using NE under-field[of r ?m] by blast
      ultimately show ?thesis by blast
    qed

definition Func where
  Func A B = {f . (∀ a ∈ A. f a ∈ B) ∧ (∀ a. a ∉ A → f a = undefined)}

lemma Func-empty:
  Func {} B = {λx. undefined}

unfolding Func-def by auto
lemma `Func-elim`:
assumes `g ∈ Func A B` and `a ∈ A`
shows `∃ b. b ∈ B ∧ g a = b`
using `assms unfolding Func-def` by (cases `g a = undefined`) `auto`

definition `curr` where
`curr A f ≡ λ a. if a ∈ A then λ b. f (a, b) else undefined`

lemma `curr-in`:
assumes `f: f ∈ Func (A <∗> B) C`
shows `curr A f ∈ Func A (Func B C)`
using `assms unfolding curr-def Func-def` by `auto`

lemma `curr-inj`:
assumes `f1 ∈ Func (A <∗> B) C` and `f2 ∈ Func (A <∗> B) C`
shows `curr A f1 = curr A f2` ←→ `f1 = f2`
proof `safe`
assume `c: curr A f1 = curr A f2`
show `f1 = f2`
proof (rule ext, clarify)
fix `a b` show `f1 (a, b) = f2 (a, b)`
proof (cases `(a, b) ∈ A <∗> B`)
case `False`
thus `?thesis` using `assms unfolding Func-def` by `auto`
next
case `True` hence `a: a ∈ A` and `b: b ∈ B` by `auto`
thus `?thesis`
using `c unfolding curr-def fun-eq-iff by (elim allE[of - a]) simp` `qed`
qed

lemma `curr-surj`:
assumes `g ∈ Func A (Func B C)`
shows `∃ f ∈ Func (A <∗> B) C. curr A f = g`
proof
let `?f = λ ab. if fst ab ∈ A ∧ snd ab ∈ B then g (fst ab) (snd ab) else undefined`
show `curr A ?f = g`
proof (rule ext)
fix `a` show `curr A ?f a = g a`
proof (cases `a ∈ A`)
case `False`
  hence `g a = undefined` using `assms unfolding Func-def` by `auto`
  thus `?thesis unfolding curr-def using False by simp`
next
case `True`
obtain `g1` where `g1 ∈ Func B C` and `g a = g1`
using `assms using Func-elim[OF assms True] by blast`
thus ?thesis using True unfolding Func-def curr-def by auto
qed
qed
show ?f ∈ Func (A <∗> B) C using assms unfolding Func-def mem-Collect-eq by auto
qed

lemma bij-betw-curr:
  bij-betw (curr A) (Func (A <∗> B) C) (Func A (Func B C))
unfolding bij-betw-def inj-on-def image-def
apply (intro impl conjI ballI)
apply (erule curr-inj[THEN iffD1], assumption+)
apply auto
apply (erule curr-in)
using curr-surj by blast

definition Func-map where
Func-map B2 f1 f2 g b2 ≡ if b2 ∈ B2 then f1 (g (f2 b2)) else undefined

lemma Func-map:
assumes g: g ∈ Func A2 A1 and f1: f1 : A1 ⊆ B1 and f2: f2 : B2 ⊆ A2
shows Func-map B2 f1 f2 g ∈ Func B2 B1
using assms unfolding Func-map-def mem-Collect-eq by auto

lemma Func-non-emp:
assumes B ≠ {} shows Func A B ≠ {}
proof −
  obtain b where b: b ∈ B using assms by auto
  hence (λ a. if a ∈ A then b else undefined) ∈ Func A B unfolding Func-def by auto
  thus ?thesis by blast
qed

lemma Func-is-emp:
Func A B = {} ↔ A ≠ {} ∧ B = {} (is ?L ↔ ?R)
proof
  assume L: ?L
  moreover {assume A = {} hence False using L Func-empty by auto}
  moreover {assume B ≠ {} hence False using L Func-non-emp[of B A] by simp }
  ultimately show ?R by blast
next
  assume R: ?R
  moreover
    { fix f assume f ∈ Func A B
      moreover obtain a where a ∈ A using R by blast
      ultimately obtain b where b ∈ B unfolding Func-def by blast
      with R have False by blast
    }
the theory "BNF-Constructions-on-Wellorders"

thus \( L \) by blast
qed

lemma Func-map-surj:
assumes \( B1: f1 \cdot A1 = B1 \) and \( A2: \text{inj-on } f2 \cdot B2 \cdot f2 \cdot B2 \subseteq A2 \)
and \( B2A2: B2 = \{\} \implies A2 = \{\} \)
shows \( \text{Func } B2 \cdot B1 = \text{Func-map } B2 \cdot f1 \cdot f2 \cdot \text{Func } A2 \cdot A1 \)
proof (cases \( B2 = \{\} \))
  case True
  thus \( \text{thesis using } B2A2 \) by (auto simp: Func-empty Func-map-def)
next
case False
  note \( B2 = \) False
  show \( \text{thesis} \)
  proof (safe)
    fix \( h \)
    assume \( h: h \in \text{Func } B2 \cdot B1 \)
    def \( j1 \equiv \text{inv-into } A1 \cdot f1 \)
    have \( \forall a2 \in f2 \cdot B2. \exists b2. b2 \in B2 \cdot f2 b2 = a2 \) by blast
    then obtain \( k \) where \( k: \forall a2 \in f2 \cdot B2. k a2 \in B2 \land f2 (k a2) = a2 \)
      by atomize-elim (rule bchoice)
    \{ fix \( b2 \) assume \( b2: b2 \in B2 \)
      hence \( f2 (k (f2 b2)) = f2 b2 \) using \( A2(2) \) by auto
      moreover have \( k (f2 b2) \in B2 \) using \( b2 \cdot A2(2) \) \( k \) by auto
      ultimately have \( k (f2 b2) = b2 \) using \( b2 \cdot A2(1) \) unfolding \( \text{inj-on-def} \) by blast \}
    note kk = this
    obtain \( b22 \) where \( b22: b22 \in B2 \) using \( B2 \) by auto
    def \( j2A2: j2 \cdot A2 \subseteq B2 \) unfolding \( j2-def \) using \( k b22 \) by auto
    have \( j2: \land b2. b2 \in B2 \implies j2 (f2 b2) = b2 \)
      using kk unfolding \( j2-def \) by auto
    def \( g \equiv \text{Func-map } A2 \cdot j1 \cdot j2 h \)
    have \( \text{Func-map } B2 \cdot f1 \cdot f2 \cdot g = h \)
    proof (rule ext)
      fix \( b2 \)
      show \( \text{Func-map } B2 \cdot f1 \cdot f2 \cdot g \cdot b2 = h \cdot b2 \)
      proof (cases \( b2 \in B2 \))
        case True
        show \( \text{thesis} \)
        proof (cases \( h b2 = \text{undefined} \))
          case True
          hence \( b1: b2 \in f1 \cdot A1 \) using \( h \cdot b2 \in B2 \) unfolding \( B1 \cdot \text{Func-def} \) by auto
          show \( \text{thesis using } A2 \cdot f\cdot \text{inv-into-f}[OF \ b1] \)
            unfolding \( \text{True } g\cdot \text{def } \text{Func-map-def } j1\cdot \text{def } j2[OF \ b2 \in B2] \) by auto
          qed(insert \( A2 \cdot \text{True } j2[OF \ \text{True}] \cdot h \cdot B1 \), unfold \( j1\cdot \text{def } g\cdot \text{def } \text{Func-def} \)
        qed(insert \( h, \text{unfold } \text{Func-def } \text{Func-map-def}, \text{auto} \))
      qed
moreover have $g \in \text{Func } A2 A1$ unfolding $g$-def apply(rule \text{Func-map}[OF } h])

using $j2A2 B1 A2$ unfolding $j1$-def by (fast intro: inv-into-into)+

ultimately show $h \in \text{Func-map } B2 f1 f2 \cdot \text{Func } A2 A1$

unfolding Func-map-def[abs-def] by auto

qed(insert $B1$ Func-map[OF - - A2(2)], auto)

qed end

31 BNF-Cardinal-Order-Relation: Cardinal-Order Relations as Needed by Bounded Natural Functors

theory BNF-Cardinal-Order-Relation
imports Zorn BNF-Constructions-on-Wellorders
begin

In this section, we define cardinal-order relations to be minim well-orders on their field. Then we define the cardinal of a set to be some cardinal-order relation on that set, which will be unique up to order isomorphism. Then we study the connection between cardinals and:

- standard set-theoretic constructions: products, sums, unions, lists, powersets, set-of finite sets operator;

- finiteness and infiniteness (in particular, with the numeric cardinal operator for finite sets, $\text{card}$, from the theory Finite-Sets.thy).

On the way, we define the canonical $\omega$ cardinal and finite cardinals. We also define (again, up to order isomorphism) the successor of a cardinal, and show that any cardinal admits a successor.

Main results of this section are the existence of cardinal relations and the facts that, in the presence of infiniteness, most of the standard set-theoretic constructions (except for the powerset) do not increase cardinality. In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

31.1 Cardinal orders

A cardinal order in our setting shall be a well-order minim w.r.t. the order-embedding relation, $\leq o$ (which is the same as being minimal w.r.t. the strict order-embedding relation, $< o$), among all the well-orders on its field.

definition card-order-on :: "a set ⇒ 'a rel ⇒ bool
where
card-order-on A r ≡ well-order-on A r ∧ (∀ r′. well-order-on A r′ → r ≤ o r′)

abbreviation Card-order r ≡ card-order-on (Field r) r
abbreviation card-order r ≡ card-order-on UNIV r

lemma card-order-on-well-order-on:
assumes card-order-on A r
shows well-order-on A r
using assms unfolding card-order-on-def by simp

lemma card-order-on-Card-order:
card-order-on A r ⇒ A = Field r ∧ Card-order r
unfolding card-order-on-def using well-order-on-Field by blast

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo’s theorem (proved in Zorn.thy as theorem well-order-on), which states that on any given set there exists a well-order;
- The well-founded-ness of <o, ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

card-order-on: ∃ r. card-order-on A r
proof-
obtain R where R-def: R = {r. well-order-on A r} by blast
have 1: R ≠ {} ∧ (∀ r ∈ R. Well-order r)
using well-order-on[of A] R-def well-order-on-Well-order by blast
hence ∃ r ∈ R. ∀ r′ ∈ R. r ≤ o r′
using exists-minim-Well-order[of R] by auto
thus ?thesis using R-def unfolding card-order-on-def by auto
qed

lemma card-order-on-ordIso:
assumes CO: card-order-on A r and CO’: card-order-on A r’
shows r = o r’
using assms unfolding card-order-on-def
using ordIso-iff-ordLeq by blast

lemma Card-order-ordIso:
assumes CO: Card-order r and ISO: r’ = o r
shows Card-order r’
using ISO unfolding ordIso-def
proof
fix p’ assume well-order-on (Field r’) p’
hence 0: Well-order p’ ∧ Field p’ = Field r’
using well-order-on-Well-order by blast
obtain f where 1: iso r’ r f and 2: Well-order r ∧ Well-order r’
using ISO unfolding ordIso-def by auto
hence 3: inj-on \( f \) (Field \( r' \)) \& \( f' \) (Field \( r' \)) = Field \( r \)
by (auto simp add: iso-iff embed-inj-on)
let \( ?p = \text{dir-image} \ p' \ f \)
have 4: \( p' = o \ ?p \ \& \ \text{Well-order} \ ?p \)
using 0 2 3 by (auto simp add: dir-image-ordIso Well-order-dir-image)
moreover have Field \( ?p = \) Field \( r \)
using 0 3 by (auto simp add: dir-image-Field)
ultimately have \( r \leq_o \ ?p \)
using ISO 4 ordLeq-ordIso-trans ordIso-ordLeq-trans ordIso-symmetric by blast
qed

lemma Card-order-ordIso2:
assumes CO: Card-order \( r \) and ISO: \( r = o \ r' \)
shows Card-order \( r' \)
using assms Card-order-ordIso ordIso-symmetric by blast

31.2 Cardinal of a set

We define the cardinal of set to be some cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.

definition card-of :: 'a set \Rightarrow 'a rel (\_\_\_ \_\_ \_\_\_ \_\_\_\_)
where card-of \( A = (\text{SOME} \ r. \ \text{card-order-on} \ A \ r) \)

lemma card-of-card-order-on: card-order-on \( A \ \langle A \rangle \)
unfolding card-of-def by (auto simp add: card-order-on someI-ex)

lemma card-of-well-order-on: well-order-on \( A \ \langle A \rangle \)
using card-of-card-order-on card-order-on-def by blast

lemma Field-card-of: Field \( \langle A \rangle = A \)
using card-of-card-order-on[of A] unfolding card-order-on-def
using well-order-on-Field by blast

lemma card-of-Card-order: Card-order \( \langle A \rangle \)
by (simp only: card-of-card-order-on Field-card-of)

corollary ordIso-card-of-imp-Card-order:
\( r = o \ \langle A \rangle \Rightarrow \text{Card-order} \ r \)
using card-of-Card-order Card-order-ordIso by blast

lemma card-of-Well-order: Well-order \( \langle A \rangle \)
using card-of-Card-order unfolding card-order-on-def by auto

lemma card-of-refl: \( \langle A \rangle = o \ \langle A \rangle \)
using card-of-Well-order ordIso-reflexive by blast
lemma card-of-least: well-order-on A r \implies |A| \leq_o r
using card-of-card-order-on unfolding card-order-on-def by blast

lemma card-of-ordIso:
(\exists f. bij-betw f A B) = ( |A| =o |B| )

proof(auto)
fix f assume \ast: bij-betw f A B
then obtain r where well-order-on B r \land |A| =o r
using Well-order-iso-copy card-of-well-order-on by blast
hence |B| \leq_o |A| using card-of-least
ordLeq-ordIso-trans ordIso-symmetric by blast
moreover
{ let \(?g = inv-into A f \)
  have bij-betw \(?g B A\) using \ast bij-betw-inv-into by blast
  then obtain r where well-order-on A r \land |B| =o r
  using Well-order-iso-copy card-of-well-order-on by blast
  hence |A| \leq_o |B| using card-of-least
  ordLeq-ordIso-trans ordIso-symmetric by blast
}
ultimately show |A| =o |B| using ordIso-iff-ordLeq by blast
next
assume |A| =o |B|
then obtain f where iso ( |A| ) ( |B| ) f
unfolding ordIso-def by auto
hence bij-betw f A B unfolding iso-def Field-card-of by simp
thus \exists f. bij-betw f A B by auto
qed

lemma card-of-ordLeq:
(\exists f. inj-on f A \land f \circ A \leq B) = ( |A| \leq_o |B| )

proof(auto)
fix f assume \ast: inj-on f A and \ast\ast: f \circ A \leq B
{ assume |B| <o |A|
  hence |B| \leq_o |A| using ordLeq-iff-ordLess-or-ordIso by blast
  then obtain g where embed ( |B| ) ( |A| ) g
  unfolding ordLeq-def by auto
  hence inj-on g B \land g \circ B \leq A using embed-inj-on[of |B| |A| g]
  embed-Field[of |B| |A| g] by auto
  obtain h where bij-betw h A B
  using \ast\ast 1 Cantor-Bernstein[of f] by fastforce
  hence |A| =o |B| using card-of-ordIso by blast
  hence |A| \leq_o |B| using ordIso-iff-ordLeq by auto
}
thus |A| \leq_o |B| using ordLess-or-ordLeq[of |B| |A|]
by (auto simp: card-of-Well-order)
next
assume \ast: |A| \leq_o |B|
obtain f where embed ( |A| ) ( |B| ) f
using * unfolding ordLeq-def by auto

hence inj-on f A ∧ f' A ≤ B using embed-inj-on[of A B f]
embed-Field[of A B f] by auto

thus ∃ f. inj-on f A ∧ f' A ≤ B by auto

qed

lemma card-of-ordLeq2:
A ≠ {} ⇒ (∃ g. g' B = A) = (|A| ≤ o |B|)
using card-of-ordLeq[of A B inj-on-iff-surj[of A B]] by auto

lemma card-of-ordLess:
(¬ (∃ f. inj-on f A ∧ f' A ≤ B)) = (|B| < o |A|)
proof–
  have (¬ (∃ f. inj-on f A ∧ f' A ≤ B)) = (¬ |A| ≤ o |B|)
  using card-of-ordLeq by blast
  also have ... = (|B| < o |A|)
  not-ordLeq-iff-ordLess by blast
finally show ?thesis.

qed

lemma card-of-ordLess2:
B ≠ {} ⇒ (¬ (∃ f. f' A = B)) = (|A| < o |B|)
using card-of-ordLess[of B A inj-on-iff-surj[of B A]] by auto

lemma card-of-ordIsoI:
assumes bij-betw f A B
shows |A| = o |B|
using assms unfolding card-of-ordIso[symmetric] by auto

lemma card-of-ordLeqI:
assumes inj-on f A and ⋀ a. a ∈ A ⇒ f a ∈ B
shows |A| ≤ o |B|
using assms unfolding card-of-ordLeq[symmetric] by auto

lemma card-of-unique:
card-order-on A r ⇒ r = o |A|
bysimp only: card-order-on-ordIso card-of-card-order-on

lemma card-of-mono1:
A ≤ B ⇒ |A| ≤ o |B|
using inj-on-id[of A] card-of-ordLeq[of A B] by fastforce

lemma card-of-mono2:
assumes r ≤ o r'
shows |Field r| ≤ o |Field r'|
proof–
  obtain f where
1: well-order-on (Field r) r ∧ well-order-on (Field r) r' embed r r' f
   using assms unfolding ordLeq-def
   by (auto simp add: well-order-on-Well-order)
hence inj-on f (Field r) ∧ f' (Field r) ≤ Field r'  
   by (auto simp add: embed-inj-on embed-Field)
thus |Field r| ≤ o |Field r'| using card-of-ordLeq by blast
qed

lemma card-of-cong: r = o r' ⇒ |Field r| = o |Field r'|
   by (simp add: ordIso-iff-ordLeq card-of-mono2)

lemma card-of-Field-ordLess: Well-order r ⇒ |Field r| ≤ o r
   using card-of-least card-of-well-order-on well-order-on-Well-order by blast

lemma card-of-Field-ordIso:
   assumes Card-order r
   shows |Field r| = o r
   proof –
     have card-order-on (Field r) r
       using assms card-order-on-Card-order by blast
     moreover have card-order-on (Field r) |Field r|
       using card-of-card-order-on by blast
     ultimately show ?thesis using card-order-on-ordIso by blast
     qed

lemma Card-order-iff-ordIso-card-of:
   Card-order r = (r = o |Field r|)
   using ordIso-card-of-imp-Card-order card-of-Field-ordIso ordIso-symmetric by blast

lemma Card-order-iff-ordLeq-card-of:
   Card-order r = (r ≤ o |Field r|)
   proof –
     have Card-order r = (r = o |Field r|)
       unfolding Card-order-iff-ordIso-card-of by simp
     also have ... = (r ≤ o |Field r| ∧ |Field r| ≤ o r)
       unfolding ordIso-iff-ordLeq by simp
     also have ... = (r ≤ o |Field r|)
       using card-of-Field-ordLess
     by (auto simp: card-of-Field-ordLess ordLeq-Well-order-simp)
     finally show ?thesis.
     qed

lemma Card-order-iff-Restr-underS:
   assumes Well-order r
   shows Card-order r = (∀ a ∈ Field r. Restr r (underS r a) < o |Field r|)
   using assms unfolding Card-order-iff-ordLeq-card-of
   using ordLeq-iff-ordLess-Restr card-of-Well-order by blast

lemma card-of-underS:
assumes \( r : \text{Card-order} r \) and \( a : a : \text{Field} r \)
shows \(|\text{underS} r a| < o r\)
proof
- let \( ?A = \text{underS} r a \) let \( ?r′ = \text{Restr} r ?A \)
  have 1: \( \text{Well-order} r \)
  using \( r \) unfolding \( \text{card-order-on-def} \) by simp
  have \( \text{Well-order} ?r′ \) using 1 \( \text{Well-order-Restr} \) by auto
  moreover have \( \text{card-order-on} \) (\( \text{Field} ?r′ \)) \( |\text{Field} ?r′| \)
  using \( \text{card-of-card-order-on} \).
  ultimately have \( |\text{Field} ?r′| \leq o ?r′ \)
  unfolding \( \text{card-order-on-def} \) by simp
  moreover have \( \text{Field} ?r′ = ?A \)
  using 1 \( \text{wo-rel.underS-ofilter} \) \( \text{Field-Restr-ofilter} \)
  unfolding \( \text{wo-rel-def} \) by fastforce
  ultimately have \( |?A| \leq o ?r′ \) by simp
  also have \( ?r′ < o |\text{Field} r| \)
  using 1 \( a \ r \) \( \text{Card-order-iff-Restr-underS} \) by blast
  also have \( |\text{Field} r| = o r \)
  using \( r \) \( \text{ordIso-symmetric} \) unfolding \( \text{Card-order-iff-ordIso-card-of} \) by auto
  finally show \( \text{?thesis} \).
qed

lemma \( \text{ordLess-Field}: \)
assumes \( r < o r′ \)
shows \( |\text{Field} r| < o r′ \)
proof
- have \( \text{well-order-on} \) (\( \text{Field} r \)) \( r \) using \( \text{assms} \) unfolding \( \text{ordLess-def} \)
  by \( (\text{auto simp add: well-order-on-Well-order}) \)
  hence \( |\text{Field} r| \leq o r \) using \( \text{card-of-least} \) by blast
  thus \( \text{?thesis} \) using \( \text{assms} \) \( \text{ordLeq-ordLess-trans} \) by blast
qed

lemma \( \text{internalize-card-of-ordLeq}: \)
(\( |A| \leq o r \) = (\( \exists B \leq \text{Field} r. |A| = o |B| \wedge |B| \leq o r \))
proof
- assume \( |A| \leq o r \)
  then obtain \( p \) where 1: \( \text{Field} p \leq \text{Field} r \wedge |A| = o p \wedge p \leq o r \)
  using \( \text{internalize-ordLeq} [\text{|A| r}] \) by blast
  hence \( \text{Card-order} p \) using \( \text{card-of-Card-order} \) \( \text{Card-order-ordIso2} \) by blast
  hence \( |\text{Field} p| = o p \) using \( \text{card-of-Field-ordIso} \) by blast
  hence \( |A| = o |\text{Field} p| \wedge |\text{Field} p| \leq o r \)
  using 1 \( \text{ordIso-equivalence} \) \( \text{ordIso-ordLeq-trans} \) by blast
  thus \( \exists B \leq \text{Field} r. |A| = o |B| \wedge |B| \leq o r \) using 1 by blast
next
- assume \( \exists B \leq \text{Field} r. |A| = o |B| \wedge |B| \leq o r \)
  thus \( |A| \leq o r \) using \( \text{ordIso-ordLeq-trans} \) by blast
qed

lemma \( \text{internalize-card-of-ordLeq2}: \)
( |A| ≤ o |C| ) = ( ∃ B ≤ C, |A| = o |B| ∧ |B| ≤ o |C| )

31.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the “cardinal identity” = o and are monotonic w.r.t. ≤ o.

lemma card-of-empty: |{}| ≤ o |A|
using card-of-ordLeq inj-on-id by blast

lemma card-of-empty1:
assumes Well-order r ∨ Card-order r
shows |{}| ≤ o r
proof –
  have Well-order r using assms unfolding card-order-on-def by auto
  hence |Field r| ≤ o r
  using assms card-of-Field-ordLess by blast
  moreover have |{}| ≤ o |Field r| by (simp add: card-of-empty)
  ultimately show thesis using ordLeq-transitive by blast
qed

corollary Card-order-empty:
Card-order r ⇒ |{}| ≤ o r by (simp add: card-of-empty1)

lemma card-of-empty2:
assumes LEQ: |A| = o |{}|
s shows A = {}
using assms card-of-ordIso[of A] bij-betw-empty2 by blast

lemma card-of-empty3:
assumes LEQ: |A| ≤ o |{}|
s shows A = {}
using assms by (simp add: ordIso-iff-ordLeq card-of-empty1 card-of-empty2
ordLeq-Well-order-simp)

lemma card-of-image:
|{}|::'a set| = o |{}|::'b set|
using card-of-ordIso unfolding bij-betw-def inj-on-def by blast

lemma card-of-image:
|f ' A| ≤ o |A|
proof (cases A = {}, simp add: card-of-empty)
  assume A ~ = {}
  hence f ' A ~ = {} by auto
  thus |f ' A| ≤ o |A|
using card-of-ordLeq2[of f ' A A] by auto

qed

lemma surj-imp-ordLeq:
assumes \( B \subseteq f ' A \)
shows \(|B| \leq o |A|\)
proof
- have \(|B| \leq o |f ' A|\) using assms card-of-mono1 by auto
  thus \(?thesis\) using card-of-image ordLeq-transitive by blast
qed

lemma card-of-singl-ordLeq:
assumes \( A \neq \{}\)
shows \(|\{b\}| \leq o |A|\)
proof
  obtain a where *: a \(\in A\) using assms by auto
  let \(?h = \lambda b': if b' = b then a else undefined\)
  have inj-on \(?h \{b\} ∧ ?h ' \{b\} \leq A\)
    using * unfolding inj-on-def by auto
  thus \(?thesis\) unfolding card-of-ordLeq[symmetric] by (intro exI)
qed

corollary Card-order-singl-ordLeq:
\[[\text{Card-order } r; \text{Field } r \neq \{}\] \implies \(|\{b\}| \leq o r\)
using card-of-singl-ordLeq[of Field r b]
  card-of-Field-ordIso[of r] ordLeq-ordIso-trans by blast

lemma card-of-Pow: \(|A| < o |Pow A|\)
using card-of-ordLess2[of Pow A A] Cantors-paradox[of A]
  Pow-not-empty[of A] by auto

corollary Card-order-Pow:
Card-order \(r \implies r < o |Pow(\text{Field } r)|\)
using card-of-Pow card-of-Field-ordIso ordIso-ordLess-trans ordIso-symmetric by blast

lemma card-of-Plus1: \(|A| \leq o |A <+> B|\)
proof
  have \(\text{Inl ' A} \leq A <+> B\) by auto
  thus \(?thesis\) using inj-Inl[of A] card-of-ordLeq by blast
qed

corollary Card-order-Plus1:
\(\text{Card-order } r \implies r \leq o |(\text{Field } r) <+> B|\)
using card-of-Plus1 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Plus2: \(|B| \leq o |A <+> B|\)
proof
have Inr ' B ≤ A <+> B by auto
thus ?thesis using inj-Inr[of B] card-of-ordLeq by blast
qed

corollary Card-order-Plus2:
Card-order r ⇒ r ≤ o |A <+> (Field r)|
using card-of-Plus2 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Plus-empty1: |A| = o |A <+> {}|
proof
  have bij-betw Inl A (A <+> {}) unfolding bij-betw-def inj-on-def by auto
  thus ?thesis using card-of-ordIso by auto
qed

lemma card-of-Plus-empty2: |A| = o |{} <+> A|
proof
  have bij-betw Inr A ({} <+> A) unfolding bij-betw-def inj-on-def by auto
  thus ?thesis using card-of-ordIso by auto
qed

lemma card-of-Plus-commute: |A <+> B| = o |B <+> A|
proof
  let ?f = λ(c::'a + 'b). case c of Inl a ⇒ Inr a
  | Inr b ⇒ Inl b
  have bij-betw ?f (A <+> B) (B <+> A)
  unfolding bij-betw-def inj-on-def by force
  thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Plus-assoc:
fixes A :: 'a set and B :: 'b set and C :: 'c set
shows |(A <+> B) <+> C| = o |A <+> B <+> C|
proof
  def f ≡ λ(k::('a + 'b) + 'c).
  case k of Inl ab ⇒ (case ab of Inl a ⇒ Inl a
  |Inr b ⇒ Inr (Inl b))
  |Inr c ⇒ Inr (Inr c)
  have A <+> B <+> C ⊆ f ′ ((A <+> B) <+> C)
proof
  fix x assume x: x ∈ A <+> B <+> C
  show x ∈ f ′ ((A <+> B) <+> C)
  proof (cases x)
    case (Inl a)
    hence a ∈ A x = f (Inl (Inl a))
    using x unfolding f-def by auto
    thus ?thesis by auto
  next
  case (Inr bc) note 1 = Inr show ?thesis
proof (cases bc)
  case (Inl b)
  hence \( b \in B \) \( x = f (\text{Inl} (\text{Inr} b)) \)
  using \( x \) unfolding \( f\)-def by auto
  thus \( \text{?thesis} \) by auto
next
  case (Inr c)
  hence \( c \in C \) \( x = f (\text{Inr} c) \)
  using \( x \) unfolding \( f\)-def by auto
  thus \( \text{?thesis} \) by auto
qed
qed
qed
hence bij-betw \( f \) \( ((A <+> B) <+> C) \) \( (A <+> B <+> C) \)
unfolding bij-betw-def inj-on-def \( f\)-def by fastforce
thus \( \text{?thesis} \) using card-of-ordIso by blast
qed

lemma card-of-Plus-mono1:
assumes \(|A| \leq o |B|\)
shows \(|A <+> C| \leq o |B <+> C|\)
proof
  obtain \( f \) where \( 1: \text{inj-on} \ f \ A \land \ f \ ' \ A \leq B \)
  using \( \text{assms} \) card-of-ordLeq[of \( A \)] by fastforce
  obtain \( g \) where \( g\)-def:
    \( g = (\lambda a. \text{case} \ d \ \text{of} \ \text{Inl} \ a \Rightarrow \text{Inl}(f \ a) \mid \text{Inr} \ (c::'c) \Rightarrow \text{Inr} \ c) \) by blast
  have \( \text{inj-on} \ g \ (A <+> C) \land \ g \ ' \ (A <+> C) \leq (B <+> C) \)
  proof
    \{ fix \( d1 \) and \( d2 \) assume \( d1 \in A <+> C \land d2 \in A <+> C \) and \( g \ d1 = g \ d2 \)
      hence \( d1 = d2 \) using \( 1 \) unfolding inj-on-def \( g\)-def by force \}
    moreover
    \{ fix \( d \) assume \( d \in A <+> C \)
      hence \( g \ d \in B <+> C \) using \( 1 \)
      by (case-tac \( d \), auto simp add: \( g\)-def) \}
  
  ultimately show \( \text{?thesis} \) unfolding inj-on-def \( g\)-def by auto
  qed
thus \( \text{?thesis} \) using card-of-ordLeq by blast
qed

corollary ordLeq-Plus-mono1:
assumes \( r \leq o \ r' \)
shows \( (\text{Field} \ r) <+> C \leq o (\text{Field} \ r') <+> C \)
using \( \text{assms} \) card-of-mono2 card-of-Plus-mono1 by blast

lemma card-of-Plus-mono2:
assumes \(|A| \leq o |B|\)
shows $|C <\leftrightarrow A| \leq o |C <\leftrightarrow B|$
using assms card-of-Plus-mono1[of $A B C$]
ordIso-ordLeq-trans[of $|C <\leftrightarrow A|$] ordLeq-ordIso-trans[of $|C <\leftrightarrow A|$]
by blast

corollary ordLeq-Plus-mono2:
assumes $r \leq o r'$
shows $|A <\leftrightarrow (\text{Field } r)| \leq o |A <\leftrightarrow (\text{Field } r')|$
using assms card-of-mono2 card-of-Plus-mono2 by blast

lemma card-of-Plus-mono:
assumes $|A| \leq o |B| \text{ and } |C| \leq o |D|$
shows $|A <\leftrightarrow C| \leq o |B <\leftrightarrow D|$
ordLeq-transitive[of $|C <\leftrightarrow A|$] by blast

corollary ordLeq-Plus-mono:
assumes $r \leq o r'$ and $p \leq o p'$
shows $|(\text{Field } r) <\leftrightarrow (\text{Field } p)| \leq o |(\text{Field } r') <\leftrightarrow (\text{Field } p')|$

lemma card-of-Plus-cong1:
assumes $|A| = o |B|$
shows $|A <\leftrightarrow C| = o |B <\leftrightarrow C|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono1)

corollary ordIso-Plus-cong1:
assumes $r = o r'$
shows $|(\text{Field } r) <\leftrightarrow C| = o |(\text{Field } r') <\leftrightarrow C|$
using assms card-of-cong card-of-Plus-cong1 by blast

lemma card-of-Plus-cong2:
assumes $|A| = o |B|$
shows $|C <\leftrightarrow A| = o |C <\leftrightarrow B|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono2)

corollary ordIso-Plus-cong2:
assumes $r = o r'$
shows $|(\text{Field } r) <\leftrightarrow A| = o |(\text{Field } r') <\leftrightarrow A|$
using assms card-of-cong card-of-Plus-cong2 by blast

lemma card-of-Plus-cong:
assumes $|A| = o |B| \text{ and } |C| = o |D|$
shows $|A <\leftrightarrow C| = o |B <\leftrightarrow D|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono)

corollary ordIso-Plus-cong:
assumes $r = o r'$ and $p = o p'$
shows $|(Field r) <+> (Field p)| = o |(Field r') <+> (Field p')|

lemma card-of-Un-Plus-ordLeq:
$|A \cup B| \leq o |A <+> B|$
proof
let $?f = \lambda c. \text{if } c \in A \text{ then Inl } c \text{ else Inr } c$
have inj-on $?f \ (A \cup B) \ \wedge \ ?f\' \ (A \cup B) \leq A <+> B$
unfolding inj-on-def by auto
thus $\text{thesis using card-of-ordLeq by blast}$
qed

lemma card-of-Times1:
assumes $A \neq \{}$
shows $|B| \leq o \ |B \times A|$
proof(cases $B = \{}$, simp add: card-of-empty)
assume $\ast: B \neq \{}$
have $\text{fst} \ ((B \times A) = B$ using assms by auto
thus $\text{thesis using inj-on-iff-surj[of } B \times A$ card-of-ordLeq[of $B \times A] \ast \text{ by blast}$
qed

lemma card-of-Times-commute:
$|A \times B| = o \ |B \times A|$
proof
let $?f = \lambda (a::'a,b::'b). (b,a)$
have bij-betw $?f \ (A \times B) \ (B \times A)$
unfolding bij-betw-def inj-on-def by auto
thus $\text{thesis using card-of-ordIso by blast}$
qed

lemma card-of-Times2:
assumes $A \neq \{}$ shows $|B| \leq o \ |A \times B|$
using assms card-of-Times1[of $A \ B$] card-of-Times-commute[of $B A$]
ordLeq-ordIso-trans by blast

corollary Card-order-Times1:
$\langle Card-order r; B \neq \{} \rangle \Rightarrow r \leq o \ (Field r) \times B$
using card-of-Times1[of $B$] card-of-Field-ordIso
ordIso-ordLeq-trans ordIso-symmetric by blast

corollary Card-order-Times2:
$\langle Card-order r; A \neq \{} \rangle \Rightarrow r \leq o \ A \times (Field r)$
using card-of-Times2[of $A$] card-of-Field-ordIso
ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Times3: $|A| \leq o \ |A \times A|$
using card-of-Times1[of $A$]
by(cases $A = \{}$, simp add: card-of-empty, blast)
lemma card-of-Plus-Times-bool:  |A <+> A| =o |A × (UNIV::bool set)|
proof –
  let ?f = λc::'a + 'a. case c of Inl a ⇒ (a,True)
                         |Inr a ⇒ (a,False)
  have bij-betw ?f (A <+> A) (A × (UNIV::bool set))
proof –
  {fix c1 and c2 assume ?f c1 = ?f c2
   hence c1 = c2
   by (case-tac c1, case-tac c2, auto, case-tac c2, auto)
  }
moreover
  {fix c assume c ∈ A <+> A
   hence ?f c ∈ A × (UNIV::bool set)
   by (case-tac c, auto)
  }
moreover
  {fix a bl assume *: (a,bl) ∈ A × (UNIV::bool set)
   have (a,bl) ∈ ?f' ( A <+> A)
   proof (cases bl)
      assume bl hence ?f(Inl a) = (a,bl) by auto
         thus ?thesis using * by force
   next
      assume ¬ bl hence ?f(Inr a) = (a,bl) by auto
         thus ?thesis using * by force
   qed
  }
ultimately show ?thesis unfolding bij-betw-def inj-on-def by auto
qed
  thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Times-mono1:
assumes |A| ≤o |B|
shows |A × C| ≤o |B × C|
proof –
obtain f where I: inj-on f A ∧ f' A ≤ B
  using assms card-of-ordLeq[of A] by fastforce
obtain g where g-def:
  g = (λ(a,c:'c). (f a,c)) by blast
have inj-on g (A × C) ∧ g' (A × C) ≤ (B × C)
  using I unfolding inj-on-def using g-def by auto
thus ?thesis using card-of-ordLeq by blast
qed

corollary ordLeq-Times-mono1:
assumes r ≤o r'
shows |(Field r) × C| ≤o |(Field r') × C|
using assms card-of-mono2 card-of-Times-mono1 by blast
lemma card-of-Times-mono2:
assumes $|A| \leq o \ |B|
shows $|C \times A| \leq o \ |C \times B|
using assms card-of-Times-mono1[of A B C]
  ordIso-ordLeq-trans[of $\ |C \times A|$] ordLeq-ordIso-trans[of $\ |C \times A|$]
by blast

corollary ordLeq-Times-mono2:
assumes $r \leq o \ r'$
shows $|A \times (\text{Field } r)| \leq o \ |A \times (\text{Field } r')|
using assms card-of-mono2 card-of-Times-mono2 by blast

lemma card-of-Sigma-mono1:
assumes $\forall i \in I . \ |A_i| \leq o \ |B_i|
shows $\Sigma_{i : I} . \ |A_i| \leq o \ |\Sigma_{i : I} . \ B_i|
proof
  have $\forall i . \ i \in I \rightarrow (\exists f . \ \text{inj-on } f\ A i \wedge f\ A i \leq B i)$
  using assms by (auto simp add: card-of-ordLeq)
  with choice[of $\lambda i f . \ i \in I \rightarrow \text{inj-on } f\ A i \wedge f\ A i \leq B i$]
  obtain $F$ where $1 : \forall i \in I . \ \text{inj-on } (F i)\ A i \wedge (F i)\ A i \leq B i$
  by atomize_elim (auto intro: bchoice)
  obtain $g$ where $g\text{-def}: g = (\lambda (i,a::'b).\ (i,F i a))$ by blast
  have inj-on $g\ (\Sigma I A) \wedge g\ (\Sigma I B)$
  using $1$ unfolding inj-on-def using $g\text{-def}$ by force
  thus $?\text{thesis}$ using card-of-ordLeq by blast
qed

lemma card-of-UNION-Sigma:
$\bigcup_{i \in I} . \ |A_i| \leq o \ |\Sigma_{i : I} . \ A_i|
using Ex-inj-on-UNION-Sigma[of I A] card-of-ordLeq by blast

lemma card-of-bool:
assumes $a1 \neq a2$
shows $|\text{UNIV}::\text{bool set}| = o \ |\{a1,a2\}|
proof
  let $?f = \lambda \text{bl}.\ \text{case bl of True }\Rightarrow a1 \ |\ \text{False }\Rightarrow a2$
  have bij-betw $?f\ \text{UNIV}\ \{a1,a2\}$
  proof
    { fix $bl1$ and $bl2$ assume $?f\ bl1 = ?f\ bl2$
      hence $bl1 = bl2$ using assms by (case-tac bl1 , case-tac bl2 , auto) }
    moreover
    { fix $bl$ have $?f\ bl \in \{a1,a2\}$ by (case-tac bl , auto) }
  moreover
  { fix $a$ assume $*: a \in \{a1,a2\}$
     have $a \in ?f\ \text{UNIV}$
  qed
proof (cases a = a1)
  assume a = a1
  hence ?True = a by auto  thus ?thesis by blast
next
  assume a ≠ a1 hence a = a2 using * by auto
  hence ?False = a by auto  thus ?thesis by blast
qed

ultimately show ?thesis unfolding bij_betw_def inj_on_def by blast
qed

lemma card-of-Plus-Times-aux:
assumes A2: a1 ≠ a2 ∧ {a1,a2} ≤ A and
  LEQ: |A| ≤ o |B|
shows |A <+> B| ≤ o |A × B|
proof -
  have 1: |UNIV::bool set| ≤ o |A|
      using A2 card-of-mono1 [of {a1,a2}] card-of-bool[of a1 a2]
      inj-on-def[of |UNIV::bool set|] by blast
  have |A <+> B| ≤ o |B <+> B|
      using LEQ card-of-Plus-mono1 by blast
  moreover have |B <+> B| = o |B × (UNIV::bool set)|
      using card-of-Plus-Times-bool by blast
  moreover have |B × (UNIV::bool set)| ≤ o |B × A|
      using 1 by (simp add: card-of-Plus-mono2)
  moreover have |B × A| = o |A × B|
      using card-of-Times-commute by blast
  ultimately show |A <+> B| ≤ o |A × B|
      using ordLeq-ordIso-trans[of |A <+> B| |B <+> B| |B × (UNIV::bool set)|]
      ordLeq-ordIso-trans[of |A <+> B| |B × (UNIV::bool set)| |B × A|]
      card-of-Plus-Times-commute[of |A <+> B| |B × A| |A × B|]
      by blast
qed

lemma card-of-Plus-Times:
assumes A2: a1 ≠ a2 ∧ {a1,a2} ≤ A and
  B2: b1 ≠ b2 ∧ {b1,b2} ≤ B
shows |A <+> B| ≤ o |A × B|
proof -
  {assume |A| ≤ o |B|
      hence ?thesis using assms by (auto simp add: card-of-Plus-Times-aux)
  }
moreover
  {assume |B| ≤ o |A|
      hence |B <+> A| ≤ o |B × A|
      using assms by (auto simp add: card-of-Plus-Times-aux)
hence \( \text{thesis} \)
using card-of-Plus-commute card-of-Times-commute

erdIso-ordLeq-trans ordLeq-ordIso-trans by blast
}

ultimately show \( \text{thesis} \)
using card-of-Well-order \( \text{of A} \) card-of-Well-order \( \text{of B} \)

ordLeq-total \( \text{of} \ |A| \) by blast

qed

lemma card-of-Times-Plus-distrib:
\[ |A \times (B + C)| = o |A \times B + A \times C| \text{ (is \( |\text{RHS}| = o |\text{LHS}| \))} \]

proof –
let \( \lambda = \lambda (a, bc) \). case bc of Inl b \( \Rightarrow \) Inl \( (a, b) \) | Inr c \( \Rightarrow \) Inr \( (a, c) \)

have bij-betw \( \lambda \) ?RHS \( \text{LHS unfolding} \) bij-betw-def inj-on-def by force
thus \( \text{thesis} \) using card-of-ordIso by blast

qed

lemma card-of-ordLeq-finite:
assumes \( |A| \leq o |B| \) and finite B
shows finite A

using assms unfolding ordLeq-def

using embed-inj-on[\( |A| \text{ of } B \)] embed-field[\( |A| \text{ of } |B| \)]

Field-card-of[\( |A| \text{ of } |B| \)] inj-on-finite[\( |A| \text{ of } |B| \)] by fastforce

lemma card-of-ordLeq-infinite:
assumes \( |A| \leq o |B| \) and \( \neg \) finite A
shows \( \neg \) finite B

using assms card-of-ordLeq-finite by auto

lemma card-of-ordIso-finite:
assumes \( |A| = o |B| \)
shows finite A = finite B

using assms unfolding ordIso-def iso-def[abs-def]
by (auto simp: bij-betw-finite Field-card-of)

lemma card-of-ordIso-finite-Field:
assumes Card-order r and \( r = o |A| \)
shows finite(Field r) = finite A

using assms card-of-Field-ordIso card-of-ordIso-finite ordIso-equivalence by blast

31.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not increase the cardinality. The cornerstone for this is theorem Card-order-Times-same-infinite, which states that self-product does not increase cardinality – the proof of this fact adapts a standard set-theoretic argument, as presented, e.g., in the proof of theorem 1.5.11 at page 47 in [?]. Then everything else follows fairly easily.
lemma infinite-iff-card-of-nat:
\[ \neg \text{finite } A \iff (|\text{UNIV}::\text{nat set}| \leq o |A|) \]
unfolding infinite-iff-countable-subset card-of-ordLeq ..

The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:

lemma Card-order-infinite-not-under:
assumes CARD: Card-order r and INF: \( \neg \text{finite } (\text{Field } r) \)
shows \( (\exists a. \text{Field } r = \text{under } r a) \)
proof (auto)
  have \( 0: \text{Well-order } r \land \text{wo-rel } r \land \text{Refl } r \)
  using CARD unfolding wo-rel-def card-order-on-def order-on-defs by auto
  fix a assume \( *: \text{Field } r = \text{under } r a \)
  show False
    proof (cases a \in Field r)
      assume Case1: \( a \notin \text{Field } r \)
      hence \( \text{under } r a = \{\} \) unfolding Field-def under-def by auto
      thus False using INF * by auto
    next
      let \( ?r' = \text{Restr } r \text{ (under } S r a) \)
      assume Case2: \( a \in \text{Field } r \)
      hence \( 1: \text{under } r a = \text{under } S r a \cup \{a\} \land a \notin \text{under } S r a \)
        using \( 0 \) Refl-under-underS[of \( r a \)] underS-notIn[of \( a r \)] by blast
      have \( 2: \text{wo-rel.ofilter } r \text{ (under } S r a) \land \text{under } S r a < \text{Field } r \)
        using \( 0 \) wo-rel.underS-ofilter \( * 1 \) Case2 by fast
      hence \( ?r' < o r \) using \( 0 \) using ofilter-ordLess by blast
      moreover
      have \( \text{Field } ?r' = \text{under } S r a \land \text{Well-order } ?r' \)
        using \( 2 0 \) Field-Restr-ofilter[of \( r \)] Well-order-Restr[of \( r \)] by blast
      ultimately have \( |\text{under } S r a| < o r \) using ordLess-Field[of \( ?r' \)] by auto
      moreover have \( |\text{under } r a| = o r \) using \( * \) CARD card-of-Field-ordIso[of \( r \)] by auto
      ultimately have \( |\text{under } S r a| < o |\text{under } r a| \)
        using ordIso-symmetric ordLess-ordIso-trans by blast
      moreover
      \{ have \( \exists f. \text{bij-betw } f \text{ (under } r a) \text{ (under } S r a) \)
        using infinite-imp-bij-betw[of Field \( r a \)] INF * 1 by auto
        hence \( |\text{under } r a| = o |\text{under } S r a| \) using card-of-ordIso by blast
      \}
      ultimately show False using not-ordLess-ordIso ordIso-symmetric by blast
    qed
qed

lemma infinite-Card-order-limit:
assumes r: Card-order r and \( \neg \text{finite } (\text{Field } r) \)
and \( a: \text{Field } r \)
shows \( \exists b: \text{Field } r. a \neq b \land (a,b): r \)
proof
  have \( \text{Field } r \neq \text{under } r a \)
using assms Card-order-infinite-not-under by blast
moreover have under r a ≤ Field r
using under-Field.
ultimately have under r a < Field r by blast
then obtain b where I: b : Field r ∧ ∼ (b,a) : r
unfolding under-def by blast
moreover have ba: b ≠ a
using I r unfolding card-order-on-def well-order-on-def
linear-order-on-def partial-order-on-def preorder-on-def refl-on-def by auto
ultimately have (a,b) : r
using ba
have temp1: ∀ r. phi r −→ Well-order r
unfolding phi-def card-order-on-def by auto
have Ft: ∼(∃ r. phi r)
proof
assume ∃ r. phi r
hence {r. phi r} ≠ {} ∧ {r. phi r} ≤ {r. Well-order r}
using temp1 by auto
then obtain r where I: phi r and 2: ∀ r'. phi r' −→ r ≤ o r' and
3: Card-order r ∧ Well-order r
using exists-minim-Well-order[α] temp1 phi-def by blast
let ?A = Field r let ?r' = bsqr r
have 4: Well-order ?r' ∧ Field ?r' = ?A × ?A ∧ |?A| = o r
using 3 bsqr-Well-order Field-bsqr card-of-Field-ordIso by blast
have 5: Card-order |?A × ?A| ∧ Well-order |?A × ?A|
using card-of-Card-order card-of-Well-order by blast
have r < o |?A × ?A|
using 1 3 5 ordLess-or-ordLeg unfolding phi-def by blast
moreover have |?A × ?A| ≤ o ?r'
ultimately have r < o ?r' using ordLess-ordLeg-trans by auto
then obtain f where 6: embed r ?r' f and 7: ¬ bij-betw f ?A (?A × ?A)
unfolding ordLess-def embedS-def[abs-def]
by (auto simp add: Field-bsqr)
let ?B = f ?A
have |?A| = o |?B|
using 3 6 embed-inj-on inj-on-imp-bij-betw card-of-ordIso by blast
hence 8: \( r = o \, |?B| \) using 4 ordIso-transitive ordIso-symmetric by blast

have wo-rel.ofilter \( ?r' \, ?B \)
using 6 embed-Field-ofilter 3 4 by blast
hence wo-rel.ofilter \( ?r' \, ?B \land \, ?B \neq \, ?A \times \, ?A \land ?B \neq \, Field \, ?r' \)
using 7 unfolding bij-betw-def using 5 3 embed-inj-on 4 by auto
using 4 wo-rel-def[of \( ?r' \) wo-rel.ofilter-def[of \( ?r' \) ?B)] by blast
have \( \neg (\exists a. \, Field \, r = \, under \, r \, a) \)
using 1 unfolding phi-def using Card-order-infinite-not-under[of \( r \)] by auto
then obtain \( A1 \) where temp3: wo-rel.ofilter \( r \) \( A1 \land A1 < \, ?A \land 9: \, ?B \leq \, A1 \times A1 \)
using temp2 3 bsqr-ofilter[of \( r \, ?B \)] by blast
hence \( |?B| \leq_o |A1 \times A1| \) using card-of-monoI by blast
hence 10: \( r \leq |A1 \times A1| \) using 8 ordIso-ordLeq-trans by blast
let \( ?r1 = \text{Restr} \, r \, A1 \)
have \( ?r1 < o \, r \) using temp3 ofilter-ordLess 3 by blast
moreover
\{have well-order-on \( A1 \) \( ?r1 \) using 3 temp3 well-order-on-Restr by blast
hence \( |A1| \leq_o ?r1 \) using 3 Well-order-Restr card-of-least by blast \}
ultimately have 11: \( |A1| < o \, r \) using ordLeq-ordLess-trans by blast

have \( \neg \, \text{finite} \, (\text{Field} \, r) \) using 1 unfolding phi-def by simp
hence \( \neg \, \text{finite} \, |?B| \) using 8 3 card-of-ordIso-finite-Field[of \( r \, ?B \)] by blast
hence \( \neg \, \text{finite} \, A1 \) using 9 finite-cartesian-product finite-subset by blast
moreover have temp2: Field \( |A1| = A1 \land \text{Well-order} \, |A1| \land \text{Card-order} \, |A1| \)
using card-of-Card-order[of \( A1 \) card-of-Well-order[of \( A1 \)]]
by (simp add: Field-card-of)
moreover have \( \neg \, r \leq_o |A1| \)
using temp4 11 3 using not-ordLeq-iff-ordLess by blast
ultimately have \( \neg \, \text{finite} (\text{Field} \, |A1|) \land \text{Card-order} \, |A1| \land \neg \, r \leq_o |A1| \)
by (simp add: card-of-card-order-on)
hence \( |\text{Field} \, |A1| \times \text{Field} \, |A1| | \leq_o |A1| \)
using 2 unfolding phi-def by blast
hence \( |A1 \times A1| \leq_o |A1| \) using temp4 by auto
hence \( r \leq_o |A1| \) using 10 ordLeq-transitive by blast
thus False using 11 not-ordLess-ordLeq by auto
qed
thus \( \text{thesis} \) using assms unfolding phi-def by blast
qed

corollary card-of-Times-same-infinite:
assumes \( \neg \, \text{finite} \, A \)
shows \( |A \times A| = o \, |A| \)
proof-
let \( ?r = |A| \)
have Field \( ?r = A \land \text{Card-order} \, ?r \)
using Field-card-of card-of-Card-order[of \( A \)] by fastforce
hence $|A \times A| \leq o |A|

using Card-order-Times-same-infinite[of ?r] assms by auto
thus ?thesis using card-of-Times3 ordIso-iff-ordLeq by blast

qed

lemma card-of-Times-infinite:
assumes INF: $\neg$finite $A$ and NE: $B \neq \{\}$ and LEQ: $|B| \leq o |A|
shows $|A \times B| = o |A| \land |B \times A| = o |A|

proof
  have $|A| \leq o |A \times B| \land |A| \leq o |B \times A|
  using assms by (simp add: card-of-Times1 card-of-Times2)

moreover
  { have $|A \times B| \leq o |A \times A| \land |B \times A| \leq o |A \times A|$
    using LEQ card-of-Times-mono1 card-of-Times-mono2 by blast

    moreover have $|A \times A| = o |A|$ using INF card-of-Times-same-infinite by blast

    ultimately have $|A \times B| \leq o |A| \land |B \times A| \leq o |A|$
    using LEQ card-of-Times-mono1 card-of-Times-mono2 by blast

    ultimately show ?thesis by (simp add: ordIso-iff-ordLeq)
  }

qed

corollary Card-order-Times-infinite:
assumes INF: $\neg$finite $(\text{Field } r)$ and CARD: Card-order $r$ and
  NE: Field $p \neq \{\}$ and LEQ: $p \leq o r$
shows $|\text{Field } r \times (\text{Field } p)| = o r \land |(\text{Field } p) \times (\text{Field } r)| = o r$

proof
  have $|\text{Field } r \times \text{Field } p| = o |\text{Field } r| \land |\text{Field } p \times \text{Field } r| = o |\text{Field } r|$
  using assms by (simp add: card-of-Times-infinite card-of-mono2)

thus ?thesis using INF card-of-Times-same-infinite by blast

qed

lemma card-of-Sigma-ordLeq-infinite:
assumes INF: $\neg$finite $B$ and
  LEQ-I: $|I| \leq o |B|$ and LEQ: $\forall i \in I. |A i| \leq o |B|$
shows $|\text{SIGMA } i : I. A i| \leq o |B|

proof(cases $I = \{\}$, simp add: card-of-empty)
  assume $*; I \neq \{\}$
  have $|\text{SIGMA } i : I. A i| \leq o |I \times B|$
  using card-of-Sigma-monoI[of LEQ] by blast

moreover have $|I \times B| = o |B|$
  using INF * LEQ-I by (auto simp add: card-of-Times-infinite)

ultimately show ?thesis using ordLeq-ordIso-trans by blast

qed

lemma card-of-Sigma-ordLeq-infinite-Field:

assumes INF: ¬finite (Field r) and r: Card-order r and
LEQ-I: |I| ≤ o r and LEQ: ∀ i ∈ I. |A i| ≤ o r
shows |SIGMA i : I. A i| ≤ o r
proof-
  let ?B = Field r
  have 1: r = o |?B| ∧ |?B| = o r using r card-of-Field-ordIso
    ordIso-symmetric by blast
  hence |I| ≤ o |?B| ∀ i ∈ I. |A i| ≤ o |?B|
  using LEQ-I LEQ ordLeq-ordIso-trans by blast+
  hence |SIGMA i : I. A i| ≤ o |?B| using INF LEQ
  card-of-Sigma-ordLeq-infinite by blast
thus ?thesis using 1 ordLeq-ordIso-trans by blast
qed

lemma card-of-Times-infinite-field:
[¬finite (Field r); |A| ≤ o r; |B| ≤ o r; Card-order r]
⇒ |A <×> B| ≤ o r
by(simp add: card-of-Sigma-ordLeq-infinite-field)

lemma card-of-times-infinite-simps:
[¬finite A; B ≠ {}; |B| ≤ o |A|] ⇒ |A × B| = o |A|
[¬finite A; B ≠ {}; |B| ≤ o |A|] ⇒ |A| = o |A × B|
[¬finite A; B ≠ {}; |B| ≤ o |A|] ⇒ |B × A| = o |A|
[¬finite A; B ≠ {}; |B| ≤ o |A|] ⇒ |A| = o |B × A|
by (auto simp add: card-of-times-infinite ordIso-symmetric)

lemma card-of-UNION-ordLeq-infinite:
assumes INF: ¬finite B and
   LEQ-I: |I| ≤ o |B| and LEQ: ∀ i ∈ I. |A i| ≤ o |B|
shows |∪ i ∈ I. A i| ≤ o |B|
proof(cases I = {}, simp add: card-of-empty)
  assume *: I ≠ {}
  have |∪ i ∈ I. A i| ≤ o |SIGMA i : I. A i|
    using card-of-UNION-Sigma by blast
  moreover have |SIGMA i : I. A i| ≤ o |B|
    using assms card-of-Sigma-ordLeq-infinite by blast
  ultimately show ?thesis using ordLeq-transitive by blast
qed

corollary card-of-UNION-ordLeq-infinite-field:
assumes INF: ¬finite (Field r) and r: Card-order r and
   LEQ-I: |I| ≤ o r and LEQ: ∀ i ∈ I. |A i| ≤ o r
shows |∪ i ∈ I. A i| ≤ o r
proof-
  let ?B = Field r
  have 1: r = o |?B| ∧ |?B| = o r using r card-of-Field-ordIso
    ordIso-symmetric by blast
  hence |I| ≤ o |?B| ∀ i ∈ I. |A i| ≤ o |?B|
  using LEQ-I LEQ ordLeq-ordIso-trans by blast+
hence $\bigcup_{i \in I} A_i \leq o \mid B\mid$ using INF LEQ
 card-of-UNION-ordLeq-infinite by blast
 thus ?thesis using 1 ordLeq-ordIso-trans by blast
 qed

lemma card-of-Plus-infinite1:
 assumes INF: ~finite A and LEQ: |B| \leq o |A|
 shows |A <+> B| = o |A|
 proof(cases B = {}, simp add: card-of-Plus-empty1 card-of-Plus-empty2 ordIso-symmetric)
 let ?Inl = Inl::'a \Rightarrow 'a + 'b let ?Inr = Inr::'a \Rightarrow 'a + 'b
 assume *: B \neq {} 
 then obtain b1 where 1: b1 \in B by blast
 show ?thesis
 proof(cases B = {b1})
   let ?Inl' = Inl::'A \Rightarrow ((?Inl ' A)) 
   unfolding bij-betw-def inj-on-def by auto
 hence 3: ~finite (?Inl' ' A)
   using INF bij-betw-finite[of ?Inl ' A] by blast
 let ?A' = ?Inl' ' A \cup {?Inr b1}
 obtain g where bij-betw g (?Inl' ' A) ?A'
   using 3 infinite-imp-bij-betw2[of ?Inl ' A] by auto
 moreover have ?A' = A <+> B using Case1 by blast
 ultimately have bij-betw g (?Inl' ' A) (A <+> B) by simp
 hence bij-betw (g o ?Inl) A (A <+> B)
 using 2 by (auto simp add: bij-betw-trans)
 thus ?thesis using card-of-ordIso ordIso-symmetric by blast
 next
   assume Case2: B \neq {b1}
 with * 1 obtain b2 where 3: b1 \neq b2 \land \{b1,b2\} \leq B by fastforce
 obtain f where inj-on f B \land f o B \leq A
 using LEQ card-of-ordLeq[of B] by fastforce
 with 3 have f b1 \neq f b2 \land \{f b1, f b2\} \leq A
 unfolding inj-on-def by auto
 with 3 have |A <+> B| \leq o |A \times B|
 by (auto simp add: card-of-Plus-Times)
 moreover have |A \times B| = o |A|
 using assms * by (simp add: card-of-Times-infinite-simps)
 ultimately have |A <+> B| \leq o |A| using ordLeq-ordIso-trans by blast
 thus ?thesis using card-of-Plus1 ordIso-iff-ordLeq by blast
 qed
 qed

lemma card-of-Plus-infinite2:
 assumes INF: ~finite A and LEQ: |B| \leq o |A|
 shows |B <+> A| = o |A|
 using assms card-of-Plus-commute card-of-Plus-infinite1
 ordIso-equivalence by blast
lemma card-of-Plus-infinite:
assumes INF: ¬finite A and LEQ: |B| ≤ o |A|
shows |A <+> B| = o |A| ∧ |B <+> A| = o |A|
using assms by (auto simp: card-of-Plus-infinite1 card-of-Plus-infinite2)

corollary Card-order-Plus-infinite:
assumes INF: ¬finite(Field r) and CARD: Card-order r and
LEQ: p ≤ o r
shows | (Field r) <+> (Field p) | = o r ∧ | (Field p) <+> (Field r) | = o r
proof–
  have | Field r <+> Field p | =o | Field r | ∧
       | Field p <+> Field r | =o | Field r |
  using assms by (simp add: card-of-Plus-infinite card-of-mono2)
thus ?thesis
using assms ordIso-transitive[|Field r <+> Field p|]
  ordIso-transitive[-|Field r|] by blast
qed

31.5 The cardinal ω and the finite cardinals

The cardinal ω, of natural numbers, shall be the standard non-strict order relation on nat, that we abbreviate by natLeq. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers n, that we abbreviate by natLeq-on n.

definition (natLeq::(nat * nat) set) ≡ {(x,y). x ≤ y}
definition (natLess::(nat * nat) set) ≡ {(x,y). x < y}

abbreviation natLeq-on :: nat ⇒ (nat * nat) set
where natLeq-on n ≡ {(x,y). x < n ∧ y < n ∧ x ≤ y}

lemma infinite-cartesian-product:
assumes ¬finite A ¬finite B
shows ¬finite (A × B)
proof
  assume finite (A × B)
  from assms(1) have A ≠ {} by auto
  with (finite (A × B)) have finite B using finite-cartesian-productD2 by auto
  with assms(2) show False by simp
qed

31.5.1 First as well-orders

lemma Field-natLeq: Field natLeq = (UNIV::nat set)
by(unfold Field-def natLeq-def, auto)

lemma natLeq-Refl: Refl natLeq
unfolding refl-on-def Field-def natLeq-def by auto
lemma natLeq-trans: trans natLeq
unfolding trans-def natLeq-def by auto

lemma natLeq-Preorder: Preorder natLeq
unfolding preorder-on-def
by (auto simp add: natLeq-Refl natLeq-trans)

lemma natLeq-antisym: antisym natLeq
unfolding antisym-def natLeq-def by auto

lemma natLeq-Partial-order: Partial-order natLeq
unfolding partial-order-on-def
by (auto simp add: natLeq-Preorder natLeq-antisym)

lemma natLeq-Total: Total natLeq
unfolding total-on-def natLeq-def by auto

lemma natLeq-Linear-order: Linear-order natLeq
unfolding linear-order-on-def
by (auto simp add: natLeq-Partial-order natLeq-Total)

lemma natLeq-natLess-Id: natLess = natLeq − Id
unfolding natLeq-def natLess-def by auto

lemma natLeq-Well-order: Well-order natLeq
unfolding well-order-on-def
using natLeq-Linear-order wf-less natLeq-natLess-Id natLeq-def natLess-def by auto

lemma Field-natLeq-on: Field (natLeq-on n) = {x. x < n}
unfolding Field-def by auto

lemma natLeq-underS-less: underS natLeq n = {x. x < n}
unfolding underS-def natLeq-def by auto

lemma Restr-natLeq: Restr natLeq {x. x < n} = natLeq-on n
unfolding natLeq-def by force

lemma Restr-natLeq2:
Restr natLeq (underS natLeq n) = natLeq-on n
by (auto simp add: Restr-natLeq natLeq-underS-less)

lemma natLeq-on-Well-order: Well-order(natLeq-on n)
using Restr-natLeq[of n] natLeq-Well-order
   Well-order-Restr[of natLeq {x. x < n}] by auto

corollary natLeq-on-well-order-on: well-order-on {x. x < n} (natLeq-on n)
using natLeq-on-Well-order Field-natLeq-on by auto
LEMMA "BNF-Cardinal-Order-Relation"

649

lemma natLeq-on-wo-rel: wo-rel(natLeq-on n)
unfolding wo-rel-def using natLeq-on-Well-order.

31.5.2 Then as cardinals

lemma natLeq-Card-order: Card-order natLeq
proof (auto simp add: natLeq-Well-order
  Card-order-iff-Restr-underS Restr-natLeq2, simp add: Field-natLeq)
fix n have finite(Field (natLeq-on n)) by (auto simp: Field-def)
moreover have ~finite(UNIV::nat set) by auto
ultimately show natLeq-on n <o |UNIV::nat set|
using finite-ordLess-infinite[of natLeq-on n |UNIV::nat set]
  Field-card-of[of UNIV::nat set]
qed

corollary card-of-Field-natLeq:
|Field natLeq| =o natLeq
using Field-natLeq natLeq-Card-order Card-order-iff-ordIso-card-of[of natLeq]
  ordIso-symmetric[of natLeq] by blast

corollary card-of-nat:
|UNIV::nat set| =o natLeq
using Field-natLeq card-of-Field-natLeq by auto

corollary infinite_iff_natLeq-ordLeq:
~finite A = ( natLeq ≤o |A| )
  ordIso-ordLeq-trans ordLeq-ordIso-trans ordIso-symmetric by blast

corollary finite_iff_ordLess-natLeq:
finite A = ( |A| <o natLeq)
using infinite_iff-natLeq-ordLeq not-ordLeq_iff-ordLess
  card-of-Well-order natLeq-Well-order by blast

31.6 The successor of a cardinal

First we define isCardSuc r r', the notion of r' being a successor cardinal
of r. Although the definition does not require r to be a cardinal, only this
case will be meaningful.

definition isCardSuc :: 'a rel ⇒ 'a set rel ⇒ bool
where
isCardSuc r r' ≡ Card-order r' ∧ r <o r' ∧
(∀ (r''::'a set rel). Card-order r'' ∧ r <o r'' → r' ≤o r'')

Now we introduce the cardinal-successor operator cardSuc, by picking some
cardinal-order relation fulfilling isCardSuc. Again, the picked item shall be
proved unique up to order-isomorphism.
**THEORY “BNF-Cardinal-Order-Relation”**

**definition** cardSuc :: 'a rel ⇒ 'a set rel
where
cardSuc r ≡ SOME r'. isCardSuc r r'

**lemma** exists-minim-Card-order:
[\[ R \neq \{\}; \forall r \in R. \text{Card-order } r \] \implies \exists r \in R. \forall r' \in R. r \leq o r'

**unfolding** card-order-on-def using exists-minim-Well-order by blast

**lemma** exists-isCardSuc:
assumes Card-order r
shows ∃ r', isCardSuc r r'
proof−
let ?R = \{ (r': 'a set rel). \text{Card-order } r' \land r < o r' \}
have \[ \text{Pow}(\text{Field } r) \] \in ?R \land (\forall r \in ?R. \text{Card-order } r) using assms
by (simp add: card-of-Card-order Card-order-Pow)
then obtain r where r \in ?R \land (\forall r' \in ?R. r \leq o r')
using exists-minim-Card-order[of ?R] by blast
thus ?thesis unfolding isCardSuc-def by auto
qed

**lemma** cardSuc-isCardSuc:
assumes Card-order r
shows isCardSuc r (cardSuc r)
unfolding cardSuc-def using assms
by (simp add: exists-isCardSuc someI-ex)

**lemma** cardSuc-Card-order:
Card-order r \implies Card-order(cardSuc r)
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

**lemma** cardSuc-greater:
Card-order r \implies r < o cardSuc r
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

**lemma** cardSuc-ordLeq:
Card-order r \implies r \leq o cardSuc r
using cardSuc-greater ordLeq-iff-ordLess-or-ordIso by blast

The minimality property of cardSuc originally present in its definition is local to the type 'a set rel, i.e., that of cardSuc r:

**lemma** cardSuc-least-aux:
[\[ \text{Card-order } (r: 'a rel); \text{Card-order } (r': 'a set rel); r < o r' \] \implies cardSuc r \leq o r'
using cardSuc-isCardSuc unfolding isCardSuc-def by blast

But from this we can infer general minimality:

**lemma** cardSuc-least:
assumes CARD: Card-order r and CARD': Card-order r' and LESS: r < o r'
sows cardSuc r \leq o r'
proof−
let \(?p = \text{cardSuc } r\)

have 0: Well-order \(?p \land\) Well-order \(?r\)
using assms cardSuc-Card-order unfolding card-order-on-def by blast
{assume \(r' < o \ ?p\)
then obtain \(?r''\) where 1: Field \(?r''\) < Field \(?p\) and 2: \(r' = o \ ?r''\) \(\land\) \(?r'' < o \ ?p\)
using internalize-ordLess[of \(?r'\) \(?p\)] by blast
}

have Card-order \(?r''\) using CARD' Card-order-ordIso2 2 by blast
moreover have \(?p \leq o \ ?r''\) using LESS 2 ordLess-ordIso-trans by blast
ultimately have \(?p < o \ ?r''\) using cardSuc-least-aux CARD by blast
hence False using 2 notLess-ordLeq by blast
}

thus \(?\thesis\) using 0 ordLess-or-ordLeq by blast
qed

lemma cardSuc-ordLess-ordLeq:
assumes CARD: Card-order \(?r\) and CARD': Card-order \(?r'\)
shows \((r < o \ r') = (\text{cardSuc } r \leq o \ r')\)
proof(auto simp add: assms cardSuc-least)
assume cardSuc \(r \leq o \ r'\)
thus \(r < o \ r'\) using assms cardSuc-greater ordLess-ordLeq-trans by blast
qed

lemma cardSuc-ordLeq-ordLess:
assumes CARD: Card-order \(?r\) and CARD': Card-order \(?r'\)
shows \((r' < o \ \text{cardSuc } r) = (r' \leq o \ r)\)
proof
have Well-order \(?r\) \(\land\) Well-order \(?r'\)
using assms unfolding card-order-on-def by auto
moreover have Well-order(cardSuc \(?r\))
using assms cardSuc-Card-order card-order-on-def by blast
ultimately show \(?\thesis\)
using assms cardSuc-ordLess-ordLeq[of \(?r\) \(?r'\)]
notLess-iff-ordLess[of \(?r\) \(?r'\)] notLess-iff-ordLess[of \(?r'\) cardSuc \(?r\)] by blast
qed

lemma cardSuc-mono-ordLeq:
assumes CARD: Card-order \(?r\) and CARD': Card-order \(?r'\)
shows \((\text{cardSuc } r) \leq o \ (\text{cardSuc } r')\)
using assms cardSuc-ordLeq-ordLess cardSuc-ordLess-ordLeq cardSuc-Card-order by blast

lemma cardSuc-invar-ordIso:
assumes CARD: Card-order \(?r\) and CARD': Card-order \(?r'\)
shows \((\text{cardSuc } r) = o \ (\text{cardSuc } r')\)
using assms by (simp add: card-order-on-well-order-on cardSuc-Card-order)
thus "thesis using ordIso-iff-ordLeq[of r r'] ordIso-iff-ordLeq
using cardSuc-mono-ordLeq[of r r'] cardSuc-mono-ordLeq[of r' r] assms by blast
qed

lemma card-of-cardSuc-finite:
finite(Field(cardSuc |A| )) = finite A
proof
  assume *: finite (Field (cardSuc |A| ))
  have 0: |Field(cardSuc |A| )| = o cardSuc |A|
    using card-of-Card-order cardSuc-Card-order card-of-Field-ordIso by blast
  hence |A| ≤ o |Field(cardSuc |A| )|
  thus finite A using * card-of-ordLeq-finite by blast
next
  assume finite A
  then have finite (Field |Pow A| ) unfolding Field-card-of by simp
  then show finite (Field (cardSuc |A| ))
    proof (rule card-of-ordLeq-finite[OF card-of-mono2, rotated])
      show cardSuc |A| ≤ o |Pow A|
        by (rule iffD1[OF cardSuc-ordLess-ordLeq card-of-Pow]) (simp-all add: card-of-Card-order)
    qed
qed

lemma cardSuc-finite:
assumes Card-order r
shows finite (Field (cardSuc r)) = finite (Field r)
proof-
  let ?A = Field r
  have |?A| = o r using assms by (simp add: card-of-Field-ordIso)
  hence cardSuc |?A| = o cardSuc r using assms
  by (simp add: card-of-Card-order cardSuc-invar-ordIso)
  moreover have |Field (cardSuc |?A| )| = o cardSuc |?A|
    by (simp add: card-of-card-order-on Field-card-of-Field-ordIso cardSuc-Card-order)
  moreover
  {have |Field (cardSuc r)| = o cardSuc r
    using assms by (simp add: card-of-Field-ordIso cardSuc-Card-order)
    hence cardSuc r = o |Field (cardSuc r)|
      using ordIso-symmetric by blast
    }
  ultimately have |Field (cardSuc |?A| )| = o |Field (cardSuc r)|
    using ordIso-transitive by blast
  hence finite (Field (cardSuc |?A| )) = finite (Field (cardSuc r))
    using card-of-ordIso-finite by blast
  thus "thesis by (simp only: card-of-cardSuc-finite)
qed

lemma card-of-Plus-ordLess-infinite:
assumes $\text{INF}$: $\neg \text{finite } C$ and $\text{LESS1}: |A| < o |C|$ and $\text{LESS2}: |B| < o |C|$
shows $|A <\leftrightarrow B| < o |C|$
proof\((\text{cases } A = \{\} \lor B = \{\})\)
\begin{itemize}
\item assume $\text{Case1}: A = \{\} \lor B = \{\}$
\item hence $|A| = o |A <\leftrightarrow B| \lor |B| = o |A <\leftrightarrow B|$
\item using \(\text{card-of-Plus-empty1 card-of-Plus-empty2 by blast}\)
\end{itemize}
\begin{itemize}
\item hence $|A <\leftrightarrow B| = o |A| \lor |A <\leftrightarrow B| = o |B|$
\item using \(\text{ordIso-symmetric}[|A|] \text{ ordIso-symmetric}[|B|] \text{ by blast}\)
\end{itemize}
thus $?\text{thesis using LESS1 LESS2}$
\begin{itemize}
\item \(\text{ordIso-ordLess-trans[ |A <\leftrightarrow B| |A|]}\)
\item \(\text{ordIso-ordLess-trans[ |A <\leftrightarrow B| |B|] by blast}\)
\end{itemize}
next
\begin{itemize}
\item assume $\text{Case2}: \neg (A = \{\} \lor B = \{\})$
\item \{assume $\ast : |C| \leq o |A <\leftrightarrow B|$
\item hence $\neg \text{finite } (A <\leftrightarrow B)$ using \(\text{INF card-of-ordLeq-finite by blast}\)
\item hence $1: \neg \text{finite } A \lor \neg \text{finite } B$ using \(\text{finite-Plus by blast}\)
\item \{assume $\text{Case21}: |A| \leq o |B|$
\item hence $\neg \text{finite } B$ using $1$ \(\text{card-of-ordLeq-finite by blast}\)
\item hence $|A <\leftrightarrow B| = o |B|$ using $\text{Case2 Case21}$
\item by \(\text{auto simp add: card-of-Plus-infinite}\)
\item hence $?\text{thesis False using LESS2 not-ordLess-ordLeq * ordLeq-ordIso-trans by blast}\}
\end{itemize}
moreover
\begin{itemize}
\item \{assume $\text{Case22}: |B| \leq o |A|$
\item hence $\neg \text{finite } A$ using $1$ \(\text{card-of-ordLeq-finite by blast}\)
\item hence $|A <\leftrightarrow B| = o |A|$ using $\text{Case2 Case22}$
\item by \(\text{auto simp add: card-of-Plus-infinite}\)
\item hence $?\text{thesis False using LESS1 not-ordLess-ordLeq * ordLeq-ordIso-trans by blast}\}
\end{itemize}
ultimately have $?\text{thesis False using ordLeq-total card-of-Well-order[ of A] card-of-Well-order[ of B] by blast}$
\begin{itemize}
\item thus $?\text{thesis using ordLess-or-ordLeq[ |A <\leftrightarrow B| |C|]}$
\item \(\text{card-of-Well-order[ of } A <\leftrightarrow B \text{] card-of-Well-order[ of } C\text{] by auto}\)
\end{itemize}
qed

lemma \(\text{card-of-Plus-ordLess-infinite-Field}:\)
assumes $\text{INF}$: $\neg \text{finite } (\text{Field } r)$ and $r$: \text{Card-order } r and $\text{LESS1}: |A| < o r$ and $\text{LESS2}: |B| < o r$
shows $|A <\leftrightarrow B| < o |?C|$
proof
\begin{itemize}
\item let $?C = \text{Field } r$
\item have $1: r = o |?C| \land |?C| = o r$ using $r$ \(\text{card-of-Field-ordIso}\)
\item \(\text{ordIso-symmetric by blast}\)
\item hence $|A| < o |?C| \land |B| < o |?C|$
\item using \(\text{LESS1 LESS2 ordIso-trans by blast}\)
\item hence $|A <\leftrightarrow B| < o |?C|$ using \(\text{INF}\)
\item \(\text{card-of-Plus-ordLess-infinite by blast}\)
\end{itemize}
thus \( ?\text{thesis} \) using \( \text{1 ordLess-ordIso-trans} \) by blast

\text{qed}

\text{lemma card-of-Plus-ordLeq-infinite-Field:}
\text{assumes } r: \neg\text{finite} (\text{Field } r) \text{ and } A: |A| \leq o r \text{ and } B: |B| \leq o r
\text{and } c: \text{Card-order } r
\text{shows } |A <+> B| \leq o r
\text{proof –}
\text{let } ?r' = \text{cardSuc } r
\text{have Card-order } ?r' \land \neg\text{finite (Field } ?r') \text{ using } \text{assms}
\text{by (simp add: cardSuc-Card-order cardSuc-finite)}
\text{moreover have } |A| < o ?r' \text{ and } |B| < o ?r' \text{ using } A \ B \ c
\text{by (auto simp: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess)}
\text{ultimately have } |A <+> B| < o ?r'
\text{using card-of-Plus-ordLess-infinite-Field by blast}
\text{thus } ?\text{thesis using } c \ r
\text{by (simp add: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess)}
\text{qed}

\text{lemma card-of-Un-ordLeq-infinite-Field:}
\text{assumes } C: \neg\text{finite (Field } r) \text{ and } A: |A| \leq o r \text{ and } B: |B| \leq o r
\text{and } r: \text{Card-order } r
\text{shows } |A \ Un B| \leq o r
\text{using } \text{assms card-of-Plus-ordLeq-infinite-Field card-of-Un-Plus-ordLeq ordLeq-transitive} \text{ by fast}

\text{31.7 Regular cardinals}

\text{definition cofinal where}
\text{cofinal } A r \equiv
\text{ALL } a: \text{Field } r. \text{ EX } b: A.\ a \neq b \land (a,b) : r

\text{definition regularCard where}
\text{regularCard } r \equiv
\text{ALL } K.\ K \leq \text{Field } r \land \text{cofinal } K r \longrightarrow |K| = o r

\text{definition relChain where}
\text{relChain } r As \equiv
\text{ALL } i\ j.\ (i,j) \in r \longrightarrow As i \leq As j

\text{lemma regularCard-UNION:}
\text{assumes } r: \text{Card-order } r \text{ regularCard } r
\text{and } As: \text{relChain } r As
\text{and } Bsub: B \leq (\text{UN } i: \text{Field } r.\ As i)
\text{and } cardB: |B| < o r
\text{shows } EX i: \text{Field } r.\ B \leq As i
\text{proof –}
\text{let } ?\phi = \%b j. j: \text{Field } r \land b: As j
\text{have ALL } b: B.\ EX j.\ ?\phi b j \text{ using } Bsub \text{ by blast}
then obtain \( f \) where \( f: \forall b. b:B \implies \phi b (f b) \)
using \( \text{bchoice}[of B \phi] \) by blast
let \( ?K = f \cdot B \)

\{assume 1: \( \forall i. i: \text{Field } r \implies \sim B \leq \ As \ i\)
have 2: cofinal \( ?K r \)
unfolding cofinal-def proof auto
fix \( i \) assume \( i: \text{Field } r \)
with 1 obtain \( b \) where \( b: B \land b \notin \ As i \) by blast
hence \( i \neq f b \land (f b, i): r \)
using As unfolding relChain-def by auto
unfolding card-order-on-def well-order-on-def linear-order-on-def
total-on-def using \( i f b \) by auto
hence \( i \neq f b \land (i, f b): r \)
using r unfolding card-of-image .

moreover have \( ?K \leq \text{Field } r \)
ultimately have \( |?K| = o r \)
proof unfolding regularCard-def by blast
moreover
\{ have \( |?K| < o |B| \) using card-of-image .
  hence \( |?K| < o r \) using cardB ordLeq-ordLess-trans by blast
\}
ultimately have False using not-ordLess-ordIso by blast
thus \( ?\text{thesis} \) by blast
qed

lemma infinite-cardSuc-regularCard:
assumes \( r\text{-inf} : \neg \text{finite } (\text{Field } r) \) and \( r\text{-card} : \text{Card\text{-}order } r \)
shows regularCard \( (\text{cardSuc } r) \)
proof
  let \( ?r' = \text{cardSuc } r \)
  have \( ?r': \text{Card\text{-}order } ?r' \)
    !! p. Card-order \( p \implies (p < o r) \) = (p < o ?r')
    using \( r\text{-card} \) by (auto simp: cardSuc-Card-order cardSuc-ordLeq-ordLess)
  show \( ?\text{thesis} \)
unfolding regularCard-def proof auto
fix \( K \) assume 1: \( K \leq \text{Field } ?r' \) and 2: cofinal \( K \ ?r' \)
hence \( |K| \leq o |\text{Field } ?r'| \) by (simp only: card-of-mono1)
also have 22: \( |\text{Field } ?r'| = o ?r' \)
  using \( r\text{-card} \) by (simp add: card-of-Field-ordIso[of ?r'])
finally have \( |K| \leq o ?r' . \)
moreover
\{ let \( ?L = \bigcup j: K. \text{underS } ?r' j \)
  let \( ?J = \text{Field } r \)
  have \( ?J = r = o |?J| \)
  using \( r\text{-card} \) card-of-Field-ordIso ordIso-symmetric by blast
  assume \( |K| < o ?r' \)
  hence \( |K| < o r \) using \( r' \) card-of-Card-order[of K] by blast
\}
hence $|K| \leq o |?J|$ using $rJ$ ordLeq-ordIso-trans by blast
moreover
\begin{enumerate}
\item have ALL $j : K$. $|\text{underS } ?r' j| < o ?r'$
  using $r'$ 1 by (auto simp: card-of-underS)
\item have ALL $j : K$. $|\text{underS } ?r' j| \leq o r$
  using $r'$ card-of-Card-order by blast
\item have ALL $j : K$. $|\text{underS } ?r' j| \leq o |?J|
  using $rJ$ ordLeq-ordIso-trans by blast
\end{enumerate}
ultimately have $|?L| \leq o |?J|$ using $r$-inf card-of-UNION-ordLeq-infinite by blast
hence $|?L| \leq o r$ using $rJ$ ordIso-symmetric ordLeq-ordIso-trans by blast
hence $|?L| < o ?r'$ using $r'$ card-of-Card-order by blast
moreover
\begin{enumerate}
\item have Field $?r' \leq ?L$
  using 2 unfolding underS-def cofinal-def by auto
\item have Field $?r' \leq o |?L|$ by (simp add: card-of-mono1)
\item have $?r' \leq o |?L|$ using 22 ordIso-ordLeq-trans ordIso-symmetric by blast
\end{enumerate}
ultimately have $|?L| < o |?L|$ using ordLess-ordLeq-trans by blast
hence False using ordLess-irreflexive by blast
ultimately show $|K| = o ?r'$ unfolding ordLeq-iff-ordLess-or-ordIso by blast
qed


lemma cardSuc-UNION:
assumes $r$: Card-order $r$ and $\neg$finite (Field $r$)
and As: relChain (cardSuc $r$) As
and Bsub: $B \leq (\text{UN } i : \text{Field (cardSuc } r). \text{As } i)$
and cardB: $|B| \ll o r$
shows EX $i : \text{Field (cardSuc } r). \text{B } \leq \text{As } i$
proof
\begin{enumerate}
\item let $?r' = \text{cardSuc } r$
\item have Card-order $?r' \land |B| < o ?r'$
  using $r$ cardB cardSuc-ordLeq-ordLess cardSuc-Card-order
card-of-Card-order by blast
\item have regularCard $?r'$
  using assms by(simp add: infinite-cardSuc-regularCard)
\item ultimately show $\neg$thesis
  using As Bsub cardB regularCard-UNION by blast
\end{enumerate}
qed

31.8 Others

lemma card-of-Func-Times:
\[ \text{Func } (A \leftrightarrow B) \text{ } C = \text{ } o \text{ } \text{Func } A \text{ } (\text{Func } B \text{ } C) \]

unfolding card-of-ordIso\[\text{symmetric}\]
using bij-betw-curr by blast

lemma card-of-Pow-Func:
\[\text{Pow } A = o \text{ } \text{Func } A \text{ } (\text{UNIV::bool set})\]
proof -
def F \equiv \lambda A' a. (if a \in A then (if a \in A' then True else False) else undefined)
have bij-betw F (Pow A) (Func A (UNIV::bool set))
unfolding bij-betw-def inj-on-def proof (intro ballIImpl conjI)
fix A1 A2 assume A1 \in Pow A A2 \in Pow A F A1 = F A2
thus A1 = A2 unfolding F-def Pow-def fun-eq-iff by (auto split: split-if-asm)
next
show F : Pow A = Func A UNIV
proof safe
fix f assume f : f \in Func A (UNIV::bool set)
show f \in F : Pow A unfolding image-def mem-Collect-eq proof (intro bexI)
let ?A1 = \{ a \in A. f a = True\}
show f = F ?A1 unfolding F-def apply (rule ext)
using f unfolding Func-def mem-Collect-eq by auto
qed auto
qed (unfold Func-def mem-Collect-eq F-def, auto)

qed
thus ?thesis unfolding card-of-ordIso\[\text{symmetric}\] by blast

qed

lemma card-of-Func-UNIV:
\[\text{Func } (\text{UNIV::'a set}) \text{ } (B::'b set) = o \text{ } \{f::'a => 'b. range f \subseteq B\}\]
apply (rule ordIso-symmetric) proof (intro card-of-ordIsoI)
let ?F = \lambda f. ((f a) :: 'b)
show bij-betw ?F \{f. range f \subseteq B\} (Func UNIV B)
unfolding bij-betw-def inj-on-def proof safe
fix h :: 'a => 'b assume h : h \in Func UNIV B
hence \forall a. \exists b. h a = b unfolding Func-def by auto
then obtain f where f : \forall a. h a = f a by blast
hence range f \subseteq B using h unfolding Func-def by auto
thus h \in (\lambda f a. f a) \cdot \{f. range f \subseteq B\} using f by auto
qed (unfold Func-def fun-eq-iff, auto)

qed

lemma Func-Times-Range:
\[\text{Func } A \text{ } (B \leftrightarrow C) = o \text{ } \text{Func } A \text{ } (\text{Func } B \leftrightarrow \text{Func } C) \text{ is } \{?LHS\} = o \text{ } \{?RHS\}\]
proof -
let ?F = \lambda f g. (\lambda x. if x \in A then fst (fg x) else undefined, 
\lambda x. if x \in A then snd (fg x) else undefined)
let ?G = \lambda f, g. \lambda x. if x \in A then (f x, g x) else undefined
have bij-betw ?F ?LHS ?RHS unfolding bij-betw-def inj-on-def
proof (intro conjI Impl ballI equalityI subsetI)
THEORY “BNF-Cardinal-Arithmetic”

fix \( f, g \) assume \(*: f \in \text{Func} A (B \times C) \) \( g \in \text{Func} A (B \times C) \) \( \tilde{F} f = \tilde{F} g \)
show \( f = g \)
proof
  fix \( x \) from \(* \) have \( \text{fst} (f x) = \text{fst} (g x) \land \text{snd} (f x) = \text{snd} (g x) \)
  by (case_tac \( x \in A \)) (auto simp: Func-def fun-eq_iff split: if-splits)
then show \( f x = g x \) by (subst (1 2) surjective_pairing) simp
qed

next
fix \( f, g \) assume \( f, g \in \text{Func} A B \times \text{Func} A C \)
thus \( f, g \in \tilde{F} \cdot \text{Func} A (B \times C) \)
by (intro image-eqI[of - - ?G fg]) (auto simp: Func-def)
qed (auto simp: Func-def fun-eq_iff)
thus \( \text{thesis} \) using card-of-ordIso by blast
qed

end

32 BNF-Cardinal-Arithmetic: Cardinal Arithmetic as Needed by Bounded Natural Functors

theory BNF-Cardinal-Arithmetic
imports BNF-Cardinal-Order-Relation
begin

lemma dir-image: \([\forall x y. (f x = f y) = (x = y); \text{Card-order } r] \Longrightarrow r = o \text{ dir-image } r f\)
by (rule dir-image-ordIso) (auto simp add: inj-on_def card-order-on_def)

lemma card-order-dir-image:
  assumes bij: \( \text{bij } f \) and co: \( \text{card-order } r \)
  shows \( \text{card-order } (\text{dir-image } r f) \)
proof
  from assms have Field (\( \text{dir-image } r f \)) = UNIV
  using card-order-on-Card-order[of UNIV r] unfolding bij-def dir-image-Field
by auto
moreover from bij have \( \forall x y. (f x = f y) = (x = y) \) unfolding bij-def
inj-on_def by auto
with co have \( \text{Card-order } (\text{dir-image } r f) \)
using card-order-on-Card-order[of UNIV r] Card-order-ordIso2[OF - dir-image]
by blast
ultimately show \( \text{thesis} \) by auto
qed

lemma ordIso-refl: \( \text{Card-order } r \Longrightarrow r = o r \)
by (rule card-order-on-ordIso)

lemma ordLeq-refl: \( \text{Card-order } r \Longrightarrow r \leq o r \)
by (rule ordIso-imp-ordLeq, rule card-order-on-ordIso)
lemma card-of-ordIso-subst: $A = B \implies |A| = o |B|
by (simp only: ordIso-refl card-of-Card-order)

lemma Field-card-order: card-order $r \implies $Field $r = UNIV
using card-order-on-Card-order[of UNIV $r] by simp

32.1 Zero

definition czero where
  $czero = card-of \{\}$

lemma czero-ordIso:
  $czero = o \ czero
using card-of-empty-ordIso by (simp add: czero-def)

lemma card-of-ordIso-czero-iff-empty:
  $|A| = o (czero :: 'b rel) \longleftrightarrow A = (\{\} :: 'a set)
unfolding czero-def by (rule iffI[OF card-of-empty2]) (auto simp: card-of-refl card-of-empty-ordIso)

abbreviation Cnotzero where
  Cnotzero ($r :: 'a rel) $equiv $¬ ($r = o (czero :: 'a rel)) $and $Card-order $r

lemma Cnotzero-imp-not-empty: Cnotzero $r \implies Field $r \neq \{\}
unfolding Card-order-iff-ordIso-card-of czero-def by force

lemma czeroI:
  $[Card-order $r; Field $r = \{\}] \implies r = o \ czero
using Cnotzero-imp-not-empty ordIso-transitive[OF - czero-ordIso] by blast

lemma czeroE:
  $r = o \ czero \implies Field r = \{\}
unfolding czero-def
by (drule card-of-cong) (simp only: Field-card-of card-of-empty2)

lemma Cnotzero-mono:
  $[Cnotzero r; Card-order q; r \leq o q] \implies Cnotzero q
by (rule ccontr)
apply auto
apply (drule czeroE)
apply (erule notE)
apply (erule czeroI)
apply (drule card-of-mono2)
apply (simp only: card-of-empty3)
done
32.2 (In)finite cardinals

definition cinfinite where
  cinfinite r = (¬ finite (Field r))

abbreviation Cinfinite where
  Cinfinite r ≡ cinfinite r ∧ Card-order r

definition cfinite where
  cfinite r = finite (Field r)

abbreviation Cfinite where
  Cfinite r ≡ cfinite r ∧ Card-order r

lemma Cfinite-ordLess-Cinfinite: [Cfinite r; Cinfinite s] ⇒ r < o s
  unfolding cfinite-def cinfinite-def
  by (blast intro: finite-ordLess-infinite card-order-on-well-order-on)

lemmas natLeq-card-order = natLeq-Card-order[unfolded Field-natLeq]

lemma natLeq-cinfinite: cinfinite natLeq
  unfolding cinfinite-def Field-natLeq by (rule infinite-UNIV-nat)

lemma natLeq-ordLeq-cinfinite: 
  assumes inf: Cinfinite r
  shows natLeq ≤ o r
  proof –
    from inf have natLeq ≤ o |Field r| unfolding cinfinite-def
      using infinite-iff-natLeq-ordLeq by blast
    also from inf have |Field r| = o r by (simp add: card-of-unique ordIso-symmetric)
    finally show ?thesis .
  qed

lemma cinfinite-not-czero: cinfinite r ⇒ ¬ (r = o (czero :: 'a rel))
  unfolding cinfinite-def by (cases Field r = {}) (auto dest: czeroE)

lemma Cinfinite-Cnotzero: Cinfinite r ⇒ Cnotzero r
  by (rule conjI[OF cinfinite-not-czero]) simp-all

lemma Cinfinite-cong: [r1 = o r2; Cinfinite r1] ⇒ Cinfinite r2
  using Card-order-ordIso2[of r1 r2] unfolding cinfinite-def ordIso-iff-ordLeq
  by (auto dest: card-of-ordLeq-infinite[OF card-of-mono2])

lemma cinfinite-mono: [r1 ≤ o r2; cinfinite r1] ⇒ cinfinite r2
  unfolding cinfinite-def by (auto dest: card-of-ordLeq-infinite[OF card-of-mono2])

32.3 Binary sum

definition csum (infixr +c 65) where
  r1 +c r2 ≡ |Field r1 <+> Field r2|
lemma Field-csum: \( \text{Field } (r + c s) = \text{Inl } \cup \text{Field } r \cup \text{Inr } \cup \text{Field } s \)
unfolding csum-def Field-card-of by auto

lemma Card-order-csum: 
Card-order \((r1 + c r2)\)
unfolding csum-def by (simp add: card-of-Card-order)

lemma csum-Cnotzero1: 
\( \text{Cnotzero } r1 \Rightarrow \text{Cnotzero } (r1 + c r2) \)
unfolding csum-def using Cnotzero-imp-not-empty[of r1] Plus-eq-empty-conv[of Field r1 Field r2]
card-of-ordIso-czero-iff-empty[of Field r1 <+> Field r2] by (auto intro: card-of-Card-order)

lemma card-order-csum: 
assumes card-order r1 card-order r2
shows card-order \((r1 + c r2)\)
proof –
have Field r1 = UNIV Field r2 = UNIV using assms card-order-on-Card-order
by auto
thus ?thesis unfolding csum-def by (auto simp: card-of-card-order-on)
qed

lemma cinfinite-csum: 
\( \text{cinfinite } r1 \lor \text{cinfinite } r2 \Rightarrow \text{cinfinite } (r1 + c r2) \)
unfolding cinfinite-def csum-def by (auto simp: Field-card-of)

lemma Cinfinite-csum1: 
\( \text{Cinfinite } r1 \Rightarrow \text{Cinfinite } (r1 + c r2) \)
unfolding cinfinite-def csum-def by (rule conjI[OF - card-of-Card-order]) (auto simp: Field-card-of)

lemma Cinfinite-csum: 
\( \text{Cinfinite } r1 \lor \text{Cinfinite } r2 \Rightarrow \text{Cinfinite } (r1 + c r2) \)
unfolding cinfinite-def csum-def by (rule conjI[OF - card-of-Card-order]) (auto simp: Field-card-of)

lemma Cinfinite-csum-weak: 
\( [\text{Cinfinite } r1; \text{Cinfinite } r2] \Rightarrow \text{Cinfinite } (r1 + c r2) \)
by (erule Cinfinite-csum1)

lemma csum-cong: \( [p1 = o r1; p2 = o r2] \Rightarrow p1 + c p2 = o r1 + c r2 \)
by (simp only: csum-def ordIso-Plus-cong)

lemma csum-cong1: \( p1 = o r1 \Rightarrow p1 + c q = o r1 + c q \)
by (simp only: csum-def ordIso-Plus-cong1)

lemma csum-cong2: \( p2 = o r2 \Rightarrow q + c p2 = o q + c r2 \)
by (simp only: csum-def ordIso-Plus-cong2)
lemma \texttt{csum-mono}: \[p1 \leq o \ r1; \ p2 \leq o \ r2\] \implies p1 + c p2 \leq o \ r1 + c r2
by (simp only: csum-def ordLeq-Plus-mono)

lemma \texttt{csum-mono1}: \[\ p1 \leq o \ r1\] \implies p1 + c q \leq o \ p1 + c q
by (simp only: csum-def ordLeq-Plus-mono1)

lemma \texttt{csum-mono2}: \[\ p2 \leq o \ r2\] \implies q + c p2 \leq o \ q + c r2
by (simp only: csum-def ordLeq-Plus-mono2)

lemma \texttt{ordLeq-csum1}: Card-order p1 \implies p1 \leq o \ p1 + c p2
by (simp only: csum-def Card-order-Plus1)

lemma \texttt{ordLeq-csum2}: Card-order p2 \implies p2 \leq o \ p1 + c p2
by (simp only: csum-def Card-order-Plus2)

lemma \texttt{csum-com}: (p1 + c p2) + c p3 = o \ p1 + c \ p2 + c \ p3
by (simp only: csum-def card-of-Plus-commute)

lemma \texttt{csum-assoc}: (p1 + c p2) + c p3 = o \ p1 + c \ p2 + c \ p3
by (simp only: csum-def Field-card-of card-of-Plus-assoc)

lemma \texttt{Cfinite-csum}: \[\ Cfinite \ r; \ Cfinite \ s\] \implies Cfinite \ (r + c s)
unfolding cfinite-def csum-def Field-card-of
using card-of-card-order-on by simp

lemma \texttt{Plus-csum}: \[A \triangleleft B\] \implies o \ A + c \ B
by (simp only: csum-def Field-card-of card-of-refl)

lemma \texttt{Un-csum}: \[A \cup B\] \leq o \ A + c \ B
using ordLeq-ordIso-trans[OF card-of-Un-Plus-ordLeq Plus-csum] by blast
32.4 One

**definition** cone where
cone = card-of {()}

**lemma** Card-order-cone: Card-order cone
**unfolding** cone-def by (rule card-of-Card-order)

**lemma** Cfinite-cone: Cfinite cone
**unfolding** cfinite-def by (simp add: Card-order-cone)

**lemma** cone-not-czero: ¬ (cone = o czero)
**unfolding** czero-def cone-def ordIso-iff-ordLeq using card-of-empty3 empty-not-insert
by blast

**lemma** cone-ordLeq-Cnotzero: Cnotzero r =⇒ cone ≤ o r
**unfolding** cone-def by (rule Card-order-singl-ordLeq) (auto intro: czeroI)

32.5 Two

**definition** ctwo where
ctwo = |UNIV :: bool set|

**lemma** Card-order-ctwo: Card-order ctwo
**unfolding** ctwo-def by (rule card-of-Card-order)

**lemma** ctwo-not-czero: ¬ (ctwo = o czero)
**using** card-of-empty3 [of UNIV :: bool set] ordIso-iff-ordLeq
**unfolding** czero-def ctwo-def using UNIV-not-empty by auto

**lemma** ctwo-Cnotzero: Cnotzero ctwo
by (simp add: ctwo-not-czero Card-order-ctwo)

32.6 Family sum

**definition** Csum where
Csum r rs ≡ |SIGMA i : Field r. Field (rs i)|

**syntax** -Csum ::
pttrn => (’a * ’a) set => ’b * ’b set => ((’a * ’b) * (’a * ’b)) set
((3CSUM :-: -) [0, 51, 10] 10)

**translations**
CSUM i:r. rs == CONST Csum r (%i. rs)

**lemma** SIGMA-CSUM: |SIGMA i : I. As i| = (CSUM i : |I|. |As i| )
by (auto simp: Csum-def Field-card-of)
32.7 Product

definition cprod (infixr \ast c 80) where
  r1 \ast c r2 = |Field r1 <=> Field r2|

lemma card-order-cprod:
  assumes card-order r1 card-order r2
  shows card-order (r1 \ast c r2)
proof
  have Field r1 = UNIV Field r2 = UNIV using assms card-order-on-Card-order
  thus ?thesis by (auto simp: cprod-def card-of-card-order-on)
qed

lemma Card-order-cprod: Card-order (r1 \ast c r2)
  by (simp only: cprod-def Field-card-of card-of-card-order-on)

lemma cprod-mono1: p1 \leq o r1 =\Rightarrow p1 \ast c q \leq o r1 \ast c q
  by (simp only: cprod-def ordLeq-Times-mono1)

lemma cprod-mono2: p2 \leq o r2 =\Rightarrow q \ast c p2 \leq o q \ast c r2
  by (simp only: cprod-def ordLeq-Times-mono2)

lemma cprod-mono: [p1 \leq o r1; p2 \leq o r2] =\Rightarrow p1 \ast c p2 \leq o r1 \ast c r2
  by (rule ordLeq-transitive[of cprod-mono1 cprod-mono2])

lemma ordLeq-cprod2: [Cnotzero p1; Card-order p2] =\Rightarrow p2 \leq o p1 \ast c p2
  unfolding cprod-def by (rule Card-order-Times2) (auto intro: czeroI)

lemma cinfinite-cprod: [cinfinite r1; cinfinite r2] =\Rightarrow cinfinite (r1 \ast c r2)
  by (simp add: cinfinite-def cprod-def Field-card-of infinite-cartesian-product)

lemma cinfinite-cprod2: [Cnotzero r1; Cinfinite r2] =\Rightarrow cinfinite (r1 \ast c r2)
  by (rule cinfinite-mono) (auto intro: ordLeq-cprod2)

lemma Cinfinite-cprod2: [Cnotzero r1; Cinfinite r2] =\Rightarrow Cinfinite (r1 \ast c r2)
  by (blast intro: cinfinite-cprod2 Card-order-cprod)

lemma cprod-cong: [p1 =o r1; p2 =o r2] =\Rightarrow p1 \ast c p2 =o r1 \ast c r2
  unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono)

lemma cprod-cong1: [p1 =o r1] =\Rightarrow p1 \ast c p2 =o r1 \ast c p2
  unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono1)

lemma cprod-cong2: p2 =o r2 =\Rightarrow q \ast c p2 =o q \ast c r2
  unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono2)

lemma cprod-comm: p1 \ast c p2 =o p2 \ast c p1
  by (simp only: cprod-def card-of-Times-commute)
lemma card-of-Csum-Times:
\[ \forall i \in I. |A_i| \leq o |B| \implies \left( \text{CSUM } i : |I|. |A_i| \right) \leq o |I| \ast c |B| \]

by (simp only: Csum-def cprod-def Field-card-of card-of-Sigma-mono1)

lemma card-of-Csum-Times':
assumes Card-order r \ \forall i \in I. |A_i| \leq o r
shows (CSUM i : |I|. |A_i|) \leq o |I| \ast c r

proof
− from assms (1) have \(*\): r = o Field r by (simp add: card-of-unique)
with assms (2) have \(\forall i \in I. |A_i| \leq o |Field r|\) by (blast intro: ordLeg-ordIso-trans)

hence (CSUM i : |I|. |A_i|) \leq o |I| \ast c r Field r by (simp only: card-of-Csum-Times)
also from \(*\) have |I| \ast c r Field r \leq o |I| \ast c r by (simp only: Field-card-of card-of-refl cprod-def ordIso-imp-ordLeg)
finally show \(*\)thesis.
qed

lemma cprod-csum-distrib1:
\[ r_1 \ast r_2 r_3 = o r_1 \ast (r_2 + c r_3) \]

unfolding csum-def cprod-def by (simp add: Field-card-of card-of-Times-Plus-distrib ordIso-symmetric)

lemma csum-absorb2': \[
\begin{align*}
\text{Card-order } r_2; & r_1 \leq o r_2; \text{cinfinite } r_1 \lor \text{cinfinite } r_2 \\
\Rightarrow & r_1 + c r_2 = o r_2
\end{align*}
\]

unfolding csum-def by (rule conjunct2[OF Card-order-Plus-infinite])

(auto simp: csum-def-dest: cinfinite-mono)

lemma csum-absorb1':
assumes card: Card-order r2
and r12: r1 \leq o r2 and cr12: cinfinite r1 \lor cinfinite r2
shows r2 + c r1 = o r2

by (rule ordIso-transitive, rule csum-com, rule csum-absorb2', (simp only: assms)+)

lemma csum-absorb1:
\[ \text{cinfinite } r_2; r_1 \leq o r_2 \implies r_2 + c r_1 = o r_2 \]

by (rule csum-absorb1') auto

32.8 Exponentiation

definition cexp (infixr "ˆc" 90) where
\[ r_1 "ˆc" r_2 \equiv |\text{Func } (\text{Field } r_2) (\text{Field } r_1)| \]

lemma Card-order-cexp: Card-order (r1 "ˆc" r2)

unfolding cexp-def by (rule card-of-Card-order)

lemma cexp-mono':
assumes I: p1 \leq o r1 and 2: p2 \leq o r2
and n: Field p2 = {} \implies Field r2 = {}
shows p1 "ˆc" p2 \leq o r1 "ˆc" r2

proof (cases Field p1 = {})
case True
hence Field p2 \neq {} \implies Field (Field p2) {} = {} unfolding Func-is-emp by
simp  
with True have \(|\text{Field}| |\text{Func} (\text{Field } p2) (\text{Field } p1)|| \leq o \text{ cone}  
unfolding \text{cone-def Field-card-of}  
by (cases Field p2 = \{\}, auto intro: surj-ordLeq simp: Func-empty)  
hence \text{Func} (\text{Field } p2) (\text{Field } p1) \leq o \text{ cone by (simp add: Field-card-of cexp-def)}  
hence p1 \circ p2 \leq o \text{ cone unfolding cexp-def} .  
thus \?thesis  
proof (cases Field p2 = \{\})  
case True  
with \(n\) have Field r2 = \{\} .  
hence cone \leq o r1 \circ r2 unfolding \text{cone-def cexp-def Func-def}  
by (auto intro: card-of-ordLeq[where f=\lambda -. undefined])  
thus \?thesis using \(p1 \circ p2 \leq o \text{ cone} \) ordLeq-transitive by auto
next  
case False with True have \(|\text{Field} (p1 \circ p2)| = o \text{ czero}  
unfolding card-of-ordIso-czero-iff-empty cexp-def Field-card-of Func-def by auto  
thus \?thesis unfolding cexp-def card-of-ordIso-czero-iff-empty Field-card-of by simp add: card-of-empty)  
qed
next  
case False  
have 1: \(|\text{Field } p1| \leq o |\text{Field } r1| \) and 2: \(|\text{Field } p2| \leq o |\text{Field } r2|  
using 1 2 by (auto simp: card-of-mono2)
obtain f1 where f1: \(f1 : \text{Field } r1 = \text{Field } p1\)  
using 1 unfolding card-of-ordLeq2[OF False, symmetric] by auto
obtain f2 where f2: inj-on f2 \((\text{Field } p2) f2 : \text{Field } p2 \subseteq \text{Field } r2\)  
using 2 unfolding card-of-ordLeq[symmetric] by blast
have 0: \text{Func-map} (\text{Field } p2) f1 f2 : (\text{Field} (r1 \circ r2)) = Field (p1 \circ p2)  
unfolding cexp-def Field-card-of using Func-map-surj[OF f1 f2 n, symmetric]  
.
have 00: Field (p1 \circ p2) \neq \{\} unfolding cexp-def Field-card-of Func-is-emp  
using False by simp
show \?thesis  
using 0 card-of-ordLeq2[OF 00] unfolding cexp-def Field-card-of by blast  
qed

lemma cexp-mono:  
assumes 1: \(p1 \leq o r1\) and 2: \(p2 \leq o r2\)  
and n: \(p2 = o \text{ czero} \Rightarrow r2 = o \text{ czero} \) and card: Card-order p2  
shows \(p1 \circ p2 \leq o r1 \circ r2\)  
by (rule cexp-mono[OF 1 2 czeroE[OF n[OF czeroI[OF card]]]])

lemma cexp-mono1:  
assumes 1: \(p1 \leq o r1\) and q: Card-order q  
shows \(p1 \circ q \leq o r1 \circ q\)  
using ordLeq-refl[OF q] by (rule cexp-mono[OF 1]) (auto simp: q)

lemma cexp-mono2':
assumes 2: \( p_2 \leq o r_2 \) and \( q: \text{Card-order} \ q \)

and \( n: \text{Field} \ p_2 = \{} \implies \text{Field} \ r_2 = \{} \)

shows \( q \ ^c p_2 \leq o q \ ^c r_2 \)

using \( \text{ordLeq-refl}[OF \ q] \) by (rule \( \text{cexp-mono} \)[OF \( \text{n} \) 2]) auto

\[ \text{lemma: cexp-mono2:} \]

assumes 2: \( p_2 \leq o r_2 \) and \( q: \text{Card-order} \ q \)

and \( n: p_2 = o \ czero \implies r_2 = o \ czero \) and \( \text{card: Card-order} \ p_2 \)

shows \( q \ ^c p_2 \leq o q \ ^c r_2 \)

using \( \text{ordLeq-refl}[OF \ q] \) by (rule \( \text{cexp-mono} \)[OF \( \text{n} \) \( \text{card} \)] 2) auto

\[ \text{lemma: cexp-mono2-Cnotzero:} \]

assumes \( p_2 \leq o r_2 \) \( \text{Card-order} \ q \) \( \text{Cnotzero} \ p_2 \)

shows \( q \ ^c p_2 \leq o q \ ^c r_2 \)

using assms (3) \( \text{czeroI} \) by (blast intro: \( \text{cexp-mono} \)[OF \( \text{assms} \ (1,2) \)] )

\[ \text{lemma: cexp-cong:} \]

assumes 1: \( p_1 = o r_1 \) and \( 2: p_2 = o r_2 \)

and \( \text{Cr: Card-order} \ r_2 \)

and \( \text{Cp: Card-order} \ p_2 \)

shows \( p_1 \ ^c p_2 = o r_1 \ ^c r_2 \)

proof –

obtain \( f \) where \( \text{bij-betw} \ f \) (Field \( p_2 \) (Field \( r_2 \))

using \( 2 \ \text{card-of-ordIso}[of \ Field \ p_2 Field \ r_2] \ \text{card-of-cong} \) by auto

hence \( \text{0: Field} \ p_2 = \{} \iff \text{Field} \ r_2 = \{} \) unfolding bij-betw-def by auto

have \( r: p_2 = o \ czero \implies r_2 = o \ czero \)

and \( p: r_2 = o \ czero \implies p_2 = o \ czero \)

using \( 0 \ \text{Cr} \ \text{Cp} \ \text{czeroE czeroI} \) by auto

show \( ?\text{thesis} \) using \( 0 \ 1 \ 2 \) unfolding \( \text{ordIso-iff-ordLeq} \)

using \( r \ p \ \text{cexp-mono}[OF \ - \ - \ \text{Cp}] \ \text{cexp-mono}[OF \ - \ - \ \text{Cr}] \) by blast

qed

\[ \text{lemma: cexp-cong1:} \]

assumes 1: \( p_1 = o r_1 \) and \( q: \text{Card-order} \ q \)

shows \( p_1 ^c q = o r_1 ^c q \)

by (rule \( \text{cexp-cong}[OF \ 1 - q \ q] \)) (rule \( \text{ordIso-refl}[OF \ q] \))

\[ \text{lemma: cexp-cong2:} \]

assumes 2: \( p_2 = o r_2 \) and \( q: \text{Card-order} \ q \) and \( p: \text{Card-order} \ p_2 \)

shows \( q ^c p_2 = o q ^c r_2 \)

by (rule \( \text{cexp-cong}[OF \ - 2] \)) (auto simp only: \( \text{ordIso-refl} \ \text{Card-order-ordIso2}[OF \ p \ 2] \ q \ p \) )

\[ \text{lemma: cexp-cone:} \]

assumes \( \text{Card-order} \ r \)

shows \( r ^c \text{cone} = o r \)

proof –

have \( r ^c \text{cone} = o \ |\text{Field} \ r| \)

unfolding \( \text{cexp-def} \ \text{cone-def} \ \text{Field-card-of} \text{ Func-empty} \)
card-of-ordIso[symmetric] bij-betw-def Func-def inj-on-def image-def

by (rule ex1[of _ λf. f ()]) auto
also have |Field r| = o r by (rule card-of-Field-ordIso[OF assms])
finally show ?thesis.
qed

lemma cexp-cprod:
  assumes r1: Card-order r1
  shows (r1 ^c r2) ^c r3 =o r1 ^c (r2 *c r3) (is ?L =o ?R)
proof -
  have ?L =o r1 ^c (r3 *c r2)
    unfolding cprod-def cexp-def Field-card-of
    using card-of-Func-Times by (rule ordIso-symmetric)
  also have r1 ^c (r3 *c r2) =o ?R
    apply (rule cexp-cong2) using cprod-com r1 by (auto simp: Card-order-cprod)
finally show ?thesis.
qed

lemma cprod-infinite1': [Cinfinite r; Cnotzero p; p ≤ o r] ==> r *c p = o r
unfolding cprod-def cprod-def
by (rule Card-order-Times-infinite[THEN conjunct1]) (blast intro: czeroI) +

lemma cprod-infinite: Cinfinite r ==> r *c r = o r
using cprod-infinite1' Cinfinite-Cnotzero ordLeq-refl by blast

lemma cexp-cprod-ordLeq:
  assumes r1: Card-order r1 and r2: Cinfinite r2
  and r3: Cnotzero r3 r3 ≤ o r2
  shows (r1 ^c r2) ^c r3 =o r1 ^c r2 (is ?L =o ?R)
proof -
  have ?L =o r1 ^c (r2 *c r3) using cexp-cprod[OF r1] .
  also have r1 ^c (r2 *c r3) =o ?R
    apply (rule cexp-cong2)
    apply (rule cprod-infinite1'[OF r2 r3]) using r1 r2 by (fastforce simp: Card-order-cprod) +
finally show ?thesis.
qed

lemma Cnotzero-UNIV: Cnotzero |UNIV|
by (auto simp: card-of-Card-order card-of-ordIso-czero-iff-empty)

lemma ordLess-ctwo-cexp:
  assumes Card-order r
  shows r <o ctwo ^c r
proof -
  have r <o |Pow (Field r)| using assms by (rule Card-order-Pow)
  also have |Pow (Field r)| = o ctwo ^c r
    unfolding ctwo-def cexp-def Field-card-of by (rule card-of-Pow-Func)
finally show ?thesis.
qed
lemma ordLeq-cexp1:
  assumes Cnotzero r Card-order q
  shows \( q \leq o q \cdot c r \)
proof (cases q = o (czero :: 'a rel))
  case True thus \(?thesis\) by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
  case False thus \(?thesis\)
  apply (rule ordIso-ordLeq-trans)
  apply (rule ordIso-symmetric)
  apply (rule cexp-cone)
  apply (rule assms(2))
  apply (rule cexp-mono2)
  apply (rule cone-ordLeq-Cnotzero)
  apply (rule assms(1))
  apply (rule assms(2))
  apply (rule notE)
  apply (rule cone-not-czero)
  apply assumption
  apply (rule Card-order-cone)
done
qed

lemma ordLeq-cexp2:
  assumes ctwo \( \leq o q \) Card-order r
  shows r \( \leq o q \cdot c r \)
proof (cases r = o (czero :: 'a rel))
  case True thus \(?thesis\) by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
  case False thus \(?thesis\)
  apply (rule ordLess-imp-ordLeq)
  apply (rule ordLess-ordLeq-trans)
  apply (rule ordLess-ctwo-cexp)
  apply (rule assms(2))
  apply (rule cexp-mono1)
  apply (rule assms(1))
  apply (rule assms(2))
done
qed

lemma cinfinite-cexp: \([\text{ctwo} \leq o q; \text{Cinfinite } r] \Rightarrow \text{cinfinite } (q \cdot c r)\)
by (rule cinfinite-mono[OF ordLeq-cexp2]) simp-all

lemma Cinfinite-cexp:
\([\text{ctwo} \leq o q; \text{Cinfinite } r] \Rightarrow \text{Cinfinite } (q \cdot c r)\)
by (simp add: cinfinite-cexp Card-order-cexp)
lemma \texttt{ctwo-ordLess-natLeq}: \texttt{ctwo} <_o \texttt{natLeq}
unfolding \texttt{ctwo-def} using \texttt{finite-UNIV natLeq-cinfinite natLeq-Card-order}
by (intro \texttt{Cfinite-ordLess-Cinfinite}) (auto simp: \texttt{cfinite-def card-of-Card-order})

lemma \texttt{ctwo-ordLess-Cinfinite}: \texttt{Cinfinite} \texttt{r} \implies \texttt{ctwo} <_o \texttt{r}
by (rule \texttt{ordLess-ordLeq-trans} [OF \texttt{ctwo-ordLess-natLeq natLeq-ordLeq-cinfinite}])

lemma \texttt{ctwo-ordLeq-Cinfinite}:
assumes \texttt{Cinfinite} \texttt{r}
shows \texttt{ctwo} \leq_o \texttt{r}
by (rule \texttt{ordLess-imp-ordLeq} [OF \texttt{ctwo-ordLess-Cinfinite} [OF \texttt{assms}]])

lemma \texttt{Un-Cinfinite-bound}:
\quad [\quad |A| \leq_o \texttt{r}; |B| \leq_o \texttt{r}; \texttt{Cinfinite} \texttt{r}\quad ] \\implies \\
|A \cup B| \leq_o \texttt{r}
by (auto simp add: \texttt{cinfinite-def card-of-Un-ordLeq-infinite-Field})

lemma \texttt{UNION-Cinfinite-bound}:
\quad [\quad |I| \leq_o \texttt{r}; \forall i \in I. |A i| \leq_o \texttt{r}; \texttt{Cinfinite} \texttt{r}\quad ] \\implies \\
|\bigcup i \in I. A i| \leq_o \texttt{r}
by (auto simp add: \texttt{card-of-UNION-ordLeq-infinite-Field cinfinite-def})

lemma \texttt{csum-cinfinite-bound}:
assumes \texttt{p} \leq_o \texttt{r} \texttt{q} \leq_o \texttt{r} \texttt{Card-order} \texttt{p} \texttt{Card-order} \texttt{q} \texttt{Cinfinite} \texttt{r}
shows \texttt{p} +_c \texttt{q} \leq_o \texttt{r}
proof -
  from \texttt{assms(1-4)} have |Field \texttt{p}| \leq_o \texttt{r} |Field \texttt{q}| \leq_o \texttt{r}
    unfolding \texttt{card-order-on-def} using \texttt{card-of-least ordLeq-transitive} by blast
  with \texttt{assms} show \texttt{?thesis}
    unfolding \texttt{cinfinite-def csum-def}
    by (blast intro: \texttt{card-of-Plus-ordLeq-infinite-Field})
qed

lemma \texttt{cprod-cinfinite-bound}:
assumes \texttt{p} \leq_o \texttt{r} \texttt{q} \leq_o \texttt{r} \texttt{Card-order} \texttt{p} \texttt{Card-order} \texttt{q} \texttt{Cinfinite} \texttt{r}
shows \texttt{p} *_c \texttt{q} \leq_o \texttt{r}
proof -
  from \texttt{assms(1-4)} have |Field \texttt{p}| \leq_o \texttt{r} |Field \texttt{q}| \leq_o \texttt{r}
    unfolding \texttt{card-order-on-def} using \texttt{card-of-least ordLeq-transitive} by blast
  with \texttt{assms} show \texttt{?thesis}
    unfolding \texttt{cinfinite-def cprod-def}
    by (blast intro: \texttt{card-of-Times-ordLeq-infinite-Field})
qed

lemma \texttt{cprod-csum-cexp}:
\quad \texttt{r1} *_c \texttt{r2} \leq_o (\texttt{r1} +_c \texttt{r2}) ^c \texttt{ctwo}
unfolding \texttt{cprod-def csum-def cexp-def ctwo-def Field-card-of}
proof -
  let \texttt{?f} = \lambda(a, b). \%x. if \texttt{x} then \texttt{Inl a} else \texttt{Inr b}
  have \texttt{inj-on ?f} (Field \texttt{r1} \times Field \texttt{r2}) (is \texttt{inj-on - ?LHS})
    by (auto simp: \texttt{inj-on-def fun-eq-iff split: bool.split})
  moreover
  have \texttt{?f \cdot ?LHS} \subseteq \texttt{Func} (UNIV :: bool set) (Field \texttt{r1} <+> Field \texttt{r2}) (is - \subseteq
ultimately show |?LHS| ≤ o |?RHS| using card-of-ordLeq by blast qed

lemma Cfinite-cprod-Cinfinite: [Cfinite r; Cinfinite s] ⇒ r *c s ≤ o s
by (intro cprod-cfinite-bound)
(auto intro: ordLeq-refl ordLess-imp-ordLeq[OF Cfinite-ordLess-Cinfinite])

lemma cprod-cexp: (r *c s) ¸c t = o r ¸c t *c s ¸c t
unfolding cprod-def cexp-def Field-card-of by (rule Func-Times-Range)

lemma cprod-cexp-csum-cexp-Cinfinite:
assumes t: Cinfinite t
shows (r *c s) ¸c t ≤ o (r +c s) ¸c t
proof –
have (r *c s) ¸c t ≤ o ((r +c s) ¸c ctwo) ¸c t
  by (rule cexp-mono1[OF cprod-csum-cexp conjunct2[OF t]])
also have ((r +c s) ¸c ctwo) ¸c t = o (r +c s) ¸c (ctwo *c t)
  by (rule cexp-cprod[OF Card-order-csum])
also have (r +c s) ¸c (ctwo *c t) = o (r +c s) ¸c (t *c ctwo)
  by (rule cexp-cong2[OF cprod-com Card-order-csum Card-order-cprod])
also have ((r +c s) ¸c t) ¸c ctwo = o ((r +c s) ¸c t) ¸c ctwo
  by (rule ordIso-symmetric[OF cexp-cprod[OF Card-order-csum]])
also have ((t +c s) ¸c t) ¸c ctwo = o (r +c s) ¸c t
  by (rule cexp-cprod-ordLeq[OF Card-order-csum t ctwo-Cnotzero ctwo-ordLeq-Cinfinite[OF t]])
finally show ?thesis .

lemma Cfinite-cexp-Cinfinite:
assumes s: Cfinite s and t: Cinfinite t
shows s ¸c t ≤ o ctwo ¸c t
proof (cases s ≤ o ctwo)
case True thus ?thesis using t by (blast intro: cexp-mono1)
next
case False
hence ctwo ≤ o s using ordLeq-total[of s ctwo] Card-order-ctwo s
  by (auto intro: card-order-on-well-order-on)
hence Cnotzero s using Cnotzero-mono[OF ctwo-Cnotzero] s by blast
hence st: Cnotzero (s *c t) by (intro Cinfinite-Cnotzero[OF Cfinite-cprod2])
(auto simp: t)
have s ¸c t ≤ o (ctwo ¸c s) ¸c t
  using assms by (blast intro: cexp-mono1 ordLess-imp-ordLeq[OF ordLess-ctwo-cexp])
also have (ctwo ¸c s) ¸c t = o ctwo ¸c (s *c t)
  by (blast intro: Card-order-ctwo cexp-cprod)
also have ctwo ¸c (s *c t) ≤ o ctwo ¸c t
  using assms st by (intro cexp-mono2-Cnotzero Cfinite-cprod-Cinfinite Card-order-ctwo)
finally show ?thesis .
qed

lemma csun-\text{Cfinite}-csun-\text{Cfinite}:
  assumes \( r: \text{Card-order} \) \text{ and } \( s: \text{Cfinite} \) \text{ and } \( t: \text{Cinfinite} \)
  shows \( (r + c s) ^{c t} \leq o (r + c \text{ctwo}) ^{c t} \)
proof (cases \text{Cfinite} r)
  case True
  hence \( r + c s = o r \) by (intro csun-absorb1 ordLess-imp-ordLeq[OF Cfinite-ordLess-Cinfinite]
  s)
  hence \( (r + c s) ^{c t} = o r ^{c t} \) \text{ using } \( (\text{blast intro: cexp-cong1}) \)
  also have \( r ^{c t} \leq o (r + c \text{ctwo}) ^{c t} \) \text{ using } \( (\text{blast intro: cexp-mono1}
  ordLeq-csun1 r) \)
  finally show \( \text{thesis} \).
next
  case False
  with \( r \) have \( \text{Cfinite} r \) \text{ unfolding cfinite-def cfinite-def} \text{ by auto}
  hence \( \text{Cfinite} (r + c s) \) \text{ by (intro Cfinite-csun s)}
  hence \( (r + c s) ^{c t} \leq o \text{ctwo} ^{c t} \) \text{ by (intro Cfinite-cexp-Cinfinite t)}
  also have \( \text{ctwo} ^{c t} \leq o (r + c \text{ctwo}) ^{c t} \) \text{ using } \( (\text{blast intro: cexp-mono1 ordLeq-csun2 Card-order-ctwo}) \)
  finally show \( \text{thesis} \).
qed

lemma Cinfinite-cardSuc: \( \text{Cinfinite} r \implies \text{Cinfinite} (\text{cardSuc} r) \)
by (simp add: cfinite-def cardSuc-Card-order cardSuc-finite)

lemma cardSuc-UNION-Cinfinite:
  assumes \( \text{Cinfinite} r \ \text{relChain} (\text{cardSuc} r) \ As B \leq (\text{UN } i : \text{Field} (\text{cardSuc} r). As i) \) \( |B| = o r \)
  shows \( \text{EX } i : \text{Field} (\text{cardSuc} r). B \leq As i \)
using cardSuc-UNION assms unfolding cfinite-def by blast

end

33  Fun-Def-Base: Function Definition Base

theory Fun-Def-Base
imports Ctr-Sugar Set Wellfounded
begin

ML-file Tools/Function/function-lib.ML
ML-file Tools/Function/function-common.ML
ML-file Tools/Function/context-tree.ML
setup Function-Ctx-Tree.setup
ML-file Tools/Function/sum-tree.ML

end
34  BNF-Def: Definition of Bounded Natural Functors

theory BNF-Def
imports BNF-Cardinal-Arithmetic Fun-Def-Base

begin

definition rel-fun where
  rel-fun A B = (λf g. ∀x y. A x y ⇒ B (f x) (g y))

lemma rel-funI [intro]:
  assumes ∀x y. A x y ⇒ B (f x) (g y)
  shows rel-fun A B f g
  using assms by (simp add: rel-fun-def)

lemma rel-funD:
  assumes rel-fun A B f g and A x y
  shows B (f x) (g y)
  using assms by (simp add: rel-fun-def)

definition collect where
  collect F x = (∪f ∈ F. f x)

lemma fstI: x = (y, z) ⇒ fst x = y
  by simp

lemma sndI: x = (y, z) ⇒ snd x = z
  by simp

lemma bijI: [∀x y. (f x = f y) = (x = y); ∃y. y = f x] ⇒ bij f
  unfolding bij-def inj-on-def by auto blast

definition Gr A f = {(a, f a) | a. a ∈ A}

definition Grp A f = (λa b. b = f a ∧ a ∈ A)

definition vimage2p where
  vimage2p f g R = (λx y. R (f x) (g y))

lemma collect-comp: collect F ∘ g = collect ((λf. f ∘ g) ∘ F)
  by (rule ext) (auto simp only: comp-apply collect-def)
definition \textit{convol} \(((\cdot, \cdot))\) where 
\((f, g) \equiv \lambda a. (f a, g a)\)

\textbf{lemma} \textit{fst-convol}: 
\(\text{fst} \circ (f, g) = f\) 
\textbf{apply (rule ext)} 
\textbf{unfolding} \textit{convol-def} \textbf{by simp}

\textbf{lemma} \textit{snd-convol}: 
\(\text{snd} \circ (f, g) = g\) 
\textbf{apply (rule ext)} 
\textbf{unfolding} \textit{convol-def} \textbf{by simp}

\textbf{lemma} \textit{convol-mem-GrpI}: 
\(x \in A \Rightarrow \langle \text{id}, g \rangle x \in (\text{Collect} (\text{split} (\text{Grp} A g)))\) 
\textbf{unfolding} \textit{convol-def \text{Grp-def} by auto}

definition \textit{csquare} where 
\(\text{csquare} A f1 f2 p1 p2 \leftrightarrow (\forall a \in A. f1 (p1 a) = f2 (p2 a))\)

\textbf{lemma} \textit{eq-alt}: \(\text{op} = = \text{Grp} \text{ UNIV id}\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{leq-conversepI}: \(R = = \Rightarrow R \leq R^\leftrightarrow\) 
\textbf{by auto}

\textbf{lemma} \textit{leq-OOI}: \(R = = \Rightarrow R \leq R \text{ OO } R\) 
\textbf{by auto}

\textbf{lemma} \textit{OO-Grp-alt}: \((\text{Grp} A f)^\leftrightarrow\text{OO} \text{ Grp} A g = (\lambda x y. \exists z. z \in A \land f z = x \land g z = y)\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{Grp-UNIV-id}: \(f = \text{id} \Rightarrow (\text{Grp} \text{ UNIV f})^\leftrightarrow\text{OO} \text{ Grp} \text{ UNIV f} = \text{Grp} \text{ UNIV f}\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{Grp-UNIV-idI}: \(x = x \Rightarrow \text{Grp} \text{ UNIV} id x y\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{Grp-mono}: \(A \leq B \Rightarrow \text{Grp} A f \leq \text{Grp} B f\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{GrpI}: \([f x = y; x \in A] \Rightarrow \text{Grp} A f x y\) 
\textbf{unfolding} \textit{Grp-def by auto}

\textbf{lemma} \textit{GrpE}: \(\text{Grp} A f x y \Rightarrow ([f x = y; x \in A] \Rightarrow R) \Rightarrow R\) 
\textbf{unfolding} \textit{Grp-def by auto}
lemma Collect-split-Grp-eqD: $z \in \text{Collect} (\text{split} (\text{Grp} A f)) \Longrightarrow (f \circ \text{fst}) z = \text{snd} z$
unfolding Grp-def comp-def by auto

lemma Collect-split-Grp-inD: $z \in \text{Collect} (\text{split} (\text{Grp} A f)) \Longrightarrow \text{fst} z \in A$
unfolding Grp-def comp-def by auto

definition pick-middlep $P$ $Q$ $a$ $c$ = ($\text{SOME} b$. $P a b \land Q b c$)

lemma pick-middlep: $(P \OO Q) a c \Longrightarrow P a (\text{pick-middlep} P Q a c) \land Q (\text{pick-middlep} P Q a c) c$
unfolding pick-middlep-def apply(rule someI-ex) by auto

definition fstOp where $\text{fstOp} P Q ac = (\text{fst} ac, \text{pick-middlep} P Q (\text{fst} ac) (\text{snd} ac), (\text{snd} ac))$
definition sndOp where $\text{sndOp} P Q ac = (\text{pick-middlep} P Q (\text{fst} ac) (\text{snd} ac), (\text{snd} ac))$

lemma fstOp-in: $ac \in \text{Collect} (\text{split} (P \OO Q)) \Longrightarrow \text{fstOp} P Q ac \in \text{Collect} (\text{split} P)$
unfolding fstOp-def mem-Collect-eq by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct1])

lemma fst-fstOp: $\text{fst} bc = (\text{fst} \circ \text{fstOp} P Q) bc$
unfolding comp-def fstOp-def by simp

lemma snd-sndOp: $\text{snd} bc = (\text{snd} \circ \text{sndOp} P Q) bc$
unfolding comp-def sndOp-def by simp

lemma sndOp-in: $ac \in \text{Collect} (\text{split} (P \OO Q)) \Longrightarrow \text{sndOp} P Q ac \in \text{Collect} (\text{split} Q)$
unfolding sndOp-def mem-Collect-eq by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct2])

lemma csquare-fstOp-sndOp: $\text{csquare} (\text{Collect} (\text{split} (P \OO Q))) \text{snd} \text{fst} (\text{fstOp} P Q) (\text{sndOp} P Q)$
unfolding csquare-def fstOp-def sndOp-def using pick-middlep by simp

lemma snd-fst-flip: $\text{snd} xy = (\text{fst} \circ (\% (x, y) . (y, x))) xy$
by (simp split: prod.split)

lemma fst-snd-flip: $\text{fst} xy = (\text{snd} \circ (\% (x, y) . (y, x))) xy$
by (simp split: prod.split)

lemma flip-pred: $A \subseteq \text{Collect} (\text{split} (R ^{--1})) \Longrightarrow (\% (x, y) . (y, x)) ^{'} A \subseteq \text{Collect} (\text{split} R)$
by auto
lemma Collect-split-mono: $A \leq B \Rightarrow \text{Collect}(\text{split } A) \subseteq \text{Collect}(\text{split } B)$
  by auto

lemma Collect-split-mono-strong:
  $[ X = \text{fst} \cdot A; Y = \text{snd} \cdot A; \forall a \in X. \forall b \in Y. P a b \rightarrow Q a b; A \subseteq \text{Collect}(\text{split } P)] \Rightarrow A \subseteq \text{Collect}(\text{split } Q)$
  by fastforce

lemma predicate2-eqD: $A = B \Rightarrow A a b \leftrightarrow B a b$
  by simp

lemma case-sum-o-inj:
  case-sum $f g \circ \text{Inl} = f$
  case-sum $f g \circ \text{Inr} = g$
  by auto

lemma card-order-csum-cone-cexp-def:
  $\text{card-order } r \Rightarrow (|A1| + c \text{ cone} \cdot c r = |\text{Func } \text{UNIV} \cdot (\text{Inl} \cdot A1 \cup \{\text{Inr} ()\})|)$
  unfolding cexp-def cone-def Field-csum Field-card-of by (auto dest: Field-card-order)

lemma If-the-inv-into-in-Func:
  $[\text{inj-on } g C; C \subseteq B \cup \{x\}] \Rightarrow (\lambda i. \text{if } i \in g \cdot C \text{ then the-inv-into } C g i \text{ else } x) \in \text{Func } \text{UNIV} (B \cup \{x\})$
  unfolding Func-def by (auto dest: the-inv-into-into)

lemma If-the-inv-into-f-f:
  $[i \in C; \text{inj-on } g C] \Rightarrow ((\lambda i. \text{if } i \in g \cdot C \text{ then the-inv-into } C g i \text{ else } x) \circ g) i = \text{id } i$
  unfolding Func-def by (auto elim: the-inv-into-f-f)

lemma the-inv-f-o-f-id: $\text{inj } f \Rightarrow (\text{the-inv } f \circ f) z = \text{id } z$
  by (simp add: the-inv-f-f)

lemma vimage2pI: $R (f z) (g y) \Rightarrow vimage2p f g R x y$
  unfolding vimage2p-def by –

lemma rel-fun-iff-leq-vimage2p: $(\text{rel-fun } R S) f g = (R \leq vimage2p f g S)$
  unfolding rel-fun-def vimage2p-def by auto

lemma convol-image-vimage2p: $(f \circ \text{fst}, g \circ \text{snd}) \cdot \text{Collect}(\text{split} (vimage2p f g R)) \subseteq \text{Collect}(\text{split } R)$
  unfolding vimage2p-def convol-def by auto

lemma vimage2p-Grp: $\text{vimage2p } f g P = \text{Grp } \text{UNIV } f \text{ OO } P \text{ OO } (\text{Grp } \text{UNIV } g)^{-1}_{-1}$
  unfolding vimage2p-def Grp-def by auto
ML-file Tools/BNF/bnf-util.ML
ML-file Tools/BNF/bnf-tactics.ML
ML-file Tools/BNF/bnf-def-tactics.ML
ML-file Tools/BNF/bnf-def.ML
end

35 BNF-Comp: Composition of Bounded Natural Functors

theory BNF-Comp
imports BNF-Def
begin

lemma empty-natural: \((\lambda \cdot \{}\} \circ f = \text{image } g \circ (\lambda \cdot \{}\})\)
by (rule ext) simp

lemma Union-natural: \(\text{Union } \circ \text{image } f = \text{image } f \circ \text{Union}\)
by (rule ext) (auto simp only: comp-apply)

lemma in-Union-o-assoc: \(x \in (\text{Union } \circ \text{gset } \circ \text{gmap}) A \Rightarrow x \in (\text{Union } \circ (\text{gset } \circ \text{gmap})) A\)
by (unfold comp-assoc)

lemma comp-single-set-bd:
assumes fbd-Card-order: Card-order fbd and
fset-bd: \(\forall x. |\text{fset } x| \leq o fbd\) and
gset-bd: \(\forall x. |\text{gset } x| \leq o gbd\)
shows \(|\bigcup (\text{fset } \circ \text{gset } x)| \leq o gbd \ast c fbd\)
apply simp
apply (rule ordLeq-transitive)
apply (rule card-of-UNION-Sigma)
apply (subst SIGMA-CSUM)
apply (rule ordLeq-transitive)
apply (rule card-of-Csum-Times')
apply (rule fbd-Card-order)
apply (rule ballI)
apply (rule fset-bd)
apply (rule ordLeq-transitive)
apply (rule cprod-mono1)
apply (rule gset-bd)
apply (rule ordIso-imp-ordLeq)
apply (rule ordIso-refl)
apply (rule Card-order-cprod)
done

lemma csum-dup: cinfinite r \(
\Rightarrow Card-order r \Rightarrow p + c p' = o r + c r \Rightarrow p + c\)}
THEORY “BNF-Comp”

\[ p' = o \ r \]

apply (erule ordIso-transitive)
apply (frule csum-absorb2')
apply (erule ordLeq-refl)
by simp

lemma cprod-dup: cinfinite r \implies Card-order r \implies p \ast c p' = o \ r \ast c r \implies p \ast c
\[ p' = o \ r \]

apply (erule ordIso-transitive)
apply (rule cprod-infinite)
by simp

lemma Union-image-insert: \( \bigcup (f \ i \ insert \ a \ B) = f \ a \cup \bigcup (f \ i \ B) \)
by simp

lemma Union-image-empty: \( A \cup \bigcup (f \ i \ \{\}) = A \)
by simp

lemma image-o-collect: collect ((\lambda f. image g o f) \cdot F) = image g o collect F
by (rule ext) (auto simp add: collect-def)

lemma conj-subset-def: \( A \subseteq \{x. P x \land Q x\} = (A \subseteq \{x. P x\} \land A \subseteq \{x. Q x\}) \)
by blast

lemma UN-image-subset: \( \bigcup (f \ i \ g \ x) \subseteq X = (g \ x \subseteq \{x. f \ x \subseteq X\}) \)
by blast

lemma comp-set-bd-Union-o-collect: \( |\bigcup \bigcup (\lambda f. f \ x) \cdot X | \leq o \ hbd \implies |(\bigcup \circ collect \ X) \cdot x | \leq o \ hbd \)
by (unfold comp-apply collect-def) simp

lemma wpull-cong:
\[
\begin{align*}
[A' = A; B1' = B1; B2' = B2; wpull A B1 B2 f1 f2 p1 p2] \implies wpull A' B1' B2' f1 f2 p1 p2
\end{align*}
\]
by simp

lemma Grp-fst-snd: (Grp (Collect (split R)) \cdot fst) -\cdot 1 \cdot OO \cdot Grp (Collect (split R))
\text{snd} = R

unfolding Grp-def fun-eq-iff relcompp.simps by auto

lemma OO-Grp-cong: A = B \implies (Grp A f) -\cdot 1 \cdot OO \cdot Grp A g = (Grp B f) -\cdot 1
\cdot OO \cdot Grp B g
by (rule arg-cong)

lemma vimage2p-relcompp-mono: R OO S \leq T \implies vimage2p f g R OO vimage2p g h S \leq vimage2p f h T

unfolding vimage2p-def by auto

lemma type-copy-map-cong0: M (g x) = N (h x) \implies (f o M o g) x = (f o N o
h) \( x \)
\begin{verbatim}
by auto

lemma type-copy-set-bd: \((\forall y. |S y| \leq o bd) \implies |(S o Rep) x| \leq o bd\)
by auto

lemma vimage2p-cong: \( R = S \implies \text{vimage2p } f g R = \text{vimage2p } f g S \)
by simp

context
fixes Rep Abs
assumes type-copy: type-definition Rep Abs UNIV
begin

lemma type-copy-map-id0: \( M = id \implies Abs o M o Rep = id \)
using type-definition.Abs-inverse[OF type-copy UNIV-I]
by auto

lemma type-copy-map-comp0: \( M = M1 o M2 \implies f o M o g = (f o M1 o Rep) o (Abs o M2 o g) \)
using type-definition.Abs-inverse[OF type-copy UNIV-I]
by (auto simp: o-def fun-eq-iff)

lemma type-copy-set-map0: \( S o M = \text{image } f o S' \implies (S o Rep) o (Abs o M o g) = \text{image } f o (S' o g) \)
using type-definition.Abs-inverse[OF type-copy UNIV-I]
by (auto simp: o-def fun-eq-iff)

lemma type-copy-wit: \( x \in (S o Rep) (Abs y) \implies x \in S y \)
using type-definition.Abs-inverse[OF type-copy UNIV-I]
by auto

lemma type-copy-vimage2p-Grp-Rep: \( \text{vimage2p } f Rep (\text{Grp } (\text{Collect } P) h) = \text{Grp } (\text{Collect } (\lambda x. P (f x))) (Abs o h o f) \)
unfolding vimage2p-def Grp-def fun-eq-iff
by (auto simp: type-definition.Abs-inverse[OF type-copy UNIV-I]
type-definition.Rep-inverse[OF type-copy dest: smg])

lemma type-copy-vimage2p-Grp-Abs: \( \forall h. \text{vimage2p } g Abs (\text{Grp } (\text{Collect } P) h) = \text{Grp } (\text{Collect } (\lambda x. P (g x))) (Rep o h o g) \)
unfolding vimage2p-def Grp-def fun-eq-iff
by (auto simp: type-definition.Abs-inverse[OF type-copy UNIV-I]
type-definition.Rep-inverse[OF type-copy dest: smg])

lemma type-copy-ex-RepI: \( \exists b. F b = (\exists b'. F (Rep b')) \)
proof safe
  fix b assume F b
  show \( \exists b'. F (Rep b') \)
  proof (rule exI)
    from :F b show F (Rep (Abs b)) using type-definition.Abs-inverse[OF type-copy]
    by auto
  qed

end
\end{verbatim}
lemma vimage2p-relcompp-converse:
  vimage2p f g (R`:−→ 1 OO S) = (vimage2p Rep f R)`:−→ 1 OO vimage2p Rep g S
unfolding vimage2p-def relcompp.simps conversep.simps fun-eq-iff image-def
by (auto simp: type-copy-ex-RepI)
end

bnf DEADID: 'a
  map: id :: 'a ⇒ 'a
  bd: natLeq
  rel: op = :: 'a ⇒ 'a ⇒ bool
by (auto simp add: Grp-def natLeq-card-order natLeq-cinfinite)
definition id-bnf-comp :: 'a ⇒ 'a where id-bnf-comp ≡ (λx. x)
lemma id-bnf-comp-apply: id-bnf-comp x = x
unfolding id-bnf-comp-def by simp

bnf ID: 'a
  map: id-bnf-comp :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b
  sets: λx. {x}
  bd: natLeq
  rel: id-bnf-comp :: ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool
unfolding id-bnf-comp-def
apply (auto simp: Grp-def fun-eq-iff relcompp.simps natLeq-card-order natLeq-cinfinite)
apply (rule ordLess-imp-ordLeq[OF finite-ordLess-infinite[OF - natLeq-Well-order]])
apply (auto simp add: Field-card-of Field-natLeq card-of-well-order-on)[3]
done

lemma type-definition-id-bnf-comp-UNIV: type-definition id-bnf-comp id-bnf-comp UNIV
unfolding id-bnf-comp-def by unfold-locales auto

ML-file Tools/BNF/bnf-comp-tactics.ML
ML-file Tools/BNF/bnf-comp.ML
hide-const (open) id-bnf-comp
hide-fact (open) id-bnf-comp-def type-definition-id-bnf-comp-UNIV
end

36 Basic-BNFs: Registration of Basic Types as Bounded Natural Functors

theory Basic-BNFs
imports BNF-Def
begin

definition setl :: 'a + 'b ⇒ 'a set where
setl x = (case x of Inl z => {z} | -. => {})
definition setr :: 'a + 'b ⇒ 'b set where
setr x = (case x of Inr z => {z} | -. => {})

lemmas sum-set-defs = setl-def[abs-def] setr-def[abs-def]
definition rel-sum :: ('a ⇒ 'c ⇒ bool) ⇒ ('b ⇒ 'd ⇒ bool) ⇒ 'a + 'b ⇒ 'c + 'd ⇒ bool
where
rel-sum R1 R2 x y =
(case (x, y) of (Inl x, Inl y) ⇒ R1 x y
| (Inr x, Inr y) ⇒ R2 x y
| - ⇒ False)

lemma rel-sum-simps[simp]:
rel-sum R1 R2 (Inl a1) (Inl b1) = R1 a1 b1
rel-sum R1 R2 (Inl a1) (Inr b2) = False
rel-sum R1 R2 (Inr a2) (Inl b1) = False
rel-sum R1 R2 (Inr a2) (Inr b2) = R2 a2 b2

unfolding rel-sum-def by simp-all

bnf 'a + 'b
map: map-sum
sets: setl setr
bd: natLeq
wits: Inl Inr
rel: rel-sum

proof –
show map-sum id id = id by (rule map-sum.id)
next
fix f1 :: 'o ⇒ 's and f2 :: 'p ⇒ 't and g1 :: 's ⇒ 'q and g2 :: 't ⇒ 'r
show map-sum (g1 o f1) (g2 o f2) = map-sum g1 g2 o map-sum f1 f2
by (rule map-sum.comp[symmetric])
next
fix x and f1 :: 'o ⇒ 'q and f2 :: 'p ⇒ 'r and g1 g2
assume a1: \(\forall z. z \in \text{setl} \ x \implies f1 \ z = g1 \ z \) and
a2: \(\forall z. z \in \text{setr} \ x \implies f2 \ z = g2 \ z \)
thus map-sum f1 f2 x = map-sum g1 g2 x

proof (cases x)
  case Inl thus ?thesis using a1 by (clarsimp simp: setl-def)
next
  case Inr thus ?thesis using a2 by (clarsimp simp: setr-def)
qed
next
THEORY “Basic-BNFs”

fix \( f_1 :: 'o \Rightarrow 'q \) and \( f_2 :: 'p \Rightarrow 'r \)
show \( \text{setl} \circ \text{map-sum} \ f_1 \ f_2 = \text{image} \ f_1 \circ \text{setl} \)
  by (rule ext, unfold o-apply) (simp add: setl-def split: sum.split)

next
fix \( f_1 :: 'o \Rightarrow 'q \) and \( f_2 :: 'p \Rightarrow 'r \)
show \( \text{setr} \circ \text{map-sum} \ f_1 \ f_2 = \text{image} \ f_2 \circ \text{setr} \)
  by (rule ext, unfold o-apply) (simp add: setr-def split: sum.split)

next
show \( \text{card-order} \ \text{natLeq} \) by (rule natLeq-card-order)

next
show \( \infinite \ \text{natLeq} \) by (rule natLeq-cinfinite)

next
fix \( x :: 'o + 'p \)
show \( \| \text{setl} \ x \| \leq o \ \text{natLeq} \)
  apply (rule ordLess-imp-ordLeq)
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: setl-def split: sum.split)

next
fix \( x :: 'o + 'p \)
show \( \| \text{setr} \ x \| \leq o \ \text{natLeq} \)
  apply (rule ordLess-imp-ordLeq)
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: setr-def split: sum.split)

next
fix \( R_1 \ R_2 \ S_1 \ S_2 \)
show \( \text{rel-sum} \ R_1 \ R_2 \ OO \ \text{rel-sum} \ S_1 \ S_2 \leq \text{rel-sum} \ (R_1 \ OO \ S_1) \ (R_2 \ OO \ S_2) \)
  by (auto simp: rel-sum-def split: sum.splits)

next
fix \( R \ S \)
show \( \text{rel-sum} \ R \ S = \)
  \( (\text{Grp} \ \{ x. \ \text{setl} \ x \subseteq \text{Collect} \ (\text{split} \ R) \land \ \text{setr} \ x \subseteq \text{Collect} \ (\text{split} \ S) \} \ (\text{map-sum} \ \text{fst} \ \text{fst}))^{-1} \ OO \)
  \( (\text{Grp} \ \{ x. \ \text{setl} \ x \subseteq \text{Collect} \ (\text{split} \ R) \land \ \text{setr} \ x \subseteq \text{Collect} \ (\text{split} \ S) \} \ (\text{map-sum} \ \text{snd} \ \text{snd})) \)
  unfolding setl-def setr-def rel-sum-def Grp-def relcompp.simps conversep.simps fun-eq-iff
  by (fastforce split: sum.splits)
qed (auto simp: sum-set-defs)

definition \( \text{fsts} :: 'a 	imes 'b \Rightarrow 'a \ \text{set} \ \text{where} \)
  \( \text{fsts} \ x = \{ \text{fst} \ x \} \)

definition \( \text{snds} :: 'a 	imes 'b \Rightarrow 'b \ \text{set} \ \text{where} \)
  \( \text{snds} \ x = \{ \text{snd} \ x \} \)

lemmas \( \text{prod-set-defs} = \text{fsts-def}[\text{abs-def}] \ \text{snds-def}[\text{abs-def}] \)

definition \( \text{rel-prod} :: ('a \Rightarrow 'b \Rightarrow 'bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow 'bool) \Rightarrow 'a 	imes 'c \Rightarrow 'b 	imes 'd \Rightarrow 'bool \)
THEORY "Basic-BNFs"

where
rel-prod R1 R2 = (λ(a, b) (c, d). R1 a c ∧ R2 b d)

lemma rel-prod-apply [simp]:
rel-prod R1 R2 (a, b) (c, d) ←→ R1 a c ∧ R2 b d
by (simp add: rel-prod-def)

bnf 'a × 'b
map: map-prod
sets: fsts snds
bd: natLeq
rel: rel-prod
proof (unfold prod-set-defs)
show map-prod id id = id by (rule map-prod.id)
next
fix f1 f2 g1 g2
show map-prod (g1 o f1) (g2 o f2) = map-prod g1 g2 o map-prod f1 f2
by (rule map-prod.comp[symmetric])
next
fix x f1 f2 g1 g2
assume ∃z. z ∈ {fst x} → f1 z = g1 z ∧ ∃z. z ∈ {snd x} → f2 z = g2 z
thus map-prod f1 f2 x = map-prod g1 g2 x by (cases x) simp
next
fix f1 f2
show (λx. {fst x}) o map-prod f1 f2 = image f1 o (λx. {fst x})
by (rule ext, unfold o-apply) simp
next
fix f1 f2
show (λx. {snd x}) o map-prod f1 f2 = image f2 o (λx. {snd x})
by (rule ext, unfold o-apply) simp
next
show card-order natLeq by (rule natLeq-card-order)
next
show cinfinite natLeq by (rule natLeq-cinfinite)
next
fix x
show |{fst x}| ≤ o natLeq
by (rule ordLess-imp-ordLeq) (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix x
show |{snd x}| ≤ o natLeq
by (rule ordLess-imp-ordLeq) (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix R1 R2 S1 S2
show rel-prod R1 R2 OO rel-prod S1 S2 ≤ rel-prod (R1 OO S1) (R2 OO S2)
by auto
next
fix R S
show rel-prod R S =

(Grp \{x. \{fst x\} \subseteq \text{Collect} (\text{split} R) \land \{snd x\} \subseteq \text{Collect} (\text{split} S)\} \text{map-prod}
\text{fst} \text{fst})^{-1} O0
Grp \{x. \{fst x\} \subseteq \text{Collect} (\text{split} R) \land \{snd x\} \subseteq \text{Collect} (\text{split} S)\} \text{map-prod}
snd \text{snd})
\text{unfolding} \text{prod-set-defs rel-prod-def Grp-def relcompp.simps conversep.simps fun-eq-iff
by auto
qed

bnf \ 'a \Rightarrow \ 'b
map: op \circ
sets: range
bd: natLeq \{+[UNIV :: \ 'a set]\}
rel: rel-fun op =
proof
fix f show id \circ f = id f by simp
next
fix f g show op \circ (g \circ f) = op \circ g \circ op \circ f
\text{unfolding} \text{comp-def[abs-def]} ..
next
fix x f g
assume \land z. z \in \text{range} x \Longrightarrow f z = g z
thus f \circ x = g \circ x by auto
next
fix f show range \circ op \circ f = op \ast f \circ range
by (auto simp add: fun-eq-iff)
next
show card-order (natLeq \{+[UNIV]\}) (\text{is - ("+c ?U")})
apply (rule card-order-csum)
apply (rule natLeq-card-order)
by (rule card-of-card-order-on)

show cinfinite (natLeq \{+[?U]\})
apply (rule cinfinite-csum)
apply (rule disjI1)
by (rule natLeq-cinfinite)
next
fix f :: \ 'd =\Rightarrow \ 'a
have |range f| \leq o \ | (UNIV::\ 'd set) | (\text{is - ("+c ?U")}) by (rule card-of-image)
also have \ ?U \leq o \text{natLeq} +c \ ?U by (rule ordLeq-csum2) (rule card-of-Card-order)
finally show |range f| \leq o \text{natLeq} +c \ ?U .
next
fix R S
show rel-fun op = R OO rel-fun op = S \leq rel-fun op = (R OO S) by (auto simp: rel-fun-def)
next
fix R
show rel-fun op = R =
(Grp \{x. \text{range} x \subseteq \text{Collect} (\text{split} R)\} (op \circ \text{fst}))^{-1} O0
Grp \{x. \text{range} x \subseteq \text{Collect} (\text{split} R)\} (op \circ \text{snd})
unfolding rel-fun-def Grp-def fun-eq-iff relcompp simps conversep.simps subset-iff
proof (safe, unfold fun-eq-iff[symmetric])
fix x xa a b c xb y aa ba
assume *: x = a xa = c a = ba b = aa c = (λx. snd (b x)) ba = (λx. fst (aa x))
and
**: ∀ t. (∃ x. t = aa x) −→ R (fst t) (snd t)
show R (x y) (xa y) unfolding * by (rule mp[OF spec[OF **]]) blast
qed force
qed

37 BNF-FP-Base: Shared Fixed Point Operations on Bounded Natural Functors

theory BNF-FP-Base
imports BNF-Comp Basic-BNFs
begin

lemma False-impliesTrue: (False ⇒ Q) ≡ Trueprop True
  by default simp-all

lemma conj-impliesimplies: (P ∧ Q ⇒ PROP R) ≡ (P ⇒ Q ⇒ PROP R)
  by default simp-all

lemma mp-conj: (P ⇒ Q) ∧ R ⇒ P ⇒ R ∧ Q
  by auto

lemma predicate2D-conj: P ≤ Q ∧ R ⇒ P x y ⇒ R ∧ Q x y
  by auto

lemma eq-sym-Unity-conv: (x = (() = ())) = x
  by blast

lemma case-unit-Unity: (case u of () ⇒ f) = f
  by (cases u) (hypsubst, rule unit.case)

lemma case-prod-Pair-iden: (case p of (x, y) ⇒ (x, y)) = p
  by simp

lemma unit-all-impI: (P () ⇒ Q ()) ⇒ ∀ x. P x ⇒ Q x
  by simp

lemma pointfree-idE: f ◦ g = id ⇒ f (g x) = x
unfolding comp-def fun-eq-iff by simp
lemma o-bij:
  assumes \( gf : g \circ f = \text{id} \) and \( fg : f \circ g = \text{id} \)
  shows bij f
unfolding bij-def inj-on-def surj-def proof safe
  fix \( a1 \) \( a2 \) assume \( f a1 = f a2 \)
  hence \( g (f a1) = g (f a2) \) by simp
  thus \( a1 = a2 \) using \( gf \) unfolding fun-eq-iff by simp
next
  fix \( b \) have \( b = f (g b) \)
      using \( fg \) unfolding fun-eq-iff by simp
  thus \( \exists a. \ b = f a \) by blast
qed

lemma ssubst-mem: \([t = s; s \in X]\) \( \Rightarrow t \in X \) by simp

lemma case-sum-step:
case-sum \( (\text{case-sum } f' g') \) \( g \) \( \text{Inl } p \) = case-sum \( f' g' \) \( p \)
case-sum \( f \) \( \text{case-sum } f' g' \) \( \text{Inr } p \) = case-sum \( f' g' \) \( p \)
by auto

lemma obj-one-pointE: \( \forall x. \ s = x \rightarrow P \rightarrow P \)
by blast

lemma type-copy-obj-one-point-absE:
  assumes \( \text{type-definition } \text{Rep} \ \text{Abs} \ \text{UNIV} \ \forall x. \ s = \text{Abs } x \rightarrow P \) shows \( P \)
  using \( \text{type-definition.} \text{Rep-inverse}[OF assms(1)] \)
  by (intro mp[OF spec[OF assms(2), of Rep s]]) simp

lemma obj-sumE-f:
  assumes \( \forall x. \ s = f (\text{Inl } x) \rightarrow P \) \( \forall x. \ s = f (\text{Inr } x) \rightarrow P \)
  shows \( \forall x. \ s = f x \rightarrow P \)
proof
  fix \( x \) from assms show \( s = f x \rightarrow P \) by (cases \( x \)) auto
qed

lemma case-sum-if:
case-sum \( f \) \( g \) \( (\text{if } p \text{ then } \text{Inl } x \text{ else } \text{Inr } y) \) = \( (\text{if } p \text{ then } f x \text{ else } g y) \)
by simp

lemma prod-set-simps:
  \( \text{fsts } (x, y) = \{x\} \)
  \( \text{snds } (x, y) = \{y\} \)
unfolding fsts-def snds-def by simp+

lemma sum-set-simps:
  \( \text{setl } (\text{Inl } x) = \{x\} \)
  \( \text{setl } (\text{Inr } x) = \{\} \)
  \( \text{setr } (\text{Inl } x) = \{\} \)
setr (Inr x) = {x}

unfolding sum-set-defs by simp

lemma Inl-Inr-False: (Inl x = Inr y) = False
    by simp

lemma Inr-Inl-False: (Inr x = Inl y) = False
    by simp

lemma spec2: ∀ x y. P x y =⇒ P x y
    by blast

lemma rewriteR-comp-comp: [ g ◦ h = r ] =⇒ f ◦ g ◦ h = f ◦ r
    unfolding comp-def fun-eq-iff by auto

lemma rewriteR-comp-comp2: [ g ◦ h = r1 ◦ r2; f ◦ r1 = l ] =⇒ f ◦ g ◦ h = l ◦ r2
    unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp: [ f ◦ g = l ] =⇒ f ◦ (g ◦ h) = l ◦ h
    unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp2: [ f ◦ g = l1 ◦ l2; l2 ◦ h = r ] =⇒ f ◦ (g ◦ h) = l1 ◦ r
    unfolding comp-def fun-eq-iff by auto

lemma convol-o: (f, g) ◦ h = (f ◦ h, g ◦ h)
    unfolding convol-def by auto

lemma map-prod-o-convol: map-prod h1 h2 ◦ (f, g) = (h1 ◦ f, h2 ◦ g)
    unfolding convol-def by auto

lemma map-prod-o-convol-id: (map-prod f id ◦ (id, g)) x = (id ◦ f, g) x
    unfolding map-prod-o-convol id-comp comp-id ..

lemma o-case-sum: h ◦ case-sum f g = case-sum (h ◦ f) (h ◦ g)
    unfolding comp-def by (auto split: sum.splits)

lemma case-sum-o-map-sum: case-sum f g ◦ map-sum h1 h2 = case-sum (f ◦ h1) (g ◦ h2)
    unfolding comp-def by (auto split: sum.splits)

lemma case-sum-o-map-sum-id: (case-sum id g ◦ map-sum f id) x = case-sum (f ◦ id) g x
    unfolding case-sum-o-map-sum id-comp comp-id ..

lemma rel-fun-def-butlast:
    rel-fun R (rel-fun S T) f g = (∀ x y. R x y → (rel-fun S T) (f x) (g y))
    unfolding rel-fun-def ..
lemma subst-eq-imp: (∀a b. a = b → P a b) ≡ (∀a. P a a)
  by auto

lemma eq-subset: op = (λa b. P a b ∨ a = b)
  by auto

lemma eq-le-Grp-id-iff: (op = ≤ (λx. x ∈ P ⇒ f x ∈ Q)) ≡
  (Grp P id x y → Grp Q id (f x) (f y))
  unfolding Grp-def by rule auto

lemma Grp-id-mono-subst: (⋀x y. Grp P id x y =⇒ Grp Q id (f x) (f y)) ≡
  (⋀x. x ∈ P =⇒ f x ∈ Q)
  unfolding Grp-def by rule auto

lemma vimage2p-mono: vimage2p f g R x y =⇒ R ≤ S =⇒ vimage2p f g S x y
  unfolding vimage2p-def by blast

lemma vimage2p-refl: (⋀x. R x x) =⇒ vimage2p f f R x x
  unfolding vimage2p-def by auto

lemma assumes type-definition Rep Abs UNIV
  shows type-copy-Rep-o-Abs: Rep o Abs = id and type-copy-Abs-o-Rep: Abs o Rep = id
  unfolding fun-eq-iff comp-apply id-apply
  by simp-all

lemma type-copy-map-comp0-undo:
  assumes type-definition Rep Abs UNIV
    type-definition Rep′ Abs′ UNIV
    type-definition Rep″ Abs″ UNIV
  shows Abs′ o M o Rep″ = (Abs′ o M1 o Rep) o (Abs o M2 o Rep″) =⇒ M1 o M2 = M
  by (rule sym) (auto simp: fun-eq-iff type-definition Abs-inject[OF assms(2) UNIV-I UNIV-I]
      type-definition Abs-inverse[OF assms(1) UNIV-I]
      type-definition Abs-inverse[OF assms(3) UNIV-I] dest: spec[of - Abs″ x for x])

lemma vimage2p-id: vimage2p id id R = R
  unfolding vimage2p-def by auto

lemma vimage2p-comp: vimage2p (f1 o f2) (g1 o g2) = vimage2p f2 g2 o vimage2p f1 g1
  unfolding fun-eq-iff vimage2p-def o-apply by simp

lemma fun-cong-unused-0: f = (λx. g) =⇒ f (λx. 0) = g
  by (erule arg-cong)
lemma inj-on-convol-ident: inj-on (λx. (x, f x)) X
  unfolding inj-on-def by simp

lemma case-prod-app: case-prod f x y = case-prod (λl r. f l r y) x
  by (case-tac x) simp

lemma case-sum-map-sum: case-sum l r (map-sum f g x) = case-sum (l ∘ f) (r ∘ g) x
  by (case-tac x) simp+

lemma case-prod-map-prod: case-prod h (map-prod f g x) = case-prod (λl r. h (f l) (g r)) x
  by (case-tac x) simp+

lemma prod-inj-map: inj f ⇒ inj g ⇒ inj (map-prod f g)
  by (simp add: inj-on-def)

lemma eq-ifI: (P → t = u1) ⇒ (¬ P → t = u2) ⇒ t = (if P then u1 else u2)
  by simp

ML-file Tools/BNF/bnf-fp-util.ML
ML-file Tools/BNF/bnf-fp-def-sugar-tactics.ML
ML-file Tools/BNF/bnf-fp-size.ML
ML-file Tools/BNF/bnf-fp-def-sugar.ML
ML-file Tools/BNF/bnf-fp-n2m-tactics.ML
ML-file Tools/BNF/bnf-fp-n2m.ML
ML-file Tools/BNF/bnf-fp-n2m-sugar.ML

ML-file Tools/Function/size.ML
setup Size.setup

lemma size-bool[code]: size (b::bool) = 0
  by (cases b) auto

lemma size-nat[simp, code]: size (n::nat) = n
  by (induct n) simp-all

declare prod.size[no-atp]

lemma size-sum-o-map: size-sum g1 g2 o map-sum f1 f2 = size-sum (g1 o f1) (g2 o f2)
  by (rule ext) (case-tac x, auto)

lemma size-prod-o-map: size-prod g1 g2 o map-prod f1 f2 = size-prod (g1 o f1) (g2 o f2)
  by (rule ext) auto

setup ▌
38  BNF-LFP: Least Fixed Point Operation on Bounded Natural Functors

theory BNF-LFP
imports BNF-FP-Base
keywords
  datatype-new :: thy-decl and
  datatype-compat :: thy-decl
begin

lemma subset-emptyI: (\(\forall x. x \in A \implies False\)) \implies A \subseteq \{
  by blast

lemma image-Collect-subsetI: (\(\forall x. P x \implies f x \in B\)) \implies f \{x. P x\} \subseteq B
  by blast

lemma Collect-restrict: \(\{x. x \in X \land P x\}\) \subseteq X
  by auto

lemma prop-restrict: \([x \in Z; Z \subseteq \{x. x \in X \land P x\}] \implies P x\)
  by auto

lemma underS-I: \([i \neq j; (i, j) \in R]\) \implies i \in underS R j
  unfolding underS-def by simp

lemma underS-E: \(i \in underS R j \implies i \neq j \land (i, j) \in R\)
  unfolding underS-def by simp

lemma underS-Field: \(i \in underS R j \implies i \in Field R\)
  unfolding underS-def Field-def by auto

lemma FieldI2: \((i, j) \in R \implies j \in Field R\)
  unfolding Field-def by auto

lemma fst-convol': \(fst ((f, g) x) = f x\)
  using fst-convol unfolding convol-def by simp

lemma snd-convol': \(snd ((f, g) x) = g x\)
using snd-convol unfolding convol-def by simp

lemma convol-expand-snd: \( \text{fst} \circ f = g \implies \langle g, \text{snd} \circ f \rangle = f \)
unfolding convol-def by auto

lemma convol-expand-snd':
assumes \( \text{fst} \circ f = g \)
shows \( h = \text{snd} \circ f \iff \langle g, h \rangle = f \)
proof
  from assms have \( \ast \): \( \langle g, \text{snd} \circ f \rangle = f \) by (rule convol-expand-snd)
  then have \( h = \text{snd} \circ f \iff h = \text{snd} \circ \langle g, \text{snd} \circ f \rangle \) by simp
  moreover have \( \ldots \iff \langle g, h \rangle = f \) by (subst \( \ast \)[symmetric]) (auto simp: convol-def fun-eq-iff)
  ultimately show \( \ast \)thesis by simp
qed

lemma bij-betwE: bij-betw \( f \) \( A \) \( B \) \( \implies \forall a \in A. \ f a \in B \)
unfolding bij-betw-def by auto

lemma bij-betw-imageE: bij-betw \( f \) \( A \) \( B \) \( \implies \ f ' \ A \ = \ B \)
unfolding bij-betw-def by auto

lemma f-the-inv-into-f-bij-betw: bij-betw \( f \) \( A \) \( B \) \( \implies \)
  \( \forall x \in B. \ f (\text{the-inv-into } A f x) = x \)
unfolding bij-betw-def inj-on-def by blast

lemma ex-bij-betw: \( |A| \leq o \ (r :: 'b rel) \implies \exists B :: 'b set. \ bij-betw \( f \) \( B \) \( A \) \)
by (auto dest: iffD2[OF card-of-ordIso ordIso-symmetric])

lemma bij-betwI':
\( \forall x y. \ [x \in X; y \in X \implies (f x = f y) = (x = y)]; \)
\( \forall x. x \in X \implies f x \in Y; \)
\( \forall y. y \in Y \implies \exists x \in X. y = f x \) \( \iff \bij-betw f X Y \)
unfolding bij-betw-def inj-on-def by blast

lemma surj-fun-eq:
assumes surj-on: \( f \cdot X = \text{UNIV} \) and eq-on: \( \forall x \in X. \ (g1 \circ f) x = (g2 \circ f) x \)
signs \( g1 = g2 \)
proof (rule ext)
fix \( y \)
from surj-on obtain \( x \) where \( x \in X \) and \( y = f x \) by blast
thus \( g1 y = g2 y \) using eq-on by simp
qed

lemma Card-order-wo-rel: Card-order \( r \implies \text{wo-rel} \ r \)
unfolding wo-rel-def card-order-on-def by blast

lemma Cinfinite-limit: \( \forall x \in \text{Field } r; \text{Cinfinite } r \) \( \implies \)
\[ \exists y \in \text{Field } r. \ x \neq y \land (x, y) \in r \]

unfolding \( \text{cinfinite-def} \) by (\text{auto simp add: infinite-Card-order-limit})

lemma \( \text{Card-order-trans} \):
\[ \begin{align*}
\text{[Card-order r; x \neq y; (x, y) \in r; y \neq z; (y, z) \in r]} & \implies x \neq z \land (x, z) \in r
\end{align*} \]

unfolding \( \text{card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def trans-def antisym-def} \) by blast

lemma \( \text{Cinfinite-limit2} \):
\[ \text{assumes x1: } x1 \in \text{Field } r \text{ and x2: } x2 \in \text{Field } r \text{ and r: } \text{Cinfinite } r \]
shows \[ \exists y \in \text{Field } r. \ (x1 \neq y \land (x1, y) \in r) \land (x2 \neq y \land (x2, y) \in r) \]

proof –
from \( r \) have \( \text{trans: } \text{trans } r \text{ and total: } \text{Total } r \text{ and antisym: } \text{antisym } r \)

unfolding \( \text{card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def trans-def antisym-def} \) by auto

obtain \( y1 \) where \( y1 \in \text{Field } r \) \( x1 \neq y1 \) \( (x1, y1) \in r \)
using \( \text{Cinfinite-limit[OF x1 r]} \) by blast

obtain \( y2 \) where \( y2 \in \text{Field } r \) \( x2 \neq y2 \) \( (x2, y2) \in r \)
using \( \text{Cinfinite-limit[OF x2 r]} \) by blast

show \( \text{thesis} \)
proof (cases \( y1 = y2 \))
case \( \text{True} \) with \( y1 \) \( y2 \) show \( \text{thesis} \) by blast
next
  case \( \text{False} \)
  with \( \text{False } y1 \) \( y2 \)
  with \( \text{False } y1 \) \( y2 \) * antisym show \( \text{thesis} \) by (cases \( x1 = y2 \)) (\text{auto simp: antisym-def})
next
  assume \( *: (y1, y2) \in r \)
  with \( \text{trans } y1(3) \) have \( (x1, y2) \in r \)
  unfolding \( \text{trans-def} \) by blast
  with \( \text{False } y1 \) \( y2 \) * antisym show \( \text{thesis} \) by (cases \( x2 = y1 \)) (\text{auto simp: antisym-def})
qed

lemma \( \text{Cinfinite-limit-finite} \): \[ \text{[finite } X; X \subseteq \text{Field } r; \text{Cinfinite } r]\]
\[ \implies \exists \ y \in \text{Field } r. \ \forall \ x \in X. \ (x \neq y \land (x, y) \in r) \]

proof (induct \( X \) rule: finite-induct)
case empty thus \( \text{thesis} \)
  unfolding \( \text{cinfinite-def} \) using \( \text{ex-in-cone[of Field } r \]
finite.empty/ by auto
next
case (insert \( x \) \( X \))
then obtain \( y \) where \( y \in \text{Field } r \) \( \forall \ x \in X. \ (x \neq y \land (x, y) \in r) \)
by blast
then obtain $z$ where $z \in \text{Field } r \not= z \land (x, z) \in r \land y \not= z \land (y, z) \in r$

using $\text{Cinfinite-limit2[OF } y(1) \text{ insert}(5), \text{ of } x \text{ insert}(4) \text{ by blast}$

show $\text{?case}$

apply (intro bexI ballI)
apply (erule insertE)
apply hypsubst
apply (rule $z(2)$)
using $\text{Card-order-trans[OF insert}(5)[\text{THEN conjunct2]} y(2) z(3)$
apply blast
apply (rule $z(1)$)
done

qed

lemma $\text{insert-subsetI}$: $[x \in A; X \subseteq A] \implies \text{insert } x X \subseteq A$
by auto

lemma $\text{well-order-induct-imp}$:
\begin{align*}
\text{wo-rel } r & \implies \left( \forall x. \forall y. y \not= x \land (y, x) \in r \implies y \in \text{Field } r \implies P y \implies x \in \text{Field } r \implies P x \right) \\
& \implies x \in \text{Field } r \implies P x
\end{align*}
by (erule wo-rel.well-order-induct)

lemma $\text{meta-spec2}$:
assumes $\left( \forall x. \forall y. \text{PROP } P x y \right)$
shows $\text{PROP } P x y$
by (rule assms)

lemma $\text{nchotomy-relcomppE}$:
assumes $\left( \exists y. \exists x. y = f x (r \ O \ O s) a c \land b. r a (f b) \implies s (f b) c \implies P \right)$
shows $P$
proof (rule relcompp.cases[OF assms(2)], hypsubst)
fix $b$ assume $r \ a \ b \ s \ b \ c$
moreover from assms(1) obtain $b'$ where $b = f b'$ by blast
ultimately show $P$ by (blast intro: assms(3))
qed

lemma $\text{vimage2p-rel-fun}$: $\text{rel-fun} (\text{vimage2p } f \ g \ R) \ R \ f \ g$
unfolding $\text{rel-fun-def} \ vimage2p-def$ by auto

lemma $\text{predicate2D-vimage2p}$: $[R \leq \text{vimage2p } f \ g \ S; R x y] \implies S (f x) (g y)$
unfolding $\text{vimage2p-def}$ by auto

lemma $\text{id-transfer}$: $\text{rel-fun } A A \ id \ id$
unfolding $\text{rel-fun-def}$ by simp

lemma $\text{ssubst-Pair-rhs}$: $[(r, s) \in R; s' = s] \implies (r, s') \in R$
by (rule ssubst)
THEORY "Num"

ML-file Tools/BNF/bnf-lfp-utill.ML
ML-file Tools/BNF/bnf-lfp-tactics.ML
ML-file Tools/BNF/bnf-lfp.ML
ML-file Tools/BNF/bnf-lfp-compat.ML
ML-file Tools/BNF/bnf-lfp-rec-sugar-more.ML

hide-fact (open) id-transfer

end

39 Num: Binary Numerals

theory Num
imports Datatype BNF-LFP
begin

39.1 The num type

datatype num = One | Bit0 num | Bit1 num

Increment function for type num

primrec inc :: num ⇒ num where
  inc One = Bit0 One |
  inc (Bit0 x) = Bit1 x |
  inc (Bit1 x) = Bit0 (inc x)

Converting between type num and type nat

primrec nat-of-num :: num ⇒ nat where
  nat-of-num One = Suc 0 |
  nat-of-num (Bit0 x) = nat-of-num x + nat-of-num x |
  nat-of-num (Bit1 x) = Suc (nat-of-num x + nat-of-num x)

primrec num-of-nat :: nat ⇒ num where
  num-of-nat 0 = One |
  num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)

lemma nat-of-num-pos: 0 < nat-of-num n
  by (induct n) simp-all

lemma nat-of-num-neg-0: nat-of-num n ≠ 0
  by (induct n) simp-all

lemma nat-of-num-inc: nat-of-num (inc x) = Suc (nat-of-num x)
  by (induct x) simp-all

lemma num-of-nat-double:
  0 < n ⇒ nat-of-num (n + n) = Bit0 (nat-of-num n)
  by (induct n) simp-all
Type \textit{num} is isomorphic to the strictly positive natural numbers.

**lemma** nat-of-num-inverse: \textit{num-of-nat} \((\textit{nat-of-num} \, x) = x\)

\begin{itemize}
    \item by (induct \(x\)) \(\text{simp-all add: nat-of-num-double nat-of-num-pos}\)
\end{itemize}

**lemma** num-of-nat-inverse: \(0 < n \implies \text{nat-of-num} \,(\text{num-of-nat} \, n) = n\)

\begin{itemize}
    \item by (induct \(n\)) \(\text{simp-all add: nat-of-num-inc}\)
\end{itemize}

**lemma** num-eq-iff: \(x = y \iff \text{nat-of-num} \, x = \text{nat-of-num} \, y\)

\begin{itemize}
    \item apply safe
    \item apply (drule arg-cong \([\text{where} \, f = \text{num-of-nat}]\))
    \item apply (simp add: \text{nat-of-num-inverse})
    \item done
\end{itemize}

**lemma** num-induct [case-names One inc]:

\begin{itemize}
    \item fixes \(P : \textit{num} \Rightarrow \textit{bool}\)
    \item assumes One: \(P \, \text{One}\)
    \item and inc: \(\forall x. \, P \, x \implies P \,(\text{inc} \, x)\)
    \item shows \(P \, x\)
\end{itemize}

\begin{itemize}
    \item proof –
    \item obtain \(n\) where \(n: \text{Suc} \, n = \text{nat-of-num} \, x\)
    \item by (cases \text{nat-of-num} \, x, simp-all add: \text{nat-of-num-neq-0})
    \item have \(P \,(\text{num-of-nat} \,(\text{Suc} \, n))\)
    \item proof (induct \(n\))
    \item case 0 show \(\text{?case}\) using One by simp
    \item next
    \item case (Suc \(n\))
    \item then have \(P \,(\text{inc} \,(\text{num-of-nat} \,(\text{Suc} \, n)))\) by (rule inc)
    \item then show \(P \,(\text{num-of-nat} \,(\text{Suc} \,(\text{Suc} \, n)))\) by simp
    \item qed
    \item with \(n\) show \(P \, x\)
    \item by (simp add: \text{nat-of-num-inverse})
    \item qed
\end{itemize}

From now on, there are two possible models for \textit{num}: as positive naturals (rule \textit{num-induct}) and as digit representation (rules \textit{num.induct}, \textit{num.cases}).

### 39.2 Numeral operations

**instantiation** \textit{num} :: \{\textit{plus},\textit{times},\textit{linorder}\}

**begin**

**definition** [code del]:
\(m + n = \text{num-of-nat} \,(\text{nat-of-num} \, m + \text{nat-of-num} \, n)\)

**definition** [code del]:
\(m * n = \text{num-of-nat} \,(\text{nat-of-num} \, m * \text{nat-of-num} \, n)\)

**definition** [code del]:
\(m \leq n \iff \text{nat-of-num} \, m \leq \text{nat-of-num} \, n\)
THEORY "Num"
by simp-all

lemma le-num-simps [simp, code]:
One ≤ n ↔ True
Bit0 m ≤ One ↔ False
Bit1 m ≤ One ↔ False
Bit0 m ≤ Bit0 n ↔ m ≤ n
Bit0 m ≤ Bit1 n ↔ m ≤ n
Bit1 m ≤ Bit1 n ↔ m ≤ n
Bit1 m ≤ Bit0 n ↔ m < n
using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

lemma less-num-simps [simp, code]:
m < One ↔ False
One < Bit0 n ↔ True
One < Bit1 n ↔ True
Bit0 m < Bit0 n ↔ m < n
Bit0 m < Bit1 n ↔ m ≤ n
Bit1 m < Bit1 n ↔ m < n
Bit1 m < Bit0 n ↔ m < n
using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

Rules using One and inc as constructors
lemma add-One: x + One = inc x
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma add-One-commute: One + n = n + One
by (induct n) simp-all

lemma add-inc: x + inc y = inc (x + y)
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma mult-inc: x * inc y = x * y + x

The num-of-nat conversion
lemma num-of-nat-One:
n ≤ 1 ⇒ num-of-nat n = One
by (cases n) simp-all

lemma num-of-nat-plus-distrib:
0 < m ⇒ 0 < n ⇒ num-of-nat (m + n) = num-of-nat m + num-of-nat n
by (induct n) (auto simp add: add-One add-One-commute add-inc)

A double-and-decrement function
primrec BitM :: num ⇒ num where
BitM One = One |
BitM (Bit0 n) = Bit1 (BitM n) |
BitM (Bit1 n) = Bit1 (Bit0 n)

lemma BitM-plus-one: BitM n + One = Bit0 n 
by (induct n) simp-all

lemma one-plus-BitM: One + BitM n = Bit0 n 
unfolding add-One-commute BitM-plus-one ..

Squaring and exponentiation

primrec sqr :: num ⇒ num where
sqr One = One |
sqr (Bit0 n) = Bit0 (Bit0 (sqr n)) |
sqr (Bit1 n) = Bit1 (Bit0 (sqr n + n))

primrec pow :: num ⇒ num ⇒ num where 
pow x One = x |
pow x (Bit0 y) = sqr (pow x y) |
pow x (Bit1 y) = sqr (pow x y) * x

lemma nat-of-num-sqr: nat-of-num (sqr x) = nat-of-num x * nat-of-num x 
by (induct x, simp-all add: algebra-simps nat-of-num-add)

lemma sqr-conv-mult: sqr x = x * x 
by (simp add: num-eq-iff nat-of-num-sqr nat-of-num-mult)

39.3 Binary numerals

We embed binary representations into a generic algebraic structure using numeral.

class numeral = one + semigroup-add
begin

primrec numeral :: num ⇒ 'a where
numeral-One: numeral One = 1 |
numeral-Bit0: numeral (Bit0 n) = numeral n + numeral n |
numeral-Bit1: numeral (Bit1 n) = numeral n + numeral n + 1

lemma numeral-code [code]:
numeral One = 1
numeral (Bit0 n) = (let m = numeral n in m + m)
numeral (Bit1 n) = (let m = numeral n in m + m + 1) 
by (simp-all add: Let-def)

lemma one-plus-numeral-commute: 1 + numeral x = numeral x + 1 
apply (induct x)
apply simp
apply (simp add: add.assoc [symmetric], simp add: add.assoc)
apply (simp add: add.assoc [symmetric], simp add: add.assoc)
done

lemma numeral-inc: numeral (inc x) = numeral x + 1
proof (induct x)
case (Bit1 x)
  have numeral x + (1 + numeral x) + 1 = numeral x + (numeral x + 1) + 1
    by (simp only: one-plus-numeral-commute)
  with Bit1 show ?case
    by (simp add: add.assoc)
qed simp-all

declare numeral.simps [simp del]

abbreviation Numeral1 ≡ numeral One
declare numeral-One [code-post]
end

Numeral syntax.
syntax -Numeral :: num-const ⇒ 'a (-)
parse-translation ⟨⟨
  let
  fun num-of-int n =
    if n > 0 then
      (case IntInf.quotRem (n, 2) of
       (0, 1) => Syntax.const @{const-syntax One}
       | (n, 0) => Syntax.const @{const-syntax Bit0} $ num-of-int n
       | (n, 1) => Syntax.const @{const-syntax Bit1} $ num-of-int n)
    else raise Match
  val numeral = Syntax.const @{const-syntax numeral}
  val uminus = Syntax.const @{const-syntax uminus}
  val one = Syntax.const @{const-syntax Groups.one}
  val zero = Syntax.const @{const-syntax Groups.zero}
  fun numeral-tr [(c as Const (@{syntax-const -constrain}, -)) $ t $ u] =
    c $ numeral-tr [t] $ u
  | numeral-tr [Const (num, -)] =
    let
      val {value, ...} = Lexicon.read-xnum num;
      in
      if value = 0 then zero else
      if value > 0
        then numeral $ num-of-int value
        else if value = ~1 then uminus $ one
        else uminus $ (numeral $ num-of-int (~ value))
    end
  end

numeral-tr ts = raise TERM (numeral-tr, ts);

in 

[(@{syntax-const -Numeral}, K numeral-tr)] end

⟩⟩

typed-print-translation ⟨⟨

let 

fun dest-num (Const (@{const-syntax Bit0}, _) $ n) = 2 * dest-num n 

| dest-num (Const (@{const-syntax Bit1}, _) $ n) = 2 * dest-num n + 1 

| dest-num (Const (@{const-syntax One}, _)) = 1;

fun num-tr’ ctxt T [n] = 

let 

val k = dest-num n;

val t’ = 

Syntax.const @{syntax-const -Numeral} $ Syntax.free (string-of-int k);

in 

(case T of 

Type (@{type-name fun}, [., T’]) =>

if Printer.type-emphasis ctxt T’ then

Syntax.const @{syntax-const -constrain} $ t’ $

Syntax-Phases.term-of-typ ctxt T’

else t’

| - => if T = dummyT then t’ else raise Match)

end;

in 

[(@{const-syntax numeral}, num-tr’)]

end

⟩⟩

ML-file Tools/numeral.ML

39.4 Class-specific numeral rules

numeral is a morphism.

39.4.1 Structures with addition: class numeral

context numeral

begin

lemma numeral-add: numeral (m + n) = numeral m + numeral n
by (induct n rule: num-induct)

(simp-all only: numeral-One add-One add-inc numeral-inc add.assoc)

lemma numeral-plus-numeral: numeral m + numeral n = numeral (m + n)
by (rule numeral-add [symmetric])

lemma numeral-plus-one: numeral n + 1 = numeral (n + One)
using numeral-add [of n One] by (simp add: numeral-One)
lemma one-plus-numeral: $1 + \text{numeral } n = \text{numeral } (\text{One } + n)$
using numeral-add [of One n] by (simp add: numeral-One)

lemma one-add-one: $1 + 1 = 2$
using numeral-add [of One One] by (simp add: numeral-One)

lemmas add-numeral-special =
numeral-plus-one one-plus-numeral one-add-one
end

39.4.2 Structures with negation: class neg-numeral

class neg-numeral = numeral + group-add
begin

lemma uminus-numeral-One:
$-\text{Numeral1} = -1$
by (simp add: numeral-One)

Numerals form an abelian subgroup.

inductive is-num :: 'a ⇒ bool where
is-num 1 | is-num x ⇒ is-num ($- x$) |
[is-num x; is-num y] ⇒ is-num (x + y)

lemma is-num-numeral: is-num (numeral k)
by (induct k, simp-all add: numeral.simps is-num.intros)

lemma is-num-add-commute:
[is-num x; is-num y] ⇒ x + y = y + x
apply (induct x rule: is-num.induct)
apply (induct y rule: is-num.induct)
apply simp
apply (rule-tac a=x in add-left-imp-eq)
apply (rule-tac a=x in add-right-imp-eq)
apply (simp add: add.assoc)
apply (simp add: add.assoc [symmetric], simp add: add.assoc)
apply (rule-tac a=x in add-left-imp-eq)
apply (rule-tac a=x in add-right-imp-eq)
apply (simp add: add.assoc)
apply (simp add: add.assoc, simp add: add.assoc [symmetric])
done

lemma is-num-add-left-commute:
[is-num x; is-num y] ⇒ x + (y + z) = y + (x + z)
by (simp only: add.assoc [symmetric] is-num-add-commute)
lemmas is-num-normalize =
  add.assoc is-num-add-commute is-num-add-left-commute
  is-num.intros is-num-numeral
  minus-add

definition dbl :: 'a ⇒ 'a where
dbl x = x + x
definition dbl-inc :: 'a ⇒ 'a where
dbl-inc x = x + x + 1
definition dbl-dec :: 'a ⇒ 'a where
dbl-dec x = x + x − 1

definition sub :: num ⇒ num ⇒ 'a where
sub k l = numeral k − numeral l

lemma numeral-BitM: numeral (BitM n) = numeral (Bit0 n) − 1
  by (simp only: BitM-plus-one [symmetric] numeral-add numeral-One eq-diff-eq)

lemma dbl-simps [simp]:
dbl (− numeral k) = − dbl (numeral k)
dbl 0 = 0
dbl 1 = 2
dbl (− 1) = − 2
dbl (numeral k) = numeral (Bit0 k)
  by (simp-all add: dbl-def numeral.simps minus-add)

lemma dbl-inc-simps [simp]:
dbl-inc (− numeral k) = − dbl-dec (numeral k)
dbl-inc 0 = 1
dbl-inc 1 = 3
dbl-inc (− 1) = − 1
dbl-inc (numeral k) = numeral (Bit1 k)
  by (simp-all add: dbl-inc-def dbl-dec-def numeral.simps numeral-BitM is-num-normalize
       algebra-simps del: add-uminus-conv-diff)

lemma dbl-dec-simps [simp]:
dbl-dec (− numeral k) = − dbl-inc (numeral k)
dbl-dec 0 = − 1
dbl-dec 1 = 1
dbl-dec (− 1) = − 3
dbl-dec (numeral k) = numeral (BitM k)
  by (simp-all add: dbl-dec-def dbl-inc-def numeral.simps numeral-BitM is-num-normalize)

lemma sub-num-simps [simp]:
sub One One = 0
sub One (Bit0 l) = − numeral (BitM l)
sub One (Bit1 l) = − numeral (Bit0 l)
sub (Bit0 k) One = numeral (BitM k)
sub (Bit1 k) One = numeral (Bit0 k)
sub (Bit0 k) (Bit0 l) = dbl (sub k l)
sub (Bit0 k) (Bit1 l) = dbl-dec (sub k l)
sub (Bit1 k) (Bit0 l) = dbl-inc (sub k l)
theory "Num"

sub (Bit1 k) (Bit1 l) = dbl (sub k l)
by (simp-all add: dbl-def dbl-dec-def dbl-inc-def sub-def numeral.simps
  numeral-BitM is-num-normalize del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma add-neg-numeral-simps:
numeral m + - numeral n = sub m n
- numeral m + numeral n = sub n m
- numeral m + - numeral n = - (numeral m + numeral n)
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
  del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma add-neg-numeral-special:
1 + - numeral m = sub One m
- numeral m + 1 = sub One m
numeral m + - 1 = sub m One
- 1 + numeral n = sub n One
- 1 + - numeral n = - numeral (inc n)
- numeral m + - 1 = - numeral (inc m)
1 + - 1 = 0
- 1 + 1 = 0
- 1 + - 1 = - 2
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize right-minus
  numeral-inc
  del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma diff-numeral-simps:
numeral m - numeral n = sub m n
numeral m - - numeral n = numeral (m + n)
- numeral m - numeral n = - numeral (m + n)
- numeral m - - numeral n = sub n m
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
  del: add-uminus-conv-diff add: diff-conv-add-uminus)

lemma diff-numeral-special:
1 - numeral n = sub One n
numeral m - 1 = sub m One
1 - - numeral n = numeral (One + n)
- numeral m - 1 = - numeral (m + One)
- 1 - numeral n = - numeral (inc n)
numeral m - - 1 = numeral (inc m)
- 1 - - numeral n = sub n One
- numeral m - - 1 = sub One m
1 - 1 = 0
- 1 - 1 = - 2
1 - - 1 = 2
- 1 - - 1 = 0
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize numeral-inc
  del: add-uminus-conv-diff add: diff-conv-add-uminus)
39.4.3 Structures with multiplication: class \textit{semiring-numeral}

\begin{verbatim}
begin

subclass numeral ..

lemma numeral-mult: numeral \((m * n) = numeral m * numeral n\)
  apply (induct n rule: num-induct)
  apply (simp add: numeral-One)
  apply (simp add: mult-inc numeral-inc numeral-add distrib-left)
  done

lemma numeral-times-numeral: numeral \(m * numeral n = numeral \(m * n\))
  by (rule numeral-mult [symmetric])

lemma mult-2: \(2 * z = z + z\)
  unfolding one-add-one [symmetric] distrib-right by simp

lemma mult-2-right: \(z * 2 = z + z\)
  unfolding one-add-one [symmetric] distrib-left by simp

end

\end{verbatim}

39.4.4 Structures with a zero: class \textit{semiring-1}

\begin{verbatim}
context semiring-1
begin

subclass semiring-numeral ..

lemma of-nat-numeral [simp]: of-nat \(\text{numeral } n\) = numeral \(n\)
  by (induct n,
       simp-all only: numeral.simps numeral-class.numeral.simps of-nat-add of-nat-1)

end

lemma nat-of-num-numeral [code-abbrev]:
  nat-of-num = numeral

proof
  fix \(n\)
  have numeral n = nat-of-num n
    by (induct n) (simp-all add: numeral.simps)
  then show nat-of-num n = numeral n by simp
qed

lemma nat-of-num-code [code]:
  nat-of-num One = 1
\end{verbatim}
nat-of-num (Bit0 n) = (let m = nat-of-num n in m + m)
nat-of-num (Bit1 n) = (let m = nat-of-num n in Suc (m + m))
by (simp-all add: Let-def)

39.4.5 Equality: class semiring-char-0
context semiring-char-0
begin

lemma numeral-eq-iff: numeral m = numeral n ⌸→ m = n
unfolding of-nat-numeral [symmetric] nat-of-num-numeral [symmetric]
  of-nat-eq-iff num-eq-iff ..

lemma numeral-eq-one-iff: numeral n = 1 ⌸→ n = One
by (rule numeral-eq-iff [of n One, unfolded numeral-One])

lemma one-eq-numeral-iff: One = numeral n ⌸→ One = n
by (rule numeral-eq-iff [of One n, unfolded numeral-One])

lemma numeral-neq-zero: numeral n ≠ 0
unfolding of-nat-numeral [symmetric] nat-of-num-numeral [symmetric]
by (simp add: nat-of-num-pos)

lemma zero-neq-numeral: 0 ≠ numeral n
unfolding eq-commute [of 0] by (rule numeral-neq-zero)

lemmas eq-numeral-simps [simp] =
  numeral-eq-iff
  numeral-eq-one-iff
  one-eq-numeral-iff
  numeral-neq-zero
  zero-neq-numeral

end

39.4.6 Comparisons: class linordered-semidom

Could be perhaps more general than here.

context linordered-semidom
begin

lemma numeral-le-iff: numeral m ≤ numeral n ⌸→ m ≤ n
proof –
  have of-nat (numeral m) ≤ of-nat (numeral n) ⌸→ m ≤ n
    unfolding less-eq-num-def nat-of-num-numeral of-nat-le-iff ..
  then show ?thesis by simp
qed

lemma one-le-numeral: 1 ≤ numeral n
using numeral-le-iff [of One n] by (simp add: numeral-One)

lemma numeral-le-one-iff: numeral n ≤ 1 ↔ n ≤ One
using numeral-le-iff [of n One] by (simp add: numeral-One)

lemma numeral-less-iff: numeral m < numeral n ↔ m < n
proof
  have of-nat (numeral m) < of-nat (numeral n) ↔ m < n
  unfolding less-num-def nat-of-num-numeral of-nat-less-iff ..
  then show ?thesis by simp
qed

lemma not-numeral-less-one: ¬ numeral n < 1
using numeral-less-iff [of n One] by (simp add: numeral-One)

lemma one-less-numeral-iff: 1 < numeral n ↔ One < n
using numeral-less-iff [of One n] by (simp add: numeral-One)

lemma zero-le-numeral: 0 ≤ numeral n
by (induct n) (simp-all add: numeral.simps)

lemma zero-less-numeral: 0 < numeral n
by (induct n) (simp-all add: numeral.simps add-pos-pos)

lemma not-numeral-le-zero: ¬ numeral n ≤ 0
by (simp add: not-le zero-less-numeral)

lemma not-numeral-less-zero: ¬ numeral n < 0
by (simp add: not-less zero-le-numeral)

lemmas le-numeral-extra =
  zero-le-one not-one-le-zero
  order-refl [of 0] order-refl [of 1]

lemmas less-numeral-extra =
  zero-less-one not-one-less-zero
  less-irrefl [of 0] less-irrefl [of 1]

lemmas le-numeral-simps [simp] =
  numeral-le-iff
  one-le-numeral
  numeral-le-one-iff
  zero-le-numeral
  not-numeral-le-zero

lemmas less-numeral-simps [simp] =
  numeral-less-iff
  one-less-numeral-iff
  not-numeral-less-one
THEORY "Num"

zero-less-numeral
not-numeral-less-zero

end

39.4.7 Multiplication and negation: class ring-1
context ring-1
begin
subclass neg-numeral ..

lemma mult-neg-numeral-simps:
− numeral m * − numeral n = numeral (m * n)
− numeral m * numeral n = − numeral (m * n)
numeral m * − numeral n = − numeral (m * n)
unfolding mult-minus-left mult-minus-right
by (simp-all only: minus-minus numeral-mult)

lemma mult-minus1 [simp]: − 1 * z = − z
unfolding numeral.simps mult-minus-left by simp

lemma mult-minus1-right [simp]: z * − 1 = − z
unfolding numeral.simps mult-minus-right by simp

end

39.4.8 Equality using iszero for rings with non-zero characteristic
context ring-1
begin

definition iszero :: 'a ⇒ bool
where iszero z ←→ z = 0

lemma iszero-0 [simp]: iszero 0
by (simp add: iszero-def)

lemma not-iszero-1 [simp]: ¬ iszero 1
by (simp add: iszero-def)

lemma not-iszero-Numeral1: ¬ iszero Numeral1
by (simp add: numeral-One)

lemma not-iszero-neg-1 [simp]: ¬ iszero (− 1)
by (simp add: iszero-def)

lemma not-iszero-neg-Numeral1: ¬ iszero (− Numeral1)
by (simp add: numeral-One)
lemma iszero-neg-numeral [simp]:
iszero (− numeral w) ←→ iszero (numeral w)
unfolding iszero-def by (rule neg-equal-0-iff-equal)

lemma eq-iff-iszero-diff: x = y ←→ iszero (x − y)
unfolding iszero-def by (rule eq-iff-diff-eq-0)

The eq-numeral-iff-iszero lemmas are not declared [simp] by default, because for rings of characteristic zero, better simp rules are possible. For a type like integers mod n, type-instantiated versions of these rules should be added to the simplifier, along with a type-specific rule for deciding propositions of the form iszero (numeral w).
bh: Maybe it would not be so bad to just declare these as simp rules anyway? I should test whether these rules take precedence over the ring-char-0 rules in the simplifier.

lemma eq-numeral-iff-iszero:
numeral x = numeral y ←→ iszero (sub x y)
numeral x = − numeral y ←→ iszero (numeral (x + y))
numeral x = − numeral y ←→ iszero (numeral (x + y))
numeral x = − numeral y ←→ iszero (sub y x)
numeral x = 1 ←→ iszero (sub x One)
1 = numeral y ←→ iszero (sub One y)
numeral x = 1 ←→ iszero (numeral (x + One))
1 = − numeral y ←→ iszero (numeral (One + y))
numeral x = 0 ←→ iszero (numeral x)
0 = numeral y ←→ iszero (numeral y)
numeral x = 0 ←→ iszero (numeral x)
0 = − numeral y ←→ iszero (numeral y)
unfolding eq-iff-iszero-diff diff-numeral-simps diff-numeral-special by simp-all

end

39.4.9 Equality and negation: class ring-char-0

class ring-char-0 = ring-1 + semiring-char-0

begin

lemma not-iszero-numeral [simp]: ¬ iszero (numeral w)
by (simp add: iszero-def)

lemma neg-numeral-eq-iff: − numeral m = − numeral n ←→ m = n
by simp

lemma numeral-neg-neg-numeral: numeral m ≠ − numeral n
unfolding eq-iff-add-eq-0
by (simp add: numeral-plus-numeral)
lemma neg-numeral-neq-numeral: \(-\text{numeral } m \neq \text{numeral } n\)
by (rule numeral-neq-neg-numeral [symmetric])

lemma zero-neq-numeral: \(0 \neq \text{numeral } n\)
unfolding neg-0-equal-iff-equal by simp

lemma neg-numeral-neq-zero: \(-\text{numeral } n \neq 0\)
unfolding neg-equal-0-iff-equal by simp

lemma one-neq-neg-numeral: \(1 \neq -\text{numeral } n\)
using numeral-neq-neg-numeral [of One n] by (simp add: numeral-One)

lemma neg-numeral-neq-one: \(-\text{numeral } n \neq 1\)
using neg-numeral-neq-numeral [of n One] by (simp add: numeral-One)

lemma neg-one-neq-numeral:
\(-1 \neq \text{numeral } n\)
using neg-numeral-neq-numeral [of One n] by (simp add: numeral-One)

lemma numeral-neq-neg-one:
\(\text{numeral } n \neq -1\)
using numeral-neq-neg-numeral [of n One] by (simp add: numeral-One)

lemma neg-one-eq-numeral-iff:
\(-1 = -\text{numeral } n \iff n = \text{One}\)
using neg-numeral-eq-iff [of One n] by (auto simp add: numeral-One)

lemma numeral-eq-neg-one-iff:
\(\text{numeral } n = -1 \iff n = \text{One}\)
using neg-numeral-eq-iff [of n One] by (auto simp add: numeral-One)

lemma neg-one-neq-zero:
\(-1 \neq 0\)
by simp

lemma zero-neq-neg-one:
\(0 \neq -1\)
by simp

lemma neg-one-neq-one:
\(-1 \neq 1\)
using neg-numeral-neq-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemma one-neq-neg-one:
\(1 \neq -1\)
using numeral-neq-neg-numeral [of One One] by (simp only: numeral-One not-False-eq-True)

lemmas eq-neg-numeral-simps [simp] =
THEORY “Num”

neg-numeral-eq-iff
numeral-neg-numeral eq-numeral
one-neg-numeral neg-numeral eq-one
zero-neg-numeral neg-numeral eq-zero
neg-one-neg-numeral numeral-neg-numeral
neg-one-neg-numeral eq-numeral-neg-one
neg-one-neg-numeral eq-neg-one
neg-one-neg-numeral eq-numeral-neg-one
neg-one-neg-numeral eq-one-neg-one

end

39.4.10 Structures with negation and order: class linordered-idom
class linordered-idom

begin

subclass ring-char-0 ..

lemma neg-numeral-le-iff: numeral m ≤ numeral n iff n ≤ m
by (simp only: neg-le-iff numeral-le-iff)

lemma neg-numeral-less-iff: numeral m < numeral n iff n < m
by (simp only: neg-less-iff numeral-less-iff)

lemma neg-numeral-less-zero: numeral n < 0
by (simp only: neg-less-zero less-numeral-le-zero)

lemma not-zero-less-numeral: 0 < numeral n
by (simp only: not-less less-zero)

lemma not-numeral-less-neg-numeral: ¬ numeral m < numeral n
by (simp only: not-less)

lemma neg-numeral-less-numeral: numeral m < numeral n
using neg-numeral-less-zero less-numeral-le-zero

lemma neg-numeral-le-numeral: numeral m ≤ numeral n
by (simp only: less-le numeral-le-numeral)

lemma not-numeral-less-numeral: ¬ numeral m < numeral n
by (simp only: not-less)

lemma not-numeral-le-numeral: ¬ numeral m ≤ numeral n
by (simp only: not-le)

lemma neg-numeral-less-one: numeral m < 1
by (rule neg-numeral-less-numeral [of m One, unfolded numeral-One])

lemma neg-numeral-le-one: − numeral m ≤ 1
by (rule neg-numeral-le-numeral [of m One, unfolded numeral-One])

lemma not-one-less-neg-numeral: ¬ 1 < − numeral m
by (simp only: not-less neg-numeral-le-one)

lemma not-one-le-neg-numeral: ¬ 1 ≤ − numeral m
by (simp only: not-le neg-numeral-less-one)

lemma not-numeral-less-neg-one:

by (cases m simp-all)

lemma not-numeral-le-neg-one:

by (cases m simp-all)

lemma neg-one-less-numeral: − 1 < numeral m
using neg-numeral-less-numeral [of One m] by (simp add: numeral-One)

lemma neg-one-le-numeral: − 1 ≤ numeral m
using neg-numeral-le-numeral [of One m] by (simp add: numeral-One)

lemma neg-numeral-less-neg-one-iff: − numeral m < − 1 ↔ m ≠ One
by (cases m) simp-all

lemma sub-non-negative:

by (simp only: sub-def le-diff-eq) simp

lemma sub-positive:

by (simp only: sub-def less-diff-eq) simp

lemma sub-non-positive:

by (simp only: sub-def diff-le-eq) simp

lemma sub-negative:

by (simp only: sub-def neg-le-eq) simp
by (simp only: sub-def diff-less-eq) simp

lemmas le-neg-numeral-simps [simp] =
  neg-numeral-le-iff
  neg-numeral-le-numeral not-numeral-le-neg-numeral
  neg-numeral-le-zero not-zero-le-neg-numeral
  neg-numeral-le-one not-one-le-neg-numeral
  neg-one-le-numeral not-numeral-le-neg-one
  neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma le-minus-one-simps [simp]:
  \(-1 \leq 0\)
  \(-1 \leq 1\)
  \(\neg 0 \leq -1\)
  \(\neg 1 \leq -1\)
  by simp-all

lemmas less-neg-numeral-simps [simp] =
  neg-numeral-less-iff
  neg-numeral-less-numeral not-numeral-less-neg-numeral
  neg-numeral-less-zero not-zero-less-neg-numeral
  neg-numeral-less-one not-one-less-neg-numeral
  neg-one-less-numeral not-numeral-less-neg-one
  neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma less-minus-one-simps [simp]:
  \(-1 < 0\)
  \(-1 < 1\)
  \(\neg 0 < -1\)
  \(\neg 1 < -1\)
  by (simp-all add: less-le)

lemma abs-numeral [simp]: abs (numeral n) = numeral n
  by simp

lemma abs-neg-numeral [simp]: abs (\(-\) numeral n) = numeral n
  by (simp only: abs-minus-cancel abs-numeral)

lemma abs-neg-one [simp]:
  abs (\(-1\)) = 1
  by simp

end

39.4.11 Natural numbers

lemma Suc-1 [simp]: Suc 1 = 2
  unfolding Suc-eq-plus1 by (rule one-add-one)
lemma Suc-numeral [simp]: Suc (numeral n) = numeral (n + One)
unfolding Suc-eq-plus1 by (rule numeral-plus-one)

definition pred-numeral :: num ⇒ nat
  where [code del]: pred-numeral k = numeral k - 1

lemma numeral-eq-Suc: numeral k = Suc (pred-numeral k)
unfolding pred-numeral-def by simp

lemma eval-nat-numeral:
  numeral One = Suc 0
  numeral (Bit0 n) = Suc (numeral (BitM n))
  numeral (Bit1 n) = Suc (numeral (Bit0 n))
by (simp-all add: numeral.simps BitM-plus-one)

lemma pred-numeral-simps [simp]:
  pred-numeral One = 0
  pred-numeral (Bit0 k) = numeral (BitM k)
  pred-numeral (Bit1 k) = numeral (Bit0 k)
unfolding pred-numeral-def eval-nat-numeral
by (simp-all only: diff-Suc-Suc diff-0)

lemma numeral-2-eq-2: 2 = Suc (Suc 0)
by (simp add: eval-nat-numeral)

lemma numeral-3-eq-3: 3 = Suc (Suc (Suc 0))
by (simp add: eval-nat-numeral)

lemma numeral-1-eq-Suc-0: Numeral1 = Suc 0
by (simp only: numeral-One One-nat-def)

lemma Suc-nat-number-of-add:
  Suc (numeral v + n) = numeral (v + One) + n
by simp

lemmas numerals = numeral-One [where 'a=nat] numeral-2-eq-2

Comparisons involving Suc.

lemma eq-numeral-Suc [simp]: numeral k = Suc n ↔ pred-numeral k = n
by (simp add: numeral-eq-Suc)

lemma Suc-eq-numeral [simp]: Suc n = numeral k ↔ n = pred-numeral k
by (simp add: numeral-eq-Suc)

lemma less-numeral-Suc [simp]: numeral k < Suc n ↔ pred-numeral k < n
by (simp add: numeral-eq-Suc)

lemma less-Suc-numeral [simp]: Suc n < numeral k ↔ n < pred-numeral k
by (simp add: numeral-eq-Suc)

lemma le-numeral-Suc [simp]: numeral k ≤ Suc n ↔ pred-numeral k ≤ n
by (simp add: numeral-eq-Suc)

lemma le-Suc-numeral [simp]: Suc n ≤ numeral k ↔ n ≤ pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-Suc-numeral [simp]: Suc n - numeral k = n - pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-numeral-Suc [simp]: numeral k - Suc n = pred-numeral k - n
by (simp add: numeral-eq-Suc)

lemma max-Suc-numeral [simp]:
max (Suc n) (numeral k) = Suc (max n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma max-numeral-Suc [simp]:
max (numeral k) (Suc n) = Suc (max (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

lemma min-Suc-numeral [simp]:
min (Suc n) (numeral k) = Suc (min n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma min-numeral-Suc [simp]:
min (numeral k) (Suc n) = Suc (min (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

For case-nat and rec-nat.

lemma case-nat-numeral [simp]:
case-nat a f (numeral v) = (let pv = pred-numeral v in f pv)
by (simp add: numeral-eq-Suc)

lemma case-nat-add-eq-if [simp]:
case-nat a f ((numeral v) + n) = (let pv = pred-numeral v in f (pv + n))
by (simp add: numeral-eq-Suc)

lemma rec-nat-numeral [simp]:
rec-nat a f (numeral v) =
(let pv = pred-numeral v in f pv (rec-nat a f pv))
by (simp add: numeral-eq-Suc Let-def)

lemma rec-nat-add-eq-if [simp]:
rec-nat a f (numeral v + n) =
(let pv = pred-numeral v in f (pv + n) (rec-nat a f (pv + n)))
by (simp add: numeral-eq-Suc Let-def)

Case analysis on n < (2::'a)
lemma less-2-cases: \( n < 2 \implies n = 0 \lor n = \text{Suc} \, 0 \)
  by (auto simp add: numeral-2_eq_2)

Removal of Small Numerals: 0, 1 and (in additive positions) 2

bh: Are these rules really a good idea?

lemma add-2-eq-Suc [simp]: \( 2 + n = \text{Suc} \, (\text{Suc} \, n) \)
  by simp

lemma add-2-eq-Suc' [simp]: \( n + 2 = \text{Suc} \, (\text{Suc} \, n) \)
  by simp

Can be used to eliminate long strings of Sucs, but not by default.

lemma Suc3-eq-add-3: \( \text{Suc} \, (\text{Suc} \, (\text{Suc} \, n)) = 3 + n \)
  by simp

lemmas nat-1-add-1 = one-add-one [where \( 'a=\text{nat} \)]

### 39.5 Numeral equations as default simplification rules

declare (in numeral) numeral-One [simp]
declare (in numeral) numeral-plus-numeral [simp]
declare (in numeral) add-numeral-special [simp]
declare (in neg-numeral) add-neg-numeral-simps [simp]
declare (in neg-numeral) add-neg-numeral-special [simp]
declare (in neg-numeral) diff-numeral-simps [simp]
declare (in neg-numeral) diff-numeral-special [simp]
declare (in semiring-numeral) numeral-times-numeral [simp]
declare (in ring-1) mult-neg-numeral-simps [simp]

### 39.6 Setting up simprocs

lemma mult-numeral-1: Numeral1 * \( a \) = \( (a :: 'a::semiring-numeral) \)
  by simp

lemma mult-numeral-1-right: \( a \) * Numeral1 = \( (a :: 'a::semiring-numeral) \)
  by simp

lemma divide-numeral-1: \( a \) / Numeral1 = \( (a :: 'a::field) \)
  by simp

lemma inverse-numeral-1:
  inverse Numeral1 = \( \text{Numeral1} :: 'a::division-ring \)
  by simp

Theorem lists for the cancellation simprocs. The use of a binary numeral
for 1 reduces the number of special cases.

lemma mult-1s:
  fixes \( a :: 'a::semiring-numeral \)
and $b :: 'b::ring_1$

shows Numeral1 * $a = a$

$a * \text{Numeral1} = a$

$- \text{Numeral1} * b = - b$

$b * - \text{Numeral1} = - b$

by simp-all

setup $\langle\langle$

Reorient-Proc.add

(fn Const (@{const-name numeral}, -) $\Rightarrow$ true

|$ Const (@{const-name uminus}, -) \Rightarrow (Const (@{const-name numeral}, -) \Rightarrow false)$

$\rangle\rangle$

simproc-setup reorient-numeral

(numeral $w = x | - \text{numeral } w = y) = \text{Reorient-Proc.proc}$

39.6.1 Simplification of arithmetic operations on integer constants.

lemmas arith-special =

add-numeral-special add-neg-numeral-special
diff-numeral-special

lemmas arith-extra-simps =

numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right
minus-zero
diff-numeral-simps diff-0 diff-0-right
numeral-times-numeral mult-neg-numeral-simps
mult-zero-left mult-zero-right
abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules. Also include simp-thms.

lemmas arith-simps =

add-num-simps mult-num-simps sub-num-simps
BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps
abs-zero abs-one arith-extra-simps

lemmas more-arith-simps =

neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas of-nat-simps =
THEORY "Num"

of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations

lemmas eq-numeral-extra =
  zero-neq-one one-neq-zero

lemmas rel-simps =
  le-num-simps less-num-simps eq-num-simps
  le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
  less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
  eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

lemma Let-numeral [simp]: Let (numeral v) f = f (numeral v)
  — Unfold all lets involving constants
  unfolding Let-def ..

lemma Let-neg-numeral [simp]: Let (- numeral v) f = f (- numeral v)
  — Unfold all lets involving constants
  unfolding Let-def ..

declaration ⟨⟨
  let fun number-of thy T n =
    if not (Sign.of-sort thy (T, @ {sort numeral}))
    then raise CTERM (number-of, [])
    else Numeral.mk-cnumber (Thm.ctyp-of thy T) n;
  in
    K (Lin-Arith.add-simps (@ {thms arith-simps}) @ {thms more-arith-simps})
    @ {thms rel-simps}
    @ {thms pred-numeral-simps}
    @ {thms arith-special numeral-One}
    @ {thms of-nat-simps})
  #> Lin-Arith.add-simps [@ {thm Suc-numeral},
    @ {thm Let-numeral}, @ {thm Let-neg-numeral}, @ {thm Let-0}, @ {thm Let-1},
    @ {thm le-Suc-numeral}, @ {thm le-numeral-Suc},
    @ {thm less-Suc-numeral}, @ {thm less-numeral-Suc},
    @ {thm Suc-eq-numeral}, @ {thm eq-numeral-Suc},
    @ {thm mult-Suc}, @ {thm mult-Suc-right},
    @ {thm of-nat-numeral}]
  #> Lin-Arith.set-number-of number-of
end
⟩⟩

39.6.2 Simplification of arithmetic when nested to the right.

lemma add-numeral-left [simp]:
  numeral v + (numeral w + z) = (numeral(v + w) + z)
  by (simp-all add: add.assoc [symmetric])
lemma add-neg-numeral-left [simp]:
numeral v + (− numeral w + y) = (sub v w + y)
− numeral v + (numeral w + y) = (sub w v + y)
− numeral v + (− numeral w + y) = (− numeral(v + w) + y)
by (simp-all add: add.assoc [symmetric])

lemma mult-numeral-left [simp]:
numeral v ∗ (numeral w ∗ z) = (numeral(v ∗ w) ∗ z :: ‘a::semiring-numeral)
− numeral v ∗ (numeral w ∗ y) = (− numeral(v ∗ w) ∗ y :: ‘b::ring-1)
numeral v ∗ (− numeral w ∗ y) = (− numeral(v ∗ w) ∗ y :: ‘b::ring-1)
− numeral v ∗ (− numeral w ∗ y) = (numeral(v ∗ w) ∗ y :: ‘b::ring-1)
by (simp-all add: mult.assoc [symmetric])

hide-const (open) One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec

39.7 code module namespace
code-identifier
code-module Num → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

40 Power: Exponentiation
theory Power
imports Num Equiv-Relations
begin

40.1 Powers for Arbitrary Monoids
class power = one + times
begin

primrec power :: ‘a ⇒ nat ⇒ ‘a (infixr ^ 80) where
  power-0: a ^ 0 = 1
  | power-Suc: a ^ Suc n = a ∗ a ^ n

notation (latex output)
power (\cdot) [1000] 1000

notation (HTML output)
power (\cdot) [1000] 1000

Special syntax for squares.
abbreviation (xsymbols)
power2 :: ‘a ⇒ ‘a ((^) [1000] 999) where
x^2 ≡ x ^ 2
THEORY "Power"

notation (latex output)
    power2 \((^2) [1000] 999\)

notation (HTML output)
    \texttt{\textbackslash power2 \((^2) [1000] 999\)\textbackslash end}

context monoid-mult
begin

subclass \texttt{power }.

lemma \texttt{power-one \[simp\]}:
    \(1 ^ n = 1\)
    by (induct \(n\)) simp-all

lemma \texttt{power-one-right \[simp\]}:
    \(a ^ 1 = a\)
    by simp

lemma \texttt{power-commutes}:
    \(a ^ n * a = a * a ^ n\)
    by (induct \(n\)) (simp-all add: mult.assoc)

lemma \texttt{power-Suc2}:
    \(a ^ \text{Suc} \ n = a ^ n * a\)
    by (simp add: power-commutes)

lemma \texttt{power-add}:
    \(a ^ (m + n) = a ^ m * a ^ n\)
    by (induct \(m\)) (simp-all add: algebra-simps)

lemma \texttt{power-mult}:
    \(a ^ (m * n) = (a ^ m) ^ n\)
    by (induct \(n\)) (simp-all add: power-add)

lemma \texttt{power2-eq-square}:
    \(a ^ 2 = a * a\)
    by (simp add: numeral-2-eq-2)

lemma \texttt{power3-eq-cube}:
    \(a ^ 3 = a * a * a\)
    by (simp add: numeral-3-eq-3 mult.assoc)

lemma \texttt{power-even-eq}:
    \(a ^ (2 * n) = (a ^ n)^2\)
    by (subst mult.commute) (simp add: power-mult)

lemma \texttt{power-odd-eq}:
    \(a ^ \text{Suc} (2*n) = a * (a ^ n)^2\)
by (simp add: power-even-eq)

lemma power-numeral-even:
\[ z \cdot\ \text{numeral} \ (\text{Num.Bit0 \ w}) = (\text{let \ w = z \cdot (\text{numeral \ w}) in \ w \cdot w}) \]
unfolding numeral.Bit0 power-add Let-def ..

lemma power-numeral-odd:
\[ z \cdot\ \text{numeral} \ (\text{Num.Bit1 \ w}) = (\text{let \ w = z \cdot (\text{numeral \ w}) in \ z \cdot w \cdot w}) \]
unfolding numeral.Bit1 One-nat-def add-Suc-right add-0-right
unfolding power-Suc power-add Let-def mult.assoc ..

lemma funpow-times-power:
\[ (\times\ x \cdot f\ x) = \times\ (x \cdot f\ x) \]
proof (induct f x arbitrary: f)
case 0 then show ?case by (simp add: fun-eq-iff)
next
case (Suc n)
def g ≡ λx. f x − 1
with Suc have n = g x by simp
with Suc have times x "' g x = times (x "' f x) by simp
moreover from Suc g-def have f x = g x + f by simp
ultimately show ?case by (simp add: power-add funpow-add fun-eq-iff mult.assoc)
qed
end

context comm-monoid-mult
begin

lemma power-mult-distrib [field-simps]:
\[ (a \cdot b) \cdot n = (a \cdot n) \cdot (b \cdot n) \]
by (induct n) (simp-all add: ac-simps)
end

context semiring-numeral
begin

lemma numeral-sqr: numeral (\text{Num.sqr \ k}) = numeral k \cdot numeral k
by (simp only: sqr-conv-mult numeral-mult)

lemma numeral-pow: numeral (\text{Num.pow \ k \ l}) = numeral k \cdot numeral l
by (induct l, simp-all only: numeral-class,numeral.simps pow.simps
numeral-sqr numeral-mult power-add power-one-right)

lemma power-numeral [simp]: numeral k \cdot numeral l = numeral (\text{Num.pow \ k \ l})
by (rule numeral-pow [symmetric])
end
context semiring-1
begin

lemma of-nat-power:
of-nat (m ^ n) = of-nat m ^ n
by (induct n) (simp-all add: of-nat-mult)

lemma power-zero-numeral [simp]: (0::'a) ^ numeral k = 0
by (simp add: numeral-eq-Suc)

lemma zero-power2: 0^2 = 0
by (rule power-zero-numeral)

lemma one-power2: 1^2 = 1
by (rule power-one)
end

context comm-semiring-1
begin

The divides relation

lemma le-imp-power-dvd:
assumes m ≤ n shows a ^ m dvd a ^ n
proof
have a ^ n = a ^ (m + (n - m))
using ⟨m ≤ n⟩ by simp
also have ... = a ^ m * a ^ (n - m)
by (rule power-add)
finally show a ^ n = a ^ m * a ^ (n - m).
qed

lemma power-le-dvd:
a ^ n dvd b =⇒ m ≤ n =⇒ a ^ m dvd b
by (rule dvd-trans [OF le-imp-power-dvd])

lemma dvd-power-same:
x dvd y =⇒ x ^ n dvd y ^ n
by (induct n) (auto simp add: mult-dvd-mono)

lemma dvd-power-le:
x dvd y =⇒ m ≥ n =⇒ x ^ n dvd y ^ m
by (rule power-le-dvd [OF dvd-power-same])

lemma dvd-power [simp]:
assumes n > (0::nat) ∨ x = 1
shows x dvd (x ^ n)
using assms proof
assume $0 < n$
then have $x^n = x^{\text{Suc } (n - 1)}$ by simp
next
assume $x = 1$
then show $x\ dvd \ (x^n)$ by simp
qed
end

context ring_1
begin

lemma power-minus:
($-a)^n = (-1)^n * a^n$

proof (induct n)
case 0 show ?case by simp
next
case (Suc n) then show ?case
  by (simp del: power-Suc add: power-Suc2 mult.assoc)
qed

lemma power-minus-Bit0:
($-x)^\text{numeral } (\text{Num.Bit0 } k) = x^{\text{numeral } (\text{Num.Bit0 } k)}$

by (induct k, simp-all only: numeral-class.numeral.simps power-add
  power-one-right mult-minus-left mult-minus-right minus-minus)

lemma power-minus-Bit1:
($-x)^\text{numeral } (\text{Num.Bit1 } k) = - (x^{\text{numeral } (\text{Num.Bit1 } k)}$

by (simp only: eval-nat-numeral(3) power-Suc power-minus-Bit0 mult-minus-left)

lemma power2-minus [simp]:
($-a)^2 = a^2$

by (rule power-minus-Bit0)

lemma power-minus1-even [simp]:
$-(2n+1)$ * $2n+1 = 1$

proof (induct n)
case 0 show ?case by simp
next
case (Suc n) then show ?case by (simp add: power-add power2-eq-square)
qed

lemma power-minus1-odd:
$-1^{\text{Suc } (2n)} = -1$

by simp

lemma power-minus-even [simp]:
$(-a)^{2n} = a^{2n}$
by (simp add: power-minus [of a])

end

context ring-1-no-zero-divisors

begin

lemma field-power-not-zero:
\( a \neq 0 \implies a^{-n} \neq 0 \)
by (induct n) auto

lemma zero-eq-power2 [simp]:
\( a^2 = 0 \iff a = 0 \)
unfolding power2-eq-square by simp

lemma power2-eq-1-iff:
\( a^2 = 1 \iff a = 1 \lor a = -1 \)
unfolding power2-eq-square by (rule square-eq-1-iff)

end

context idom

begin

lemma power2-eq-iff:
\( x^2 = y^2 \iff x = y \lor x = -y \)
unfolding power2-eq-square by (rule square-eq-1-iff)

end

context division-ring

begin

FIXME reorient or rename to nonzero-inverse-power

lemma nonzero-power-inverse:
\( a \neq 0 \implies \text{inverse} (a^{-n}) = (\text{inverse} a)^{-n} \)
by (induct n)
(simp-all add: nonzero-inverse-mult-distrib power-commutes field-power-not-zero)

end

context field

begin

lemma nonzero-power-divide:
\( b \neq 0 \implies (a/b)^{-n} = a^{-n} / b^{-n} \)
by (simp add: divide-inverse power-mult-distrib nonzero-power-inverse)

end
40.2 Exponentiation on ordered types

context linordered-ring
begin

lemma sum-squares-ge-zero:
\( 0 \leq x \cdot x + y \cdot y \)
by (intro add-nonneg-nonneg zero-le-square)

lemma not-sum-squares-lt-zero:
\( \neg x \cdot x + y \cdot y < 0 \)
by (simp add: not-less sum-squares-ge-zero)

end

context linordered-semidom
begin

lemma zero-less-power [simp]:
\( 0 < a \Rightarrow 0 < a ^ n \)
by (induct n) simp-all

lemma zero-le-power [simp]:
\( 0 \leq a \Rightarrow 0 \leq a ^ n \)
by (induct n) simp-all

lemma power-mono:
\( a \leq b \Rightarrow 0 \leq a \Rightarrow a ^ n \leq b ^ n \)
by (induct n) (auto intro: mult-mono order-trans [of 0 a b])

lemma one-le-power [simp]: \( 1 \leq a \Rightarrow 1 \leq a ^ n \)
using power-mono [of 1 a n] by simp

lemma power-le-one: \( [0 \leq a; a \leq 1] \Rightarrow a ^ n \leq 1 \)
using power-mono [of a 1 n] by simp

lemma power-gt1-lemma:
assumes gt1: \( 1 < a \)
shows \( 1 < a \cdot a ^ n \)
proof –
  from gt1 have \( 0 \leq a \)
    by (fact order-trans [OF zero-le-one less-imp-le])
  have \( 1 \cdot 1 < a \cdot 1 \)
    using gt1 by simp
  also have \( \ldots \leq a \cdot a ^ n \)
    using gt1
  by (simp only: mult-mono \( 0 \leq a \) one-le-power order-less-imp-le order-refl)
finally show \( \text{thesis by simp} \)
qed

lemma power-gt1:
1 < a \implies 1 < a \cdot \text{Suc } n
\text{by (simp add: power-gt1-lemma)}

**lemma** one-less-power [simp]:
1 < a \implies 0 < n \implies 1 < a \cdot n
\text{by (cases n) (simp-all add: power-gt1-lemma)}

**lemma** power-le-imp-le-exp:
\text{assumes } gt1: 1 < a
\text{shows } a \cdot m \le a \cdot n \implies m \le n
\text{proof (induct m arbitrary: n)}
\text{case } 0
\text{show } \text{?case by simp}
\text{next}
\text{case } (\text{Suc } m)
\text{show } \text{?case}
\text{proof (cases n)}
\text{case } 0
\text{with Suc.prems Suc.hyps have } a \cdot a \cdot m \le 1 \text{ by simp}
\text{with gt1 show } \text{?thesis}
\text{by (force simp only: power-gt1-lemma}
\text{not-less [symmetric])}
\text{next}
\text{case } (\text{Suc } n)
\text{with Suc.prems Suc.hyps show } \text{?thesis}
\text{by (force dest: mult-left-le-imp-le}
\text{simp add: less-trans [OF zero-less-one gt1])}
\text{qed}
\text{qed}

Surely we can strengthen this? It holds for 0 < a < 1 too.

**lemma** power-inject-exp [simp]:
1 < a \implies a \cdot m = a \cdot n \iff m = n
\text{by (force simp add: order-antisym power-le-imp-le-exp)}

Can relax the first premise to (0::a) < a in the case of the natural numbers.

**lemma** power-less-imp-le-exp:
1 < a \implies a \cdot m < a \cdot n \implies m < n
\text{by (simp add: order-less-le \ of m n \ less-le \ of a \cdot m \ a \cdot n\ power-le-imp-le-exp)}

**lemma** power-strict-mono [rule-format]:
a < b \implies 0 \le a \implies 0 < n \implies a \cdot n < b \cdot n
\text{by (induct n)}
\text{(auto simp add: mult-strict-mono le-less-trans \ of 0 a b))}

Lemma for power-strict-decreasing

**lemma** power-Suc-less:
0 < a \implies a < 1 \implies a \cdot a \cdot n < a \cdot n
by (induct n)
  (auto simp add: mult-strict-left-mono)

lemma power-strict-decreasing [rule-format]:
  \( n < N \implies 0 < a \implies a < 1 \implies a ^ N < a ^ n \)
proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
    apply (auto simp add: power-Suc-less less-Suc-eq)
    apply (subgoal-tac a * a \^ N < 1 * a ^ n)
    apply simp
    apply (rule mult-strict-mono) apply auto
    done
qed

Proof resembles that of power-strict-decreasing

lemma power-decreasing [rule-format]:
  \( n \leq N \implies 0 \leq a \implies a \leq 1 \implies a ^ N \leq a ^ n \)
proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
    apply (auto simp add: le-Suc-eq)
    apply (subgoal-tac a * a ^ N \leq 1 * a ^ n, simp)
    apply (rule mult-mono) apply auto
    done
qed

lemma power-Suc-less-one:
  \( 0 < a \implies a < 1 \implies a ^ \text{Suc} n < 1 \)
using power-strict-decreasing [of 0 Suc n a] by simp

Proof again resembles that of power-strict-decreasing

lemma power-increasing [rule-format]:
  \( n \leq N \implies 1 \leq a \implies a ^ n \leq a ^ \text{Suc} N \)
proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
    apply (auto simp add: le-Suc-eq)
    apply (subgoal-tac 1 * a ^ n \leq a * a ^ Suc N, simp)
    apply (rule mult-mono) apply (auto simp add: order-trans [OF zero-le-one])
    done
qed

Lemma for power-strict-increasing

lemma power-less-power-Suc:
  \( 1 < a \implies a ^ n < a \cdot a ^ n \)
by (induct n) (auto simp add: mult-strict-left-mono less-trans [OF zero-less-one])

lemma power-strict-increasing [rule-format]:
  \( n < N \implies 1 < a \implies a^n < a^N \)
proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
    apply (auto simp add: power-less-power-Suc less-Suc-eq)
    apply (rule mult-strict-mono)
    apply (auto simp add: less-trans [OF zero-less-one])
    done
qed

lemma power-increasing-iff [simp]:
  \( 1 < b \implies b^x \leq b^y \iff x \leq y \)
by (blast intro: power-le-imp-le-exp power-increasing less-imp-le)

lemma power-strict-increasing-iff [simp]:
  \( 1 < b \implies b^x < b^y \iff x < y \)
by (blast intro: power-less-imp-less-exp power-strict-increasing)

lemma power-le-imp-le-base:
  assumes le: \( a^{\mathrm{Suc} \; n} \leq b^{\mathrm{Suc} \; n} \)
  and ynonneg: \( 0 \leq b \)
  shows \( a \leq b \)
proof (rule ccontr)
  assume \( \sim a \leq b \)
  then have \( b < a \) by (simp only: linorder-not-le)
  then have \( b^{\mathrm{Suc} \; n} < a^{\mathrm{Suc} \; n} \)
    by (simp only: assms power-strict-mono)
  from le and this show False
    by (simp add: linorder-not-less [symmetric])
qed

lemma power-less-imp-less-base:
  assumes less: \( a^n < b^n \)
  and nonneg: \( 0 \leq b \)
  shows \( a < b \)
proof (rule contrapos-pp [OF less])
  assume \( \sim a < b \)
  hence \( b \leq a \) by (simp only: linorder-not-less)
  hence \( b^n \leq a^n \) using nonneg by (rule power-mono)
  thus \( \sim a^n < b^n \) by (simp only: linorder-not-less)
qed

lemma power-inject-base:
  \( a^{\mathrm{Suc} \; n} = b^{\mathrm{Suc} \; n} \implies 0 \leq a \implies 0 \leq b \implies a = b \)
by \(\text{blast intro: power-le-imp-le-base antisym eq-refl sym}\)

**lemma** power-eq-imp-eq-base:
\[a ^ n = b ^ n \implies 0 \leq a \implies 0 \leq b \implies 0 < n \implies a = b\]
by (cases n) (simp-all del: power-Suc, rule power-inject-base)

**lemma** power2-le-imp-le:
\[x^2 \leq y^2 \implies 0 \leq y \implies x \leq y\]
unfolding numeral-2-eq-2 by (rule power-le-imp-le-base)

**lemma** power2-less-imp-less:
\[x^2 < y^2 \implies 0 \leq y \implies x < y\]
by (rule power-less-imp-less-base)

**lemma** power2-eq-imp-eq:
\[x^2 = y^2 \implies 0 \leq x \implies 0 \leq y \implies x = y\]
unfolding numeral-2-eq-2 by (erule (2) power-eq-imp-eq-base) simp

end

context linordered-ring-strict
begin

**lemma** sum-squares-eq-zero-iff:
\[x \times x + y \times y = 0 \iff x = 0 \land y = 0\]
by (simp add: add-nonneg-eq-0-iff)

**lemma** sum-squares-le-zero-iff:
\[x \times x + y \times y \leq 0 \iff x = 0 \land y = 0\]
by (simp add: le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff)

**lemma** sum-squares-gt-zero-iff:
\[0 < x \times x + y \times y \iff x \neq 0 \lor y \neq 0\]
by (simp add: not-le [symmetric] sum-squares-le-zero-iff)

end

context linordered-idom
begin

**lemma** power-abs:
\[\text{abs} \ (a ^ n) = \text{abs} \ a ^ n\]
by (induct n) (auto simp add: abs-mult)

**lemma** abs-power-minus [simp]:
\[\text{abs} \ ((-a) ^ n) = \text{abs} \ (a ^ n)\]
by (simp add: power-abs)

**lemma** zero-less-power-abs-iff [simp]:

\[ 0 < \text{abs } a \cdot n \iff a \neq 0 \lor n = 0 \]

**proof** (induct \( n \))
- **case** 0 **show** ?case by simp
**next**
- **case** (Suc \( n \)) **show** ?case by (auto simp add: Suc zero-less-mult-iff)

**qed**

**lemma** zero-le-power-abs [simp]:
\[ 0 \leq \text{abs } a \cdot n \]
by (rule zero-le-power [OF abs-ge-zero])

**lemma** zero-le-power2 [simp]:
\[ 0 \leq a^2 \]
by (simp add: power2-eq-square)

**lemma** zero-less-power2 [simp]:
\[ 0 < a^2 \iff a \neq 0 \]
by (force simp add: power2-eq-square zero-less-mult-iff linorder-neq-iff)

**lemma** power2-less-0 [simp]:
\[ \neg a^2 < 0 \]
by (force simp add: power2-eq-square mult-less-0-iff)

**lemma** abs-power2 [simp]:
\[ \text{abs } (a^2) = a^2 \]
by (simp add: power2-eq-square abs-mult abs-mult-self)

**lemma** power2-abs [simp]:
\[ (\text{abs } a)^2 = a^2 \]
by (simp add: power2-eq-square abs-mult-self)

**lemma** odd-power-less-zero:
\[ a < 0 \implies a \cdot \text{Suc } (2\cdot n) < 0 \]
**proof** (induct \( n \))
- **case** 0 **then show** ?case by simp
**next**
- **case** (Suc \( n \))
  - **have** \( a \cdot \text{Suc } (2 \cdot \text{Suc } n) = (a\cdot a) \cdot a \cdot \text{Suc}(2\cdot n) \)
    - by (simp add: ac-simps power-add power2-eq-square)
  **thus** ?case
    - by (simp del: power-Suc add: Suc mult-less-0-iff mult-neg-neg)

**qed**

**lemma** odd-0-le-power-imp-0-le:
\[ 0 \leq a \cdot \text{Suc } (2\cdot n) \implies 0 \leq a \]
**using** odd-power-less-zero [of a n]
  - by (force simp add: linorder-not-less [symmetric])
lemma zero-le-even-power [simp]:
\( \theta \leq a \cdot (2^{\cdot} n) \)

proof (induct n)
  case 0
    show ?case by simp
next
  case (Suc n)
    have \( a \cdot (2^{\cdot} \text{Suc} n) = (a \cdot a) \cdot (2^{\cdot} n) \)
    by (simp add: ac_simps power-add power2_eq_square)
    thus ?case
    by (simp add: Suc zero-le_mult_iff)
qed

lemma sum-power2_ge_zero:
\( \theta \leq x^2 + y^2 \)
by (intro add-nonneg-nonneg zero_le_power2)

lemma not-sum-power2_lt_zero:
\( \neg x^2 + y^2 < 0 \)
unfolding not_less by (rule sum-power2_ge_zero)

lemma sum-power2_eq_zero_iff:
\( x^2 + y^2 = 0 \iff x = 0 \land y = 0 \)
unfolding power2_eq_square by (simp add: add-nonneg_eq_0_iff)

lemma sum-power2_le_zero_iff:
\( x^2 + y^2 \leq 0 \iff x = 0 \land y = 0 \)
by (simp add: le_less sum-power2_eq_zero_iff not-sum-power2_lt_zero)

lemma sum-power2_gt_zero_iff:
\( 0 < x^2 + y^2 \iff x \neq 0 \lor y \neq 0 \)
unfolding not_le [symmetric] by (simp add: sum-power2_le_zero_iff)

end

40.3 Miscellaneous rules

lemma self_le_power:
  fixes \( x \cdot a \cdot \text{linordered-semidom} \)
  shows \( 1 \leq x \iff 0 < n \implies x \leq x \cdot n \)
  using power-increasing[of 1 n x] power-one_right[of x] by auto

lemma power_eq_if: \( p \cdot m = \text{(if m=0 then 1 else p \cdot (p \cdot (m - 1)))} \)
  unfolding One_not_def by (cases m) simp_all

lemma power2_sum:
  fixes \( x \cdot y \cdot \text{a\cdot comm-semiring-1} \)
  shows \( (x + y)^2 = x^2 + y^2 + 2 \cdot x \cdot y \)
  by (simp add: algebra_simps power2_eq_square mult_right)
lemma power2-diff:
  fixes x y :: 'a::comm-ring-1
  shows \((x - y)^2 = x^2 + y^2 - 2 * x * y\) 
  by (simp add: ring-distrib power2-eq-square mult-2) (rule mult.commute)

lemma power-0-Suc [simp]:
  \((0::'a::{power, semiring-0}) ^ Suc n = 0\)
  by simp

It looks plausible as a simprule, but its effect can be strange.

lemma power-0-left:
  \(0 ^ n = (\text{if } n = 0 \text{ then } 1 \text{ else } (0::'a::{power, semiring-0}))\)
  by (induct n) simp-all

lemma power-eq-0-iff [simp]:
  \(a ^ n = 0 \iff a = (0::'a::{mult-zero, zero-neq-one, no-zero-divisors, power}) \wedge n \neq 0\)
  by (induct n)
    (auto simp add: no-zero-divisors elim: contrapos-pp)

lemma (in field) power-diff:
  assumes nz: \(a \neq 0\)
  shows \(n \leq m \Rightarrow a ^ (m - n) = a ^ m / a ^ n\)
  by (induct m n rule: diff-induct) (simp-all add: nz field-power-not-zero)

Perhaps these should be simprules.

lemma power-inverse:
  fixes a :: 'a::division-ring-inverse-zero
  shows inverse (a ^ n) = inverse a ^ n
  apply (cases a = 0)
  apply (simp add: power-0-left)
  apply (simp add: nonzero-power-inverse)
  done

lemma power-one-over:
  \(1 / (a::'a::{field-inverse-zero, power}) ^ n = (1 / a) ^ n\)
  by (simp add: divide-inverse) (rule power-inverse)

lemma power-divide [field-simps, divide-simps]:
  \((a / b) ^ n = (a::'a::field-inverse-zero) ^ n / b ^ n\)
  apply (cases b = 0)
  apply (simp add: power-0-left)
  apply (rule nonzero-power-divide)
  apply assumption
  done

Simprules for comparisons where common factors can be cancelled.

lemmas zero-compare-simps =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

40.4  Exponentiation for the Natural Numbers

lemma nat-one-le-power [simp]:
  Suc 0 ≤ i ⇒ Suc 0 ≤ i ^ n
  by (rule one-le-power [of i n, unfolded One-nat-def])

lemma nat-zero-less-power-iff [simp]:
  x ^ n > 0 ⇔ x > (0 :: nat) ∨ n = 0
  by (induct n) auto

lemma nat-power-eq-Suc-0-iff [simp]:
  x ^ m = Suc 0 ⇔ m = 0 ∨ x = Suc 0
  by (induct m) auto

lemma power-Suc-0 [simp]:
  Suc 0 ^ n = Suc 0
  by simp

Valid for the naturals, but what if 0 < i < 1? Premises cannot be weakened:
consider the case where i = (0 :: a), m = (1 :: a) and n = (0 :: a).

lemma nat-power-less-imp-less:
  assumes nonneg: 0 < (i :: nat)
  assumes less: i ^ m < i ^ n
  shows m < n
proof (cases i = 1)
  case True with less power-one [where 'a = nat] show ?thesis by simp
next
  case False with nonneg have 1 < i by auto
  from power-strict-increasing-iff [OF this] less show ?thesis ..
qed

lemma power-dvd-imp-le:
  i ^ m dvd i ^ n ⇒ (1 :: nat) < i ⇒ m ≤ n
  apply (rule power-le-imp-le-exp, assumption)
  apply (erule dvd-imp-le, simp)
  done

lemma power2-nat-le-eq-le:
  fixes m n :: nat
  shows m ^ 2 ≤ n ^ 2 ⇔ m ≤ n
  by (auto intro: power2-le-imp-le power-mono)
lemma power2-nat-le-imp-le:
fixes m n :: nat
assumes \( m^2 \leq n \)
shows \( m \leq n \)
proof (cases m)
case 0 then show \( \text{thesis} \) by simp
next
case (Suc k)
show \( \text{thesis} \)
proof (rule ccontr)
  assume \( \neg m \leq n \)
  then have \( n < m \) by simp
  with assms Suc show False
  by (auto simp add: algebra-simps) (simp add: power2-eq-square)
qed
qed

40.4.1 Cardinality of the Powerset

lemma card-UNIV-bool [simp]: card (UNIV :: bool set) = 2
unfolding UNIV-bool by simp

lemma card-Pow: finite A \( \Rightarrow \) card (Pow A) = \( 2 \cdot \) card A
proof (induct rule: finite-induct)
case empty
  show \( \text{?case} \) by auto
next
case (insert x A)
then have inj-on (insert x) (Pow A)
  unfolding inj-on-def by (blast elim!: equalityE)
then have card (Pow A) + card (insert x · Pow A) = \( 2 \cdot 2 \cdot \) card A
  by (simp add: mult-2 card-image Pow-insert insert.hyps)
then show \( \text{?case} \) using insert
  apply (simp add: Pow-insert)
  apply (subst card-Un-disjoint, auto)
done
qed

40.4.2 Generalized sum over a set

lemma setsum-zero-power [simp]:
fixes c :: nat \Rightarrow 'a::division-ring
shows \( \sum_{i \in A.} c \cdot 0^i \) = (if finite A \& 0 \in A then c 0 else 0)
apply (cases finite A)
by (induction A rule: finite-induct) auto

lemma setsum-zero-power' [simp]:
fixes c :: nat \Rightarrow 'a::field
shows \( \sum_{i \in A.} c \cdot 0^i / d \cdot i \) = (if finite A \& 0 \in A then c 0 / d 0 else 0)
using setsum-zero-power [of \( \lambda i. c \cdot i / d \cdot i \cdot A \)]
by auto

40.4.3 Generalized product over a set

lemma setprod-constant: finite A ==> (\Pi x\in A. (y::'a::{comm-monoid-mul}))(x) = y^' (card A)
apply (erule finite-induct)
apply auto
done

lemma setprod-power-distrib:
  fixes f :: 'a => 'b::comm-semiring-1
  shows setprod f A ^ n = setprod (\lambda x. (f x) ^ n) A
proof (cases finite A)
  case True then show ?thesis
  by (induct A rule: finite-induct) (auto simp add: power-mult-distrib)
next
  case False then show ?thesis
  by simp
qed

lemma setprod-gen-delta:
  assumes fS: finite S
  shows setprod (\lambda k. if k=a then b k else c) S = (if a \in S then (b a ::'a::comm-monoid-mul) ^ c ^ (card S - 1) else c ^ card S)
proof
  let ?f = (\lambda k. if k=a then b k else c)
  {assume a: a \notin S
    hence \forall k\in S. if k = c by simp
    hence ?thesis using a setprod-constant[OF fS, of c] by simp }
  moreover
  {assume a: a \in S
    let ?A = S - {a}
    let ?B = {a}
    have eq: S = ?A \cup ?B using a by blast
    have dj: ?A \cap ?B = {} by simp
    from fS have fAB: finite ?A finite ?B by auto
    have fA0: setprod ?f ?A = setprod (\lambda i. c) ?A
    apply (rule setprod.cong) by auto
    have cA: card ?A = card S - 1 using fS a by auto
    have fA1: setprod ?f ?A = c ^ card ?A unfolding fA0 apply (rule setprod-constant)
    using fS by auto
    using setprod.union-disjoint[OF fAB dj, of ?f, unfolded eq[symmetric]]
    by simp
    then have ?thesis using a cA
    by (simp add: fA1 field-simps cong add: setprod.cong cong del: if-weak-cong)
  }
ultimately show ?thesis by blast
qed
THEORY “Meson”

lemma Domain-dprod [simp]: Domain (dprod r s) = uprod (Domain r) (Domain s)
  by auto

lemma Domain-dsum [simp]: Domain (dsum r s) = usum (Domain r) (Domain s)
  by auto

40.5 Code generator tweak

lemma power-power-power [code]:
  power = power.power (1::a::{power}) (op *)
unfolding power-def power.power-def ..

declare power.power.simps [code]

code-identifier
code-module Power ↦ (SML) Arith and (OCaml) Arith and (Haskell) Arith
end

41 Meson: MESON Proof Method

theory Meson
imports Nat
begin

41.1 Negation Normal Form

de Morgan laws

lemma not-conjD: ∼(P&Q) ==> ∼P | ∼Q
  and not-disjD: ∼(P|Q) ==> ∼P & ∼Q
  and not-notD: ∼∼P ==> P
  and not-allD: !!P. ∼(∀x. P(x)) ==> ∃x. ∼P(x)
  and not-exD: !!P. ∼(∃x. P(x)) ==> ∀x. ∼P(x)
  by fast+

Removal of −−> and <−−> (positive and negative occurrences)

lemma imp-to-disjD: P −−> Q ==> ∼P | Q
  and not-impD: ∼(P −−> Q) ==> P & ∼Q
  and iff-to-disjD: P=Q ==> (∼P | Q) & (∼Q | P)
  and not-iffD: ∼(P=Q) ==> (P | Q) & (∼P | ∼Q)
  — Much more efficient than P ∧ ∼ Q ∨ Q ∧ ∼ P for computing CNF
  and not-refl-disj-D: x ∼= x | P ==> P
  by fast+
41.2 Pulling out the existential quantifiers

Conjunction

**lemma** `conj-exD1`: $!P \land Q \implies \exists x. P(x) \land Q$

and **lemma** `conj-exD2`: $!P \land (\exists x. Q(x)) \implies \exists x. P \land Q(x)$

by `fast`+

Disjunction

**lemma** `disj-exD1`: $!P \lor Q \implies \exists x. P(x) \lor Q(x)$

— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!

— With ex-Skolemization, makes fewer Skolem constants

and **lemma** `disj-exD2`: $!P \land (\exists x. Q(x)) \implies \exists x. P \land Q(x)$

by `fast`+

**lemma** `disj-assoc`: $(P \lor Q) \lor R \implies P \lor (Q \lor R)$

and **lemma** `disj-comm`: $P \lor Q \implies Q \lor P$

and **lemma** `disj-FalseD1`: $\bot \lor P \implies P$

and **lemma** `disj-FalseD2`: $P \lor \bot \implies P$

by `fast`+

Generation of contrapositives

Inserts negated disjunct after removing the negation; $P$ is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

**lemma** `make-neg-rule`: $\neg P \land Q \implies (\neg P \implies P) \implies Q$

by `blast`

Version for Plaisted's "Positive refinement" of the Meson procedure

**lemma** `make-refined-neg-rule`: $\neg P \land Q \implies (P \implies \neg P) \implies Q$

by `blast`

$P$ should be a literal

**lemma** `make-pos-rule`: $P \land Q \implies (P \implies \neg P) \implies Q$

by `blast`

Versions of `make-neg-rule` and `make-pos-rule` that don’t insert new assumptions, for ordinary resolution.

**lemmas** `make-neg-rule‘ = make-refined-neg-rule`

**lemma** `make-pos-rule‘`$: (P \land Q; \neg P) \implies Q$

by `blast`

Generation of a goal clause – put away the final literal

**lemma** `make-neg-goal`: $\neg P \implies (\neg P \implies P) \implies \bot$

by `blast`
lemma make-pos-goal: $P \Longrightarrow ((P \Longrightarrow \neg P) \Longrightarrow \text{False})$
by blast

41.3 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma conj-forward: $[[P \& Q']; P' \Longrightarrow P; Q' \Longrightarrow Q]] \Longrightarrow P\&Q$
by blast

lemma disj-forward: $[[P | Q']; P' \Longrightarrow P; Q' \Longrightarrow Q]] \Longrightarrow P|Q$
by blast

lemma disj-forward2:
$[[P | Q']; P' \Longrightarrow P; [Q'; P==>\text{False}]] \Longrightarrow Q]] \Longrightarrow P|Q$
apply blast
done

lemma all-forward: $[[\forall x. P'(x); \forall x. P'(x) \Longrightarrow P(x)]] \Longrightarrow \forall x. P(x)$
by blast

lemma ex-forward: $[[\exists x. P'(x); \exists x. P'(x) \Longrightarrow P(x)]] \Longrightarrow \exists x. P(x)$
by blast

41.4 Clausification helper

lemma TruepropI: $P \equiv Q \Longrightarrow \text{Trueprop} P \equiv \text{Trueprop} Q$
by simp

lemma ext-cong-neq: $F \neq F \Longrightarrow F \neq F \land (\exists x. g \neq h x)$
apply (erule contrapos-np)
apply clarsimp
apply (rule cong[where f = F])
by auto

Combinator translation helpers

definition COMBI :: 'a \Rightarrow 'a where
COMBI P = P

definition COMBK :: 'a \Rightarrow 'b \Rightarrow 'a where
COMBK P Q = P

definition COMBB :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c where
COMBB P Q R = P (Q R)

definition COMBC :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c where
COMBC P Q R = P R Q
definition COMBS :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c where
COMBS P Q R = P R (Q R)

lemma abs-S: λx. (f x) (g x) ≡ COMBS f g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBS-def)
done

lemma abs-I: λx. x ≡ COMBI
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBI-def)
done

lemma abs-K: λx. y ≡ COMBK y
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBK-def)
done

lemma abs-B: λx. a (g x) ≡ COMBB a g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBB-def)
done

lemma abs-C: λx. (f x) b ≡ COMBC f b
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBC-def)
done

41.5 Skolemization helpers

definition skolem :: 'a ⇒ 'a where
skolem = (λx. x)

lemma skolem-COMBK-iff: P ←→ skolem (COMBK P (i::nat))
unfolding skolem-def COMBK-def by (rule refl)

lemmas skolem-COMBK-I = iffD1 [OF skolem-COMBK-iff]
lemmas skolem-COMBK-D = iffD2 [OF skolem-COMBK-iff]

41.6 Meson package

ML-file Tools/Meson/meson.ML
ML-file Tools/Meson/meson-clausify.ML
ML-file Tools/Meson/meson-tactic.ML
42 ATP: Automatic Theorem Provers (ATPs)

theory ATP
imports Meson
begin

42.1 ATP problems and proofs

ML-file Tools/ATP/atp-util.ML
ML-file Tools/ATP/atp-problem.ML
ML-file Tools/ATP/atp-proof.ML
ML-file Tools/ATP/atp-proof-redirect.ML

42.2 Higher-order reasoning helpers

definition fFalse :: bool where
fFalse ↔ False

definition fTrue :: bool where
fTrue ↔ True

definition fNot :: bool ⇒ bool where
fNot P ↔ ¬ P

definition fComp :: ('a ⇒ bool) ⇒ 'a ⇒ bool where
fComp P = (λx. ¬ P x)

definition fconj :: bool ⇒ bool ⇒ bool where
fconj P Q ↔ P ∧ Q

definition fdisj :: bool ⇒ bool ⇒ bool where
fdisj P Q ↔ P ∨ Q

definition fimplies :: bool ⇒ bool ⇒ bool where
fimplies P Q ↔ (P → Q)

definition fAll :: ('a ⇒ bool) ⇒ bool where
theory "ATP"

fAll P \iff\ All P

definition fEx :: ('a \Rightarrow bool) \Rightarrow bool where
fEx P \iff\ Ex P

definition fequal :: 'a \Rightarrow 'a \Rightarrow bool where
fequal x y \iff\ (x = y)

lemma fTrue-ne-fFalse: fFalse \neq fTrue
unfolding fFalse-def fTrue-def by simp

lemma fNot-table:
\[ fNot fFalse = fTrue \]
\[ fNot fTrue = fFalse \]
unfolding fFalse-def fTrue-def fNot-def by auto

lemma fconj-table:
\[ fconj fFalse P = fFalse \]
\[ fconj P fFalse = fFalse \]
\[ fconj fTrue fTrue = fTrue \]
unfolding fFalse-def fTrue-def fconj-def by auto

lemma fdisj-table:
\[ fdisj fTrue P = fTrue \]
\[ fdisj P fTrue = fTrue \]
\[ fdisj fFalse fFalse = fFalse \]
unfolding fFalse-def fTrue-def fdisj-def by auto

lemma fimplies-table:
\[ fimplies P fTrue = fTrue \]
\[ fimplies fFalse P = fTrue \]
\[ fimplies fTrue fFalse = fFalse \]
unfolding fFalse-def fTrue-def fimplies-def by auto

lemma fAll-table:
\[ Ex (fComp P) \vee fAll P = fTrue \]
\[ All P \vee fAll P = fFalse \]
unfolding fFalse-def fTrue-def fComp-def fAll-def by auto

lemma fEx-table:
\[ All (fComp P) \vee fEx P = fTrue \]
\[ Ex P \vee fEx P = fFalse \]
unfolding fFalse-def fTrue-def fComp-def fEx-def by auto

lemma fequal-table:
\[ fequal x x = fTrue \]
\[ x = y \vee fequal x y = fFalse \]
unfolding fFalse-def fTrue-def fequal-def by auto
THEORY "ATP"

lemma fNot-law:
\( f\text{Not} \ P \not= \ P \)

unfolding fNot-def by auto

lemma fComp-law:
\( f\text{Comp} \ P \ x \iff \sim P \ x \)

unfolding fComp-def..

lemma fconj-laws:
\( f\text{conj} \ P \ P \iff P \)
\( f\text{conj} \ P \ Q \iff f\text{conj} \ Q \ P \)
\( f\text{Not} \ (f\text{conj} \ P \ Q) \iff f\text{disj} \ (f\text{Not} \ P) \ (f\text{Not} \ Q) \)

unfolding fNot-def fconj-def fdisj-def by auto

lemma fdisj-laws:
\( f\text{disj} \ P \ P \iff P \)
\( f\text{disj} \ P \ Q \iff f\text{disj} \ Q \ P \)
\( f\text{Not} \ (f\text{disj} \ P \ Q) \iff f\text{conj} \ (f\text{Not} \ P) \ (f\text{Not} \ Q) \)

unfolding fNot-def fconj-def fdisj-def by auto

lemma fimplies-laws:
\( f\text{implies} \ P \ Q \iff f\text{disj} \ (\sim P) Q \)
\( f\text{Not} \ (f\text{implies} \ P \ Q) \iff f\text{conj} \ P \ (f\text{Not} \ Q) \)

unfolding fNot-def fconj-def fdisj-def fimplies-def by auto

lemma fAll-law:
\( f\text{Not} \ (f\text{All} \ R) \iff f\text{Ex} \ (f\text{Comp} \ R) \)

unfolding fNot-def fComp-def fAll-def fEx-def by auto

lemma fEx-law:
\( f\text{Not} \ (f\text{Ex} \ R) \iff f\text{All} \ (f\text{Comp} \ R) \)

unfolding fNot-def fComp-def fAll-def fEx-def by auto

lemma fequal-laws:
\( f\text{equal} \ x \ y = f\text{equal} \ y \ x \)
\( f\text{equal} \ x \ y = f\text{False} \lor f\text{equal} \ y \ z = f\text{False} \lor f\text{equal} \ x \ z = f\text{True} \)
\( f\text{equal} \ x \ y = f\text{False} \lor f\text{equal} \ (f \ x) \ (f \ y) = f\text{True} \)

unfolding fFalse-def fTrue-def fequal-def by auto

42.3 Waldmeister helpers

lemmas waldmeister-fol = simp-thms(1–34) disj-absorb disj-left-absorb conj-absorb conj-left-absorb eq-ac disj-comms disj-assoc conj-comms conj-assoc

42.4 Basic connection between ATPs and HOL

ML-file Tools/lambda-lifting.ML
ML-file Tools/monomorph.ML
ML-file Tools/ATP/atp-problem-generate.ML
43 Metis: Metis Proof Method

theory Metis
imports ATP
begin

declare [[ML-print-depth = 0]]
ML-file ~/src/Tools/Metis/metis.ML
declare [[ML-print-depth = 10]]

43.1 Literal selection and lambda-lifting helpers

definition select :: 'a ⇒ 'a where
  select = (λx. x)

lemma not-atomize: (¬ A ⇒ False) ≡ Trueprop A
  by (cut-tac atomize-not [of ¬ A]) simp

lemma atomize-not-select: (A ⇒ select False) ≡ Trueprop (¬ A)
  unfolding select-def by (rule atomize-not)

lemma not-atomize-select: (¬ A ⇒ select False) ≡ Trueprop A
  unfolding select-def by (rule not-atomize)

lemma select-FalseI: False ⇒ select False by simp

definition lambda :: 'a ⇒ 'a where
  lambda = (λx. x)

lemma eq-lambdaI: x ≡ y ⇒ x ≡ lambda y
  unfolding lambda-def by assumption

43.2 Metis package

ML-file Tools/Metis/metis-generate.ML
ML-file Tools/Metis/metis-reconstruct.ML
ML-file Tools/Metis/metis-tactic.ML

setup "Metis-Tactic.setup"

hide-fact (open) waldmeister-fol
THEORY “Option” 743

hide-const (open) select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx fequal lambda
hide-fact (open) select-def not-atomize atomize-not-select select-FalseI
fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def
fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table
fequal-table fAll-table fEx-table fNot-law fComp-law fconj-laws fdisj-laws fimplies-laws
fequal-laws fAll-law fEx-law lambda-def eq-lambdaI

end

44 Option: Datatype option

theory Option imports BNF-LFP Datatype Finite-Set begin

datatype-new 'a option =
  None
| Some (the: 'a)

datatype-compat option

lemma [case-names None Some, cases type: option]:
  — for backward compatibility - names of variables differ
  (y = None ==> P) ==> (\a. y = Some a ==> P) ==> P
by (rule option.exhaust)

lemma [case-names None Some, induct type: option]:
  — for backward compatibility - names of variables differ
  P None ==> (\option. P (Some option)) ==> P option
by (rule option.induct)

Compatibility:
setup ⟨⟨ Sign.mandatory-path option ⟩⟩

lemmas inducts = option.induct
lemmas cases = option.case

setup ⟨⟨ Sign.parent-path ⟩⟩

lemma not-None-eq [iff]: (x ~= None) = (EX y. x = Some y)
by (induct x) auto

lemma not-Some-eq [iff]: (ALL y. x ~= Some y) = (x = None)
by (induct x) auto

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.
lemma inj-Some [simp]: inj-on Some A
by (rule inj-onI) simp

lemma case-optionE:
  assumes c: (case x of None => P | Some y => Q y)
  obtains
     (None) x = None and P | (Some) y where x = Some y and Q y
  using c by (cases x) simp-all

lemma split-option-all: (∀x. P x) ↔ P None ∧ (∀x. P (Some x))
by (auto intro: option.induct)

lemma split-option-ex: (∃x. P x) ↔ P None ∨ (∃x. P (Some x))
using split-option-all[of λx. ¬P x] by blast

lemma UNIV-option-conv: UNIV = insert None (range Some)
by(auto intro: classical)

44.0.1 Operations

lemma ospec [dest]: (ALL x: set-option A. P x) ==> A = Some x ==> P x
  by simp

setup ⟨⟨ map-theory-claset (fn ctxt => ctxt addSD2 (ospec, @{thm ospec})) ⟩⟩

lemma elem-set [iff]: (x : set-option xo) = (xo = Some x)
  by (cases xo) auto

lemma set-empty-eq [simp]: (set-option xo = {}) = (xo = None)
  by (cases xo) auto

lemma map-option-case: map-option f y = (case y of None => None | Some x => Some (f x))
  by (auto split: option.split)

lemma map-option-is-None [iff]:
  (map-option f opt = None) = (opt = None)
  by (simp add: map-option-case split add: option.split)

lemma map-option-eq-Some [iff]:
  (map-option f xo = Some y) = (EX z. xo = Some z & f z = y)
  by (simp add: map-option-case split add: option.split)

lemma map-option-o-case-sum [simp]:
  map-option f o case-sum g h = case-sum (map-option f o g) (map-option f o h)
  by (rule o-case-sum)

lemma map-option-cong: x = y ==> (∀a. y = Some a ==> f a = g a) ==>
map-option f x = map-option g y
by (cases x) auto

functor map-option: map-option proof –
fix f g
show map-option f o map-option g = map-option (f o g)
proof
fix x
show (map-option f o map-option g) x = map-option (f o g) x
by (cases x) simp-all
qed
next
show map-option id = id
proof
fix x
show map-option id x = id x
by (cases x) simp-all
qed
qed

lemma case-map-option [simp]:
case-option g h (map-option f x) = case-option g (h o f) x
by (cases x) simp-all

primrec bind :: 'a option ⇒ ('a ⇒ 'b option) ⇒ 'b option
where
bind-lzero: bind None f = None |
bind-lunit: bind (Some x) f = f x

lemma bind-runit[simp]: bind x Some = x
by (cases x) auto

lemma bind-assoc[simp]: bind (bind x f) g = bind x (λy. bind (f y) g)
by (cases x) auto

lemma bind-rzero[simp]: bind x (λx. None) = None
by (cases x) auto

lemma bind-cong: x = y ⇒ (∀a. y = Some a ⇒ f a = g a) ⇒ bind x f = bind y g
by (cases x) auto

definition these :: 'a option set ⇒ 'a set
where
these A = the ' {x ∈ A. x ≠ None}

lemma these-empty [simp]:
these {} = {}
by (simp add: these-def)
lemma these-insert-None [simp]:
  these (insert None A) = these A
by (auto simp add: these-def)

lemma these-insert-Some [simp]:
  these (insert (Some x) A) = insert x (these A)
proof -
  have \{ y ∈ insert (Some x) A. y ≠ None \} = insert (Some x) \{ y ∈ A. y ≠ None \}
    by auto
  then show \?thesis by (simp add: these-def)
qed

lemma in-these-eq:
  x ∈ these A ←→ Some x ∈ A
proof
  assume Some x ∈ A
  then obtain B where A = insert (Some x) B by auto
  then show x ∈ these A by (auto simp add: these-def intro!: image-eqI)
next
  assume x ∈ these A
  then show Some x ∈ A by (auto simp add: these-def)
qed

lemma these-image-Some-eq [simp]:
  these (Some ' A) = A
by (auto simp add: these-def intro!: image-eqI)

lemma Some-image-these-eq:
  Some ' these A = \{ x ∈ A. x ≠ None \}
by (auto simp add: these-def image-image intro!: image-eqI)

lemma these-empty-eq:
  these B = {} ←→ B = {} ∨ B = {None}
by (auto simp add: these-def)

lemma these-not-empty-eq:
  these B ≠ {} ←→ B ≠ {} ∧ B ≠ {None}
by (auto simp add: these-empty-eq)

hide-const (open) bind these
hide-fact (open) bind-cong

44.0.2 Interaction with finite sets

lemma finite-option-UNIV [simp]:
  finite (UNIV :: 'a option set) = finite (UNIV :: 'a set)
by (auto simp add: UNIV-option-cone elim: finite-imageD intro: inj-Some)

instance option :: (finite) finite
44.0.3 Code generator setup

definition is-none :: 'a option ⇒ bool where
    [code-post]: is-none x ⇔ x = None

lemma is-none-code [code]:
    shows is-none None ⇔ True
    and is-none (Some x) ⇔ False

unfolding is-none-def by simp-all

lemma [code-unfold]:
    HOL.equal x None ⇔ is-none x
    HOL.equal None = is-none
    by (auto simp add: equal is-none-def)

hide-const (open) is-none

code-printing
    type-constructor option →
    (SML) - option
    and (OCaml) - option
    and (Haskell) Maybe -
    and (Scala) !Option[(-)]
| constant None →
    (SML) NONE
    and (OCaml) None
    and (Haskell) Nothing
    and (Scala) !None
| constant Some →
    (SML) SOME
    and (OCaml) Some -
    and (Haskell) Just
    and (Scala) Some
| class-instance option :: equal →
    (Haskell) –
| constant HOL.equal :: 'a option ⇒ 'a option ⇒ bool ⇒
    (Haskell) infix 4 ==

code-reserved SML
    option NONE SOME

code-reserved OCaml
    option None Some

code-reserved Scala
    Option None Some
45 Transfer: Generic theorem transfer using relations

theory Transfer
imports Hilbert-Choice BNF-FP-Base Metis Option
begin

45.1 Relator for function space

locale lifting-syntax
begin
  notation rel-fun (infixr ===> 55)
  notation map-fun (infixr −−−> 55)
end

context begin
interpretation lifting-syntax.

lemma rel-funD2: assumes rel-fun A B f g and A x x shows B (f x) (g x)
  using assms by (rule rel-funD)

lemma rel-funE: assumes rel-fun A B f g and A x y obtains B (f x) (g y)
  using assms by (simp add: rel-fun-def)

lemmas rel-fun-eq = fun.rel-eq

lemma rel-fun-eq-rel:
  shows rel-fun (op =) R = (λf g. ∀x. R (f x) (g x))
    by (simp add: rel-fun-def)

45.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition Rel :: ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool
  where Rel r ≡ r

Handling of equality relations

definition is-equality :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where is-equality R ≡ R = (op =)

lemma is-equality-eq: is-equality (op =)
unfolding is-equality-def by simp

Reverse implication for monotonicity rules

definition rev-implies where
rev-implies x y ←→ (y → x)

Handling of meta-logic connectives

definition transfer-forall where
transfer-forall ≡ All

definition transfer-implies where
transfer-implies ≡ op →

definition transfer-bforall :: (′a ⇒ bool) ⇒ (′a ⇒ bool) ⇒ bool
where transfer-bforall ≡ (λP Q. ∀ x. P x → Q x)

lemma transfer-forall-eq: (⋀ x. P x) ≡ Trueprop (transfer-forall (λx. P x))
unfolding atomize-all transfer-forall-def ..

lemma transfer-implies-eq: (A =⇒ B) ≡ Trueprop (transfer-implies A B)
unfolding atomize-imp transfer-implies-def ..

lemma transfer-bforall-unfold:
Trueprop (transfer-bforall P (λx. Q x)) ≡ (∀ x. P x → Q x)
unfolding transfer-bforall-def atomize-imp atomize-all ..

lemma transfer-start: [P; Rel (op =) P Q] =⇒ Q
unfolding Rel-def by simp

lemma transfer-start′: [P; Rel (op −→) P Q] =⇒ Q
unfolding Rel-def by simp

lemma transfer-prover-start: [x = x'; Rel R x y] =⇒ Rel R x y
by simp

lemma untransfer-start: [Q; Rel (op =) P Q] =⇒ P
unfolding Rel-def by simp

lemma Rel-eq-refl: Rel (op =) x x
unfolding Rel-def ..

lemma Rel-app:
assumes Rel (A =⇒ B) f g and Rel A x y
shows Rel B (f x) (g y)
using assms unfolding Rel-def rel-fun-def by fast

lemma Rel-abs:
assumes ∃x y. Rel A x y =⇒ Rel B (f x) (g y)
s shows Rel (A =⇒ B) (λx. f x) (λy. g y)
using assms unfolding Rel-def rel-fun-def by fast

45.3 Predicates on relations, i.e. “class constraints”

definition left-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where left-total R ←→ (∀ x. ∃ y. R x y)

definition left-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where left-unique R ←→ (∀ x y z. R x z → R y z → x = y)

definition right-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where right-total R ←→ (∀ y. ∃ x. R x y)

definition right-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where right-unique R ←→ (∀ x y z. R x y → R x z → y = z)

definition bi-total :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where bi-total R ←→ (∀ x. ∃ y. R x y) ∧ (∀ y. ∃ x. R x y)

definition bi-unique :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where bi-unique R ←→ (∀ x y z. R x z → y = z) ∧ (∀ y z x. R x z → y = z)

lemma left-uniqueI: (∀ x y z. [ A x z; A y z ] → x = y) → left-unique A
unfolding left-unique-def by blast

lemma left-uniqueD: [[ left-unique A; A x z; A y z ]] → x = y
unfolding left-unique-def by blast

lemma left-totalI:
(∀ x. ∃ y. R x y) → left-total R
unfolding left-total-def by blast

lemma left-totalE:
assumes left-total R
obtains (∀ x. ∃ y. R x y)
using assms unfolding left-total-def by blast

lemma bi-uniqueDr: [[ bi-unique A; A x y; A x z ]] → y = z
by(simp add: bi-unique-def)

lemma bi-uniqueDr: [[ bi-unique A; A x y; A z y ]] → x = z
by(simp add: bi-unique-def)

lemma right-uniqueI: (∀ x y z. [ A x z; A y z ] → y = z) → right-unique A
unfolding right-unique-def by fast

lemma right-uniqueD: [[ right-unique A; A x y; A x z ]] → y = z
unfolding right-unique-def by fast

lemma right-total-alt-def2:
right-total $R \iff (R \Rightarrow op \Rightarrow op \Rightarrow) \forall \forall$
unfolding right-total-def rel-fun-def
apply (rule iffI, fast)
apply (rule allI)
apply (drule-tac $x = \lambda x. True$ in spec)
apply (drule-tac $x = \lambda y. \exists x. R x y$ in spec)
apply fast
done

lemma right-unique-alt-def2:
right-unique $R \iff (R \Rightarrow \Rightarrow R \Rightarrow \Rightarrow \Rightarrow \Rightarrow op \Rightarrow \Rightarrow \Rightarrow) (op =) (op =)$
unfolding right-unique-def rel-fun-def by auto

lemma bi-total-alt-def2:
bi-total $R \iff ((R \Rightarrow \Rightarrow \Rightarrow op =) \Rightarrow \Rightarrow \Rightarrow op =) \forall \forall$
unfolding bi-total-def rel-fun-def
apply (rule iffI, fast)
apply safe
apply (drule-tac $x = \lambda x. \exists y. R x y$ in spec)
apply (drule-tac $x = \lambda y. True$ in spec)
apply fast
apply (drule-tac $x = \lambda x. True$ in spec)
apply (drule-tac $x = \lambda y. \exists x. R x y$ in spec)
apply fast
done

lemma bi-unique-alt-def2:
bi-unique $R \iff (R \Rightarrow \Rightarrow \Rightarrow R \Rightarrow \Rightarrow \Rightarrow \Rightarrow \Rightarrow \Rightarrow) (op =) (op =)$
unfolding bi-unique-def rel-fun-def by auto

lemma [simp]:
shows left-unique-conversep: left-unique $A^{-1-1} \iff right-unique A$
and right-unique-conversep: right-unique $A^{-1-1} \iff left-unique A$
by(auto simp add: left-unique-def right-unique-def)

lemma [simp]:
sshows left-total-conversep: left-total $A^{-1-1} \iff right-total A$
and right-total-conversep: right-total $A^{-1-1} \iff left-total A$
by(simp-all add: left-total-def right-total-def)

lemma bi-unique-conversep [simp]: bi-unique $R^{-1-1} = bi-unique R$
by(auto simp add: bi-unique-def)

lemma bi-total-conversep [simp]: bi-total $R^{-1-1} = bi-total R$
by(auto simp add: bi-total-def)
lemma right-unique-alt-def: right-unique $R = (\text{conversep } R \circ \circ R \leq \text{op})$ unfolding right-unique-def by blast

lemma left-unique-alt-def: left-unique $R = (R \circ \circ (\text{conversep } R) \leq \text{op})$ unfolding left-unique-def by blast

lemma right-total-alt-def: right-total $R = (\text{conversep } R \circ \circ R \geq \text{op})$ unfolding right-total-def by blast

lemma left-total-alt-def: left-total $R = (R \circ \circ (\text{conversep } R) \geq \text{op})$ unfolding left-total-def by blast

lemma bi-total-alt-def: bi-total $A = (\text{left-total } A \land \text{right-total } A)$ unfolding bi-total-alt-def by blast

lemma bi-unique-alt-def: bi-unique $A = (\text{left-unique } A \land \text{right-unique } A)$ unfolding bi-unique-alt-def by blast

lemma bi-totalI: left-total $R \implies$ right-total $R \implies$ bi-total $R$ unfolding bi-total-alt-def ..

lemma bi-uniqueI: left-unique $R \implies$ right-unique $R \implies$ bi-unique $R$ unfolding bi-unique-alt-def ..

end

45.4 Equality restricted by a predicate

definition $\text{eq-onp} :: (\forall a \Rightarrow \text{bool}) \Rightarrow \forall a \Rightarrow \forall a \Rightarrow \text{bool}$
where $\text{eq-onp} R = (\lambda x y. R x \land x = y)$

lemma eq-onp-Grp: $\text{eq-onp } P = \text{BNF-Def.Grp (Collect } P \text{) id}$ unfolding eq-onp-def Grp-def by auto

lemma eq-onp-to-eq:
  assumes $\text{eq-onp } P \ x \ y$
  shows $x = y$
  using assms by (simp add: eq-onp-def)

lemma eq-onp-top-eq-eq: $\text{eq-onp } \top = \text{op}$
by (simp add: eq-onp-def)

lemma eq-onp-same-args:
  shows $\text{eq-onp } P \ x \ x = P \ x$
  using assms by (auto simp add: eq-onp-def)

lemma Ball-Collect: $\text{Ball } A \ P = (A \subseteq (\text{Collect } P))$
by auto

ML-file Tools/Transfer/transfer.ML
setup Transfer.setup
declare refl [transfer-rule]

hide-const (open) Rel

context begin
interpretation lifting-syntax .

Handling of domains

lemma Domainp-iff: Domainp T x ⇔ (∃ y. T x y)
  by auto

lemma Domainp-refl[transfer-domain-rule]:
  Domainp T = Domainp T ..

lemma Domainp-prod-fun-eq[relator-domain]:
  Domainp (op= ===> T) = (λf. ∀ x. (Domainp T) (f x))
  by (auto intro: choice simp: Domainp-iff rel-fun-def fun-eq-iff)

Properties are preserved by relation composition.

lemma OO-def: R OO S = (λx z. ∃ y. R x y ∧ S y z)
  by auto

lemma bi-total-OO: [bi-total A; bi-total B] ⇔ bi-total (A OO B)
  unfolding bi-total-def OO-def by fast

lemma bi-unique-OO: [bi-unique A; bi-unique B] ⇔ bi-unique (A OO B)
  unfolding bi-unique-def OO-def by blast

lemma right-total-OO:
  [right-total A; right-total B] ⇔ right-total (A OO B)
  unfolding right-total-def OO-def by fast

lemma right-unique-OO:
  [right-unique A; right-unique B] ⇔ right-unique (A OO B)
  unfolding right-unique-def OO-def by fast

lemma left-total-OO: left-total R ⇒ left-total S ⇒ left-total (R OO S)
  unfolding left-total-def OO-def by fast

lemma left-unique-OO: left-unique R ⇒ left-unique S ⇒ left-unique (R OO S)
  unfolding left-unique-def OO-def by blast

45.5 Properties of relators

lemma left-total-eq[transfer-rule]: left-total op=
  unfolding left-total-def by blast

lemma left-unique-eq[transfer-rule]: left-unique op=
THEORY "Transfer"

unfolding left-unique-def by blast

lemma right-total-eq [transfer-rule]: right-total op=
  unfolding right-total-def by simp

lemma right-unique-eq [transfer-rule]: right-unique op=
  unfolding right-unique-def by simp

lemma bi-total-eq[transfer-rule]: bi-total (op =)
  unfolding bi-total-def by simp

lemma bi-unique-eq[transfer-rule]: bi-unique (op =)
  unfolding bi-unique-def by simp

lemma left-total-fun[transfer-rule]:
  [left-unique A; left-total B] => left-total (A ===> B)
  unfolding left-total-def rel-fun-def
  apply (rule allI, rename-tac f)
  apply (rule-tac x=λy. SOME z. B (f (THE x. A x y)) z in ex1)
  apply clarify
  apply (subgoal-tac (THE x. A x y) = x, simp)
  apply (rule someI-ex)
  apply (simp)
  apply (rule the-equality)
  apply assumption
  apply (simp add: left-unique-def)
  done

lemma left-unique-fun[transfer-rule]:
  [left-total A; left-unique B] => left-unique (A ===> B)
  unfolding left-total-def rel-fun-def
  by (clarify, rule ext, fast)

lemma right-total-fun [transfer-rule]:
  [right-unique A; right-total B] => right-total (A ===> B)
  unfolding right-total-def rel-fun-def
  apply (rule allI, rename-tac g)
  apply (rule-tac x=λx. SOME z. B z (g (THE y. A x y)) in ex1)
  apply clarify
  apply (subgoal-tac (THE y. A x y) = y, simp)
  apply (rule someI-ex)
  apply (simp)
  apply (rule the-equality)
  apply assumption
  apply (simp add: right-unique-def)
  done

lemma right-unique-fun [transfer-rule]:
  [right-total A; right-unique B] => right-unique (A ===> B)
unfolding right-total-def right-unique-def rel-fun-def
by (clarify, rule ext, fast)

lemma bi-total-fun[transfer-rule]:
[bi-unique A; bi-total B] \implies bi-total (A ===> B)
unfolding bi-unique-alt-def bi-total-alt-def
by (blast intro: right-total-fun left-total-fun)

lemma bi-unique-fun[transfer-rule]:
[bi-total A; bi-unique B] \implies bi-unique (A ===> B)
unfolding bi-unique-alt-def bi-total-alt-def
by (blast intro: right-unique-fun left-unique-fun)

end

ML-file Tools/Transfer/transfer-bnf.ML

declare pred-fun-def [simp]
declare rel-fun-eq [relator-eq]

45.6 Transfer rules

context
begin
interpretation lifting-syntax .

lemma Domainp-forall-transfer [transfer-rule]:
assumes right-total A
shows ((A ===> op =) ===> op =)
  (transfer-bforall (Domainp A)) transfer-forall
using assms unfolding right-total-def
unfolding transfer-forall-def transfer-bforall-def rel-fun-def Domainp-iff
by fast

Transfer rules using implication instead of equality on booleans.

lemma transfer-forall-transfer [transfer-rule]:
bi-total A \implies ((A ===> op =) ===> op =) transfer-forall transfer-forall
right-total A \implies ((A ===> op =) ===> implies) transfer-forall transfer-forall
right-total A \implies ((A ===> implies) ===> implies) transfer-forall transfer-forall
bi-total A \implies ((A ===> op =) ===> rev-implies) transfer-forall transfer-forall
bi-total A \implies ((A ===> rev-implies) ===> rev-implies) transfer-forall transfer-forall
unfolding transfer-forall-def rev-implies-def rel-fun-def right-total-def bi-total-def
by fast

lemma transfer-implies-transfer [transfer-rule]:
(op = ====> op = ====> op = ) transfer-implies transfer-implies
(rev-implies ====> implies ====> implies ) transfer-implies transfer-implies
(rev-implies ====> op = ====> implies ) transfer-implies transfer-implies
(op = ====> implies ====> implies ) transfer-implies transfer-implies
THEORY “Transfer”

(\text{op} = \implies \implies \text{implies} \implies \implies ) \text{transfer-implies} \text{transfer-implies}
(\text{implies} \implies \implies \revimplies \implies \implies \implies ) \text{transfer-implies} \text{transfer-implies}
(\text{implies} \implies \implies \text{op} = \implies \implies \revimplies \implies \implies ) \text{transfer-implies} \text{transfer-implies}
(\text{op} = \implies \implies \implies \implies \implies \implies ) \text{transfer-implies} \text{transfer-implies}

\text{unfolding} \text{transfer-implies-def} \text{rev-implies-def} \text{rel-fun-def} \text{by} \text{auto}

\text{lemma} \text{eq-imp-transfer} [\text{transfer-rule}]:
right-unique \text{A} \implies (A \implies \implies \text{A} \implies \implies \text{op} =) \text{(op =)} \text{(op =)}
\text{unfolding} \text{right-unique-alt-def2} .

Transfer rules using equality.

\text{lemma} \text{left-unique-transfer} [\text{transfer-rule}]:
assumes right-total \text{A}
assumes right-total \text{B}
assumes bi-unique \text{A}
shows \((A \implies \implies \text{B} \implies \implies \text{op=} \implies \implies \text{implies}) \text{left-unique left-unique} \)
using \text{assms} \text{unfolding} \text{left-unique-def} \text{right-total-def} \text{bi-unique-def} \text{rel-fun-def} \text{by} \text{metis}

\text{lemma} \text{eq-transfer} [\text{transfer-rule}]:
assumes bi-unique \text{A}
shows \((A \implies \implies \text{A} \implies \implies \text{op =}) \text{(op =)} \text{(op =)} \)
using \text{assms} \text{unfolding} \text{bi-unique-def} \text{rel-fun-def} \text{by} \text{metis}

\text{lemma} \text{right-total-Ex-transfer}[\text{transfer-rule}]:
assumes right-total \text{A}
shows \((A \implies \implies \text{op=}) \implies \implies \text{op=} \implies \text{implies}) \text{Ex (Bex (Collect (Domainp A))) Ex}
using \text{assms} \text{unfolding} \text{right-total-def} \text{Bex-def} \text{rel-fun-def} \text{Domainp-iff} \text{[abs-def]} \text{by} \text{fast}

\text{lemma} \text{right-total-All-transfer}[\text{transfer-rule}]:
assumes right-total \text{A}
shows \((A \implies \implies \text{op =}) \implies \implies \text{op =}) \text{(Ball (Collect (Domainp A))) All}
using \text{assms} \text{unfolding} \text{right-total-def} \text{Ball-def} \text{rel-fun-def} \text{Domainp-iff} \text{[abs-def]} \text{by} \text{fast}

\text{lemma} \text{All-transfer} [\text{transfer-rule}]:
assumes bi-total \text{A}
shows \((A \implies \implies \text{op =}) \implies \implies \text{op =}) \text{All All}
using \text{assms} \text{unfolding} \text{bi-total-def} \text{rel-fun-def} \text{by} \text{fast}

\text{lemma} \text{Ex-transfer} [\text{transfer-rule}]:
assumes bi-total \text{A}
shows \((A \implies \implies \text{op =}) \implies \implies \text{op =}) \text{Ex Ex}
using \text{assms} \text{unfolding} \text{bi-total-def} \text{rel-fun-def} \text{by} \text{fast}

\text{lemma} \text{If-transfer} [\text{transfer-rule}]: \text{(op = \implies \implies \text{A \implies \implies \text{A} \implies \implies \text{A}) If If}
\text{unfolding} \text{rel-fun-def} \text{by} \text{simp}
lemma Let-transfer [transfer-rule]: \( (A \implies (A \implies B) \implies B) \) Let Let
unfolding rel-fun-def by simp

lemma id-transfer [transfer-rule]: \( (A \implies A) \) id id
unfolding rel-fun-def by simp

lemma comp-transfer [transfer-rule]:
\[
((B \implies C) \implies (A \implies B) \implies (A \implies C)) \circ \circ
\]
unfolding rel-fun-def by simp

lemma fun-upd-transfer [transfer-rule]:
\[
\text{assumes} \quad \text{[transfer-rule]: bi-unique } A
\]
\[
\text{shows} \quad ((A \implies B) \implies A \implies B) \implies (A \implies B) \text{ fun-upd fun-upd}
\]
unfolding fun-upd-def [abs-def] by transfer-prover

lemma case-nat-transfer [transfer-rule]:
\[
(A \implies (op = \implies A) \implies op = \implies A) \text{ case-nat case-nat}
\]
unfolding rel-fun-def by (simp split: nat.split)

lemma rec-nat-transfer [transfer-rule]:
\[
(A \implies (op = \implies A) \implies op = \implies A) \text{ rec-nat rec-nat}
\]
unfolding rel-fun-def by (clarsimp, rename-tac n, induct-tac n, simp-all)

lemma funpow-transfer [transfer-rule]:
\[
(op = \implies (A \implies A) \implies (A \implies A)) \text{ compow compow}
\]
unfolding funpow-def by transfer-prover

lemma mono-transfer[transfer-rule]:
\[
\text{assumes} \quad \text{[transfer-rule]: bi-total } A
\]
\[
\text{assumes} \quad \text{[transfer-rule]: } \text{op} \leq \text{op} \leq \text{op}
\]
\[
\text{assumes} \quad \text{[transfer-rule]: } \text{op} \leq \text{op} \leq \text{op}
\]
\[
\text{shows} \quad ((A \implies B) \implies op = \implies \text{mono mono}
\]
unfolding mono-def[abs-def] by transfer-prover

lemma right-total-relcompp-transfer[transfer-rule]:
\[
\text{assumes} \quad \text{[transfer-rule]: right-total } B
\]
\[
\text{shows} \quad ((A \implies B \implies op =) \implies (B \implies C \implies op =) \implies \text{Ar S x z. } \exists y \in \text{Collect } (\text{Domainp } B). R x y \land S y z \text{ op OO}
\]
unfolding OO-def[abs-def] by transfer-prover

lemma relcompp-transfer[transfer-rule]:
\[
\text{assumes} \quad \text{[transfer-rule]: bi-total } B
\]
\[
\text{shows} \quad ((A \implies B \implies op =) \implies (B \implies C \implies op =) \implies \text{op OO op OO}
\]
unfolding OO-def[abs-def] by transfer-prover

lemma right-total-Domainp-transfer[transfer-rule]:
assumes [transfer-rule]: right-total B
shows \((A \implies B) \implies \lambda x \exists y \in \text{Collect}(\text{Domain}p B). T x y : \text{Domain}p\)
apply (subst (2) Domainp-iff [abs-def]) by transfer-prover

lemma Domainp-transfer [transfer-rule]:
assumes [transfer-rule]: bi-total B
shows \((A \implies B) \implies \lambda x \exists y \in \text{Collect}(\text{Domain}p B). T x y : \text{Domain}p\)
unfolding Domainp-iff [abs-def] by transfer-prover

lemma reflp-transfer [transfer-rule]:
\begin{align*}
\text{bi-total } A &\implies ((A \implies A \implies \text{op} =) \implies \text{reflp} \text{ reflp}) \\
\text{right-total } A &\implies ((A \implies A \implies \text{implies}) \implies \text{reflp} \text{ reflp}) \\
\text{bi-total } A &\implies ((A \implies A \implies \text{rev-implies}) \implies \text{reflp} \text{ reflp}) \\
\text{bi-total } A &\implies ((A \implies A \implies \text{op} =) \implies \text{rev-implies} \text{ reflp} \text{ reflp})
\end{align*}
using assms unfolding reflp-def [abs-def] rev-implies-def bi-total-def right-total-def rel-fun-def
by fast+

lemma right-unique-transfer [transfer-rule]:
assumes [transfer-rule]: right-total A
assumes [transfer-rule]: right-total B
assumes [transfer-rule]: bi-unique B
shows \((A \implies B) \implies \text{right-unique} \text{ right-unique})
using assms unfolding right-unique-def [abs-def] right-total-def bi-unique-def rel-fun-def
by metis

lemma rel-fun-eq-eq-onp: \((\text{op}= \implies \text{eq-onp } P) = \text{eq-onp } (\lambda f. \forall x. P(f x))\)
unfolding eq-onp-def rel-fun-def by auto

lemma rel-fun-eq-onp-rel:
shows \((\text{eq-onp } R) \implies S) = (\lambda f. \forall x. R x \implies S(f x)(g x))\)
by (auto simp add: eq-onp-def rel-fun-def)

lemma eq-onp-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows \((A \implies \text{op} =) \implies A \implies A \implies \text{op=} \text{ eq-onp eq-onp}\)
unfolding eq-onp-def [abs-def] by transfer-prover

lemma rtranclp-parametric [transfer-rule]:
assumes bi-unique A bi-total A
shows \((A \implies A \implies \text{op} =) \implies A \implies A \implies \text{op} = \text{ rtranclp rtranclp}\)
proof (rule rel-funI iffI)+
  fix R :: 'a => 'a => bool and R' x y x' y'
  assume R: \((A \implies A \implies \text{op} =) \implies R R' \text{ and } A x x'\)
  { assume R** x y A y y'
thus $R^{**} x' y'$

proof (induction arbitrary: $y'$)

\begin{itemize}
  \item case base
    \begin{itemize}
      \item with $\langle$ bi-unique $A$ $\rangle \langle A x x' \rangle$ have $x' = y'$ by (rule bi-uniqueDr)
    \end{itemize}
  \item thus ?case by simp
\end{itemize}

next

\begin{itemize}
  \item case (step $y z z'$)
    \begin{itemize}
      \item from $\langle$ bi-total $A$ $\rangle$ obtain $y'$ where $A y y'$ unfolding bi-total-def by blast
      \item hence $R^{**} x' y' y$ by (rule step.IH)
      \item moreover from $R (A y y') (A z z') (R y z)$
      \item have $R' y' z' (A z z')$ by (auto dest: rel-funD)
      \item ultimately show ?case ..
      \item qed
    \end{itemize}
  \end{itemize}

next

\begin{itemize}
  \item assume $R^{**} x' y' A y y'$
  \item thus $R^{**} x y$
\end{itemize}

proof (induction arbitrary: $y$)

\begin{itemize}
  \item case base
    \begin{itemize}
      \item with $\langle$ bi-unique $A$ $\rangle \langle A x x' \rangle$ have $x = y$ by (rule bi-uniqueDr)
    \end{itemize}
  \item thus ?case by simp
\end{itemize}

next

\begin{itemize}
  \item case (step $y' z' z$)
    \begin{itemize}
      \item from $\langle$ bi-total $A$ $\rangle$ obtain $y$ where $A y y'$ unfolding bi-total-def by blast
      \item hence $R^{**} x y y'$ by (rule step.IH)
      \item moreover from $R (A y y') (A z z') (R y z)$
      \item have $R y z (A z z')$ by (auto dest: rel-funD)
      \item ultimately show ?case ..
      \item qed
    \end{itemize}
  \end{itemize}

}\}

qed

end

\end{itemize}

46 Lifting: Lifting package

\begin{itemize}
  \item theory Lifting
  \item imports Equiv-Relations Transfer
  \item keywords parametric and
    \begin{itemize}
      \item print-quot-maps print-quotients :: diag and
      \item lift-definition :: thy-goal and
      \item setup-lifting lifting-forget lifting-update :: thy-decl
    \end{itemize}
  \item begin
\end{itemize}

46.1 Function map

context
begin
interpretation lifting-syntax .

lemma map-fun-id:
  \((\text{id} \mapsto \text{id}) = \text{id}\)
  by (simp add: fun-eq-iff)

46.2 Quotient Predicate

definition
  Quotient \(R\) Abs Rep T ←→
  \((\forall a. \text{Abs (Rep a) = a}) \land\)
  \((\forall a. \text{R (Rep a) (Rep a)}) \land\)
  \((\forall r s. \text{R r s} \leftrightarrow \text{R r} \land \text{R s} \land \text{Abs r = Abs s}) \land\)
  \(T = (\lambda x y. \text{R x x} \land \text{Abs x = y})\)

lemma QuotientI:
  assumes \(\forall a. \text{Abs (Rep a) = a}\)
  and \(\forall a. \text{R (Rep a) (Rep a)}\)
  and \(\forall r s. \text{R r s} \leftrightarrow \text{R r} \land \text{R s} \land \text{Abs r = Abs s}\)
  and \(T = (\lambda x y. \text{R x x} \land \text{Abs x = y})\)
  shows Quotient \(R\) Abs Rep T
  using \(\text{assms}\) unfolding Quotient-def by blast

context
  fixes \(R\) Abs Rep T
  assumes \(a\): Quotient \(R\) Abs Rep T
begin

lemma Quotient-abs-rep: \(\text{Abs (Rep a) = a}\)
  using \(a\) unfolding Quotient-def
  by simp

lemma Quotient-rep-reflp: \(\text{R (Rep a) (Rep a)}\)
  using \(a\) unfolding Quotient-def
  by blast

lemma Quotient-rel:
  \(\text{R r r} \land \text{R s s} \land \text{Abs r = Abs s} \leftrightarrow \text{R r s}\) — orientation does not loop on rewriting
  using \(a\) unfolding Quotient-def
  by blast

lemma Quotient-cr-rel: \(T = (\lambda x y. \text{R x x} \land \text{Abs x = y})\)
  using \(a\) unfolding Quotient-def
  by blast

lemma Quotient-refl1: \(\text{R r s} \rightarrow \text{R r r}\)
  using \(a\) unfolding Quotient-def
by fast

lemma Quotient-refl2: $R \, r \, s \implies R \, s \, s$

  using a unfolding Quotient-def
  by fast

lemma Quotient-rel-rep: $R \, (\operatorname{Rep} \, a) \, (\operatorname{Rep} \, b) \iff a = b$

  using a unfolding Quotient-def
  by metis

lemma Quotient-rep-abs: $R \, r \, r \implies R \, (\operatorname{Rep} \, (\operatorname{Abs} \, r)) \, r$

  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-eq: $R \, t \, t \implies R \leq \operatorname{op} = \implies \operatorname{Rep} \, (\operatorname{Abs} \, t) = t$

  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-fold-unmap:

  assumes $x' \equiv \operatorname{Abs} \, x$ and $R \, x \, x$ and $\operatorname{Rep} \, x' \equiv \operatorname{Rep'} \, x'$

  shows $R \, (\operatorname{Rep'} \, x') \, x$

  proof −
    have $R \, (\operatorname{Rep} \, x') \, x$ using assms(1-2) Quotient-rep-abs by auto
    then show ?thesis using assms(3) by simp
  qed

lemma Quotient-Rep-eq:

  assumes $x' \equiv \operatorname{Abs} \, x$

  shows $\operatorname{Rep} \, x' \equiv \operatorname{Rep} \, x'$

  by simp

lemma Quotient-rel-abs: $R \, r \, s \implies \operatorname{Abs} \, r = \operatorname{Abs} \, s$

  using a unfolding Quotient-def
  by blast

lemma Quotient-rel-abs2:

  assumes $R \, (\operatorname{Rep} \, x) \, y$

  shows $x = \operatorname{Abs} \, y$

  proof −
    from assms have $\operatorname{Abs} \, (\operatorname{Rep} \, x) = \operatorname{Abs} \, y$ by (auto intro: Quotient-rel-abs)
    then show ?thesis using assms(1) by (simp add: Quotient-abs-rep)
  qed

lemma Quotient-symp: symp $R$

  using a unfolding Quotient-def using sympI by (metis (full-types))

lemma Quotient-transp: transp $R$

  using a unfolding Quotient-def using transpI by (metis (full-types))
lemma Quotient-part-eqv: part-eqv R
by (metis Quotient-rep-reflp Quotient-symp Quotient-transp part-eqvI)

end

lemma identity-quotient: Quotient (op =) id id (op =)
unfolding Quotient-def by simp

TODO: Use one of these alternatives as the real definition.

lemma Quotient-alt-def: Quotient R Abs Rep T ←→
(∀ a b. T a b → Abs a = b) ∧
(∀ b. T (Rep b) b) ∧
(∀ x y. R x y ↔ T x (Abs x) ∧ T y (Abs y) ∧ Abs x = Abs y)
apply safe
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (simp (no-asm-use) only: Quotient-def, fast)
apply (rule QuotientI)
apply simp
apply metis
apply simp
apply (rule ext, rule ext, metis)
done

lemma Quotient-alt-def2: Quotient R Abs Rep T ←→
(∀ a b. T a b → Abs a = b) ∧
(∀ b. T (Rep b) b) ∧
(∀ x y. R x y ↔ T x (Abs y) ∧ T y (Abs x))
unfolding Quotient-alt-def by (safe, metis+)

lemma Quotient-alt-def3: Quotient R Abs Rep T ←→
(∀ a b. T a b → Abs a = b) ∧ (∀ b. T (Rep b) b) ∧
(∀ x y. R x y ↔ (∃ z. T x z ∧ T y z))
unfolding Quotient-alt-def2 by (safe, metis+)

lemma Quotient-alt-def4: Quotient R Abs Rep T ←→
(∀ a b. T a b → Abs a = b) ∧ (∀ b. T (Rep b) b) ∧ R = T OO conversep T
unfolding Quotient-alt-def3 fun-eq-iff by auto

lemma Quotient-alt-def5: Quotient R Abs Rep T ←→
T ≤ BNF-Def . Grp UNIV Abs ∧ BNF-Def . Grp UNIV Rep ≤ T⁻¹⁻¹ ∧ R = T
THEORY “Lifting”

\[ \text{theory “Lifting”} \]

Orient-1

unfolding Quotient-alt-def4 Grp-def by blast

lemma fun-quotient:
assumes 1: Quotient R1 abs1 rep1 T1
assumes 2: Quotient R2 abs2 rep2 T2
shows Quotient (R1 ===> R2) (rep1 ===> abs2) (abs1 ===> rep2) (T1 ===> T2)
using assms unfolding Quotient-alt-def2
unfolding rel-fun-def fun-eq-iff map-fun-apply
by (safe, metis+)

lemma apply-rsp:
fixes f g::'a => 'c
assumes q: Quotient R1 Abs1 Rep1 T1
and a: (R1 ===> R2) f g R1 x y
shows R2 (f x) (g y)
using a by (auto elim: rel-funE)

lemma apply-rsp':
assumes a: (R1 ===> R2) f g R1 x y
shows R2 (f x) (g y)
using a by (auto elim: rel-funE)

lemma apply-rsp'":
assumes Quotient R Abs Rep T
and (R ===> S) f f
shows S (f (Rep x)) (f (Rep x))
proof –
from assms(1) have R (Rep x) (Rep x) by (rule Quotient-rep-reflp)
then show \"thesis using assms(2) by (auto intro: apply-rsp')
qed

46.3 Quotient composition

lemma Quotient-compose:
assumes 1: Quotient R1 Abs1 Rep1 T1
assumes 2: Quotient R2 Abs2 Rep2 T2
shows Quotient (T1 OO R2 OO conversep T1) (Abs2 o Abs1) (Rep1 o Rep2)
(T1 OO T2)
using assms unfolding Quotient-alt-def4 by fastforce

lemma equivp-reflp2:
equivp R ===> reflp R
by (erule equivpE)

46.4 Respects predicate

definition Respects :: (‘a => ‘a => bool) => ‘a set
where Respects R = {x. R x x}
**THEORY “Lifting”**

**Lemma in-respects**: \(x \in \text{Respects } R \iff R x x\)
- **unfolding** \(\text{Respects-def} \text{ by simp}\)

**Lemma UNIV-typedef-to-Quotient**:
- **assumes** \(\text{type-definition } \text{Rep } \text{Abs } \text{UNIV}\)
- **and** \(T\)-def: \(T \equiv (\lambda x. y. x = \text{Rep } y)\)
- **shows** \(\text{Quotient } (\text{op } =) \text{ Abs Rep } T\)

**proof** —
- **interpret** \(\text{type-definition } \text{Rep } \text{Abs } \text{UNIV} \text{ by fact}\)
- **from** \(\text{Abs-inject } \text{Rep-inverse } \text{Abs-inverse } T\)-def **show** ?thesis
  - (fastforce intro!: QuotientI fun-eq-iff)

**Qed**

**Lemma UNIV-typedef-to-equivp**:
- **fixes** \(\text{Abs} :: 'a \Rightarrow 'b\)
- **and** \(\text{Rep} :: 'b \Rightarrow 'a\)
- **assumes** \(\text{type-definition } \text{Rep } \text{Abs } (\text{UNIV}::'a \text{ set})\)
- **shows** \(\text{equivp } (\text{op}::='a\Rightarrow'a\Rightarrow\text{bool})\)
  - by (rule identity-equivp)

**Lemma typedef-to-Quotient**:
- **assumes** \(\text{type-definition } \text{Rep } \text{Abs } S\)
- **and** \(T\)-def: \(T \equiv (\lambda x. y. x = \text{Rep } y)\)
- **shows** \(\text{Quotient } (\text{eq-onp } (\lambda x. x \in S)) \text{ Abs Rep } T\)

**proof** —
- **interpret** \(\text{type-definition } \text{Rep } \text{Abs } S \text{ by fact}\)
- **from** \(\text{Rep Abs-inject } \text{Rep-inverse } \text{Abs-inverse } T\)-def **show** ?thesis
  - (auto intro!: QuotientI simp: eq-onp-def fun-eq-iff)

**Qed**

**Lemma typedef-to-part-equivp**:
- **assumes** \(\text{type-definition } \text{Rep } \text{Abs } S\)
- **shows** \(\text{part-equivp } (\text{eq-onp } (\lambda x. x \in S))\)

**proof** —
- **intro** part-equivpI
- **interpret** \(\text{type-definition } \text{Rep } \text{Abs } S \text{ by fact}\)
- **show** \(\exists x. \text{eq-onp } (\lambda x. x \in S) x x\)
  - using Rep by (auto simp: eq-onp-def)
- **next**
  - **show** \(\text{symp } (\text{eq-onp } (\lambda x. x \in S))\)
    - by (auto intro: sympI simp: eq-onp-def)
- **next**
  - **show** \(\text{transp } (\text{eq-onp } (\lambda x. x \in S))\)
    - by (auto intro: transpI simp: eq-onp-def)

**Qed**

**Lemma open-typedef-to-Quotient**:
- **assumes** \(\text{type-definition } \text{Rep } \text{Abs } \{x. P x\}\)
- **and** \(T\)-def: \(T \equiv (\lambda x. y. x = \text{Rep } y)\)
- **shows** \(\text{Quotient } (\text{eq-onp } P) \text{ Abs Rep } T\)

**using** typedef-to-Quotient [OF assms] **by simp**
THEORY “Lifting”

lemma open-typedef-to-part-eqivp:
  assumes type-definition Rep Abs \{ x. P x \}
  shows part-eqivp (eq-onp P)
  using typedef-to-part-eqivp \[ OF assms \] by simp

Generating transfer rules for quotients.

context
  fixes R Abs Rep T
  assumes I: Quotient R Abs Rep T
begin

lemma Quotient-right-unique: right-unique T
  using I unfolding Quotient-alt-def right-unique-def by metis

lemma Quotient-right-total: right-total T
  using I unfolding Quotient-alt-def right-total-def by metis

lemma Quotient-rel-eq-transfer: (T ===> T ===> op =) R (op =)
  using I unfolding Quotient-alt-def rel-fun-def by simp

lemma Quotient-abs-induct:
  assumes \A y. R y y \implies P (Abs y) shows P x
  using I assms unfolding Quotient-def by metis

end

Generating transfer rules for total quotients.

context
  fixes R Abs Rep T
  assumes I: Quotient R Abs Rep T and 2: reflp R
begin

lemma Quotient-left-total: left-total T
  using I 2 unfolding Quotient-alt-def left-total-def reflp-def by auto

lemma Quotient-bi-total: bi-total T
  using I 2 unfolding Quotient-alt-def bi-total-def reflp-def by auto

lemma Quotient-id-abs-transfer: (op ===> T) (\lambda x. x) Abs
  using I 2 unfolding Quotient-alt-def reflp-def rel-fun-def by simp

lemma Quotient-total-abs-induct: (\A y. P (Abs y)) \implies P x
  using I 2 assms unfolding Quotient-alt-def reflp-def by metis

lemma Quotient-total-abs-eq-iff: Abs x = Abs y \iff R x y
  using Quotient-rel \[ OF I \] 2 unfolding reflp-def by simp

end

Generating transfer rules for a type defined with typedef.
context
fixes Rep Abs A T
assumes type: type-definition Rep Abs A
assumes T-def: T ≡ (λ(x::'a) (y::'b). x = Rep y)

begin

lemma typedef-left-unique: left-unique T
  unfolding left-unique-def T-def
  by (simp add: type-definition. Rep-inject [OF type])

lemma typedef-bi-unique: bi-unique T
  unfolding bi-unique-def T-def
  by (simp add: type-definition. Rep-inject [OF type])

lemma typedef-right-unique: right-unique T
  using T-def type Quotient-right-unique typedef-to-Quotient
  by blast

lemma typedef-right-total: right-total T
  using T-def type Quotient-right-total typedef-to-Quotient
  by blast

lemma typedef-rep-transfer: (T === op =) (λx. x) Rep
  unfolding rel-fun-def T-def by simp

end

Generating the correspondence rule for a constant defined with lift-definition.

lemma Quotient-to-transfer:
  assumes Quotient R Abs Rep T and R c c' and c' ≡ Abs c
  shows T c c'
  using assms by (auto dest: Quotient-cr-rel)

Proving reflexivity

lemma Quotient-to-left-total:
  assumes q: Quotient R Abs Rep T
  and r-R: reflp R
  shows left-total T
  using r-R Quotient-cr-rel[OF q] unfolding left-total-def by (auto elim: reflpE)

lemma Quotient-composition-ge-eq:
  assumes left-total T
  assumes R ≥ op=
  shows (T OO R OO T⁻¹⁻¹) ≥ op=
  using assms unfolding left-total-def by fast

lemma Quotient-composition-le-eq:
assumes left-unique $T$
assumes $R \leq \text{op}=$
shows $(T \text{ OO } R \text{ OO } T^{-1}) \leq \text{op}=$
using assms unfolding left-unique-def by blast

lemma eq-onp-le-eq:
eq-onp $P \leq \text{op}=$ unfolding eq-onp-def by blast

lemma reflp-ge-eq:
$\text{reflp } R \implies R \geq \text{op}=$ unfolding reflp-def by blast

lemma ge-eq-refl:
$R \geq \text{op}=$ \implies $R x x$ by blast

Proving a parametrized correspondence relation

definition $\text{POS} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$ where
$\text{POS } A \text{ B } \equiv A \leq B$

definition $\text{NEG} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$ where
$\text{NEG } A \text{ B } \equiv B \leq A$

lemma pos-OO-eq:
shows $\text{POS } (A \text{ OO op } =) A$
unfolding $\text{POS-def OO-def}$ by blast

lemma pos-eq-OO:
shows $\text{POS } (\text{op } = \text{ OO } A) A$
unfolding $\text{POS-def OO-def}$ by blast

lemma neg-OO-eq:
shows $\text{NEG } (A \text{ OO op } =) A$
unfolding $\text{NEG-def OO-def}$ by auto

lemma neg-eq-OO:
shows $\text{NEG } (\text{op } = \text{ OO } A) A$
unfolding $\text{NEG-def OO-def}$ by auto

lemma $\text{POS-trans}$:
assumes $\text{POS } A \text{ B}$
assumes $\text{POS } B \text{ C}$
shows $\text{POS } A \text{ C}$
using assms unfolding $\text{POS-def}$ by auto

lemma $\text{NEG-trans}$:
assumes $\text{NEG } A \text{ B}$
assumes $\text{NEG } B \text{ C}$
shows $\text{NEG } A \text{ C}$
using assms unfolding $\text{NEG-def}$ by auto
lemma \text{POS-NEG}:
\begin{align*}
\text{POS} \ A \ B & \equiv \text{NEG} \ B \ A \\
\text{unfolding} & \quad \text{POS-def NEG-def by auto}
\end{align*}

lemma \text{NEG-POS}:
\begin{align*}
\text{NEG} \ A \ B & \equiv \text{POS} \ B \ A \\
\text{unfolding} & \quad \text{POS-def NEG-def by auto}
\end{align*}

lemma \text{POS-per-rule}:
\begin{align*}
\text{assumes} & \quad \text{POS} \ (A \ \text{OO} \ B) \ C \\
\text{shows} & \quad \text{POS} \ (A \ \text{OO} \ B \ \text{OO} \ X) \ (C \ \text{OO} \ X) \\
\text{using} & \quad \text{assms unfolding POS-def OO-def by blast}
\end{align*}

lemma \text{NEG-per-rule}:
\begin{align*}
\text{assumes} & \quad \text{NEG} \ (A \ \text{OO} \ B) \ C \\
\text{shows} & \quad \text{NEG} \ (A \ \text{OO} \ B \ \text{OO} \ X) \ (C \ \text{OO} \ X) \\
\text{using} & \quad \text{assms unfolding NEG-def OO-def by blast}
\end{align*}

lemma \text{POS-apply}:
\begin{align*}
\text{assumes} & \quad \text{POS} \ R \ R' \\
\text{assumes} & \quad R \ f \ g \\
\text{shows} & \quad R' \ f \ g \\
\text{using} & \quad \text{assms unfolding POS-def by auto}
\end{align*}

Proving a parametrized correspondence relation

lemma \text{fun-mono}:
\begin{align*}
\text{assumes} & \quad A \ \geq \ C \\
\text{assumes} & \quad B \ \leq \ D \\
\text{shows} & \quad (A \ \Longrightarrow \ B) \ \leq \ (C \ \Longrightarrow \ D) \\
\text{using} & \quad \text{assms unfolding rel-fun-def by blast}
\end{align*}

lemma \text{pos-fun-distr}:
\begin{align*}
\text{assumes} & \quad \text{left-unique} \ R \Rightarrow \text{right-total} \ R \Rightarrow \forall \ x. \ \exists! y. \ R \ x \ y \\
\text{using} & \quad \text{assms unfolding fun-def right-total-def by blast}
\end{align*}

lemma \text{functional-relation}: \text{right-unique} \ R \Rightarrow \text{left-total} \ R \Rightarrow \forall \ x. \ \exists! y. \ R \ x \ y
\text{unfolding} \quad \text{right-unique-def left-total-def by blast}

lemma \text{functional-converse-relation}: \text{left-unique} \ R \Rightarrow \text{right-total} \ R \Rightarrow \forall \ y. \ \exists! x. \ R \ x \ y
\text{unfolding} \quad \text{left-unique-def right-total-def by blast}

lemma \text{neg-fun-distr}:\text{1}: \text{left-unique} \ R \ \text{right-total} \ R
\text{assumes} \text{2}: \text{right-unique} \ R' \ \text{left-total} \ R'
\text{shows} \quad (R \ \text{OO} \ R' \ \Longrightarrow \ S \ \text{OO} \ S') \ \leq \ ((R \ \Longrightarrow \ S) \ \text{OO} \ (R' \ \Longrightarrow \ S'))
\text{using} \quad \text{functional-relation[OF 2]} \ \text{functional-converse-relation[OF 1]}
\text{unfolding} \quad \text{rel-fun-def OO-def}
\text{apply} \quad \text{clarify}
apply (subst all-comm)
apply (subst all-conj-distrib[symmetric])
apply (intro choice)
by metis

lemma neg-fun-distr2:
assumes 1: right-unique R' left-total R'
assumes 2: left-unique S' right-total S'
shows \((R \OO R' \Longrightarrow S \OO S') \leq ((R \Longrightarrow S) \OO (R' \Longrightarrow S'))\)
unfolding rel-fun-def OO-def
apply clarify
apply (subst all-comm)
apply (subst all-conj-distrib[symmetric])
apply (intro choice)
by metis

46.5 Domains

lemma composed-equiv-rel-eq-onp:
assumes left-unique R
assumes \((R \Longrightarrow op =) P P'\)
assumes Domainp R = P''
shows \((R \OO op = \OO P' OO R^{-1}^{-1}) = eq-onp (inf P'' P)\)
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
rel-fun-def eq-onp-def
fun-eq-iff by blast

lemma composed-equiv-rel-eq-eq-onp:
assumes left-unique R
assumes Domainp R = P
shows \((R \OO op = \OO P' OO R^{-1}^{-1}) = eq-onp P\)
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
eq-onp-def
eq-eq-iff is-equality-def by metis

lemma pcr-Domainp-par-left-total:
assumes Domainp B = P
assumes left-total A
assumes \((A \Longrightarrow op =) P' P\)
shows Domainp (A \OO B) = P'
using assms
unfolding Domainp-iff[abs-def] OO-def bi-unique-def left-total-def rel-fun-def
by (fast intro: fun-eq-iff)

lemma pcr-Domainp-par:
assumes Domainp B = P2
assumes Domainp A = P1
assumes \((A \Longrightarrow op =) P2' P2\)
THEORY "Lifting"

shows Domainp (A OO B) = (inf P1 P2)
using assms unfolding rel-fun-def Domainp-iff[abs-def] OO-def
by (fast intro: fun-eq-iff)

definition rel-pred-comp :: ('a => 'b => bool) => ('b => bool) => 'a => bool
where rel-pred-comp R P ≡ λx. ∃y. R x y ∧ P y

lemma per-Domainp:
assumes Domainp B = P
shows Domainp (A OO B) = (λx. ∃y. A x y ∧ P y)
using assms by blast

lemma per-Domainp-total:
  assumes left-total B
  assumes Domainp A = P
  shows Domainp (A OO B) = P
using assms unfolding left-total-def
by fast

lemma Quotient-to-Domainp:
  assumes Quotient R Abs Rep T
  shows Domainp T = (λx. R x x)
by (simp add: Domainp-iff[abs-def] Quotient-cr-rel[OF assms])

lemma eq-onp-to-Domainp:
  assumes Quotient (eq-onp P) Abs Rep T
  shows Domainp T = P
by (simp add: eq-onp-def Domainp-iff[abs-def] Quotient-cr-rel[OF assms])

end

46.6 ML setup

ML-file Tools/Lifting/lifting-utit.ML

ML-file Tools/Lifting/lifting-info.ML
setup Lifting-Info.setup

declare fun-quotient[quot-map]
declare fun-mono[relator-mono]
lemmas [relator-distr] = pos-fun-distr neg-fun-distr1 neg-fun-distr2

ML-file Tools/Lifting/lifting-bnf.ML

ML-file Tools/Lifting/lifting-term.ML

ML-file Tools/Lifting/lifting-def.ML
47 Quotient: Definition of Quotient Types

Definition of Quotient Types

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation
  rel-conj :: 
  \lambda (\forall a \cdot R (\Re a) (\Re a)) \land 
  \lambda (\forall r s \cdot R r r \land R s s \land Abs r = Abs s)

lemma Quotient3I:
  assumes \lambda a. Abs (\Re a) = a 
  \land \lambda a. R (\Re a) (\Re a) 
  \land \lambda r s. R r r \land R s s \land Abs r = Abs s
  shows Quotient3 R Abs Rep
  using assms unfolding Quotient3-def by blast
assumes a: Quotient3 R Abs Rep

begin

lemma Quotient3-abs-rep:
  Abs (Rep a) = a
  using a
  unfolding Quotient3-def
  by simp

lemma Quotient3-rep-reflp:
  R (Rep a) (Rep a)
  using a
  unfolding Quotient3-def
  by blast

lemma Quotient3-rel:
  R r r ∧ R s s ∧ Abs r = Abs s <-> R r s — orientation does not loop on rewriting
  using a
  unfolding Quotient3-def
  by blast

lemma Quotient3-refl1:
  R r s ==> R r r
  using a unfolding Quotient3-def
  by fast

lemma Quotient3-refl2:
  R r s ==> R s s
  using a unfolding Quotient3-def
  by fast

lemma Quotient3-rel-rep:
  R (Rep a) (Rep b) <-> a = b
  using a
  unfolding Quotient3-def
  by metis

lemma Quotient3-rep-abs:
  R r r ==> R (Rep (Abs r)) r
  using a unfolding Quotient3-def
  by blast

lemma Quotient3-rel-abs:
  R r s ==> Abs r = Abs s
  using a unfolding Quotient3-def
  by blast

lemma Quotient3-symp:
symp R
using a unfolding Quotient3-def using sympI by metis

lemma Quotient3-transp:
transp R
using a unfolding Quotient3-def using transpI by (metis (full-types))

lemma Quotient3-part-eqivp:
part-eqivp R
by (metis Quotient3-rep-reflp Quotient3-symp Quotient3-transp part-eqivpI)

lemma abs-o-rep:
Abs o Rep = id
unfolding fun-eq-iff
by (simp add: Quotient3-abs-rep)

lemma equals-rsp:
assumes b: R xa xb R ya yb
shows R xa ya = R xb yb
using b Quotient3-symp Quotient3-transp
by (blast elim: sympE transpE)

lemma rep-abs-rsp:
assumes b: R x1 x2
shows R x1 (Rep (Abs x2))
using b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp
by metis

lemma rep-abs-rsp-left:
assumes b: R x1 x2
shows R (Rep (Abs x1)) x2
using b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp
by metis

end

lemma identity-quotient3:
Quotient3 (op =) id id
unfolding Quotient3-def id-def
by blast

lemma fun-quotient3:
assumes q1: Quotient3 R1 abs1 rep1
and q2: Quotient3 R2 abs2 rep2
shows Quotient3 (R1 ===> R2) (rep1 ===> reps2) (abs1 ===> rep2)
proof -
have \( \forall a. (\text{rep1} ===> \text{abs2}) ((\text{abs1} ===> \text{rep2}) a) = a \)
  using \( q1 \ q2 \) by (simp add: Quotient3-def fun-eq-iff)
moreover
have \( \land a. (R1 \implies R2) ((\text{abs}1 \implies \text{rep}2) a) ((\text{abs}1 \implies \text{rep}2) a) \)
by (rule rel-funI)
(insert q1 q2 Quotient3-rel-abs[of R1 \text{abs}1 \text{rep}1] Quotient3-rel-rep[of R2 \text{abs}2 \text{rep}2],
simp (no-asmp) add: Quotient3-def, simp)

moreover
\{
fix \( r \) \( s \)
have \( (R1 \implies R2) \) \( r \) \( s = ((R1 \implies R2) \) \( r \) \( r \) \( \land (R1 \implies R2) \) \( s \) \( s \) \( \land \)
\( (\text{rep}1 \implies \text{abs}2) \) \( r = (\text{rep}1 \implies \text{abs}2) \) \( s \)
\)
proof –

have \( (R1 \implies R2) \) \( r \) \( s \implies (R1 \implies R2) \) \( r \) \( r \) unfolding rel-fun-def
using Quotient3-part-equivp[of q1] Quotient3-part-equivp[of q2]
by (metis (full-types) part-equivp-def)

moreover have \( (R1 \implies R2) \) \( r \) \( s \implies (R1 \implies R2) \) \( s \) \( s \) unfolding rel-fun-def
using Quotient3-part-equivp[of q1] Quotient3-part-equivp[of q2]
by (metis (full-types) part-equivp-def)

moreover have \( (R1 \implies R2) \) \( r \) \( s \implies (\text{rep}1 \implies \text{abs}2) \) \( r \) \( = (\text{rep}1 \implies \text{abs}2) \) \( s \)
apply (auto simp add: rel-fun-def fun-eq-iff) using q1 q2 unfolding Quotient3-def
by metis

moreover have \( ((R1 \implies R2) \) \( r \) \( r \) \( \land (R1 \implies R2) \) \( s \) \( s \) \( \land \)
\( (\text{rep}1 \implies \text{abs}2) \) \( r = (\text{rep}1 \implies \text{abs}2) \) \( s \) \implies (R1 \implies R2) \) \( r \) \( s \)
apply (auto simp add: rel-fun-def fun-eq-iff) using q1 q2 unfolding Quotient3-def
by (metis map-fun-apply)

ultimately show \?thesis by blast
qed
\}
ultimately show \?thesis by (intro Quotient3I) (assumption+) 
qed

lemma lambda-prs:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows \( (\text{Rep}1 \implies \text{Abs}2) \) \( (\lambda x. \text{Rep}2 (f (\text{Abs}1 x))) = (\lambda x. f x) \)
unfolding fun-eq-iff
using Quotient3-abs-rep[of q1] Quotient3-abs-rep[of q2]
by simp

lemma lambda-prs1:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows \( (\text{Rep}1 \implies \text{Abs}2) \) \( (\lambda x. (\text{Abs}1 \implies \text{Rep}2) f x) = (\lambda x. f x) \)
unfolding fun-eq-iff
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by simp

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use apply-rsp and not the primed version

lemma apply-rspQ3:
  fixes f g :: 'a ⇒ 'c
  assumes q: Quotient3 R1 Abs1 Rep1
  and a: (R1 ===> R2) f g R1 x y
  shows R2 (f x) (g y)
  using a by (auto elim: rel-funE)

lemma apply-rspQ3'":
  assumes Quotient3 R Abs Rep
  and (R ===> S) f f
  shows S (f (Rep x)) (f (Rep x))
proof –
  from assms(1) have R (Rep x) (Rep x) by (rule Quotient3-rep-reflp)
  then show ?thesis using assms(2) by (auto intro: apply-rsp')
qed

47.2 lemmas for regularisation of ball and bex

lemma ball-reg-eqv:
  fixes P :: 'a ⇒ bool
  assumes a: equivp R
  shows Ball (Respects R) P = (All P)
  using a
  unfolding equivp-def
  by (auto simp add: in-respects)

lemma bex-reg-eqv:
  fixes P :: 'a ⇒ bool
  assumes a: equivp R
  shows Bex (Respects R) P = (Ex P)
  using a
  unfolding equivp-def
  by (auto simp add: in-respects)

lemma ball-reg-right:
  assumes a: ∀x. x ∈ R ⇒ P x → Q x
  shows All P → Ball R Q
  using a by fast

lemma bex-reg-left:
  assumes a: ∀x. x ∈ R ⇒ Q x → P x
  shows Bex R Q → Ex P
using 
a 

lemma ball-reg-left:
  assumes a: equivp R
  shows \( \bigwedge x. (Q x \rightarrow P x) \) \implies Ball (Respects R) Q \rightarrow All P
  using a 

lemma bex-reg-right:
  assumes a: equivp R
  shows \( \bigwedge x. (Q x \rightarrow P x) \) \implies Ex Q \rightarrow Bex (Respects R) P
  using a 

lemma all-reg:
  assumes a: !x :: 'a \Rightarrow \bigwedge y. (P x \rightarrow Q x)
  and b: All P
  shows All Q
  using a b 

lemma ex-reg:
  assumes a: !x :: 'a \Rightarrow (P x \rightarrow Q x)
  and b: Ex P
  shows Ex Q
using \( a \) \( b \) by fast

**lemma** ball-reg:

assumes \( a : !x :: 'a. (x \in R --> P x --> Q x) \)

and \( b : \text{Ball} \ R \ P \)

shows \( \text{Ball} \ R \ Q \)

using \( a \) \( b \) by fast

**lemma** bex-reg:

assumes \( a : !x :: 'a. (x \in R --> P x --> Q x) \)

and \( b : \text{Bex} \ R \ P \)

shows \( \text{Bex} \ R \ Q \)

using \( a \) \( b \) by fast

**lemma** ball-all-comm:

assumes \( \forall y. (\forall x \in P. A x y) --> (\forall x. B x y) \)

shows \( (\forall x \in P. \forall y. A x y) --> (\forall x. \forall y. B x y) \)

using asms by auto

**lemma** bex-ex-comm:

assumes \( \exists y. \exists x. A x y \) ---\( \exists x. \exists y. B x y \)

shows \( \exists x. \exists y. A x y \) ---\( \exists x. \exists y. B x y \)

using asms by auto

### 47.3 Bounded abstraction

**definition**

\( \text{Babs} :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \)

**where**

\( x \in p \Rightarrow \text{Babs} \ p \ m \ x = m \ x \)

**lemma** babs-rsp:

assumes \( q : \text{Quotient3} \ R1 \ Abs1 \ Rep1 \)

and \( a : (R1 ===> R2) \ f \ g \)

shows \( (R1 ===> R2) \ (\text{Babs} \ (\text{Respects} \ R1) \ f) \ (\text{Babs} \ (\text{Respects} \ R1) \ g) \)

apply (auto simp add: Babs-def in-respects rel-fun-def)

apply (subgoal-tac \( x \in \text{Respects} \ R1 \ \&\ y \in \text{Respects} \ R1 \))

using a apply (simp add: Babs-def rel-fun-def)

using Quotient3-rel[OF q]

by metis

**lemma** babs-prs:

assumes q1: \( \text{Quotient3} \ R1 \ Abs1 \ Rep1 \)

and q2: \( \text{Quotient3} \ R2 \ Abs2 \ Rep2 \)

shows \( ((\text{Rep1} ===> \text{Abs2}) \ (\text{Babs} \ (\text{Respects} \ R1) \ ((\text{Abs1} ===> \text{Rep2}) \ f))) = f \)

apply (rule ext)
apply (simp add:)
apply (subgoal-tac Rep1 x ∈ Respects R1)
apply (simp add: Babs-def Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2])
apply (simp add: in-respects Quotient3-rel-rep[OF q1])
done

lemma babs-simp:
assumes q: Quotient3 R1 Abs Rep
shows ((R1 ===> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)) =
((R1 ===> R2) f g)
apply (rule iffI)
apply (simp-all only: babs-rsp[OF q])
apply (auto simp add: Babs-def rel-fun-def)
apply (subgoal-tac x ∈ Respects R1 ∧ y ∈ Respects R1)
apply (metis Babs-def)
apply (simp add: in-respects)
using Quotient3-rel[OF q]
by (metis)

lemma babs-reg-eqv:
shows equivp R ⇒ Babs (Respects R) P = P
by (simp add: fun-eq-iff Babs-def in-respects equivp-reflp)

lemma ball-rsp:
assumes a: (R ===> (op =)) f g
shows Ball (Respects R) f = Ball (Respects R) g
using a by (auto simp add: Ball-def in-respects elim: rel-funE)

lemma bex-rsp:
assumes a: (R ===> (op =)) f g
shows Bex (Respects R) f = Bex (Respects R) g
using a by (auto simp add: Bex-def in-respects elim: rel-funE)

lemma bex1-rsp:
assumes a: (R ===> (op =)) f g
shows Ex1 (λx. x ∈ Respects R ∧ f x) = Ex1 (λx. x ∈ Respects R ∧ g x)
using a by (auto elim: rel-funE simp add: Ex1-def in-respects)

lemma all-prs:
assumes a: Quotient3 R absf repf
shows Ball (Respects R) ((absf ===> id) f) = All f
using a unfolding Quotient3-def Ball-def in-respects id-apply comp-def map-fun-def
by (metis)

lemma ex-prs:
assumes $a$: Quotient3 $R$ $\text{abf}$ $\text{repf}$

shows $\text{Bex} ((\text{Respects} \ R) ((\text{abf} \Rightarrow \text{id}) \ f) = \text{Ex} \ f$

using a unfolding Quotient3-def Bex-def in-respects id-apply comp-def map-fun-def
by metis

47.4 $\text{Bex1-rel}$ quantifier

definition
$\text{Bex1-rel} ::= (a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow (a \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where
$\text{Bex1-rel} \ R \ P \iff \exists x \in \text{Respects} \ R. P x \land \forall x \in \text{Respects} \ R. \forall y \in \text{Respects} \ R. ((P x \land P y) \rightarrow (R x y))$

lemma $\text{bex1-rel-aux}$:
$\forall xa ya. R xa ya \rightarrow x xa = y ya; \text{Bex1-rel} \ R x \Rightarrow \text{Bex1-rel} \ R y$

unfolding $\text{Bex1-rel-def}$

apply (erule conjE)+

apply (erule bexE)

apply rule

apply (rule-tac $x=xa$ in bexI)

apply metis

apply metis

apply rule+

apply (erule-tac $x=xa$ in ballE)

prefer 2

apply (metis)

apply (erule-tac $x=ya$ in ballE)

prefer 2

apply (metis)

apply (metis in-respects)

done

lemma $\text{bex1-rel-aux2}$:
$\forall xa ya. R xa ya \rightarrow x xa = y ya; \text{Bex1-rel} \ R y \Rightarrow \text{Bex1-rel} \ R x$

unfolding $\text{Bex1-rel-def}$

apply (erule conjE)+

apply (erule bexE)

apply rule

apply (rule-tac $x=xa$ in bexI)

apply metis

apply metis

apply rule+

apply (erule-tac $x=xa$ in ballE)

prefer 2

apply (metis)

apply (erule-tac $x=ya$ in ballE)

prefer 2

apply (metis)

apply (metis in-respects)
THEORY “Quotient”

done


lemma \textit{bex1-rel-rsp}:
\begin{itemize}
  \item \textbf{assumes} \( a : \text{Quotient3} \ R \ \text{absf repf} \)
  \item \textbf{shows} \(( (R \Longrightarrow op =) \Longrightarrow op =) (\text{Bex1-rel} \ R) (\text{Bex1-rel} \ R) \)
  \item \textbf{apply} \((\text{simp add: rel-fun-def})\)
  \item \textbf{apply} \text{clarify}
  \item \textbf{apply} \text{rule}
  \item \textbf{apply} \((\text{simp-all add: bex1-rel-aux bex1-rel-aux2})\)
  \item \textbf{apply} \((\text{erule bex1-rel-aux2})\)
  \item \textbf{apply} \text{assumption}
\end{itemize}
done


lemma \textit{ex1-prv}:
\begin{itemize}
  \item \textbf{assumes} \( a : \text{Quotient3} \ R \ \text{absf repf} \)
  \item \textbf{shows} \(( (\text{absf \longrightarrow id}) \longrightarrow id) (\text{Bex1-rel} \ R) f = \text{Ex1} f \)
  \item \textbf{apply} \((\text{simp add:})\)
  \item \textbf{apply} \((\text{subst Bex1-rel-def})\)
  \item \textbf{apply} \((\text{subst Bex-def})\)
  \item \textbf{apply} \((\text{subst Ex1-def})\)
  \item \textbf{apply} \text{simp}
  \item \textbf{apply} \text{rule}
  \item \textbf{apply} \((\text{erule conjE})+\)
  \item \textbf{apply} \((\text{erule-tac exE})\)
  \item \textbf{apply} \((\text{erule conjE})\)
  \item \textbf{apply} \((\text{subgoal-tac \forall y. R y y \Longrightarrow f (absf y) \Longrightarrow R x y})\)
  \item \textbf{apply} \((\text{rule-tac \forall x=absf x \in exI})\)
  \item \textbf{apply} \((\text{simp})\)
  \item \textbf{apply} \text{rule}+
  \item \textbf{using} \text{a unfolding Quotient3-def}
  \item \textbf{apply} \text{metis}
  \item \textbf{apply} \text{rule}+
  \item \textbf{apply} \((\text{erule-tac \forall x=x \in ballE})\)
  \item \textbf{apply} \((\text{erule-tac \forall x=y \in ballE})\)
  \item \textbf{apply} \text{simp}
  \item \textbf{apply} \((\text{simp add: in-respects})\)
  \item \textbf{apply} \((\text{simp add: in-respects})\)
  \item \textbf{apply} \((\text{erule-tac exE})\)
  \item \textbf{apply} \text{rule}
  \item \textbf{apply} \((\text{rule-tac \forall x=repf x \in exI})\)
  \item \textbf{apply} \((\text{simp only: in-respects})\)
  \item \textbf{apply} \text{rule}
  \item \textbf{apply} \((\text{metis Quotient3-rel-rep[OF a]})\)
\end{itemize}
\begin{itemize}
  \item \textbf{using} \text{a unfolding Quotient3-def}
  \item \textbf{apply} \((\text{simp})\)
  \item \textbf{apply} \text{rule}+
  \item \textbf{using} \text{a unfolding Quotient3-def in-respects}
  \item \textbf{apply} \text{metis}
\end{itemize}
done
lemma bex1-bexeq-reg:
  shows (\exists! x \in \text{Respects } R. P x) \longrightarrow (Bex1-rel R (\lambda x. P x))
by (auto simp add: Ex1-def Bex1-rel-def Bex-def Ball-def in-respects)

lemma bex1-bexeq-reg-eqv:
  assumes a: equivp R
  shows (\exists! x. P x) \longrightarrow Bex1-rel R P
using equivp-reflp[OF a]
apply (intro impI)
apply (elim ex1E)
apply (rule mp[OF bex1-bexeq-reg])
apply (rule-tac a = x in ex1I)
apply (subst in-respects)
apply (rule conjI)
apply assumption
apply assumption
apply clarify
apply (erule-tac x = xa in allE)
apply simp
done

47.5 Various respects and preserve lemmas

lemma quot-rel-rsp:
  assumes a: Quotient3 R Abs Rep
  shows (R === R === op =) R R
apply (rule rel-funI)+
apply (rule equals-rsp[OF a])
apply (assumption)+
done

lemma o-prs:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and q2: Quotient3 R2 Abs2 Rep2
  and q3: Quotient3 R3 Abs3 Rep3
  shows ((Abs2 ----> Rep3) ----> (Abs1 ----> Rep2) ----> (Rep1 ----> Abs3)) op o = op o
  and (id ----> (Abs1 ----> id) ----> Rep1 ----> id) op o = op o
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3]
by (simp-all add: fun-eq-iff)

lemma o-rsp:
  ((R2 === R3) ===> (R1 ===> R2) ===> (R1 ===> R3)) op o op o
  (op = ===> (R1 ===> op =) ===> R1 ===> op =) op o op o
by (force elim: rel-fanE)+

lemma cond-prs:
assumes a: Quotient3 R absf repf
shows absf (if a then repf b else repf c) = (if a then b else c)
using a unfolding Quotient3-def by auto

lemma if-prs:
assumes q: Quotient3 R Abs Rep
shows (id ===> Rep ===> Rep ===> Abs) If = If
using Quotient3-abs-rep[OF q]
by (auto simp add: fun-eq-iff)

lemma if-rsp:
assumes q: Quotient3 R Abs Rep
shows (op == ===> R ===> R ===> R) If If
by force

lemma let-prs:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows (Rep2 ===> (Abs2 ===> Rep1) ===> Abs1) Let = Let
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by (auto simp add: fun-eq-iff)

lemma let-rsp:
shows (R1 ===> (R1 ===> R2) ===> R2) Let Let
by (force elim: rel-funE)

lemma id-rsp:
shows (R ===> R) id id
by auto

lemma id-prs:
assumes a: Quotient3 R Abs Rep
shows (Rep ===> Abs) id = id
by (simp add: fun-eq-iff Quotient3-abs-rep [OF a])
end

locale quot-type =
  fixes R :: 'a ⇒ 'a ⇒ bool
  and Abs :: 'a set ⇒ 'b
  and Rep :: 'b ⇒ 'a set
  assumes equivp: part-equivp R
  and rep-prop: ∀y. ∃x. R x x ∧ Rep y = Collect (R x)
  and rep-inverse: ∀x. Abs (Rep x) = x
  and abs-inverse: ∀c. (∃x. ((R x x) ∧ (c = Collect (R x)))) ⇒ (Rep (Abs c)) = c
  and rep-inject: ∀x y. (Rep x = Rep y) = (x = y)
begin
definition
abs :: 'a ⇒ 'b
where
abs x = Abs (Collect (R x))

definition
rep :: 'b ⇒ 'a
where
rep a = (SOME x. x ∈ Rep a)

lemma some-collect:
  assumes R r r
  shows R (SOME x. x ∈ Collect (R r)) = R r
  apply simp
  by (metis assms exE-some equivp [simplified part-equivp-def])

lemma Quotient:
  shows Quotient3 R abs rep
  unfolding Quotient3-def abs-def rep-def
  proof (intro conjI allI)
    fix a r s
    show x: R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a) proof -
      obtain x where r: R x x and rep: Rep a = Collect (R x) using rep-prop [of a] by auto
      have R (SOME x. x ∈ Rep a) x using r rep some-collect by metis
      then have R x (SOME x. x ∈ Rep a) using part-equivp-symp [OF equivp] by fast
      then show R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a)
        using part-equivp-transp [OF equivp] by (metis ( SOME x. x ∈ Rep a) x)
    qed
    have Collect (R (SOME x. x ∈ Rep a)) = (Rep a) by (metis some-collect rep-prop)
    then show Abs (Collect (R (SOME x. x ∈ Rep a))) = a using rep-inverse by auto
    have R r r =⇒ R s s =⇒ Abs (Collect (R r)) = Abs (Collect (R s)) =⇒ R r = R s
      proof -
        assume R r r and R s s
        then have Abs (Collect (R r)) = Abs (Collect (R s)) =⇒ Collect (R s)
          by (metis abs-inverse)
        also have Collect (R r) = Collect (R s) =⇒ (λA x. x ∈ A) (Collect (R r))
          = (λA x. x ∈ A) (Collect (R s))
          by rule simp-all
        finally show Abs (Collect (R r)) = Abs (Collect (R s)) =⇒ R r = R s by simp
      qed
    then show R r s =⇒ R r r ∧ R s s ∧ (Abs (Collect (R r)) = Abs (Collect (R s))) =⇒ R r = R s
      by (metis ( SOME x. x ∈ Rep a) x)
THEORY "Quotient"

(R s))

using equivp[simplified part-equivp-def] by metis

qed

end

47.6 Quotient composition

lemma OOO-quotient3:
  fixes R1 :: 'a ⇒ 'a ⇒ bool
  fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
  fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
  fixes R2 :: 'a ⇒ 'a ⇒ bool
  assumes R1: Quotient3 R1 Abs1 Rep1
  assumes R2: Quotient3 R2 Abs2 Rep2
  assumes Abs1: ∀ x y. R2' x y =⇒ R1 x x =⇒ R1 y y =⇒ R2 (Abs1 x) (Abs1 y)
  assumes Rep1: ∀ x y. R2 x y =⇒ R2' (Rep1 x) (Rep1 y)
  shows Quotient3 (R1 OO R2' OO R1) (Abs2 ◦ Abs1) (Rep1 ◦ Rep2)
apply (rule Quotient3I)
apply (simp add: o-def Quotient3-abs-rep [OF R2] Quotient3-abs-rep [OF R1])
apply simp
apply (rule-tac b=Rep1 (Rep2 a) in relcomppI)
  apply (rule Quotient3-rep-reflp [OF R1])
apply (rule-tac b=Rep1 (Rep2 a) in relcomppI [rotated])
  apply (rule Quotient3-rep-reflp [OF R1])
apply (rule Rep1)
apply (rule Quotient3-rep-reflp [OF R2])
apply safe:
  apply (rename-tac x y)
  apply (drule Abs1)
    apply (erule Quotient3-refl2 [OF R1])
    apply (erule Quotient3-refl1 [OF R1])
  apply (drule Quotient3-refl1 [OF R2], drule Rep1)
  apply (subgoal-tac R1 r (Rep1 (Abs1 x)))
  apply (rule-tac b=Rep1 (Abs1 x) in relcomppI, assumption)
  apply (erule relcomppI)
  apply (erule Quotient3-symp [OF R1, THEN sympD])
  apply (rule Quotient3-sym[symmetric, OF R1, THEN iffD2])
  apply (rule conjI, erule Quotient3-refl1 [OF R1])
  apply (rule conjI, rule Quotient3-reflp [OF R1])
  apply (subst Quotient3-abs-rep [OF R1])
  apply (erule Quotient3-refl-abs [OF R1])
  apply (rename-tac x y)
  apply (drule Abs1)
    apply (erule Quotient3-refl2 [OF R1])
    apply (erule Quotient3-refl1 [OF R1])
  apply (drule Quotient3-refl2 [OF R2], drule Rep1)
apply (subgoal-tac R1 s (Rep1 (Abs1 y)))
apply (rule-tac b=Rep1 (Abs1 y) in relcomppI, assumption)
apply (erule relcomppI)
apply (erule Quotient3-symp [OF R1, THEN sympD])
apply (rule Quotient3-rel[symmetric, OF R1, THEN iffD2])
apply (rule conjI, erule Quotient3-refl2 [OF R1])
apply (rule conjI, rule Quotient3-rep-reflp [OF R1])
apply (subst Quotient3-abs-rep [OF R1])
apply (erule Quotient3-rel-abs [OF R1, THEN sym])
apply simp
apply (rule Quotient3-rel-abs [OF R2])
apply (rule Quotient3-rel-abs [OF R1, THEN subst], assumption)
apply (rule Quotient3-rel-abs [OF R1, THEN subst], assumption)
apply (erule Abs1)
apply (erule Quotient3-refl2 [OF R1])
apply (erule Quotient3-refl1 [OF R1])
apply (rename-tac a b c d)
apply simp
apply (rule-tac b=Rep1 (Abs1 r) in relcomppI)
apply (rule Quotient3-rel[symmetric, OF R1, THEN iffD2])
apply (rule conjI, erule Quotient3-refl1 [OF R1])
apply (simp add: Quotient3-abs-rep [OF R1] Quotient3-rep-reflp [OF R1])
apply (rule-tac b=Rep1 (Abs1 s) in relcomppI [rotated])
apply (rule Quotient3-rel[symmetric, OF R1, THEN iffD2])
apply (simp add: Quotient3-abs-rep [OF R1] Quotient3-rep-reflp [OF R1])
apply (erule Quotient3-refl2 [OF R1])
apply (rule Rep1)
apply (drule Abs1)
apply (erule Quotient3-refl2 [OF R1])
apply (erule Quotient3-refl1 [OF R1])
apply (drule Abs1)
apply (erule Quotient3-refl2 [OF R1])
apply (erule Quotient3-refl1 [OF R1])
apply (drule Quotient3-rel-abs [OF R1])
apply (drule Quotient3-rel-abs [OF R1])
apply (drule Quotient3-rel-abs [OF R1])
apply simp
apply (rule Quotient3-rel[symmetric, OF R2, THEN iffD2])
apply simp
done

lemma OOO-eq-quotient3:
  fixes R1 :: 'a ⇒ 'a ⇒ bool
  fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
  fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
  assumes R1: Quotient3 R1 Abs1 Rep1
  assumes R2: Quotient3 op= Abs2 Rep2
  shows Quotient3 (R1 OOO op=) (Abs2 ∘ Abs1) (Rep1 ∘ Rep2)
47.7 Quotient3 to Quotient

**Lemma Quotient3-to-Quotient:**

**Assumes** Quotient3 R Abs Rep

**And** \( T \equiv \lambda x y. R x x \land Abs x = y \)

**Shows** Quotient R Abs Rep T

**Using** assms unfolding Quotient3-def by (intro QuotientI) blast+

**Lemma Quotient3-to-Quotient-equivp:**

**Assumes** q: Quotient3 R Abs Rep

**And** T-def: \( T \equiv \lambda x y. Abs x = y \)

**And** eR: equivp R

**Shows** Quotient R Abs Rep T

**Proof** (intro QuotientI)

fix a

show Abs (Rep a) = a using q by (rule Quotient3-abs-rep)

next

fix a

show R (Rep a) (Rep a) using q by (rule Quotient3-rep-reflp)

next

fix r s

show R r s = (R r r \land R s s \land Abs r = Abs s) using q by (rule Quotient3-rel[symmetric])

next

show T = (\lambda x y. R x x \land Abs x = y) using T-def equivp-reflp[OF eR] by simp

qed

47.8 ML setup

Auxiliary data for the quotient package

**ML-file** Tools/Quotient/quotient-info.ML

**Setup** Quotient-Info.setup

**Declare** [[mapQ3 fun = (rel-fun, fun-quotient3))]]

**Lemmas**

[quot-thm] = fun-quotient3

[quot-respect] = quot-rel-rsp if-rsp o-rsp let-rsp id-rsp

[quot-preserve] = if-prs o-prs let-prs id-prs

[quot-equiv] = identity-equivp

Lemmas about simplifying id’s.

**Lemmas**

[id-simps] =

id-def[symmetric]

map-fun-id

id-apply

id-o

o-id
Translation functions for the lifting process.

**ML-file** Tools/Quotient/quotient-term.ML

Definitions of the quotient types.

**ML-file** Tools/Quotient/quotient-type.ML

Definitions for quotient constants.

**ML-file** Tools/Quotient/quotient-def.ML

An auxiliary constant for recording some information about the lifted theorem in a tactic.

**definition**

\[ \text{Quot-True} :: 'a \Rightarrow \text{bool} \]

**where**

\[ \text{Quot-True} \; x \leftrightarrow \text{True} \]

**lemma**

shows \( \text{QT-all}: \text{Quot-True} \; (\text{All} \; P) \implies \text{Quot-True} \; P \)

and \( \text{QT-ex}: \text{Quot-True} \; (\text{Ex} \; P) \implies \text{Quot-True} \; P \)

and \( \text{QT-ext}: \text{Quot-True} \; (\lambda x. \; P \; x) \implies (\forall x. \; \text{Quot-True} \; (P \; x)) \)

and \( \text{QT-lam}: (\forall x. \; \text{Quot-True} \; (a \; x) \implies f \; x = g \; x) \implies (\text{Quot-True} \; a \implies f = g) \)

by \((\text{simp-all add: Quot-True-def ext})\)

**lemma** \( \text{QT-imp}: \text{Quot-True} \; a \equiv \text{Quot-True} \; b \)

by \((\text{simp add: Quot-True-def})\)

**context**

**begin**

**interpretation** lifting-syntax .

Tactics for proving the lifted theorems

**ML-file** Tools/Quotient/quotient-tacs.ML

end

### 47.9 Methods / Interface

**method-setup** lifting =

\[
\langle \langle \text{Attrib.thms >> } (\text{fn thms => fn ctxt => SIMPLE-METHOD'} \; (\text{Quotient-Tacs.lift-tac ctxt \} \; \text{thms})) \rangle) \\
\langle \langle \text{lift theorems to quotient types} \rangle\rangle
\]

**method-setup** lifting-setup =
(Attrib.thm >> (fn thm => fn ctxt =>
    SIMPLE-METHOD' (Quotient-Tacs.lift-procedure-tac ctxt [] thm)))

method-setup descending =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.descend-tac ctxt [])))

method-setup descending-setup =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.descend-procedure-tac ctxt [])))

method-setup partiality-descending =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-tac ctxt [])))

method-setup partiality-descending-setup =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-procedure-tac ctxt [])))

method-setup regularize =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.regularize-tac ctxt)))

method-setup injection =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.all-injection-tac ctxt)))

method-setup cleaning =
  (Scan.succeed (fn ctxt =>
      SIMPLE-METHOD' (Quotient-Tacs.clean-tac ctxt)))

attribute-setup quot-lifted =
  (Scan.succeed Quotient-Tacs.lifted-attrib)

no-notation
  rel-conj (infixr OOO 75)
48  Complete-Partial-Order: Chain-complete partial orders and their fixpoints

theory  Complete-Partial-Order
imports  Product-Type
begin

48.1  Monotone functions

Dictionary-passing version of mono.

definition monotone :: \('a \Rightarrow 'a \Rightarrow \text{bool}\) \Rightarrow \('b \Rightarrow 'b \Rightarrow \text{bool}\) \Rightarrow \('a \Rightarrow 'b \Rightarrow \text{bool}\)
where  monotone orda ordb f \leftrightarrow (\forall x y. orda x y \rightarrow ordb (f x) (f y))

lemma monotoneI[intro?): (\forall x y. orda x y \Rightarrow ordb (f x) (f y)) \Rightarrow monotone orda ordb f
unfolding monotone-def by iprover

lemma monotoneD[dest?): monotone orda ordb f \Rightarrow orda x y \Rightarrow ordb (f x) (f y)
unfolding monotone-def by iprover

48.2  Chains

A chain is a totally-ordered set. Chains are parameterized over the order for maximal flexibility, since type classes are not enough.

definition chain :: \('a \Rightarrow 'a \Rightarrow \text{bool}\) \Rightarrow \text{'a set} \Rightarrow \text{bool}
where  chain ord S \leftrightarrow (\forall x \in S. \forall y \in S. ord x y \lor ord y x)

lemma chainI:
  assumes \(\forall x y. x \in S \Rightarrow y \in S \Rightarrow ord x y \lor ord y x\)
  shows  chain ord S
using asms unfolding chain-def by fast

lemma chainD:
  assumes chain ord S and x \in S and y \in S
  shows  ord x y \lor ord y x
using asms unfolding chain-def by fast

lemma chainE:
  assumes chain ord S and x \in S and y \in S
  obtains  ord x y | ord y x
using asms unfolding chain-def by fast

lemma chain-empty: chain ord {}
by(simp add: chain-def)
48.3 Chain-complete partial orders

A ccpo has a least upper bound for any chain. In particular, the empty set is a chain, so every ccpo must have a bottom element.

class ccpo = order + Sup +
  assumes ccpo-Sup-upper: [chain (op ≤) A; x ∈ A] ⇒ x ≤ Sup A
  assumes ccpo-Sup-least: [chain (op ≤) A; \x. x ∈ A ⇒ x ≤ z] ⇒ Sup A ≤ z
begin

48.4 Transfinite iteration of a function

inductive-set iterates :: ('a ⇒ 'a) ⇒ 'a set
for f :: 'a ⇒ 'a
where
  step: x ∈ iterates f ⇒ f x ∈ iterates f
| Sup: chain (op ≤) M ⇒ ∀x∈M. x ∈ iterates f ⇒ Sup M ∈ iterates f

lemma iterates-le-f:
  x ∈ iterates f ⇒ monotone (op ≤) (op ≤) f ⇒ x ≤ f x
by (induct x rule: iterates.induct)
  (force dest: monotoneD intro: ccpo-Sup-upper ccpo-Sup-least)+

lemma chain-iterates:
  assumes f: monotone (op ≤) (op ≤) f
  shows chain (op ≤) (iterates f) (is chain - ?C)
proof (rule chainI)
fix x y assume x ∈ ?C y ∈ ?C
then show x ≤ y ∨ y ≤ x
proof (induct x arbitrary: y rule: iterates.induct)
  fix x y assume y: y ∈ ?C
  and IH: \z. z ∈ ?C ⇒ x ≤ z ∨ z ≤ x
  from y show f x ≤ y ∨ y ≤ f x
proof (induct y rule: iterates.induct)
  case (step y) with IH f show ?case by (auto dest: monotoneD)
next
  case (Sup M)
  then have chM: chain (op ≤) M
    and IH': \z. z ∈ M ⇒ f x ≤ z ∨ z ≤ f x by auto
  show f x ≤ Sup M ∨ Sup M ≤ f x
proof (cases \z∈M. f x ≤ z)
    case True then have f x ≤ Sup M
      apply rule
      apply (erule order-trans)
      by (rule ccpo-Sup-upper[OF chM])
    thus ?thesis ..
next
  case False with IH'
  show ?thesis by (auto intro: ccpo-Sup-least[OF chM])
qed
qed
next
case (Sup M y)
  show ?case
  proof (cases \( \exists x \in M. \ y \leq x \))
    case True then have \( y \leq \Sup M \)
      apply rule
      apply (erule order-trans)
      by (rule ccpo-Sup-upper[of Sup(1)])
    thus ?thesis ..
  next
    case False with Sup
    show ?thesis by (auto intro: ccpo-Sup-least)
qed
qed

lemma bot-in-iterates: Sup {} \in iterates f
by(auto intro: iterates.Sup simp add: chain-empty)

48.5 Fixpoint combinator

definition
  fixp :: ('a \Rightarrow 'a) \Rightarrow 'a
where
  fixp f = Sup (iterates f)

lemma iterates-fixp:
  assumes f: monotone (op \leq) (op \leq) f
  shows fixp f \in iterates f
unfolding fixp-def
by (simp add: iterates.Sup chain-iterates f)

lemma fixp-unfold:
  assumes f: monotone (op \leq) (op \leq) f
  shows fixp f = f (fixp f)
proof (rule antisym)
  show fixp f \leq f (fixp f)
    by (intro iterates-le-f iterates-fixp f)
  have f (fixp f) \leq Sup (iterates f)
    by (intro ccpo-Sup-upper chain-iterates f iterates.step iterates-fixp)
  thus f (fixp f) \leq fixp f
    unfolding fixp-def .
qed

lemma fixp-lowerbound:
  assumes f: monotone (op \leq) (op \leq) f and z: f z \leq z
  shows fixp f \leq z
unfolding fixp-def
proof (rule ccpo-Sup-least[of chain-iterates[of f]])
fix \( x \) assume \( x \in \text{iterates} \, f \)
thus \( x \leq z \)
proof (induct \( x \) rule: \text{iterates.induct})
fix \( x \) assume \( x \leq z \) with \( f \) have \( f \, x \leq f \, z \) by (rule \text{monotoneD})
also note \( z \) finally show \( f \, x \leq z \).
qed (auto intro: ccpo-Sup-least)
qed

48.6 Fixpoint induction

setup ⟨⟨ Sign.map-naming (Name-Space.mandatory-path ccpo) ⟩⟩

definition admissible :: ('a set ⇒ 'a) ⇒ ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ bool
where admissible lub ord P = (∀ A. chain ord A −→ (A ≠ {}) −→ (∀ x∈A. P \( x \)) −→ P (lub A))

lemma admissibleI:
names \( A. \chainord A \Rightarrow A ≠ {} \Rightarrow (∀ x∈A. P \( x \)) \Rightarrow P (\lub A) \)
shows ccpo.admissible lub ord P
using assms unfolding ccpo.admissible-def by fast

lemma admissibleD:
names ccpo.admissible lub ord P
assumes chain ord A
assumes \( A ≠ {} \)
assumes (∀ x. x ∈ A ⇒ P \( x \))
shows P (lub A)
using assms by (auto simp: ccpo.admissible-def)

setup ⟨⟨ Sign.map-naming Name-Space.parent-path ⟩⟩

lemma (in ccpo) fixp-induct:
names adm: ccpo.admissible Sup (op ≤) P
assumes mono: monotone (op ≤) (op ≤) \( f \)
assumes bot: P (Sup \{\})
assumes step: (∀ x. P \( x \) ⇒ P \( f \, x \))
shows P (fixp \( f \))
unfolding fixp-def using adm chain-iterates[OF mono]
proof (rule ccpo.admissibleD)
show \( \text{iterates} \, f ≠ {} \) using bot-in-iterates by auto
fix \( x \) assume \( x \in \text{iterates} \, f \)
thus P \( x \)
  by (induct rule: \text{iterates.induct})
  (case_tac \( M = {} \), auto intro: step bot ccpo.admissibleD adm)
qed

lemma admissible-True: ccpo.admissible lub ord (λx. True)
unfolding  \texttt{ccpo.admissible-def} by \texttt{simp}

\textbf{lemma \texttt{admissible-const}}:\ \texttt{ccpo.admissible lub ord (λx. t)}
\textbf{by (auto intro: ccpo.admissibleI)}

\textbf{lemma \texttt{admissible-conj}}:
- \texttt{assumes ccpo.admissible lub ord (λx. P x)}
- \texttt{assumes ccpo.admissible lub ord (λx. Q x)}
- \texttt{shows ccpo.admissible lub ord (λx. P x \land Q x)}
\textbf{using assms unfolding ccpo.admissible-def by simp}

\textbf{lemma \texttt{admissible-all}}:
- \texttt{assumes \(\forall y. ccpo.admissible lub ord (λx. P x y)\)}
- \texttt{shows ccpo.admissible lub ord (λx. \(\forall y \in A. P x y\))}
\textbf{using assms unfolding ccpo.admissible-def by fast}

\textbf{lemma \texttt{admissible-ball}}:
- \texttt{assumes \(\forall y. y \in A \Rightarrow ccpo.admissible lub ord (λx. P x y)\)}
- \texttt{shows ccpo.admissible lub ord (λx. \(\forall y \in A. P x y\))}
\textbf{using assms unfolding ccpo.admissible-def by fast}

\textbf{context ccpo begin}

\textbf{lemma \texttt{admissible-disj-lemma}}:
- \texttt{assumes A: chain (op ≤) A}
- \texttt{assumes P: \(\forall x \in A. \exists y \in A. x \leq y \land P y\)}
- \texttt{shows Sup A = Sup \{x \in A. P x\}}
\textbf{proof (rule antisym)}
\textbf{have *: chain (op ≤) \{x \in A. P x\}}
\textbf{by (rule chain-compr [OF A])}
\textbf{show Sup A ≤ Sup \{x \in A. P x\}}
\textbf{apply (rule ccpo-Sup-least [OF A])}
\textbf{apply (erule order-trans)}
\textbf{apply (simp add: ccpo-Sup-upper [OF *])}
\textbf{done}
\textbf{show Sup \{x \in A. P x\} ≤ Sup A}
\textbf{apply (rule ccpo-Sup-least [OF *])}
\textbf{apply clarify}
\textbf{apply simp add: ccpo-Sup-upper [OF A]}
\textbf{done}
\textbf{qed}

\textbf{lemma \texttt{admissible-disj}}:
\textbf{fixes P Q :: 'a ⇒ bool}
assumes $P$: ccpo.admissible $\operatorname{Sup} (\operatorname{op} \leq) (\lambda x. \, P \, x)$
assumes $Q$: ccpo.admissible $\operatorname{Sup} (\operatorname{op} \leq) (\lambda x. \, Q \, x)$
shows ccpo.admissible $\operatorname{Sup} (\operatorname{op} \leq) (\lambda x. \, P \, x \lor Q \, x)$

proof (rule ccpo.admissibleI)
  fix $A :: \{a\}$ set
  assume $A$: chain $(\operatorname{op} \leq) A$

  and $\forall x \in A. \, P \, x \lor Q \, x$
  hence $(\exists x \in A. \, P \, x) \land (\forall x \in A. \exists y \in A. \, x \leq y \land P \, y) \lor (\exists x \in A. \, Q \, x) \land (\forall x \in A. \exists y \in A. \, x \leq y \land Q \, y)$
  using chainD[OF $A$] by blast
  hence $(\exists x. \, x \in A \land P \, x) \land \operatorname{Sup} \, A = \operatorname{Sup} \{x \in A. \, P \, x\} \lor (\exists x. \, x \in A \land Q \, x) \land \operatorname{Sup} \, A = \operatorname{Sup} \{x \in A. \, Q \, x\}$
  using admissible-disj-lemma [OF $A$] by blast
  thus $P \, (\operatorname{Sup} \, A) \lor Q \, (\operatorname{Sup} \, A)$
  apply (rule disjE, simp-all)
  apply (rule disjI1, rule ccpo.admissibleD [OF $P$ chain-compr [OF $A$], simp, simp])
  apply (rule disjI2, rule ccpo.admissibleD [OF $Q$ chain-compr [OF $A$], simp, simp])
  done

qed

hide-const (open) iterates fixp

end

instance complete-lattice $\subseteq$ ccpo
  by default (fast intro: Sup-upper Sup-least)+

lemma lfp-eq-fixp:
  assumes $f$: mono $f$ shows lfp $f$ = fixp $f$
proof (rule antisym)
  from $f$ have $f'$: monotone $(\operatorname{op} \leq) (\operatorname{op} \leq) f$
    unfolding mono-def monotone-def .
  show lfp $f$ $\leq$ fixp $f$
    by (rule lfp-lowerbound, subst fixp-unfold [OF $f'$], rule order-refl)
  show fixp $f$ $\leq$ lfp $f$
    by (rule fixp-lowerbound [OF $f'$], subst lfp-unfold [OF $f$], rule order-refl)

qed

hide-const (open) iterates fixp

end

49 Partial-Function: Partial Function Definitions

theory Partial-Function
imports Complete-Partial-Order Fun-Def-Base Option
keywords partial-function :: thy-decl
begin
49.1 Axiomatic setup

This technical locale contains the requirements for function definitions with ccpo fixed points.

**Definition**

fun-ord \( f \leftrightarrow_g (\forall x. \text{ord} (f x) (g x)) \)

fun-lub \( L A = (\lambda x. L \{y. \exists f \in A. y = f x\}) \)

img-ord \( f \leftrightarrow (\lambda xy. \text{ord} (f x) (f y)) \)

img-lub \( f g \leftrightarrow (\lambda A. \text{g} (\text{lub} (f \cdot A))) \)

**Lemma**

chain-fun:

assumes \( A: \text{chain} (\text{fun-ord} \text{ord}) A \)

shows \( \text{chain ord} \{y. \exists f \in A. y = f a\} (\text{is chain ?C}) \)

proof (rule chainI)

fix \( x y \) assume \( x \in ?C y \in ?C \)

then obtain \( f g \) where \( fg \) \( f \in A g \in A \)

and [simp]: \( x = f a y = g a \) by blast

from chainD [OF A fg]

show \( \text{ord} x y \lor \text{ord} y x \) unfolding fun-ord-def by auto

qed

**Lemma**

call-mono[partial-function-mono]: monotone (fun-ord ord) ord \((\lambda f. f t)\)

by (rule monotoneI) (auto simp: fun-ord-def)

**Lemma**

let-mono[partial-function-mono]:

\((\forall x. \text{monotone orda ordb} (\lambda f. b f x))\)

\(\Longrightarrow\) monotone orda ordb \((\lambda f. \text{Let} t (b f))\)

by (simp add: Let-def)

**Lemma**

if-mono[partial-function-mono]: monotone orda ordb \( F \)

\(\Longrightarrow\) monotone orda ordb \( G \)

\(\Longrightarrow\) monotone orda ordb \((\lambda f. \text{if } c \text{ then } F f \text{ else } G f)\)

unfolding monotone-def by simp

**Definition**

mk-less \( R = (\lambda xy. R x y \land \neg R y x) \)

**Locale**

partial-function-definitions =

fixes leq :: 'a => 'a => bool

fixes lub :: 'a set => 'a

assumes leq-refl: leq x x

assumes leq-trans: leq x y \(\Rightarrow\) leq y z \(\Rightarrow\) leq x z

assumes leq-antisym: leq x y \(\Rightarrow\) leq y x \(\Rightarrow\) x = y

assumes lub-upper: chain leq A \(\Longrightarrow\) x \(\in\) A \(\Longrightarrow\) leq x (lub A)

assumes lub-least: chain leq A \(\Longrightarrow\) \((\forall x. x \in A \Longrightarrow leq x z) \Longrightarrow leq (lub A) z\)

**Lemma**

partial-function-lift:

assumes partial-function-definitions ord lb
shows partial-function-definitions (fun-ord ord) (fun-lub lb) (is partial-function-definitions ordf lubf)
proof −
interpret partial-function-definitions ord lb by fact

show ?thesis
proof
  fix x show ?ordf x x
    unfolding fun-ord-def by (auto simp: leq-refl)
next
  fix x y z assume ?ordf x y ?ordf y z
  thus ?ordf x z unfolding fun-ord-def
    by (force dest: leq-trans)
next
  fix x y assume ?ordf x y ?ordf y x
  thus x = y unfolding fun-ord-def
    by (force intro!: dest: leq-antisym)
next
  fix A :: ('b ⇒ 'a) set and f :: 'b ⇒ 'a
  assume A: chain ?ordf A and f: ∀f. f ∈ A → ?ordf f g
  show ?ordf (lub A) g unfolding fun-lub-def fun-ord-def
    by (blast intro: lub-upper chain-fun[OF A] f)
next
  let ?iord = img-ord f ord
  let ?ilub = img-lub f g Lub
interpret partial-function-definitions ord Lub by fact
show ?thesis
proof
  fix A x assume chain ?iord A x ∈ A
  then have chain ord (f ∘ A) f x ∈ f ∘ A
by (auto simp: img-ord-def intro: chainI dest: chainD)
thus ?iord x (?iub A)
unfolding inv img-lub-def img-ord-def by (rule lub-upper)

next
fix A x assume chain ?iord A
and 1: \( \forall z. z \in A \implies ?iord z x \)
then have chain ord (\( f \) A)
by (auto simp: img-ord-def intro: chainI dest: chainD)
thus ?iord (?iub A) x
unfolding inv img-lub-def img-ord-def
by (rule lub-least) (auto dest: 1[unfolded img-ord-def])
qed (auto simp: img-ord-def intro: leq-refl dest: leq-trans leq-antisym inj)

qed

context partial-function-definitions
begin

abbreviation le-fun \( \equiv \) fun-ord leq
abbreviation lub-fun \( \equiv \) fun-lub lub
abbreviation fixp-fun \( \equiv \) ccpo.fixp lub-fun le-fun
abbreviation mono-body \( \equiv \) monotone le-fun leq
abbreviation admissible \( \equiv \) ccpo.admissible lub-fun le-fun

Interpret manually, to avoid flooding everything with facts about orders

lemma ccpo: class.ccpo lub-fun le-fun (\( \mathit{mk-less} \) le-fun)
apply (rule ccpo)
apply (rule partial-function-lift)
apply (rule partial-function-definitions-axioms)
done

The crucial fixed-point theorem

lemma mono-body-fixp:
(\( \forall x. \) mono-body (\( \lambda f. F f x \)) \implies fixp-fun F = F (fixp-fun F)
by (rule ccpo.fixp-unfold[OF ccpo]) (auto simp: monotone-def fun-ord-def)

Version with curry/uncurry combinators, to be used by package

lemma fixp-rule-ac:
fixes F :: 'c \Rightarrow 'c
U :: 'c \Rightarrow 'b \Rightarrow 'a
C :: ('b \Rightarrow 'a) \Rightarrow 'c
assumes mono: \( \forall x. \) mono-body (\( \lambda f. U (F (C f)) \) x)
assumes eq: f \equiv C (fixp-fun (\( \lambda f. U (F (C f)))))
assumes inverse: \( \forall f. C (U f) = f \)
shows f = F f

proof -
  have f = C (fixp-fun (\( \lambda f. U (F (C f))))) by (simp add: eq)
  also have ... = C (U (F (C (fixp-fun (\( \lambda f. U (F (C f))))))))
    by (subst mono-body-fixp[OF \%f. U (F (C f)), OF mono]) (rule refl)
  also have ... = F (C (fixp-fun (\( \lambda f. U (F (C f))))) by (rule inverse)
also have \( \ldots = F \, f \) by \((\text{simp add: eq})\)

finally show \( f = F \, f \).

qed

Fixpoint induction rule

\textbf{lemma} \texttt{fixp-induct-uc}: \\
\textbf{fixes} \( F :: \texttt{'}c \Rightarrow \texttt{'}c \) \textbf{and} \\
\( U :: \texttt{'}c \Rightarrow \texttt{'}b \Rightarrow \texttt{'}a \) \textbf{and} \\
\( C :: (\texttt{'}b \Rightarrow \texttt{'}a) \Rightarrow \texttt{'}c \) \textbf{and} \\
\( P :: (\texttt{'}b \Rightarrow \texttt{'}a) \Rightarrow \text{bool} \)

\textbf{assumes} \( \text{mono} \) : \( \forall x. \text{mono-body} (\lambda f. U \, (F \, (C \, f)) \, x) \)

\textbf{assumes} \( \text{eq} \) : \( f \equiv C \, (\text{fixp-fun} (\lambda f. U \, (F \, (C \, f)))) \)

\textbf{assumes} \( \text{inverse} \) : \( \forall f. U \, (C \, f) = f \)

\textbf{assumes} \( \text{adm} \) : \( \text{ccpo. admissible lub-fun le-fun P} \)

\textbf{and} \( \text{bot} \) : \( P \, (\lambda _. \text{lub} \, \{\} ) \)

\textbf{assumes} \( \text{step} \) : \( \forall f. P \, (U \, f) \Longrightarrow P \, (U \, (F \, f)) \)

\textbf{shows} \( P \, (U \, f) \)

unfolding \( \text{eq} \) \( \text{inverse} \)

apply \((\text{rule ccpo.fixp-induct[OF ccpo adm]})\)

apply \((\text{insert mono, auto simp: monotone-def fun-ord-def bot fun-lub-def}[2])\)

by \((\text{rule-tac f=C x in step, simp add: inverse})\)

Rules for \( \text{mono-body} \):

\textbf{lemma} \( \text{const-mono}[\text{partial-function-mono}] \): \( \text{monotone ord leq} (\lambda f. c) \)

by \((\text{rule monotoneI}) \) \((\text{rule leq-refl})\)

end

\section{49.2 Flat interpretation: tailrec and option}

\textbf{definition} \\
\( \text{flat-ord} b \, x \, y \iff x = b \lor x = y \)

\textbf{definition} \\
\( \text{flat-lub} b \, A = (\text{if} \, A \subseteq \{b\} \, \text{then} \, b \, \text{else} \, \text{THE} \, x. \, x \in A - \{b\}) \)

\textbf{lemma} \( \text{flat-interpretation} \): \\
(\text{partial-function-definitions (flat-ord b) (flat-lub b)})

\textbf{proof} \\
\textbf{fix} \( A \, x \) \textbf{assume} \( I : \text{chain} \, (\text{flat-ord b}) \, A \, x \in A \)

\textbf{show} \( \text{flat-ord b} \, x \, (\text{flat-lub b} \, A) \)

\textbf{proof cases} \\
\textbf{assume} \( x = b \) \\
\textbf{thus} \( \text{?thesis by (simp add: flat-ord-def)} \)

\textbf{next} \\
\textbf{assume} \( x \neq b \) \\
\textbf{with} \( f \) \textbf{have} \( A - \{b\} = \{x\} \) \\
\textbf{by} \((\text{auto elim: chainE simp: flat-ord-def})\)

\textbf{then have} \( \text{flat-lub b} \, A \, x \)
by (auto simp: flat-lub-def)
thus ?thesis by (auto simp: flat-ord-def)
qed

next
fix A z assume A: chain (flat-ord b) A
and z: ∀x. x ∈ A → flat-ord b x z
show flat-ord b (flat-lub b A) z
proof cases
assume A ⊆ {b}
thus ?thesis
by (auto simp: flat-lub-def flat-ord-def)
next
assume nb: ¬ A ⊆ {b}
then obtain y where y: y ∈ A y ≠ b by auto
with A have A − {b} = {y}
by (auto elim: chainE simp: flat-ord-def)
with nb have flat-lub b A = y
by (auto simp: flat-lub-def)
with z y show ?thesis by auto
qed

interpretation tailrec!: partial-function-definitions flat-ord undefined flat-lub undefined
where flat-lub undefined {} ≡ undefined
by (rule flat-interpretation)(simp add: flat-lub-def)

interpretation option!: partial-function-definitions flat-ord None flat-lub None
where flat-lub None {} ≡ None
by (rule flat-interpretation)(simp add: flat-lub-def)

abbreviation tailrec-ord ≡ flat-ord undefined
abbreviation mono-tailrec ≡ monotone (fun-ord tailrec-ord) tailrec-ord

lemma tailrec-admissible:
ccpo.admissible (fun-lub (flat-lub c)) (fun-ord (flat-ord c))
(∀a. ∀x. a x ≠ c → P x (a x))
proof(intro ccpo.admissibleI strip)
fix A x
assume chain: Complete-Partial-Order.chain (fun-ord (flat-ord c)) A
and P [rule-format]: ∀f∈A. ∀x. f x ≠ c → P x (f x)
and defined: fun-lub (flat-lub c) A x ≠ c
from defined obtain f where f: f ∈ A f x ≠ c
by(auto simp add: fun-lub-def flat-lub-def split: split-if-asm)
hence P x (f x) by(rule P)
moreover from chain f have ∀f′ ∈ A. f′ x = c ∨ f′ x = f x
by(auto 4 4 simp add: Complete-Partial-Order.chain-def flat-ord-def fun-ord-def)
hence \( \text{fun-lub} (\text{flat-lub} \ c) \ A \ x = f \ x \)

using \( f \) by (auto simp add: fun-lub-def flat-lub-def)

ultimately show \( P \ x \ (\text{fun-lub} (\text{flat-lub} \ c) \ A \ x) \) by simp

qed

lemma fixp-induct-tailrec:

fixes \( F :: \ 'c \Rightarrow \ 'c \) and
\( U :: \ 'c \Rightarrow \ 'b \Rightarrow \ 'a \) and
\( C :: (\ 'b \Rightarrow \ 'a) \Rightarrow \ 'c \) and
\( P :: \ 'b \Rightarrow \ 'a \Rightarrow \ \text{bool} \) and
\( x :: \ 'b \)

assumes mono: \( \forall x. \text{monotone} (\text{fun-ord} (\text{flat-ord} \ c)) (\text{flat-ord} \ c) (\lambda f. \ U (F (C f))) x \)

assumes eq: \( f \equiv C \ (\text{ccpo.fixp} (\text{fun-lub} (\text{flat-lub} \ c)) (\text{fun-ord} (\text{flat-ord} \ c))) (\lambda f. \ U (F (C f))) \)

assumes inverse2: \( \lambda f. \ U (C (\text{ccpo.fixp} (\text{fun-lub} (\text{flat-lub} \ c)) (\text{fun-ord} (\text{flat-ord} \ c)))) (\lambda f. \ U (F (C f))) f \)

assumes step: \( \lambda f \ x \ y. \ (\lambda x \ y. \ U f x = y \Rightarrow y \neq c \Rightarrow P x y) \Rightarrow U (F f) x = y \Rightarrow y \neq c \Rightarrow P x y \)

assumes result: \( U f x = y \)

assumes defined: \( y \neq c \)

shows \( P \ x \ y \)

proof

have \( \forall x \ y. \ U f x = y \Rightarrow y \neq c \Rightarrow P x y \)

by (rule partial-function-definitions.fixp-induct-uc [OF flat-interpretation, of - U F C, OF mono eq inverse2])

(auto intro: step tailrec-admissible simp add: fun-lub-def flat-lub-def)

thus \( \text{thesis} \) using result defined by blast

qed

lemma admissible-image:

assumes pfun: partial-function-definitions le lub

assumes adm: ccpo.admissible lub le (P o g)

assumes inj: \( \forall x \ y. \ f x = f y \Rightarrow x = y \)

assumes inv: \( \forall x. \ f (g x) = x \)

shows ccpo.admissible (img-lub f g lub) (img-ord f le) P

proof (rule ccpo.admissibleI)

fix A assume chain: (img-ord f le) A
then have ch': chain le (f ' A)

by (auto simp: img-ord-def intro: chainI dest: chainD)

assume A \( \neq \) \{\}

assume P-A: \( \forall x \in A. \ P x \)

have (P o g) (lub (f ' A)) using adm ch'

proof (rule ccpo.admissibleD)

fix x assume x \( \in \) f ' A

with P-A show (P o g) x by (auto simp: inj[OF inv])

qed (simp add: A \( \neq \) \{}\)

thus P (img-lub f g lub A) unfolding img-lub-def by simp

qed
lemma admissible-fun:
  assumes pfun: partial-function-definitions le lub
  assumes adm: \( \forall x. \text{cpo.admissible} \, \text{lub} \, \text{le} \, (Q \, x) \)
  shows \( \text{cpo.admissible} \, (\text{fun-lub} \, \text{lub}) \, (\text{fun-ord} \, \text{le}) \, (\lambda f. \forall x. Q \, x \, (f \, x)) \)
proof (rule ccpo.admissibleI)
  fix \( A :: (b \Rightarrow a) \, \text{set} \)
  assume Q: \( \forall f \in A. \forall x. Q \, x \, (f \, x) \)
  assume ch: chain (fun-ord le A)
  assume A \( \neq \) \( \{ \} \)
  hence non-empty: \( \forall a. \exists f \in A. y = f \, a \) \( \neq \) \( \{ \} \) by auto
  show \( \forall x. Q \, x \, (\text{fun-lub} \, \text{lub} \, A \, x) \)
  unfolding fun-lub-def by (rule allI, rule ccpo.admissibleD[OF adm chain-fun[OF ch] non-empty])
  (auto simp: Q)
qed

abbreviation option-ord \( \equiv \) flat-ord None
abbreviation mono-option \( \equiv \) monotone (fun-ord option-ord) option-ord

lemma bind-mono[partial-function-mono]
  assumes mf: mono-option B and mg: \( \forall y. \text{mono-option} \, (\lambda f. C \, y \, f) \)
  shows \( \text{mono-option} \, (\lambda f. \text{Option.bind} \, (B \, f) \, (\lambda y. C \, y \, f)) \)
proof (rule monotoneI)
  fix f g :: \( \Rightarrow \, \text{option} \, \text{option} \, a \Rightarrow \, \text{option} \, \text{option} \, b \)
  assume fg: fun-ord option-ord f g
  with mf have option-ord (B f) (B g) by (rule monotoneD[of - - f g])
  then have option-ord (Option.bind (B f) (\lambda y. C y f)) (Option.bind (B g) (\lambda y. C y f))
  unfolding flat-ord-def by auto
  also from mg
  have \( \forall y'. \, \text{option-ord} \, (C \, y' \, f) \, (C \, y' \, g) \)
  by (rule monotoneD)[of - - fg]
  then have option-ord (Option.bind (B g) (\lambda y'. C y' f)) (Option.bind (B g) (\lambda y'. C y' g))
  unfolding flat-ord-def by (cases B g) auto
  finally (option.leq-trans)
  show option-ord (Option.bind (B f) (\lambda y. C y f)) (Option.bind (B g) (\lambda y'. C y' g)) .
qed

lemma flat-lub-in-chain:
  assumes ch: chain (flat-ord b) A
  assumes lub: flat-lub b A = a
  shows a = b \( \lor \) a \( \in \) A
proof (cases A \( \subseteq \) \( \{ b \} \))
  case True
  then have flat-lub b A = b unfolding flat-lub-def by simp
  with lub show ?thesis by simp
next
case False
then obtain c where c ∈ A and c ≠ b by auto
{ fix z assume z ∈ A
  from chain[D[OF ch |c ∈ A| this] have z = c ∨ z = b
  unfolding flat-ord-def using \( c ≠ b \) by auto }
with False have A = { b } = { c } by auto
with False have flat-lub b A = c by (auto simp: flat-lub-def)
with \( c ∈ A \): lub show \( \text{thesis} \) by simp
qed

lemma option-admissible: option.admissible \( (%(f::'a ⇒ 'b option). \( \forall x y. f x = Some y → P x y )\)
proof (rule ccpo.admissible)
fix A :: \( ('a ⇒ 'b option) \) set
assume ch: chain option.le-fun A
and IH: \( ∀ f ∈ A. \forall x y. f x = Some y → P x y \)
from ch have ch': \( ∀ x. \exists f ∈ A. y = f x \) by (rule chain-fun)
show \( \forall x y. \text{option}.\text{lub-fun} A x = Some y → P x y \)
proof (intro allI impI)
fix x y assume \( \text{option}.\text{lub-fun} A x = Some y \)
from \( \text{flat-lub-in-chain} [OF ch' this[unfolded fun-lub-def]] \)
have \( \exists f ∈ A. f x = Some y \) by simp
then have \( \exists f ∈ A. f x = Some y \) by auto
with IH show \( P x y \) by auto
qed
qed

lemma fixp-induct-option:
fixes F :: \( 'c ⇒ 'c \) and
U :: \( 'c ⇒ 'b ⇒ 'a option \) and
C :: \( ('b ⇒ 'a option) ⇒ 'c \) and
P :: \( 'b ⇒ 'a ⇒ bool \)
assumes mono: \( λ x. \text{mono-option} (λ f. U (F (C f))) \)
assumes eq: \( f ≡ C (ccpo.fixp (\text{fun-lub} (\text{flat-lub None})) (\text{fun-ord} \text{option-ord}) (λ f. U (F (C f)))) )\)
assumes inverse2: \( λ f. U (C f) x = f \)
assumes step: \( λ f x y. (λ x y. U f x = Some y → P x y ) → U (F f) x = Some y → P x y \)
assumes defined: \( U f x = Some y \)
shows \( P x y \)
using step defined option.fixp-induct-uc[of \( U F C \), \( \text{OF} \) mono eq inverse2 option-admissible]
unfolding \( \text{fun-lub-def} \text{flat-lub-def} \) by(auto 9 2)

declaration ⟨⟨ Partial-Function.init tailrec @(\{term tailrec.fixp-fun\})
@\{term tailrec.mono-body\} @(\{thm tailrec.fixp-rule-uc\} @\{thm tailrec.fixp-induct-uc\}
(SOME @\{thm fixp-induct-tailrec[where c=undefined]\})) ⟩⟩

declaration ⟨⟨ Partial-Function.init option @(\{term option.fixp-fun\}) ⟩⟩
50 SAT: Reconstructing external resolution proofs for propositional logic

theory SAT
imports HOL
begin

ML-file Tools/prop-logic.ML
ML-file Tools/sat-solver.ML
ML-file Tools/sat.ML

method-setup sat = (\ Scan.succeed (SIMPLE-METHOD' o SAT.sat-tac) \)
SAT solver

method-setup satx = (\ Scan.succeed (SIMPLE-METHOD' o SAT.satx-tac) \)
SAT solver (with definitional CNF)

end

51 Fun-Def: Function Definitions and Termination Proofs

theory Fun-Def
imports Partial-Function SAT
keywords function termination :: thy-goal and fun fun-cases :: thy-decl
begin

51.1 Definitions with default value

definition
THE-default :: 'a ⇒ ('a ⇒ bool) ⇒ 'a where
THE-default d P = (if (∃!x. P x) then (THE x. P x) else d)

lemma THE-defaultI: (∃!x. P x) ⇒ THE-default d P
by (simp add: theI' THE-default-def)

lemma THE-default1-equality:
[∃!x. P x; P a] ⇒ THE-default d P = a
by (simp add: the1-equality THE-default-def)
lemma THE-default-none:
\(\neg (\exists! x. \ P x) \Rightarrow \text{THE-default} \ d \ P = d\)
by (simp add:THE-default-def)

lemma fundef-ex1-existence:
assumes f-def: \(f == (\lambda x::'a. \ \text{THE-default} \ (d \ x) \ (\lambda y. \ G \ x \ y))\)
assumes ex1: \(\exists! y. \ G \ x \ y\)
shows \(G \ x \ (f \ x)\)
apply (simp only: f-def)
apply (rule THE-defaultI')
apply (rule ex1)
done

lemma fundef-ex1-uniqueness:
assumes f-def: \(f == (\lambda x::'a. \ \text{THE-default} \ (d \ x) \ (\lambda y. \ G \ x \ y))\)
assumes ex1: \(\exists! y. \ G \ x \ y\)
assumes elm: \(G \ x \ (h \ x)\)
shows \(h \ x = f \ x\)
apply (simp only: f-def)
apply (rule THE-default1-equality [symmetric])
apply (rule ex1)
apply (rule elm)
done

lemma fundef-ex1-iff:
assumes f-def: \(f == (\lambda x::'a. \ \text{THE-default} \ (d \ x) \ (\lambda y. \ G \ x \ y))\)
assumes ex1: \(\exists! y. \ G \ x \ y\)
shows \(\ (G \ x \ y) = (f \ x = y)\)
apply (auto simp:ex1 f-def THE-default1-equality)
apply (rule THE-defaultI')
apply (rule ex1)
done

lemma fundef-default-value:
assumes f-def: \(f == (\lambda x::'a. \ \text{THE-default} \ (d \ x) \ (\lambda y. \ G \ x \ y))\)
assumes graph: \(\forall x y. \ G \ x \ y \Rightarrow D \ x\)
assumes \(\neg D \ x\)
shows \(f \ x = d \ x\)
proof -
have \(\neg (\exists y. \ G \ x \ y)\)
proof
  assume \(\exists y. \ G \ x \ y\)
hence \(D \ x\) using graph ..
  with \(\neg D \ x\) show False ..
qed
hence \(\neg (\exists y. \ G \ x \ y)\) by blast
thus \(\text{thesis}\)
THEORY “Fun-Def”

unfolding f-def
  by (rule THE-default-none)
qed

definition in-rel-def[simp]:
in-rel R x y == (x, y) ∈ R

lemma wf-in-rel:
wf R ⇒ wfP (in-rel R)
  by (simp add: wfP-def)

ML-file Tools/Function/function-core.ML
ML-file Tools/Function/mutual.ML
ML-file Tools/Function/pattern-split.ML
ML-file Tools/Function/relation.ML
ML-file Tools/Function/function-elims.ML

method-setup relation = ⟨⟨
  Args.term >> (fn t => fn ctxt => SIMPLE-METHOD' (Function-Relation.relation-infer-tac ctxt t))
⟩⟩ prove termination using a user-specified wellfounded relation

ML-file Tools/Function/function.ML
ML-file Tools/Function/pat-completeness.ML

method-setup pat-completeness = ⟨⟨
  Scan.succeed (SIMPLE-METHOD' o Pat-Completeness.pat-completeness-tac)
⟩⟩ prove completeness of datatype patterns

ML-file Tools/Function/func.ML
ML-file Tools/Function/induction-schema.ML

method-setup induction-schema = ⟨⟨
  Scan.succeed (RAW-METHOD o Induction-Schema.induction-schema-tac)
⟩⟩ prove an induction principle

setup ⟨⟨
  Function.setup
  #> Function-Fun.setup
⟩⟩

51.2 Measure functions

inductive is-measure :: ('a ⇒ nat) ⇒ bool
  where is-measure-trivial: is-measure f

ML-file Tools/Function/measure-functions.ML
setup MeasureFunctions.setup
THEORY "Fun-Def"

lemma measure-size [measure-function]: is-measure size
by (rule is-measure-trivial)

lemma measure-fst [measure-function]: is-measure f \implies is-measure (\lambda p. f (fst p))
by (rule is-measure-trivial)
lemma measure-snd [measure-function]: is-measure f \implies is-measure (\lambda p. f (snd p))
by (rule is-measure-trivial)

ML-file Tools/Function/lexicographic-order.ML

method-setup lexicographic-order = ⟨⟨
Method.sections clasimp-modifiers >>
(K (SIMPLE-METHOD o Lexicographic-Order. lexicographic-order-tac false))
⟩⟩ termination prover for lexicographic orderings

setup Lexicographic-Order.setup

51.3 Congruence rules

lemma let-cong [fundef-cong]:
M = N \implies (\forall x. x = N \implies f x = g x) \implies Let M f = Let N g
unfolding Let-def by blast

lemmas [fundef-cong] =
if-cong image-cong INF-cong SUP-cong
bex-cong ball-cong imp-cong map-option-cong Option.bind-cong

lemma split-cong [fundef-cong]:
(\forall x y. (x, y) = q \implies f x y = g x y) \implies p = q
\implies split f p = split g q
by (auto simp: split-def)

lemma comp-cong [fundef-cong]:
f (g x) = f' (g' x') \implies (f o g) x = (f' o g') x'
unfolding o-apply .

51.4 Simp rules for termination proofs

declare
trans-less-add1 [termination-simp]
trans-less-add2 [termination-simp]
trans-le-add1 [termination-simp]
trans-le-add2 [termination-simp]
less-imp-le-nat [termination-simp]
le-imp-less-Suc [termination-simp]

lemma size-prod-simp [termination-simp]:
size-prod f g p = f (fst p) + g (snd p) + Suc 0
by (induct p) auto
51.5 Decomposition

**Lemma** less-by-empty:
\[ A = \{} \Rightarrow A \subseteq B \]

**And** union-comp-emptyL:
\[ [ A \cup C = \{} ; B \cup C = \{} ] \Rightarrow (A \cup B) \cup C = \{} \]

**And** union-comp-emptyR:
\[ [ A \cup B = \{} ; A \cup C = \{} ] \Rightarrow A \cup (B \cup C) = \{} \]

**And** wf-no-loop:
\[ R \cup R = \{} \Rightarrow \text{wf } R \]

**By** (auto simp add: wf-comp-self[of R])

51.6 Reduction pairs

**Definition**
reduction-pair P = (wf (fst P) \land \text{fst } \text{P} \cup \text{snd } \text{P} \subseteq \text{fst } \text{P})

**Lemma** reduction-pairI[intro]: \( \text{wf } R \Rightarrow R \cup S \subseteq R \Rightarrow \text{reduction-pair } (R, S) \)

**Unfolding** reduction-pair-def by auto

**Lemma** reduction-pair-lemma:
- assumes rp: reduction-pair P
- assumes R \subseteq \text{fst } \text{P}
- assumes S \subseteq \text{snd } \text{P}
- assumes \text{wf } S
- shows \text{wf } (R \cup S)

**Proof** —
- from \( \text{rp} : S \subseteq \text{snd } \text{P} \) have \( \text{wf } (\text{fst } \text{P}) \text{fst } \text{P} \cup S \subseteq \text{fst } \text{P} \)
  unfolding reduction-pair-def by auto
- with \( \text{wf } S \) have \( \text{wf } (\text{fst } \text{P} \cup S) \)
  by (auto intro: wf-union-compatible)
- moreover from \( \text{R} \subseteq \text{fst } \text{P} \) have \( R \cup S \subseteq \text{fst } \text{P} \cup S \) by auto
- ultimately show ?thesis by (rule wf-subset)

**Qed**

**Definition**
\( \text{rp-inv-image } = (\lambda (R,S). f. (\text{inv-image } R f, \text{inv-image } S f)) \)

**Lemma** rp-inv-image-rp:
reduction-pair P \Rightarrow reduction-pair (rp-inv-image P f)

**Unfolding** reduction-pair-def rp-inv-image-def split-def by force

51.7 Concrete orders for SCNP termination proofs

**Definition** pair-less = less-than <*lex*> less-than
**Definition** pair-leq = pair-less \^=
**Definition** max-strict = max-ext pair-less
**Definition** max-weak = max-ext pair-leq \cup \{\{\}, \{\}\}
**Definition** min-strict = min-ext pair-less
definition min-weak = min-ext pair-leq \cup \{\{\},\{\}\} \\

lemma wf-pair-less[simp]: wf pair-less 
by (auto simp: pair-less-def)

Introduction rules for pair-less/pair-leq

lemma pair-leqI1: \( a < b \Rightarrow ((a, s), (b, t)) \in \text{pair-leq} \)
and pair-leqI2: \( a \leq b \Rightarrow s \leq t \Rightarrow ((a, s), (b, t)) \in \text{pair-leq} \)
and pair-lessI1: \( a < b \Rightarrow ((a, s), (b, t)) \in \text{pair-less} \)
and pair-lessI2: \( a \leq b \Rightarrow s < t \Rightarrow ((a, s), (b, t)) \in \text{pair-less} \)
unfolding pair-leq-def pair-less-def by auto

Introduction rules for max

lemma smax-emptyI:
finite \( Y \) \( \Rightarrow \) \( Y \neq \{\} \Rightarrow (\{\}, Y) \in \text{max-strict} \)
and smax-insertI:
\( y \in Y; (x, y) \in \text{pair-less}; (X, Y) \in \text{max-strict} \) \( \Rightarrow \) (insert \( x \) \( X \), \( Y \)) \( \in \text{max-strict} \)
and wmax-emptyI:
finite \( X \) \( \Rightarrow \) \( (\{\}, X) \in \text{max-weak} \)
and wmax-insertI:
\( y \in YS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{max-weak} \) \( \Rightarrow \) (insert \( x \) \( XS \), \( YS \)) \( \in \text{max-weak} \)
unfolding max-strict-def max-weak-def by (auto elim!: max-ext.cases)

Introduction rules for min

lemma smin-emptyI:
\( X \neq \{\} \Rightarrow (X, \{\}) \in \text{min-strict} \)
and smin-insertI:
\( x \in XS; (x, y) \in \text{pair-less}; (XS, YS) \in \text{min-strict} \) \( \Rightarrow \) (\( XS \), insert \( y \) \( YS \)) \( \in \text{min-strict} \)
and wmin-emptyI:
\( (X, \{\}) \in \text{min-weak} \)
and wmin-insertI:
\( x \in XS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{min-weak} \) \( \Rightarrow \) (\( XS \), insert \( y \) \( YS \)) \( \in \text{min-weak} \)
by (auto simp: min-strict-def min-weak-def min-ext-def)

Reduction Pairs

lemma max-ext-compat:
assumes \( R O S \subseteq R \)
shows \( \text{max-ext} R O (\text{max-ext} S \cup \{\{\},\{\}\}) \subseteq \text{max-ext} R \)
using assms
apply auto
apply (elim max-ext.cases)
apply rule
apply auto[3]
apply (drule-tac x=xa in meta-spec)
apply simp
apply (erule bexE)
apply (drule-tac x=xb in meta-spec)
by auto

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
apply (intro reduction-pairI max-ext-wf)
apply simp
apply (rule max-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

lemma min-ext-compat:
  assumes R O S ⊆ R
  shows min-ext R O (min-ext S ∪ {{{}, {}}}) ⊆ min-ext R
using assms
apply (auto simp: min-ext-def)
apply (erule bexE)
apply (drule-tac x=ya in bspec, assumption)
apply (erule bexE)
apply (drule-tac x=xc in bspec)
apply assumption
by auto

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
apply (intro reduction-pairI min-ext-wf)
apply simp
apply (rule min-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

51.8 Tool setup
ML-file Tools/Function/termination.ML
ML-file Tools/Function/scnp-solve.ML
ML-file Tools/Function/scnp-reconstruct.ML
ML-file Tools/Function/fun-cases.ML

setup ScnpReconstruct.setup

ML-val — setup inactive
  ⟨ ⟨ Context.theory-map (Function-Common.set-termination-prover
    (ScnpReconstruct.decomp-scnp-tac [ScnpSolve.MAX, ScnpSolve.MIN, ScnpSolve.MS])) ⟩ ⟩
end
52 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory Int
imports Equiv-Relations Power Quotient Fun-Def
begin

52.1 Definition of integers as a quotient type

definition intrel :: (nat × nat) ⇒ (nat × nat) ⇒ bool where
intrel = (λ(x, y) (u, v). x + v = u + y)

lemma intrel-iff [simp]: intrel (x, y) (u, v) ←→ x + v = u + y
  by (simp add: intrel-def)

quotient-type int = nat × nat / intrel
morphisms Rep-Integ Abs-Integ
proof (rule equivpI)
  show reflp intrel
    unfolding reflp-def by auto
  show symp intrel
    unfolding symp-def by auto
  show transp intrel
    unfolding transp-def by auto
qed

lemma eq-Abs-Integ [case-names Abs-Integ, cases type: int]:
  (!!x y. z = Abs-Integ (x, y) ==> P) ==> P
by (induct z) auto

52.2 Integers form a commutative ring

instantiation int :: comm-ring-1
begin

lift-definition zero-int :: int is (0, 0).

lift-definition one-int :: int is (1, 0).

lift-definition plus-int :: int ⇒ int ⇒ int
  is λ(x, y) (u, v). (x + u, y + v)
  by clarsimp

lift-definition uminus-int :: int ⇒ int
  is λ(x, y). (y, x)
  by clarsimp

lift-definition minus-int :: int ⇒ int ⇒ int
  is λ(x, y) (u, v). (x + v, y + u)
THEORY “Int”  

by clarsimp

lift-definition times-int :: int ⇒ int ⇒ int
is λ(x, y) (u, v). (x*u + y*v, x*v + y*u)

proof (clarsimp)
  fix s t u v w x y z :: nat
  assume s + v = u + t and w + z = y + x
  hence (s + v) * w + (u + t) * x + u * (w + z) + v * (y + x)
  = (u + t) * w + (s + v) * x + u * (y + x) + v * (w + z)
    by simp
  thus (s * w + t * x) + (u * z + v * y) = (u * y + v * z) + (s * x + t * w)
    by (simp add: algebra-simps)
qed

instance
  by default (transfer, clarsimp simp: algebra-simps)+

end

abbreviation int :: nat ⇒ int where
  int ≡ of-nat

lemma int-def: int n = Abs-Integ (n, 0)
  by (induct n, simp add: zero-int.abs-eq, simp add: one-int.abs-eq plus-int.abs-eq)

lemma int-transfer [transfer-rule]:
  (rel-fun (op =) pcr-int) (λn. (n, 0)) int
unfolding rel-fun-def int.pcr-cr-eq cr-int-def int-def by simp

lemma int-diff-cases:
  obtains (diff) m n where z = int m - int n
  by transfer clarsimp

52.3 Integers are totally ordered

instantiation int :: linorder
begin

lift-definition less-eq-int :: int ⇒ int ⇒ bool
is λ(x, y) (u, v). x + v ≤ u + y
  by auto

lift-definition less-int :: int ⇒ int ⇒ bool
is λ(x, y) (u, v). x + v < u + y
  by auto

instance
  by default (transfer, force)+
end

instantiation int :: distrib-lattice
begin

definition
\((\text{inf} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int}) = \text{min}\)

definition
\((\text{sup} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int}) = \text{max}\)

instance
by intro-classes
\((\text{auto simp add: inf-int-def sup-int-def max-min-distrib2})\)
end

52.4 Ordering properties of arithmetic operations

instance int :: ordered-cancel-ab-semigroup-add

proof
fix \(i\ j\ k :: \text{int}\)

show \(i \leq j \implies k + i \leq k + j\)
by transfer clarsimp

qed

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on \(k\); 0

lemma zmult-zless-mono2-lemma:
\((i :: \text{int}) < j =\Rightarrow 0 < k =\Rightarrow \text{int} k * i < \text{int} k * j\)
apply (induct \(k\))
apply simp
apply (simp add: distrib-right)
apply (case-tac \(k = 0\))
apply (simp-all add: add-strict-mono)
done

lemma zero-le-imp-eq-int: \((0 :: \text{int}) \leq k =\Rightarrow \exists n. k = \text{int} n\)
apply transfer
apply clarsimp
apply (rule-tac \(x = a - b\ in\ exI, simp\))
done

lemma zero-less-imp-eq-int: \((0 :: \text{int}) < k =\Rightarrow \exists n > 0. k = \text{int} n\)
apply transfer
apply clarsimp
apply (rule-tac \(x = a - b\ in\ exI, simp\))
done
lemma zmult-zless-mono2: \[ | i < j; (0::int) < k | \implies k \cdot i < k \cdot j \]
apply (drule zero-less-imp-eq-int)
apply (auto simp add: zmult-zless-mono2-lemma)
done

The integers form an ordered integral domain

instantiation int :: linordered-idom
begin

definition zabs-def: \(|i::int| = (if i < 0 then \neg i else i) \]
definition zsgn-def: \(\text{sgn } (i::int) = (if i=0 then 0 else if 0<i then 1 else -1) \)

instance proof
fix i j k :: int
show \(i < j \implies 0 < k \implies k \cdot i < k \cdot j \)
  by (rule zmult-zless-mono2)
show \(|i| = (if i < 0 then \neg i else i) \)
  by (simp only: zabs-def)
show \(\text{sgn } (i::int) = (if i=0 then 0 else if 0<i then 1 else -1) \)
  by (simp only: zsgn-def)
qed

end

lemma zless-imp-add1-zle: \( w < z \implies w + (1::int) \leq z \)
by transfer clarsimp

lemma zless-iff-Suc-zadd:
  \((w :: int) < z \iff (\exists n. z = w + \text{int } (\text{Suc } n))\)
apply transfer
apply auto
apply (rename-tac a b c d)
apply (rule-tac x=c+b = Suc(a+d) in exI)
apply arith
done

lemmas int-distrib =
  distrib-right [of z1 z2 w]
  distrib-left [of w z1 z2]
  left-diff-distrib [of z1 z2 w]
  right-diff-distrib [of w z1 z2]
for z1 z2 w :: int
52.5 Embedding of the Integers into any ring-1: of-int

context ring-1
begin

lift-definition of-int :: int ⇒ 'a is λ(i, j). of-nat i − of-nat j
by (clarsimp simp add: diff-eq-eq eq-diff-eq diff-add-eq

lemma of-int-0 [simp]: of-int 0 = 0
by transfer simp

lemma of-int-1 [simp]: of-int 1 = 1
by transfer simp

lemma of-int-add [simp]: of-int (w+z) = of-int w + of-int z
by transfer (clarsimp simp add: algebra-simps)

lemma of-int-minus [simp]: of-int (−z) = − (of-int z)
by (transfer fixing: uminus) clarsimp

lemma of-int-diff [simp]: of-int (w−z) = of-int w − of-int z
using of-int-add [of w−z] by simp

lemma of-int-mult [simp]: of-int (w*z) = of-int w * of-int z
by (transfer fixing: times) (clarsimp simp add: algebra-simps of-int-mult)

Collapse nested embeddings

lemma of-int-of-nat-eq [simp]: of-int (int n) = of-nat n
by (induct n) auto

lemma of-int-numeral [simp, code-post]: of-int (numeral k) = numeral k
by (simp add: of-nat-numeral [symmetric] of-int-of-nat-eq [symmetric])

lemma of-int-neg-numeral [code-post]: of-int (− numeral k) = − numeral k
by simp

lemma of-int-power:
of-int (z ^ n) = of-int z ^ n
by (induct n) simp-all

end

context ring-char-0
begin

lemma of-int-eq-iff [simp]:
of-int w = of-int z ⟷ w = z
by transfer (clarsimp simp add: algebra-simps
THEORY “Int”

Special cases where either operand is zero

lemma of-int-eq-0-iff [simp]:
\[ \text{of-int } z = 0 \iff z = 0 \]
using of-int-eq-iff [of z 0] by simp

lemma of-int-0-eq-iff [simp]:
\[ 0 = \text{of-int } z \iff z = 0 \]
using of-int-eq-iff [of 0 z] by simp

deleted context linordered-idom

begin

Every linordered-idom has characteristic zero.

subclass ring-char-0 ..

lemma of-int-le-iff [simp]:
\[ \text{of-int } w \leq \text{of-int } z \iff w \leq z \]
by (transfer fixing: less-eq) (clarsimp simp add: algebra-simps

lemma of-int-less-iff [simp]:
\[ \text{of-int } w < \text{of-int } z \iff w < z \]
by (simp add: less-le order-less-le)

lemma of-int-0-le-iff [simp]:
\[ 0 \leq \text{of-int } z \iff 0 \leq z \]
using of-int-le-iff [of 0 z] by simp

lemma of-int-0-less-iff [simp]:
\[ 0 < \text{of-int } z \iff 0 < z \]
using of-int-less-iff [of 0 z] by simp

lemma of-int-less-0-iff [simp]:
\[ \text{of-int } z < 0 \iff z < 0 \]
using of-int-less-0-iff [of z 0] by simp

end

lemma of-nat-less-of-int-iff:
\[ (\text{of-nat } n :: 'a::linordered-idom) < \text{of-int } x \iff \text{int } n < x \]
by (metis of-int-of-nat-eq of-int-less-iff)

lemma of-int-eq-id [simp]: of-int = id
proof
  fix z show of-int z = id z
  by (cases z rule: int-diff-cases, simp)
qed

instance int :: no-top
  apply default
  apply (rule-tac x="x + 1" in exI)
  apply simp
  done

instance int :: no-bot
  apply default
  apply (rule-tac x="x - 1" in exI)
  apply simp
  done

52.6 Magnitude of an Integer, as a Natural Number: nat

lift-definition nat :: int ⇒ nat is λ(x, y). x - y
  by auto

lemma nat-int [simp]: nat (int n) = n
  by transfer simp

lemma int-nat-eq [simp]: int (nat z) = (if 0 ≤ z then z else 0)
  by transfer clarsimp

corollary nat-0-le: 0 ≤ z ==> int (nat z) = z
  by simp

lemma nat-le-0 [simp]: z ≤ 0 ==> nat z = 0
  by transfer clarsimp

lemma nat-le-eq-zle: 0 < w | 0 ≤ z ==> (nat w ≤ nat z) = (w≤z)
  by transfer (clarsimp, arith)

An alternative condition is (0::'a) ≤ w

lemma nat-le-eq-zless: 0 ≤ w ==> (nat w < nat z) = (w<z)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma nat-less-eq-zless: 0 ≤ w ==> (nat w < nat z) = (w<z)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma zless-nat-conj [simp]: (nat w < nat z) = (0 < z & w < z)
  by transfer (clarsimp, arith)

lemma nonneg-eq-int:
fixes $z :: \text{int}$
assumes $0 \leq z$ and $\forall m. \; z = \text{int } m \implies P$
shows $P$
using assms by (blast dest: nat-0-le sym)

lemma nat-eq-iff:
$$\text{nat } w = m \iff (if \; 0 \leq w \; then \; w = \text{int } m \; else \; m = 0)$$
by transfer (clarsimp simp add: le-imp-diff-is-add)

corollary nat-eq-iff2:
$$m = \text{nat } w \iff (if \; 0 \leq w \; then \; w = \text{int } m \; else \; m = 0)$$
using nat-eq-iff[of $w$ $m$] by auto

lemma nat-0 [simp]:
$$\text{nat } 0 = 0$$
by (simp add: nat-eq-iff)

lemma nat-1 [simp]:
$$\text{nat } 1 = \text{Suc } 0$$
by (simp add: nat-eq-iff)

lemma nat-numeral [simp]:
$$\text{nat } (\text{numeral } k) = \text{numeral } k$$
by (simp add: nat-eq-iff)

lemma nat-neg-numeral [simp]:
$$\text{nat } (- \text{numeral } k) = 0$$
by simp

lemma nat-2: $\text{nat } 2 = \text{Suc } (\text{Suc } 0)$
by simp

lemma nat-less-iff: $0 \leq w \implies (\text{nat } w < m) = (w < \text{of-nat } m)$
by transfer (clarsimp simp add: arith)

lemma nat-le-iff: $\text{nat } x \leq n \iff x \leq \text{int } n$
by transfer (clarsimp simp add: le-diff-conv)

lemma nat-mono: $z \leq y \implies \text{nat } x \leq \text{nat } y$
by transfer auto

lemma nat-0 iff simp: $\text{nat } (i :: \text{int}) = 0 \iff i \leq 0$
by transfer clarsimp

lemma int-eq iff: $(\text{of-nat } m = z) = (m = \text{nat } z \& 0 \leq z)$
by (auto simp add: nat-eq-iff2)

lemma zero-less-nat-eq simp: $(0 < \text{nat } z) = (0 < z)$
by (insert zless-nat-conj[of 0], auto)
lemma nat-add-distrib:
  \( \theta \leq z \implies 0 \leq z' \implies \text{nat} (z + z') = \text{nat} z + \text{nat} z' \)
by transfer clarsimp

lemma nat-diff-distrib':
  \( \theta \leq x \implies 0 \leq y \implies \text{nat} (x - y) = \text{nat} x - \text{nat} y \)
by transfer clarsimp

lemma nat-diff-distrib:
  \( \theta \leq z' \implies z' \leq z \implies \text{nat} (z - z') = \text{nat} z - \text{nat} z' \)
by (rule nat-diff-distrib') auto

lemma nat-zminus-int [simp]: \(\text{nat} (- \text{int} n) = 0\)
by transfer simp

lemma le-nat-iff:
  \( k \geq 0 \implies n \leq \text{nat} k \iff \text{int} n \leq k \)
by transfer auto

lemma zless-nat-eq-int-zless: \((m < \text{nat} z) = (\text{int} m < z)\)
by transfer (clarsimp simp add: less-diff-conv)

context ring-1
begin

lemma of-nat-nat:
  \( \theta \leq z \implies \text{of-nat} (\text{nat} z) = \text{of-int} z \)
by transfer (clarsimp simp add: of-nat-diff)

end

lemma measure-function-int [measure-function]: \(\text{is-measure (nat o abs)}\) ..

52.7 Lemmas about the Function of-nat and Orderings

lemma negative-zless-0: \((\text{int} (\text{Suc} n)) < (0 :: \text{int})\)
by (simp add: order-less-le del: of-nat-Suc)

lemma negative-zless [iff]: \((\text{int} (\text{Suc} n)) < \text{int} m\)
by (rule negative-zless-0 [THEN order-less-le-trans], simp)

lemma negative-zle-0: \(\text{int} n \leq 0\)
by (simp add: minus-le-iff)
lemma negative-zle [iff]: \(-\text{int } n \leq \text{int } m\)
by (rule order-trans [OF negative-zle-0 of-nat-0-le-iff])

lemma not-zle-0-negative [simp]: \(\sim (0 \leq (\text{int } (\text{Suc } n)))\)
by (subst le-minus-iff, simp del: of-nat-Suc)

lemma int-zle-neg: \((\text{int } n \leq -\text{int } m) = (n = 0 \& m = 0)\)
by transfer simp

lemma not-int-zless-negative [simp]: \(\sim (\text{int } n < -\text{int } m)\)
by (simp add: linorder-not-less)

lemma negative-eq-positive [simp]: \((-\text{int } n = \text{of-nat } m) = (n = 0 \& m = 0)\)
by (force simp add: order-eq-iff [of - of-nat n]
int-zle-neg)

lemma zle-iff-zadd: \(w \leq z \iff (\exists n. z = w + \text{int } n)\)
proof
  have \((w \leq z) = (0 \leq z - w)\)
  by (simp only: le-diff-eq add-0-left)
  also have \(\ldots = (\exists n. z - w = \text{of-nat } n)\)
  by (auto elim: zero-le-imp-eq-int)
  also have \(\ldots = (\exists n. z = w + \text{of-nat } n)\)
  by (simp only: algebra-simps)
  finally show \(?thesis\).
qed

lemma zadd-int-left: \(\text{int } m + (\text{int } n + z) = \text{int } (m + n) + z\)
by simp

lemma int-Suc0-eq-1: \((\text{int } (\text{Suc } 0)) = 1\)
by simp

This version is proved for all ordered rings, not just integers! It is proved here because attribute arith-split is not available in theory Rings. But is it really better than just rewriting with abs-if?

lemma abs-split [arith-split, no-atp]:
  \(P(\text{abs}(a::\text{linordered-idom})) = ((0 \leq a \longrightarrow P a) \& (a < 0 \longrightarrow P(-a)))\)
by (force simp add: order-less-le-trans simp dest: order-less-le-trans)

lemma negD: \(x < 0 \Longrightarrow \exists n. x = -(\text{int } (\text{Suc } n))\)
apply transfer
apply clarsimp
apply (rule-tac x=b - Suc a in exI, arith)
done

52.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.
THEORY "Int"

theorem int-cases [case-names nonneg neg, cases type: int]:
[[!! n. z = int n ==> P; !! n. z = - (int (Suc n)) ==> P |] ==> P
apply (cases z < 0)
apply (blast dest: negD)
apply (simp add: linorder-not-less del: of-nat-Suc)
apply auto
apply (blast dest: nat-0-le [THEN sym])
done

theorem int-of-nat-induct [case-names nonneg neg, induct type: int]:
[[!! n. P (int n); !!n. P (- (int (Suc n))) |] ==> P z
by (cases z) auto

lemma nonneg-int-cases:
assumes 0 <= k obtains n where k = int n
using assms by (rule nonneg-eq-int)

lemma Let-numeral [simp]: Let (numeral v) f = f (numeral v)
— Unfold all lets involving constants
by (fact Let-numeral) — FIXME drop

lemma Let-neg-numeral [simp]: Let (- numeral v) f = f (- numeral v)
— Unfold all lets involving constants
by (fact Let-neg-numeral) — FIXME drop

Unfold \texttt{min} and \texttt{max} on numerals.

lemmas max-number-of [simp] =
max-def [of numeral u numeral v]
max-def [of numeral u - numeral v]
max-def [of - numeral u numeral v]
max-def [of - numeral u - numeral v] for u v

lemmas min-number-of [simp] =
min-def [of numeral u numeral v]
min-def [of numeral u - numeral v]
min-def [of - numeral u numeral v]
min-def [of - numeral u - numeral v] for u v

52.8.1 Binary comparisons

Preliminaries

lemma even-less-0-iff:
a + a < 0 <-> a < (0::'a::linordered-idom)
proof
have a + a < 0 <-> (1+1)*a < 0 by (simp add: distrib-right del: one-add-one)
also have (1+1)*a < 0 <-> a < 0
  by (simp add: mult-less-0-iff zero-less-two
      order-less-not-sym [OF zero-less-two])
finally show ?thesis .
lemma le-imp-0-less:
  assumes le: \(0 \leq z\)
  shows \((0::\text{int}) < 1 + z\)
proof -
  have \(0 \leq z\) by fact
  also have ... < \(z + 1\) by (rule less-add-one)
  also have ... = \(1 + z\) by (simp add: ac-simps)
  finally show \(0 < 1 + z\).
qed

lemma odd-less-0-iff:
  \((1 + z + z < 0) = (z < (0::\text{int}))\)
proof (cases \(z\))
  case (nonneg \(n\))
  thus \(?thesis\) by (simp add: linorder-not-less add_assoc add-increasing
  le-imp-0-less [THEN order-less-imp-le])
next
  case (neg \(n\))
  thus \(?thesis\) by (simp del: of-nat-Suc of-nat-add of-nat-1
  add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])
qed

52.8.2 Comparisons, for Ordered Rings

lemmas double-eq-0-iff = double-zero

lemma odd-nonzero:
  \(1 + z + z \neq (0::\text{int})\)
proof (cases \(z\))
  case (nonneg \(n\))
  have le: \(0 \leq z+z\) by (simp add: nonneg add-increasing)
  thus \(?thesis\) using le-imp-0-less [OF le]
  by (auto simp add: add.assoc)
next
  case (neg \(n\))
  show \(?thesis\)
proof
  assume eq: \(1 + z + z = 0\)
  have \((0::\text{int}) < 1 + (\text{int} n + \text{int} n)\)
    by (simp add: le-imp-0-less add-increasing)
  also have ... = \(- (1 + z + z)\)
    by (simp add: neg add.assoc [symmetric])
  also have ... = \(0\) by (simp add: eq)
  finally have \(0<0\)
  thus False by blast
qed
52.9 The Set of Integers

context ring-1
begin

definition Ints :: 'a set where
    Ints = range of-int

notation (xsymbols)
    Ints (\mathbb{Z})

lemma Ints-of-int [simp]: of-int z ∈ \mathbb{Z}
    by (simp add: Ints-def)

lemma Ints-of-nat [simp]: of-nat n ∈ \mathbb{Z}
    using Ints-of-int [of of-nat n] by simp

lemma Ints-0 [simp]: 0 ∈ \mathbb{Z}
    using Ints-of-int [of 0] by simp

lemma Ints-1 [simp]: 1 ∈ \mathbb{Z}
    using Ints-of-int [of 1] by simp

lemma Ints-add [simp]: a ∈ \mathbb{Z} ⇒ b ∈ \mathbb{Z} ⇒ a + b ∈ \mathbb{Z}
    apply (auto simp add: Ints-def)
    apply (rule range-eqI)
    apply (rule of-int-add [symmetric])
    done

lemma Ints-minus [simp]: a ∈ \mathbb{Z} ⇒ −a ∈ \mathbb{Z}
    apply (auto simp add: Ints-def)
    apply (rule range-eqI)
    apply (rule of-int-minus [symmetric])
    done

lemma Ints-diff [simp]: a ∈ \mathbb{Z} ⇒ b ∈ \mathbb{Z} ⇒ a − b ∈ \mathbb{Z}
    apply (auto simp add: Ints-def)
    apply (rule range-eqI)
    apply (rule of-int-diff [symmetric])
    done

lemma Ints-mult [simp]: a ∈ \mathbb{Z} ⇒ b ∈ \mathbb{Z} ⇒ a * b ∈ \mathbb{Z}
    apply (auto simp add: Ints-def)
    apply (rule range-eqI)
    apply (rule of-int-mult [symmetric])
    done

lemma Ints-power [simp]: a ∈ \mathbb{Z} ⇒ a ^ n ∈ \mathbb{Z}
    by (induct n) simp-all
THEORY "Int"

lemma Ints-cases [cases set: Ints]:
assumes q ∈ Z
obtains (of-int) z where q = of-int z
unfolding Ints-def
proof -
  from q ∈ Z: have q ∈ range of-int unfolding Ints-def .
  then obtain z where q = of-int z ..
  then show thesis ..
qed

lemma Ints-induct [case-names of-int, induct set: Ints]:
q ∈ Z ⇒ (⋀z. P (of-int z)) ⇒ P q
by (rule Ints-cases) auto
end

The premise involving Z prevents a = (1::'a) / (2::'a).

lemma Ints-double-eq-0-iff:
assumes in-Ints: a ∈ Ints
shows (a + a = 0) = (a = (0::'a::ring-char-0))
proof -
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume eq: a + a = 0
    thus a + a = 0 by simp
  next
    assume eq: a + a = 0
    hence of-int (z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence z + z = 0 by (simp only: of-int-eq-iff)
    hence z = 0 by (simp only: double-eq-0-iff)
    thus a = 0 by (simp add: a)
  qed
qed

lemma Ints-odd-nonzero:
assumes in-Ints: a ∈ Ints
shows 1 + a + a ≠ (0::'a::ring-char-0)
proof -
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume eq: 1 + a + a = 0
    hence of-int (1 + z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

lemma Nats-numeral \[simp\]: numeral \(w\) ∈ Nats
  using of-nat-in-Nats \[of numeral \(w\)\] by simp

lemma Ints-odd-less-0:
  assumes in-Ints: \(a\) ∈ Ints
  shows \((1 + a + a < 0) = (a < (0::'a::linordered-idom))\)
proof –
  from in-Ints have \(a\) ∈ range of-int unfolding Ints-def \[symmetric\] .
  then obtain \(z\) where \(a\): \(a = \text{of-int } z\) ..
  hence \((1::'a) + a + a < 0) = (\text{of-int } (1 + z + z) < (\text{of-int } 0 :: 'a))\)
    by (simp add: a)
  also have ... = \((z < 0)\) by (simp only: of-int-less-iff odd-less-0-iff)
  also have ... = \((a < 0)\) by (simp add: a)
  finally show \(?thesis\).
qed

52.10 \setsum and \setprod

lemma of-nat-setsum: \text{of-nat} \(\setsum f A\) = \(\sum_{x\in A.} \text{of-nat}(f x)\)
apply (cases finite A)
apply (erule finite-induct, auto)
done

lemma of-int-setsum: \text{of-int} \(\setsum f A\) = \(\sum_{x\in A.} \text{of-int}(f x)\)
apply (cases finite A)
apply (erule finite-induct, auto)
done

lemma of-nat-setprod: \text{of-nat} \(\setprod f A\) = \(\prod_{x\in A.} \text{of-nat}(f x)\)
apply (cases finite A)
apply (erule finite-induct, auto simp add: of-nat-mult)
done

lemma of-int-setprod: \text{of-int} \(\setprod f A\) = \(\prod_{x\in A.} \text{of-int}(f x)\)
apply (cases finite A)
apply (erule finite-induct, auto)
done

lemmas int-setsum = of-nat-setsum [where 'a=int]
lemmas int-setprod = of-nat-setprod [where 'a=int]

Legacy theorems

lemmas zle-int = of-nat-le-iff [where 'a=int]
lemmas int-int-eq = of-nat-eq-iff [where 'a=int]
lemmas numeral-1-eq-1 = numeral-One
52.11 Setting up simplification procedures

lemmas of-int-simps =
     of-int-0 of-int-1 of-int-add of-int-mult

lemmas int-arith-rules =
     numeral-One more-arith-simps of-nat-simps of-int-simps

ML-file Tools/int-arith.ML
declaration ⟨⟨ K Int-Arith.
 setup ⟩⟩
simproc-setup fast-arith ((m::'a::linordered-idom) < n |
      (m::'a::linordered-idom) <= n |
      (m::'a::linordered-idom) = n) =
⟨⟨ fn - => fn ss => fn ct => Lin-Arith.simproc ss (term-of ct) ⟩⟩

52.12 More Inequality Reasoning

lemma zless-add1-eq: (w < z + (1::int)) = (w<z | w=z)
by arith

lemma add1-zle-eq: (w + (1::int) ≤ z) = (w<z)
by arith

lemma zle-diff1-eq [simp]: (w ≤ z - (1::int)) = (w<z)
by arith

lemma zle-add1-eq-le [simp]: (w < z + (1::int)) = (w≤z)
by arith

lemma int-one-le-iff-zero-less: ((1::int) ≤ z) = (0 < z)
by arith

52.13 The functions nat and int

Simplify the term w + − z

lemma one-less-nat-eq [simp]: (Suc 0 < nat z) = (1 < z)
apply (insert zless-nat-conj [of 1 z])
apply auto
done

This simplifies expressions of the form int n = z where z is an integer literal.

lemmas int-eq-iff-numeral [simp] = int-eq-iff [of - numeral v] for v

lemma split-nat [arith-split]:
P(nat(i::int)) = ((∀ n. i = int n → P n) & (i < 0 → P 0))
(is ?P = (?L & ?R))
proof (cases i < 0)
  case True thus ?thesis by auto
next
case False
have \( \mathcal{P} = \mathcal{L} \)
proof
  assume \( \mathcal{P} \) thus \( \mathcal{L} \) using False by clarsimp
next
  assume \( \mathcal{L} \) thus \( \mathcal{P} \) using False by simp
qed
with False show \( \)thesis by simp
qed

context ring-1
begin

lemma of-int-of-nat [nitpick-simp]:
  \( \text{of-int } k = (\text{if } k < 0 \text{ then } - \text{of-nat } (\text{nat } (- k)) \text{ else of-nat } (\text{nat } k)) \)
proof (cases \( k < 0 \))
  case True then have \( 0 \leq - k \) by simp
  then have of-nat (nat (- k)) = of-int (- k) by (rule of-nat-nat)
  with True show \( \)thesis by simp
next
  case False then show \( \)thesis by (simp add: not-less of-nat-nat)
qed

end

lemma nat-mult-distrib:
  fixes \( z, z' :: \text{int} \)
  assumes \( 0 \leq z \)
  shows \( \text{nat } (z \times z') = \text{nat } z \times \text{nat } z' \)
proof (cases \( 0 \leq z' \))
  case False with assms have \( z \times z' \leq 0 \)
    by (simp add: not-le mult-le-0-iff)
  then have nat (z * z') = 0 by simp
moreover from False have \( nat z' = 0 \) by simp
ultimately show \( \)thesis by simp
next
  case True with assms have \( gc-0: \text{ z \times z' } \geq 0 \) by (simp add: zero-le-mult-iff)
  show \( \)thesis
    by (rule injD [of of-nat :: nat \Rightarrow int, OF inj-of-nat])
      (simp only: of-nat-mult of-nat-nat [OF True]
        of-nat-nat [OF assms] of-nat-nat [OF gc-0], simp)
qed

lemma nat-mult-distrib-neg: \( z \leq (0::int) \implies \text{nat}(z^{*}z') = \text{nat}(-z) \times \text{nat}(-z') \)
apply (rule trans)
apply (rule-tac [2] nat-mult-distrib, auto)
done
lemma nat-abs-mult-distrib: nat (abs (w * z)) = nat (abs w) * nat (abs z)
apply (cases z=0 | w=0)
apply (auto simp add: abs-if nat-mult-distrib [symmetric]
    nat-mult-distrib-neg [symmetric] mult-less-0-iff)
done

lemma Suc-nat-eq-nat-zadd1: (0::int) <= z ==> Suc (nat z) = nat (1 + z)
apply (rule sym)
apply (simp add: nat-eq-iff)
done

lemma diff-nat-eq-if:
    nat z - nat z' =
      (if z'<0 then nat z
      else let d = z-z' in
               if d < 0 then 0 else nat d)
by (simp add: Let-def nat-diff-distrib [symmetric])

lemma nat-numeral-diff-1 [simp]:
    numeral v - (1::nat) = nat (numeral v - 1)
using diff-nat-numeral [of v Num.One] by simp

52.14 Induction principles for int

Well-founded segments of the integers

definition
    int-ge-less-than :: int => (int * int) set
where
    int-ge-less-than d = \{(z',z). d <= z' & z' < z\}

theorem wf-int-ge-less-than: wf (int-ge-less-than d)
proof -
  have int-ge-less-than d <= measure (%z. nat (z-d))
    by (auto simp add: int-ge-less-than-def)
  thus ?thesis
    by (rule wf-subset [OF wf-measure])
qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition
    int-ge-less-than2 :: int => (int * int) set
where
    int-ge-less-than2 d = \{(z',z). d <= z & z' < z\}

theorem wf-int-ge-less-than2: wf (int-ge-less-than2 d)
proof -
  have int-ge-less-than2 d <= measure (%z. nat (1+z-d))
by (auto simp add: int-ge-less-than2-def)
thus thesis by (rule wf-subset [OF wf-measure])
qed

theorem int-ge-induct [case-names base step, induct set: int]:
  fixes i :: int
  assumes ge: k ≤ i and
  base: P k and
  step: \( \forall i. k ≤ i \implies P i \implies P (i + 1) \)
  shows P i
  proof –
    { fix n
      have \( \forall i::int. n = \text{nat} (i - k) \implies k ≤ i \implies P i \)
      proof (induct n)
        case 0
        hence i = k by arith
        thus P i using base by simp
      next
        case (Suc n)
        then have n = \text{nat}((i - 1) - k) by arith
        moreover
        have ki1: k ≤ i - 1 using Suc.prems by arith
        ultimately
        have P (i - 1) by (rule Suc.kyps)
        from step [OF ki1 this] show ?case by simp
      qed
    }
    with ge show thesis by fast
  qed

theorem int-gr-induct [case-names base step, induct set: int]:
  assumes gr: k < (i::int) and
  base: P(k+1) and
  step: \( \forall i. [k < i ; P i] \implies P(i+1) \)
  shows P i
  apply (rule int-ge-induct[of k + 1])
  using gr apply arith
  apply (rule base)
  apply (rule step, simp+)
  done

theorem int-le-induct [consumes 1, case-names base step]:
  assumes le: i ≤ (k::int) and
  base: P(k) and
  step: \( \forall i. [i ≤ k ; P i] \implies P(i - 1) \)
  shows P i
proof -
{ fix n
  have \( \forall i::\text{int}. \ n = \text{nat}(k-i) \implies i \leq k \implies P i \)
  proof (induct n)
    case 0
    hence \( i = k \) by arith
    thus \( P i \) using base by simp
  next
    case (Suc n)
    hence \( n = \text{nat}(k - (i + 1)) \) by arith
    moreover
    have \( ki1: i + 1 \leq k \) using Suc.prems by arith
    ultimately
    have \( P (i + 1) \) by (rule Suc.hyps)
    from step[OF ki1 this] show ?case by simp
  qed
}
with le show ?thesis by fast
qed

**theorem** int-less-induct [consumes 1, case-names base step]:
  assumes less: \( i::\text{int} < k \) and
  base: \( P(k - 1) \) and
  step: \( \forall i. [i < k; P i] \implies P(i - 1) \)
  shows \( P i \)
apply (rule int-le-induct[of \(- k - 1\)])
using less apply arith
apply (rule base)
apply (rule step, simp+)
done

**theorem** int-induct [case-names base step1 step2]:
  fixes \( k :: \text{int} \)
  assumes base: \( P k \)
  and step1: \( \forall i. k \leq i \implies P i \implies P (i + 1) \)
  and step2: \( \forall i. k \geq i \implies P i \implies P (i - 1) \)
  shows \( P i \)
proof -
  have \( i \leq k \lor i \geq k \) by arith
  then show ?thesis
  proof
    assume \( i \geq k \)
    then show ?thesis using base
    by (rule int-ge-induct) (fact step1)
  next
    assume \( i \leq k \)
    then show ?thesis using base
    by (rule int-le-induct) (fact step2)
  qed
**52.15 Intermediate value theorems**

**Lemma int-val-lemma:**
\[
(\forall i < n \cdot \text{abs}(f(i+1) - f(i)) \leq 1) \implies \\
0 \leq k \implies k \leq f n \implies (\exists i \leq n. f(i) = (k::int))
\]

unfolding One-nat-def
apply (induct n)
apply simp
apply (intro strip)
apply (erule impE, simp)
apply (erule_tac x = n in allE, simp)
apply (case_tac k = f (Suc n))
apply force
apply (erule impE)
apply (simp add: abs-if split add: split-if-asm)
apply (blast intro: le-SucI)

done

**Lemmas** nat0-intermed-int-val = int-val-lemma [rule-format (no-asmp)]

**Lemma** nat-intermed-int-val:
\[
\forall i. m \leq i \land i < n \implies \text{abs}(f(i+1) - f(i)) \leq 1; m < n; \\
f m \leq k; k \leq f n \implies (\exists i. m \leq i \land i \leq n \land f(i) = (k::int))
\]

apply (cut_tac n = n-m and f = %i. f(i+m) and k = k in int-val-lemma)
unfolding One-nat-def
apply simp
apply (erule exE)
apply (rule_tac x = i+m in exI, arith)

done

**52.16 Products and 1, by T. M. Rasmussen**

**Lemma** abs-less-one-iff [simp]: \(|z| < 1\) = \((z = (0::int))\)
by arith

**Lemma** abs-zmult-eq-1:
assumes mn: \(|m \ast n| = 1\)
shows \(|m| = (1::int)\)

proof
  have 0: \(m \neq 0 \land n \neq 0\) using mn
    by auto
  have \(2 \leq |m|\)
    proof
      assume 2 \(\leq |m|\)
      hence \(2 \ast |n| \leq |m\ast n|\)
        by (simp add: mult-mono 0)
      also have \(\ldots = |m\ast n|\)
    done
by (simp add: abs-mult)
also have ... = 1
by (simp add: mn)
finally have $2 \cdot |n| \leq 1$.
thus False using 0
by arith
qed
thus ?thesis using 0
by auto
qed

lemma pos-zmult-eq-1-iff-lemma: $(m \cdot n = 1) \implies m = (1 :: int) | m = -1$
by (insert abs-zmult-eq-1[of m n], arith)

lemma pos-zmult-eq-1-iff:
  assumes $0 < (m :: int)$
  shows $(m \cdot n = 1) = (m = 1 & n = 1)$
proof
  from assms have $m \cdot n = 1 \implies m = 1$ by (auto dest: pos-zmult-eq-1-iff-lemma)
  thus ?thesis by (auto dest: pos-zmult-eq-1-iff-lemma)
qed

lemma zmult-eq-1-iff: $(m \cdot n = (1 :: int)) = ((m = 1 & n = 1) | (m = -1 & n = -1))$
apply (rule iffI)
apply (frule pos-zmult-eq-1-iff-lemma)
apply (simp add: mult.commute[of m])
apply (frule pos-zmult-eq-1-iff-lemma, auto)
done

lemma infinite-UNIV-int: $\neg$ finite (UNIV :: int set)
proof
  assume finite (UNIV :: int set)
  moreover have inj $(\lambda i :: int. 2 \cdot i)$
    by (rule injI)
  ultimately have surj $(\lambda i :: int. 2 \cdot i)$
    by (rule finite-UNIV-inj-surj)
  then obtain $i :: int$ where $1 = 2 \cdot i$ by (rule surjE)
  then show False by (simp add: pos-zmult-eq-1-iff)
qed

52.17 Further theorems on numerals
52.17.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas distrib-right-numeral [simp] = distrib-right [of - numeral v] for v
lemmas distrib-left-numeral [simp] = distrib-left [of numeral v] for v
lemmas left-diff-distrib-numeral [simp] = left-diff-distrib [of - numeral v] for v
lemmas right-diff-distrib-numeral [simp] = right-diff-distrib [of numeral v] for v
These are actually for fields, like real: but where else to put them?

\textbf{lemmas} \quad \text{zero-less-divide-iff-numeral} \quad \text{[simp, no-atp]} = \quad \text{zero-less-divide-iff} \quad \text{[of numeral } w \text{]} \quad \text{for } w

\textbf{lemmas} \quad \text{divide-less-0-iff-numeral} \quad \text{[simp, no-atp]} = \quad \text{divide-less-0-iff} \quad \text{[of numeral } w \text{]} \quad \text{for } w

\textbf{lemmas} \quad \text{zero-le-divide-iff-numeral} \quad \text{[simp, no-atp]} = \quad \text{zero-le-divide-iff} \quad \text{[of numeral } w \text{]} \quad \text{for } w

\textbf{lemmas} \quad \text{divide-le-0-iff-numeral} \quad \text{[simp, no-atp]} = \quad \text{divide-le-0-iff} \quad \text{[of numeral } w \text{]} \quad \text{for } w

Replaces \textit{inverse} \#nn by \(1/\#nn\). It looks strange, but then other simprocs simplify the quotient.

\textbf{lemmas} \quad \text{inverse-eq-divide-numeral} \quad \text{[simp]} = \quad \text{inverse-} \quad \text{eq-} \quad \text{divide} \quad \text{[of numeral } w \text{]} \quad \text{for } w

\textbf{lemmas} \quad \text{inverse-eq-divide-neg-numeral} \quad \text{[simp]} = \quad \text{inverse-} \quad \text{eq-} \quad \text{divide} \quad \text{[of } - \text{ numeral } w \text{]} \quad \text{for } w

These laws simplify inequalities, moving unary minus from a term into the literal.

\textbf{lemmas} \quad \text{equation-minus-iff-numeral} \quad \text{[no-atp]} = \quad \text{equation-minus-iff} \quad \text{[of numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{minus-equation-iff-numeral} \quad \text{[no-atp]} = \quad \text{minus-equation-iff} \quad \text{[of } - \text{ numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{le-minus-iff-numeral} \quad \text{[no-atp]} = \quad \text{le-minus-iff} \quad \text{[of numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{minus-le-iff-numeral} \quad \text{[no-atp]} = \quad \text{minus-le-iff} \quad \text{[of } - \text{ numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{less-minus-iff-numeral} \quad \text{[no-atp]} = \quad \text{less-minus-iff} \quad \text{[of numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{minus-less-iff-numeral} \quad \text{[no-atp]} = \quad \text{minus-less-iff} \quad \text{[of } - \text{ numeral } v \text{]} \quad \text{for } v

— \text{FIXME} maybe simproc

Cancellation of constant factors in comparisons (< and ≤)

\textbf{lemmas} \quad \text{mult-less-cancel-left-numeral} \quad \text{[simp, no-atp]} = \quad \text{mult-less-cancel-left} \quad \text{[of numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{mult-less-cancel-right-numeral} \quad \text{[simp, no-atp]} = \quad \text{mult-less-cancel-right} \quad \text{[of } - \text{ numeral } v \text{]} \quad \text{for } v

\textbf{lemmas} \quad \text{mult-le-cancel-left-numeral} \quad \text{[simp, no-atp]} = \quad \text{mult-le-cancel-left} \quad \text{[of numeral } v \text{]} \quad \text{for } v
lemmas mult-le-cancel-right-numeral [simp, no-atp] = mult-le-cancel-right [of - numeral v] for v

Multiplying out constant divisors in comparisons (<, ≤ and =)

lemmas le-divide-eq-numeral1 [simp] =
    pos-le-divide-eq [of numeral w, OF zero-less-numeral]
    neg-le-divide-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas divide-le-eq-numeral1 [simp] =
    pos-divide-le-eq [of numeral w, OF zero-less-numeral]
    neg-divide-le-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas less-divide-eq-numeral1 [simp] =
    pos-less-divide-eq [of numeral w, OF zero-less-numeral]
    neg-less-divide-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas divide-less-eq-numeral1 [simp] =
    pos-divide-less-eq [of numeral w, OF zero-less-numeral]
    neg-divide-less-eq [of - numeral w, OF neg-numeral-less-zero] for w

lemmas eq-divide-eq-numeral1 [simp] =
    eq-divide-eq [of - - numeral w]
    eq-divide-eq [of - - - numeral w] for w

lemmas divide-eq-eq-numeral1 [simp] =
    divide-eq-eq [of - - numeral w]
    divide-eq-eq [of - - - numeral w] for w

52.17.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas le-divide-eq-numeral =
    le-divide-eq [of numeral w]
    le-divide-eq [of - numeral w] for w

lemmas divide-le-eq-numeral =
    divide-le-eq [of - - numeral w]
    divide-le-eq [of - - - numeral w] for w

lemmas less-divide-eq-numeral =
    less-divide-eq [of numeral w]
    less-divide-eq [of - numeral w] for w

lemmas divide-less-eq-numeral =
    divide-less-eq [of - - numeral w]
    divide-less-eq [of - - - numeral w] for w

lemmas eq-divide-eq-numeral =
    eq-divide-eq [of numeral w]
**THEORY “Int”**

\[
\text{eq\textunderscore divide\textunderscore eq} \ [\text{of - numeral w}] \text{ for w}
\]

**lemmas** divide\textunderscore eq\textunderscore eq\textunderscore numeral =

\[\text{divide\textunderscore eq} \ [\text{of - numeral w}]
\]

\[\text{divide\textunderscore eq} \ [\text{of - - numeral w}] \text{ for w}
\]

Not good as automatic simprules because they cause case splits.

**lemmas** divide\textunderscore const\textunderscore simps =

\[\text{le\textunderscore divide\textunderscore eq\textunderscore numeral} \text{ divide\textunderscore le\textunderscore eq\textunderscore numeral}
\]

\[\text{less\textunderscore divide\textunderscore eq\textunderscore numeral} \text{ divide\textunderscore eq\textunderscore eq\textunderscore numeral}
\]

\[\text{le\textunderscore divide\textunderscore eq\textunderscore 1} \text{ divide\textunderscore le\textunderscore eq\textunderscore 1} \text{ divide\textunderscore less\textunderscore eq\textunderscore 1}
\]

Division By \(-1\)

**lemma** divide\textunderscore minus1 [simp]: \((x::'a::field) / -1 = -x\)

unfolding non\textunderscore zero\textunderscore minus\textunderscore divide\textunderscore right [OF one\textunderscore neq\textunderscore zero, symmetric]

by simp

**lemma** half\textunderscore gt\textunderscore zero\textunderscore iff:

\((0 < r/2) = (0 < (r::'a::linordered\textunderscore field\textunderscore inverse\textunderscore zero))\)

by auto

**lemmas** half\textunderscore gt\textunderscore zero [simp] = half\textunderscore gt\textunderscore zero\textunderscore iff [THEN iffD2]

**lemma** divide\textunderscore Numeral1: \((x::'a::field) / \text{Numeral1} = x\)

by (fact divide\textunderscore numeral\textunderscore 1)

**52.18 The divides relation**

**lemma** zdvd\textunderscore antisym\textunderscore nonneg:

\(0 <= m ==> 0 <= n ==> m \text{ dvd} n ==> n \text{ dvd} m ==> m = (n::int)\)

apply (simp add: dvd\textunderscore def, auto)

apply (auto simp add: mult\textunderscore assoc zero\textunderscore le\textunderscore mult\textunderscore iff zmult\textunderscore eq\textunderscore 1\textunderscore iff)

done

**lemma** zdvd\textunderscore antisym\textunderscore abs: assumes \((a::int) \text{ dvd} b \text{ and} b \text{ dvd} a\)

shows \(|a| = |b|\)

proof cases

assume \(a = 0\) with assms show \(?thesis by simp\)

next

assume \(a \neq 0\)

from \((a \text{ dvd} b) \text{ obtain} k \text{ where} k\cdot b = a\cdot k\) unfolding dvd\textunderscore def by blast

from \((b \text{ dvd} a) \text{ obtain} k' \text{ where} k'\cdot a = b\cdot k'\) unfolding dvd\textunderscore def by blast

from \(k' \text{ have} a = a\cdot k'\cdot k'\) by simp

with mult\textunderscore cancel\textunderscore left [where \(c=a\) and \(b=k\cdot k'\)]

have \(kk'\cdot k'\cdot k' = 1\) using \((a\neq0)\) by (simp add: mult\textunderscore assoc)

hence \(k = 1 \land k' = 1 \lor k = -1 \land k' = -1\) by (simp add: zmult\textunderscore eq\textunderscore 1\textunderscore iff)

thus \(?thesis using k k' by auto\)

qed
lemma zdvd-zdiffD: \( k \mid m - n \implies k \mid n \)\( \implies k \mid n \)\( \implies k \mid m \)\( m \in \text{int} \)
  apply (subgoal_tac \( m = n + (m - n) \))
  apply (erule ssubst)
  apply (blast intro: dvd-add, simp)
  done

lemma zdvd-reduce: \( k \mid n + k \cdot m \) = \( k \mid n \)\( \cdot k \cdot m \)
  apply (rule iffI)
  apply (erule-tac [2] dvd-add)
  apply (subgoal-tac \( n = (n + k \cdot m) - k \cdot m \))
  apply (erule ssubst)
  apply (erule dvd-diff)
  apply (simp-all)
  done

lemma dvd-imp-le-int:
  fixes d i :: int
  assumes \( i \neq 0 \) and \( d \mid i \)
  shows \( |d| \leq |i| \)
  proof
    from assms have \( 0 < n \) by auto
    assume \( n \mid m \)
    then obtain k where \( m = n \cdot k \) ..
      with \( i \neq 0 \) have \( k \neq 0 \) by auto
      then have \( 1 \leq |k| \) and \( \theta \leq |d| \) by auto
      then have \( |d| \cdot 1 \leq |d| \cdot |k| \) by (rule mult-left-mono)
      with \( i = d \cdot k \) show ?thesis by (simp add: abs-mult)
    qed

lemma zdvd-not-zless:
  fixes m n :: int
  assumes \( 0 < m \) and \( m < n \)
  shows \( \neg n \mid m \)
  proof
    from assms have \( 0 < n \) by auto
    assume \( n \mid m \)
    then obtain k where \( m = n \cdot k \) ..
      with \( 0 < m \) have \( 0 < n \cdot k \) by auto
      with \( 0 < n \) have \( 0 < k \) by (simp add: zero-less-mult-iff)
      with \( k \cdot 0 < n \cdot k \cdot n \) have \( n \cdot k < n \cdot 1 \) by simp
      with \( 0 < n \cdot 0 < k \) show False unfolding mult-less-cancel-left by auto
    qed

lemma zdvd-mult-cancel:
  assumes \( d \cdot k \cdot m \mid d \cdot k \cdot n \) and \( \neg (0 \cdot i) \)
  shows \( m \mid d \cdot n \)
  proof
    from \( d \) obtain h where \( h \cdot k \cdot n = k \cdot m \cdot h \) unfolding dvd-def by blast
    \{ assume \( n \neq m \cdot h \) hence \( k \cdot n \neq k \cdot (m \cdot h) \) using \( kz \) by simp
      with \( h \) have False by (simp add: mult.assoc) \}
    hence \( n = m \cdot h \) by blast
    thus ?thesis by simp
  qed
theorem zdvd-int: \((x \ dvd y) = (\text{int } x \ dvd \text{int } y)\)

proof –
  have \:\:\\⋀k. \text{int } y = \text{int } x * k \Longrightarrow x \ dvd y
  proof
    fix k
    assume A: \text{int } y = \text{int } x * k
    then show x \ dvd y
      proof
        (cases k)
        case (nonneg n)
        with A have y = x * n by (simp add: of-nat-mult [symmetric])
        then show ?thesis ..
      next
        case (neg n)
        with A have \\ldots = \text{int } (x * Suc n) by (simp only: of-nat-mult [symmetric])
        also have \\ldots = \text{int } (x * Suc n) = \text{int } y \ldots
        finally have \ldots = \text{int } (x * Suc n) = \text{int } y ..
        then show ?thesis by (simp only: negative-eq-positive) auto
      qed
    qed
  then show ?thesis by (auto elim!: dvdE simp only: dvd-triv-left of-nat-mult)
  qed

lemma zdvd1-eq[simp]: 
  \((x :: \text{int}) \ dvd 1 = (|x| = 1)\)

proof
  assume d: \(x \ dvd 1\) hence \(\text{nat } |x| \ dvd \text{nat } (\text{nat } 1)\) by simp
  hence \(\text{nat } |x| \ dvd 1\) by (simp add: zdvd-int)
  hence \(\text{nat } |x| = 1\) by simp
  thus \(|x| = 1\) by (cases x < 0) auto
next
  assume |x|=1
  then have \(x = 1 \lor x = -1\) by auto
  then show \(x \ dvd 1\) by (auto intro: dvdI)
  qed

lemma zdvd-mult-cancel1:
  assumes mp:m ≠(0::int) shows \((m * n \ dvd m) = (|n| = 1)\)

proof
  assume nI: \(|n| = 1\) thus \(m * n \ dvd m\)
    by (cases n > 0) (auto simp add: minus-equation-iff)
next
  assume H: \(m * n \ dvd m\) hence H2: \(m * n \ dvd m * 1\) by simp
  from zdvd-mult-cancel[OF H2 mp] show \(|n| = 1\) by (simp only: zdvd1-eq)
  qed

lemma int-dvd-iff: 
  \((\text{int } m \ dvd z) = (m \ dvd \text{nat } (\text{abs } z))\)

unfolding zdvd-int by (cases z ≥ 0) simp-all
lemma dvd-int-iff: \( (z \text{ dvd } \text{int } m) = (\text{nat } (\text{abs } z) \text{ dvd } m) \)
unfolding zdvd-int by (cases \( z \geq 0 \)) simp-all

lemma nat-dvd-iff: \( (\text{nat } z \text{ dvd } m) = (\text{if } 0 \leq z \text{ then } (z \text{ dvd } \text{int } m) \text{ else } m = 0) \)
by (auto simp add: dvd-int-iff)

lemma eq-nat-nat-iff: 0 \( \leq \) \( z \) = \( \Rightarrow \) 0 \( \leq \) \( z' \) = \( \Rightarrow \) \( \text{nat } z \) = \( \text{nat } z' \) \( \iff \) \( z \) = \( z' \)
by (auto elim!: nonneg-eq-int)

lemma nat-power-eq: 0 \( \leq \) \( z \) = \( \Rightarrow \) \( \text{nat } (z \^ n) \) = \( \text{nat } z \^ n \)
by (induct \( n \)) (simp-all add: nat-mult-distrib)

lemma zdvd-imp-le: 0 \( \leq \) \( z \) \( \Rightarrow \) \( 0 < n \) \( \Rightarrow \) \( z \leq (n::\text{int}) \)
apply (cases \( n \))
apply (auto simp add: dvd-int-iff)
apply (cases \( z \))
apply (auto simp add: dvd-imp-le)
done

lemma zdvd-period:
fixes \( a \) \( d \) :: \text{int}
assumes \( a \text{ dvd } d \)
shows \( a \text{ dvd } (x + t) \iff a \text{ dvd } ((x + c \* d) + t) \)
proof -
from assms obtain \( k \) where \( d = a \* k \) by (rule dvdE)
show ?thesis
proof
  assume \( a \text{ dvd } (x + t) \)
  then obtain \( l \) where \( x + t = a \* l \) by (rule dvdE)
  then have \( x = a \* l - t \) by simp
  with \( d = a \* k \) show \( a \text{ dvd } x + c \* d + t \) by simp
next
  assume \( a \text{ dvd } x + c \* d + t \)
  then obtain \( l \) where \( x + c \* d + t = a \* l \) by (rule dvdE)
  then have \( x = a \* l - c \* d - t \) by simp
  with \( d = a \* k \) show \( a \text{ dvd } (x + t) \) by simp
qed

52.19 Finiteness of intervals

lemma finite-interval-int1 [iff]: finite \( \{i :: \text{int} \mid a <= i \& i <= b \} \)
proof (cases \( a <= b \))
case True
  from this show ?thesis
  proof (induct \( b \) rule: int-ge-induct)
case base

qed

qed
have \{ i \cdot a < i & i < a \} = \{ a \} by auto
from this show \textit{?case} by simp
next
case (step b)
from this have \{ i \cdot a < i & i < b + 1 \} = \{ i \cdot a < i & i < b \} \cup \{ b + 1 \} by auto
from this step show \textit{?case} by simp
qed
next
case False from this show \textit{?thesis}
by (metis (lifting, no-types) Collect-empty-eq finite.emptyI order-trans)
qed

lemma finite-interval-int2 [iff]: finite \{ i \cdot i < a \}
by (rule rev-finite-subset[OF finite-interval-int1[OF of a b]]) auto

lemma finite-interval-int3 [iff]: finite \{ i \cdot i < a \}
by (rule rev-finite-subset[OF finite-interval-int1[OF of a b]]) auto

lemma finite-interval-int4 [iff]: finite \{ i \cdot i < a \}
by (rule rev-finite-subset[OF finite-interval-int1[OF of a b]]) auto

52.20 Configuration of the code generator

Constructors

definition Pos :: num ⇒ int where
\[\text{simp, code-abbrev}: \text{Pos} = \text{numeral}\]
definition Neg :: num ⇒ int where
\[\text{simp, code-abbrev}: \text{Neg} n = - (\text{Pos} n)\]

code-datatype 0::int Pos Neg

Auxiliary operations

definition dup :: int ⇒ int where
\[\text{simp}: \text{dup} k = k + k\]

lemma dup-code [code]:
dup 0 = 0
dup (Pos n) = Pos (Num.Bit0 n)
dup (Neg n) = Neg (Num.Bit0 n)
unfolding Pos-def Neg-def
by (simp-all add: numeral-Bit0)

definition sub :: num ⇒ num ⇒ int where
\[\text{simp}: \text{sub} m n = \text{numeral} m - \text{numeral} n\]

lemma sub-code [code]:
\[\text{sub} \text{Num.One Num.One} = 0\]
THEORY “Int”

\[
\text{sub} (\text{Num.Bit0 } m) \text{ Num.One} = \text{Pos} (\text{Num.Bit0 } m)
\]
\[
\text{sub} (\text{Num.Bit1 } m) \text{ Num.One} = \text{Pos} (\text{Num.Bit0 } m)
\]
\[
\text{sub} \text{ Num.One} (\text{Num.Bit0 } n) = \text{Neg} (\text{Num.Bit0 } n)
\]
\[
\text{sub} \text{ Num.One} (\text{Num.Bit1 } n) = \text{Neg} (\text{Num.Bit0 } n)
\]
\[
\text{sub} (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = \text{dup} (\text{sub } m n)
\]
\[
\text{sub} (\text{Num.Bit1 } m) (\text{Num.Bit1 } n) = \text{dup} (\text{sub } m n) + 1
\]
\[
\text{sub} (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = \text{dup} (\text{sub } m n) - 1
\]
\[
\text{apply} (\text{simp-all only: sub-def dup-def numeral.simps Pos-def Neg-def numeral-BitM})
\]
\[
\text{apply} (\text{simp-all only: algebra-simps minus-diff-eq})
\]
\[
\text{apply} (\text{simp-all only: add.commute [of - (numeral } n + \text{ numeral } n)])
\]
\[
\text{apply} (\text{simp-all only: minus-add add.assoc left-minus})
\]
\[
\text{done}
\]

Implementations

\text{lemma one-int-code} [\text{code, code-unfold}]:

1 = \text{Pos Num.One}

\text{by simp}

\text{lemma plus-int-code} [\text{code}]:

k + 0 = (k::int)
\theta + l = (l::int)
\text{Pos } m + \text{Pos } n = \text{Pos } (m + n)
\text{Pos } m + \text{Neg } n = \text{sub } m n
\text{Neg } m + \text{Pos } n = \text{sub } n m
\text{Neg } m + \text{Neg } n = \text{Neg } (m + n)

\text{by simp-all}

\text{lemma uminus-int-code} [\text{code}]:

\text{uminus } 0 = (0::int)
\text{uminus } (\text{Pos } m) = \text{Neg } m
\text{uminus } (\text{Neg } m) = \text{Pos } m

\text{by simp-all}

\text{lemma minus-int-code} [\text{code}]:

k - 0 = (k::int)
\theta - l = \text{uminus } (l::int)
\text{Pos } m - \text{Pos } n = \text{sub } m n
\text{Pos } m - \text{Neg } n = \text{Pos } (m + n)
\text{Neg } m - \text{Pos } n = \text{Neg } (m + n)
\text{Neg } m - \text{Neg } n = \text{sub } n m

\text{by simp-all}

\text{lemma times-int-code} [\text{code}]:

k * 0 = (0::int)
\theta * l = (0::int)
\text{Pos } m * \text{Pos } n = \text{Pos } (m * n)
\text{Pos } m * \text{Neg } n = \text{Neg } (m * n)
\text{Neg } m * \text{Pos } n = \text{Neg } (m * n)
\textit{Neg }m \ast \textit{Neg }n = \textit{Pos } (m \ast n)
\textbf{by } simp-all

\textbf{instantiation} \textit{int :: equal}
\begin{verbatim}
definition 
HOL.equal k l \iff k = (l::int)
\end{verbatim}

\textbf{instance} \textbf{by} default \textbf{(rule equal-int-def)}

\textbf{end}

\textbf{lemma} \textit{equal-int-code} \textbf{[code]}:
\begin{verbatim}
HOL.equal 0 (0::int) \iff True
HOL.equal 0 (Pos l) \iff False
HOL.equal 0 (Neg l) \iff False
HOL.equal (Pos k) 0 \iff False
HOL.equal (Pos k) (Pos l) \iff HOL.equal k l
HOL.equal (Pos k) (Neg l) \iff False
HOL.equal (Neg k) 0 \iff False
HOL.equal (Neg k) (Pos l) \iff False
HOL.equal (Neg k) (Neg l) \iff HOL.equal k l
by (auto simp add: equal)
\end{verbatim}

\textbf{lemma} \textit{equal-int-refl} \textbf{[code nbe]}:
\begin{verbatim}
HOL.equal (k::int) k \iff True
by (fact equal-refl)
\end{verbatim}

\textbf{lemma} \textit{less-eq-int-code} \textbf{[code]}:
\begin{verbatim}
0 \leq (0::int) \iff True
0 \leq Pos l \iff True
0 \leq Neg l \iff False
Pos k \leq 0 \iff False
Pos k \leq Pos l \iff k \leq l
Pos k \leq Neg l \iff False
Neg k \leq 0 \iff True
Neg k \leq Pos l \iff True
Neg k \leq Neg l \iff l \leq k
by simp-all
\end{verbatim}

\textbf{lemma} \textit{less-int-code} \textbf{[code]}:
\begin{verbatim}
0 < (0::int) \iff False
0 < Pos l \iff True
0 < Neg l \iff False
Pos k < 0 \iff False
Pos k < Pos l \iff k < l
Pos k < Neg l \iff False
Neg k < 0 \iff True
\end{verbatim}
Neg $k < \text{Pos} \ l \iff \text{True} \\
Neg \ k < \text{Neg} \ l \iff l < k$
\text{by simp-all}

\text{lemma nat-code [code]:}
\text{nat} (\text{Int.Neg} \ k) = 0 \\
nat \ 0 = 0 \\
nat (\text{Int.Pos} \ k) = \text{nat-of-num} \ k \\
\text{by (simp-all add: nat-of-num-numeral)}

\text{lemma (in ring-1) of-int-code [code]:}
\text{of-int} (\text{Int.Neg} \ k) = - \text{numeral} \ k \\
\text{of-int} \ 0 = 0 \\
\text{of-int} (\text{Int.Pos} \ k) = \text{numeral} \ k \\
\text{by simp-all}

\text{Serializer setup}
\text{code-identifier}
\text{code-module} \text{Int \rightarrow (SML) Arith and (OCaml) Arith and (Haskell) Arith}

\text{quickcheck-params [default-type = int]}

\text{hide-const (open) Pos Neg sub dup}

\textbf{52.21 Legacy theorems}

\text{lemmas inj-int = inj-of-nat [where 'a=int]}
\text{lemmas zadd-int = of-nat-add [where 'a=int, symmetric]}
\text{lemmas int-mult = of-nat-mult [where 'a=int]}
\text{lemmas zmult-int = of-nat-mult [where 'a=int, symmetric]}
\text{lemmas int-eq-0-conv = of-nat-eq-0-iff [where 'a=int and m=n] for n}
\text{lemmas zless-int = of-nat-less-iff [where 'a=int]}
\text{lemmas int-less-0-conv = of-nat-less-0-iff [where 'a=int and m=k] for k}
\text{lemmas zero-less-int-conv = of-nat-0-less-iff [where 'a=int]}
\text{lemmas zero-le-int = of-nat-0-le-iff [where 'a=int]}
\text{lemmas int-le-0-conv = of-nat-0-le-iff [where 'a=int and m=n] for n}
\text{lemmas int-0 = of-nat-0 [where 'a=int]}
\text{lemmas int-1 = of-nat-1 [where 'a=int]}
\text{lemmas int-Suc = of-nat-Suc [where 'a=int]}
\text{lemmas int-numeral = of-nat-numeral [where 'a=int]}
\text{lemmas abs-int-eq = abs-of-nat [where 'a=int and n=m] for m}
\text{lemmas of-int-int-eq = of-int-of-nat-eq [where 'a=int]}
\text{lemmas zdifff-int = of-nat-diff [where 'a=int, symmetric]}
\text{lemmas spow-numeric-even = power-numeric-even [where 'a=int]}
\text{lemmas spow-numeric-odd = power-numeric-odd [where 'a=int]}

\text{lemma zpower-zpower:}
\text{(x ^ y) ^ z = (x ^ (y * z)):int)}
\text{by (rule power-mult [symmetric])}
lemma int-power:
  int (m ^ n) = int m ^ n
by (fact of-nat-power)

lemmas zpower-int = int-power [symmetric]

De-register int as a quotient type:
lifting-update int.lifting
lifting-forget int.lifting

end

53  Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back.

theory Nat-Transfer
imports Int
begin

53.1  Generic transfer machinery

definition transfer-morphism:: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ bool
  where transfer-morphism f A ←→ True

lemma transfer-morphismI[intro]: transfer-morphism f A
  by (simp add: transfer-morphism-def)

ML-file Tools/legacy-transfer.ML
setup Legacy-Transfer.setup

53.2  Set up transfer from nat to int

set up transfer direction

lemma transfer-morphism-nat-int: transfer-morphism nat (op <= (0::int)) ..

declare transfer-morphism-nat-int [transfer add
  mode: manual
  return: nat-0-le
  labels: nat-int
]

basic functions and relations

lemma transfer-nat-int-numerals [transfer key: transfer-morphism-nat-int]:
  (0::nat) = nat 0
  (1::nat) = nat 1
  (2::nat) = nat 2
(3::nat) = nat 3
by auto

definition
  tsub :: int ⇒ int ⇒ int
where
  tsub x y = (if x >= y then x - y else 0)

lemma tsub-eq: x >= y ⇒ tsub x y = x - y
by (simp add: tsub-def)

lemma transfer-nat-int-functions [transfer key: transfer-morphism-nat-int]:
  (x::int) >= 0 ⇒ y >= 0 ⇒ (nat x) + (nat y) = nat (x + y)
  (x::int) >= 0 ⇒ y >= 0 ⇒ (nat x) * (nat y) = nat (x * y)
  (x::int) >= 0 ⇒ y >= 0 ⇒ (nat x) - (nat y) = nat (tsub x y)
  (x::int) >= 0 ⇒ (nat x) *n = nat (x*n)
by (auto simp add: eq-nat-nat-iff nat-mult-distrib
  nat-power-eq tsub-def)

lemma transfer-nat-int-function-closures [transfer key: transfer-morphism-nat-int]:
  (x::int) >= 0 ⇒ y >= 0 ⇒ x + y >= 0
  (x::int) >= 0 ⇒ y >= 0 ⇒ x * y >= 0
  (x::int) >= 0 ⇒ y >= 0 ⇒ tsub x y >= 0
  (0::int) >= 0
  (1::int) >= 0
  (2::int) >= 0
  (3::int) >= 0
  int z >= 0
by (auto simp add: zero-le-mult-iff tsub-def)

lemma transfer-nat-int-relations [transfer key: transfer-morphism-nat-int]:
  x >= 0 ⇒ y >= 0 ⇒
    (nat (x::int) = nat y) = (x = y)
  x >= 0 ⇒ y >= 0 ⇒
    (nat (x::int) < nat y) = (x < y)
  x >= 0 ⇒ y >= 0 ⇒
    (nat (x::int) <= nat y) = (x <= y)
  x >= 0 ⇒ y >= 0 ⇒
    (nat (x::int) dvd nat y) = (x dvd y)
by (auto simp add: zdvd-int)

first-order quantifiers

lemma all-nat: (∀ x. P x) ←→ (∀ x≥0. P (nat x))
by (simp split add: split-nat)

lemma ex-nat: (∃ x. P x) ←→ (∃ x. 0 ≤ x ∧ P (nat x))
proof
  assume ∃ x. P x
then obtain $x$ where $P \ x$ ..
then have $\text{int} \ x \geq 0 \land P \ (\text{nat} \ (\text{int} \ x))$ by simp
then show $\exists x \geq 0. \ P \ (\text{nat} \ x)$ ..

next
assume $\exists x \geq 0. \ P \ (\text{nat} \ x)$
then show $\exists x. \ P \ x$ by auto
qed

lemma transfer-nat-int-quantifiers [transfer key: transfer-morphism-nat-int):

$(\text{ALL} \ (x:\text{nat}). \ P \ x) = (\text{ALL} \ (x:\text{int}). \ x \geq 0 \rightarrow P \ (\text{nat} \ x))$
$(\text{EX} \ (x:\text{nat}). \ P \ x) = (\text{EX} \ (x:\text{int}). \ x \geq 0 \land P \ (\text{nat} \ x))$
by (rule all-nat, rule ex-nat)

lemma all-cong: $(\bigwedge x. \ Q \ x \Rightarrow P \ x = P' \ x) \Rightarrow$
$(\text{ALL } x. \ Q \ x \rightarrow P \ x) = (\text{ALL } x. \ Q \ x \rightarrow P' \ x)$
by auto

lemma ex-cong: $(\bigwedge x. \ Q \ x \Rightarrow P \ x = P' \ x) \Rightarrow$
$(\text{EX } x. \ Q \ x \land P \ x) = (\text{EX } x. \ Q \ x \land P' \ x)$
by auto

declare transfer-morphism-nat-int [transfer add cong: all-cong ex-cong]

if

lemma nat-if-cong [transfer key: transfer-morphism-nat-int):
$(\text{if } P \text{ then } (\text{nat} \ x) \text{ else } (\text{nat} \ y)) = \text{nat} \ (\text{if } P \text{ then } x \text{ else } y)$
by auto

operations with sets

definition nat-set :: int set $\Rightarrow$ bool
where
nat-set $S = (\text{ALL } x:S. \ x \geq 0)$

lemma transfer-nat-int-set-functions:

card $A$ = card $(\text{int} \ ('A))$
\$
\{ \} = \text{nat} \ (\text{int} \ ('\{\}::int \ set))$
\$A \cup B = \text{nat} \ (\text{int} \ ('A \cup \text{int} \ ('B))$
\$A \cap B = \text{nat} \ (\text{int} \ ('A \cap \text{int} \ ('B))$
\$
\{x. \ P \ x\} = \text{nat} \ (\{x. \ x \geq 0 \land P(\text{nat} \ x)\}$
apply (rule card-image [symmetric])
apply (auto simp add: inj-on-def image-def)
apply (rule-tac $x = \text{int} \ x \in \text{bexI}$)
apply auto
apply (rule-tac $x = \text{int} \ x \in \text{bexI}$)
apply auto
apply (rule-tac $x = \text{int} \ x \in \text{bexI}$)
apply auto
apply (rule_tac x = int x in exI)
apply auto
done

lemma transfer-nat-int-set-function-closures:
  nat-set {}
  nat-set A \implies nat-set B \implies nat-set (A Un B)
  nat-set A \implies nat-set B \implies nat-set (A Int B)
  nat-set {x. x >= 0 & P x}
  nat-set (int ' C)
  nat-set A \implies x : A \implies x >= 0

unfolding nat-set-def apply auto
done

lemma transfer-nat-int-set-relations:
  (finite A) = (finite (int ' A))
  (x : A) = (int x : int ' A)
  (A = B) = (int ' A = int ' B)
  (A < B) = (int ' A < int ' B)
  (A <= B) = (int ' A <= int ' B)

apply (rule iffI)
apply (erule finite-imageI)
apply (erule finite-imageD)
apply (auto simp add: image-def set-eq-iff inj-on-def)
apply (drule_tac x = int x in spec, auto)
apply (drule_tac x = int x in spec, auto)
apply (drule-tac x = int x in spec, auto)
done

lemma transfer-nat-int-set-return-embed: nat-set A \implies
  (int ' nat ' A = A)
by (auto simp add: nat-set-def image-def)

lemma transfer-nat-int-set-cong: (!!!x. x >= 0 \implies P x = P' x) \implies
  \{x::int. x >= 0 & P x\} = \{x. x >= 0 & P' x\}
by auto

declare transfer-morphism-nat-int [transfer add
  return: transfer-nat-int-set-functions
  transfer-nat-int-set-function-closures
  transfer-nat-int-set-relations
  transfer-nat-int-set-return-embed
  cong: transfer-nat-int-set-cong
]

setsum and setprod

lemma transfer-nat-int-sum-prod:
  setsum f A = setsum (\%x. f (nat x)) (int ' A)
setprod f A = setprod (%x. f (nat x)) (int ' A)
apply (subst setsum.reindex)
apply (unfold inj-on-def, auto)
apply (subst setprod.reindex)
apply (unfold inj-on-def o-def, auto)
done

lemma transfer-nat-int-sum-prod2:
  setsum f A = nat(setsum (%x. int (f x)) A)
  setprod f A = nat(setprod (%x. int (f x)) A)
apply (subst int-setsum [symmetric])
apply auto
apply (subst int-setprod [symmetric])
apply auto
done

lemma transfer-nat-int-sum-prod-closure:
  nat-set A =⇒ (!!x. x >= 0 =⇒ f x >= (0::int)) =⇒ setsum f A >= 0
  nat-set A =⇒ (!!x. x >= 0 =⇒ f x >= (0::int)) =⇒ setprod f A >= 0
unfolding nat-set-def
apply (rule setsum-nonneg)
apply auto
apply (rule setprod-nonneg)
apply auto
done

lemma transfer-nat-int-sum-prod-cong:
  A = B =⇒ nat-set B =⇒ (!!x. x >= 0 =⇒ f x = g x) =⇒
  setsum f A = setsum g B
  A = B =⇒ nat-set B =⇒ (!!x. x >= 0 =⇒ f x = g x) =⇒
  setprod f A = setprod g B
unfolding nat-set-def
apply (subst setsum.cong, assumption)
apply auto [2]
apply (subst setprod.cong, assumption, auto)
done

declare transfer-morphism-nat-int [transfer add
  return: transfer-nat-int-sum-prod transfer-nat-int-sum-prod2
  transfer-nat-int-sum-prod-closure
  cong: transfer-nat-int-sum-prod-cong]
53.3 Set up transfer from int to nat

set up transfer direction

lemma transfer-morphism-int-nat: transfer-morphism int (λn. True) ..

declare transfer-morphism-int-nat [transfer add
  mode: manual
  return: nat-int
  labels: int-nat
]

basic functions and relations

definition
  is-nat :: int ⇒ bool
  where
  is-nat x = (x >= 0)

lemma transfer-int-nat-numerals:
  0 = int 0
  1 = int 1
  2 = int 2
  3 = int 3
  by auto

lemma transfer-int-nat-functions:
  (int x) + (int y) = int (x + y)
  (int x) ∗ (int y) = int (x ∗ y)
  tsub (int x) (int y) = int (x − y)
  (int x) ^n = int (x^n)
  by (auto simp add: int-mult tsub-def int-power)

lemma transfer-int-nat-function-closures:
  is-nat x ⇒ is-nat y ⇒ is-nat (x + y)
  is-nat x ⇒ is-nat y ⇒ is-nat (x ∗ y)
  is-nat x ⇒ is-nat y ⇒ is-nat (tsub x y)
  is-nat x ⇒ is-nat (x^n)
  is-nat 0
  is-nat 1
  is-nat 2
  is-nat 3
  is-nat (int z)
  by (simp-all only: is-nat-def transfer-nat-int-function-closures)

lemma transfer-int-nat-relations:
  (int x = int y) = (x = y)
  (int x < int y) = (x < y)
  (int x <= int y) = (x <= y)
  (int x dvd int y) = (x dvd y)
  by (auto simp add: zdvd-int)
declare transfer-morphism-int-nat [transfer add return:
  transfer-int-nat-numerals
  transfer-int-nat-functions
  transfer-int-nat-function-closures
  transfer-int-nat-relations
]

first-order quantifiers

lemma transfer-int-nat-quantifiers:

(ALL (x::int) >= 0. P x) = (ALL (x::nat). P (int x))
(EX (x::int) >= 0. P x) = (EX (x::nat). P (int x))
apply (subst all-nat)
apply auto [1]
apply (subst ex-nat)
apply auto
done

lemma int-if-cong: (if P then (int x) else (int y)) = int (if P then x else y)
by auto

declare transfer-morphism-int-nat [transfer add return: int-if-cong]

operations with sets

lemma transfer-int-nat-set-functions:

nat-set A =⇒ card A = card (nat ' A)
{ } = int ' ({ }::nat set)
nat-set A =⇒ nat-set B =⇒ A Un B = int ' (nat ' A Un nat ' B)
nat-set A =⇒ nat-set B =⇒ A Int B = int ' (nat ' A Int nat ' B)
{x. x >= 0 & P x} = int ' {x. P(int x)}

by (simp-all only: is-nat-def transfer-nat-int-set-functions
  transfer-nat-int-set-function-closures
  transfer-nat-int-set-return-embed nat-0-le
  cong: transfer-nat-int-set-cong)

lemma transfer-int-nat-set-function-closures:

nat-set {}
nat-set A =⇒ nat-set B =⇒ nat-set (A Un B)
nat-set A =⇒ nat-set B =⇒ nat-set (A Int B)
nat-set {x. x >= 0 & P x}
nat-set (int ' C)
nat-set A =⇒ x : A =⇒ is-nat x

by (simp-all only: transfer-nat-int-set-function-closures is-nat-def)
THEORY "Nat-Transfer"

lemma transfer-int-nat-set-relations:
  nat-set A \implies finite A = finite (nat' A)
  is-nat x \implies nat-set A \implies (x : A) = (nat x : nat' A)
  nat-set A \implies nat-set B \implies (A = B) = (nat' A = nat' B)
  nat-set A \implies nat-set B \implies (A < B) = (nat' A < nat' B)
  nat-set A \implies nat-set B \implies (A <= B) = (nat' A <= nat' B)
by (simp-all only: is-nat-def transfer-nat-int-set-relations
 transfer-nat-int-set-return-embed nat-0-le)

lemma transfer-int-nat-set-return-embed: nat' int' A = A
by (simp only: transfer-nat-int-set-relations
 transfer-nat-int-set-function-closures
 transfer-nat-int-set-return-embed nat-0-le)

lemma transfer-int-nat-set-cong: (!!x. P x = P' x) \implies
  \{ x::nat. P x \} = \{ x. P' x \}
by auto

declare transfer-morphism-int-nat [transfer add
 return: transfer-int-nat-set-functions
 transfer-int-nat-set-function-closures
 transfer-int-nat-set-relations
 transfer-int-nat-set-return-embed
 cong: transfer-int-nat-set-cong
]

setsum and setprod

lemma transfer-int-nat-set-sum-prod:
  nat-set A \implies setsum f A = setsum (%x. f (int x)) (nat' A)
  nat-set A \implies setprod f A = setprod (%x. f (int x)) (nat' A)
apply (subst setsum.reindex)
apply (unfold inj-on-def nat-set-def, auto simp add: eq-nat-nat-iff)
apply (subst setprod.reindex)
apply (unfold inj-on-def nat-set-def o-def, auto simp add: eq-nat-nat-iff
 cong: setprod.cong)
done

lemma transfer-int-nat-set-sum-prod2:
  (!!x. x:A \implies is-nat (f x)) \implies setsum f A = int(setsum (%x. nat (f x)) A)
  (!!x. x:A \implies is-nat (f x)) \implies
   setprod f A = int(setprod (%x. nat (f x)) A)
unfolding is-nat-def
apply (subst int-setsum, auto)
apply (subst int-setprod, auto simp add: cong: setprod.cong)
done

declare transfer-morphism-int-nat [transfer add
54 Divides: The division operators div and mod

theory Divides
imports Nat-Transfer
begin

54.1 Syntactic division operations

class div = dvd +
fixes div :: 'a ⇒ 'a ⇒ 'a (infixl div 70)
and mod :: 'a ⇒ 'a ⇒ 'a (infixl mod 70)

54.2 Abstract division in commutative semirings.

class semiring-div = comm-semiring-1-cancel + no-zero-divisors + div +
assumes mod-div-equality: a div b * b + a mod b = a
and div-by-0 [simp]: a div 0 = 0
and div-0 [simp]: 0 div a = 0
and div-mult-self1 [simp]: b ≠ 0 ⇒ (a + c * b) div b = c + a div b
and div-mult-mult1 [simp]: c ≠ 0 ⇒ (c * a) div (c * b) = a div b
begin

op div and op mod

lemma mod-div-equality2: b * (a div b) + a mod b = a
  unfolding mult.commute [of b]
  by (rule mod-div-equality)

lemma mod-div-equality': a mod b + a div b * b = a
  using mod-div-equality [of a b]
  by (simp only: ac-simps)

lemma div-mod-equality: ((a div b) * b + a mod b) + c = a + c
  by (simp add: mod-div-equality)

lemma div-mod-equality2: (b * (a div b) + a mod b) + c = a + c
  by (simp add: mod-div-equality2)

lemma mod-by-0 [simp]: a mod 0 = a
  using mod-div-equality [of a zero] by simp

lemma mod-0 [simp]: 0 mod a = 0
  using mod-div-equality [of zero a] div-0 by simp

lemma div-mult-self2 [simp]:
assumes $b 
eq 0$
shows $(a + b \cdot c) \div b = c + a \div b$
using assms div-mult-self1 [of $b\ a\ c$] by (simp add: mult.commute)

lemma div-mult-self3 [simp]:
assumes $b \neq 0$
shows $(c \cdot b + a) \div b = c + a \div b$
using assms by (simp add: add.commute)

lemma div-mult-self4 [simp]:
assumes $b \neq 0$
shows $(b \cdot c + a) \div b = c + a \div b$
using assms by (simp add: add.commute)

lemma mod-mult-self1 [simp]: $(a + c \cdot b) \mod b = a \mod b$
proof (cases $b = 0$)
case True then show ?thesis by simp
next
case False have $a + c \cdot b = (a + c \cdot b) \div b \cdot b + (a + c \cdot b) \mod b$
  by (simp add: mod-div-equality)
also from False div-mult-self1 [of $b\ a\ c$] have
  $\ldots = (c + a \div b) \cdot b + (a + c \cdot b) \mod b$
  by (simp add: algebra-simps)
finally have $a = a \div b \cdot b + (a + c \cdot b) \mod b$
  by (simp add: add.commute [of $a$] add.assoc distrib-right)
then have $a \div b \cdot b + (a + c \cdot b) \mod b = a \div b \cdot b + a \mod b$
  by (simp add: mod-div-equality)
then show ?thesis by simp
qed

lemma mod-mult-self2 [simp]:
$(a + b \cdot c) \mod b = a \mod b$
by (simp add: mult.commute [of $b$])

lemma mod-mult-self3 [simp]:
$(c \cdot b + a) \mod b = a \mod b$
by (simp add: add.commute)

lemma mod-mult-self4 [simp]:
$(b \cdot c + a) \mod b = a \mod b$
by (simp add: add.commute)

lemma div-mult-self1-is-id [simp]: $b \neq 0 \implies b \div b = a$
using div-mult-self2 [of $b\ 0\ a$] by simp

lemma div-mult-self2-is-id [simp]: $b \neq 0 \implies a \div b = a$
using div-mult-self1 [of $b\ 0\ a$] by simp
THEORY "Divides"

lemma mod-mult-self1-is-0 [simp]: \( b \times a \mod b = 0 \)
  using mod-mult-self2 [of 0 b a] by simp

lemma mod-mult-self2-is-0 [simp]: \( a \times b \mod b = 0 \)
  using mod-mult-self1 [of 0 a b] by simp

lemma div-by-1 [simp]: \( a \div 1 = a \)
  using div-mult-self2-is-id [of 1 a] zero-neq-one by simp

lemma div-self [simp]: \( a \div a = 1 \)
  using div-mult-self2-is-id [of -1] by simp

lemma div-add-self1 [simp]:
  assumes \( b \neq 0 \)
  shows \( (b + a) \div b = a \div b + 1 \)
  using assms div-mult-self1 [of b a 1] by (simp add: add.commute)

lemma div-add-self2 [simp]:
  assumes \( b \neq 0 \)
  shows \( (a + b) \div b = a \div b + 1 \)
  using assms div-add-self1 [of b a] by (simp add: add.commute)

lemma mod-add-self1 [simp]:
  \((b + a) \mod b = a \mod b\)
  using mod-mult-self1 [of a 1 b] by (simp add: add.commute)

lemma mod-add-self2 [simp]:
  \((a + b) \mod b = a \mod b\)
  using mod-mult-self1 [of a 1 b] by simp

lemma mod-div-decomp:
  fixes \( a \)
  obtains \( q, r \) where \( q = a \div b \) and \( r = a \mod b \)
  and \( a = q \times b + r \)
  proof -
    from mod-div-equality have \( a = a \div b \times b + a \mod b \) by simp
    moreover have \( a \div b = a \div b \) ..
    moreover have \( a \mod b = a \mod b \) ..
    note that ultimately show thesis by blast
qed

lemma dvd-eq-mod-eq-0 [code]: \(a \text{ dvd } b \iff b \mod a = 0\)
proof
  assume \(b \mod a = 0\)
  with \(\text{mod-div-equality \ [of \ b \ a]}\) have \(b \div a * a = b\) by simp
  then have \(b = a * (b \div a)\) unfolding \text{mult.commute} ..
  then have \(\exists c. b = a * c\ ..\)
  then show \(a \text{ dvd } b\) unfolding \text{dvd-def} .
next
  assume \(a \text{ dvd } b\)
  then have \(\exists c. b = a * c\) unfolding \text{dvd-def} ..
  then obtain \(c\) where \(b = a * c\ ..\)
  then have \(b \mod a = a * c \mod a\) by simp
  then have \(b \mod a = c * a \mod a\) by (simp add: \text{mult.commute})
  then show \(b \mod a = 0\) by simp
qed

lemma mod-div-trivial [simp]: \(a \mod b \dir b = 0\)
proof (cases \(b = 0\))
  assume \(b = 0\)
  thus \(\text{thesis}\) by simp
next
  assume \(b \neq 0\)
  hence \(a \div b + a \mod b \div b = (a \mod b + a \div b * b) \div b\)
    by (rule \text{div-mult-self1 \ [symmetric]})
  also have \(\ldots = a \div b\)
    by (simp only: \text{mod-div-equality}')
  also have \(\ldots = a \div b + 0\)
    by simp
  finally show \(\text{thesis}\)
    by (rule \text{add-left-imp-eq})
qed

lemma mod-mod-trivial [simp]: \(a \mod b \mod b = a \mod b\)
proof
  have \(a \mod b \mod b = (a \mod b + a \div b * b) \mod b\)
    by (simp only: mod-div-equality')
  also have \(\ldots = a \mod b\)
    by (simp only: \text{mod-div-equality}')
  finally show \(\text{thesis}\).
qed

lemma dvd-imp-mod-0: \(a \text{ dvd } b \Longrightarrow b \mod a = 0\)
by (rule dvd-eq-mod-eq-0[THEN iffD1])

lemma dvd-div-mult-self: \(a \text{ dvd } b \Longrightarrow (b \div a) * a = b\)
by (subst (2) \text{mod-div-equality \ [of \ b \ a, symmetric]} \ (simp add:dvd-imp-mod-0)
lemma dvd-mult-div-cancel: \( a \mid b \Rightarrow a \cdot (b \div a) = b \)
by (drule dvd-div-mult-self) (simp add: mult.commute)

lemma dvd-div-mult: \( a \mid b \Rightarrow (b \div a) \cdot c = b \cdot c \div a \)
apply (cases \( a = 0 \))
apply simp
apply (auto simp: dvd_def mult_assoc)
done

lemma div-dvd-div[simp]:
\[ a \mid b \Rightarrow a \mid c \Rightarrow (b \div a) \mid c \div a = (b \mid c) \]
apply (cases \( a = 0 \))
apply simp
apply (unfold dvd_def)
apply auto
apply (blast intro: mult_assoc[symmetric])
apply (fastforce simp add: mult_assoc)
done

lemma dvd-mod-imp-dvd: 
\[ k \mid m \mod n; \ k \mid n \] \(\Rightarrow\) \( k \mid m \)
apply (subgoal_tac \( k \mid m \div n \cdot n + m \mod n \))
apply (simp add: mod-div-equality)
apply (simp only: dvd_add dvd_mult)
done

Addition respects modular equivalence.

lemma mod-add-left-eq: \( (a + b) \mod c = (a \mod c + b) \mod c \)
proof -
  have \( (a + b) \mod c = (a \div c \cdot c + a \mod c + b) \mod c \)
    by (simp only: mod-div-equality)
  also have \( \ldots = (a \mod c + b + a \div c \cdot c) \mod c \)
    by (simp only: ac-simps)
  also have \( \ldots = (a \mod c + b) \mod c \)
    by (rule mod-mult-self1)
  finally show \( \vdots \).
qed

lemma mod-add-right-eq: \( (a + b) \mod c = (a + b \mod c) \mod c \)
proof -
  have \( (a + b) \mod c = (a + (b \div c \cdot c + b \mod c)) \mod c \)
    by (simp only: mod-div-equality)
  also have \( \ldots = (a + b \mod c + b \div c \cdot c) \mod c \)
    by (simp only: ac-simps)
  also have \( \ldots = (a + b \mod c) \mod c \)
    by (rule mod-mult-self1)
  finally show \( \vdots \).
qed

lemma mod-add-eq: \( (a + b) \mod c = (a \mod c + b \mod c) \mod c \)
by (rule trans [OF mod-add-left-eq mod-add-right-eq])

lemma mod-add-cong:
  assumes a mod c = a' mod c
  assumes b mod c = b' mod c
  shows (a + b) mod c = (a' + b') mod c
proof –
  have (a mod c + b mod c) mod c = (a' mod c + b' mod c) mod c
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-add-eq [symmetric])
qed

lemma div-add [simp]: z dvd x =⇒ z dvd y
  ⇒ (x + y) dvd z = x dvd z + y dvd z
by (cases z = 0, simp, unfold dvd_def, auto simp add: algebra-simps)

Multiplication respects modular equivalence.

lemma mod-mult-left-eq: (a * b) mod c = ((a mod c) * b) mod c
proof –
  have (a * b) mod c = ((a div c * c + a mod c) * b) mod c
    by (simp only: mod-div-equality)
  also have ... = (a mod c * b + a div c * b * c) mod c
    by (simp only: algebra-simps)
  also have ... = (a mod c * b) mod c
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

lemma mod-mult-right-eq: (a * b) mod c = (a * (b mod c)) mod c
proof –
  have (a * b) mod c = (a * (b div c * c + b mod c)) mod c
    by (simp only: mod-div-equality)
  also have ... = (a * (b mod c) + a * (b div c) * c) mod c
    by (simp only: algebra-simps)
  also have ... = (a * (b mod c)) mod c
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

lemma mod-mult-eq: (a * b) mod c = ((a mod c) * (b mod c)) mod c
by (rule trans [OF mod-mult-left-eq mod-mult-right-eq])

lemma mod-mult-cong:
  assumes a mod c = a' mod c
  assumes b mod c = b' mod c
  shows (a * b) mod c = (a' * b') mod c
proof –
  have (a mod c * (b mod c)) mod c = (a' mod c * (b' mod c)) mod c
Exponentiation respects modular equivalence.

lemma power-mod: \((a \mod b)^n \mod b = a^n \mod b\)
apply (induct n, simp-all)
apply (rule mod-mult-right-eq [THEN trans])
apply (simp (no-asm-simp))
apply (rule mod-mult-eq [symmetric])
done

lemma mod-mod-cancel:
assumes \(c \text{ dvd } b\)
shows \(a \mod b \mod c = a \mod c\)
proof
 from \(\langle c \text{ dvd } b\rangle\) obtain \(k\) where \(b = c \ast k\)
 by (rule dvdE)
 have \(a \mod b \mod c = a \mod (c \ast k) \mod c\)
 by (simp only: \(\langle b = c \ast k\rangle\))
 also have \(\ldots = (a \mod (c \ast k) + a \div (c \ast k) \ast k \ast c) \mod c\)
 by (simp only: mod-mult-self1)
 also have \(\ldots = (a \div (c \ast k) \ast (c \ast k) + a \mod (c \ast k)) \mod c\)
 by (simp only: ac-simps ac-simps)
 also have \(\ldots = a \mod c\)
 by (simp only: mod-div-equality)
 finally show \(\text{thesis}\).
qed

lemma div-mult-swap:
assumes \(c \text{ dvd } b\)
shows \(a \ast (b \div c) = (a \ast b) \div c\)
proof 
 from assms have \(b \div c \ast (a \div 1) = b \ast a \div (c \ast 1)\)
 by (simp only: div-mult-div-if-dvd one-dvd)
 then show \(\text{thesis}\) by (simp add: mult.commute)
qed

lemma div-mult-mult2 [simp]:

...
\( c \neq 0 \implies (a \cdot c) \div (b \cdot c) = a \div b \)
by (drule \text{div-mul-mult1}) (simp add: \text{mult.commute})

\textbf{lemma} \text{div-mul-mult1-if} [simp]:
\((c \cdot a) \div (c \cdot b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a \div b)\)
by simp-all

\textbf{lemma} \text{mod-mul-mult1}:
\((c \cdot a) \mod (c \cdot b) = c \cdot (a \mod b)\)
\textbf{proof} (cases \(c = 0\))
\begin{enumerate}
\item \text{case} True then show \?thesis by simp
\item \text{case} False from \text{mod-div-equality} have \((\cdot a \div b) \cdot \cdot b + (\cdot a) \mod (\cdot b) = c \cdot a \cdot)
\begin{enumerate}
\item \text{with} False have \(c \cdot a + c \cdot (a \mod b)\) by (simp add: algebra-simps)
\item \text{with} \text{mod-div-equality} show \?thesis by simp
\end{enumerate}
\end{enumerate}
qed

\textbf{lemma} \text{mod-mul-mult2}:
\((a \cdot c) \mod (b \cdot c) = (a \mod b) \cdot c\)
using \text{mod-mul-mult1} \(\text{of } c \cdot a \cdot b\) by (simp add: \text{mult.commute})

\textbf{lemma} \text{mult-mod-left} (a \mod b) \cdot c = (a \cdot c) \mod (b \cdot c)
by (fact \text{mod-mul-mult2 [symmetric]})

\textbf{lemma} \text{mult-mod-right} (c \cdot a) \mod (c \cdot b)
by (fact \text{mod-mul-mult1 [symmetric]})

\textbf{lemma} \text{dvd-mod} \(k \cdot \text{dvd } m \implies k \cdot \text{dvd } n \implies k \cdot \text{dvd } (m \mod n)\)
\textbf{unfolding} \text{dvd-def} by (auto simp add: \text{mod-mul-mult1})

\textbf{lemma} \text{dvd-mod-iff} \(k \cdot \text{dvd } n \implies k \cdot \text{dvd } (m \mod n) \iff k \cdot \text{dvd } m\)
by (blast intro: \text{dvd-mod-imp-dvd dvd-mod})

\textbf{lemma} \text{div-power}:
\(y \cdot \text{dvd } x \implies (x \div y) \cdot n = x \cdot n \div y \cdot n\)
apply (induct \(n\))
apply simp
apply (simp add: \text{div-mult-div-if-dvd dvd-powersame})
done

\textbf{lemma} \text{dvd-div-eq-mult}:
\(\text{assumes } a \neq 0 \text{ and } a \cdot \text{dvd } b\)
\(\text{shows } b \div a = c \iff b = c \cdot a\)
\textbf{proof}
\begin{enumerate}
\item \text{assume} \(b = c \cdot a\)
\item then show \(b \div a = c\) by (simp add: \text{assms})
\end{enumerate}
next
  assume b div a = c
  then have b div a * a = c * a by simp
moreover from ⟨a dvd b⟩ have b div a * a = b by (simp add: dvd-div-mult-self)
  ultimately show b = c * a by simp
qed

lemma dvd-div-div-eq-mult:
  assumes a ≠ 0 c ≠ 0 and a dvd b c dvd d
  shows b div a = d div c ←→ b * c = a * d
div-mult-swap intro: sym)
end

class ring-div = semiring-div + comm-ring-1
begin

subclass ring-1-no-zero-divisors ..

Negation respects modular equivalence.

lemma mod-minus-eq: (− a) mod b = (− (a mod b)) mod b
proof −
  have (− a) mod b = (− (a div b * b + a mod b)) mod b
    by (simp only: mod-div-equality)
  also have ... = (− (a mod b) + − (a div b) * b) mod b
    by (simp add: ac-simps)
  also have ... = (− (a mod b)) mod b
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

lemma mod-minus-cong:
  assumes a mod b = a' mod b
  shows (− a) mod b = (− a') mod b
proof −
  have (− (a mod b)) mod b = (− (a' mod b)) mod b
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-minus-eq [symmetric])
qed

Subtraction respects modular equivalence.

lemma mod-diff-left-eq:
  (a − b) mod c = (a mod c − b) mod c
  using mod-add-cong [of a c a mod c − b − b] by simp

lemma mod-diff-right-eq:
  (a − b) mod c = (a − b mod c) mod c
using mod-add-cong \([\text{of } a \ b \ (b \mod c)]\) mod-minus-cong \([\text{of } b \mod c \ c \ b]\) by simp

lemma mod-diff-eq:
\((a - b) \mod c = (a \mod c - b \mod c) \mod c\)
using mod-add-cong \([\text{of } a \ b \mod c \ c \ b]\) by simp

lemma mod-diff-cong:
assumes \(a \mod c = a' \mod c\)
assumes \(b \mod c = b' \mod c\)
shows \((a - b) \mod c = (a' - b') \mod c\)
using assms mod-add-cong \([\text{of } a \ b \mod c \ c \ b]\) by simp

lemma dvd-neg-div: \(y \dvd x \implies -x \div y = -(x \div y)\)
apply (case-tac \(y = 0\)) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac \(-y * k = y * -k\))
apply (simp only:)
apply (erule div-mul-self1-is-id)
apply simp
apply simp
done

lemma dvd-div-neg: \(y \dvd x \implies x \div -y = -(x \div y)\)
apply (case-tac \(y = 0\)) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac \(y * k = -y * -k\))
apply (erule ssubst, rule div-mul-self1-is-id)
apply simp
apply simp
apply simp
done

lemma div-minus-minus [simp]: \((-a) \div (-b) = a \div b\)
using div-mult-mult1 \([\text{of } \neg 1 \ a \ b]\)

unfolding neg-equal-0-iff-equal by simp

lemma mod-minus-minus [simp]: \((-a) \mod (-b) = -(a \mod b)\)
using mod-mult-mult1 \([\text{of } \neg 1 \ a \ b]\) by simp

lemma div-minus-right: \(a \div (-b) = (-a) \div b\)
using div-minus-minus \([\text{of } \neg a \ b]\) by simp

lemma mod-minus-right: \(a \mod (-b) = -((-a) \mod b)\)
using mod-minus-minus \([\text{of } \neg a \ b]\) by simp

lemma div-minus1-right [simp]: \(a \div (-1) = -a\)
using div-minus-right \([\text{of } a \ 1]\) by simp
lemma mod-minus1-right [simp]: a mod (−1) = 0
  using mod-minus-right [of a 1] by simp

lemma minus-mod-self2 [simp]:
  (a − b) mod b = a mod b
  by (simp add: mod-diff-right-eq)

lemma minus-mod-self1 [simp]:
  (b − a) mod b = −a mod b
  using mod-add-self2 [of −a b]
  by simp

end

class semiring-div-parity = semiring-div + semiring-numeral +
  assumes parity: a mod 2 = 0 ∨ a mod 2 = 1
begin

lemma parity-cases [case-names even odd]:
  assumes a mod 2 = 0 ⇒ P
  assumes a mod 2 = 1 ⇒ P
  shows P
  using assms parity
  by blast

lemma not-mod-2-eq-0-eq-1 [simp]:
  a mod 2 ≠ 0 ↔ a mod 2 = 1
  by (cases a rule: parity-cases) simp-all

lemma not-mod-2-eq-1-eq-0 [simp]:
  a mod 2 ≠ 1 ↔ a mod 2 = 0
  by (cases a rule: parity-cases) simp-all

end

54.3 Generic numeral division with a pragmatic type class

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments. This is its primary motivation, and it could surely be formulated using a more fine-grained, more algebraic and less technical class hierarchy.

class semiring-numeral-div = linordered-semidom + minus + semiring-div +
  assumes diff-invert-add1: a + b = c ⇒ a = c − b
  and le-add-diff-inverse2: b ≤ a ⇒ a − b + b = a
  assumes mult-div-cancel: b * (a div b) = a − a mod b
  and div-less: 0 ≤ a ⇒ a < b ⇒ a div b = 0
  and mod-less: 0 ≤ a ⇒ a < b ⇒ a mod b = a
  and div-positive: 0 < b ⇒ b ≤ a ⇒ a div b > 0
  and mod-less-eq-dividend: 0 ≤ a ⇒ a mod b ≤ a
  and pos-mod-bound: 0 < b ⇒ a mod b < b
and pos-mod-sign: 0 < b \implies 0 \leq a \mod b \\
and mod-mult2-eq: 0 \leq c \implies a \mod (b \times c) = b \times (a \div b \mod c) + a \mod b \\
and div-mult2-eq: 0 \leq c \implies a \div (b \times c) = a \div b \div c \\
assumes discrete: a < b \iff a + 1 \leq b 

begin 

lemma diff-zero [simp]: 
\begin{align*} 
  a - 0 &= a \\
  \text{by (rule diff-invert-add1 [symmetric]) simp} 
\end{align*} 

subclass semiring-div-parity 
proof 
fix a 
show a \mod 2 = 0 \lor a \mod 2 = 1 
proof (rule ccontr) 
assume \neg (a \mod 2 = 0 \lor a \mod 2 = 1) 
then have a \mod 2 \neq 0 \and a \mod 2 \neq 1 \by simp-all 
have 0 < 2 \by simp 
with pos-mod-bound pos-mod-sign have 0 \leq a \mod 2 \a \mod 2 < 2 \by simp-all 
with (a \mod 2 \neq 0 \or) have 0 < a \mod 2 \by simp 
with discrete have 1 \leq a \mod 2 \by simp 
with (a \mod 2 \neq 1 \or) have 1 < a \mod 2 \by simp 
with discrete have 2 \leq a \mod 2 \by simp 
with (a \mod 2 < 2) show False \by simp 
qed 

qed 

lemma divmod-digit-1: 
assumes 0 \leq a \and 0 < b \and b \leq a \mod (2 \times b) 
shows 2 \times (a \div (2 \times b)) + 1 = a \div b \is ?P 
  \and a \mod (2 \times b) - b = a \mod b \is ?Q 
proof 
from assms mod-less-eq-dividend [of a 2 * b] have b \leq a 
by (auto intro: trans) 
with (0 < b) have 0 < a \div b \by (auto intro: div-positive) 
then have [simp]: 1 \leq a \div b \by (simp add: discrete) 
with (0 < b) have mod-less: a \mod b < b \by (simp add: pos-mod-bound) 
def w \equiv a \div b \mod 2 \with parity have w-exhaust: w = 0 \lor w = 1 \by auto 
have mod-w: a \mod (2 \times b) = a \mod b + b \times w 
  \by (simp add: w-def mod-mult2-eq ac-simps) 
from assms w-exhaust have w = 1 
by (auto simp add: mod-w) (insert mod-less, auto) 
with mod-w have mod: a \mod (2 \times b) = a \mod b + b \by simp 
have 2 \times (a \div (2 \times b)) = a \div b - w 
  \by (simp add: w-def div-mult2-eq mult-div-cancel ac-simps) 
with (w = 1) have div: 2 \times (a \div (2 \times b)) = a \div b - 1 \by simp 
then show \is ?P and \is ?Q 
  \by (simp-all add: div mod diff-invert-add1 [symmetric] le-add-diff-inverse2) 
qed
lemma divmod-digit-0:
  assumes 0 < b and a mod (2 * b) < b
  shows 2 * (a div (2 * b)) = a div b (is ?P)
      and a mod (2 * b) = a mod b (is ?Q)
proof -
  def w ≡ a div b mod 2 with parity have w-exhaust: w = 0 ∨ w = 1 by auto
  moreover have b ≤ a mod b + b
  proof -
    from ⟨0 < b⟩ pos-mod-sign have 0 ≤ a mod b by blast
    then have 0 + b ≤ a mod b + b by (rule add-right-mono)
    then show ?thesis by simp
  qed
  moreover note assms w-exhaust
  ultimately have w = 0 by auto
  with mod-w have mod: a mod (2 * b) = a mod b by simp
  have 2 * (a div (2 * b)) = a div b - w
      by (simp add: w-def mod-mult2-eq ac-simps)
  with ⟨w = 0⟩ have div: 2 * (a div (2 * b)) = a div b by simp
  then show ?P and ?Q
      by (simp-all add: div mod)
  qed

definition divmod :: num ⇒ num ⇒ 'a × 'a
where
  divmod m n = (numeral m div numeral n, numeral m mod numeral n)

lemma fst-divmod [simp]:
  fst (divmod m n) = numeral m div numeral n
by (simp add: divmod-def)

lemma snd-divmod [simp]:
  snd (divmod m n) = numeral m mod numeral n
by (simp add: divmod-def)

definition divmod-step :: num ⇒ 'a × 'a ⇒ 'a × 'a
where
  divmod-step l qr = (let (q, r) = qr
                        in if r ≥ numeral l then (2 * q + 1, r - numeral l)
                           else (2 * q, r))

This is a formulation of one step (referring to one digit position) in school-
method division: compare the dividend at the current digit position with
the remainder from previous division steps and evaluate accordingly.

lemma divmod-step-eq [code]:
  divmod-step l (q, r) = (if numeral l ≤ r
                          then (2 * q + 1, r - numeral l) else (2 * q, r))
This is a formulation of school-method division. If the divisor is smaller than the dividend, terminate. If not, shift the dividend to the right until termination occurs and then reiterate single division steps in the opposite direction.
ultimately show ?thesis by (simp only: divmod-def)
qed
then have divmod m n =
divmod-step n (numeral m div numeral (Num.Bit0 n),
numeral m mod numeral (Num.Bit0 n))
by (simp only: numeral.simps distrib mult-1)
then have divmod m n = divmod-step n (divmod m (Num.Bit0 n))
by (simp add: divmod-def)
with False show ?thesis by simp
qed

lemma divmod-cancel [code]:
divmod (Num.Bit0 m) (Num.Bit0 n) = (case divmod m n of (q, r) ⇒ (q, 2 * r)) (is ?P)
divmod (Num.Bit1 m) (Num.Bit0 n) = (case divmod m n of (q, r) ⇒ (q, 2 * r + 1)) (is ?Q)
proof –
have *: ⋀q. numeral (Num.Bit0 q) = 2 * numeral q
⋀q. numeral (Num.Bit1 q) = 2 * numeral q + 1
by (simp-all only: numeral-mult numeral.simps distrib simp-all)
have 1 div 2 = 0 1 mod 2 = 1 by (auto intro: div-less mod-less)
then show ?P and ?Q
by (simp-all add: prod-eq-iff split-def * [of m] * [of n] mod-mult-mult1
qed

end

hide-fact (open) diff-invert-add1 le-add-diff-inverse2 diff-zero
— restore simple accesses for more general variants of theorems

54.4 Division on nat

We define op div and op mod on nat by means of a characteristic relation with two input arguments m, n and two output arguments q(quotient) and r(remainder).

definition divmod-nat-rel :: nat ⇒ nat ⇒ nat × nat ⇒ bool where
divmod-nat-rel m n qr ⟷
m = fst qr * n + snd qr ∧
(if n = 0 then fst qr = 0 else if n > 0 then 0 ≤ snd qr ∧ snd qr < n else n < snd qr ∧ snd qr ≤ 0)
divmod-nat-rel is total:

lemma divmod-nat-rel-ex:
obtains q r where divmod-nat-rel m n (q, r)
proof (cases n = 0)
case True with that show thesis
by (auto simp add: divmod-nat-rel-def)
case False
have \( \exists q \ r. \ m = q \ast n + r \land r < n \)
proof (induct m)
  case 0 with \( n \neq 0 \)
  have \( (0::nat) = 0 \ast n + 0 \land 0 < n \) by simp
  then show \(?case\) by blast
next
  case (Suc m) then obtain \( q' \ r' \)
    where \( m = q' \ast n + r' \) and \( n: r' < n \) by auto
  then show \(?case\) proof (cases Suc r' < n)
    case True
    from \( m \ n \) have Suc m = \( q' \ast n + Suc \ r' \) by simp
    with True show \(?thesis\) by blast
next
  case False then have \( n \leq Suc \ r' \) by auto
  moreover from \( n \) have Suc r' \( \leq \) n by auto
  ultimately have \( n = Suc \ r' \) by auto
  with \( m \) have Suc m = Suc \( q' \ast n + 0 \) by simp
  with \( n \neq 0 \) show \(?thesis\) by blast
qed
qed
with that show thesis
  using \( n \neq 0 \) by (auto simp add: divmod-nat-rel-def)
qed

\textit{divmod-nat-rel} is injective:

\textbf{lemma} \textit{divmod-nat-rel-unique}:
  assumes \textit{divmod-nat-rel} \( m \ n \ qr \)
  and \textit{divmod-nat-rel} \( m \ n \ qr' \)
  shows \( qr = qr' \)
proof (cases \( n = 0 \))
  case True with assms show \(?thesis\)
    by (cases qr, cases qr')
    (simp add: divmod-nat-rel-def)
next
  case False
  have aux: \( \forall q \ r \ q' \ r'. \ q' \ast n + r' = q \ast n + r \Longrightarrow r < n \Longrightarrow q' \leq (q::nat) \)
    apply (rule leI)
    apply (subst less-if-Suc-add)
    apply (auto simp add: add-mult-distrib)
    done
  from \( n \neq 0 \) assms have \*: \( fst \ qr = fst \ qr' \)
    by (auto simp add: divmod-nat-rel-def intro: order-antisym dest: aux sym)
  with assms have \( snd \ qr = snd \ qr' \)
    by (simp add: divmod-nat-rel-def)
  with \* show \(?thesis\) by (cases qr, cases qr') simp
qed

We instantiate divisibility on the natural numbers by means of \textit{divmod-nat-rel}:
definition divmod-nat :: nat ⇒ nat ⇒ nat × nat where
  divmod-nat m n = (THE qr. divmod-nat-rel m n qr)

lemma divmod-nat-rel-divmod-nat:
  divmod-nat-rel m n (divmod-nat m n)
proof –
  from divmod-nat-rel-ex obtain qr where rel: divmod-nat-rel m n qr .
  then show ?thesis
qed

lemma divmod-nat-unique:
  assumes divmod-nat-rel m n qr
  shows divmod-nat m n = qr
  using assms by (auto intro: divmod-nat-rel-unique divmod-nat-rel-divmod-nat)

instantiation nat :: semiring-div
begin

definition div-nat where
  m div n = fst (divmod-nat m n)

lemma fst-divmod-nat [simp]:
  fst (divmod-nat m n) = m div n
  by (simp add: div-nat-def)

definition mod-nat where
  m mod n = snd (divmod-nat m n)

lemma snd-divmod-nat [simp]:
  snd (divmod-nat m n) = m mod n
  by (simp add: mod-nat-def)

lemma divmod-nat-rel-div-mod:
  divmod-nat m n = (m div n, m mod n)
  by (simp add: prod-eq-iff)

lemma div-nat-unique:
  assumes divmod-nat-rel m n (q, r)
  shows m div n = q
  using assms by (auto dest!: divmod-nat-unique simp add: prod-eq-iff)

lemma mod-nat-unique:
  assumes divmod-nat-rel m n (q, r)
  shows m mod n = r
  using assms by (auto dest!: divmod-nat-unique simp add: prod-eq-iff)

lemma divmod-nat-rel: divmod-nat-rel m n (m div n, m mod n)
using divmod-nat-rel-divmod-nat by (simp add: divmod-nat-div-mod)

lemma divmod-nat-zero: divmod-nat m 0 = (0, m)
  by (simp add: divmod-nat-unique divmod-nat-rel-def)

lemma divmod-nat-zero-left: divmod-nat 0 n = (0, 0)
  by (simp add: divmod-nat-unique divmod-nat-rel-def)

lemma divmod-nat-base: m < n ⇒ divmod-nat m n = (0, m)
  by (simp add: divmod-nat-unique divmod-nat-rel-def)

lemma divmod-nat-step:
  assumes 0 < n and n ≤ m
  shows divmod-nat m n = (Suc ((m − n) div n), (m − n) mod n)
proof (rule divmod-nat-unique)
  have divmod-nat-rel (m − n) n ((m − n) div n, (m − n) mod n)
    by (rule divmod-nat-rel)
  thus divmod-nat-rel m n (Suc ((m − n) div n), (m − n) mod n)
    unfolding divmod-nat-rel-def using assms by auto
qed

The ”recursion” equations for op div and op mod

lemma div-less [simp]:
  fixes m n :: nat
  assumes m < n
  shows m div n = 0
using assms divmod-nat-base by (simp add: prod-eq-iff)

lemma le-div-geq:
  fixes m n :: nat
  assumes 0 < n and n ≤ m
  shows m div n = Suc ((m − n) div n)
using assms divmod-nat-step by (simp add: prod-eq-iff)

lemma mod-less [simp]:
  fixes m n :: nat
  assumes m < n
  shows m mod n = m
using assms divmod-nat-base by (simp add: prod-eq-iff)

lemma le-mod-geq:
  fixes m n :: nat
  assumes n ≤ m
  shows m mod n = (m − n) mod n
using assms divmod-nat-step by (cases n = 0) (simp-all add: prod-eq-iff)

instance proof
  fix m n :: nat
  show m div n * n + m mod n = m
using divmod-nat-rel [of m n] by (simp add: divmod-nat-rel-def)

next
  fix m n q :: nat
  assume n ≠ 0
  then show (q + m * n) div n = m + q div n
    by (induct m) (simp-all add: le-div-geq)

next
  fix m n q :: nat
  assume m ≠ 0
  hence ∀a b. divmod-nat-rel n q (a, b) ⇒ divmod-nat-rel (m * n) (m * q) (a, m * b)
    unfolding divmod-nat-rel-def
    by (auto split: split-if-asm, simp-all add: algebra-simps)
  moreover from divmod-nat-rel have divmod-nat-rel n q (n div q, n mod q).
  ultimately have divmod-nat-rel (m * n) (m * q) (n div q, m * (n mod q)).
  thus (m * n) div (m * q) = n div q by (rule div-nat-unique)

next
  fix n :: nat show n div 0 = 0
    by (simp add: div-nat-def divmod-nat-zero)

next
  fix n :: nat show 0 div n = 0
    by (simp add: div-nat-def divmod-nat-zero-left)

qed

end

lemma divmod-nat-if [code]: divmod-nat m n = (if n = 0 ∨ m < n then (0, m) else let (q, r) = divmod-nat (m - n) n in (Suc q, r))
  by (simp add: prod-eq-iff case-prod-beta not-less le-div-geq le-mod-geq)

Simproc for cancelling op div and op mod

ML-file ∼/src/Provers/Arith/cancel-div-mod.ML

ML ⟨⟨
structure Cancel-Div-Mod-Nat = Cancel-Div-Mod
{
  val div-name = @{const-name div};
  val mod-name = @{const-name mod};
  val mk-binop = HLogic.mk-binop;
  val mk-plus = HLogic.mk-binop @{const-name Groups.plus};
  val dest-plus = HLogic.dest-bin @{const-name Groups.plus} HLogic.natT;
  fun mk-sum [] = HLogic.zero
    | mk-sum [t] = t
    | mk-sum (t :: ts) = mk-plus (t, mk-sum ts); 
  fun dest-sum tm =
    if HLogic.is-zero tm then []
    else (case try HLogic.dest-Suc tm of
SOME \( t \) => HOLogic.Suc-zero :: dest-sum \( t \)
| NONE =>
  \{ case try dest-plus tm of
    SOME (t, u) => dest-sum t @ dest-sum u
    | NONE => [tm]; \}

val div-mod-eqs = map mk-meta-eq [@{thm div-mod-equality}, @{thm div-mod-equality2}];

val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
  (@{thm add-0-left} :: @{thm add-0-right} :: @{thms ac-simps}))
)

simproc-setup cancel-div-mod-nat ((m::nat) + n) = (\langle K Cancel-Div-Mod-Nat.proc
\rangle

54.4.1 Quotient

lemma div-geq: \( 0 < n \Longrightarrow \neg m < n \Longrightarrow m \ div \ n = Suc ((m - n) \ div \ n) \)
by (simp add: le-div-geq linorder-not-less)

lemma div-if: \( 0 < n \Longrightarrow m \ div \ n = (if m < n then 0 else Suc ((m - n) \ div \ n)) \)
by (simp add: div-geq)

lemma div-mult-self-is-m [simp]: \( \langle n <\Longrightarrow (m*n) \ div \ n = (m::nat) \)
by simp

lemma div-mult-self1-is-m [simp]: \( \langle n <\Longrightarrow (n*m) \ div \ n = (m::nat) \)
by simp

lemma div-positive:
  fixes m n :: nat
  assumes n > 0
  assumes m \geq n
  shows m \ div \ n > 0
proof
  from \( \langle m \geq n \rangle \) obtain q where m = n + q
  by (auto simp add: le-iff-add)
  with \( \langle n > 0 \rangle \) show ?thesis by simp
qed

54.4.2 Remainder

lemma mod-less-divisor [simp]:
  fixes m n :: nat
  assumes n > 0
  shows m mod n < (n::nat)
  using assms diemod-nat-rel [of m n] unfolding diemod-nat-rel-def by auto

lemma mod-Suc-le-divisor [simp]:
theory "Divides"

m mod Suc n ≤ n 
using mod-less-divisor [of Suc n m] by arith

lemma mod-less-eq-dividend [simp]:
  fixes m n :: nat
  shows m mod n ≤ m
proof (rule add-leD2)
  from mod-div-equality have m div n * n + m mod n = m .
  then show m div n * n + m mod n ≤ m by auto
qed

lemma mod-geq: ¬ m < (n::nat) =⇒ m mod n = (m - n) mod n
by (simp add: le-mod-geq linorder-not-less)

lemma mod-if: m mod (n::nat) = (if m < n then m else (m - n) mod n)
by (simp add: le-mod-geq)

lemma mod-1 [simp]: m mod Suc 0 = 0
by (induct m) (simp-all add: mod-geq)

lemma mult-div-cancel: (n::nat) * (m div n) = m - (m mod n)
using mod-div-equality2 [of n m] by arith

lemma mod-le-divisor[simp]: 0 < n =⇒ m mod n ≤ (n::nat)
  apply (drule mod-less-divisor [where m = m])
  apply simp
  done

section{54.4.3 Quotient and Remainder}

lemma divmod-nat-rel-mult1-eq:
divmod-nat-rel b c (q, r) =⇒ divmod-nat-rel (a * b) c (a * q + a * r div c, a * r mod c)
by (auto simp add: split-ifs divmod-nat-rel-def algebra-simps)

lemma div-mult1-eq:
(a * b) div c = a * (b div c) + a * (b mod c) div (c::nat)
by (blast intro: divmod-nat-rel-mult1-eq [THEN div-nat-unique] divmod-nat-rel)

lemma divmod-nat-rel-add1-eq:
divmod-nat-rel a c (aq, ar) =⇒ divmod-nat-rel b c (bq, br)
=⇒ divmod-nat-rel (a + b) c (aq + bq + (ar + br) div c, (ar + br) mod c)
by (auto simp add: split-ifs divmod-nat-rel-def algebra-simps)

lemma div-add1-eq:
(a+b) div (c::nat) = a div c + b div c + ((a mod c + b mod c) div c)
by (blast intro: divmod-nat-rel-add1-eq [THEN div-nat-unique] divmod-nat-rel)
lemma mod-lemma: \[(\emptyset::nat) < c; r < b \] == b * (q mod c) + r < b * c
apply (cut-tac m = q and n = c in mod-less-divisor)
apply (drule-tac [2] m = q mod c in less-imp-Suc-add, auto)
apply (erule-tac P = %x. ?lhs < ?rhs x in ssubst)
apply (simp add: add-mult-distrib2)
done

lemma divmod-nat-rel-mult2-eq:
 divmod-nat-rel a b (q, r)
 == divmod-nat-rel a (b * c) (q div c, b *(q mod c) + r)
by (auto simp add: mult.commute mult.left-commute divmod-nat-rel-def add-mult-distrib2 [symmetric] mod-lemma)

lemma div-mult2-eq: a div (b * c) = (a div b) div (c::nat)
by (force simp add: divmod-nat-rel [THEN divmod-nat-rel-mult2-eq, THEN div-nat-unique])

lemma mod-mult2-eq: a mod (b * c) = b * (a div b mod c) + a mod (b::nat)
by (auto simp add: mult.commute divmod-nat-rel [THEN divmod-nat-rel-mult2-eq, THEN mod-nat-unique])

54.4.4 Further Facts about Quotient and Remainder

lemma div-1 [simp]: m div Suc 0 = m
by (induct m) (simp-all add: div-geq)

lemma div-le-mono [rule-format (no-asm)]:
 \forall m::nat. m \leq n ---\rightarrow (m div k) \leq (n div k)
apply (case-tac k=0, simp)
apply (induct n rule: nat-less-induct, clarify)
apply (case-tac n<k)
apply simp

apply (case-tac m<k)
apply simp
apply (simp add: div-geq diff-le mono)
done

lemma div-le-mono2: !!m::nat. 0<m; m\leq n \Rightarrow (k div n) \leq (k div m)
apply (subgoal-tac 0<n)
pref 2 apply simp
apply (induct-tac k rule: nat-less-induct)
apply (rename-tac k)
apply (case-tac k<n, simp)
apply (subgoal-tac ~ (k<m) )
prefer 2 apply simp
apply (simp add: div-geq)
apply (subgoal-tac (k−n) div n ≤ (k−m) div n)
prefer 2
apply (blast intro: div-le-mono diff-le-mono2)
apply (rule le-trans, simp)
apply (simp)
done

lemma div-le-dividend [simp]: m div n ≤ (m::nat)
apply (case-tac n=0, simp)
apply (subgoal-tac m div n ≤ m div 1, simp)
apply (rule div-le-mono2)
apply (simp-all (no_asm-simp))
done

lemma div-less-dividend [simp]:
[(1::nat) < n; 0 < m] ⇒ m div n < m
apply (induct m rule: nat-less-induct)
apply (rename-tac m)
apply (case-tac m<n, simp)
apply (subgoal-tac 0<n)
prefer 2 apply simp
apply (simp add: div-le-mono)
apply (case-tac n<m)
apply (subgoal-tac (m−n) div n < (m−n) )
apply (rule inst less-Suc+)
apply assumption
apply (simp-all)
done

A fact for the mutilated chess board

lemma mod-Suc: Suc(m) mod n = (if Suc(m mod n) = n then 0 else Suc(m mod n))
apply (case-tac n=0, simp)
apply (induct m rule: nat-less-induct)
apply (case-tac Suc (na) <n)
apply (frule lessI [THEN less-trans], simp add: less-not-refl3)
apply (simp add: linorder-not-less le-Suc-eq mod-geq)
apply (auto simp add: Suc-diff-le le-mod-geq)
done

lemma mod-eq-0-iff: (m mod d = 0) = (∃q::nat. m = d*q)
by (auto simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)
lemmas mod-eq-0D [dest!] = mod-eq-0-iff [THEN iffD1]

lemma mod-eqD:
  fixes m d r q :: nat
  assumes m mod d = r
  shows \exists q. m = r + q * d
proof
  from mod-div-equality obtain q where q * d + m mod d = m by blast
  with assms have m = r + q * d by simp
  then show \?thesis ..
qed

lemma split-div:
  P(n div k :: nat) =
  ((k = 0 \rightarrow P 0) \land (k \neq 0 \rightarrow (\forall i. \forall j<k. n = k*i + j \rightarrow P i)))
(is \?P = \?Q is = (∨ (\land (\land \rightarrow ?R))))
proof
  assume P: \?P
  show \?Q
  proof (cases)
    assume k = 0
    with P show \?Q by simp
  next
    assume not0: k \neq 0
    thus \?Q
  proof (simp, intro allI impI)
    fix i j
    assume n: n = k*i + j and j: j < k
    show P i
      proof (cases)
        assume i = 0
        with n j P show P i by simp
      next
        assume i \neq 0
        with not0 n j P show P i by(simp add:ac-simps)
      qed
    qed
  qed
next
  assume Q: \?Q
  show \?P
  proof (cases)
    assume k = 0
    with Q show \?P by simp
  next
    assume not0: k \neq 0
    with Q have R: \?R by simp
    from not0 R[THEN spec,of n div k,THEN spec, of n mod k]
show \( \texttt{P} \) by simp

qed

qed

lemma split-div-lemma:
assumes \( 0 < n \)
shows \( n * q \leq m \land m < n * \text{Suc } q \iff q = ((m :: \text{nat}) \div n) \) (is \( \texttt{lhs} \iff \texttt{rhs} \))

proof
assume \( \texttt{rhs} \)
with \texttt{mult-div-cancel} have \( \texttt{nq} : n * q = m - (m \mod n) \) by simp
then have \( \texttt{A} : n * q \leq m \) by simp
have \( n - (m \mod n) > 0 \) using \texttt{mod-less-divisor} assms by auto
then have \( m < n + (m - (m \mod n)) \) by simp
with \( \texttt{nq} \) have \( m < n + n * q \) by simp
then have \( \texttt{B} : m < n * \text{Suc } q \) by simp
from \( \texttt{A} \ \texttt{B} \) show \( \texttt{lhs} .. \)
next
assume \( \texttt{P} : \texttt{lhs} \)
then have \texttt{divmod-nat-rel} \( m \div n \) \( (q, m - n * q) \)
  unfolding \texttt{divmod-nat-rel-def} by (auto simp add: \texttt{ac-simps})
with \texttt{divmod-nat-rel-unique} \texttt{divmod-nat-rel} \( \texttt{of } m \ n \)
have \( (q, m - n * q) = (m \div n, m \mod n) \) by simp
then show \( \texttt{rhs} \) by simp
qed

theorem split-div':
\( \texttt{P} ((m :: \text{nat}) \div n) = ((n = 0 \land P 0) \lor
(\exists q. (n * q \leq m \land m < n * (\text{Suc } q)) \land P q)) \)
apply (case-tac \( 0 < n \))
apply (simp only: add: split-div-lemma)
apply simp-all
done

lemma split-mod:
\( \texttt{P(n mod k :: nat)} =
((k = 0 \rightarrow P n) \land (k \neq 0 \rightarrow (\forall i. \forall j. n = k*i + j \rightarrow P j))) \)
(is \( \texttt{?P = ?Q is - = (- \lor (- \rightarrow \text{?R}))} \))

proof
assume \( \texttt{P} : \texttt{?P} \)
show \( \texttt{?Q} \)
proof (cases)
  assume \( k = 0 \)
  with \( \texttt{P} \) show \( \texttt{?Q} \) by simp
next
assume \( \texttt{not0} : k \neq 0 \)
thus \( \texttt{?Q} \)
proof (simp, intro \texttt{allI impI})
  fix \( i \ j \)
THEORY "Divides"

assume \( n = k \cdot i + j \cdot j < k \)
thus \( P \) using \( \text{not0 } P \) by (simp add: ac-simps ac-simps)
qed

qed
next
assume \( Q: \ ?Q \)
show \( \ ?P \)
proof (cases)
  assume \( k = 0 \)
  with \( Q \) show \( \ ?P \) by simp
next
  assume \( \text{not0: } k \neq 0 \)
  with \( Q \) have \( R: \ ?R \) by simp
  from \( \text{not0 } R[\text{THEN spec,of } n \text{ die } k,\text{THEN spec, of } n \text{ mod } k] \)
  show \( \ ?P \) by simp
qed

qed

theorem mod-div-equality': \((m::nat) \text{ mod } n = m - (m \text{ div } n) \cdot n\)
using mod-div-equality [of \( m \ n \)] by arith

lemma div-mod-equality': \((m::nat) \text{ div } n \cdot n = m - m \text{ mod } n\)
using mod-div-equality [of \( m \ n \)] by arith

lemma div-eq-dividend-iff: \( a \neq 0 \Rightarrow (a :: nat) \text{ div } b = a \leftrightarrow b = 1 \)
apply rule
apply (cases \( b = 0 \))
apply simp-all
apply (metis (full-types) One-nat-def Suc-lessI div-less-dividend less-not-refl3)
done

54.4.5 An “induction” law for modulus arithmetic.

lemma mod-induct-0:
  assumes \( \forall i < p. \ P i \rightarrow P ((\text{Suc } i) \mod p) \)
  and base: \( P i \) and \( i < p \)
  shows \( P 0 \)
proof (rule ccontr)
  assume contra: \( \neg(P 0) \)
  from \( i \) have \( p: \ 0 < p \) by simp
  have \( \forall k. \ 0 < k \rightarrow \neg P (p-k) \) (is \( \forall k. \ ?A k \))
  proof
    fix \( k \)
    show \( ?A k \)
    proof (induct \( k \))
      show \( ?A 0 \) by simp — by contradiction
    next
      fix \( n \)
THEORY "Divides"

assume ih: ?A n
show ?A (Suc n)
proof (clarsimp)
  assume y: P (p – Suc n)
  have n: Suc n < p
  proof (rule ccontr)
    assume ¬(Suc n < p)
    hence p – Suc n = 0
    by simp
    with y contra show False
    by simp
  qed
hence n2: Suc (p – Suc n) = p – n by arith
from p have p – Suc n < p by arith
with y step have z: P ((Suc (p – Suc n)) mod p)
  by blast
show False
proof (cases n=0)
  case True
  with z n2 contra show ?thesis by simp
next
  case False
  with p have p – n < p by arith
  with z n2 False ih show ?thesis by simp
qed
qed
qed

moreover
from i obtain k where 0 < k ∧ i+k=p
  by (blast dest: less-imp-add-positive)
hence 0 < k ∧ i=p-k by auto
moreover
note base
ultimately
show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i< p. P i → P ((Suc i) mod p)
  and base: P i and i: i< p and j: j< p
  shows P j
proof
  have ∀ j< p. P j
  proof
    fix j
    show j< p → P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
by (auto elim: mod-induct-0)
next
fix k
assume ih: ?A k
show ?A (Suc k)
proof
assume suc: Suc k < p
hence k: k < p by simp
with ih have P k ..
with step k have P (Suc k mod p)
  by blast
moreover
from suc have Suc k mod p = Suc k
  by simp
ultimately
have P (Suc k) by simp
qed
qed
qed
with j show ?thesis by blast
qed

lemma div2-Suc-Suc [simp]: Suc (Suc m) div 2 = Suc (m div 2)
  by (simp add: numeral-2-eq-2 le-div-geq)

lemma mod2-Suc-Suc [simp]: Suc (Suc m) mod 2 = m mod 2
  by (simp add: numeral-2-eq-2 le-mod-geq)

lemma add-self-div-2 [simp]: (m + m) div 2 = (m::nat)
  by (simp add: mult-2 [symmetric])

lemma mod2-gr-0 [simp]: 0 < (m::nat) mod 2 ↔ m mod 2 = 1
proof
{ fix n :: nat have (n::nat) < 2 → n = 0 ∨ n = 1 by (cases n) simp-all }
moreover have m mod 2 < 2 by simp
ultimately have m mod 2 = 0 ∨ m mod 2 = 1 .
then show ?thesis by auto
qed

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma div-Suc-eq-div-add3 [simp]: m div (Suc (Suc (Suc n))) = m div (3+n)
  by (simp add: Suc3-eq-add-3)

lemma mod-Suc-eq-mod-add3 [simp]: m mod (Suc (Suc (Suc n))) = m mod (3+n)
  by (simp add: Suc3-eq-add-3)

lemma Suc-div-eq-add3-div: (Suc (Suc (Suc m))) div n = (3+m) div n
theory "Divides"

by (simp add: Suc3-eq-add-3)

lemma Suc-mod-eq-add3-mod: (Suc (Suc (Suc m))) mod n = (3+m) mod n
by (simp add: Suc3-eq-add-3)


lemma Suc-times-mod-eq: 1<k ==> Suc (k * m) mod k = 1
apply (induct m)
apply (simp-all add: mod-Suc)
done

declare Suc-times-mod-eq [of numeral w, simp] for w

lemma Suc-div-le-mono [simp]: n div k <= (Suc n) div k
by (simp add: div-le-mono)

lemma Suc-n-div-2-gt-zero [simp]: (0::nat) < n ==> 0 < (n + 1) div 2
by (cases n) simp-all

lemma div-2-gt-zero [simp]: assumes A: (1::nat) < n shows 0 < n div 2
proof
  from A have B: 0 < n - 1 and C: n - 1 + 1 = n by simp-all
  from Suc-n-div-2-gt-zero [OF B] C show ?thesis by simp
qed

lemma mod-mult-self3 [simp]: (k*n + m) mod n = m mod (n::nat)
by (simp add: ac-simps ac-simps)

lemma mod-mult-self4 [simp]: Suc (k*n + m) mod n =Suc m mod n
proof
  have Suc (k * n + m) mod n = (k * n + Suc m) mod n by simp
  also have ... = Suc m mod n by (rule mod-mult-self3)
  finally show ?thesis .
qed

lemma mod-Suc-eq-Suc-mod: Suc m mod n = Suc (m mod n) mod n
apply (subst mod-Suc [of m])
apply (subst mod-Suc [of m mod n], simp)
done

lemma mod-2-not-eq-zero-eq-one-nat:
  fixes n :: nat
  shows n mod 2 ≠ 0 <-> n mod 2 = 1
by simp

instance nat :: semiring-numeral-div
  by intro-classes (auto intro: div-positive simp add: mult-div-cancel mod-mult2-eq div-mult2-eq)

54.5 Division on int

definition divmod-int-rel :: int ⇒ int ⇒ int × int ⇒ bool where
  — definition of quotient and remainder
  divmod-int-rel a b = (λ(q, r). a = b * q + r ∧
  (if 0 < b then 0 ≤ r ∧ r < b else if b < 0 then b < r ∧ r ≤ 0 else q = 0))

The following algorithmic development actually echoes what has already been developed in class semiring-numeral-div. In the long run it seems better to derive division on int just from division on nat and instantiate semiring-numeral-div accordingly.

definition adjust :: int ⇒ int × int ⇒ int × int where
  — for the division algorithm
  adjust b = (λ(q, r). if 0 ≤ r − b then (2 * q + 1, r − b) else (2 * q, r))

algorithm for the case a≥0, b>0

function posDivAlg :: int ⇒ int ⇒ int × int where
  posDivAlg a b = (if a < b ∨ b ≤ 0 then (0, a)
  else adjust b (posDivAlg a (2 * b)))
  by auto
termination by (relation measure (λ(a, b). nat (a − b + 1)))
  (auto simp add: mult-2)

algorithm for the case a<0, b>0

function negDivAlg :: int ⇒ int ⇒ int × int where
  negDivAlg a b = (if 0 ≤ a + b ∨ b ≤ 0 then (−1, a + b)
  else adjust b (negDivAlg a (2 * b)))
  by auto
termination by (relation measure (λ(a, b). nat (− a − b)))
  (auto simp add: mult-2)

algorithm for the general case b ≠ (0::'a)

definition divmod-int :: int ⇒ int ⇒ int × int where
  — The full division algorithm considers all possible signs for a, b including the special case a=0, b<0 because negDivAlg requires a < (0::'a).
  divmod-int a b = (if 0 ≤ a then if 0 ≤ b then posDivAlg a b
  else if a = 0 then (0, 0)
  else apsnd uminus (negDivAlg (−a) (−b))
  else
  if 0 < b then negDivAlg a b
  else apsnd uminus (posDivAlg (−a) (−b)))
instantiation int :: Divides.div
begin

definition div-int where
  a div b = fst (divmod-int a b)

lemma fst-divmod-int [simp]:
  fst (divmod-int a b) = a div b
  by (simp add: div-int-def)

definition mod-int where
  a mod b = snd (divmod-int a b)

lemma snd-divmod-int [simp]:
  snd (divmod-int a b) = a mod b
  by (simp add: mod-int-def)

instance ..
end

lemma divmod-int-mod-div:
  divmod-int p q = (p div q, p mod q)
  by (simp add: prod-eq-iff)

Here is the division algorithm in ML:

fun posDivAlg (a,b) = 
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
    in if 0\leq r-b then (2*q+1, r-b) else (2*q, r)
    end

fun negDivAlg (a,b) = 
  if 0\leq a+b then (~1,a+b)
  else let val (q,r) = negDivAlg(a, 2*b)
    in if 0\leq r-b then (2*q+1, r-b) else (2*q, r)
    end;

fun negateSnd (q,r:int) = (q,\neg r);

fun divmod (a,b) = if 0\leq a then
  if b>0 then posDivAlg (a,b)
  else if a=0 then (0,0)
    else negateSnd (negDivAlg (~a,~b))
  else

54.5.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma unique-quotient-lemma:

\[ | b \cdot q' + r' \leq b \cdot q + r; \ 0 \leq r'; \ r' < b; \ r < b | \]

\[ \Longrightarrow \ q' \leq (q::int) \]

apply (subgoal-tac r' + b * (q' - q) \leq r)

prefer \ 2 \ apply (simp add: right-diff-distrib)

apply (erule-tac \ 2 \ order-le-less-trans)

apply (simp add: right-diff-distrib distrib-left)

prefer \ 2 \ apply (simp add: right-diff-distrib distrib-left)

apply (simp add: mult-less-cancel-left)

done

lemma unique-quotient-lemma-neg:

\[ | b \cdot q' + r' \leq b \cdot q + r; \ r \leq 0; \ b < r; \ b < r' | \]

\[ \Longrightarrow \ q \leq (q'::int) \]

by (rule-tac b = -b and r = -r' and r' = -r in unique-quotient-lemma, auto)

lemma unique-quotient:

\[ | \text{divmod-int-rel a b (q, r); divmod-int-rel a b (q', r') |} \]

\[ \Longrightarrow \ q = q' \]

apply (simp add: divmod-int-rel-def linorder-neq-iff split: split-if-asm)

apply (blast intro: order-antisym

dest: order-eq-refl [THEN unique-quotient-lemma]

order-eq-refl [THEN unique-quotient-lemma-neg] sym)+

done

lemma unique-remainder:

\[ | \text{divmod-int-rel a b (q, r); divmod-int-rel a b (q', r') |} \]

\[ \Longrightarrow \ r = r' \]

apply (subgoal-tac q = q')

apply (simp add: divmod-int-rel-def)

apply (blast intro: unique-quotient)

done

54.5.2 Correctness of posDivAlg, the Algorithm for Non-Negative Dividends

And positive divisors
THEORY "Divides"

lemma adjust-eq [simp]:
  adjust b (q, r) =
  (let diff = r - b in
   if 0 ≤ diff then (2 * q + 1, diff)
   else (2*q, r))
by (simp add: Let-def adjust-def)

declare posDivAlg.simps [simp del]

use with a simproc to avoid repeatedly proving the premise

lemma posDivAlg-eqn:
  0 < b ==>
  posDivAlg a b = (if a<b then (0,a) else adjust b (posDivAlg a (2*b)))
by (rule posDivAlg.simps [THEN trans], simp)

Correctness of posDivAlg: it computes quotients correctly

theorem posDivAlg-correct:
  assumes 0 ≤ a and 0 < b
  shows divmod-int-rel a b (posDivAlg a b)
  using assms
  apply (induct a b rule: posDivAlg.induct)
  apply auto
  apply (simp add: divmod-int-rel-def)
  apply (case_tac a < b)
  apply simp-all
  apply (erule splitE)
  apply (auto simp add: distrib-left Let-def ac-simps mult-2-right)
  done

54.5.3 Correctness of negDivAlg, the Algorithm for Negative Dividends

And positive divisors

declare negDivAlg.simps [simp del]

use with a simproc to avoid repeatedly proving the premise

lemma negDivAlg-eqn:
  0 < b ==>
  negDivAlg a b =
  (if 0≤a+b then (-1,a+b) else adjust b (negDivAlg a (2*b)))
by (rule negDivAlg.simps [THEN trans], simp)

lemma negDivAlg-correct:
  assumes a < 0 and b > 0
  shows divmod-int-rel a b (negDivAlg a b)
  using assms
apply (induct a b rule: negDivAlg.induct)
apply (auto simp add: linorder-not-le)
apply (simp add: divmod-int-rel-def)
apply (subst negDivAlg-eqn, assumption)
apply (case-tac a + b < (0::int))
apply simp-all
apply (erule splitE)
apply (auto simp add: distrib-left Let-def ac-simps mult-2-right)
done

54.5.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma divmod-int-rel-0: divmod-int-rel 0 b (0, 0)
by (auto simp add: divmod-int-rel-def linorder-neq-iff)

lemma posDivAlg-0: simp: posDivAlg 0 b = (0, 0)
by (subst posDivAlg.simps, auto)

lemma posDivAlg-0-right: simp: posDivAlg a 0 = (0, a)
by (subst posDivAlg.simps, auto)

lemma negDivAlg-minus1: simp: negDivAlg -1 b = (-1, b - 1)
by (auto simp add: divmod-int-rel-def)

lemma divmod-int-correct: divmod-int-rel a b (divmod-int a b)
apply (cases b = 0, simp add: divmod-int-def divmod-int-rel-def)
by (force simp add: linorder-neq-iff divmod-int-rel-0 divmod-int-def divmod-int-rel-neg
    posDivAlg-correct negDivAlg-correct)

lemma divmod-int-rel-div-mod: divmod-int-rel a b (a div b, a mod b)
using divmod-int-correct by (simp add: divmod-int-rel-def)

lemma div-int-unique: divmod-int-rel a b (q, r) ≤q div b = q
by (simp add: divmod-int-rel-def [THEN unique-quotient])

lemma mod-int-unique: divmod-int-rel a b (q, r) ≤q a mod b = r
by (simp add: divmod-int-rel-def [THEN unique-remainder])
instance  int :: ring-div

proof
  fix  a b :: int
  show  a div b * b + a mod b = a
       using  divmod-int-rel-div-mod [of a b]
       unfolding  divmod-int-rel-def by (simp add: mult.commute)

next
  fix  a b c :: int
  assume  b ≠ 0
  hence  divmod-int-rel (a + c * b) b (c + a div b, a mod b)
       using  divmod-int-rel-div-mod [of a b]
       unfolding  divmod-int-rel-def by (auto simp: algebra-simps)
  thus  (a + c * b) div b = c + a div b
       by (rule div-int-unique)

next
  fix  a b c :: int
  assume  c ≠ 0
  hence  q  r. divmod-int-rel a b (q, r)
       ==> divmod-int-rel (c * a) (c * b) (q, c * r)
       unfolding  divmod-int-rel-def
       by (rule linorder-cases [of 0 b], auto simp: algebra-simps
           mult-less-0-iff zero-less-mult-iff mult-strict-right-monotonic
           zero-le-mult-iff)
  hence  divmod-int-rel (c * a) (c * b) (a div b, c * (a mod b))
       using  divmod-int-rel-div-mod [of a b]
  thus  (c * a) div (c * b) = a div b
       by (rule div-int-unique)

next
  fix  a :: int
  show  a div 0 = 0
       by (rule div-int-unique, simp add: divmod-int-rel-def)

next
  fix  a :: int
  show  0 div a = 0
       by (rule div-int-unique, auto simp add: divmod-int-rel-def)

qed

Basic laws about division and remainder

lemma  zmod-zdiv-equality: (a::int) = b * (a div b) + (a mod b)
by (fact mod-div-equality2 [symmetric])

Tool setup

lemmas  add-0s = add-0-left add-0-right

ML ⟨⟨
structure Cancel-Div-Mod-Int = Cancel-Div-Mod
{
  val div-name = @{const-name div};
  val mod-name = @{const-name mod};
  val mk-binop = HOLogic.mk-binop;
val mk-sum = Arith-Data.mk-sum HOLogic.intT;
val dest-sum = Arith-Data.dest-sum;

val div-mod-eqs = map mk-meta-eq [@{thm div-mod-equality}, @{thm div-mod-equality2}];
val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
(@{thm diff-conv-add-uminus} :: @{thms add-0s} @ (@{thms ac-simps})))

lemma pos-mod-conj: (0::int) < b ==> 0 ≤ a mod b ∧ a mod b < b
  using divmod-int-correct [of a b]
  by (auto simp add: divmod-int-rel-def prod-eq-iff)
lemmas pos-mod-sign [simp] = pos-mod-conj [THEN conjunct1]
  and pos-mod-bound [simp] = pos-mod-conj [THEN conjunct2]

lemma neg-mod-conj: b < (0::int) ==> a mod b ≤ 0 ∧ b < a mod b
  using divmod-int-correct [of a b]
  by (auto simp add: divmod-int-rel-def prod-eq-iff)
lemmas neg-mod-sign [simp] = neg-mod-conj [THEN conjunct1]
  and neg-mod-bound [simp] = neg-mod-conj [THEN conjunct2]

54.5.5 General Properties of div and mod

lemma div-pos-pos-trivial: [| (0::int) ≤ a; a < b || |] ==> a div b = 0
  apply (rule div-int-unique)
  apply (auto simp add: divmod-int-rel-def)
  done

lemma div-neg-neg-trivial: [| a ≤ (0::int); b < a || |] ==> a div b = 0
  apply (rule div-int-unique)
  apply (auto simp add: divmod-int-rel-def)
  done

lemma div-pos-neg-trivial: [| (0::int) < a; a+b ≤ 0 || |] ==> a div b = -1
  apply (rule div-int-unique)
  apply (auto simp add: divmod-int-rel-def)
  done

lemma mod-pos-pos-trivial: [| (0::int) ≤ a; a < b || |] ==> a mod b = a
  apply (rule-tac q = 0 in mod-int-unique)
  apply (auto simp add: divmod-int-rel-def)
done

lemma mod-neg-neg-trivial: \( \langle a \leq (0::\text{int}); \ b < a \rangle \Rightarrow a \mod b = a \)
apply (rule-tac q = 0 in mod-int-unique)
apply (auto simp add: divmod-int-rel-def)
done

lemma mod-pos-neg-trivial: \( \langle (0::\text{int}) < a; \ a+b \leq 0 \rangle \Rightarrow a \mod b = a+b \)
apply (rule-tac q = -1 in mod-int-unique)
apply (auto simp add: divmod-int-rel-def)
done

There is no mod-neg-pos-trivial.

54.5.6 Laws for div and mod with Unary Minus

lemma zminus1-lemma:
\[ \text{divmod-int-rel} \ a \ b \ (q, r) \Rightarrow b \neq 0 \]
\[ \Rightarrow \text{divmod-int-rel} \ (-a) \ b \ (\text{if } r=0 \text{ then } -q \text{ else } -q - 1, \]
\[ \text{if } r=0 \text{ then } 0 \text{ else } b-r) \]
by (force simp add: split_ifs divmod-int-rel_def linorder_neq_iff right_diff_distrib)

lemma zdiv-zminus1-eq-if:
\( b \neq (0::\text{int}) \)
\[ \Rightarrow (-a) \div b = \]
\( (\text{if } a \mod b = 0 \text{ then } - (a \div b) \text{ else } - (a \div b) - 1) \)
by (blast intro: divmod-int-rel-div-mod [THEN zminus1-lemma, THEN div-int-unique])

lemma zmod-zminus1-eq-if:
\( (-a::\text{int}) \mod b = (\text{if } a \mod b = 0 \text{ then } 0 \text{ else } b - (a \mod b)) \)
apply (case-tac b = 0, simp)
apply (blast intro: divmod-int-rel-div-mod [THEN zminus1-lemma, THEN mod-int-unique])
done

lemma zmod-zminus1-not-zero:
fixes k l :: \text{int}
shows \( k \mod l \neq 0 \Rightarrow k \mod l \neq 0 \)
unfolding zmod-zminus1-eq-if by auto

lemma zdiv-zminus2-eq-if:
\( b \neq (0::\text{int}) \)
\[ \Rightarrow a \div (-b) = \]
\( (\text{if } a \mod b = 0 \text{ then } - (a \div b) \text{ else } - (a \div b) - 1) \)
by (simp add: zdiv-zminus1-eq-if div-minus_right)

lemma zmod-zminus2-eq-if:
\( a \mod (-b::\text{int}) = (\text{if } a \mod b = 0 \text{ then } 0 \text{ else } (a \mod b) - b) \)
by (simp add: zmod-zminus1-eq-if mod-minus_right)
theorem "Divides" 887

lemma zmod-zminus2-not-zero:
  fixes k l :: int
  shows k mod - l ≠ 0 ⇒ k mod l ≠ 0
  unfolding zmod-zminus2-eq-if by auto

54.5.7 Computation of Division and Remainder

lemma div-eq-minus1: (0::int) < b ==> -1 div b = -1
  by (simp add: div-int-def divmod-int-def)

lemma zmod-minus1: (0::int) < b ==> -1 mod b = b - 1
  by (simp add: mod-int-def divmod-int-def)

a positive, b positive

lemma div-pos-pos: [| 0 < a; 0 ≤ b |] ==> a div b = fst (posDivAlg a b)
  by (simp add: div-int-def divmod-int-def)

lemma mod-pos-pos: [| 0 < a; 0 ≤ b |] ==> a mod b = snd (posDivAlg a b)
  by (simp add: mod-int-def divmod-int-def)

a negative, b positive

lemma div-neg-pos: [| a < 0; 0 < b |] ==> a div b = fst (negDivAlg a b)
  by (simp add: div-int-def divmod-int-def)

lemma mod-neg-pos: [| a < 0; 0 < b |] ==> a mod b = snd (negDivAlg a b)
  by (simp add: mod-int-def divmod-int-def)

a negative, b negative

lemma div-pos-neg: 
  [| 0 < a; b < 0 |] ==> a div b = fst (apsnd uminus (negDivAlg (-a) (-b)))
  by (simp add: div-int-def divmod-int-def)

lemma mod-pos-neg: 
  [| 0 < a; b < 0 |] ==> a mod b = snd (apsnd uminus (negDivAlg (-a) (-b)))
  by (simp add: mod-int-def divmod-int-def)

a negative, b negative

lemma div-neg-neg: 
  [| a < 0; b ≤ 0 |] ==> a div b = fst (apsnd uminus (posDivAlg (-a) (-b)))
  by (simp add: div-int-def divmod-int-def)

lemma mod-neg-neg: 
  [| a < 0; b ≤ 0 |] ==> a mod b = snd (apsnd uminus (posDivAlg (-a) (-b)))
  by (simp add: mod-int-def divmod-int-def)

Simplify expressions in which div and mod combine numerical constants
lemma int-div-pos-eq: 
\[(a::int) = b * q + r; \ 0 \leq r; \ r < b] \implies a \div b = q\]
by (rule div-int-unique [of a b q r] simp add: divmod-int-rel-def)

lemma int-div-neg-eq: 
\[(a::int) = b * q + r; \ r \leq 0; \ b < r] \implies a \div b = q\]
by (rule div-int-unique [of a b q r], simp add: divmod-int-rel-def)

lemma int-mod-pos-eq: 
\[(a::int) = b * q + r; \ 0 \leq r; \ r < b] \implies a \mod b = r\]
by (rule mod-int-unique [of a b q r], simp add: divmod-int-rel-def)

lemma int-mod-neg-eq: 
\[(a::int) = b * q + r; \ r \leq 0; \ b < r] \implies a \mod b = r\]
by (rule mod-int-unique [of a b q r], simp add: divmod-int-rel-def)

numeral simprocs – high chance that these can be replaced by divmod algorithm from semiring-numeral-div
**THEORY “Divides”**

\[\text{simproc-setup binary-int-div} \]

\[
\begin{align*}
\text{(numeral } m \div \text{ numeral } n :& : \text{ int } | \\
\text{numeral } m \div - \text{ numeral } n :& : \text{ int } | \\
- \text{ numeral } m \div \text{ numeral } n :& : \text{ int } | \\
- \text{ numeral } m \div - \text{ numeral } n :& : \text{ int } \\
\langle K (\text{divmod-proc } \oplus \{\text{thm int-div-pos-eq}\} \oplus \{\text{thm int-div-neg-eq}\}) \rangle
\end{align*}
\]

\[\text{simproc-setup binary-int-mod} \]

\[
\begin{align*}
\text{(numeral } m \mod \text{ numeral } n :& : \text{ int } | \\
\text{numeral } m \mod - \text{ numeral } n :& : \text{ int } | \\
- \text{ numeral } m \mod \text{ numeral } n :& : \text{ int } | \\
- \text{ numeral } m \mod - \text{ numeral } n :& : \text{ int } \\
\langle K (\text{divmod-proc } \oplus \{\text{thm int-mod-pos-eq}\} \oplus \{\text{thm int-mod-neg-eq}\}) \rangle
\end{align*}
\]

**lemmas** posDivAlg-eqn-numeral [simp] = posDivAlg-eqn [of numeral v numeral w, OF zero-less-numeral] for v w

**lemmas** negDivAlg-eqn-numeral [simp] = negDivAlg-eqn [of numeral v - numeral w, OF zero-less-numeral] for v w

Special-case simplification: \(\pm 1 \div z\) and \(\pm 1 \mod z\)

**lemma** [simp]:

\[\text{shows } \text{div-one-bit0: } 1 \div \text{ numeral } (\text{Num.Bit0 } v) = (0 :: \text{ int})\]

\[\text{and } \text{mod-one-bit0: } 1 \mod \text{ numeral } (\text{Num.Bit0 } v) = (1 :: \text{ int})\]

\[\text{and } \text{div-one-bit1: } 1 \div \text{ numeral } (\text{Num.Bit1 } v) = (0 :: \text{ int})\]

\[\text{and } \text{mod-one-bit1: } 1 \mod \text{ numeral } (\text{Num.Bit1 } v) = (1 :: \text{ int})\]

\[\text{and } \text{div-one-neg-numeral: } 1 \div \text{ numeral } v = (-1 :: \text{ int})\]

\[\text{and } \text{mod-one-neg-numeral: } 1 \mod \text{ numeral } v = (1 :: \text{ int}) \mod \text{ numeral } v\]

by (simp-all del: arith-special

\[\text{add: } \text{div-pos-pos mod-pos-pos div-pos-neg mod-pos-neg posDivAlg-eqn}\]

**lemma** [simp]:

\[\text{shows } \text{div-neg-one-numeral: } -1 \div \text{ numeral } v = (-1 :: \text{ int})\]

\[\text{and } \text{mod-neg-one-numeral: } -1 \mod \text{ numeral } v = \text{ numeral } v - (1 :: \text{ int})\]

\[\text{and } \text{div-neg-one-bit0: } -1 \div \text{ numeral } (\text{Num.Bit0 } v) = (0 :: \text{ int})\]

\[\text{and } \text{mod-neg-one-bit0: } -1 \mod \text{ numeral } (\text{Num.Bit0 } v) = (-1 :: \text{ int})\]

\[\text{and } \text{div-neg-one-bit1: } -1 \div \text{ numeral } (\text{Num.Bit1 } v) = (0 :: \text{ int})\]

\[\text{and } \text{mod-neg-one-bit1: } -1 \mod \text{ numeral } (\text{Num.Bit1 } v) = (-1 :: \text{ int})\]

by (simp-all del: div-eq-minus1 zmod-minus1)

\[\text{by (simp-all add: div-eq-minus1 zmod-minus1)}\]

**54.5.8 Monotonicity in the First Argument (Dividend)**

**lemma** zdiv-mono1: [\(\left[ a \leq a' : \theta < (b::\text{int}) \right] \implies a \div b \leq a' \div b\]

apply (cut-tac a = a and b = b in zmod-zdiv-equality)

apply (cut-tac a = a' and b = b in zmod-zdiv-equality)

apply (rule unique-quotient-lemma)

apply (erule subst)

apply (erule subst, simp-all)
THEORY "Divides"

54.5.9 Monotonicity in the Second Argument (Divisor)

lemma q-pos-lemma:
\[ | 0 \leq b' \cdot q' + r'; r' < b' \cdot 0 < b' | \implies 0 \leq (q'::int) \]
apply (frule q-pos-lemma, assumption+)
apply (subgoal-tac b' \cdot q < b' \cdot (q' + 1))
apply (simp add: mult-less-cancel-left)
apply (subgoal-tac b' \cdot q = r' - r + b' \cdot q')
prefer 2 apply simp
apply (simp (no-asm-simp) add: distrib-left)
apply (subgoal-tac add.commute, rule add-less-le-mono, arith)
apply (rule mult-right-mono, auto)
done

lemma zdiv-mono2-lemma:
\[ | b \cdot q + r = b' \cdot q' + r'; 0 \leq b' \cdot q' + r'; r' < b'; 0 \leq r'; 0 < b'; b' \leq b | \implies q \leq (0'::int) \]
apply (rule q-pos-lemma, assumption+)
apply (subgoal-tac b' \cdot q < b' \cdot (q' + 1))
apply (simp add: mult-less-cancel-left)
apply (subgoal-tac b' \cdot q = r' - r + b' \cdot q')
prefer 2 apply simp
apply (simp (no-asm-simp) add: distrib-left)
apply (subgoal-tac add.commute, rule add-less-le-mono, arith)
apply (rule mult-right-mono, auto)
done
lemma \texttt{zdiv-mono2-neg-lemma}:\[
\begin{align*}
\forall b \cdot q + r = b' q' + r';
\quad b' q' + r' < 0; \\
r < b; 0 \leq r'; 0 < b'; b' \leq b \Rightarrow q' \leq (q::int)
\end{align*}
\]
apply (frule \texttt{q-neg-lemma}, assumption+) 
apply (subgoal-tac \texttt{b*q' < b*(q + 1)}) 
apply (simp add: mult-less-cancel-left) 
apply (simp add: distrib-left) 
apply (subgoal-tac \texttt{b*q' < b*(q + 1)}) 
prefer 2 apply (simp add: \texttt{mult-right-mono-neg}, arith) 
done

lemma \texttt{zdiv-mono2-neg}:\[
\begin{align*}
a < (0::int); 0 < b'; b' \leq b \Rightarrow a \div b' \leq a \div b
\end{align*}
\]
apply (cut-tac \texttt{a = a and b = b} in \texttt{zmod-zdiv-equality}) 
apply (cut-tac \texttt{a = a and b = b'} in \texttt{zmod-zdiv-equality}) 
apply (rule \texttt{zdiv-mono2-neg-lemma}) 
apply (erule subst) 
apply (erule subst, simp-all) 
done

\textbf{54.5.10 More Algebraic Laws for div and mod}

proving \((a*b) \div c = a \ast (b \div c) + a \ast (\mod{b c})\)

\textbf{lemma \texttt{zmult1-lemma}}:\[
\begin{align*}
\forall b \cdot c \cdot (q, r) \Rightarrow \text{divmod-int-rel} (a \ast b) c (a \ast q + a \ast r \div c, a \ast r \mod c)
\end{align*}
\]
by (auto simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left \texttt{ac-simps})

\textbf{lemma \texttt{zdiv-zmult1-eq}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zadd1-lemma}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zdiv-zadd1-eq}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zadd1-lemma}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zdiv-zadd1-eq}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zadd1-lemma}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zdiv-zadd1-eq}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)

\textbf{lemma \texttt{zadd1-lemma}}:\[
\begin{align*}
\forall a \cdot \forall b \cdot \forall c \cdot (\text{divmod-int-rel} a c (aq, ar); \text{divmod-int-rel} b c (bq, br)) \Rightarrow \text{divmod-int-rel} (a+b) c (aq + bq + (ar+br) \div c, (ar+br) \mod c)
\end{align*}
\]
by (force simp add: split-ifs divmod-int-rel-def linorder-neq-iff distrib-left)
lemma posDivAlg-div-mod:
assumes \( k \geq 0 \) and \( l \geq 0 \)
shows \( \text{posDivAlg} k l = (k \div l, k \mod l) \)
proof (cases \( l = 0 \))
case True then show \(?thesis\) by (simp add: posDivAlg.simps)
next
case False with assms posDivAlg-correct
have divmod-int-rel \( k l \) \((\text{fst} (\text{posDivAlg} k l), \text{snd} (\text{posDivAlg} k l))\)
by simp
from div-int-unique [OF this] mod-int-unique [OF this]
show ?thesis by simp
qed

lemma negDivAlg-div-mod:
assumes \( k < 0 \) and \( l > 0 \)
shows \( \text{negDivAlg} k l = (k \div l, k \mod l) \)
proof
from assms have \( l \neq 0 \) by simp
from assms negDivAlg-correct
have divmod-int-rel \( k l \) \((\text{fst} (\text{negDivAlg} k l), \text{snd} (\text{negDivAlg} k l))\)
by simp
from div-int-unique [OF this] mod-int-unique [OF this]
show ?thesis by simp
qed

lemma zmod-eq-0-iff: \((m \mod d = 0)\) = \((\exists q :: \text{int}. m = d \times q)\)
by (simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

lemmas zmod-eq-0D [dest!] = zmod-eq-0-iff [THEN iffD1]

lemma zmod-zdiv-equality': \((m::\text{int}) \mod n = m - (m \div n) \times n\)
using mod-div-equality [of m n] by arith

54.5.11 Proving \( a \div (b \times c) = a \div b \div c \)

first, four lemmas to bound the remainder for the cases \( b|0 \) and \( b|0 \)

lemma zmult2-lemma-aux1: \[ (\theta::\text{int}) < c \; ; \; b < r \; ; \; r \leq \theta \] \implies \( b \times c < b \times (q \mod c) + r \)
apply (subgoal-tac \( b \times (c - q \mod c) < r \times 1 \))
apply (simp add: algebra-simps)
apply (rule order-le-less-trans)
apply (erule-tac [2] mult-strict-right-mono)
apply (rule mult-left-mono-neg)
using add1-zle-eq[of q mod c] apply (simp add: algebra-simps)
apply (simp)
apply (simp)
done

lemma zmult2-lemma-aux2: 
  \([| (0::int) < c; \ b < r; \ r \leq 0 |] \Longrightarrow b \ast (q \mod c) + r \leq 0\)
apply (subgoal-tac \ b \ast (q \mod c) \leq 0)
apply arith
apply (simp add: mult-le-0-iff)
done

lemma zmult2-lemma-aux3: 
  \([| (0::int) < c; \ b < r \leq 0 |] \Longrightarrow 0 \leq b \ast (q \mod c) + r \leq 0\)
apply (subgoal-tac \ 0 \leq b \ast (q \mod c))
apply arith
apply (simp add: zero-le-mult-iff)
done

lemma zmult2-lemma-aux4: 
  \([| (0::int) < c; \ b < r \leq 0 |] \Longrightarrow b \ast (q \mod c) + r < b \ast c\)
apply (subgoal-tac \ r \ast 1 < \ b \ast (c \mod q \mod c))
apply (simp add: right-diff-distrib)
apply (rule order-less-le-trans)
apply (erule mult-strict-right-mono)
apply (rule_tac [\ 2] mult-left-mono)
apply simp
using add1-zle-eq [of q \mod c] apply (simp add: algebra-simps)
apply simp
done

lemma zmult2-lemma: 
  \([| divmod-int-rel a b \ (q, r); \ 0 < c |] \Longrightarrow \ divmod-int-rel a \ (b \ast c) \ (q \div c, b \ast (q \mod c) + r)\)
by (auto simp add: mult.assoc divmod-int-rel-def linorder-neq-iff
zero-less-mult-iff distrib-left [symmetric]
zmult2-lemma-aux1 zmult2-lemma-aux2 zmult2-lemma-aux3
zmult2-lemma-aux4 mult-less-0-iff split: split-if-asm)

lemma zdiv-zmult2-eq:
fixes a b c :: int
shows \(0 \leq c \Longrightarrow a \div (b \ast c) = (a \div b) \div c\)
apply (case_tac \ b = 0, simp)
apply (force simp add: le-less divmod-int-rel-div-mod [THEN zmult2-lemma, THEN
div-int-unique])
done

lemma zmod-zmult2-eq:
fixes a b c :: int
shows \(0 \leq c \Longrightarrow a \mod (b \ast c) = b \ast (a \div b \mod c) + a \mod b\)
apply (case_tac \ b = 0, simp)
apply (force simp add: le-less divmod-int-rel-div-mod [THEN zmult2-lemma, THEN
div-int-unique])
done
lemma \textit{div-pos-geq}:
\begin{verbatim}
  fixes k l :: int
  assumes 0 < l and l ≤ k
  shows k div l = (k - l) div l + 1
proof –
  have k = (k - l) + l by simp
  then obtain j where k: k = j + l ..
  with assms show ?thesis by simp
qed
\end{verbatim}

lemma \textit{mod-pos-geq}:
\begin{verbatim}
  fixes k l :: int
  assumes 0 < l and l ≤ k
  shows k mod l = (k - l) mod l
proof –
  have k = (k - l) + l by simp
  then obtain j where k: k = j + l ..
  with assms show ?thesis by simp
qed
\end{verbatim}

54.5.12 Splitting Rules for \textit{div} and \textit{mod}

The proofs of the two lemmas below are essentially identical

lemma \textit{split-pos-lemma}:
\begin{verbatim}
  0 < k ==> 
  P(n div k :: int)(n mod k) = (\forall i j. 0 ≤ j & j < k & n = k * i + j ---> P i j)
apply (rule iffI, clarify)
apply (erule-tac P = P ?x ?y in rev-mp)
apply (subst mod-add-eq)
apply (subst zdiv-zadd1-eq)
apply (simp add: div-pos-pos-trivial mod-pos-pos-trivial)
converse direction
apply (drule-tac x = n div k in spec)
apply (drule-tac x = n mod k in spec, simp)
done
\end{verbatim}

lemma \textit{split-neg-lemma}:
\begin{verbatim}
  k < 0 ==> 
  P(n div k :: int)(n mod k) = (\forall i j. k < j & j ≤ 0 & n = k * i + j ---> P i j)
apply (rule iffI, clarify)
apply (erule-tac P = P ?x ?y in rev-mp)
apply (subst mod-add-eq)
apply (subst zdiv-zadd1-eq)
apply (simp add: div-neg-neg-trivial mod-neg-neg-trivial)
\end{verbatim}
THEORY “Divides”

converse direction

apply (drule-tac x = n div k in spec)
apply (drule-tac x = n mod k in spec, simp)
done

lemma split-zdiv:
\[ P(n \text{ div } k :: \text{int}) = \]
\[
((k = 0 \rightarrow P \ 0) \&
(\neg k \rightarrow (\forall i j. 0 \leq j \& j < k \& n = k \ast i + j \rightarrow P \ i)) \&
(\neg k \rightarrow (\forall i j. k < j \& j \leq 0 \& n = k \ast i + j \rightarrow P \ i)))
\]
apply (case-tac k=0, simp)
apply (simp only: linorder-neq_iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl: %x y. P x]
split-neg-lemma [of concl: %x y. P x])
done

lemma split-zmod:
\[ P(n \text{ mod } k :: \text{int}) = \]
\[
((k = 0 \rightarrow P \ n) \&
(\neg k \rightarrow (\forall i j. 0 \leq j \& j < k \& n = k \ast i + j \rightarrow P \ j)) \&
(\neg k \rightarrow (\forall i j. k < j \& j \leq 0 \& n = k \ast i + j \rightarrow P \ j)))
\]
apply (case-tac k=0, simp)
apply (simp only: linorder-neq_iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl: %x y. P y]
split-neg-lemma [of concl: %x y. P y])
done

Enable (lin)arith to deal with op div and op mod when these are applied to some constant that is of the form numeral k:

declare split-zdiv [of - - numeral k, arith-split] for k
declare split-zmod [of - - numeral k, arith-split] for k

54.5.13 Computing div and mod with shifting

lemma pos-divmod-int-rel-mult-2:
assumes 0 \leq b
assumes divmod-int-rel a b (q, r)
shows divmod-int-rel (1 + 2*a) (2*b) (q, 1 + 2*r)
using assms unfolding divmod-int-rel-def by auto

declaration ⟨⟨ K (Lin-Arith.add-simps @{thms uminus-numeral-One}) ⟩⟩

lemma neg-divmod-int-rel-mult-2:
assumes b \leq 0
assumes divmod-int-rel (a + 1) b (q, r)
shows divmod-int-rel (1 + 2*a) (2*b) (q, 2*r - 1)
using assms unfolding divmod-int-rel-def by auto
computing div by shifting

**lemma** pos-zdiv-mult-2: \((0::\text{int}) \leq a \implies (1 + 2 \cdot b) \div (2 \cdot a) = b \div a\)

**using** pos-divmod-int-rel-mult-2 [OF - divmod-int-rel-div-mod]

**by** (rule div-int-unique)

**lemma** neg-zdiv-mult-2:

**assumes** \(a \leq (0::\text{int})\)

**shows** \((1 + 2 \cdot b) \div (2 \cdot a) = (b+1) \div a\)

**using** neg-divmod-int-rel-mult-2 [OF A divmod-int-rel-div-mod]

**by** (rule div-int-unique)

**lemma** zdiv-numeral-Bit0 [simp]:

\[
\text{numeral (Num.Bit0 v)} \div \text{numeral (Num.Bit0 w)} = \text{numeral v div (numeral w :: int)}
\]

**unfolding** numeral.simps

**unfolding** mult-2 [symmetric]

**by** (rule div-mul1, simp)

**lemma** zdiv-numeral-Bit1 [simp]:

\[
\text{numeral (Num.Bit1 v)} \div \text{numeral (Num.Bit0 w)} = \text{(numeral v div (numeral w :: int))}
\]

**unfolding** numeral.simps

**unfolding** mult-2 [symmetric] add.commute [of - 1]

**by** (rule pos-zdiv-mult-2, simp)

**lemma** pos-zmod-mult-2:

**fixes** \(a \cdot b :: \text{int}\)

**assumes** \(0 \leq a\)

**shows** \((1 + 2 \cdot b) \mod (2 \cdot a) = 1 + 2 \cdot (b \mod a)\)

**using** pos-divmod-int-rel-mult-2 [OF assms divmod-int-rel-div-mod]

**by** (rule mod-int-unique)

**lemma** neg-zmod-mult-2:

**fixes** \(a \cdot b :: \text{int}\)

**assumes** \(a \leq 0\)

**shows** \((1 + 2 \cdot b) \mod (2 \cdot a) = 2 \cdot ((b + 1) \mod a) - 1\)

**using** neg-divmod-int-rel-mult-2 [OF assms divmod-int-rel-div-mod]

**by** (rule mod-int-unique)

**lemma** zmod-numeral-Bit0 [simp]:

\[
\text{numeral (Num.Bit0 v)} \mod \text{numeral (Num.Bit0 w)} = (2::\text{int}) \cdot (\text{numeral v mod numeral w})
\]

**unfolding** numeral-Bit0 [of v] numeral-Bit0 [of w]

**unfolding** mult-2 [symmetric] by (rule mod-mul1)

**lemma** zmod-numeral-Bit1 [simp]:

\[
\text{numeral (Num.Bit1 v)} \mod \text{numeral (Num.Bit0 w)} = 2 \cdot (\text{numeral v mod numeral w}) + (1::\text{int})
\]

**unfolding** numeral-Bit1 [of v] numeral-Bit0 [of w]
unfolding mult-2 [symmetric] add.commute [of - 1]
by (rule pos-zmod-mult-2, simp)

lemma zdiv-eq-0-iff:
(i::int) div k = 0 ↔ k=0 ∨ 0≤i ∧ i<k ∨ i≤0 ∧ k<i (is ?L = ?R)
proof
  assume ?L
  have ?L → ?R by (rule split-zdiv[THEN iffD2]) simp
  with ⟨?L⟩ show ?R by blast
next
  assume ?R thus ?L by (auto simp: div-pos-pos-trivial div-neg-neg-trivial)
qed

54.5.14 Quotients of Signs

lemma div-neg-pos-less0: [| a < (0::int); 0 < b |] ==> a div b < 0
apply (subgoal-tac a div b ≤−1, force)
apply (rule order-trans)
apply (rule-tac a' = −1 in zdiv-mono1)
apply (auto simp add: div-eq-minus1)
done

lemma div-nonneg-neg-le0: [| (0::int) ≤ a; b < 0 |] ==> a div b ≤ 0
by (drule zdiv-mono1-neg, auto)

lemma div-nonpos-pos-le0: [| (a::int) ≤ 0; b > 0 |] ==> a div b ≤ 0
by (drule zdiv-mono1, auto)

Now for some equivalences of the form a div b >=< 0 ↔ ... conditional
upon the sign of a or b. There are many more. They should all be simp
rules unless that causes too much search.

lemma pos-imp-zdiv-nonneg-iff: (0::int) < b ==> (0 ≤ a div b) = (0 ≤ a)
apply auto
apply (drule-tac [2] zdiv-mono1)
apply (auto simp add: linorder-not-less [symmetric])
apply (blast intro: div-neg-pos-less0)
done

lemma neg-imp-zdiv-nonneg-iff:
  b < (0::int) ==> (0 ≤ a div b) = (a ≤ (0::int))
apply (subst div-minus-minus [symmetric])
apply (subst pos-imp-zdiv-nonneg-iff, auto)
done

lemma pos-imp-zdiv-neg-iff: (0::int) < b ==> (a div b < 0) = (a < 0)
by (simp add: linorder-not-le [symmetric] pos-imp-zdiv-nonneg-iff)
**THEORY “Divides”**

**lemma** pos-imp-zdiv-pos-iff:
\[ 0 < k \implies 0 < (i :: int) \div k \iff k \leq i \]
using pos-imp-zdiv-nonneg-iff[of k i] zdiv-eq-0-iff[of i k]
by arith

**lemma** neg-imp-zdiv-neg-iff:
\[ b < (0 :: int) \implies (a \div b < 0) = (0 < a) \]
by (simp add: linorder-not-le [symmetric] neg-imp-zdiv-nonneg-iff)

**lemma** nonneg1-imp-zdiv-pos-iff:
\[ (0 :: int) < a \implies (a \div b > 0) = (a \geq b \land b > 0) \]
apply rule
apply rule
  using div-pos-pos-trivial[of a b] apply arith
apply (cases b = 0) apply simp
using div-nonneg-neg-le0[of a b] apply arith
using int-one-le-iff-zero-less[of a div b] zdiv-mono1[of b a b] apply simp
done

**lemma** zmod-le-nonneg-dividend:
\[ (m :: int) \geq 0 \implies m \mod k \leq m \]
apply (rule split-zmod[THEN iffD2])
apply (fastforce dest: q-pos-lemma intro: split-mult-pos-le)
done

### 54.5.15 The Divides Relation

**lemma** dvd-neg-numeral-left [simp]:
**fixes** y :: 'a::comm-ring-1
**shows** (- numeral k) dvd y \iff (numeral k) dvd y
**by** (fact minus-dvd-iff)

**lemma** dvd-neg-numeral-right [simp]:
**fixes** x :: 'a::comm-ring-1
**shows** x dvd (- numeral k) \iff x dvd (numeral k)
**by** (fact dvd-minus-iff)

**lemmas** dvd-eq-mod-eq-0-numeral [simp] =
  dvd-eq-mod-0-numeral [of numeral x numeral y] for x y

### 54.5.16 Further properties

**lemma** zmult-div-cancel:
\[ (n :: int) * (m \div n) = m - (m \mod n) \]
using zmod-zdiv-equality[where a=m and b=n]
**by** (simp add: algebra-simps)

**lemma** zdiv-int: int (a \div b) = (int a) \div (int b)
**apply** (subst split-div, auto)
**apply** (subst split-zdiv, auto)
apply (rule-tac a=int (b * i) + int j and b=int b and r=int j and r'=ja in unique-quotient)
apply (auto simp add: divmod-int-rel-def of-nat-mult)
done

lemma zmod-int: int (a mod b) = (int a) mod (int b)
apply (subst split-mod, auto)
apply (subst split-zmod, auto)
apply (rule-tac a=int (b * i) + int j and b=int b and q=int i and q'=ia in unique-remainder)
apply (auto simp add: divmod-int-rel-def of-nat-mult)
done

lemma abs-div: (y::int) dvd x \implies abs (x div y) = abs x div abs y
by (unfold dvd-def, cases y=0, auto simp add: abs-mult)

Suggested by Matthias Daum

lemma int-power-div-base:
[\[0 < m; 0 < k]\] \implies k ` m div k = (k::int) ` (m - Suc 0)
apply (erule sssubst)
apply (simp only: power-add)
apply simp-all
done

by Brian Huffman

lemma zminus-zmod: - ((x::int) mod m) mod m = - x mod m
by (rule mod-minus-eq [symmetric])

lemma zdiff-zmod-left: (x mod m - y) mod m = (x - y) mod (m::int)
by (rule mod-diff-left-eq [symmetric])

lemma zdiff-zmod-right: (x - y mod m) mod m = (x - y) mod (m::int)
by (rule mod-diff-right-eq [symmetric])

lemmas zmod-simps =
mod-add-left-eq [symmetric]
mod-add-right-eq [symmetric]
mod-mult-right-eq [symmetric]
mod-mult-left-eq [symmetric]
power-mod
zminus-zmod zdiff-zmod-left zdiff-zmod-right

Distributive laws for function nat.

lemma nat-div-distrib: 0 \leq x \implies nat (x div y) = nat x div nat y
apply (rule linorder-cases [of y 0])
apply (simp add: div-nonneg-neg-le0)
apply simp
apply (simp add: nat-eq-iff pos-imp-zdiv-nonneg-iff zdiv-int)
THEORY “Divides”

done

lemma nat-mod-distrib:
\[
\begin{align*}
0 \leq x; 0 \leq y \Rightarrow \text{nat}(x \mod y) = \text{nat}(x) \mod \text{nat}(y)
\end{align*}
\]
apply (case_tac y = 0, simp)
apply (simp add: nat-eq-iff zmod-int)
done

transfer setup

lemma transfer-nat-int-functions:
\[
\begin{align*}
(x::int) \geq 0 \Rightarrow y \geq 0 \Rightarrow (\text{nat } x) \text{ div } (\text{nat } y) = \text{nat } (x \text{ div } y) \\
(x::int) \geq 0 \Rightarrow y \geq 0 \Rightarrow (\text{nat } x) \text{ mod } (\text{nat } y) = \text{nat } (x \text{ mod } y)
\end{align*}
\]
by (auto simp add: nat-div-distrib nat-mod-distrib)

lemma transfer-nat-int-function-closures:
\[
\begin{align*}
(x::int) \geq 0 \Rightarrow y \geq 0 \Rightarrow x \text{ div } y \geq 0 \\
(x::int) \geq 0 \Rightarrow y \geq 0 \Rightarrow x \text{ mod } y \geq 0
\end{align*}
\]
apply (cases y = 0)
apply (auto simp add: pos-imp-zdiv-nonneg-iff)
apply (cases y = 0)
done

declare transfer-morphism-nat-int [transfer add return:
transfer-nat-int-functions
transfer-nat-int-function-closures]

lemma transfer-int-nat-functions:
\[
\begin{align*}
\text{int } x \text{ div } (\text{int } y) = \text{int } (x \text{ div } y) \\
\text{int } x \text{ mod } (\text{int } y) = \text{int } (x \text{ mod } y)
\end{align*}
\]
by (auto simp add: zdiv-int zmod-int)

lemma transfer-int-nat-function-closures:
\[
\begin{align*}
is-nat x \Rightarrow is-nat y \Rightarrow is-nat (x \text{ div } y) \\
is-nat x \Rightarrow is-nat y \Rightarrow is-nat (x \text{ mod } y)
\end{align*}
\]
by (simp-all only: is-nat-def transfer-nat-int-function-closures)

declare transfer-morphism-int-nat [transfer add return:
transfer-int-nat-functions
transfer-int-nat-function-closures]

Suggested by Matthias Daum

lemma int-div-less-self: \[
\begin{align*}
0 < x; 1 < k \Rightarrow x \text{ div } k < (x::int)
\end{align*}
\]
apply (subgoal_tac nat x div nat k < nat x)
apply (simp add: nat-div-distrib [symmetric])
apply (rule Divides.div-less-dividend, simp-all)
done

lemma \textit{zmod-eq-dvd-iff}: (x::int) \ mod n = y \ mod n \ \longleftrightarrow \ n \ \operatorname{dvd} \ x - y

proof
  assume \(H\): \(x \mod n = y \mod n\)
  hence \(x \mod n - y \mod n = 0\) \ by simp
  hence \((x \mod n - y \mod n) \mod n = 0\) \ by simp
  hence \((x - y) \mod n = 0\) \ by \(\text{(simp add: mod-diff-eq[symmetric])}\)
  thus \(n \ \operatorname{dvd} x - y\) \ by \(\text{(simp add: dvd-eq-mod-eq-0)}\)
next
  assume \(H\): \(n \ \operatorname{dvd} x - y\)
  then obtain \(k\) where \(k\): \(x - y = n \ast k\) \ unfolding \(\text{dvd-def}\) \ by blast
  hence \(x = n \ast k + y\) \ by simp
  hence \(x \mod n = (n \ast k + y) \mod n\) \ by simp
  thus \(x \mod n = y \mod n\) \ by \(\text{(simp add: mod-add-left-eq)}\)
qed

lemma \textit{nat-mod-eq-lemma}: assumes \(xyn\): \((x::nat) \mod n = y \mod n\) and \(xy\): \(y \leq x\)

shows \(\exists \ q. \ x = y + n \ast q\)

proof
  from \(xy\) have \(\text{th}: \ \text{int} \ x - \text{int} \ y = \text{int} \ (x - y)\) \ by simp
  from \(xyn\) have \(\text{int} \ x \mod \text{int} \ n = \text{int} \ y \mod \text{int} \ n\)
    \ by \(\text{(simp add: zmod-int [symmetric])}\)
  hence \(\text{int} \ n \ \operatorname{dvd} \text{int} \ x - \text{int} \ y\) \ by \(\text{(simp only: zmod-eq-dvd-iff [symmetric])}\)
  hence \(n \ \operatorname{dvd} x - y\) \ by \(\text{(simp add: th zdvd-int)}\)
  then show \(?\text{thesis using xy unfolding dvd-def apply clarsimp apply (rule-tac x=k in exI)}\) \ by arith
qed

lemma \textit{nat-mod-eq-iff}: \((x::nat) \mod n = y \mod n\) \ \longleftrightarrow \ ((\exists q1 q2. x + n \ast q1 = y + n \ast q2)\)

(is \?\text{lhs = ?rhs})

proof
  assume \(H\): \(x \mod n = y \mod n\)
  \{ assume \(xy\): \(x \leq y\) \ from \(H\) have \(\text{th}: \ y \mod n = x \mod n\) \ by simp \ from \(nat-mod-eg-lemma[OF th xy]\) \ have \(?rhs\)
    apply clarify \ apply \(\text{(rule-tac x=q in exI)}\) \ by \(\text{(rule exI[where x=0], simp)}\)\}
  moreover
  \{ assume \(xy\): \(y \leq x\) \ from \(nat-mod-eg-lemma[OF H xy]\) \ have \(?rhs\)
    apply clarify \ apply \(\text{(rule-tac x=0 in exI)}\) \ by \(\text{(rule-tac x=q in exI, simp)}\)\}
  ultimately show \(?rhs\) \ using \(\text{linear[of x y]}\) \ by blast
next
  assume \(?rhs\) then obtain \(q1 q2\) where \(q12\): \(x + n \ast q1 = y + n \ast q2\) \ by blast
  hence \((x + n \ast q1) \mod n = (y + n \ast q2) \mod n\) \ by simp
  thus \(?\text{lhs by simp}\)
qed
This re-embedding of natural division on integers goes back to the time when numerals had been signed numerals. It should now be replaced by the algorithm developed in `semiring-numeral-div`.

**Lemma** `div-nat-numeral [simp]`:
\[
\text{numeral } v :: \text{nat} \div \text{numeral } v' = \text{nat} (\text{numeral } v \div \text{numeral } v')
\]
by `(simp add: nat-div-distrib)

**Lemma** `one-div-nat-numeral [simp]`:
\[
\text{Suc } 0 \div \text{numeral } v' = \text{nat} (1 \div \text{numeral } v')
\]
by `(subst nat-div-distrib, simp-all)

**Lemma** `mod-nat-numeral [simp]`:
\[
\text{numeral } v :: \text{nat} \mod \text{numeral } v' = \text{nat} (\text{numeral } v \mod \text{numeral } v')
\]
by `(simp add: nat-mod-distrib)

**Lemma** `one-mod-nat-numeral [simp]`:
\[
\text{Suc } 0 \mod \text{numeral } v' = \text{nat} (1 \mod \text{numeral } v')
\]
by `(subst nat-mod-distrib) simp-all

**Instance** `int :: semiring-numeral-div`

by `intro_classes`
\[
\text{auto intro: zmod-le-nonneg-dividend}
\]
\[
\text{simp add: zmult-div-cancel}
\]
\[
\text{pos-imp-zdiv-pos-iff mod-pos-pos-trivial mod-pos-pos-trivial}
\]
\[
\text{zmod-zmult2-eq zdiv-zmult2-eq}
\]

### 54.5.17 Tools setup

Nitpick

**Lemmas** `[nitpick-unfold] = dvd-eq-mod-eq-0 mod-div-equality' zmod-zdiv-equality'

### 54.5.18 Code generation

**Definition** `divmod-abs :: int ⇒ int ⇒ int × int`

where
\[
\text{divmod-abs } k \ l = (|k| \div |l|, |k| \mod |l|)
\]

**Lemma** `fst-divmod-abs [simp]`:
\[
\text{fst} (\text{divmod-abs } k \ l) = |k| \div |l|
\]
by `(simp add: divmod-abs-def)

**Lemma** `snd-divmod-abs [simp]`:
\[
\text{snd} (\text{divmod-abs } k \ l) = |k| \mod |l|
\]
by `(simp add: divmod-abs-def)

**Lemma** `divmod-abs-code [code]`:
\[
\text{divmod-abs } (\text{Int.Pos } k) (\text{Int.Pos } l) = \text{divmod } k \ l
\]
\[
\text{divmod-abs } (\text{Int.Neg } k) (\text{Int.Neg } l) = \text{divmod } k \ l
\]
\[
\text{divmod-abs } (\text{Int.Neg } k) (\text{Int.Pos } l) = \text{divmod } k \ l
\]
theory "Numeral-Simprocs"

by (simp-all add: prod-eq-iff)

lemma divmod-int-divmod-abs:
divmod-int k l = (if k = 0 then (0, 0) else if l = 0 then (0, k) else
  apsnd ((op *) (sgn l)) (if 0 < l ∧ 0 ≤ k ∨ l < 0 ∧ k < 0
  then divmod-abs k l
  else (let (r, s) = divmod-abs k l in
    if s = 0 then (−r, 0) else (−r − 1, |l| − s))))
proof −
  have aux: ∃q::int. −k = l * q ←→ k = l * −q by auto
  show ?thesis
    by (simp add: prod-eq-iff split-def Let-def)
      (auto simp add: aux not-less not-le zdiv-zminus1-eq-if
               zmod-zminus1-eq-if zdiv-zminus2-eq-if zmod-zminus2-eq-if)
qed

lemma divmod-int-code [code]:
divmod-int k l = (if k = 0 then (0, 0) else if l = 0 then (0, k) else
  apsnd ((op *) (sgn l)) (if sgn k = sgn l
  then divmod-abs k l
  else (let (r, s) = divmod-abs k l in
    if s = 0 then (−r, 0) else (−r − 1, |l| − s))))
proof −
  have k ≠ 0 ⟹ l ≠ 0 ⟹ 0 < l ∧ 0 ≤ k ∨ l < 0 ∧ k < 0 ⟹ sgn k = sgn l
    by (auto simp add: not-less sgn-if)
  then show ?thesis by (simp add: divmod-int-divmod-abs)
qed

hide-const (open) divmod-abs

code-identifier
code-module Divides → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

55 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions

theory Numeral-Simprocs
imports Divides
begin

ML-file ~/src/Provers/Arith/assoc-fold.ML
ML-file ~/src/Provers/Arith/cancel-numerals.ML
ML-file ~/src/Provers/Arith/combine-numerals.ML
lemmas semiring-norm =
  Let-def arith-simps diff-nat-numeral rel-simps
  if-False if-True
  add-0 add-Suc add-natural-left
  add-neg-natural-left mult-natural-left
  natural-One [symmetric] uminus-natural-One [symmetric] Suc-eq-plus1
  eq-natural-iff-iseq zero nat-eq-natural-Numerals1

declare split-div [of - - numeral k, arith-split] for k
declare split-mod [of - - numeral k, arith-split] for k

For combine-numerals

lemma left-add-mult-distrib: i*u + (j*u + k) = (i+j)*u + (k::nat)
  by (simp add: add-mult-distrib)

For cancel-numerals

lemma nat-diff-add-eq1:
  j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)
  by (simp split add: nat-diff-split add: add-mult-distrib)

lemma nat-diff-add-eq2:
  i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))
  by (simp split add: nat-diff-split add: add-mult-distrib)

lemma nat-eq-add-iff1:
  j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)
  by (auto split add: nat-diff-split simp add: add-mult-distrib)

lemma nat-eq-add-iff2:
  i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)
  by (auto split add: nat-diff-split simp add: add-mult-distrib)

lemma nat-less-add-iff1:
  j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)
  by (auto split add: nat-diff-split simp add: add-mult-distrib)

lemma nat-less-add-iff2:
  i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)
  by (auto split add: nat-diff-split simp add: add-mult-distrib)

lemma nat-le-add-iff1:
  j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)
  by (auto split add: nat-diff-split simp add: add-mult-distrib)

lemma nat-le-add-iff2:
  i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)
by (auto split add: nat-diff-split simp add: add-mult-distrib)

For cancel-numeral-factors

lemma nat-mult-le-cancel1: (0::nat) < k ==> (k * m <= k * n) = (m <= n)
  by auto

lemma nat-mult-less-cancel1: (0::nat) < k ==> (k * m < k * n) = (m < n)
  by auto

lemma nat-mult-eq-cancel1: (0::nat) < k ==> (k * m = k * n) = (m = n)
  by auto

lemma nat-mult-div-cancel1: (0::nat) < k ==> (k * m) div (k * n) = (m div n)
  by auto

lemma nat-mult-dvd-cancel-disj [simp]:
      (k * m) dvd (k * n) = (k = 0 | m dvd (n::nat))
  by (auto simp: dvd-eq-mod-eq-0 mod-mult-mult1)

lemma nat-mult-dvd-cancel1: 0 < k ==> (k * m) dvd (k * n::nat) = (m dvd n)
  by (auto)

For cancel-factor

lemmas nat-mult-le-cancel-disj = mult-le-cancel1

lemmas nat-mult-less-cancel-disj = mult-less-cancel1

lemma nat-mult-eq-cancel-disj:
  fixes k m n :: nat
  shows k * m = k * n ==> k = 0 ∨ m = n
  by auto

lemma nat-mult-div-cancel-disj [simp]:
  fixes k m n :: nat
  shows (k * m) div (k * n) = (if k = 0 then 0 else m div n)
  by (fact div-mult-mult1-if)

ML-file Tools/numeral-simprocs.ML

simproc-setup semiring-assoc-fold
  ((a::'a::comm-semiring-1-cancel) * b) =
  { fn phi => Numeral-Simprocsassoc-fold }

simproc-setup int-combine-numerals
  ((i::'a::comm-ring-1) + j | (i::'a::comm-ring-1) - j) =
  { fn phi => Numeral-Simprocs.combine-numerals }

simproc-setup field-combine-numerals
THEORY “Numeral-Simprocs”

\[
\begin{align*}
(i::a::\{\text{field-inverse-zero, ring-char-0}\}) + j &= \langle\langle fn \phi \mapsto \text{Numeral-Simprocs.field-combine-numerals} \rangle\rangle \\
(i::a::\{\text{field-inverse-zero, ring-char-0}\}) - j &= \langle\langle fn \phi \mapsto \text{Numeral-Simprocs.eq-cancel-numeral-factor} \rangle\rangle \\
\end{align*}
\]

\text{simproc-setup inteq-cancel-numerals}

\[
\begin{align*}
(l::a::\text{comm-ring-1}) + m &= n \\
(l::a::\text{comm-ring-1}) &= m + n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
(l::a::\text{comm-ring-1}) &= m - n \\
\end{align*}
\]

\text{simproc-setup intless-cancel-numerals}

\[
\begin{align*}
(l::a::\text{linordered-idom}) + m &= n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
\end{align*}
\]

\text{simproc-setup intle-cancel-numerals}

\[
\begin{align*}
(l::a::\text{linordered-idom}) + m &= n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
(l::a::\text{linordered-idom}) &= m + n \\
\end{align*}
\]

\text{simproc-setup ring-eq-cancel-numeral-factor}

\[
\begin{align*}
(l::a::\{\text{idom, ring-char-0}\}) * m &= n \\
(l::a::\{\text{idom, ring-char-0}\}) &= m * n \\
\end{align*}
\]

\text{simproc-setup ring-less-cancel-numeral-factor}

\[
\begin{align*}
(l::a::\text{linordered-idom}) * m &= n \\
(l::a::\text{linordered-idom}) * m &= n \\
\end{align*}
\]

\text{simproc-setup ring-le-cancel-numeral-factor}

\[
\begin{align*}
(l::a::\text{linordered-idom}) * m &= n \\
(l::a::\text{linordered-idom}) * m &= n \\
\end{align*}
\]
\[ ((\text{l}::\text{linordered-idom}) \leq m \cdot n) = \implies \text{fn phi} \Rightarrow \text{Numeral-Simprocs.le-cancel-numeral-factor} \]

**simproc-setup** int-div-cancel-numeral-factors
\[ ((\text{l}::\{\text{semiring-div, comm-ring-1, ring-char-0}\}) \cdot m) \div n \]
\[ (\text{l}::\{\text{semiring-div, comm-ring-1, ring-char-0}\}) \div (m \cdot n) = \implies \text{fn phi} \Rightarrow \text{Numeral-Simprocs.div-cancel-numeral-factor} \]

**simproc-setup** divide-cancel-numeral-factor
\[ ((\text{l}::\{\text{field-inverse-zero, ring-char-0}\}) \cdot m) / n \]
\[ ((\text{numeral v}::\{\text{field-inverse-zero, ring-char-0}\}) / (m \cdot n)) = \implies \text{fn phi} \Rightarrow \text{Numeral-Simprocs.divide-cancel-numeral-factor} \]

**simproc-setup** ring-eq-cancel-factor
\[ ((\text{l}::\text{idom}) \cdot m = n) \implies (\text{l}::\text{idom}) = m \cdot n \]
\[ (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.eq-cancel-factor}) \]

**simproc-setup** linordered-ring-le-cancel-factor
\[ ((\text{l}::\text{linordered-idom}) \cdot m \leq n) \]
\[ (\text{l}::\text{linordered-idom}) \leq m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.le-cancel-factor}) \]

**simproc-setup** linordered-ring-less-cancel-factor
\[ ((\text{l}::\text{linordered-idom}) \cdot m < n) \]
\[ (\text{l}::\text{linordered-idom}) < m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.less-cancel-factor}) \]

**simproc-setup** int-div-cancel-factor
\[ ((\text{l}::\text{semiring-div}) \cdot m) \div n \]
\[ (\text{l}::\text{semiring-div}) \div (m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.div-cancel-factor}) \]

**simproc-setup** int-mod-cancel-factor
\[ ((\text{l}::\text{semiring-div}) \cdot m) \mod n \]
\[ (\text{l}::\text{semiring-div}) \mod (m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.mod-cancel-factor}) \]

**simproc-setup** dvd-cancel-factor
\[ ((\text{l}::\text{idom}) \cdot m) \dvd n \]
\[ (\text{l}::\text{idom}) \dvd (m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.dvd-cancel-factor}) \]

**simproc-setup** divide-cancel-factor
\[ ((\text{l}::\text{field-inverse-zero}) \cdot m) / n \]
\[ (\text{l}::\text{field-inverse-zero}) / (m \cdot n) = \implies (\text{fn phi} \Rightarrow \text{Numeral-Simprocs.divide-cancel-factor}) \]
THEORY "Numeral-Simprocs"

ML-file Tools/nat-numeral-simprocs.ML

simproc-setup nat-combine-numerals
((i::nat) + j | Suc (i + j)) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.combine-numerals ⟩⟩

simproc-setup nateq-cancel-numerals
((l::nat) + m = n | (l::nat) = m + n |
 (l::nat) * m = n | (l::nat) = m * n |
 Suc m = n | m = Suc n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.eq-cancel-numerals ⟩⟩

simproc-setup natle-cancel-numerals
((l::nat) + m <= n | (l::nat) <= m + n |
 (l::nat) * m <= n | (l::nat) <= m * n |
 Suc m <= n | m <= Suc n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.le-cancel-numerals ⟩⟩

simproc-setup natdiff-cancel-numerals
(((l::nat) + m) - n | (l::nat) - (m + n) |
 (l::nat) * m - n | (l::nat) - m * n |
 Suc m - n | m - Suc n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.diff-cancel-numerals ⟩⟩

simproc-setup nat-eq-cancel-numeral-factor
((l::nat) * m = n | (l::nat) = m * n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.eq-cancel-numeral-factor ⟩⟩

simproc-setup nat-less-cancel-numeral-factor
((l::nat) * m < n | (l::nat) < m * n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.less-cancel-numeral-factor ⟩⟩

simproc-setup nat-le-cancel-numeral-factor
((l::nat) * m <= n | (l::nat) <= m * n) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.le-cancel-numeral-factor ⟩⟩

simproc-setup nat-div-cancel-numeral-factor
(((l::nat) + m) div n | (l::nat) div (m + n)) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.div-cancel-numeral-factor ⟩⟩

simproc-setup nat-dvd-cancel-numeral-factor
(((l::nat) + m) dvd n | (l::nat) dvd (m + n)) =
  ⟨⟨ fn phi => Nat-Numeral-Simprocs.dvd-cancel-numeral-factor ⟩⟩
simproc-setup nat-eq-cancel-factor
  \((l::nat) * m = n | (l::nat) = m * n) = 
  \langle\langle fn phi => Nat-Numeral-Simprocs.eq-cancel-factor \rangle\rangle

simproc-setup nat-less-cancel-factor
  \((l::nat) * m < n | (l::nat) < m * n) = 
  \langle\langle fn phi => Nat-Numeral-Simprocs.less-cancel-factor \rangle\rangle

simproc-setup nat-le-cancel-factor
  \((l::nat) * m <= n | (l::nat) <= m * n) = 
  \langle\langle fn phi => Nat-Numeral-Simprocs.le-cancel-factor \rangle\rangle

simproc-setup nat-div-cancel-factor
  \((l::nat) * m \text{ div } n | (l::nat) \text{ div } (m * n)) = 
  \langle\langle fn phi => Nat-Numeral-Simprocs.div-cancel-factor \rangle\rangle

simproc-setup nat-dvd-cancel-factor
  \((l::nat) * m \text{ dvd } n | (l::nat) \text{ dvd } (m * n)) = 
  \langle\langle fn phi => Nat-Numeral-Simprocs.dvd-cancel-factor \rangle\rangle

declaration \( \langle\langle K (Lin-Arith.add-simprocs \\
  @\{simproc semiring-assoc-fold\}, \\
  @\{simproc int-combine-numerals\}, \\
  @\{simproc inteq-cancel-numerals\}, \\
  @\{simproc intless-cancel-numerals\}, \\
  @\{simproc intle-cancel-numerals\}, \\
  @\{simproc field-combine-numerals\}\rangle \rangle \)

end

56  Semiring-Normalization: Semiring normalization

theory Semiring-Normalization
imports Numeral-Simprocs Nat-Transfer
begin

ML-file Tools/semiring-normalizer.ML

Prelude
class \textit{comm-semiring-1-cancel-crossproduct} = \textit{comm-semiring-1-cancel} + 
assumes \textit{crossproduct-eq}: \( w \ast y + x \ast z = w \ast z + x \ast y \) \( \leadsto w = x \land y = z \)
begin

lemma \textit{crossproduct-noteq}:
\( a \neq b \land c \neq d \) \( \leadsto a \ast c + b \ast d \neq a \ast d + b \ast c \)
by \((\text{simp add: crossproduct-eq})\)

lemma \textit{add-scale-eq-noteq}:
\( r \neq 0 \Rightarrow a = b \land c \neq d \Rightarrow a + r \ast c \neq b + r \ast d \)
proof \((\text{rule notI})\)
assume \( nz: r \neq 0 \) and \( \text{cnd}: a = b \land c \neq d \)
and \( \text{eq}: a + (r \ast c) = b + (r \ast d) \)
have \((0 \ast d) + (r \ast c) = (0 \ast c) + (r \ast d) \)
using \textit{add-imp-eq eq mult-zero-left} by \((\text{simp add: cnd})\)
then show False using \textit{crossproduct-eq} \([of 0 d] \) \( nz \) \text{cnd} by \text{simp}
qed

lemma \textit{add-0-iff}:
\( b = b + a \) \( \leadsto a = 0 \)
using \textit{add-imp-eq} \([of b a 0] \) by \text{auto}

end

subclass \((\text{in idom})\) \textit{comm-semiring-1-cancel-crossproduct}
proof
fix \( w \ x \ y \ z \)
show \( w \ast y + x \ast z = w \ast z + x \ast y \) \( \leadsto w = x \land y = z \)
proof
assume \( w \ast y + x \ast z = w \ast z + x \ast y \)
then have \( w \ast y + x \ast z - w \ast z - x \ast y = 0 \) by \((\text{simp add: algebra-simps})\)
then have \( w \ast (y - z) - x \ast (y - z) = 0 \) by \((\text{simp add: algebra-simps})\)
then have \( (y - z) \ast (w - x) = 0 \) by \((\text{simp add: algebra-simps})\)
then have \( y - z = 0 \lor w - x = 0 \) by \((\text{rule divisors-zero})\)
then show \( w = x \land y = z \) by \text{auto}
qed \((\text{auto simp add: ac-simps})\)
qed

instance \textit{nat} :: \textit{comm-semiring-1-cancel-crossproduct}
proof
fix \( w \ x \ y \ z :: \text{nat} \)
have \( \text{aux}: \bigwedge y \ z. y < z \Longrightarrow w \ast y + x \ast z = w \ast z + x \ast y \Longrightarrow w = x \)
proof 
fix \( y \ z :: \text{nat} \)
assume \( y < z \) then have \( \exists k. z = y + k \land k \neq 0 \) by \((\text{intro exI} \ [of - z - y])\)
auto
then obtain \( k \) where \( z = y + k \) and \( k \neq 0 \) by \text{blast}
assume \( w \ast y + x \ast z = w \ast z + x \ast y \)
then have \((w \ast y + x \ast y) + x \ast k = (w \ast y + x \ast y) + w \ast k \) by \((\text{simp add: algebra-simps})\)
qed 

add: \((z = y + k)\) algebra-simps
  then have \(x * k = w * k\) by simp
  then show \(w = x\) using \((k \neq 0)\) by simp
qed
show \(w * y + x * z = w * z + x * y \iff w = x \lor y = z\)
  by (auto simp add: neq-iff dest!: aux)
qed

Semiring normalization proper

setup Semiring-Normalizer.

setup context comm-semiring-1

begin

lemma normalizing-semiring-ops:
  shows TERM \((x + y)\) and TERM \((x * y)\) and TERM \((x ^ n)\)
  and TERM \(0\) and TERM \(1\).

lemma normalizing-semiring-rules:
  \((a * m) + (b * m) = (a + b) * m\)
  \((a * m) + m = (a + 1) * m\)
  \(m + (a * m) = (a + 1) * m\)
  \(m + m = (1 + 1) * m\)
  \(\theta + a = a\)
  \(a + \theta = a\)
  \(a * b = b * a\)
  \((a + b) * c = (a * c) + (b * c)\)
  \(\theta * a = 0\)
  \(a * \theta = 0\)
  \(1 * a = a\)
  \(a * 1 = a\)
  \((lx * ly) * (rx * ry) = (lx * rx) * (ly * ry)\)
  \((lx * ly) * (rx * ry) = lx * (ly * (rx * ry))\)
  \((lx * ly) * (rx * ry) = rx * ((lx * ly) * ry)\)
  \((lx * ly) * rx = (lx * rx) * ly\)
  \((lx * ly) * rx = lx * (ly * rx)\)
  \(lx * (rx * ry) = (lx * rx) * ry\)
  \(lx * (rx * ry) = rx * (lx * ry)\)
  \((a + b) + (c + d) = (a + c) + (b + d)\)
  \((a + b) + c = a + (b + c)\)
  \(a + (c + d) = c + (a + d)\)
  \((a + b) + c = (a + c) + b\)
  \(a + c = c + a\)
  \(a + (c + d) = (a + c) + d\)
  \((x ^ p) * (x ^ q) = x ^ (p + q)\)
  \(x * (x ^ q) = x ^ (Suc q)\)
  \((x ^ q) * x = x ^ (Suc q)\)
  \(x * x = x^2\)
  \((x * y) ^ q = (x ^ q) * (y ^ q)\)
THEORY “Semiring-Normalization”

\[(x \cdot p) \cdot q = x \cdot (p \cdot q)\]
\[x \cdot 0 = 1\]
\[x \cdot 1 = x\]
\[x \cdot (y + z) = (x \cdot y) + (x \cdot z)\]
\[x \cdot (\text{Suc } q) = x \cdot (x \cdot q)\]
\[x \cdot (2\cdot n) = (x \cdot n) \cdot (x \cdot n)\]
\[x \cdot (\text{Suc } (2\cdot n)) = x \cdot ((x \cdot n) \cdot (x \cdot n))\]

by (simp-all add: algebra-simps power-add power2-eq-square power-mult-distrib power-mult del: one-add-one)

lemmas normalizing-comm-semiring-1-axioms =
comm-semiring-1-axioms [normalizer
semiring ops: normalizing-semiring-ops
semiring rules: normalizing-semiring-rules]

declaration

\[\{\text{Semiring-Normalizer.semiring-funs @\{thm normalizing-comm-semiring-1-axioms\}\}}\]

end

context comm-ring-1
begin

lemma normalizing-ring-ops: shows TERM (x− y) and TERM (− x).

lemma normalizing-ring-rules:
\[- x = (− 1) \cdot x\]
\[x − y = x + (− y)\]
by simp-all

lemmas normalizing-comm-ring-1-axioms =
comm-ring-1-axioms [normalizer
semiring ops: normalizing-semiring-ops
semiring rules: normalizing-semiring-rules
ring ops: normalizing-ring-ops
ring rules: normalizing-ring-rules]

declaration

\[\{\text{Semiring-Normalizer.semiring-funs @\{thm normalizing-comm-ring-1-axioms\}\}}\]

end

context comm-semiring-1-cancel-crossproduct
begin

declare

normalizing-comm-semiring-1-axioms [normalizer del]
lemmas
normalizing-comm-semiring-1-cancel-crossproduct-axioms =
comm-semiring-1-cancel-crossproduct-axioms [normalizer
  semiring ops: normalizing-semiring-ops
  semiring rules: normalizing-semiring-rules
  idom rules: crossproduct-noteq add-scale-eq-noteq]

declaration
⟨⟨ Semiring-Normalizer.semiring-funs @{thm normalizing-comm-semiring-1-cancel-crossproduct-axioms} ⟩⟩

end

context idom
begin
 declare normalizing-comm-ring-1-axioms [normalizer del]

lemmas normalizing-idom-axioms = idom-axioms [normalizer
  semiring ops: normalizing-semiring-ops
  semiring rules: normalizing-semiring-rules
  ring ops: normalizing-ring-ops
  ring rules: normalizing-ring-rules
  idom rules: crossproduct-noteq add-scale-eq-noteq
  ideal rules: right-minus-eq add-0-iff]

declaration
⟨⟨ Semiring-Normalizer.semiring-funs @{thm normalizing-idom-axioms} ⟩⟩

end

context field
begin

lemma normalizing-field-ops:
  shows TERM (x / y) and TERM (inverse x).

lemmas normalizing-field-rules = divide-inverse inverse-eq-divide

lemmas normalizing-field-axioms =
  field-axioms [normalizer
    semiring ops: normalizing-semiring-ops
    semiring rules: normalizing-semiring-rules
    ring ops: normalizing-ring-ops
    ring rules: normalizing-ring-rules
    field ops: normalizing-field-ops
    field rules: normalizing-field-rules
    idom rules: crossproduct-noteq add-scale-eq-noteq
Ideal rules: right-minus-eq add-0-iff]

declaration

⟨⟨ Semiring-Normalizer.field-funs @{thm normalizing-field-axioms} ⟩⟩

deny-fact (open) normalizing-comm-semiring-1-axioms
normalizing-comm-semiring-1-cancel-crossproduct-axioms normalizing-semiring-ops
normalizing-semiring-rules

deny-fact (open) normalizing-comm-ring-1-axioms
normalizing-idom-axioms normalizing-ring-ops normalizing-ring-rules

deny-fact (open) normalizing-field-axioms normalizing-field-ops normalizing-field-rules

code-identifier
code-module Semiring-Normalization → (SML) Arith and (OCaml) Arith and
(Haskell) Arith

de-identifier

code-module Groebner-Basis imports Semiring-Normalization
keywords try0 :: diag
begin

57 Groebner-Basis: Groebner bases

theory Groebner-Basis
imports Semiring-Normalization
keywords try0 :: diag
begin

57.1 Groebner Bases

lemmas bool-simps = simp-thms(1-34) — FIXME move to HOL

lemma nnf-simps: — FIXME shadows fact binding in HOL
(¬(P ∧ Q)) = (¬P ∨ ¬Q) (¬P ∧ ¬Q)
(P → Q) = (¬P ∨ Q)
(P = Q) = ((P ∧ Q) ∨ (¬P ∧ ¬ Q)) (¬ (¬(P)) = P
by blast

lemma dnf:
(P & (Q | R)) = ((P&Q) | (P&R))
((Q | R) & P) = ((Q&P) | (R&P))
(P ∧ Q) = (Q ∧ P)
(P ∨ Q) = (Q ∨ P)
by blast

lemmas weak-dnf-simps = dnf bool-simps

lemma PFalse:
\[ P \equiv \text{False} \implies \neg P \]
\[ \neg P \implies (P \equiv \text{False}) \]

by auto

**ML**

```ML
structure Algebra-Simplification = Named-Thms {
  val name = @{binding algebra}
  val description = pre-simplification rules for algebraic methods
}
```

**setup** Algebra-Simplification.setup

**ML-file** Tools/groebner.ML

**method-setup** algebra = 

```
let
  fun keyword k = Scan.lift (Args.$$ k -- Args.colon) >> K ()
  val addN = add
  val delN = del
  val any-keyword = keyword addN || keyword delN
  val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
  in
  Scan.optional (keyword addN |-- thms) []
  Scan.optional (keyword delN |-- thms) [] >>
  (fn (add-ths, del-ths) => fn ctxt =>
    SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))
  end
```

**solve polynomial equations over (semi)rings and ideal membership problems using Groebner bases**

**declare** dvd-def [algebra]
**declare** dvd-eq-mod-eq-0 [symmetric, algebra]
**declare** mod-div-trivial [algebra]
**declare** mod-mod-trivial [algebra]
**declare** div-by-0 [algebra]
**declare** mod-by-0 [algebra]
**declare** zmod-zdiv-equality [symmetric, algebra]
**declare** div-mod-equality2 [symmetric, algebra]
**declare** div-minus-minus [algebra]
**declare** mod-minus-minus [algebra]
**declare** div-minus-right [algebra]
**declare** mod-minus-right [algebra]
**declare** div-0 [algebra]
**declare** mod-0 [algebra]
**declare** mod-by-1 [algebra]
**declare** div-by-1 [algebra]
**declare** mod-minus-l-right [algebra]
end

58 Lattices-Big: Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

theory Lattices-Big
imports Finite-Set Option
begin

58.1 Generic lattice operations over a set

no-notation times (infixl * 70)
no-notation Groups.one (1)

58.1.1 Without neutral element

locale semilattice-set = semilattice
begin

interpretation comp-fun-idem f
  by default (simp-all add: fun-eq-iff left-commute)

definition F :: 'a set ⇒ 'a
where
eq-fold': F A = the (Finite-Set.fold (λx y. Some (case y of None ⇒ x | Some z ⇒ f x z)) None A)

lemma eq-fold:
  assumes finite A
  shows F (insert x A) = Finite-Set.fold f x A
proof (rule sym)
  let ?f = λx y. Some (case y of None ⇒ x | Some z ⇒ f x z)
  interpret comp-fun-idem ?f
  by default (simp-all add: fun-eq-iff commute left-commute split: option.split)
  from assms show Finite-Set.fold f x A = F (insert x A)
proof induct
  case empty then show ?case by (simp add: eq-fold')
  next
    case (insert y B) then show ?case by (simp add: insert-commute [of x] eq-fold')
qed

lemma singleton [simp]:
  \( F \{ x \} = x \)
  by (simp add: eq-fold)

lemma insert-not-elem:
  assumes finite A and \( x \notin A \) and \( A \neq \{} \)
  shows \( F (insert x A) = x \ast F A \)
proof (\!
  \!
  \!
  \!
  \!
  from \langle A \neq \{\} \rangle \!
  \!
  \!
  \!
  \!
  \!
  obtain b where b \in A \!
  \!
  \!
  \!
  \!
  \!
  by blast
  \!
  \!
  \!
  \!
  \!
  then obtain B where *: \langle A = insert b B \rangle \!
  \!
  \!
  \!
  \!
  \!
  and \( x \notin B \) by auto
  \!
  \!
  \!
  \!
  \!
  then have \( F (insert b (insert x B)) = x \ast F (insert b B) \)
  \!
  \!
  \!
  \!
  \!
  by (simp add: eq-fold)
  \!
  \!
  \!
  \!
  \!
  then show ?thesis by (simp add: * insert-commute)
qed

lemma in-idem:
  assumes finite A and \( x \in A \)
  shows \( x \ast F A = F A \)
proof (\!
  \!
  \!
  \!
  \!
  from assms have \( A \neq \{\} \) by auto
  \!
  \!
  \!
  \!
  \!
  with \langle finite A \rangle \!
  \!
  \!
  \!
  \!
  \!
  show ?thesis using \langle x \in A \rangle \!
  \!
  \!
  \!
  \!
  \!
  by (induct A rule: finite-ne-induct) (auto simp add: ac-simps insert-not-elem)\)
qed

lemma insert [simp]:
  assumes finite A and \( A \neq \{} \)
  shows \( F (insert x A) = x \ast F A \)
using assms by (cases \( x \in A \)) (simp-all add: insert-absorb in-idem insert-not-elem)

lemma union:
  assumes finite A A \( \neq \{\} \) and finite B B \( \neq \{\} \)
  shows \( F (A \cup B) = F A \ast F B \)
using assms by (induct A rule: finite-ne-induct) (simp-all add: ac-simps)

lemma remove:
  assumes finite A and \( x \in A \)
  shows \( F A = (if A - \{ x \} = \{\} \ then x else x \ast F (A - \{ x \})) \)
proof (\!
  \!
  \!
  \!
  \!
  \!
  from assms obtain B where \langle A = insert x B \rangle \!
  \!
  \!
  \!
  \!
  \!
  and \( x \notin B \) by (blast dest:\!
  \!
  \!
  \!
  \!
  \!
  mk-disjoint-insert)
  \!
  \!
  \!
  \!
  \!
  with assms show ?thesis by simp
qed

lemma insert-remove:
assumes finite A
shows \( F(\text{insert } x A) = (\text{if } A - \{x\} = \{} \text{ then } x \text{ else } x \ast F(A - \{x\})\) using assms by (cases \( x \in A \)) (simp-all add: insert-absorb remove)

lemma subset:
assumes finite A B \( \neq \{} \) and B \( \subseteq \) A
shows \( F(B) \ast F(A) = F(A) \)
proof –
from assms have A \( \neq \{} \) and finite B by (auto dest: finite-subset)
with assms show thesis by (simp add: union [symmetric] Un-absorb1) qed

lemma closed:
assumes finite A A \( \neq \{} \) and elem: \( \forall x y. x \ast y \in \{x, y\} \)
shows \( F(A) \in A \)
proof (induct rule: finite-ne-induct)
case singleton then show ?case by simp
next
case insert with elem show ?case by force qed

lemma hom-commute:
assumes hom: \( \forall x y. h(x \ast y) = h(x) \ast h(y) \)
and N: finite N N \( \neq \{} \)
shows \( h(F(N)) = F(h \cdot N) \)
using N proof (induct rule: finite-ne-induct)
case singleton thus ?case by simp
next
case (insert n N)
then have \( h(F(\text{insert } n N)) = h(n \ast F(N)) \) by simp
also have \( \ldots = h(n \ast h(F(N)) \) by (rule hom)
also have \( h(F(N)) = F(h \cdot N) \) by (rule insert)
also have \( h(n \ast \ldots = F(\text{insert } h(n)(h \cdot N)) \)
using insert by simp
also have \( \text{insert } h(n)(h \cdot N) = h \cdot \text{insert } n N \) by simp
finally show ?case . qed

lemma infinite: \( \neg \) finite A \( \Rightarrow \) F A = the None
unfolding eq-fold' by (cases finite (UNIV::'a set)) (auto intro: finite-subset fold-infinite)

end

locale semilattice-order-set = binary?: semilattice-order + semilattice-set
begin
lemma bounded-iff:
assumes finite A and A \( \neq \{} \)
shows $x \leq F A \iff (\forall a \in A. x \leq a)$

using assms by (induct rule: finite-ne-induct) (simp-all add: bounded-iff)

lemma boundedI:
assumes finite A
assumes $A \neq \{\}$
assumes $\bigwedge a. a \in A \implies x \leq a$
shows $x \leq F A$
using assms by (simp add: bounded-iff)

lemma boundedE:
assumes finite A and $A \neq \{\}$ and $x \leq F A$
obtains $\bigwedge a. a \in A \implies x \leq a$
using assms by (simp add: bounded-iff)

lemma coboundedI:
assumes finite A
and $a \in A$
shows $F A \leq a$
proof -
from assms have $A \neq \{\}$ by auto
from (finite A) (A \neq \{\}) (a \in A) show ?thesis
proof (induct rule: finite-ne-induct)
  case singleton thus ?case by (simp add: refl)
next
  case (insert x B)
  from insert have $a = x \lor a \in B$ by simp
  then show ?case using insert by (auto intro: coboundedI2)
qed
qed

lemma antimono:
assumes $A \subseteq B$ and $A \neq \{\}$ and finite B
shows $F B \leq F A$
proof (cases $A = B$)
  case True then show ?thesis by (simp add: refl)
next
  case False
  have $B = A \cup (B - A)$ using (A \subseteq B) by blast
  then have $F B = F (A \cup (B - A))$ by simp
  also have $\ldots = F A \ast F (B - A)$ using False assms by (subst union) (auto intro: finite-subset)
  also have $\ldots \leq F A$ by simp
  finally show ?thesis .
qed
end
58.1.2 With neutral element

locale semilattice-neutr-set = semilattice-neutr
begin

interpretation comp-fun-idem f
by default (simp-all add: fun-eq-iff left-commute)

definition F :: 'a set ⇒ 'a
where
eq-fold: F A = Finite-Set.fold f 1 A

lemma infinite [simp]:
¬ finite A ⇒ F A = 1
by (simp add: eq-fold)

lemma empty [simp]:
F {} = 1
by (simp add: eq-fold)

lemma insert [simp]:
assumes finite A
shows F (insert x A) = x * F A
using assms by (simp add: eq-fold)

lemma in-idem:
assumes finite A and x ∈ A
shows x * F A = F A
proof –
from assms have A ≠ {} by auto
with ⟨finite A⟩ show ?thesis using ⟨x ∈ A⟩
  by (induct A rule: finite-ne-induct) (auto simp add: ac-simps)
qed

lemma union:
assumes finite A and finite B
shows F (A ∪ B) = F A * F B
using assms by (induct A) (simp-all add: ac-simps)

lemma remove:
assumes finite A and x ∈ A
shows F A = x * F (A − {x})
proof –
from assms obtain B where A = insert x B and x ∉ B by (blast dest: mk-disjoint-insert)
with assms show ?thesis by simp
qed

lemma insert-remove:
assumes finite A
shows \( F (\text{insert } x A) = x \ast F (A - \{x\}) \)
using assms by (cases \( x \in A \)) (simp-all add: insert-absorb remove)

lemma subset:
assumes finite A and \( B \subseteq A \)
sows \( F B \ast F A = F A \)
proof –
from assms have finite \( B \) by (auto dest: finite-subset)
with assms show ?thesis by (simp add: union [symmetric] Un-absorb1)
qed

lemma closed:
assumes finite A \( A \neq \{\} \) and elem: \( \forall x \ y. x \ast y \in \{x, y\} \)
shows \( F A \in A \)
proof
from assms obtain finite \( A \) \( \langle A \neq \{\} \rangle \)
proof (induct rule: finite-ne-induct)
case singleton then show ?case by simp
next
case insert with elem show ?case by force
qed

end

locale semilattice-order-neutr-set = binary?: semilattice-neutr-order + semilattice-neutr-set
begin
lemma bounded-iff:
assumes finite A
shows \( x \preceq F A \iff (\forall a \in A. x \preceq a) \)
using assms by (induct A) (simp-all add: bounded-iff)

lemma boundedI:
assumes finite A
assumes \( \bigwedge a. a \in A \Longrightarrow x \preceq a \)
shows \( x \preceq F A \)
using assms by (simp add: bounded-iff)

lemma boundedE:
assumes finite A and \( x \preceq F A \)
obtains \( \bigwedge a. a \in A \Longrightarrow x \preceq a \)
using assms by (simp add: bounded-iff)

lemma coboundedI:
assumes finite A
and \( a \in A \)
sows \( F A \succeq a \)
proof –
from assms have \( A \neq \{\} \) by auto
from \( \langle \text{finite } A; \ A \neq \{\} \rangle \langle a \in A \rangle \) show ?thesis
proof (induct rule: finite-ne-induct)
THEORY "Lattices-Big"

case singleton thus ?case by (simp add: refl)
next
case (insert x B)
from insert have a = x ∨ a ∈ B by simp
then show ?case using insert by (auto intro: coboundedI2)
qed

lemma antimono:
  assumes A ⊆ B and finite B
  shows F B ≤ F A
proof (cases A = B)
case True then show ?thesis by (simp add: refl)
next
case False
have B: B = A ∪ (B − A) using ⟨A ⊆ B⟩ by blast
then have F B = F (A ∪ (B − A)) by simp
also have ... = F A * F (B − A) using False assms by (subst union) (auto intro: finite-subset)
also have ... ≤ F A by simp
finally show ?thesis .
qed

end

notation times (infixl * 70)
notation Groups.one (1)

58.2 Lattice operations on finite sets
context semilattice-inf
begin

definition Inf-fin :: 'a set ⇒ 'a (Π fin− [900] 900)
where
  Inf-fin = semilattice-set.F inf

sublocale Inf-fin!: semilattice-order-set inf less-eq less
where
  semilattice-set.F inf = Inf-fin
proof –
  show semilattice-order-set inf less-eq less ..
  then interpret Inf-fin!: semilattice-order-set inf less-eq less .
  from Inf-fin-def show semilattice-set.F inf = Inf-fin by rule
qed

end

context semilattice-sup
THEORY “Lattices-Big”

begin

definition Sup-fin :: 'a set ⇒ 'a (⋃_{f in [900]} 900)
where
  Sup-fin = semilattice-set.F sup

sublocale Sup-fin!: semilattice-order-set sup greater-eq greater
where
  semilattice-set.F sup = Sup-fin
proof –
  show semilattice-order-set sup greater-eq greater ..
  then interpret Sup-fin!: semilattice-order-set sup greater-eq greater .
  from Sup-fin-def show semilattice-set.F sup = Sup-fin by rule
qed

end

58.3 Infimum and Supremum over non-empty sets

context lattice
begin

lemma Inf-fin-le-Sup-fin [simp]:
  assumes finite A and A ≠ {}
  shows \( \bigcap_{f in A} \leq \bigcup_{f in A} \)
proof –
  from \( A \neq {} \) obtain a where a ∈ A by blast
  with (finite A) have \( \bigcap_{f in A} \leq a \) by (rule Inf-fin.coboundedI)
  moreover from (finite A) (a ∈ A) have \( a \leq \bigcup_{f in A} \) by (rule Sup-fin.coboundedI)
  ultimately show \( \text{thesis} \) by (rule order-trans)
qed

lemma sup-Inf-absorb [simp]:
  finite A ⇒ a ∈ A ⇒ \( \bigcap_{f in A} \cup a = a \)
  by (rule sup-absorb2) (rule Inf-fin.coboundedI)

lemma inf-Sup-absorb [simp]:
  finite A ⇒ a ∈ A ⇒ a ∩ \( \bigcup_{f in A} \) = a
  by (rule inf-absorb1) (rule Sup-fin.coboundedI)

end

context distrib-lattice
begin

lemma sup-Inf1-distrib:
  assumes finite A and A ≠ {}
  shows sup x (\( \bigcap_{f in A} \)) = \( \bigcap_{f in \{sup x a | a \in A\}} \)

end
THEORY “Lattices-Big”

using assms by (simp add: image-def Inf-fin.hom-commute [where h=sup x, OF sup-inf-distrib1])
  (rule arg-cong [where f=Inf-fin], blast)

lemma sup-Inf2-distrib:
  assumes A: finite A A ≠ {} and B: finite B B ≠ {}
  shows sup (⋃ f∈finA) (⋂ f∈finB) = ⋂ f∈fin{sup a b | a ∈ A ∧ b ∈ B}
  using assms

proof
  have {sup a b | a ∈ A ∧ b ∈ B} = (UN a:A. UN b:B. {sup a b})
    by blast
  thus ?thesis by (simp add: insert I B(1))

qed

have ne: {sup a b | a ∈ A ∧ b ∈ B} ≠ {}
  using insert B by blast

have sup (⋃ f∈finA) (⋂ f∈finB) = sup (inf x (⋂ f∈finA)) (⋂ f∈finB)
  using insert by simp
also have . . . = inf (sup x (⋂ f∈finB)) (sup (⋃ f∈finA) (⋂ f∈finB)) by(rule sup-inf-distrib2)
also have . . . = inf (⋂ f∈fin{sup x b | b ∈ B}) (⋂ f∈fin{sup a b | a ∈ A ∧ b ∈ B})
  using insert by(simp add:sup-Inf1-distrib[OF B])
also have . . . = ⋂ f∈fin ({sup x b | b ∈ B} ∪ {sup a b | a ∈ A ∧ b ∈ B})
  (is . . = ⋂ f∈fin ?M)
  using B insert
    by (simp add: Inf-fin.union [OF finB - finAB ne])
also have ?M = {sup a b | a ∈ insert x A ∧ b ∈ B}
  by blast
finally show ?case.

qed

lemma inf-Sup1-distrib:
  assumes A: finite A A ≠ {} and A ≠ {}
  shows inf x (⋃ f∈finA) = ⋃ f∈fin{inf x a | a ∈ A}
  using assms by (simp add: image-def Sup-fin.hom-commute [where h=inf x, OF inf-sup-distrib1])
  (rule arg-cong [where f=Sup-fin], blast)

lemma inf-Sup2-distrib:
  assumes A: finite A A ≠ {} and B: finite B B ≠ {}
  shows inf (⋃ f∈finA) (⋃ f∈finB) = ⋃ f∈fin{inf a b | a ∈ A ∧ b ∈ B}
  using assms

proof
  (induct rule: finite-ne-induct)
  case singleton thus ?case
    by(simp add: inf-Sup1-distrib [OF B])
next
  case (insert x A)
  have finB: finite {inf x b | b ∈ B} 
    by (rule finite-surj[where f = %b. inf x b, OF B(1)], auto)
  have finAB: finite {inf a b | a ∈ A ∧ b ∈ B}
    proof -
      have {inf a b | a ∈ A ∧ b ∈ B} = (UN a:A. UN b:B. {inf a b})
        by blast
      thus ?thesis by (simp add: insert(1) B(1))
    qed
  have ne: {inf a b | a ∈ A ∧ b ∈ B} ≠ {} using insert B by blast
  have inf (∩ f_in(insert A)) (∪ f_inB) = inf (sup x (∪ f_inA)) (∪ f_inB)
    using insert by simp
  also have ... = sup (inf x (∪ f_inB)) (inf (∪ f_inA) (∪ f_inB)) by (rule inf-sup-distrib2)
  also have ... = sup (∪ f_in{inf x b | b ∈ B}) (∪ f_in{inf a b | a ∈ A ∧ b ∈ B})
    using insert by (simp add: Sup-fin.union[OF finB - finAB ne])
  also have ?M = {inf a b | a ∈ insert x A ∧ b ∈ B}
    by blast
  finally show ?case .
qed

context complete-lattice
begin

lemma Inf-fin-Inf:
  assumes finite A and A ≠ {} 
  shows (∩ f_inA) = ∩ A
proof -
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis 
    by (simp add: Inf-fin.eq-fold inf-inf-commute[of b])
qed

lemma Sup-fin-Sup:
  assumes finite A and A ≠ {} 
  shows (∪ f_inA) = ∪ A
proof -
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis 
    by (simp add: Sup-fin.eq-fold sup-sup-commute[of b])
qed
58.4 Minimum and Maximum over non-empty sets

context linorder
begin

definition Min :: 'a set ⇒ 'a
where
  Min = semilattice-set.F min

definition Max :: 'a set ⇒ 'a
where
  Max = semilattice-set.F max

sublocale Min!: semilattice-order-set min less-eq less
+ Max!: semilattice-order-set max greater-eq greater
where
  semilattice-set.F min = Min
and semilattice-set.F max = Max
proof –
  show semilattice-order-set min less-eq less by default (auto simp add: min-def)
  then interpret Min!: semilattice-order-set min less-eq less .
  show semilattice-order-set max greater-eq greater by default (auto simp add: max-def)
  then interpret Max!: semilattice-order-set max greater-eq greater .
from Min-def show semilattice-set.F min = Min by rule
from Max-def show semilattice-set.F max = Max by rule
qed

end

An aside: Min/Max on linear orders as special case of Inf-fin/Sup-fin

lemma Inf-fin-Min:
  Inf-fin = (Min :: 'a::{semilattice-inf, linorder} set ⇒ 'a)
by (simp add: Inf-fin-def Min-def inf-min)

lemma Sup-fin-Max:
  Sup-fin = (Max :: 'a::{semilattice-sup, linorder} set ⇒ 'a)
by (simp add: Sup-fin-def Max-def sup-max)

context linorder
begin

lemma dual-min:
  ord.min greater-eq = max
by (auto simp add: ord.min-def max-def fun-eq-iff)

lemma dual-max:
ord.max greater-eq = min
by (auto simp add: ord.max-def min-def fun-eq-iff)

lemma dual-Min:
linorder.Min greater-eq = Max
proof –
interpret dual!: linorder greater-eq greater by (fact dual-linorder)
show ?thesis by (simp add: dual.Min-def dual-min Max-def)
qed

lemma dual-Max:
linorder.Max greater-eq = Min
proof –
interpret dual!: linorder greater-eq greater by (fact dual-linorder)
show ?thesis by (simp add: dual.Max-def dual-max Min-def)
qed

lemmas Min-singleton = Min.singleton
lemmas Max-singleton = Max.singleton
lemmas Min-insert = Min.insert
lemmas Max-insert = Max.insert
lemmas Min-Un = Min.union
lemmas Max-Un = Max.union
lemmas hom-Min-commute = Min.hom-commute
lemmas hom-Max-commute = Max.hom-commute

lemma Min-in [simp]:
assumes finite A and A ≠ {}
shows Min A ∈ A
using assms by (auto simp add: min-def Min.closed)

lemma Max-in [simp]:
assumes finite A and A ≠ {}
shows Max A ∈ A
using assms by (auto simp add: max-def Max.closed)

lemma Min-le [simp]:
assumes finite A and x ∈ A
shows Min A ≤ x
using assms by (fact Min.coboundedI)

lemma Max-ge [simp]:
assumes finite A and x ∈ A
shows x ≤ Max A
using assms by (fact Max.coboundedI)

lemma Min-eqI:
assumes finite A
assumes ∀y. y ∈ A → y ≥ x
proof (rule antisym)
  from \(x \in A\) have \(A \neq \{\}\) by auto
with assms show \(Min A \geq x\) by simp
next
  from assms show \(x \geq Min A\) by simp
qed

lemma Max-eqI:
  assumes finite A
  assumes \(\forall y. y \in A \implies y \leq x\)
  and \(x \in A\)
  shows \(Max A = x\)
proof (rule antisym)
  from \(x \in A\) have \(A \neq \{\}\) by auto
with assms show \(Max A \leq x\) by simp
next
  from assms show \(x \leq Max A\) by simp
qed

context
  fixes A :: 'a set
  assumes fin-nonempty: finite A A \neq \{\}
begin

lemma Min-le-iff [simp]:
  \(x \leq Min A \iff (\forall a \in A. x \leq a)\)
using fin-nonempty by (fact Min.bounded iff)

lemma Max-le-iff [simp]:
  \(Max A \leq x \iff (\forall a \in A. a \leq x)\)
using fin-nonempty by (fact Max.bounded iff)

lemma Min-gr-iff [simp]:
  \(x < Min A \iff (\forall a \in A. x < a)\)
using fin-nonempty by (induct rule: finite-ne-induct) simp-all

lemma Max-less-iff [simp]:
  \(Max A < x \iff (\forall a \in A. a < x)\)
using fin-nonempty by (induct rule: finite-ne-induct) simp-all

lemma Min-le-iff:
  \(Min A \leq x \iff (\exists a \in A. a \leq x)\)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: min-le-iff-disj)

lemma Max-le-iff:
  \(x \leq Max A \iff (\exists a \in A. x \leq a)\)
using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: le-max-iff-disj)
lemma \textit{Min-less-iff}:
\begin{align*}
\text{Min } A < x \iff (\exists a \in A. a < x)
\end{align*}
\textit{using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: min-less-iff-disj)}

lemma \textit{Max-gr-iff}:
\begin{align*}
x < \text{Max } A \iff (\exists a \in A. x < a)
\end{align*}
\textit{using fin-nonempty by (induct rule: finite-ne-induct) (simp-all add: less-max-iff-disj)}

\textit{end}

lemma \textit{Min-antimono}:
\begin{align*}
\text{assumes } M \subseteq N \text{ and } M \neq \{\} \text{ and finite } N & \\
\text{shows } \text{Min } N \leq \text{Min } M & \\
\text{using assms by (fact Min.antimono)}
\end{align*}

lemma \textit{Max-mono}:
\begin{align*}
\text{assumes } M \subseteq N \text{ and } M \neq \{\} \text{ and finite } N & \\
\text{shows } \text{Max } M \leq \text{Max } N & \\
\text{using assms by (fact Max.antimono)}
\end{align*}

\textit{end}

context \textit{linorder}
\begin{align*}
\text{lemma mono-Min-commute:} & \\
\text{assumes mono } f & \\
\text{assumes finite } A \text{ and } A \neq \{\} & \\
\text{shows } f (\text{Min } A) = \text{Min } (f \cdot A) & \\
\text{proof } (\text{rule linorder-class.Min-eqI \texttt{[symmetric]}}) & \\
\text{from } \text{finite } A \text{ show finite } (f \cdot A) \text{ by simp} & \\
\text{from assms show } f (\text{Min } A) \in f \cdot A \text{ by simp} & \\
\text{fix } x & \\
\text{assume } x \in f \cdot A & \\
\text{then obtain } y \text{ where } y \in A \text{ and } x = f y .. & \\
\text{with assms have } \text{Min } A \leq y \text{ by auto} & \\
\text{with } (\text{mono } f) \text{ have } f (\text{Min } A) \leq f y \text{ by (rule monoE)} & \\
\text{with } (x = f y) \text{ show } f (\text{Min } A) \leq x \text{ by simp} & \\
\text{qed}
\end{align*}

\textit{lemma mono-Max-commute:} & \\
\text{assumes mono } f & \\
\text{assumes finite } A \text{ and } A \neq \{\} & \\
\text{shows } f (\text{Max } A) = \text{Max } (f \cdot A) & \\
\text{proof } (\text{rule linorder-class.Max-eqI \texttt{[symmetric]}}) & \\
\text{from } \text{finite } A \text{ show finite } (f \cdot A) \text{ by simp} & \\
\text{from assms show } f (\text{Max } A) \in f \cdot A \text{ by simp} & \\
\text{fix } x &
assume $x \in f \cdot A$
then obtain $y$ where $y \in A$ and $x = f y$ ..
with assort have $y \leq \operatorname{Max} A$ by auto
with (\text{mono} f) have $f y \leq f (\operatorname{Max} A)$ by (rule monoE)
with $(x = f y)$ show $x \leq f (\operatorname{Max} A)$ by simp
qed

\textbf{lemma finite-linorder-max-induct [consumes 1, case-names empty insert]}:
assumes fin: finite $A$
and empty: $P \{\}$
and insert: $\forall b A. \text{finite} A \Rightarrow \forall a \in A. a < b \Rightarrow P A \Rightarrow P (\text{insert} b A)$
shows $P A$
using fin empty insert
proof (induct rule: finite-psubset-induct)
case (psubset $A$
have IH: $\forall B. [B < A; P \{\}; (\forall A b. [\text{finite} A; \forall a \in A. a < b; P A] \Rightarrow P (\text{insert} b A))] \Rightarrow P B$ by fact
have fin: finite $A$ by fact
have empty: $P \{\}$ by fact
have step: $\forall b A. [\text{finite} A; \forall a \in A. a < b; P A] \Rightarrow P (\text{insert} b A)$ by fact
show $P A$

proof (cases $A = \{\}$
  assume $A = \{\}$
  then show $P A$ using $P \{\}$ by simp
next
  let $?B = A - \{\text{Max} A\}$
  let $?A = \text{insert} (\text{Max} A) ?B$
  have finite $?B$ using finite $A$ by simp
  assume $A \neq \{\}$
  with (finite $A$) have $\text{Max} A : A$ by auto
  then have $A : ?A = A$ using insert-Diff-single insert-absorb by auto
  then have $P ?B$ using $P \{\}$ step IH [of $?B$] by blast
  moreover
  have $\forall a \in ?B. a < \text{Max} A$ using Max-ge [OF (finite $A$)] by fastforce
  ultimately show $P A$ using $A$ insert-Diff-single step [OF (finite $?B$)] by fastforce
qed
qed

\textbf{lemma finite-linorder-min-induct [consumes 1, case-names empty insert]}:
$[\text{finite} A; P \{\}; \forall b A. [\text{finite} A; \forall a \in A. b < a; P A] \Rightarrow P (\text{insert} b A)] \Rightarrow P A$
by (rule linorder.finite-linorder-max-induct [OF dual-linorder])

\textbf{lemma Least-Min}:
assumes finite $\{a. P a\}$ and $\exists a. P a$
shows $(\text{LEAST} a. P a) = \text{Min} \{a. P a\}$
proof –
{ fix $A :: \text{a set}$
assume $A$: finite $A$ $A \neq \{\}$

have $(\text{LEAST } a. \ a \in A) = \text{Min } A$

using $A$ proof (induct $A$ rule: finite-ne-induct)

case singleton show ?case by (rule Least-equality) simp-all

next

case (insert $a A$)

have $(\text{LEAST } b. \ b = a \lor b \in A) = \text{min } (\text{LEAST } a. \ a \in A)$

by (auto intro!: Least-equality simp add: min-def not-le Min-le-iff insert.hyps dest!: less-imp-le)

with insert show ?case by simp

qed

} from this [of $\{a.\ P a\}$] assms show ?thesis by simp

qed

end

context linordered-ab-semigroup-add

begin

lemma add-Min-commute:

fixes $k$

assumes finite $N$ and $N \neq \{\}$

shows $k + \text{Min } N = \text{Min } \{k + m \mid m.\ m \in N\}$

proof –

have $\forall x \ y. \ k + \text{min } x \ y = \text{min } (k + x) (k + y)$

by (simp add: min-def not-le)

(blast intro: antisym less-imp-le add-left-mono)

with assms show ?thesis

using hom-Min-commute [of plus $k N$]

by simp (blast intro: arg-cong [where $f = \text{Min}$])

qed

lemma add-Max-commute:

fixes $k$

assumes finite $N$ and $N \neq \{\}$

shows $k + \text{Max } N = \text{Max } \{k + m \mid m.\ m \in N\}$

proof –

have $\forall x \ y. \ k + \text{max } x \ y = \text{max } (k + x) (k + y)$

by (simp add: max-def not-le)

(blast intro: antisym less-imp-le add-left-mono)

with assms show ?thesis

using hom-Max-commute [of plus $k N$]

by simp (blast intro: arg-cong [where $f = \text{Max}$])

qed

end

context linordered-ab-group-add

begin
lemma minus-Max-eq-Min [simp]:
finite S \implies S \neq \{\} \implies -\operatorname{Max} S = \operatorname{Min} (\operatorname{uminus} ' S)
by (induct S rule: finite-ne-induct) (simp-all add: minus-max-eq-min)

lemma minus-Min-eq-Max [simp]:
finite S \implies S \neq \{\} \implies -\operatorname{Min} S = \operatorname{Max} (\operatorname{uminus} ' S)
by (induct S rule: finite-ne-induct) (simp-all add: minus-min-eq-max)
end

context complete-linorder
begin
lemma Min-Inf:
assumes finite A and A \neq \{\}
shows \operatorname{Min} A = \operatorname{Inf} A
proof -
from assms obtain b B where A = insert b B and finite B by auto
then show \?thesis
  by (simp add: Min.eq-fold complete-linorder-inf-min [symmetric] inf-Inf-fold-inf
inf.commute [of b])
qed

lemma Max-Sup:
assumes finite A and A \neq \{\}
shows \operatorname{Max} A = \operatorname{Sup} A
proof -
from assms obtain b B where A = insert b B and finite B by auto
then show \?thesis
  by (simp add: Max.eq-fold complete-linorder-sup-max [symmetric] sup-Sup-fold-sup
sup.commute [of b])
qed
end

end

59 Set-Interval: Set intervals

theory Set-Interval
imports Lattices-Big Nat-Transfer
begin

context ord
begin

definition lessThan :: 'a => 'a set ((1\{..<\})) where
\{..u\} == \{x. x < u\}

definition
\texttt{atMost} :: 'a => 'a set ((1 \{\ldots\})) where
\{..u\} == \{x. x \leq u\}

definition
\texttt{greaterThan} :: 'a => 'a set ((1\{\ldots\})) where
\{l<..\} == \{x. l<x\}

definition
\texttt{atLeast} :: 'a => 'a set ((1\{\ldots\})) where
\{l..\} == \{x. l\leq x\}

definition
\texttt{greaterThanLessThan} :: 'a => 'a => 'a set ((1\{\ldots\})) where
\{l<..<u\} == \{l<.. Int \{..u\}

definition
\texttt{atLeastLessThan} :: 'a => 'a => 'a set ((1\{\ldots\})) where
\{l..<u\} == \{l..< Int \{..u\}

definition
\texttt{greaterThanAtMost} :: 'a => 'a => 'a set ((1\{\ldots\})) where
\{l<..u\} == \{l<.. Int \{..u\}

definition
\texttt{atLeastAtMost} :: 'a => 'a => 'a set ((1\{\ldots\})) where
\{l..<u\} == \{l..< Int \{..u\}

end

A note of warning when using \{..n\} on type \textit{nat}: it is equivalent to \{0..n\} but some lemmas involving \{m..n\} may not exist in \{..n\}-form as well.

\texttt{syntax}

-UNION-le :: 'a => 'a => 'b set => 'b set ((3\texttt{UN} \ldots\texttt{/} \ldots) [0, 0, 10] 10)
-UNION-less :: 'a => 'a => 'b set => 'b set ((3\texttt{UN} \ldots\texttt{/} \ldots) [0, 0, 10] 10)
-INTER-le :: 'a => 'a => 'b set => 'b set ((3\texttt{INT} \ldots\texttt{/} \ldots) [0, 0, 10] 10)
-INTER-less :: 'a => 'a => 'b set => 'b set ((3\texttt{INT} \ldots\texttt{/} \ldots) [0, 0, 10] 10)

\texttt{syntax} \texttt{(latex output)}
translations

\[ UN \ i \leq n \ A \equiv UN \ i\{..n\}. \ A \]
\[ UN \ i < n \ A \equiv UN \ i\{..<n\}. \ A \]
\[ INT \ i \leq n \ A \equiv INT \ i\{..n\}. \ A \]
\[ INT \ i < n \ A \equiv INT \ i\{..<n\}. \ A \]

59.1 Various equivalences

**lemma** (in ord) lessThan-iff [iff]: \((i: \text{lessThan } k) = (i < k)\)
**by** (simp add: lessThan-def)

**lemma** Compl-lessThan [simp]:
\[ \neg \text{lessThan } k \equiv \text{atLeast } k \]
**apply** (auto simp add: lessThan-def atLeast-def)
**done**

**lemma** single-Diff-lessThan [simp]: \(\neg \text{lessThan } k \equiv \{k\} \)
**by** auto

**lemma** (in ord) greaterThan-iff [iff]: \((i: \text{greaterThan } k) = (k < i)\)
**by** (simp add: greaterThan-def)

**lemma** Compl-greaterThan [simp]:
\[ \neg \text{greaterThan } k \equiv \text{atMost } k \]
**by** (auto simp add: greaterThan-def atMost-def)

**lemma** Compl-atMost [simp]: \(\neg \text{atMost } k \equiv \text{greaterThan } k\)
**apply** (subst Compl-greaterThan [symmetric])
**apply** (rule double-complement)
**done**

**lemma** (in ord) atLeast-iff [iff]: \((i: \text{atLeast } k) = (k \leq i)\)
**by** (simp add: atLeast-def)

**lemma** Compl-atLeast [simp]:
\[ \neg \text{atLeast } k \equiv \text{lessThan } k \]
**by** (auto simp add: lessThan-def atLeast-def)

**lemma** (in ord) atMost-iff [iff]: \((i: \text{atMost } k) = (i \leq k)\)
**by** (simp add: atMost-def)
THEORY “Set-Interval”

lemma atMost-Int-atLeast: \(\forall n::\text{`a::order}.\ \text{atMost } n \text{ atLeast } n = \{n\}\)
by (blast intro: order-antisym)

lemma (in linorder) lessThan-Int-lessThan: \(\{a<..\} \cap \{b<..\} = \{\max a b <..\}\)
by auto

lemma (in linorder) greaterThan-Int-greaterThan: \(\{..< a\} \cap \{..< b\} = \{..< \min a b\}\)
by auto

59.2 Logical Equivalences for Set Inclusion and Equality

lemma atLeast-subset-iff [iff]:
\((\text{atLeast } x \subseteq \text{atLeast } y) = (y \leq (x::`a::order))\)
by (blast intro: order-trans)

lemma atLeast-eq-iff [iff]:
\((\text{atLeast } x = \text{atLeast } y) = (x = (y::`a::linorder))\)
by (blast intro: order-antisym order-trans)

lemma greaterThan-subset-iff [iff]:
\((\text{greaterThan } x \subseteq \text{greaterThan } y) = (y \leq (x::`a::linorder))\)
apply (auto simp add: greaterThan-def)
apply (subst linorder-not-less [symmetric], blast)
done

lemma greaterThan-eq-iff [iff]:
\((\text{greaterThan } x = \text{greaterThan } y) = (x = (y::`a::linorder))\)
apply (rule iffI)
apply (erule equalityE)
apply simp-all
done

lemma atMost-subset-iff [iff]: \((\text{atMost } x \subseteq \text{atMost } y) = (x \leq (y::`a::order))\)
by (blast intro: order-trans)

lemma atMost-eq-iff [iff]: \((\text{atMost } x = \text{atMost } y) = (x = (y::`a::linorder))\)
by (blast intro: order-antisym order-trans)

lemma lessThan-subset-iff [iff]:
\((\text{lessThan } x \subseteq \text{lessThan } y) = (x \leq (y::`a::linorder))\)
apply (auto simp add: lessThan-def)
apply (subst linorder-not-less [symmetric], blast)
done

lemma lessThan-eq-iff [iff]:
\((\text{lessThan } x = \text{lessThan } y) = (x = (y::`a::linorder))\)
apply (rule iffI)
apply (erule equalityE)
apply simp-all
done

lemma lessThan-strict-subset-iff:
  fixes m n :: 'a::linorder
  shows \{..<m\} < \{..<n\} \iff m < n
  by (metis leD lessThan-subset-iff linorder-linear not-less_iff_gr_or_eq psubset_eq)

lemma (in linorder) Ici-subset-Ioi-iff: \{a ..\} \subseteq \{b <..<\} \iff b < a
  by auto

lemma (in linorder) Iic-subset-Iio-iff: \{..<a\} \subseteq \{..<b\} \iff a < b
  by auto

59.3 Two-sided intervals

context ord
begin

lemma greaterThanLessThan-iff [simp]:
  (i : \{l..<..<u\}) = (l < i & i < u)
  by (simp add: greaterThanLessThan-def)

lemma atLeastLessThan-iff [simp]:
  (i : \{l..<u\}) = (l <= i & i < u)
  by (simp add: atLeastLessThan-def)

lemma greaterThanAtMost-iff [simp]:
  (i : \{l..<u\}) = (l < i & i <= u)
  by (simp add: greaterThanAtMost-def)

lemma atLeastAtMost-iff [simp]:
  (i : \{l..u\}) = (l <= i & i <= u)
  by (simp add: atLeastAtMost-def)

The above four lemmas could be declared as iffs. Unfortunately this breaks
many proofs. Since it only helps blast, it is better to leave them alone.

lemma greaterThanLessThan-eq: \{ a <..< b \} = \{ a <..< \} \cap \{..< b \}
  by auto

end

59.3.1 Emptiness, singletons, subset

context order
begin

lemma atLeastAtMost-empty[simp]:

end
$b < a \implies \{a..b\} = \{\}$
by(auto simp: atLeastAtMost-def atLeast-def atMost-def)

**lemma atLeastAtMost-empty-iff [simp]:**
\[
\{a..b\} = \{\} \iff \sim a <= b
\]
by auto (blast intro: order-trans)

**lemma atLeastAtMost-empty-iff2 [simp]:**
\[
\{\} = \{a..b\} \iff \sim a <= b
\]
by auto (blast intro: order-trans)

**lemma atLeastLessThan-empty [simp]:**
\[
b <= a \implies a..<b = \{\}
\]
by(auto simp: atLeastLessThan-def)

**lemma atLeastLessThan-empty-iff [simp]:**
\[
\{a..<b\} = \{\} \iff \sim a < b
\]
by auto (blast intro: le-less-trans)

**lemma atLeastLessThan-empty-iff2 [simp]:**
\[
\{\} = \{a..<b\} \iff \sim a < b
\]
by auto (blast intro: le-less-trans)

**lemma greaterThanAtMost-empty [simp]:**
\[
l <= k \implies k<..l = \{\}
\]
by(auto simp: greaterThanAtMost-def greaterThan-def atMost-def)

**lemma greaterThanAtMost-empty-iff [simp]:**
\[
\{k<..l\} = \{\} \iff \sim k < l
\]
by auto (blast intro: less-le-trans)

**lemma greaterThanAtMost-empty-iff2 [simp]:**
\[
\{\} = \{k<..l\} \iff \sim k < l
\]
by auto (blast intro: less-le-trans)

**lemma greaterThanLessThan-empty [simp]:**
\[
l <= k \implies k<..<l = \{\}
\]
by(auto simp: greaterThanLessThan-def greaterThan-def lessThan-def)

**lemma atLeastAtMost-singleton [simp]:**
\[
\{a..a\} = \{a\}
\]
by(auto simp add: atLeastAtMost-def atMost-def atLeast-def)

**lemma atLeastAtMost-singleton':**
\[
a = b \implies \{a .. b\} = \{a\}
\]
by simp

**lemma atLeastAtMost-subset-iff [simp]:**
\[
\{a..b\} <= \{c..d\} \iff \sim a <= b \mid c <= a \& b <= d
\]
unfolding atLeastAtMost-def atLeast-def atMost-def
by (blast intro: order-trans)

**lemma atLeastAtMost-psubset-iff:**
\[
\{a..b\} <\{c..d\} \iff (\sim a <= b \mid c <= a \& b <= d \& (c < a \mid b < d)) \& c <= d
\]
by(simp add: psubset-eq set-eq-iff less-le-not-le)(blast intro: order-trans)
lemma Icc-eq-Icc[simp]:
\{l..h\} = \{l'..h'\} = (l\leq l' \land h=h' \lor \neg l\leq h \land \neg l'\leq h')
by(simp add: order-class.eq-iff)(auto intro: order-trans)

lemma atLeastAtMost-singleton-iff[simp]:
\{a..b\} = \{c\} \iff a = b \land b = c
proof
  assume \{a..b\} = \{c\}
  hence \*: \neg (\neg a \leq b) unfolding atLeastAtMost-empty-iff[symmetric] by simp
  with \{a..b\} = \{c\} have c \leq a \land b \leq c by auto
  with \* show a = b \land b = c by auto
qed simp

lemma Icc-subset-Ici-iff[simp]:
\{l..h\} \subseteq \{l'..\} = (\neg l \leq h \lor l \geq l')
by(auto simp: subset-eq intro: order-trans)

lemma Icc-subset-Iic-iff[simp]:
\{l..h\} \subseteq \{..h'\} = (\neg l \leq h \lor h \leq h')
by(auto simp: subset-eq intro: order-trans)

lemma not-Ici-eq-empty[simp]: \{l..\} \neq \{}
by(auto simp: set-eq-iff)

lemma not-Iic-eq-empty[simp]: \{..h\} \neq \{}
by(auto simp: set-eq-iff)

lemmas not-empty-eq-Iic-eq-empty[simp] = not-Iic-eq-empty[symmetric]

end

context no-top
begin

lemma not-UNIV-le-Icc[simp]: \neg UNIV \subseteq \{l..h\}
using gt-ex[of h] by(auto simp: subset-eq less-le-not-le)

lemma not-UNIV-le-Iic[simp]: \neg UNIV \subseteq \{..h\}
using gt-ex[of h] by(auto simp: subset-eq less-le-not-le)

lemma not-Ici-le-Icc[simp]: \neg \{l..\} \subseteq \{l'..h'\}
using gt-ex[of h'] by(auto simp: subset-eq less-le)(blast dest:antisym-cone intro: order-trans)

lemma not-Ici-le-Iic[simp]: \neg \{l..\} \subseteq \{..h'\}
using gt-ex[of h']
by (auto simp: subset_eq less-le) (blast dest:antisym_conv intro: order-trans)

end

context no-bot

begin

lemma not-UNIV-le-Ici[simp]: \( \neg \text{UNIV} \subseteq \{l..\} \)
using lt-ex[of l] by (auto simp: subset_eq less-le-not-le)

lemma not-Iic-le-Icc[simp]: \( \neg \{..h\} \subseteq \{l'..h'\} \)
using lt-ex[of l'] by (auto simp: subset_eq less-le)

lemma not-Iic-le-Ici[simp]: \( \neg \{..h\} \subseteq \{l'..\} \)
using lt-ex[of l'] by (auto simp: subset_eq less-le)

lemma not-Iic-le-Ici[simp]: \( \neg \{..h\} \subseteq \{l'..\} \)
using lt-ex[of l'] by (auto simp: subset_eq less-le)

end

context no-top

begin

lemma not-UNIV-eq-Icc[simp]: \( \neg \text{UNIV} = \{l'..h'\} \)
using gt-ex[of h'] by (auto simp: set-eq_iff less-le-not-le)

lemmas not-Icc-eq-UNIV[simp] = not-UNIV-eq-Icc[symmetric]

lemma not-UNIV-eq-Iic[simp]: \( \neg \text{UNIV} = \{..h'\} \)
using gt-ex[of h'] by (auto simp: set-eq_iff less-le-not-le)

lemmas not-Iic-eq-UNIV[simp] = not-UNIV-eq-Iic[symmetric]

lemma not-Icc-eq-Ici[simp]: \( \neg \{l..h\} = \{l'..\} \)
unfolding atLeastAtMost-def using not-Ici-le-Iic[of l'] by blast

lemmas not-Ici-eq-Icc[simp] = not-Icc-eq-Ici[symmetric]

lemma not-Iic-eq-Ici[simp]: \( \neg \{..h\} = \{l'..\} \)
using not-Ici-le-Iic[of l' h] by blast

lemmas not-Ici-eq-Icc[simp] = not-Icc-eq-Ici[symmetric]

end

context no-bot
begin

lemma not-UNIV-eq-Ici[simp]: \( \neg \text{UNIV} = \{l'..\} \)
using lt-ex[of \(l'\)] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Ici-eq-UNIV[simp] = not-UNIV-eq-Ici[symmetric]

lemma not-Icc-eq-Iic[simp]: \( \neg\{l..h\} = \{..h'\} \)
unfolding atLeastAtMost-def using not-Iic-le-Ici[of \(h'\)] by blast

lemmas not-Iic-eq-Icc[simp] = not-Icc-eq-Iic[symmetric]
end

context dense-linorder
begin

lemma greaterThanLessThan-empty-iff[simp]:
\( \{a..<b\} = \{} \leftrightarrow b \leq a \)
using dense[of a b] by (cases a < b) auto

lemma greaterThanLessThan-empty-iff2[simp]:
\( \{\} = \{a..<b\} \leftrightarrow b \leq a \)
using dense[of a b] by (cases a < b) auto

lemma atLeastLessThan-subseteq-atLeastAtMost-iff:
\( \{a..<b\} \subseteq \{c..d\} \leftrightarrow (a < b \rightarrow c \leq a \land b \leq d) \)
using dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanAtMost-subseteq-atLeastAtMost-iff:
\( \{a..<b\} \subseteq \{c..d\} \leftrightarrow (a < b \rightarrow c \leq a \land b \leq d) \)
using dense[of a min c b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-atLeastLessThan-iff:
\( \{a..<b\} \subseteq \{c..<d\} \leftrightarrow (a < b \rightarrow c \leq a \land b < d) \)
using dense[of max a d b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma atLeastAtMost-subseteq-atLeastLessThan-iff:
\( \{a..b\} \subseteq \{c..<d\} \leftrightarrow (a \leq b \rightarrow c \leq a \land b < d) \)
using dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanAtMost-subseteq-atLeastLessThan-iff:
\( \{a..<b\} \subseteq \{c..d\} \leftrightarrow (a < b \rightarrow c \leq a \land b < d) \)
using dense[of a min c b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-atLeastLessThan-iff:
\{a <..< b\} ⊆ \{c ..< d\} ←→ (a < b → c ≤ a ∧ b ≤ d)
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-greaterThanAtMost-iff:
\{a <..< b\} ⊆ \{c ..< d\} ←→ (a < b → c ≤ a ∧ b ≤ d)
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

end

context no-top begin

lemma greaterThan-non-empty[simp]: \{x <..\} ≠ {}
using gt-ex[of x] by auto

end

context no-bot begin

lemma lessThan-non-empty[simp]: {..< x} ≠ {}
using lt-ex[of x] by auto

end

lemma (in linorder) atLeastLessThan-subset-iff:
\{a ..< b\} ⊆ \{c ..< d\} =⇒ b <= a | c<=a & b<=d
apply (auto simp:subset-eq Ball-def)
apply(frule-tac x=a in spec)
apply(erule-tac x=d in allE)
apply (simp add: less-imp-le)
done

lemma atLeastLessThan-inj:
fixes a b c d :: 'a::linorder
assumes eq: \{a ..< b\} = \{c ..< d\} and a < b c < d
shows a = c b = d
using assms by (metis atLeastLessThan-subset-iff eq less-le-not-le linorder-antisym-cone2 subset-refl)+

lemma atLeastLessThan-eq-iff:
fixes a b c d :: 'a::linorder
assumes a < b c < d
shows \{a ..< b\} = \{c ..< d\} ←→ a = c ∧ b = d
using atLeastLessThan-inj assms by auto

lemma (in linorder) Ioc-inj: \{a <.. b\} = \{c <.. d\} \iff (b \leq a \land d \leq c) \lor a = c \land b = d
  by (metis eq_iff greaterThanAtMost-iff2 greaterThanAtMost-iff le-cases not-le)

lemma (in order) Iio-Int-singleton: \{..<k\} \cap \{x\} = (if x < k then \{x\} else \{\})
  by auto

lemma (in linorder) Ioc-subset-iff: \{a<..<b\} \subseteq \{c<..<d\} \iff (b \leq a \lor c \leq a \land b \leq d)
  by (auto simp: subset_eq Ball_def (metis less-le not-less)

lemma (in order-bot) atLeast-eq-UNIV-iff: \{x..\} = UNIV \iff x = bot
  by (auto simp: set_eq_iff intro: le-bot)

lemma (in order-top) atMost-eq-UNIV-iff: \{..x\} = UNIV \iff x = top
  by (auto simp: set_eq_iff intro: top-le)

lemma (in bounded-lattice) atLeastAtMost-eq-UNIV-iff: \{x..y\} = UNIV \iff x = bot \land y = top
  by (auto simp: set_eq_iff intro: top-le le-bot)

lemma Iio-eq-empty-iff: \{..<n::'a::\text{linorder, order-bot}\} = {} \iff n = bot
  by (auto simp: set_eq_iff not-less le-bot)

lemma Iio-eq-empty-iff-nat: \{..<n::\text{nat}\} = {} \iff n = 0
  by (simp add: Iio-eq-empty-iff bot-nat-def)

59.4 Infinite intervals
context dense-linorder
begin

lemma infinite-Ioo:
  assumes a < b
  shows \neg\ finite \{a<..<b\}
proof
  assume fin: finite \{a<..<b\}
  moreover have ne: \{a<..<b\} \neq {}
    using (a < b) by auto
  ultimately have a < Max \{a<..<b\} Max \{a<..<b\} < b
    using Max-in[of \{a<..<b\}] by auto
  then obtain x where \Max \{a<..<b\} < x < b
    using dense[of Max \{a<..<b\}] by auto
  then have x \in \{a<..<b\}
    using (a < Max \{a<..<b\}) by auto
  then have x \leq Max \{a<..<b\}

end
using fin by auto
with ⟨Max {a <..< b} < x⟩ show False by auto
qed

lemma infinite-Icc: a < b \implies \neg finite \{a .. b\}
  using greaterThanLessThan-subseteq-atLeastAtMost-iff[of a b a b] infinite-Ioo[of a b]
  by (auto dest: finite-subset)

lemma infinite-Ico: a < b \implies \neg finite \{a ..< b\}
  using greaterThanLessThan-subseteq-atLeastLessThan-iff[of a b a b] infinite-Ioo[of a b]
  by (auto dest: finite-subset)

lemma infinite-Ioc: a < b \implies \neg finite \{a..< b\}
  using greaterThanLessThan-subseteq-greaterThanAtMost-iff[of a b a b] infinite-Ioo[of a b]
  by (auto dest: finite-subset)

end

lemma infinite-Iio: \neg finite \{..< a :: 'a :: no-bot, linorder\}
proof
  assume finite \{..< a\}
  then have \*: \(\forall x. x < a \implies Min \{..< a\} \leq x\)
    by auto
  obtain x where x < a
    using lt-ex by auto

  obtain y where y < Min \{..< a\}
    using lt-ex by auto
  also have Min \{..< a\} \leq x
    using (x < a) by fact
  also note \(x < a\)
  finally have Min \{..< a\} \leq y
    by fact
  with \(y < Min \{..< a\}\) show False by auto
qed

lemma infinite-Iic: \neg finite \{a :: 'a :: no-bot, linorder\}
using infinite-Ioo[of a] finite-subset[of {..< a} {.. a}]
by (auto simp: subset-eq less-imp-le)

lemma infinite-Ioi: \neg finite \{a :: 'a :: no-top, linorder\} <..}
proof
  assume finite \{a <..\}
  then have \*: \(\forall x. a < x \implies x \leq Max \{a <..\}\)
    by auto
obtain \( y \) where \( \text{Max} \{ a <.. \} < y \)
using \( \text{gt-ex} \) by \( \text{auto} \)

obtain \( x \) where \( a < x \)
using \( \text{gt-ex} \) by \( \text{auto} \)
also then have \( x \leq \text{Max} \{ a <.. \} \)
by \( \text{fact} \)
also note \( \langle \text{Max} \{ a <.. \} < y \rangle \)
finally have \( y \leq \text{Max} \{ a <.. \} \)
by \( \text{fact} \)
with \( \langle \text{Max} \{ a <.. \} < y \rangle \) show \( \text{False} \) by \( \text{auto} \)
qed

lemma \( \text{infinite-Ici} \): \( \neg \text{finite} \{ a :: 'a :: \{ \text{no-top, linorder} \} .. \} \)
using \( \text{infinite-Ioi[of a]} \) \( \text{finite-subset[of} \{ a <.. \} \{ a .. \}] \)
by \( \langle \text{auto simp: subset-eq less-imp-le} \rangle \)

59.4.1 Intersection
context \( \text{linorder} \)
begins

lemma \( \text{Int-atLeastAtMost[simp]:} \{ a..b \} \text{ Int} \{ c..d \} = \{ \text{max} a c .. \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-atLeastAtMostR1[simp]:} \{ ..b \} \text{ Int} \{ c..d \} = \{ c .. \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-atLeastAtMostR2[simp]:} \{ a.. \} \text{ Int} \{ c..d \} = \{ \text{max} a c .. d \} \)
by \( \text{auto} \)

lemma \( \text{Int-atLeastAtMostL1[simp]:} \{ a..b \} \text{ Int} \{ ..d \} = \{ a .. \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-atLeastAtMostL2[simp]:} \{ a..b \} \text{ Int} \{ .. \} = \{ \text{max} a c .. b \} \)
by \( \text{auto} \)

lemma \( \text{Int-atLeastLessThan[simp]:} \{ a..<b \} \text{ Int} \{ c..<d \} = \{ \text{max} a c ..< \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-greaterThanAtMost[simp]:} \{ a<..b \} \text{ Int} \{ c<..d \} = \{ \text{max} a c <.. \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-greaterThanLessThan[simp]:} \{ a<..<b \} \text{ Int} \{ c<..<d \} = \{ \text{max} a c <..< \text{min} b d \} \)
by \( \text{auto} \)

lemma \( \text{Int-atMost[simp]:} \{ ..a \} \cap \{ ..b \} = \{ .. \text{min} a b \} \)
THEORY “Set-Interval”

by (auto simp: min-def)

lemma \textit{loc-disjoint}: \{a..b\} \cap \{c..d\} = \{\} \iff b \leq a \lor d \leq c \lor b \leq c \lor d \leq a

using assms by auto

end

context complete-lattice begin

lemma shows Sup-atLeast\[simp\]: Sup \{x ..\} = top

and Sup-greaterThanAtLeast\[simp\]: x < top \implies Sup \{x ..\} = top

and Sup-atMost\[simp\]: Sup \{.. y\} = y

and Sup-atLeastAtMost\[simp\]: x \leq y \implies Sup \{x .. y\} = y

and Sup-greaterThanAtMost\[simp\]: x < y \implies Sup \{x .. y\} = y

by (auto intro!: Sup-eqI)

end

lemma shows Inf-atMost\[simp\]: Inf \{.. x\} = bot

and Inf-atMostLessThan\[simp\]: top < x \implies Inf \{..< x\} = bot

and Inf-atLeast\[simp\]: Inf \{x ..\} = x

and Inf-atLeastAtMost\[simp\]: x \leq y \implies Inf \{x .. y\} = x

and Inf-atLeastLessThan\[simp\]: x < y \implies Inf \{x ..< y\} = x

by (auto intro!: Inf-eqI Sup-eqI intro: dense-le dense-le-bounded dense-ge dense-ge-bounded)

59.5 Intervals of natural numbers

59.5.1 The Constant \textit{lessThan}

lemma lessThan-0 \[simp\]: lessThan (0::nat) = \{}

by (simp add: lessThan-def)

lemma lessThan-Suc: lessThan (Suc k) = insert k (lessThan k)

by (simp add: lessThan-def less-Suc-eq, blast)

The following proof is convenient in induction proofs where new elements
get indices at the beginning. So it is used to transform $\{..<\text{Suc } n\}$ to $\emptyset$ and $\{..<n\}$.

**Lemma** \texttt{lessThan-Suc-eq-insert-0}: $\{..<\text{Suc } n\} = \text{insert } 0 (\text{Suc } \{..<n\})$

**Proof**

```proof safe
fix $x$
assume $x < \text{Suc } n$
$x \notin \text{Suc } \{..<n\}$
then have $x \neq \text{Suc } (x - 1)$ by auto
with $\langle x < \text{Suc } n \rangle$ show $x = 0$ by auto
qed
```

**Lemma** \texttt{lessThan-Suc-atMost}: $\text{lessThan (Suc } k\text{)} = \text{atMost } k$

**Proof**

```by (simp add: \texttt{lessThan-def} \texttt{atMost-def} \texttt{less-Suc-eq-le})
```

**Lemma** \texttt{UN-lessThan-UNIV}: $(\text{UN } m::\text{nat. lessThan } m) = \text{UNIV}$

**Proof**

```by blast
```

### 59.5.2 The Constant \texttt{greaterThan}

**Lemma** \texttt{greaterThan-0 [simp]}: $\text{greaterThan } 0 = \text{range Suc}$

**Proof**

```apply (simp add: \texttt{greaterThan-def})
apply (blast dest: gr0-conv-Suc [THEN iffD1])
done
```

**Lemma** \texttt{greaterThan-Suc}: $\text{greaterThan } (\text{Suc } k) = \text{greaterThan } k - \{\text{Suc } k\}$

**Proof**

```apply (simp add: \texttt{greaterThan-def})
apply (auto elim: linorder-neqE)
done
```

**Lemma** \texttt{INT-greaterThan-UNIV}: $(\text{INT } m::\text{nat. greaterThan } m) = \{\}$

**Proof**

```by blast
```

### 59.5.3 The Constant \texttt{atLeast}

**Lemma** \texttt{atLeast-0 [simp]}: $\text{atLeast } (0::\text{nat}) = \text{UNIV}$

**Proof**

```by (unfold \texttt{atLeast-def} \texttt{UNIV-def}, simp)
```

**Lemma** \texttt{atLeast-Suc}: $\text{atLeast } (\text{Suc } k) = \text{atLeast } k - \{k\}$

**Proof**

```apply (simp add: \texttt{atLeast-def})
apply (simp add: Suc-le-eq)
apply (simp add: order-le-less, blast)
done
```

**Lemma** \texttt{atLeast-Suc-greaterThan}: $\text{atLeast } (\text{Suc } k) = \text{greaterThan } k$

**Proof**

```by (auto simp add: \texttt{greaterThan-def} \texttt{atLeast-def} \texttt{less-Suc-eq-le})
```

**Lemma** \texttt{UN-atLeast-UNIV}: $(\text{UN } m::\text{nat. atLeast } m) = \text{UNIV}$

**Proof**

```by blast
```
59.5.4 The Constant atMost

**lemma atMost-0** [simp]: atMost (0::nat) = {0}
by (simp add: atMost-def)

**lemma atMost-Suc**: atMost (Suc k) = insert (Suc k) (atMost k)
apply (simp add: atMost-def)
apply (simp add: less-Suc-eq order-le-less, blast)
done

**lemma UN-atMost-UNIV**: (UN m::nat. atMost m) = UNIV
by blast

59.5.5 The Constant atLeastLessThan

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

**lemma atLeast0LessThan**: {0::nat..<n} = {..<n}
by(simp add:lessThan-def atLeastLessThan-def)

**lemma atLeast0AtMost**: {0..n::nat} = {..n}
by(simp add:atMost-def atLeastAtMost-def)

declare atLeast0LessThan[symmetric, code-unfold]
atLeast0AtMost[symmetric, code-unfold]

**lemma atLeastLessThan0**: {m..<0::nat} = {}
by (simp add: atLeastLessThan-def)

59.5.6 Intervals of nats with Suc

Not a simprule because the RHS is too messy.

**lemma atLeastLessThanSuc**: 
{m..<Suc n} = (if m ≤ n then insert n {m..<n} else {}}
by (auto simp add: atLeastLessThan-def)

**lemma atLeastLessThan-singleton** [simp]: {m..<Suc m} = {m}
by (auto simp add: atLeastLessThan-def)

**lemma atLeastLessThanSuc-atLeastAtMost**: {l..<Suc u} = {l..u}
by (simp add: less-Suc-atMost atLeastAtMost-def atLeastLessThan-def)

**lemma atLeastSucAtMost-greaterThanAtMost**: {Suc l..u} = {l..<u}
by (simp add: atLeast-Suc-greaterThan atLeastAtMost-def greaterThanAtMost-def)
lemma atLeastSucLessThan-greaterThanLessThan: \{\text{Suc}\ l..<u\} = \{l..<u\}
  by (simp add: atLeast-Suc-greaterThan atLeastLessThan-def
greaterThanLessThan-def)

lemma atLeastAtMostSuc-conv: \(m \leq \text{Suc}\ n \implies \{m..\text{Suc}\ n\} = \text{insert} (\text{Suc} n)\{m..n\}\)
  by (auto simp add: atLeastAtMost-def)

lemma atLeastAtMost-insertL: \(m \leq n \implies \text{insert} m \{\text{Suc}\ m..n\} = \{m..n\}\)
  by auto

The analogous result is useful on \(\text{int}\):

lemma atLeastAtMostPlus1-int-conv: \(m < 1+\text{Suc}\ n \implies \text{insert} (1+\text{Suc}\ n)\{m..\text{Suc}\ n::\text{int}\} = \text{insert} \{m..\text{Suc}\ n::\text{int}\}\)
  by (auto intro: set-eqI)

lemma atLeastLessThan-add-Un: \(i \leq j \implies \{i..<\text{Suc}\ j+k\} = \{i..<\text{Suc}\ j\} \cup \{j..<\text{Suc}\ j+k::\text{nat}\}\)
  apply (induct k)
  apply (simp-all add: atLeastLessThanSuc)
  done

59.5.7 Intervals and numerals

lemma lessThan-nat-numeral: — Evaluation for specific numerals
  \(\text{lessThan} \ (\text{numeral}\ k::\text{nat}) = \text{insert} (\text{pred-numeral}\ k) \ (\text{lessThan} \ (\text{pred-numeral}\ k))\)
  by (simp add: numeral-eq-Suc lessThan-Suc)

lemma atMost-nat-numeral: — Evaluation for specific numerals
  \(\text{atMost} \ (\text{numeral}\ k::\text{nat}) = \text{insert} (\text{numeral}\ k) \ (\text{atMost} \ (\text{pred-numeral}\ k))\)
  by (simp add: numeral-eq-Suc atMost-Suc)

lemma atLeastLessThan-nat-numeral: — Evaluation for specific numerals
  \(\text{atLeastLessThan}\ m \ (\text{numeral}\ k::\text{nat}) =\)
  \(\text{if}\ m \leq (\text{pred-numeral}\ k)\ \text{then} \text{insert} (\text{pred-numeral}\ k) \ (\text{atLeastLessThan}\ m \ (\text{pred-numeral}\ k))\)
  \(\text{else}\ \{\}\)
  by (simp add: numeral-eq-Suc atLeastLessThanSuc)

59.5.8 Image

lemma image-add-atLeastAtMost: \(\{\text{Suc}\ n..n+k\} \cdot \{i..j\} = \{i+k..j+k\}\)
  by (auto simp add: atLeast-atLeast)

proof
  show \(?A \subseteq \{\}\) by auto
next
  show \(?B \subseteq \{\}\)
  proof
    fix n assume a: \(n : ?B\)
hence $n - k : \{i..j\}$ by auto
moreover have $n = (n - k) + k$ using $a$ by auto
ultimately show $n : ?A$ by blast
qed

lemma image-add-atLeastLessThan:
($\%n::nat. n+k \cdot \{i..<j\} = \{i+k..<j+k\}$ (is $?A = ?B$)
proof
show $?A \subseteq ?B$ by auto
next
show $?B \subseteq ?A$
proof
fix $n$ assume $a: n : ?B$
hence $n - k : \{i..<j\}$ by auto
moreover have $n = (n - k) + k$ using $a$ by auto
ultimately show $n : ?A$ by blast
qed

qed

corollary image-Suc-atLeastAtMost[simp]:
Suc $\cdot \{i..j\} = \{Suc i..Suc j\}$
using image-add-atLeastAtMost[where $k=Suc 0$] by simp

corollary image-Suc-atLeastLessThan[simp]:
Suc $\cdot \{i..<j\} = \{Suc i..<Suc j\}$
using image-add-atLeastLessThan[where $k=Suc 0$] by simp

lemma image-add-int-atLeastLessThan:
($\%x. x + (l::int) \cdot \{0..<u-l\} = \{l..<u\}$
apply (auto simp add: image-def)
apply (rule-tac $x = x - l$ in bexI)
apply auto
done

lemma image-minus-const-atLeastLessThan-nat:
fixes $c :: nat$
sows ($\lambda i. i - c \cdot \{x..<y\} =$
(if $c < y$ then $\{x - c..<y - c\}$ else if $x < y$ then $\{0\}$ else $\{\}$)
(is $- = ?right$)
proof safe
fix $a$ assume $a: a \in ?right$
show $a \in (\lambda i. i - c \cdot \{x..<y\}$
proof cases
assume $c < y$ with a show $?thesis$
by (auto intro!: image-eqI[of - - a + c])
next
assume $\neg c < y$ with a show $?thesis$
by (auto intro!: image-eqI[of - - x] split: split-if_asm)
lemma image-int-atLeastLessThan: int ' {a..<b} = {int a..<int b}
  by (auto intro!: image-eqI [where x = nat x for x])

context ordered-ab-group-add begin

lemma fixes x :: 'a
  shows image-uminus-greaterThan[simp]: uminus ' {x<..} = {..<−x}
  and image-uminus-atLeast[simp]: uminus ' {x..} = {..−x}

proof safe
  fix y assume y < −x
  hence x < −y using neg-less-iff-less[of −y x] by simp
  have −(−y) ∈ uminus ' {x<..}
    by (rule imageI) (simp add: ∗)
  thus y ∈ uminus ' {x<..} by simp
next
  fix y assume y ≤ −x
  have −(−y) ∈ uminus ' {x..}
    by (rule imageI) (insert ⟨y ≤ −x⟩ THEN le_imp_neg_le, simp)
  thus y ∈ uminus ' {x..} by simp
qed simp-all

lemma fixes x :: 'a
  shows image-uminus-lessThan[simp]: uminus ' {..<x} = {−x<..}
  and image-uminus-atMost[simp]: uminus ' {..x} = {−x..}

proof −
  have uminus ' {..<x} = uminus ' uminus ' {−x<..}
    and uminus ' {..x} = uminus ' uminus ' {−x..} by simp-all
  thus uminus ' {..<x} = {−x<..} and uminus ' {..x} = {−x..}
    by (simp-all add: image-image
del: image-uminus-greaterThan image-uminus-atLeast)
qed

lemma fixes x :: 'a
  shows image-uminus-atLeastAtMost[simp]: uminus ' {x..y} = {−y..−x}
  and image-uminus-greaterThanAtMost[simp]: uminus ' {x<y..} = {−y..<−x}
  and image-uminus-atLeastLessThan[simp]: uminus ' {x..<y} = {−y<..<−x}
  and image-uminus-greaterThanLessThan[simp]: uminus ' {x..<y} = {−y<..<−x}
  by (simp-all add: atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def
       greaterThanLessThan-def image-Int[OF inj-uminus] Int-commute)
end
59.5.9 Finiteness

lemma finite-lessThan [iff]: fixes k :: nat shows finite {..<k} 
  by (induct k) (simp-all add: lessThan-Suc)

lemma finite-atMost [iff]: fixes k :: nat shows finite {..k} 
  by (induct k) (simp-all add: atMost-Suc)

lemma finite-greaterThanLessThan [iff]: 
  fixes l :: nat shows finite {l..<..<u} 
  by (simp add: greaterThanLessThan-def)

lemma finite-atLeastLessThan [iff]: 
  fixes l :: nat shows finite {l..<..<u} 
  by (simp add: atLeastLessThan-def)

lemma finite-greaterThanAtMost [iff]: 
  fixes l :: nat shows finite {l..<..<u} 
  by (simp add: greaterThanAtMost-def)

lemma finite-atLeastAtMost [iff]: 
  fixes l :: nat shows finite {l..<..<u} 
  by (simp add: atLeastAtMost-def)

A bounded set of natural numbers is finite.

lemma bounded-nat-set-is-finite: 
  (ALL i::N. i < (n::nat)) ==> finite N 
  apply (rule finite-subset) 
  apply (rule-tac [2] finite-lessThan, auto) 
  done

A set of natural numbers is finite iff it is bounded.

lemma finite-nat-set-iff-bounded: 
  finite(N::nat set) = (EX m. ALL n:N. n<m) (is ?F = ?B) 
  proof 
  assume f:?F show ?B 
    using Max-ge[OF (?F), simplified less-Suc-eq-le[symmetric]] by blast 
  next 
  assume ?B show ?F using (?B) by(blast intro:bounded-nat-set-is-finite) 
  qed

lemma finite-nat-set-iff-bounded-le: 
  finite(N::nat set) = (EX m. ALL n:N. n<=m) 
  apply(simp add:finite-nat-set-iff-bounded) 
  apply(blast dest:less-imp-le-nat le-imp-less-Suc) 
  done

lemma finite-less-ub: 
  !!f::nat=>nat. (!n. n < f n) ==> finite {n. f n < u} 
  by (rule-tac B={..u} in finite-subset, auto intro: order-trans)
Any subset of an interval of natural numbers the size of the subset is exactly that interval.

**Lemma** `subset-card-intvl-is-intvl`:
- **Assumes** \( A \subseteq \{k..<k + \text{card } A\} \)
- **Shows** \( A = \{k..<k + \text{card } A\} \)

**Proof** (cases finite \( A \))

**Case** `True`

**From** this and `assms` **Show** `?thesis`

**Proof** (induct \( A \) rule: `finite-linorder-max-induct`)
- **Case** `empty` **Thus** `?case` by `auto`
- **Next**
  - **Case** `(insert b A)` **Hence** `*: b \notin A` by `auto`
  - **With** `insert` **Have** \( A \subseteq \{k..<k + \text{card } A\} \) and \( b = k + \text{card } A \)` by `fastforce`
  - **With** `insert *` **Show** `?case` by `auto`

**Qed**

**Next**
- **Case** `False`
- **With** `assms` **Show** `?thesis` by `simp`

**Qed**

59.5.10 Proving Inclusions and Equalities between Unions

**Lemma** `UN-le-eq-Un0`:

\( (\bigcup i \leq n :: \text{nat}. \ M i) = (\bigcup i \in \{1..n\}. \ M i) \cup M 0 \) (is `?A = ?B`)

**Proof**

**Show** `?A \subseteq ?B`

**Proof**
- **Fix** \( x \) **Assume** \( x : ?A \)
- **Then** **Obtain** \( i \) **Where** \( i : i \leq n \cdot x : M i \) by `auto`
- **Show** `x : ?B`
- **Proof** (cases \( i \))
  - **Case** `0` **With** \( i \) **Show** `?thesis` by `simp`
  - **Next**
    - **Case** `(Suc j)` **With** \( i \) **Show** `?thesis` by `auto`

**Qed**

**Next**
- **Show** `?B \subseteq ?A` by `auto`

**Qed**

**Lemma** `UN-le-add-shift`:

\( (\bigcup i \leq n :: \text{nat}. \ M (i+k)) = (\bigcup i \in \{k..n+k\}. \ M i) \) (is `?A = ?B`)

**Proof**

**Show** `?A \subseteq ?B` by `fastforce`

**Next**
- **Show** `?B \subseteq ?A`

**Proof**
THEORY “Set-Interval"

fix x assume x : ?B
then obtain i where i: i : {k..n+k} x : M(i) by auto
hence i−k≤n & x : M((i−k)+k) by auto
thus x : ?A by blast
qed

qed

lemma UN-UN-finite-eq: (∪ i::nat. ∪ i\in\{0..<n\}. A i) = (∪ n. A n)
by (auto simp add: atLeast0LessThan)

lemma UN-finite-subset: (!! i::nat. (∪ i\in\{0..<n\}. A i) ⊆ C) ⇒ (∪ n. A n) ⊆ C
by (subst UN-UN-finite-eq [symmetric]) blast

lemma UN-finite2-subset:
(!! i::nat. (∪ i\in\{0..<n\}. A i) ⊆ (∪ i\in\{0..<n+k\}. B i)) ⇒ (∪ n. A n) ⊆ (∪ n. B n)
apply (rule UN-finite-subset)
apply (subst UN-UN-finite-eq [symmetric, of B])
apply blast
done

lemma UN-finite2-eq:
(!! i::nat. (∪ i\in\{0..<n\}. A i) = (∪ i\in\{0..<n+k\}. B i)) ⇒ (∪ n. A n) = (∪ n. B n)
apply (rule subset-antisym)
apply (rule UN-finite2-subset, blast)
apply (rule UN-finite2-subset [where k=k])
apply (force simp add: atLeastLessThan-add-Un [of 0])
done

59.5.11 Cardinality

lemma card-lessThan [simp]: card {..<u} = u
by (induct u, simp-all add: lessThan-Suc)

lemma card-atMost [simp]: card {..u} = Suc u
by (simp add: lessThan-Suc-atMost [THEN sym])

lemma card-atLeastLessThan [simp]: card {l..<u} = u − l
proof −
have {l..<u} = (%x. x + l) ′ {..<u−l}
  apply (auto simp add: image-def atLeastLessThan-def lessThan-def)
  apply (rule-tac x = x − l in exI)
  apply arith
done
then have card {l..<u} = card {..<u−l}
  by (simp add: card-image inj-on-def)
then show ?thesis
  by simp
qed

lemma card-atLeastAtMost [simp]: \( \text{card} \{l..u\} = \text{Suc} \ u - l \)
by (subst atLeastLessThanSuc-atLeastAtMost [THEN sym], simp)

lemma card-greaterThanAtMost [simp]: \( \text{card} \{l<..u\} = u - l \)
by (subst atLeastSucAtMost-greaterThanAtMost [THEN sym], simp)

lemma card-greaterThanLessThan [simp]: \( \text{card} \{l<..<u\} = u - \text{Suc} \ l \)
by (subst atLeastSucLessThan-greaterThanLessThan [THEN sym], simp)

lemma ex-bij-betw-nat-finite:
finite M \( \Rightarrow \exists h. \text{bij-betw} \ h \ \{0..<\text{card} \ M\} \ M \)
apply(drule finite-imp-nat-seq-image-inj-on)
apply(auto simp: atLeast0LessThan [symmetric] lessThan-def [symmetric] card-image bij-betw-def)
done

lemma ex-bij-betw-finite-nat:
finite M \( \Rightarrow \exists h. \text{bij-betw} \ h \ M \ \{0..<\text{card} \ M\} \)
by (blast dest: ex-bij-betw-nat-finite bij-betw-inv)

lemma finite-same-card-bij:
finite A \( \Rightarrow \) finite B \( \Rightarrow \) \( \text{card} \ A = \text{card} \ B \Rightarrow \exists h. \text{bij-betw} \ h \ A \ B \)
apply(drule ex-bij-betw-finite-nat)
apply(drule ex-bij-betw-nat-finite)
apply(auto intro!: bij-betw-trans)
done

lemma ex-bij-betw-nat-finite-1:
finit\(e\ M \Rightarrow \exists h. \text{bij-betw} \ h \ \{1..<\text{card} \ M\} \ M \)
by (rule finite-same-card-bij) auto

lemma bij-betw-iff-card:
assumes \( \text{FIN}: \ \text{finite} \ A \ \text{and} \ \text{FIN'}: \ \text{finite} \ B \)
shows \( \text{BIJ}: (\exists f. \text{bij-betw} \ f \ A \ B) \leftrightarrow (\text{card} \ A = \text{card} \ B) \)
using assms
proof(auto simp add: bij-betw-same-card)
  assume *: \(\text{card} \ A = \text{card} \ B\)
  obtain f where bij-betw f A \( \{0..<\text{card} \ A\} \)
  using \(\text{FIN} \ \text{ex-bij-betw-finite-nat} \) by blast
  moreover obtain g where bij-betw g \( \{0..<\text{card} \ B\} \ B \)
  using \(\text{FIN'} \ \text{ex-bij-betw-nat-finite} \) by blast
  ultimately have bij-betw \((g \circ f) \ A \ B\)
  using * by (auto simp add: bij-betw-trans)
  thus \(\exists f. \text{bij-betw} \ f \ A \ B\) by blast
qed

lemma inj-on-iff-card-le:
assumes FIN: finite A and FIN': finite B
shows (\exists f. inj-on f A \land f' A \leq B) = (\card A \leq \card B)
proof (safe intro!: card-inj-on-le)
  assume \cdot:*: \card A \leq \card B
  obtain f where 1: inj-on f A and 2: f' A = \{0 ..< \card A\} using FIN ex-bij-betw-finite-nat unfolding bij-betw-def by force
  moreover obtain g where inj-on g \{0 ..< \card B\} and 3: g' \{0 ..< \card B\} = B using FIN' ex-bij-betw-nat-finite unfolding bij-betw-def by force
  ultimately have inj-on g (f' A) using subset-inj-on[of g - f' A] * by force
  moreover {have \{0 ..< \card A\} \leq \{0 ..< \card B\} using \cdot:* by force
    with 2 have f' A \leq \{0 ..< \card B\} by blast
    hence (g o f)' A \leq B unfolding comp-def using 3 by force
  }
  ultimately show (\exists f. inj-on f A \land f' A \leq B) by blast
qed (insert assms, auto)

59.6 Intervals of integers
lemma atLeastLessThanPlusOne-atLeastAtMost-int: \{l..u+1\} = \{l..(u::int)\}
by (auto simp add: atLeastAtMost-def atLeastLessThan-def)

lemma atLeastPlusOneAtMost-greaterThanAtMost-int: \{l+1..u\} = \{l..<(u::int)\}
by (auto simp add: atLeastAtMost-def greaterThanAtMost-def)

lemma atLeastPlusOneLessThan-greaterThanLessThan-int:
  \{l+1..<u\} = \{l..<u::int\}
by (auto simp add: atLeastLessThan-def greaterThanLessThan-def)

59.6.1 Finiteness
lemma image-atLeastZeroLessThan-int: \{0::int..<u\} = int ' \{..<nat u\}
apply (unfold image-def lessThan-def)
apply auto
apply (rule_tac x = nat x in ex1)
apply (auto simp add: zless-nat-eq-int-zless [THEN sym])
done

lemma finite-atLeastZeroLessThan-int: finite \{0::int..<u\}
apply (cases 0 \leq u)
apply (subst image-atLeastZeroLessThan-int, assumption)
apply (rule finite-image1)
apply auto
done

lemma finite-atLeastLessThan-int [iff]: finite \{l..<u::int\}
apply (subgoal-tac (%x. x + l) ' \{0..<u-l\} = \{l..<u\})
apply (erule subst)
apply (rule finite-imageI)
apply (rule finite-atLeastZeroLessThan-int)
apply (rule image-add-int-atLeastLessThan)
done

lemma finite-atLeastAtMost-int [iff]: finite {l..(u::int)}
  by (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp)

lemma finite-greaterThanAtMost-int [iff]: finite {l..<u::int}
  by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

lemma finite-greaterThanLessThan-int [iff]: finite {l<..<u::int}
  by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

59.6.2 Cardinality

lemma card-atLeastZeroLessThan-int: card {(0::int)..<u} = nat u
  apply (cases 0 ≤ u)
  apply (subst image-atLeastZeroLessThan-int, assumption)
  apply (subst card-image)
  apply (auto simp add: inj-on-def)
done

lemma card-atLeastLessThan-int [simp]: card {l..<u} = nat (u - l)
  apply (subgoal-tac (\x. x + l) \{0..<u-l\} = \{l..<u\})
  apply (erule ssubst, rule card-atLeastZeroLessThan-int)
  apply (subgoal-tac \(\%x. x + l\) \{0..<u-l\} = \{l..<u\})
  apply (erule subst)
  apply (rule card-image)
  apply (simp add: inj-on-def)
  apply (rule image-add-int-atLeastLessThan)
done

lemma card-atLeastAtMost-int [simp]: card {l..u} = nat (u - l + 1)
  apply (subgoal-tac \{0..<u\} = \{0..<u-l\})
  apply (erule subst, rule card-atLeastZeroLessThan-int)
  apply (erule subst)
  apply (rule card-image)
  apply (simp add: inj-on-def)
  apply (rule image-add-int-atLeastLessThan)
done

lemma card-greaterThanAtMost-int [simp]: card {l<..<u} = nat (u - (l + 1))
  by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

lemma card-greaterThanLessThan-int [simp]: card {l<..<u} = nat (u - (l + 1))
  by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

lemma finite-M-bounded-by-nat: finite {k. P k ∧ k < (i::nat)}
proof
  have \{k. P k ∧ k < i\} ⊆ {..<i} by auto
  with finite-lessThan[of i] show ?thesis by (simp add: finite-subset)
qed

lemma card-less:
assumes zero-in-M: 0 ∈ M
shows card {k ∈ M. k < Suc i} ≠ 0
proof –
  from zero-in-M have {k ∈ M. k < Suc i} ≠ {} by auto
qed

lemma card-less-Suc2: 0 /∈ M ⇒ card {k. Suc k ∈ M ∧ k < i} = card {k ∈ M. k < Suc i}
apply (rule card-bij-eq [of Suc - - λx. x − Suc 0])
apply auto
apply (rule inj-on-diff-nat)
apply auto
apply (case-tac x)
apply auto
apply (case-tac xa)
apply auto
apply (case-tac xa)
apply auto
done

lemma card-less-Suc:
assumes zero-in-M: 0 ∈ M
  shows Suc (card {k. Suc k ∈ M ∧ k < i}) = card {k ∈ M. k < Suc i}
proof –
  from assms have a: 0 ∈ {k ∈ M. k < Suc i} by simp
  hence c: {k ∈ M. k < Suc i} = insert 0 ({{k ∈ M. k < Suc i} − {0}})
    by (auto simp only: insert-Diff)
  have b: {k ∈ M. k < Suc i} − {0} = {k ∈ M − {0}. k < Suc i} by auto
from finite-M-bounded-by-nat[of λx. x ∈ M Suc i]
have Suc (card {k. Suc k ∈ M ∧ k < i}) = card (insert 0 ({{k ∈ M. k < Suc i} − {0}}))
  apply (subst card-insert)
  apply simp-all
  apply (subst b)
  apply (subst card-less-Suc2[symmetric])
  apply simp-all
done
with c show ?thesis by simp
qed

59.7 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy
59.7.1 Disjoint Unions

Singletons and open intervals

lemma ivl-disj-un-singleton:
{\{l::'a::linorder\} Un \{l<..\} = \{l..\}\}
{\{..<u\} Un \{u::'a::linorder\} = \{..u\}\}
(t::'a::linorder) < u ==> {l} Un \{l<..<u\} = \{l..<u\}\)
(t::'a::linorder) < u ==> {l..<u} Un \{u\} = \{l..<u\}\)
(t::'a::linorder) <= u ==> {l} Un \{l..<u\} = \{l..u\}\)
(t::'a::linorder) <= u ==> \{l..<u\} Un \{u\} = \{l..<u\}\)
by auto

One- and two-sided intervals

lemma ivl-disj-un-one:
\begin{align*}
(t::'a::linorder) < u ==> \{..l\} Un \{l<..<u\} = \{..<u\}\)
(t::'a::linorder) <= u ==> \{..<l\} Un \{l..<u\} = \{..<u\}\)
(t::'a::linorder) <= u ==> \{..l\} Un \{l..<u\} = \{..<u\}\)
(t::'a::linorder) <= u ==> \{l..<u\} Un \{u..\} = \{l..<u\}\)
(t::'a::linorder) <= u ==> \{l..<u\} Un \{u..<\} = \{l..<\}\)
by auto
\end{align*}

Two- and two-sided intervals

lemma ivl-disj-un-two:
\begin{align*}
&\{1::'a::linorder\} \langle m; m <= u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
&\{1::'a::linorder\} \langle m; m < u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
&\{1::'a::linorder\} \langle m; m <= u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
&\{1::'a::linorder\} \langle m; m < u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
&\{1::'a::linorder\} \langle m; m <= u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
&\{1::'a::linorder\} \langle m; m < u \rangle == \{l<..<m\} Un \{m..<u\} = \{l..<u\}\)
by auto
\end{align*}

lemmas ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two

59.7.2 Disjoint Intersections

One- and two-sided intervals

lemma ivl-disj-int-one:
\begin{align*}
\{..l::'a::order\} Int \{l<..<u\} = \{}\)
\{..<l\} Int \{l..<u\} = \{}\)
\{..\} Int \{l..<u\} = \{}\)
\{l..<u\} Int \{u..<\} = \{}\)
\{l<..<u\} Int \{u..\} = \{}\)
\end{align*}
\{l..u\} Int \{u..<\} = \{\}
\{l..<u\} Int \{u..\} = \{\}
by auto

Two- and two-sided intervals

\textbf{lemma} \textit{ivl-disj-int-two}:
\{l::a:order..<\ <m\} Int \{m..<u\} = \{\}
\{l..<m\} Int \{m..<u\} = \{\}
\{l..<m\} Int \{m..<u\} = \{\}
\{l..<u\} Int \{m..<u\} = \{\}
\{l..<m\} Int \{m..<u\} = \{\}
\{l..<m\} Int \{m..<u\} = \{\}
\{l..<m\} Int \{m..<u\} = \{\}
by auto

\textbf{lemmas} \textit{ivl-disj-int} = \textit{ivl-disj-int-one} \textit{ivl-disj-int-two}

\textbf{59.7.3 Some Differences}

\textbf{lemma} \textit{ivl-diff[simp]}:
i \leq n \implies \{i..<m\} - \{i..<n\} = \{n..<(m::a::linorder)\}
\textbf{by (auto)}

\textbf{lemma} \textbf{(in linorder)} \textit{lessThan-minus-lessThan} [simp]:
\{..<n\} - \{..<m\} = \{m..<n\}
\textbf{by auto}

\textbf{59.7.4 Some Subset Conditions}

\textbf{lemma} \textit{ivl-subset [simp]}:
\{(i..<j) \subseteq \{m..<n\}\} = \{j \leq i \mid m \leq i \& j \leq (n::a::linorder)\}
\textbf{apply (auto simp:linorder-not-le)}
\textbf{apply (rule \textit{contr})}
\textbf{apply (insert \textit{linorder-le-less-linear[of i n]})}
\textbf{apply (clarsimp simp:linorder-not-le)}
\textbf{apply (fastforce)}
done

\textbf{59.8 Summation indexed over intervals}

\textbf{syntax}
- \textbf{from-to-setsum} :: \textit{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\textit{SUM} - = -.-/-) [0,0,0,10] 10)
- \textbf{from-upto-setsum} :: \textit{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\textit{SUM} - = -.-/-) [0,0,0,10] 10)
- \textbf{apt-setsum} :: \textit{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\textit{SUM} - = -.-/-) [0,0,10] 10)
- \textbf{upto-setsum} :: \textit{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\textit{SUM} - = -.-/-) [0,0,10] 10)
\textbf{syntax (\textit{symbols})}
- \textbf{from-to-setsum} :: \textit{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\textit{SUM} - = -.-/-) [0,0,0,10] 10)
Note that for uniformity on \( \text{nat} \) it only works well with italic-style formulae, not tt-style. Also in the special form for \( \sum \) than be activated explicitly by setting the print mode to a special syntax. The latter is only meaningful for latex output and has to be introduced some pretty alternative syntaxes for summation over intervals:

\[
\begin{align*}
\text{Old} & \quad \text{New} & \quad \text{I\TeX} \\
\sum x \in \{ a..b \}. e & \quad \sum x = a..b. e & \quad \sum_{x=a}^{b} e \\
\sum x \in \{ a..<b \}. e & \quad \sum x = a..<b. e & \quad \sum_{x=a}^{b} e \\
\sum x \in \{.b \}. e & \quad \sum x \leq b. e & \quad \sum_{x} \leq b e \\
\sum x \in \{.<b \}. e & \quad \sum x < b. e & \quad \sum_{x} < b e
\end{align*}
\]

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to \textit{latex-sum} (e.g. via \textit{mode = latex-sum} in antiquotations). It is not the default \textit{I\TeX} output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on \( \text{nat} \) it is better to use \( \sum x = 0..<n. e \) rather than \( \sum x < n. e \): \( \text{setsum} \) may not provide all lemmas available for \( \{ m..<n \} \) also in the special form for \( \{.n \} \).

This congruence rule should be used for sums over intervals as the standard theorem \textit{setsum.cong} does not work well with the simplifier who adds the unsimplified premise \( x \in B \) to the context.
THEORY “Set-Interval”

lemma setsum-ivl-cong:
\[ a = c; \, b = d; \, \forall x. \, \{ c \leq x; \, x < d \} \implies f x = g x \implies \]
\[ \text{setsum } f \{ a..<b \} = \text{setsum } g \{ c..<d \} \]
by (rule setsum.cong, simp-all)

lemma setsum-atMost-Suc [simp]:
\[ (\sum i \leq \text{Suc } n . \, f i) = (\sum i \leq n . \, f i) + f (\text{Suc } n) \]
by (simp add: atMostSuc ac-simps)

lemma setsum-lessThan-Suc [simp]:
\[ (\sum i < \text{Suc } n . \, f i) = (\sum i < n . \, f i) + f n \]
by (simp add: lessThanSuc ac-simps)

lemma setsum-cl-ivl-Suc [simp]:
\[ \text{setsum } f \{ m..\text{Suc } n \} = (\text{if } \text{Suc } n < m \text{ then } 0 \text{ else } \text{setsum } f \{ m..n \} + f (\text{Suc } n)) \]
by (auto simp: ac-simps atLeastAtMostSuc-conv)

lemma setsum-op-ivl-Suc [simp]:
\[ \text{setsum } f \{ m..<\text{Suc } n \} = (\text{if } n < m \text{ then } 0 \text{ else } \text{setsum } f \{ m..<n \} + f n) \]
by (auto simp: ac-simps atLeastLessThanSuc)

lemma setsum-head:
\[ \text{fixes } n :: \text{nat} \]
\[ \text{assumes } mn: \, m \leq n \]
\[ \text{shows } (\sum x \in \{ m..n \}. \, P x) = P m + (\sum x \in \{ m..<n \}. \, P x) \text{ (is } ?lhs = ?rhs) \]
proof –
\[ \text{from } mn \]
\[ \text{have } \{ m..n \} = \{ m \} \cup \{ m..<n \} \]
\[ \text{by (auto intro: iel-disj-un-singleton)} \]
\[ \text{hence } ?lhs = (\sum x \in \{ m \} \cup \{ m..<n \}. \, P x) \]
\[ \text{by (simp add: atLeast0LessThan)} \]
\[ \text{also have } \ldots = ?rhs \text{ by simp} \]
\[ \text{finally show } ?thesis . \]
qed

lemma setsum-head-Suc:
\[ m \leq n \implies \text{setsum } f \{ m..n \} = f m + \text{setsum } f \{ \text{Suc } m..n \} \]
by (simp add: setsum-head atLeastSucAtMost-greaterThanAtMost)

lemma setsum-head-upt-Suc:
\[ m < n \implies \text{setsum } f \{ m..<n \} = f m + \text{setsum } f \{ \text{Suc } m..<n \} \]
aply (insert setsum-head-Suc[of m n - Suc 0 f])
aply (simp add: atLeastLessThanSuc-atLeastAtMost[symmetric] algebra-simps)
done

lemma setsum-ab-add-nat: \[ \text{assumes } (m::nat) \leq n + 1 \]
\[ \text{shows } \text{setsum } f \{ m..n + p \} = \text{setsum } f \{ m..n \} + \text{setsum } f \{ n + 1..n + p \} \]
proof –
have \( \{ m .. n+p \} = \{ m .. n \} \cup \{ n+1 .. n+p \} \) using \( m \leq n+1 \) by auto

thus \( \text{thesis} \) by (auto simp: ivl-disj-int setsum.union-disjoint atLeastSucAtMost-greaterThanAtMost)

qed

lemma setsum-add-nat-ivl: \[ m \leq n; \ n \leq p \] \[ \rightarrow \]
setsum \( f \) \( \{ m ..< n \} \) + setsum \( f \) \( \{ n ..< p \} \) = setsum \( f \) \( \{ m ..< p :: \text{nat} \} \)

by (simp add: setsum.union-disjoint [symmetric] ivl-disj-int ivl-disj-un)

lemma setsum-diff-nat-ivl:
fixes \( f :: \text{nat} \Rightarrow \) \( 'a::\text{ab-group-add} \)
shows \[ m \leq n; \ n \leq p \] \[ \rightarrow \]
setsum \( f \) \( \{ m ..< p \} \) − setsum \( f \) \( \{ m ..< n \} \) = setsum \( f \) \( \{ n ..< p \} \)

using setsum-add-nat-ivl [of \( m \ n \ p \ f \) , symmetric]
apply (simp add: ac-simps)
done

lemma setsum-shift-bounds-cl-nat-ivl:
setsum \( f \) \( \{ m+k .. n+k \} \) = setsum \( \lambda i. \ f (i+k) \) \( \{ m ..< n+k \} \)

by (rule setsum.reindex-bij-witness [where \( i = \lambda i. i + k \) and \( j = \lambda i. i - k \)])

59.9 Shifting bounds

lemma setsum-shift-bounds-cl-Suc-ivl:
setsum \( f \) \( \{ \text{Suc } m ..\text{Suc } n \} \) = setsum \( \lambda i. \ f (i+k) \) \( \{ m ..< n \} \)

by (simp add: setsum-shift-bounds-cl-nat-ivl [where \( k = \text{Suc } 0 \), simplified])

corollary setsum-shift-bounds-Suc-ivl:
setsum \( f \) \( \{ \text{Suc } m ..< \text{Suc } n \} \) = setsum \( \lambda i. \ f (i+k) \) \( \{ m ..< n \} \)

by (simp add: setsum-shift-bounds-cl-nat-ivl [where \( k = \text{Suc } 0 \), simplified])

lemma setsum-shift-lb-Suc0-0:
f(0::nat) = (0::nat) → setsum f {Suc 0..k} = setsum f {0..k}
by(simp add:setsum-head-Suc)

lemma setsum-shift-lb-Suc0-0-upt:
f(0::nat) = 0 ⇒ setsum f {Suc 0..<k} = setsum f {0..<k}
apply(cases k)apply simp
apply(simp add:setsum-head-upt-Suc)
done

lemma setsum-atMost-Suc-shift:
  fixes f :: nat ⇒ 'a::comm-monoid-add
  shows (∑ i≤Suc n. f i) = f 0 + (∑ i≤n. f (Suc i))
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n) note IH = this
  have (∑ i≤Suc (Suc n). f i) = (∑ i≤Suc n. f i) + f (Suc (Suc n))
    by (rule setsum-atMost-Suc)
  also have (∑ i≤Suc n. f i) = f 0 + (∑ i≤n. f (Suc i))
    by (rule IH)
  also have f 0 + (∑ i≤n. f (Suc i)) + f (Suc (Suc n)) =
    f 0 + ((∑ i≤n. f (Suc i)) + f (Suc (Suc n)))
    by (rule add.assoc)
  also have (∑ i≤n. f (Suc i)) + f (Suc (Suc n)) = (∑ i≤Suc n. f (Suc i))
    by (rule setsum-atMost-Suc [symmetric])
  finally show ?case .
qed

lemma setsum-last-plus: fixes n::nat shows m <= n ⇒ (∑ i = m..<n. f i) = f m
  + (∑ i = m..<n. f i)
  by (cases n) (auto simp: atLeastLessThanSuc-atLeastAtMost add.commute)

lemma setsum-Suc-diff:
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ Suc n
  shows (∑ i = m..<n. f (Suc i) − f i) = f (Suc n) − f m
using assms by (induct n) (auto simp: le-Suc-eq)

lemma nested-setsum-swap:
  (∑ i = 0..n. (∑ j = 0..<i. a i j)) = (∑ j = 0..<n. ∑ i = Suc j..n. a i j)
  by (induction n) (auto simp: setsum.distrib)

lemma nested-setsum-swap':
  (∑ i≤n. ∑ j<i. a i j) = (∑ j<n. ∑ i = Suc j..n. a i j)
  by (induction n) (auto simp: setsum.distrib)

lemma setsum-zero-power [simp]:
  fixes c :: nat ⇒ 'a::division-ring
  shows (∑ i∈A. c i ∗ 0^i) = (if finite A ∧ 0 ∈ A then c 0 else 0)
apply (cases finite A)
by (induction A rule: finite-induct) auto

lemma setsum-zero-power' [simp]:
fixes c :: nat ⇒ 'a::field
shows (\sum i \in A. c i * 0^i / d i) = (if finite A ∧ 0 \in A then c 0 / d 0 else 0)
using setsum-zero-power [of λi. c i / d i A]
by auto

59.10 The formula for geometric sums

lemma geometric-sum:
assumes x ≠ 1
shows (\sum i < n. x ^ i) = (x ^ n - 1) / (x - 1::'a::field)
proof –
from assms obtain y where y = x - 1 and y ≠ 0 by simp-all
moreover have (\sum i < n. (y + 1) ^ i) = ((y + 1) ^ n - 1) / y
by (induct n) (simp-all add: field-simps :y ≠ 0)
ultimately show ?thesis by simp
qed

59.11 The formula for arithmetic sums

lemma gauss-sum:
(2::'a::comm-semiring-1) * (∑ i ∈ {1..<n}. of-nat i) = of-nat n*((of-nat n)+1)
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  then show ?case
    by (simp add: algebra-simps add: one-add-one [symmetric] del: one-add-one)
qed

theorem arith-series-general:
(2::'a::comm-semiring-1) * (∑ i ∈ {..<n}. a + of-nat i * d) =
of-nat n * (a + (a + of-nat(n - 1)*d))
proof cases
  assume ngt1: n > 1
  let \?I = λi. of-nat i and \?n = of-nat n
  have (∑ i ∈ {..<n}. a + \?I i*d) =
       (∑ i ∈ {..<n}. a) + (∑ i ∈ {..<n}. \?I i*d)
  by (rule setsum.distrib)
  also from ngt1 have ... = \?n*a + (∑ i ∈ {..<n}. \?I i*d) by simp
  also from ngt1 have ... = (\?n*a + \?d*(∑ i ∈ {1..<n}. \?I i))
    unfolding One-nat-def
  by (simp add: setsum-right-distrib atLeast0LessThan[symmetric] setsum-shift-lb-Suc0-0-upt ac-simps)
also have $2\times\ldots = 2\times\{1\ldots<n\} = 2\times\{1\ldots<n\} - 1$
by (simp add: algebra-simps)
also from ngt1 have $\{1\ldots<n\} = \{1\ldots<n\}$
by (cases n) (auto simp: atLeastLessThanSuc-atLeastAtMost)
also from ngt1 have $2\times\{1\ldots<n\} = 2\times\{1\ldots<n\}$
by (simp only: mult.assoc gauss-sum [of n - 1], unfold One-nat-def)
finally show ?thesis
unfolding mult-2 by (simp add: algebra-simps)
next
assume $\neg(n > 1)$
hence $n = 1 \lor n = 0$ by auto
thus ?thesis by (auto simp: mult-2)
qed

lemma arith-series-nat:
$(2\cdot\text{nat}) \times (\sum i \in \{1\ldots<n\}. a+i\cdot d) = n \times \{a\times(n+1)\times d\}$

proof
have $2\times\{1\ldots<n\} = 2\times\{1\ldots<n\}$
by (rule arith-series-general)
thus ?thesis
unfolding One-nat-def by auto
qed

lemma arith-series-int:
$2\times(\sum i \in \{1\ldots<n\}. a+\text{int}i \times d) = \text{int} n \times \{a+\text{int}(n+1)\times d\}$
by (fact arith-series-general)

lemma sum-diff-distrib:
fixes $P :: \text{nat} \Rightarrow \text{nat}$
shows $(\forall x. Q \leq P x \implies (\sum x\in\text{<n}. P x) - (\sum x\in\text{<n}. Q x) = (\sum x\in\text{<n}. P x - Q x))$
proof (induct n)
case 0 show ?case by simp
next
case (Suc n)

let ?lhs = $(\sum x\in\text{<n}. P x) - (\sum x\in\text{<n}. Q x)$
let ?rhs = $\sum x\in\text{<n}. P x - Q x$
from Suc have ?lhs = ?rhs by simp
moreover
from Suc have ?lhs + P n - Q n = ?rhs + (P n - Q n) by simp
moreover
from Suc have
THEORY "Set-Interval"

\[(\sum_{x<n. \ P \ x} + P n) - ((\sum_{x<n. \ Q \ x} + Q n) = ?rhs + (P n - Q n)\]

by (subt diff-diff-left[ symmetric],
    subst diff-add-assoc2)

(auto simp: diff-add-assoc2 intro: setsum-mono)

ultimately

show ?case by simp

qed

lemma nat-diff-setsum-reindex: \[(\sum_{i<n. \ f \ (n - Suc \ i)} = \sum_{i<n. \ f \ i})\]

by (rule setsum.reindex-bij-witness[where \ i=\lambda i. \ n - Suc \ i \ and \ j=\lambda i. \ n - Suc \ i])

59.12 Products indexed over intervals

syntax

(from-to-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((PROD = -../-) [0,0,0,10] 10)
        from-upto-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((PROD = -..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((PROD = <..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((PROD = <-.</-) [0,0,0,10] 10)

syntax (xsymbols)

(from-to-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -../-) [0,0,0,10] 10)
        from-upto-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <-.</-) [0,0,0,10] 10)

syntax (HTML output)

(from-to-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -../-) [0,0,0,10] 10)
        from-upto-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <-.</-) [0,0,0,10] 10)

syntax (latex-prod output)

(from-to-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -../-) [0,0,0,10] 10)
        from-upto-setprod :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = -..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <..</-) [0,0,0,10] 10)
        apt-setprod :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\Pi = <-.</-) [0,0,0,10] 10)

translations

\[(\prod_{x=a..b. \ t} \ = \ CONST \ setprod \ (%x. \ t) \ \{a..b\}\]
\[(\prod_{x=a..<b. \ t} \ = \ CONST \ setprod \ (%x. \ t) \ \{a..<b\}\]
\[(\prod_{i\leq n. \ t} \ = \ CONST \ setprod \ (%i. \ t) \ \{..n\}\]
\[(\prod_{i<n. \ t} \ = \ CONST \ setprod \ (%i. \ t) \ \{..<n\}\]
section "Set-Interval">

lemma transfer-nat-int-set-functions:
\{\ldots n\} = nat \{\ldots \text{int } n\}
\{m..n\} = \text{int } \{m..\text{int } n\}
apply (auto simp add: image-def)
apply (rule-tac x = int x in bexI)
apply auto
apply (rule-tac x = int x in bexI)
apply auto
done

lemma transfer-nat-int-set-function-closures:
x >= 0 \Rightarrow \text{nat-set } \{x..y\}
by (simp add: nat-set-def)

declare transfer-morphism-nat-int[transfer add
return: transfer-nat-int-set-functions
transfer-nat-int-set-function-closures
]

lemma transfer-int-nat-set-functions:
is-nat m \Rightarrow is-nat n \Rightarrow \{m..n\} = \text{int } \{\text{nat } m..\text{nat } n\}
by (simp only: is-nat-def transfer-nat-int-set-functions
transfer-nat-int-set-function-closures
transfer-nat-int-set-return-embed nat-0-le
cong: transfer-nat-int-set-cong)

lemma transfer-int-nat-set-function-closures:
is-nat x \Rightarrow \text{nat-set } \{x..y\}
by (simp only: transfer-nat-int-set-function-closures is-nat-def)

declare transfer-morphism-int-nat[transfer add
return: transfer-int-nat-set-functions
transfer-int-nat-set-function-closures
]

lemma setprod-int-plus-eq: setprod int \{i..i+j\} = \prod \{\text{int } i..\text{int } (i+j)\}
by (induct j) (auto simp add: atLeastAtMostSuc-conv atLeastAtMostPlus1-int-conv)

lemma setprod-int-eq: setprod int \{i..j\} = \prod \{\text{int } i..\text{int } j\}
proof (cases i \leq j)
case True
then show ?thesis
by (metis Nat.le-iff-add setprod-int-plus-eq)
next
case False
then show ?thesis
by auto
qed
60 Presburger: Decision Procedure for Presburger Arithmetic

theory Presburger
imports Groebner-Basis Set-Interval
begin

ML-file Tools/Qelim/gelim.ML
ML-file Tools/Qelim/cooper-procedure.ML

60.1 The $-\infty$ and $+\infty$ Properties

lemma minf:
\[ \exists (z :: linorder).orall x < z. P x = P' x \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (P x \land Q x) = (P' x \land Q' x) \]
\[ \exists (z :: linorder).orall x < z. P x = P' x \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists z' \forall x < z. (P x \lor Q x) = (P' x \lor Q' x) \]
\[ \exists (z :: linorder).orall x < z. (x = t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (x \neq t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (x < t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (x \leq t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (x > t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x < z. (x \geq t) \iff \exists z' \forall x < z'. Q x = Q' x \]
\[ \exists z \forall (x :: b :: linorder, plus, Rings.ded) \forall x < z. (d dvd x + s) = (d dvd x + s) \]
\[ \exists z \forall (x :: b :: linorder, plus, Rings.ded) \forall x < z. (\neg d dvd x + s) = (\neg d dvd x + s) \]
\[ \exists z \forall x < z. F = F \]
by ((erule exE, erule exE, rule-tac x=min z za in exI,simp)+, (rule-tac x=t in exI,fastforce)+) simp-all

lemma pinf:
\[ \exists (z :: linorder).orall x > z. P x = P' x \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (P x \land Q x) = (P' x \land Q' x) \]
\[ \exists (z :: linorder).orall x > z. P x = P' x \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists z' \forall x > z. (P x \lor Q x) = (P' x \lor Q' x) \]
\[ \exists (z :: linorder).orall x > z. (x = t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (x \neq t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (x < t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (x \leq t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (x > t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists (z :: linorder).orall x > z. (x \geq t) \iff \exists z' \forall x > z'. Q x = Q' x \]
\[ \exists z \forall (x :: b :: linorder, plus, Rings.ded) \forall x > z. (d dvd x + s) = (d dvd x + s) \]
\[ \exists z \forall (x :: b :: linorder, plus, Rings.ded) \forall x > z. (\neg d dvd x + s) = (\neg d dvd x + s) \]
\[ \exists z \forall x > z. F = F \]
by ((erule exE, erule exE, rule-tac x=max z za in exI,simp)+, (rule-tac x=t in exI,fastforce)+) simp-all
lemma inf-period:

\[ \forall x. k. P x = P (x - k*D); \forall x. k. Q x = Q (x - k*D) \]
\[ \implies \forall x. k. (P x \land Q x) = (P (x - k*D) \land Q (x - k*D)) \]
\[ \forall x. k. P x = P (x - k*D); \forall x. k. Q x = Q (x - k*D) \]
\[ \implies \forall x. k. (P x \lor Q x) = (P (x - k*D) \lor Q (x - k*D)) \]
\[(d::a::\{comm-ring,Rings,dvd\}) \implies \forall x. k. (d dvd x + t) = (d dvd (x - k*D) + t) \]
\[(d::a::\{comm-ring,Rings,dvd\}) \implies \forall x. k. (\neg d dvd x + t) = (\neg d dvd (x - k*D) + t) \]

apply (auto elim!: dvdE simp add: algebra-simps)

unfolding mult.assoc [symmetric] distrib-right [symmetric] left-diff-distrib [symmetric]

unfolding dvd-def mult.commute [atf d]

by auto

60.2 The A and B sets

lemma bset:

\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow P x \rightarrow P(x - D); \]
\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow Q x \rightarrow Q(x - D)) \]
\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (P x \land Q x) \rightarrow (P(x - D) \land Q (x - D)) \]
\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (P x \lor Q x) \rightarrow (P(x - D) \lor Q (x - D)) \]

\[ [D>0; t - 1 \in B] \implies (\forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x \neq t) \longrightarrow (x - D = t)) \]
\[ [D>0 \land t \in B] \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \]
\[ D>0 \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x < t) \longrightarrow (x - D < t)) \]
\[ D>0 \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)) \]
\[ [D>0 \land t \in B] \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x > t) \longrightarrow (x - D > t)) \]
\[ [D>0 \land t - 1 \in B] \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)) \]
\[ d dvd D \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (d dvd x + t) \longrightarrow (d dvd (x - D) + t)) \]
\[ d dvd D \implies (\forall (x:int). (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (\neg d dvd x + t) \longrightarrow (\neg d dvd (x - D) + t)) \]
\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (\neg F) \longrightarrow F \]

proof (blast, blast)

assume dp: D > 0 and tB: t - 1 \in B

show (\forall x. (\forall j \in \{1 \ldots D\}. \forall b\in B. x \neq b + j \longrightarrow (x = t) \longrightarrow (x - D = t))

apply (rule allI, rule impI, rule ballE[where x=1], rule ballE[where x=t-1])
apply algebra using dp tB by simp-all
next
assume dp: D > 0 and tB: t \in B
show (\forall x.(\forall j \in \{1 .. D\}, \forall b \in B. x \neq b + j) \implies (x \neq t) \implies (x - D \neq t))
  apply (rule allI; rule impI;erule ballE[where x=D];erule ballE[where x=t])
  apply algebra
  using dp tB by simp-all
next
assume dp: D > 0 thus (\forall x.(\forall j \in \{1 .. D\}, \forall b \in B. x \neq b + j) \implies (x < t) \implies (x - D < t)) by arith
next
assume dp: D > 0 thus (\forall x.(\forall j \in \{1 .. D\}, \forall b \in B. x \neq b + j) \implies (x \leq t) \implies (x - D \leq t)) by arith
next
assume dp: D > 0 and tB:t \in B
\{ fix x assume nob: \forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j and g: x > t and ng: \neg (x - D) > t
  hence x - t \leq D and 1 \leq x - t by simp+
  hence \exists j \in \{1 .. D\}. x - t = j by auto
  hence \exists j \in \{1 .. D\}. x = t + j by (simp add: algebra-simps)
  with nob tB have False by simp\}
thus (\forall x.\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x > t) \implies (x - D > t) by blast
next
assume dp: D > 0 and tB:t - 1 \in B
\{ fix x assume nob: \forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j and g: x \geq t and ng: \neg (x - D) \geq t
  hence x - (t - 1) \leq D and 1 \leq x - (t - 1) by simp+
  hence \exists j \in \{1 .. D\}. x - (t - 1) = j by auto
  hence \exists j \in \{1 .. D\}. x = (t - 1) + j by (simp add: algebra-simps)
  with nob tB have False by simp\}
thus (\forall x.\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (x \geq t) \implies (x - D \geq t) by blast
next
assume d: d dvd D
\{ fix x assume H: d dvd x + t with d have d dvd (x - D) + t by algebra\}
thus (\forall (x::int).\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (d dvd x+t) \implies (d dvd (x - D) + t) by simp
next
assume d: d dvd D
\{ fix x assume H: \neg(d dvd x + t) with d have \neg d dvd (x - D) + t
  by (clarsimp simp add: dvd-def,erule_tac x= ka + k in allE,simp add: algebra-simps)\}
thus (\forall (x::int).\forall j \in \{1 .. D\}. \forall b \in B. x \neq b + j) \implies (\neg d dvd x+t) \implies (\neg d dvd (x - D) + t) by auto
qed blast

lemma aset:
  [\forall x.\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j] \implies P x \implies P(x + D);
  \forall x.\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j \implies Q x \implies Q(x + D)] \implies
\forall x.\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j \implies (P x \land Q x) \implies (P(x + D) \land Q (x
\[ \forall d. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow P x \rightarrow P(x + D) \]

\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow Q x \rightarrow Q(x + D) \]

\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (P x \lor Q x) \rightarrow (P(x + D) \lor Q(x + D)) \]

\[ [D > 0; t + 1 \in A] \implies (\forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D = t)) \]

\[ [D > 0; t \in A] \implies (\forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x \neq t) \rightarrow (x + D \neq t)) \]

\[ [D > 0; t + 1 \in A] \implies (\forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x < t) \rightarrow (x + D < t)) \]

\[ [D > 0; t + 1 \in A] \implies (\forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x > t) \rightarrow (x + D > t)) \]

\[ (\forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x \geq t) \rightarrow (x + D \geq t)) \]

\[ (d dvd D \implies \forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (d dvd x + t) \rightarrow (d dvd x)) \]

\[ (d dvd D \implies \forall (x::int). (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (¬ d dvd x + t) \rightarrow (¬ d dvd x + t)) \]

\[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow F \rightarrow F \]

**proof** (blast, blast)

assume dp: D > 0 and ta: t + 1 \in A

show (\forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D = t))

apply (rule allI, rule impI, rmultE where x=t+1, rmultE[where x=t+1])

using dp ta by simp-all

next

assume dp: D > 0 and ta: t \in A

show (\forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x \neq t) \rightarrow (x + D \neq t))

apply (rule allI, rule impI, rmultE where x=D, rmultE[where x=t])

using dp ta by simp-all

next

assume dp: D > 0 thus (\forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x > t) \rightarrow (x + D > t)) by arith

next

assume dp: D > 0 thus (\forall x. (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x \geq t) \rightarrow (x + D \geq t)) by arith

next

assume dp: D > 0 and ta: t \in A

\{fix x assume nob: \forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j and g: x < t and ng: ¬ (x + D) < t

hence t - x \leq D and I = t - x \leq D by simp+

hence \exists j \in \{1 \ldots D\}. t - x = j by auto

hence \exists j \in \{1 \ldots D\}. x = t - j by (auto simp add: algebra-simps)

with nob ta have False by simp\}

thus (\forall j \in \{1 \ldots D\}. \forall b \in A. x \neq b - j) \rightarrow (x < t) \rightarrow (x + D < t) by blast
60.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version

60.3.1 First some trivial facts about periodic sets or predicates

lemma periodic-finite-ex:

assumes dpos: $(0::\text{int}) < d$ and modd: $\text{ALL} \ x \ k. \ P \ x = P(x - k*d)$

shows $(\exists x. \ P x) = (\exists x : \{1..d\}. \ P x)$

(is ?LHS = ?RHS)

proof

assume ?LHS

then obtain $x$ where \( P : P x \) ..

have $x \mod d = x - (x \div d)*d$ by (simp add: zmod-zdiv-equality ac-simps eq-diff-eq)

hence $P \mod: P x = P(x \mod d)$ using modd by simp

show ?RHS

proof (cases)

assume $x \mod d = 0$

hence $P 0$ using $P \mod$ by simp

moreover have $P 0 = P(0 - (-1)*d)$ using modd by blast

ultimately have $P d$ by simp

moreover have $d : \{1..d\}$ using dpos by simp

ultimately show ?RHS ..

next

assume not0: $x \mod d \neq 0$

have $P(x \mod d)$ using dpos $P \mod$ by simp

moreover have $x \mod d : \{1..d\}$
proof
from dpos have \(0 \leq x \mod d\) by (rule pos-mod-sign)
moreover from dpos have \(x \mod d < d\) by (rule pos-mod-bound)
ultimately show \(?thesis\) using not0 by simp
qed
ultimately show \(?RHS\) ..
qed
qed auto

60.3.2 The \(-\infty\) Version

lemma decre-lemma: \(0 < (d::int) \Longrightarrow x - (\text{abs}(x-z)+1) * d < z\)
by (induct rule: int-gr-induct, simp-all add: int-distrib)

lemma incr-lemma: \(0 < (d::int) \Longrightarrow z < x + (\text{abs}(x-z)+1) * d\)
by (induct rule: int-gr-induct, simp-all add: int-distrib)

lemma decre-mult-lemma:
  assumes dpos: \((0::int) < d\) and minus: \(\forall x. P x \rightarrow P(x - d)\) and knneg: \(0 < k\)
  shows \(\forall x. P x \rightarrow P(x - k*d)\)
using knneg
proof (induct rule: int-ge-induct)
case base thus \(?case\) by simp
next
case (step i)
  { fix x
    have \(P x \rightarrow P(x - i * d)\) using step.hyps by blast
    also have \(\ldots \rightarrow P(x - (i + 1) * d)\) using minus[THEN spec, of x - i * d]
    by (simp add: algebra-simps)
    ultimately have \(P x \rightarrow P(x - (i + 1) * d)\) by blast
  }
thus \(?case\) ..
qed

lemma minusinfinity:
  assumes dpos: \(0 < d\) and
  \(P1eqP1:: ALL x k. P1 x = P1(x - k*d)\) and \(cPeqP1:: \exists z::\text{int}. \forall x. x < z \rightarrow (P x = P1 x)\)
  shows \(\exists x. P1 x \rightarrow (\exists x. P x)\)
proof
  assume eP1: \(\exists x. P1 x\)
  then obtain \(x\) where P1: \(P1 x\) ..
  from cPeqP1 obtain \(z\) where P1eqP: \(\forall x. x < z \rightarrow (P x = P1 x)\) ..
  let \(?w = x - (\text{abs}(x-z)+1) * d\)
  from dpos have \(?w < z\) by (rule decre-lemma)
  have \(P1 x = P1 \ ?w\) using P1eqP1 by blast
  also have \(\ldots = P(?w)\) using \(?w\) P1eqP by blast
  finally have \(P \ ?w\) using \(P1\) by blast
  thus \(\exists x. P x\) ..
qed

lemma cpmi:
  assumes dp: \(0 < D\) and p1: \(\exists z. \forall x < z. P x = P' x\)
  and nb/\(\forall x. (\forall j \in \{1..D\}, \forall (b::\text{int}) \in B. x \neq b+j)\) \(-\rightarrow\) \(P (x) \rightarrow P (x - D)\)
  and pd: \(\forall x k. P' x = P' (x - k*D)\)
  shows \((\exists x. P x) = ((\exists j \in \{1..D\}. P' j) | (\exists j \in \{1..D\}. \exists b \in B. P (b+j)))\)
  (is \(?L = (?R1 \lor ?R2)\))
proof
  {assume \(?R2\) hence \(?L\) by blast}
moreover
  {assume \(?R1\) hence \(?L\) using minusinfinity[OF dp pd p1] periodic-finite-ex[OF dp pd] by simp}
moreover
  {fix x
    assume P: \(P x\) and H: \(\neg ?R2\)
    {fix y assume \(\neg (\exists j \in \{1..D\}. \exists b \in B. P (b + j))\) and P: \(P y\)
      hence \(\neg (EX (j::\text{int}) : \{1..D\}. EX (b::\text{int}) : B. y = b+j)\) by auto
      with \(\neg b\ P\ have P (y - D)\) by auto }
    hence \(\forall x. \neg (EX (j::\text{int}) : \{1..D\}. EX (b::\text{int}) : B. P (b+j))\) \(-\rightarrow\) \(P (x) \rightarrow P (x - D)\) by blast
      with \(H\ P\ have th: \(\forall x. P x \rightarrow P (x - D)\) by auto
    from p1 obtain z where z: \(\forall x. \exists z. \forall x < z \rightarrow (P x = P' x)\) by blast
    let \(?y = x - ((x - z) + 1)*D\)
    have \(zp: \emptyset \leq (\exists j \in \{1..D\}. \exists b \in B. P (b + j))\) by arith
    from dp have \(yz: \emptyset \leq (\exists j \in \{1..D\}. \exists b \in B. P (b + j))\) by simp
    from \(z[\text{rule-format}, OF yz]\) \(decr-mult-lemma[OF dp th zp, rule-format, OF P]\) have \(th2: \(P' \emptyset\)\) by auto
      with periodic-finite-ex[OF dp pd] have \(?R1\) by blast}
ultimately show \(?\)thesis by blast
qed

60.3.3 The \(+\infty\) Version

lemma plusinfinity:
  assumes dp: \((0::\text{int}) < d\) and 
  \(P1eqP1: \forall x k. P' x = P(x - k*d)\) and \(ePeqP1: \exists z. \forall x > z. P x = P' x\)
  shows \((\exists x. P' x) \rightarrow (\exists x. P x)\)
proof
  assume eP1: \(EX x. P' x\)
  then obtain x where P1: \(P1 x\) ..
  from ePeqP1 obtain z where P1eqP: \(\forall x > z. P x = P' x\) ..
  let \(?w' = x + (\text{abs}(x-z)+1) * d\)
  let \(?w = x - (\text{abs}(x-z) + 1)*d\)
  have \(ww[\text{simp}]: \emptyset = ?w \lor ?w'\) by simp add: algebra-simps
  from dp\(\) have \(w: \emptyset > z\) by simp only: \(ww[\text{incr-lemma}]\)
  hence \(P' x = P' \emptyset\) using \(P1eqP1\) by blast
also have ... = P(?w) using w P1eqP by blast
finally have P ?w using P1 by blast
thus EX x. P x ..
qed

lemma incr-mult-lemma:
assumes dpos: (0::int) < d and plus: ALL x::int. P x --> P(x + d) and knneg:
0 <= k
shows ALL x. P x --> P(x + k*d)
using knneg
proof (induct rule:int-ge-induct)
case base thus ?case by simp
next
case (step i)
  { fix x
      have P x --> P (x + i * d) using step.hyps by blast
      also have ... --> P(x + (i + 1) * d) using plus[THEN spec, of x + i * d]
        by (simp add:int-distrib ac-simps)
      ultimately have P x --> P(x + (i + 1) * d) by blast }
  thus ?case ..
qed

lemma cppi:
assumes dp: 0 < D and p1:EX z. ALL x> z. P x = P' x
and nb:ALL x. (EX j: {1..D}. (EX (b::int): A. x # b - j) --> P (x) --> P (x + D)
 and pd: ALL x. P' x = P' (x-k*D)
supports (EX j: {1..D}. (EX (b::int): A. x # b - j)) (is ~L = (?R1 OR ?R2))
proof
{ assume ?R2 hence ?L by blast }
moreover
moreover
{ fix x
  assume P: P x and H: ~ ?R2
  { fix y assume ~ (EX j: {1..D}. (EX b: A. P (b - j)) and P: P y
    hence ~ (EX (j::int): {1..D}. EX (b::int): A. y = b - j) by auto
    with nb P have P (y + D) by auto } }
  hence ALL x. (EX (j::int): {1..D}. EX (b::int): A. P (b - j)) --> P (x)
     --> P (x + D) by blast
  with H P have th: ALL x. P x --> P (x + D) by auto
from p1 obtain z where z: ALL x. x > z --> (P x = P' x) by blast
let ?y = x + (0 - z) + 1 + D
have zp: 0 <= (|x - z| + 1) by arith
from dp have yz: ?y > z using incr-lemma[OF dp] by simp
from yz[rule-format, OF yz] incr-mult-lemma[OF dp th zp, rule-format, OF P]
have th2: P' ?y by auto
with periodic-finite-ex[OF dp pd]
have (?R1 by blast)
ultimately show (?thesis by blast)
qed

lemma simp-from-to: \{i..j::int\} = (if j < i then {} else insert i \{i+1..j\})
apply(simp add:atLeastAtMost-def atLeast-def atMost-def)
apply(fastforce)
done

theorem unity-coeff-ex: (\exists (x::\semiring-0.Rings.dvd). P (l * x)) \equiv (\exists x. l dvd (x + 0) \land P x)
apply (rule eq-reflection [symmetric])
apply (rule iffI)
defer
apply (erule exE)
apply (erule exE)
apply (rule_tac x = l * x in exI)
apply (simp add: dvd-def)
apply (rule_tac x = k in exI)
apply simp
apply (erule dvdE)
apply (rule-tac x = k in exI)
apply simp
apply (erule exE)
apply simp
apply (erule conjE)
apply simp
done

lemma zdvd-mono:
  fixes k m t :: int
  assumes k \neq 0
  shows m dvd t \equiv k * m dvd k * t
  using assms by simp

lemma uminus-dvd-conv:
  fixes d t :: int
  shows d dvd t \equiv - d dvd t and d dvd t \equiv d dvd - t
  by simp-all

Theorems for transforming predicates on nat to predicates on int

lemma zdiff-int-split: P (int (x - y)) =
((y \leq x \longrightarrow P (int x - int y)) \land (x < y \longrightarrow P 0))
by (cases y \leq x) (simp-all add: zdiff-int)

Specific instances of congruence rules, to prevent simplifier from looping.

theorem imp-le-cong:
[x = x'; \theta \leq x' \Longrightarrow P = P'] \Longrightarrow (\theta \leq (x::int) \longrightarrow P) = (\theta \leq x' \longrightarrow P')
by simp
THEORY "Presburger"

theorem conj-le-cong:
  \([x = x'; 0 \leq x' \implies P = P'] \implies (0 \leq (x::int) \land P) = (0 \leq x' \land P')\)
by (simp cong: conj-cong)

ML-file Tools/Qelim/cooper.ML
setup Cooper.setup

method-setup presburger = \\[
let
  fan keyword k = Scan.lift (Args.$$ k -- Args.colon) >> K ()
  fan simple-keyword k = Scan.lift (Args.$$ k) >> K ()
  val addN = add
  val delN = del
  val elimN = elim
  val any-keyword = keyword addN || keyword delN || simple-keyword elimN
  val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
  in
  Scan.optional (simple-keyword elimN >> K false) true --
  Scan.optional (keyword addN |-- thms) [] --
  Scan.optional (keyword delN |-- thms) [] >>
  (fn ((elim, add-ths), del-ths) => fn ctxt =>
    SIMPLE-METHOD' (Cooper.tac elim add-ths del-ths ctxt))
end]

declare dvd-eq-mod-eq-0 [symmetric, presburger]
declare mod-1 [presburger]
declare mod-0 [presburger]
declare mod-by-1 [presburger]
declare mod-self [presburger]
declare div-by-0 [presburger]
declare mod-by-0 [presburger]
declare mod-div-trivial [presburger]
declare div-mod-equality2 [presburger]
declare div-mod-equality [presburger]
declare mod-div-equality2 [presburger]
declare mod-div-equality [presburger]
declare mod-mult-self1 [presburger]
declare mod-mult-self2 [presburger]
declare mod2-Suc-Suc [presburger]
declare not-mod-2-eq-0-eq-1 [presburger]
declare nat-zero-less-power-iff [presburger]

lemma [presburger, algebra]: m mod 2 = (1::nat) \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0 \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat) \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0 \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: \( m \mod 2 = (1::\text{int}) \leftrightarrow \neg 2 \divd m \) by presburger

60.4 Try0
ML-file Tools/try0.ML
end

61 SMT2: Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2

theory SMT2
imports Divides
keywords smt2-status :: diag
begin

61.1 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

typedecl 'a symb-list

consts
  Symb-Nil :: 'a symb-list
  Symb-Cons :: 'a ⇒ 'a symb-list ⇒ 'a symb-list

typedecl pattern

consts
  pat :: 'a ⇒ pattern
  nopat :: 'a ⇒ pattern

definition trigger :: pattern symb-list symb-list ⇒ bool ⇒ bool where
  trigger - P = P

61.2 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.
definition fun-app :: 'a ⇒ 'a where fun-app f = f

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas array-rules = ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def

61.3 Normalization

lemma case-bool-if[abs-def]: case-bool x y P = (if P then x else y)
  by simp

lemmas Ex1-def-raw = Ex1-def[abs-def]
lemmas Ball-def-raw = Ball-def[abs-def]
lemmas Bex-def-raw = Bex-def[abs-def]
lemmas abs-if-raw = abs-if[abs-def]
lemmas min-def-raw = min-def[abs-def]
lemmas max-def-raw = max-def[abs-def]

61.4 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where \( l = 0 \). This Schönheitsfehler is corrected in the \( \text{div-as-z3div} \) and \( \text{mod-as-z3mod} \) theorems.

definition z3div :: int ⇒ int ⇒ int where
  z3div k l = (if \( l \geq 0 \) then \( k \div l \) else \( k \div (\neg l) \))

definition z3mod :: int ⇒ int ⇒ int where
  z3mod k l = \( k \mod (if \( l \geq 0 \) then \( l \) else \( \neg l \)) \)

lemma div-as-z3div:
  \( \forall \ k \ l. \ k \div l = (if \ l = 0 \ then \ 0 \ else \ if \ l > 0 \ then \ z3div k l \ else \ z3div (\neg k) (\neg l)) \)
  by (simp add: z3div-def)

lemma mod-as-z3mod:
  \( \forall \ k \ l. \ k \mod l = (if \ l = 0 \ then \ k \ else \ if \ l > 0 \ then \ z3mod k l \ else \ z3mod (\neg k) (\neg l)) \)
  by (simp add: z3mod-def)

61.5 Setup

ML-file Tools/SMT2/smt2-util.ML
ML-file Tools/SMT2/smt2-failure.ML
ML-file Tools/SMT2/smt2-config.ML
ML-file Tools/SMT2/smt2-builtin.ML
ML-file Tools/SMT2/smt2-datatypes.ML
ML-file Tools/SMT2/smt2-normalize.ML
method-setup smt2 = ⟨
  Scan.optional Attrib.thms [] >>
  (fn thms => fn ctxt =>
    METHOD (fn facts => HEADGOAL (SMT2-Solver.smt2-tac ctxt (thms @ facts))))
⟩ apply an SMT solver to the current goal (based on SMT-LIB 2)

61.6 Configuration

The current configuration can be printed by the command smt2-status, which shows the values of most options.

61.7 General configuration options

The option smt2-solver can be used to change the target SMT solver. The possible values can be obtained from the smt2-status command.

Due to licensing restrictions, Z3 is not enabled by default. Z3 is free for non-commercial applications and can be enabled by setting Isabelle system option z3-non-commercial to yes.

declare [[smt2-solver = z3]]

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

declare [[smt2-timeout = 20]]

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

declare [[smt2-random-seed = 1]]

In general, the binding to SMT solvers runs as an oracle, i.e., the SMT solvers are fully trusted without additional checks. The following option can
cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently only implemented for Z3.

```declare [[smt2-oracle = false]]```

Each SMT solver provides several commandline options to tweak its behaviour. They can be passed to the solver by setting the following options.

```declare [[cvc3-new-options = ]]
declare [[cvc4-new-options = ]]
declare [[z3-new-options = ]]```

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

```declare [[smt2-infer-triggers = false]]```

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

```declare [[z3-new-extensions = false]]```

### 61.8 Certificates

By setting the option `smt2-certificates` to the name of a file, all following applications of an SMT solver are cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending `.certs` instead of `.thy`) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

```declare [[smt2-certificates = ]]
```

The option `smt2-read-only-certificates` controls whether only stored certificates are used or invocation of an SMT solver is allowed. When set to `true`, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to `false` and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

```declare [[smt2-read-only-certificates = false]]```

### 61.9 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to `false`.

```declare [[smt2-read-only-certificates = false]]```
**THEORY “SMT2”**

**declare** \([\text{smt2-verbose} = \text{true}]\)

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option **smt2-trace** should be set to **true**.

**declare** \([\text{smt2-trace} = \text{false}]\)

### 61.10 Schematic rules for Z3 proof reconstruction

Several prof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in **z3-rule** are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in **z3-simp** are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

**lemmas** \([\text{z3-new-rule}] =\)

refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distrib field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not

**lemma** \([\text{z3-new-rule}]:\)

\((P \land Q) = (\neg (\neg P \lor \neg Q))\)
\((P \land Q) = (\neg (\neg Q \lor \neg P))\)
\((\neg P \land Q) = (\neg (P \lor \neg Q))\)
\((\neg P \land Q) = (\neg (\neg Q \lor P))\)
\((P \land \neg Q) = (\neg (\neg P \lor Q))\)
\((P \land \neg Q) = (\neg (Q \lor \neg P))\)
\((\neg P \land \neg Q) = (\neg (P \lor Q))\)
\((\neg P \land \neg Q) = (\neg (Q \lor P))\)
by **auto**

**lemma** \([\text{z3-new-rule}]:\)

\((P \Rightarrow Q) = (Q \lor \neg P)\)
\((\neg P \Rightarrow Q) = (P \lor Q)\)
\((\neg P \Rightarrow Q) = (Q \lor P)\)
\((\text{True} \Rightarrow P) = P\)
\((P \Rightarrow \text{True}) = \text{True}\)
\((\text{False} \Rightarrow P) = \text{True}\)
\((P \Rightarrow P) = \text{True}\)
by **auto**

**lemma** \([\text{z3-new-rule}]:\)

\(((P = Q) \Rightarrow R) = (R \mid (Q = (\neg P)))\)
by **auto**

**lemma** \([\text{z3-new-rule}]:\)

\((\neg \text{True}) = \text{False}\)
\((\neg \text{False}) = \text{True}\)
\((x = x) = \text{True}\)
THEORY “SMT2”

(P = True) = P
(True = P) = P
(P = False) = (¬ P)
(False = P) = (¬ P)
((¬ P) = P) = False
(P = (¬ P)) = False
((¬ P) = (¬ Q)) = (P = Q)
¬ (P = (¬ Q)) = (P = Q)
¬ ((¬ P) = Q) = (P = Q)
(P ≠ Q) = (Q = (¬ P))
(P = Q) = ((¬ P ∨ Q) ∧ (P ∨ ¬ Q))
(P ≠ Q) = ((¬ P ∨ ¬ Q) ∧ (P ∨ Q))

by auto

lemma [z3-new-rule]:
(if P then P else ¬ P) = True
(if ¬ P then ¬ P else P) = True
(if P then True else False) = P
(if P then False else True) = (¬ P)
(if P then Q else True) = ((¬ P) ∨ Q)
(if P then Q else True) = (Q ∨ (¬ P))
(if P then Q else ¬ Q) = (P = Q)
(if P then Q else ¬ Q) = (Q = P)
(if P then ¬ Q else Q) = (P = (¬ Q))
(if P then ¬ Q else Q) = ((¬ Q) = P)
(if ¬ P then x else y) = (if P then y else x)
(if P then (if Q then x else y) else x) = (if P ∧ (¬ Q) then y else x)
(if P then (if Q then x else y) else x) = (if (¬ Q) ∧ P then y else x)
(if P then (if Q then x else y) else y) = (if P ∧ Q then x else y)
(if P then (if Q then x else y) else y) = (if Q ∧ P then x else y)
(if P then x else if P then y else z) = (if P then x else z)
(if P then x else if Q then x else y) = (if P ∨ Q then x else y)
(if P then x else if Q then x else y) = (if Q ∨ P then x else y)
(if P then x = y else x = z) = (x = (if P then y else z))
(if P then x = y else y = z) = (y = (if P then x else z))
(if P then x = y else z = y) = (y = (if P then x else z))

by auto

lemma [z3-new-rule]:
θ + (x::int) = x
x + 0 = x
x + x = 2 * x
0 * x = 0
1 * x = x
x + y = y + x

by (auto simp add: mult-2)

lemma [z3-new-rule]:
P = Q ∨ P ∨ Q
\[ P = Q \lor \neg P \lor \neg Q \]
\[ (\neg P) = Q \lor P \lor \neg Q \]
\[ (\neg P) = Q \lor \neg P \lor \neg Q \]
\[ P = (\neg Q) \lor \neg P \lor \neg Q \]
\[ P \neq Q \lor P \lor \neg Q \]
\[ P \neq (\neg Q) \lor P \lor \neg Q \]
\[ (\neg P) \neq Q \lor P \lor \neg Q \]
\[ P \lor Q \lor (\neg P) \neq Q \]
\[ P \lor \neg Q \lor P \neq Q \]
\[ \neg P \lor Q \lor P \neq Q \]
\[ P \lor y = (\text{if } P \text{ then } x \text{ else } y) \]
\[ P \lor (\text{if } P \text{ then } x \text{ else } y) = y \]
\[ \neg P \lor x = (\text{if } P \text{ then } x \text{ else } y) \]
\[ \neg P \lor (\text{if } P \text{ then } x \text{ else } y) = x \]
\[ P \lor R \lor \neg (\text{if } P \text{ then } Q \text{ else } R) \]
\[ \neg (\text{if } P \text{ then } Q \text{ else } R) \lor \neg P \lor Q \]
\[ (\text{if } P \text{ then } Q \text{ else } R) \lor P \lor R \]
\[ (\text{if } P \text{ then } Q \text{ else } R) \lor \neg P \lor \neg Q \]
\[ (\text{if } P \text{ then } Q \text{ else } R) \lor P \lor \neg R \]
\[ (\text{if } P \text{ then } \neg Q \text{ else } R) \lor P \lor \neg Q \]
\[ (\text{if } P \text{ then } Q \text{ else } R) \lor P \lor R \]
\[ \text{by auto} \]

\text{hide-type (open) symb-list pattern }
\text{hide-const (open) Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod }

end

62 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup

theory Sledgehammer
imports Presburger SMT2
keywords sledgehammer :: diag and sledgehammer-params :: thy-decl
begin

lemma size-ne-size-imp-ne: size \( x \neq \) size \( y \implies x \neq y \)
by (erule contrapos-nn) (rule arg-cong)

ML-file Tools/Sledgehammer/async-manager.ML
ML-file Tools/Sledgehammer/sledgehammer-util.ML
ML-file Tools/Sledgehammer/sledgehammer-fact.ML
ML-file Tools/Sledgehammer/sledgehammer-proof-methods.ML
ML-file Tools/Sledgehammer/sledgehammer-isar-annotate.ML
63  Code-Numeral: Numeric types for code generation onto target language numerals only

theory  Code-Numeral
imports  Nat-Transfer Divides Lifting
begin

63.1  Type of target language integers

typedef  integer = UNIV :: int set

morphism  int-of-integer integer-of-int ..

setup-lifting (no-code) type-definition-integer

lemma  integer-eq-iff:
\( k = l \iff \text{int-of-integer} k = \text{int-of-integer} l \)
by  transfer rule

lemma  integer-eq1:
\( \text{int-of-integer} k = \text{int-of-integer} l \implies k = l \)
using  integer-eq-iff [of k l] by  simp

lemma  int-of-integer-int-of-int [simp]:
\( \text{int-of-integer} (\text{integer-of-int} k) = k \)
by  transfer rule

lemma  integer-of-int-int-of-integer [simp]:
\( \text{integer-of-int} (\text{int-of-integer} k) = k \)
by  transfer rule

instantiation  integer :: ring-1
begin
lif-definition zero-integer :: integer
  is 0 :: int
.

declare zero-integer.rep-eq [simp]

lif-definition one-integer :: integer
  is 1 :: int
.

declare one-integer.rep-eq [simp]

lif-definition plus-integer :: integer ⇒ integer ⇒ integer
  is plus :: int ⇒ int ⇒ int
.

declare plus-integer.rep-eq [simp]

lif-definition uminus-integer :: integer ⇒ integer
  is uminus :: int ⇒ int
.

declare uminus-integer.rep-eq [simp]

lif-definition minus-integer :: integer ⇒ integer ⇒ integer
  is minus :: int ⇒ int ⇒ int
.

declare minus-integer.rep-eq [simp]

lif-definition times-integer :: integer ⇒ integer ⇒ integer
  is times :: int ⇒ int ⇒ int
.

declare times-integer.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps)+
end

lemma [transfer-rule]:
rel-fun HOL.eq pcr-integer (of-nat :: nat ⇒ int) (of-nat :: nat ⇒ integer)
by (unfold of-nat-def [abs-def]) transfer-prover

lemma [transfer-rule]:
rel-fun HOL.eq pcr-integer (λk :: int. k :: int) (of-int :: int ⇒ integer)
proof –
have rel-fun HOL.eq pcr-integer (of-int :: int ⇒ int) (of-int :: int ⇒ integer)
  by (unfold of-int-of-nat [abs-def]) transfer-prover
then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  rel-fun HOL.eq pcr-integer (numeral :: num ⇒ int) (numeral :: num ⇒ integer)
proof –
  have rel-fun HOL.eq pcr-integer (numeral :: num ⇒ int) (λn. af-int (numeral n))
    by transfer-prover
  then show ?thesis by simp
qed

lemma int-of-integer-of-nat [simp]:
  int-of-integer (of-nat n) = of-nat n
by transfer rule

lift-definition integer-of-nat :: nat ⇒ integer
  is of-nat :: nat ⇒ int
.

lemma integer-of-nat-eq-of-nat [code]:
  integer-of-nat = of-nat
by transfer rule

lemma int-of-integer-integer-of-nat [simp]:
  int-of-integer (integer-of-nat n) = of-nat n
by transfer rule

lift-definition nat-of-integer :: integer ⇒ nat
  is Int.nat
.

lemma nat-of-integer-of-nat [simp]:
  nat-of-integer (of-nat n) = n
by transfer simp

lemma int-of-integer-of-int [simp]:
  int-of-integer (of-int k) = k
by transfer simp

lemma nat-of-integer-integer-of-nat [simp]:
  nat-of-integer (integer-of-nat n) = n
by transfer simp

lemma integer-of-int-eq-of-int [simp, code-abbrev]:
  integer-of-int = of-int
by transfer (simp add: fun-eq-iff)

lemma of-int-integer-of [simp]:
of-int (int-of-integer k) = (k :: integer)
by transfer rule

lemma int-of-integer-numeral [simp]:
  int-of-integer (numeral k) = numeral k
by transfer rule

lemma int-of-integer-sub [simp]:
  int-of-integer (Num.sub k l) = Num.sub k l
by transfer rule

instantiation integer :: {ring-div, equal, linordered-idom}
begin

lift-definition div-integer :: integer ⇒ integer ⇒ integer
  is Divides.div :: int ⇒ int ⇒ int
  .

declare div-integer.rep-eq [simp]

lift-definition mod-integer :: integer ⇒ integer ⇒ integer
  is Divides.mod :: int ⇒ int ⇒ int
  .

declare mod-integer.rep-eq [simp]

lift-definition abs-integer :: integer ⇒ integer
  is abs :: int ⇒ int
  .

declare abs-integer.rep-eq [simp]

lift-definition sgn-integer :: integer ⇒ integer
  is sgn :: int ⇒ int
  .

declare sgn-integer.rep-eq [simp]

lift-definition less-eq-integer :: integer ⇒ integer ⇒ bool
  is less-eq :: int ⇒ int ⇒ bool
  .
lift-definition less-integer :: integer ⇒ integer ⇒ bool
is less :: int ⇒ int ⇒ bool
.

lift-definition equal-integer :: integer ⇒ integer ⇒ bool
is HOL.equal :: int ⇒ int ⇒ bool
.

instance proof
qed (transfer, simp add: algebra-simps equal less-le-not-le [symmetric] mult-strict-right-mono linear)+

end

lemma [transfer-rule]:
rel-fun pcr-integer (rel-fun pcr-integer pcr-integer) (min :: - ⇒ - ⇒ int) (min :: - ⇒ - ⇒ integer)
by (unfold min-def [abs-def]) transfer-prover

lemma [transfer-rule]:
rel-fun pcr-integer (rel-fun pcr-integer pcr-integer) (max :: - ⇒ - ⇒ int) (max :: - ⇒ - ⇒ integer)
by (unfold max-def [abs-def]) transfer-prover

lemma int-of-integer-min [simp]:
int-of-integer (min k l) = min (int-of-integer k) (int-of-integer l)
by transfer rule

lemma int-of-integer-max [simp]:
int-of-integer (max k l) = max (int-of-integer k) (int-of-integer l)
by transfer rule

lemma nat-of-integer-non-positive [simp]:
k ≤ 0 ⇒ nat-of-integer k = 0
by transfer simp

lemma of-nat-of-integer [simp]:
of-nat (nat-of-integer k) = max 0 k
by transfer auto

instance integer :: semiring-numeral-div
by intro-classes (transfer,
  fact semiring-numeral-div-class.diff-invert-add1
  semiring-numeral-div-class.le-add-diff-inverse2
  semiring-numeral-div-class.mult-div-cancel
  semiring-numeral-div-class.div-less
  semiring-numeral-div-class.mod-less
  semiring-numeral-div-class.div-positive
  semiring-numeral-div-class.mod-less-eq-dividend)
semiring-numeral-div-class.pos-mod-bound
semiring-numeral-div-class.pos-mod-sign
semiring-numeral-div-class.mod-mult2-eq
semiring-numeral-div-class.div-mult2-eq
semiring-numeral-div-class.discrete)

lemma integer-of-nat-0: integer-of-nat 0 = 0
by transfer simp

lemma integer-of-nat-1: integer-of-nat 1 = 1
by transfer simp

lemma integer-of-nat-numeral:
  integer-of-nat (numeral n) = numeral n
by transfer simp

63.2 Code theorems for target language integers

Constructors

definition Pos :: num ⇒ integer
where
  [simp, code-abbrev]: Pos = numeral

lemma [transfer-rule]:
  rel-fun HOL.eq pcr-integer numeral Pos
by simp transfer-prover

definition Neg :: num ⇒ integer
where
  [simp, code-abbrev]: Neg n = − Pos n

lemma [transfer-rule]:
  rel-fun HOL.eq pcr-integer (λ n. − numeral n) Neg
by (simp add: Neg-def [abs-def]) transfer-prover

code-datatype 0::integer Pos Neg

Auxiliary operations

lift-definition dup :: integer ⇒ integer
  is λk::int. k + k

lemma dup-code [code]:
  dup 0 = 0
dup (Pos n) = Pos (Num.Bit0 n)
dup (Neg n) = Neg (Num.Bit0 n)
by (transfer, simp only: numeral-Bit0 minus-add-distrib)+

lift-definition sub :: num ⇒ num ⇒ integer
is $\lambda m. \text{numeral } m - \text{numeral } n :: \text{int}$.

lemma sub-code [code]:

- $\text{sub Num.} \text{One Num.} \text{One} = 0$
- $\text{sub (Num.Bit0 } m\text{) Num.} \text{One} = \text{Pos (Num.BitM } m\text{)}$
- $\text{sub (Num.Bit1 } m\text{) Num.} \text{One} = \text{Pos (Num.Bit0 } m\text{)}$
- $\text{sub Num.} \text{One (Num.Bit0 } n\text{) = Neg (Num.BitM } n\text{)}$
- $\text{sub Num.} \text{One (Num.Bit1 } n\text{) = Neg (Num.Bit0 } n\text{)}$
- $\text{sub (Num.Bit0 } m\text{) (Num.Bit0 } n\text{) = dup (sub } m\text{ } n\text{)}$
- $\text{sub (Num.Bit1 } m\text{) (Num.Bit1 } n\text{) = dup (sub } m\text{ } n\text{)}$
- $\text{sub (Num.Bit0 } m\text{) (Num.Bit0 } n\text{) = dup (sub } m\text{ } n\text{)} + 1$
- $\text{sub (Num.Bit0 } m\text{) (Num.Bit1 } n\text{) = dup (sub } m\text{ } n\text{)} - 1$

by (transfer, simp add: dbl-def dbl-inc-def dbl-dec-def)+

Implementations

lemma one-integer-code [code, code-unfold]:

- $1 = \text{Pos Num.} \text{One}$
 by simp

lemma plus-integer-code [code]:

- $k + 0 = (k::\text{integer})$
- $0 + l = (l::\text{integer})$
- $\text{Pos } m + \text{Pos } n = \text{Pos (} m + n\text{)}$
- $\text{Pos } m + \text{Neg } n = \text{sub } m\text{ } n$
- $\text{Neg } m + \text{Pos } n = \text{sub } n\text{ } m$
- $\text{Neg } m + \text{Neg } n = \text{Neg (} m + n\text{)}$

by (transfer, simp)+

lemma uminus-integer-code [code]:

- $\text{uminus } 0 = (0::\text{integer})$
- $\text{uminus (Pos } m\text{) = Neg } m$
- $\text{uminus (Neg } m\text{) = Pos } m$

by simp-all

lemma minus-integer-code [code]:

- $k - 0 = (k::\text{integer})$
- $0 - l = \text{uminus (} l::\text{integer})$
- $\text{Pos } m - \text{Pos } n = \text{sub } m\text{ } n$
- $\text{Pos } m - \text{Neg } n = \text{Pos (} m + n\text{)}$
- $\text{Neg } m - \text{Pos } n = \text{Neg (} m + n\text{)}$
- $\text{Neg } m - \text{Neg } n = \text{sub } n\text{ } m$

by (transfer, simp)+

lemma abs-integer-code [code]:

- $|k| = (\text{if (} k::\text{integer}) < 0 \text{ then } - k\text{ else } k)$
 by simp

lemma sgn-integer-code [code]:
\[ \text{sgn } k = (\text{if } k = 0 \text{ then } 0 \text{ else if } (k::\text{integer}) < 0 \text{ then } -1 \text{ else } 1) \]

by simp

**lemma** times-integer-code [code]:
\[
\begin{align*}
    k \ast 0 &= (0::\text{integer}) \\
    0 \ast l &= (0::\text{integer}) \\
    \text{Pos } m \ast \text{Pos } n &= \text{Pos } (m \ast n) \\
    \text{Pos } m \ast \text{Neg } n &= \text{Neg } (m \ast n) \\
    \text{Neg } m \ast \text{Pos } n &= \text{Neg } (m \ast n) \\
    \text{Neg } m \ast \text{Neg } n &= \text{Pos } (m \ast n)
\end{align*}
\]

by simp-all

**definition** divmod-integer :: integer \(\Rightarrow\) integer \(\Rightarrow\) integer \(\times\) integer

where
\[
\text{divmod-integer } k \ l = (k \text{ div } l, k \text{ mod } l)
\]

**lemma** fst-divmod [simp]:
\[
\text{fst } (\text{divmod-integer } k \ l) = k \text{ div } l
\]

by (simp add: divmod-integer-def)

**lemma** snd-divmod [simp]:
\[
\text{snd } (\text{divmod-integer } k \ l) = k \text{ mod } l
\]

by (simp add: divmod-integer-def)

**definition** divmod-abs :: integer \(\Rightarrow\) integer \(\Rightarrow\) integer \(\times\) integer

where
\[
\text{divmod-abs } k \ l = (|k| \text{ div } |l|, |k| \text{ mod } |l|)
\]

**lemma** fst-divmod-abs [simp]:
\[
\text{fst } (\text{divmod-abs } k \ l) = |k| \text{ div } |l|
\]

by (simp add: divmod-abs-def)

**lemma** snd-divmod-abs [simp]:
\[
\text{snd } (\text{divmod-abs } k \ l) = |k| \text{ mod } |l|
\]

by (simp add: divmod-abs-def)

**lemma** divmod-abs-code [code]:
\[
\begin{align*}
    \text{divmod-abs } \text{Pos } k \ (\text{Pos } l) &= \text{divmod } k \ l \\
    \text{divmod-abs } \text{Neg } k \ (\text{Neg } l) &= \text{divmod } k \ l \\
    \text{divmod-abs } \text{Neg } k \ (\text{Pos } l) &= \text{divmod } k \ l \\
    \text{divmod-abs } \text{Pos } k \ (\text{Neg } l) &= \text{divmod } k \ l \\
    \text{divmod-abs } j \ 0 &= (0, |j|) \\
    \text{divmod-abs } 0 \ j &= (0, 0)
\end{align*}
\]

by (simp-all add: prod-eq-iff)

**lemma** divmod-integer-code [code]:
\[
\text{divmod-integer } k \ l =
\begin{align*}
&= (\text{if } k = 0 \text{ then } 0, 0 \text{ else if } l = 0 \text{ then } (0, k) \text{ else } \\
&\quad \text{apsnd } \circ \text{times } \circ \text{ sgn}) \ l \ (\text{if } \text{sgn } k = \text{sgn } l)
\end{align*}
\]
then divmod-abs k l
else (let (r, s) = divmod-abs k l in
  if s = 0 then (−r, 0) else (−r − 1, |l| − s)))

proof −
  have aux1: \k l::int. sgn k = sgn l \iff k = 0 ∧ l = 0 ∨ 0 < l ∧ 0 < k ∨ l < 0 ∧ k < 0
    by (auto simp add: sgn-if)
  have aux2: \q::int. int-of-integer k = int-of-integer l * q \iff int-of-integer k = int-of-integer l * −q by auto
  show \thesis
    by (simp add: prod-eq-iff integer-eq-iff case-prod-beta aux1 aux2)
qed

lemma div-integer-code [code]:
  k div l = fst (divmod-integer k l)
  by simp

lemma mod-integer-code [code]:
  k mod l = snd (divmod-integer k l)
  by simp

lemma equal-integer-code [code]:
  HOL.equal 0 (0::integer) \iff True
  HOL.equal 0 (Pos l) \iff False
  HOL.equal 0 (Neg l) \iff False
  HOL.equal (Pos k) 0 \iff False
  HOL.equal (Pos k) (Pos l) \iff HOL.equal k l
  HOL.equal (Pos k) (Neg l) \iff False
  HOL.equal (Neg k) 0 \iff False
  HOL.equal (Neg k) (Pos l) \iff False
  HOL.equal (Neg k) (Neg l) \iff HOL.equal k l
  by (simp-all add: equal)

lemma equal-integer-refl [code nbe]:
  HOL.equal (k::integer) k \iff True
  by (fact equal-refl)

lemma less-eq-integer-code [code]:
  0 ≤ (0::integer) \iff True
  0 ≤ Pos l \iff True
  0 ≤ Neg l \iff False
  Pos k ≤ 0 \iff False
  Pos k ≤ Pos l \iff k ≤ l
  Pos k ≤ Neg l \iff False
  Neg k ≤ 0 \iff True
  Neg k ≤ Pos l \iff True
  Neg k ≤ Neg l \iff l ≤ k
by simp-all

lemma less-integer-code [code]:
0 < (0::integer) ↔ False
0 < Pos l ↔ True
0 < Neg l ↔ False
Pos k < 0 ↔ False
Pos k < Pos l ↔ k < l
Pos k < Neg l ↔ False
Neg k < 0 ↔ True
Neg k < Pos l ↔ True
Neg k < Neg l ↔ l < k
by simp-all

lift-definition integer-of-num :: num ⇒ integer
is numeral :: num ⇒ int
.

lemma integer-of-num [code]:
integer-of-num num.One = 1
integer-of-num (num.Bit0 n) = (let k = integer-of-num n in k + k)
integer-of-num (num.Bit1 n) = (let k = integer-of-num n in k + k + 1)
by (transfer, simp only: numeral.simps Let-def)+

lift-definition num-of-integer :: integer ⇒ num
is num-of-nat ◦ nat
.

lemma num-of-integer-code [code]:
um-of-integer k = (if k ≤ 1 then Num.One else let
(l, j) = divmod-integer k 2;
l′ = num-of-integer l;
l″ = l′ + l′
in if j = 0 then l″ else l″ + Num.One)
proof –
{
  assume int-of-integer k mod 2 = 1
  then have nat (int-of-integer k mod 2) = nat 1 by simp
  moreover assume *: 1 < int-of-integer k
  ultimately have **: nat (int-of-integer k) mod 2 = 1 by (simp add: nat-mod-distrib)
  have num-of-nat (nat (int-of-integer k)) =
    num-of-nat (2 * (nat (int-of-integer k) div 2) + nat (int-of-integer k) mod 2)
  by simp
  then have num-of-nat (nat (int-of-integer k)) =
    num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + nat (int-of-integer k) mod 2)
  by (simp add: mult-2)
  with ** have num-of-nat (nat (int-of-integer k)) =
THEORY “Code-Numeral”

num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + 1)
  by simp
}

note aux = this

show ?thesis
  by (auto simp add: num-of-integer-def nat-of-integer-def Let-def case-prod-beta
    nat-le integer-eq-iff less-eq-integer-def
    nat-mult-distrib nat-div-distrib num-of-nat-One num-of-nat-plus-distrib
    mult-2 [where 'a=nat] aux add-One)

qed

lemma nat-of-integer-code [code]:
  nat-of-integer k = (if k ≤ 0 then 0
    else let
      (l, j) = divmod-integer k 2;
      l' = nat-of-integer l;
      l'' = l' + l'
in if j = 0 then l'' else l'' + 1)

proof –
  obtain j where k = integer-of-int j

proof
  show k = integer-of-int (int-of-integer k) by simp

qed

lemma int-of-integer-code [code]:
  int-of-integer k = (if k < 0 then – (int-of-integer (– k))
    else if k = 0 then 0
    else let
      (l, j) = divmod-int k 2;
      l' = 2 * int-of-integer l
      in if j = 0 then l' else l' + 1)
  by (auto simp add: split-def Let-def integer-eq-iff zmult-div-cancel)

lemma integer-of-int-code [code]:
  integer-of-int k = (if k < 0 then – (integer-of-int (– k))
    else if k = 0 then 0
    else let
      (l, j) = divmod-int k 2;
      l' = 2 * integer-of-int l
      in if j = 0 then l' else l' + 1)
  by (auto simp add: split-def Let-def integer-eq-iff zmult-div-cancel)
hide-const \textbf{(open)} Pos Neg sub dup divmod-abs

63.3 Serializer setup for target language integers

code-reserved Eval int Integer abs

code-printing

\textbf{type-constructor} integer \rightarrow
\begin{align*}
\text{(SML)} & \text{IntInf.int} \\
\text{and (OCaml)} & \text{Big'}\cdot\text{int}.big'\cdot\text{int} \\
\text{and (Haskell)} & \text{Integer} \\
\text{and (Scala)} & \text{BigInt} \\
\text{and (Eval)} & \text{int}
\end{align*}

| \textbf{class-instance} integer :: equal \rightarrow
| \text{(Haskell)} -

code-printing

\textbf{constant} 0::integer \rightarrow
\begin{align*}
\text{(SML)} & 0 \\
\text{and (OCaml)} & \text{Big'}\cdot\text{int}.zero'\cdot\text{big'}\cdot\text{int} \\
\text{and (Haskell)} & 0 \\
\text{and (Scala)} & \text{BigInt}(0)
\end{align*}

setup \langle\langle
\text{fold (Numeral.add-code @{const-name Code-Numeral.Pos}}
\begin{align*}
\text{false Code-Printer.literal-numeral) [SML, OCaml, Haskell, Scala]}
\end{align*}
\rangle\rangle

setup \langle\langle
\text{fold (Numeral.add-code @{const-name Code-Numeral.Neg}}
\begin{align*}
\text{true Code-Printer.literal-numeral) [SML, OCaml, Haskell, Scala]}
\end{align*}
\rangle\rangle

code-printing

\textbf{constant} plus :: integer \Rightarrow - \Rightarrow - \Rightarrow
\begin{align*}
\text{(SML)} & \text{IntInf.}+ ((-), (-)) \\
\text{and (OCaml)} & \text{Big'}\cdot\text{int.add'}\cdot\text{big'}\cdot\text{int} \\
\text{and (Haskell)} & \text{infixl 6 }+ \\
\text{and (Scala)} & \text{infixl 7 }+ \\
\text{and (Eval)} & \text{infixl 8 }+
\end{align*}

| \textbf{constant} uminus :: integer \Rightarrow - \Rightarrow
\begin{align*}
\text{(SML)} & \text{IntInf.} - \\
\text{and (OCaml)} & \text{Big'}\cdot\text{int.minus'}\cdot\text{big'}\cdot\text{int} \\
\text{and (Haskell)} & \text{negate} \\
\text{and (Scala)} & !((- -)) \\
\text{and (Eval)} & - \sim / -
\end{align*}

| \textbf{constant} minus :: integer \Rightarrow - \Rightarrow
\begin{align*}
\text{(SML)} & \text{IntInf.- ((-), (-))} \\
\text{and (OCaml)} & \text{Big'}\cdot\text{int.sub'}\cdot\text{big'}\cdot\text{int}
\end{align*}
and (Haskell) infixl 6 −
and (Scala) infixl 7 −
and (Eval) infixl 8 −

| constant Code-Numeral.dup →
  (SML) IntInf.*/ (2, / (−))
  (OCaml) Big'-'int.mult'-'big'-'int/ (Big'-'int.big'-'int-of'-'int/ 2)
  (Haskell) !(2 * -)
  (Scala) !(2 * -)

| constant Code-Numeral.sub →
  (SML) !(raise/ Fail/ sub)
  (OCaml) failwith/ sub
  (Haskell) error/ sub
  (Scala) !sys.error(sub)

| constant times :: integer ⇒ - ⇒ - ⇒
  (SML) IntInf.*/ ((-), (-))
  (OCaml) Big'-'int.mult'-'big'-'int
  (Haskell) infixl 7 *
  (Scala) infixl 8 *
  (Eval) infixl 9 *

| constant Code-Numeral.divmod-abs →
  (SML) IntInf.divMod/ (IntInf.abs -/ IntInf.abs -)
  (OCaml) Big'-'int.quomod'-'big'-'int/ (Big'-'int.abs'-'big'-'int -)/ (Big'-'int.abs'-'big'-'int -)
  (Haskell) divMod/ (abs -)/ (abs -)
  (Scala) !((k: BigInt) => (l: BigInt) => if (l == 0) (BigInt(0), k) else/ (k.abs '/% l.abs))
  (Eval) Integer.div'-'mod/ (abs -)/ (abs -)

| constant HOL.equal :: integer ⇒ - ⇒ bool →
  (SML) !(l : IntInf.int) = (−)
  (OCaml) Big'-'int.eq'-'big'-'int
  (Haskell) infix 4 ==
  (Scala) infixl 5 ==
  (Eval) infixl 6 ==

| constant less-eq :: integer ⇒ - ⇒ bool →
  (SML) IntInf.<= ((-), (-))
  (OCaml) Big'-'int.le'-'big'-'int
  (Haskell) infix 4 <=
  (Scala) infixl 4 <=
  (Eval) infixl 6 <=

| constant less :: integer ⇒ - ⇒ bool →
  (SML) IntInf.< ((-), (-))
  (OCaml) Big'-'int.lt'-'big'-'int
  (Haskell) infix 4 <
  (Scala) infixl 4 <
  (Eval) infixl 6 <

code-identifier

code-module Code-Numeral → (SML) Arith and (OCaml) Arith and (Haskell)
63.4 Type of target language naturals

typedef natural = UNIV :: nat set
  morphisms nat-of-natural natural-of-nat ..

setup-lifting (no-code) type-definition-natural

lemma natural-eq-iff [termination-simp]:
  m = n ↔ nat-of-natural m = nat-of-natural n
  by transfer rule

lemma natural-eqI:
  nat-of-natural m = nat-of-natural n ⇒ m = n
  using natural-eq-iff [of m n] by simp

lemma nat-of-natural-of-nat-inverse [simp]:
  nat-of-natural (natural-of-nat n) = n
  by transfer rule

lemma natural-of-nat-of-natural-inverse [simp]:
  natural-of-nat (nat-of-natural n) = n
  by transfer rule

instantiation natural :: {comm-monoid-diff, semiring-1}

begin

lift-definition zero-natural :: natural
  is 0 :: nat

  declare zero-natural.rep-eq [simp]

lift-definition one-natural :: natural
  is 1 :: nat

  declare one-natural.rep-eq [simp]

lift-definition plus-natural :: natural ⇒ natural ⇒ natural
  is plus :: nat ⇒ nat ⇒ nat

  declare plus-natural.rep-eq [simp]

lift-definition minus-natural :: natural ⇒ natural ⇒ natural
  is minus :: nat ⇒ nat ⇒ nat

  .
declare minus-natural.rep-eq [simp]

lift-definition times-natural :: natural ⇒ natural ⇒ natural
  is times :: nat ⇒ nat ⇒ nat
.

declarer times-natural.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps)+

end

lemma [transfer-rule]:
  rel-fun HOL.eq pcr-natural (λn::nat. n) (of-nat :: nat ⇒ natural)
proof –
  have rel-fun HOL.eq pcr-natural (of-nat :: nat ⇒ nat) (of-nat :: nat ⇒ natural)
    by (unfold of-nat-def [abs-def]) transfer-prover
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  rel-fun HOL.eq pcr-natural (numeral :: num ⇒ nat) (numeral :: num ⇒ natural)
proof –
  have rel-fun HOL.eq pcr-natural (numeral :: num ⇒ nat) (λn. of-nat (numeral n))
    by transfer-prover
  then show ?thesis by simp
qed

lemma nat-of-natural-of-nat [simp]:
  nat-of-natural (of-nat n) = n
by transfer rule

lemma natural-of-nat-of-nat [simp, code-abbrev]:
  natural-of-nat = of-nat
by transfer rule

lemma of-nat-of-natural [simp]:
  of-nat (nat-of-natural n) = n
by transfer rule

lemma nat-of-natural-numeral [simp]:
  nat-of-natural (numeral k) = numeral k
by transfer rule

instantiation natural :: {semiring-div, equal, linordered-semiring}
begin
lift-definition \textit{\texttt{div-natural :: natural ⇒ natural ⇒ natural}}
\textit{\texttt{is Divides.div :: nat ⇒ nat ⇒ nat}}.

declare \textit{\texttt{div-natural.rep-eq [simp]}}

lift-definition \textit{\texttt{mod-natural :: natural ⇒ natural ⇒ natural}}
\textit{\texttt{is Divides.mod :: nat ⇒ nat ⇒ nat}}.

declare \textit{\texttt{mod-natural.rep-eq [simp]}}

lift-definition \textit{\texttt{less-eq-natural :: natural ⇒ natural ⇒ bool}}
\textit{\texttt{is less-eq :: nat ⇒ nat ⇒ bool}}.

declare \textit{\texttt{less-eq-natural.rep-eq [termination-simp]}}

lift-definition \textit{\texttt{less-natural :: natural ⇒ natural ⇒ bool}}
\textit{\texttt{is less :: nat ⇒ nat ⇒ bool}}.

declare \textit{\texttt{less-natural.rep-eq [termination-simp]}}

lift-definition \textit{\texttt{equal-natural :: natural ⇒ natural ⇒ bool}}
\textit{\texttt{is HOL.equal :: nat ⇒ nat ⇒ bool}}.

instance proof
qed (transfer, simp add: algebra-simps equal less-le-not-le [symmetric] linear)+

end

lemma [transfer-rule]:
rel-fun pcr-natural (rel-fun pcr-natural pcr-natural) (min :: - ⇒ - ⇒ nat) (min :: - ⇒ - ⇒ natural)
by (unfold min-def [abs-def]) transfer-prover

lemma [transfer-rule]:
rel-fun pcr-natural (rel-fun pcr-natural pcr-natural) (max :: - ⇒ - ⇒ nat) (max :: - ⇒ - ⇒ natural)
by (unfold max-def [abs-def]) transfer-prover

lemma nat-of-natural-min [simp]:
nat-of-natural (min k l) = min (nat-of-natural k) (nat-of-natural l)
by transfer rule

lemma nat-of-natural-max [simp]:
nat-of-natural (\text{max } k \ l) = \text{max} (\text{nat-of-natural } k) (\text{nat-of-natural } l)
by transfer rule

lift-definition natural-of-integer :: integer \Rightarrow \text{natural}
  is nat :: int \Rightarrow nat
  .

lift-definition integer-of-natural :: \text{natural} \Rightarrow integer
  is of-nat :: \text{nat} \Rightarrow \text{int}
  .

lemma natural-of-integer-of-natural [simp]:
  natural-of-integer (integer-of-natural n) = n
by transfer simp

lemma integer-of-natural-of-integer [simp]:
  integer-of-natural (natural-of-integer k) = \text{max } 0 \ k
by transfer auto

lemma int-of-integer-of-natural [simp]:
  int-of-integer (integer-of-natural n) = of-nat (\text{nat-of-natural } n)
by transfer rule

lemma integer-of-natural-of-nat [simp]:
  integer-of-natural (of-nat n) = of-nat n
by transfer rule

lemma [measure-function]:
  is-measure nat-of-natural
by (rule is-measure-trivial)

63.5 Inductive representation of target language naturals

lift-definition Suc :: \text{natural} \Rightarrow \text{natural}
  is Nat.Suc
  .

declare Suc.rep-eq [simp]

rep-datatype 0::natural Suc
by (transfer, fact nat.induct nat.inject nat.distinct)+

lemma natural-cases [case-names nat, cases type: natural]:
  fixes m :: \text{natural}
  assumes \(\forall \ n. \ m = \text{of-nat } n \Rightarrow \text{P}\)
  shows \text{P}
  using assms by transfer blast

lemma [simp, code]:
THEORY “Code-Numeral”

size-natural = nat-of-natural
proof (rule ext)
  fix n
  show size-natural n = nat-of-natural n
    by (induct n) simp-all
qed

lemma [simp, code]:
  size = nat-of-natural
proof (rule ext)
  fix n
  show size n = nat-of-natural n
    by (induct n) simp-all
qed

lemma natural-decr [termination-simp]:
  n ≠ 0 ⇒ nat-of-natural n − Nat.Suc 0 < nat-of-natural n
  by transfer simp

lemma natural-zero-minus-one:
  (0::natural) − 1 = 0
  by simp

lemma Suc-natural-minus-one:
  Suc n − 1 = n
  by transfer simp

hide-const (open) Suc

63.6 Code refinement for target language naturals

lift-definition Nat :: integer ⇒ natural
  is nat
  .

lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
  by (transfer, simp)+

lemma [code abstype]:
  Nat (integer-of-natural n) = n
  by transfer simp

lemma [code abstract]:
  integer-of-natural (natural-of-nat n) = of-nat n
  by simp
lemma [code abstract]:
  integer-of-natural (natural-of-integer k) = max 0 k
  by simp

lemma [code-abbrev]:
  natural-of-integer (Code-Numeral.Pos k) = numeral k
  by transfer simp

lemma [code abstract]:
  integer-of-natural 0 = 0
  by transfer simp

lemma [code abstract]:
  integer-of-natural 1 = 1
  by transfer simp

lemma [code abstract]:
  integer-of-natural (Code-Numeral.Suc n) = integer-of-natural n + 1
  by transfer simp

lemma [code]:
  nat-of-natural = nat-of-integer o integer-of-natural
  by transfer (simp add: fun-eq-iff)

lemma [code, code-unfold]:
  case-natural f g n = (if n = 0 then f else g (n - 1))
  by (cases n rule: natural.exhaust) (simp-all, simp add: Suc-def)

declare natural.rec [code del]

lemma [code abstract]:
  integer-of-natural (m + n) = integer-of-natural m + integer-of-natural n
  by transfer simp

lemma [code abstract]:
  integer-of-natural (m - n) = max 0 (integer-of-natural m - integer-of-natural n)
  by transfer simp

lemma [code abstract]:
  integer-of-natural (m * n) = integer-of-natural m * integer-of-natural n
  by transfer (simp add: of-nat-mult)

lemma [code abstract]:
  integer-of-natural (m div n) = integer-of-natural m div integer-of-natural n
  by transfer (simp add: zdiv-int)

lemma [code abstract]:
  integer-of-natural (m mod n) = integer-of-natural m mod integer-of-natural n
by transfer (simp add: zmod-int)

lemma [code]:
  HOL.equal m n ↔ HOL.equal (integer-of-natural m) (integer-of-natural n)
by transfer (simp add: equal)

lemma [code nbe]:
  HOL.equal n (n::natural) ↔ True
by (simp add: equal)

lemma [code]:
  m ≤ n ↔ integer-of-natural m ≤ integer-of-natural n
by transfer simp

lemma [code]:
  m < n ↔ integer-of-natural m < integer-of-natural n
by transfer simp

hide-const (open) Nat

lifting-update integer.lifting
lifting-forget integer.lifting

lifting-update natural.lifting
lifting-forget natural.lifting

code-reflect Code-Numeral
  datatypes natural = -
  functions integer-of-natural natural-of-integer
end

64 Lifting-Set: Setup for Lifting/Transfer for the set type

theory Lifting-Set
imports Lifting
begin

64.1 Relator and predicator properties

definition rel-set :: (‘a ⇒ ‘b ⇒ bool) ⇒ ‘a set ⇒ ‘b set ⇒ bool
  where rel-set R = (λA B. (∀x∈A. ∃y∈B. R x y) ∧ (∀y∈B. ∃x∈A. R x y))

lemma rel-setI:
  assumes ∀. x ∈ A → ∃y∈B. R x y
  assumes ∀. y ∈ B → ∃x∈A. R x y
  shows rel-set R A B
using assms unfolding rel-set-def by simp

lemma rel-setD1: \[ \text{rel-set } R \ A \ B \ ; \ x \in A \implies \exists y \in B. \ R \ x \ y \]
and rel-setD2: \[ \text{rel-set } R \ A \ B \ ; \ y \in B \implies \exists x \in A. \ R \ x \ y \]
by(simp-all add: rel-set-def)

lemma rel-set-conversep [simp]: \[ \text{rel-set } A^{-1-1} = (\text{rel-set } A)^{-1-1} \]
unfolding rel-set-def by auto

lemma rel-set-eq [relator-eq]: \[ \text{rel-set} (\text{op} =) = (\text{op} =) \]
unfolding rel-set-def fun-eq-iff by auto

lemma rel-set-mono [relator-mono]:
assumes A ≤ B
shows rel-set A ≤ rel-set B
using assms unfolding rel-set-def by blast

lemma rel-set-OO [relator-distr]: \[ \text{rel-set } R \OO \text{rel-set } S = \text{rel-set } (R \OO S) \]
apply (rule sym)
apply (intro ext, rename-tac X Z)
apply (rule iffI)
apply (rule-tac b={y. \exists x\in X. R x y} \land (\exists z\in Z. S y z) \in \text{relcomppI})
apply (simp add: rel-set-def, fast)
apply (simp add: rel-set-def, fast)
apply (simp add: rel-set-def, fast)
done

lemma Domainp-set [relator-domain]:
\[ \text{Domainp } (\text{rel-set } T) = (\lambda A. \text{Ball } A \ (\text{Domainp } T)) \]
unfolding rel-set-def Domainp-iff [abs-def]
apply (intro ext)
apply (rule iffI)
apply blast
apply (rename-tac A, rule-tac x={y. \exists x\in A. T x y} \in \text{exI, fast})
done

lemma left-total-rel-set [transfer-rule]:
\[ \text{left-total } A \implies \text{left-total } (\text{rel-set } A) \]
unfolding left-total-def rel-set-def
apply safe
apply (rename-tac X, rule-tac x={y. \exists x\in X. A x y} \in \text{exI, fast})
done

lemma left-unique-rel-set [transfer-rule]:
\[ \text{left-unique } A \implies \text{left-unique } (\text{rel-set } A) \]
unfolding left-unique-def rel-set-def
by fast

lemma right-total-rel-set [transfer-rule]:
right-total $A \implies$ right-total (rel-set $A$)

using left-total-rel-set[of $A^{-1}$] by simp

lemma right-unique-rel-set [transfer-rule]:
right-unique $A \implies$ right-unique (rel-set $A$)

unfolding right-unique-def rel-set-def by fast

lemma bi-total-rel-set [transfer-rule]:
bi-total $A \implies$ bi-total (rel-set $A$)

by (simp add: bi-total-alt-def left-total-rel-set right-total-rel-set)

lemma bi-unique-rel-set [transfer-rule]:
bi-unique $A \implies$ bi-unique (rel-set $A$)

unfolding bi-unique-def rel-set-def by fast

lemma set-relator-eq-onp [relator-eq-onp]:
rel-set (eq-onp $P$) = eq-onp ($\lambda A. \text{Ball} A P$)

unfolding fun-eq-iff rel-set-def eq-onp-def Ball-def by fast

lemma bi-unique-rel-set-lemma:
assumes bi-unique $R$ and rel-set $R X Y$
obtains $f$ where $Y = \text{image} f X$ and inj-on $f X$ and $\forall x \in X. R x (f x)$

proof
def $f \equiv \lambda x. \text{THE} y. R x y$
{ fix $x$ assume $x \in X$
  with (rel-set $R X Y$) (bi-unique $R$) have $R x (f x)$
  by (simp add: bi-unique-def rel-set-def $f$-def) (metis theI)
  with assms $x \in X$
  have $R x (f x) \forall x' \in X. R x' (f x) \iff x = x' \forall y \in Y. R x y \iff y = f x f x$
  $\in Y$
  by (fastforce simp add: bi-unique-def rel-set-def)+ }

note $*$ = this

moreover
{ fix $y$ assume $y \in Y$
  with (rel-set $R X Y$) $*$(3) $\forall y \in Y$ have $\exists x \in X. y = f x$
  by (fastforce simp: rel-set-def) }

ultimately show $\forall x \in X. R x (f x) Y = \text{image} f X$ inj-on $f X$
  by (auto simp: inj-on-def image-iff)

qed

64.2 Quotient theorem for the Lifting package

lemma Quotient-set[quot-map]:
assumes Quotient $R$ Abs Rep $T$

shows Quotient (rel-set $R$) (image Abs) (image Rep) (rel-set $T$)

using assms unfolding Quotient-alt-def4
apply (simp add: rel-set-OO[ symmetric])
apply (simp add: rel-set-def, fast)
done
64.3 Transfer rules for the Transfer package

64.3.1 Unconditional transfer rules

context begin
interpretation lifting-syntax.

lemma empty-transfer [transfer-rule]: (rel-set A) {} {}
unfolding rel-set-def by simp

lemma insert-transfer [transfer-rule]:
(A ===> rel-set A ===> rel-set A) insert insert
unfolding rel-fun-def rel-set-def by auto

lemma union-transfer [transfer-rule]:
(rel-set A ===> rel-set A ===> rel-set A) union union
unfolding rel-fun-def rel-set-def by auto

lemma Union-transfer [transfer-rule]:
(rel-set (rel-set A) ===> rel-set A) Union Union
unfolding rel-fun-def rel-set-def by simp fast

lemma image-transfer [transfer-rule]:
(A ===> B) ===> rel-set A ===> rel-set B) image image
unfolding rel-fun-def rel-set-def by simp fast

lemma UNION-transfer [transfer-rule]:
(rel-set A ===> (rel-set B) ===> rel-set B) UNION UNION
unfolding Union-image-eq [symmetric, abs-def] by transfer-prover

lemma Ball-transfer [transfer-rule]:
(rel-set A ===> (A ===> op =) ===> op =) Ball Ball
unfolding rel-set-def rel-fun-def by fast

lemma Bex-transfer [transfer-rule]:
(rel-set A ===> (A ===> op =) ===> op =) Bex Bex
unfolding rel-set-def rel-fun-def by fast

lemma Pow-transfer [transfer-rule]:
(rel-set A ===> rel-set (rel-set A)) Pow Pow
apply (rule rel-funI; rename-tac X Y, rule rel-setI)
apply (rename-tac X′, rule-tac x=\{y∈ Y. ∃ x∈ X′. A x y\} in rev-bexI, clarsimp)
apply (simp add: rel-set-def, fast)
apply (rename-tac Y′, rule-tac x=\{x∈ X. ∃ y∈ Y′. A x y\} in rev-bexI, clarsimp)
apply (simp add: rel-set-def, fast)
done

lemma rel-set-transfer [transfer-rule]:
((A ===> B ===> op =) ===> rel-set A ===> rel-set B ===> op =)
rel-set rel-set
  unfolding rel-fun-def rel-set-def by fast

lemma bind-transfer [transfer-rule]:
  (rel-set A ===> (A ===> rel-set B) ===> rel-set B) Set.bind Set.bind
  unfolding bind- UNION [abs-def] by transfer-prover

lemma INF-parametric [transfer-rule]:
  (rel-set A ===> (A ===> HOL.eq) ===> HOL.eq) INFIMUM INFIMUM
  unfolding INF-def [abs-def] by transfer-prover

lemma SUP-parametric [transfer-rule]:
  (rel-set R ===> (R ===> HOL.eq) ===> HOL.eq) SUPREMUM SUPREMUM
  unfolding SUP-def [abs-def] by transfer-prover

64.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma member-transfer [transfer-rule]:
  assumes bi-unique A
  shows (A ===> rel-set A ===> op =) (op ∈) (op ∈)
  using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast

lemma right-total-Collect-transfer [transfer-rule]:
  assumes right-total A
  shows ((A ===> op =) ===> rel-set A) (λP. Collect (λx. P x ∧ Domainp A x)) Collect
  using assms unfolding right-total-def rel-set-def rel-fun-def Domainp-iff by fast

lemma Collect-transfer [transfer-rule]:
  assumes bi-total A
  shows ((A ===> op =) ===> rel-set A) Collect Collect
  using assms unfolding rel-fun-def rel-set-def bi-total-def by fast

lemma inter-transfer [transfer-rule]:
  assumes bi-unique A
  shows (rel-set A ===> rel-set A ===> rel-set A) inter inter
  using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast

lemma Diff-transfer [transfer-rule]:
  assumes bi-unique A
  shows (rel-set A ===> rel-set A ===> rel-set A) (op −) (op −)
  using assms unfolding rel-fun-def rel-set-def bi-unique-def
  unfolding Ball-def Bex-def Diff-eq
  by (safe, simp, metis, simp, metis)

lemma subset-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (rel-set A ===> rel-set A ===> op =) (op ⊆) (op ⊆)
  unfolding subset-eq [abs-def] by transfer-prover
lemma right-total-UNIV-transfer[transfer-rule]:
assumes right-total A
shows (rel-set A) (Collect (Domainp A)) UNIV
using assms unfolding right-total-def rel-set-def Domainp-iff by blast

lemma UNIV-transfer [transfer-rule]:
assumes bi-total A
shows (rel-set A) UNIV UNIV
using assms unfolding rel-set-def bi-total-def by simp

lemma right-total-Compl-transfer [transfer-rule]:
assumes bi-unique A and right-total A
shows (rel-set A) === rel-set A uminus uminus
unfolding Compl-eq [abs-def]
by (subst Collect-conj-eq [symmetric]) transfer-prover

lemma Compl-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
shows (rel-set A) === rel-set A uminus uminus

lemma right-total-Inter-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
shows (rel-set (rel-set A)) === rel-set A (\lambda S. uminus S \cap Collect (Domainp A)) Inter
unfolding Inter-eq [abs-def]
by (subst Collect-conj-eq [symmetric]) transfer-prover

lemma Inter-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
shows (rel-set (rel-set A)) === rel-set A Inter Inter
unfolding Inter-eq [abs-def] by transfer-prover

lemma filter-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows ((A === op=) ===> rel-set A ===> rel-set A) Set.filter Set.filter
unfolding Set.filter-def [abs-def] rel-fun-def rel-set-def by blast

lemma finite-transfer [transfer-rule]:
bi-unique A ==> (rel-set A ===> op=) finite finite
by (rule rel-funI, erule (1) bi-unique-rel-set-lemma)
(auto dest: finite-imageD)

lemma card-transfer [transfer-rule]:
bi-unique A ==> (rel-set A ===> op=) card card
by (rule rel-funI, erule (1) bi-unique-rel-set-lemma)
(simp add: card-image)
THEORY “Lifting-Option”

lemma vimage-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-unique B
  shows ((A ===> B) ===> rel-set B ===> rel-set A) vimage vimage
  unfolding vimage-def[abs-def] by transfer-prover

lemma Image-parametric [transfer-rule]:
  assumes bi-unique A
  shows (rel-set (rel-prod A B) ===> rel-set A ===> rel-set B) op “ op “
  by(intro rel-funI rel-setI)
  (force dest: rel-setD1 bi-uniqueDr[OF assms], force dest: rel-setD2 bi-uniqueDl[OF assms])

end

lemma (in comm-monoid-set) F-parametric [transfer-rule]:
  fixes A :: 'b ⇒ 'c ⇒ bool
  assumes bi-unique A
  shows rel-fun (rel-fun A (op =)) (rel-fun (rel-set A) (op =)) F F
  proof(rule rel-funI)+
  fix f :: 'b ⇒ 'a and g S T
  assume rel-fun A (op =) f g rel-set A S T
  with [bi-unique A] obtain i where bij-betw i S T \x. x ∈ S ==> f x = g (i x)
  by (auto elim: bi-unique-rel-set-lemma simp: rel-fun-def bij-betw-def)
  then show F f S = F g T
  by (simp add: reindex-bij-betw)
  qed

lemmas setsum-parametric = setsum.F-parametric
lemmas setprod-parametric = setprod.F-parametric
end

65 Lifting-Option: Setup for Lifting/Transfer for the option type

theory Lifting-Option
imports Lifting Option
begin

65.1 Relator and predicator properties

lemma rel-option-iff:
  rel-option R x y = (case (x, y) of (None, None) ⇒ True
  | (Some x, Some y) ⇒ R x y
  | _ ⇒ False)
  by (auto split: prod.split option.split)
65.2 Transfer rules for the Transfer package

context
begin
interpretation lifting-syntax .

lemma None-transfer [transfer-rule]: (rel-option A) None None
  by (rule option.rel-inject)

lemma Some-transfer [transfer-rule]: (A ===> rel-option A) Some Some
  unfolding rel-fun-def by simp

lemma case-option-transfer [transfer-rule]:
  (B ===> (A ===> B) ===> rel-option A ===> B) case-option case-option
  unfolding rel-fun-def split-option-all by simp

lemma map-option-transfer [transfer-rule]:
  (A ===> B) ===> rel-option A ===> rel-option B) map-option map-option
  unfolding map-option-case[abs-def] by transfer-prover

lemma option-bind-transfer [transfer-rule]:
  (rel-option A ===> (A ===> rel-option B) ===> rel-option B)
  Option.bind Option.bind
  unfolding rel-fun-def split-option-all by simp

end
end

66 Lifting-Product: Setup for Lifting/Transfer for the product type

theory Lifting-Product
imports Lifting Basic-BNFs
begin

lemma prod-pred-inject [simp]:
  pred-prod P1 P2 (a, b) = (P1 a ∧ P2 b)
  unfolding pred-prod-def fun-eq-iff prod-set-simps by blast

66.1 Transfer rules for the Transfer package

context
begin
interpretation lifting-syntax .

lemma Pair-transfer [transfer-rule]: (A ===> B ===> rel-prod A B) Pair Pair
  unfolding rel-fun-def rel-prod-def by simp
lemma fst-transfer [transfer-rule]: (rel-prod A B ===> A) fst fst
  unfolding rel-fun-def rel-prod-def by simp

lemma snd-transfer [transfer-rule]: (rel-prod A B ===> B) snd snd
  unfolding rel-fun-def rel-prod-def by simp

lemma case-prod-transfer [transfer-rule]:
  ((A ===> B ===> C) ===> rel-prod A B ===> C) case-prod case-prod
  unfolding rel-fun-def rel-prod-def by simp

lemma curry-transfer [transfer-rule]:
  ((rel-prod A B ===> C) ===> A ===> B ===> C) curry curry
  unfolding curry-def by transfer-prover

lemma map-prod-transfer [transfer-rule]:
  ((rel-prod A B ===> C) ===> (B ===> D) ===> rel-prod A B ===> rel-prod C D)
  map-prod map-prod
  unfolding map-prod-def [abs-def] by transfer-prover

lemma rel-prod-transfer [transfer-rule]:
  ((A ===> B ===> op =) ===> (C ===> D ===> op =) ===> rel-prod A C ===> rel-prod B D ===> op =) rel-prod rel-prod
  unfolding rel-fun-def by auto

end

end

67  List: The datatype of finite lists

theory List
imports Sledgehammer Code-Numeral Lifting-Set Lifting-Option Lifting-Product
begin

datatype-new (set: 'a) list =
  Nil  ([]) | Cons (hd: 'a) (tl: 'a list)  (infixr # 65)
for
  map: map
rel: list-all2
where
  tl [] = []

datatype-compat list

lemma [case-names Nil Cons, cases type: list]:
  — for backward compatibility — names of variables differ
  (y = [] ===> P) ===> (\forall a list. y = a # list ===> P) ===> P
by (rule list.exhaust)

lemma [case-names Nil Cons, induct type: list]:
— for backward compatibility — names of variables differ
\[ P \implies (\forall a \text{ list. } P \text{ list } \implies P (a \# \text{ list})) \implies P \text{ list} \]
by (rule list.induct)

Compatibility:

setup ⟨⟨ Sign.mandatory-path list ⟩⟩

lemmas inducts = list.induct
lemmas recs = list.rec
lemmas cases = list.case

setup ⟨⟨ Sign.parent-path ⟩⟩

syntax
— list Enumeration
 clumsi : args => 'a list ([(|)])

translations
\[ [x, xs] = x \# [xs] \]
\[ [x] = x \# [] \]

67.1 Basic list processing functions

primrec last :: 'a list => 'a where
last (x # xs) = (if xs = [] then x else last xs)

primrec butlast :: 'a list => 'a list where
butlast [] = [] |
butlast (x # xs) = (if xs = [] then [] else x # butlast xs)

declare list.set[simp del, code del]

lemma set-simps[simp, code, code-post]:
set [] = {}
set (x # xs) = insert x (set xs)
by (simp-all add: list.set)

lemma set-rec: set xs = rec-list {} (\lambda x. insert x) xs
by (induct xs) auto

definition coset :: 'a list => 'a set where
[simp]: coset xs = - set xs

primrec append :: 'a list => 'a list => 'a list (infixr @ 65) where
append-Nil: [] @ ys = ys |
append-Cons: (x#xs) @ ys = x # xs @ ys
primrec rev :: 'a list ⇒ 'a list where
rev [] = [] |
rev (x # xs) = rev xs @ [x]

primrec filter :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list where
filter P [] = [] |
filter P (x # xs) = (if P x then x # filter P xs else filter P xs)

syntax — Special syntax for filter
-filter :: [pttrn, 'a list, bool] => 'a list ((1[-<- . -]))
translations
[x<-xs . P]== \text{CONST} \text{filter}(\%x. P) xs

primrec fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b where
fold-Nil: fold f [] = id |
fold-Cons: fold f (x # xs) = fold f xs o f x

primrec foldr :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b where
foldr-Nil: foldr f [] = id |
foldr-Cons: foldr f (x # xs) = f x o foldr f xs

primrec foldl :: ('b ⇒ 'a ⇒ 'b) ⇒ 'a list ⇒ 'b where
foldl-Nil: foldl f a [] = a |
foldl-Cons: foldl f a (x # xs) = foldl f (f a x) xs

primrec concat:: 'a list list ⇒ 'a list where
concat [] = [] |
concat (x # xs) = x @ concat xs

definition (in monoid-add) listsum :: 'a list ⇒ 'a where
listsum xs = foldr plus xs 0

primrec drop:: nat ⇒ 'a list ⇒ 'a list where
drop-Nil: drop n [] = [] |
drop-Cons: drop n (x # xs) = (case n of \theta ⇒ x # xs \mid Suc m ⇒ drop m xs)

— Warning: simpset does not contain this definition, but separate theorems for
n = \theta and n = Suc k

primrec take:: nat ⇒ 'a list ⇒ 'a list where
take-Nil: take n [] = [] |
take-Cons: take n (x # xs) = (case n of \theta ⇒ [] \mid Suc m ⇒ x # take m xs)
— Warning: simpset does not contain this definition, but separate theorems for
\( n = 0 \) and \( n = \text{Suc} \ k \)

**primrec nth :: 'a list => nat => 'a (infixl 100) where**

\[ \text{nth-Cons: } (x \# xs)! n = \begin{cases} x & \text{if } n = 0 \\ \text{Suc} k & \text{if } x \# xs! k \end{cases} \]

— Warning: simpset does not contain this definition, but separate theorems for
\( n = 0 \) and \( n = \text{Suc} \ k \)

**primrec list-update :: 'a list => nat => 'a => 'a list where**

\[ \text{list-update } [] i v = [] \\
\text{list-update } (x \# xs) i v = \begin{cases} v \# xs & \text{if } i = 0 \\ x \# \text{list-update } xs j v & \text{if } i = \text{Suc} j \end{cases} \]

**nonterminal lupdbinds and lupdbind**

**syntax**

\[ -\text{lupdbind}: ['a, 'a] => \text{lupdbind} \ ((?: := / :)) \]
\[ :: \text{lupdbind} => \text{lupdbinds} (-) \]
\[ -\text{lupdbinds} :: [\text{lupdbind}, \text{lupdbinds}] => \text{lupdbinds} (-, / -) \]
\[ -\text{LUpdate} :: ['a, \text{lupdbinds}] => 'a (-/[-][900,0] 900) \]

**translations**

\[ -\text{LUpdate } xs (-\text{lupdbinds } b \ bs) => -\text{LUpdate } (-\text{LUpdate } xs \ b \ bs) \]
\[ xs[i:=x] => \text{CONST list-update } xs \ i \ x \]

**primrec takeWhile :: ('a => bool) => 'a list => 'a list where**

\[ \text{takeWhile } P [] = [] \\
\text{takeWhile } P (x \# xs) = (\begin{cases} P x & \text{if } x \# \text{takeWhile } P xs \text{ else } [] \end{cases}) \]

**primrec dropWhile :: ('a => bool) => 'a list => 'a list where**

\[ \text{dropWhile } P [] = [] \\
\text{dropWhile } P (x \# xs) = (\begin{cases} P x & \text{if } x \# \text{dropWhile } P xs \text{ else } x \# xs \end{cases}) \]

**primrec zip :: 'a list => 'b list => ('a x 'b) list where**

\[ \text{zip } [] = [] \\
\text{zip-Cons: } \text{zip } xs (y \# ys) = \begin{cases} \text{case } xs \text{ of } [] => [] \\ z \# xs => (z, y) \# \text{zip } zs \ ys \end{cases} \]

— Warning: simpset does not contain this definition, but separate theorems for
\( xs = [] \) and \( xs = z \# zs \)

**primrec product :: 'a list => 'b list => ('a x 'b) list where**

\[ \text{product } [] = [] \\
\text{product } (x\#xs) \ ys = \text{map } (\lambda x. \text{map } (\text{Cons} \ x) \ (\text{product-lists } xss)) \ xs \]

**hide-const (open) product**

**primrec product-lists :: 'a list list => 'a list list where**

\[ \text{product-lists } [] = [[]] \\
\text{product-lists } (xs \# xss) = \text{concat } (\lambda x. \text{map } (\text{Cons} \ x) \ (\text{product-lists } xss)) \ xs \]
primrec upt :: nat ⇒ nat ⇒ nat list where 
upt-0: [i..<0] = [] | 
upt-Suc: [i..<Suc j] = (if i ≤ j then [i..<j] @ [j] else [])

definition insert :: 'a ⇒ 'a list ⇒ 'a list where 
insert x xs = (if x ∈ set xs then xs else x # xs)

definition union :: 'a list ⇒ 'a list ⇒ 'a list where 
union = fold insert

hide-const (open) insert union 
hide-fact (open) insert-def union-def

primrec find :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option where 
find - [] = None | 
find P (x#xs) = (if P x then Some x else find P xs)

hide-const (open) find

definition extract :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a list * 'a * 'a list) option where 
extract P xs = 
(case dropWhile (Not o P) xs of
  [] ⇒ None |
y#ys ⇒ Some(takeWhile (Not o P) xs, y, ys))

hide-const (open) extract

primrec those :: 'a option list ⇒ 'a list option where 
those [] = Some [] |
those (x # xs) = (case x of
  None ⇒ None |
  Some y ⇒ map-option (Cons y) (those xs))

primrec remove1 :: 'a ⇒ 'a list ⇒ 'a list where 
remove1 x [] = [] |
remove1 x (y # xs) = (if x = y then xs else y # remove1 x xs)

primrec removeAll :: 'a ⇒ 'a list ⇒ 'a list where 
removeAll x [] = [] |
removeAll x (y # xs) = (if x = y then removeAll x xs else y # removeAll x xs)

primrec distinct :: 'a list ⇒ bool where 
distinct [] ⇐⇒ True |
distinct (x # xs) ⇐⇒ x ∉ set xs ∧ distinct xs

primrec remdups :: 'a list ⇒ 'a list where
remdups [] = []
remdups (x # xs) = (if x ∈ set xs then remdups xs else x # remdups xs)

fun remdups-adj :: 'a list ⇒ 'a list where
remdups-adj [] = []
remdups-adj [x] = [x]
remdups-adj (x # y # xs) = (if x = y then remdups-adj (x # xs) else x # remdups-adj (y # xs))

primrec replicate :: nat ⇒ 'a ⇒ 'a list where
replicate-0: replicate 0 x = []
replicate-Suc: replicate (Suc n) x = x # replicate n x

Function size is overloaded for all datatypes. Users may refer to the list version as length.

abbreviation length :: 'a list ⇒ nat where
length ≡ size

definition enumerate :: nat ⇒ 'a list ⇒ (nat × 'a) list where
enumerate-eq-zip: enumerate n xs = zip [n..<n + length xs] xs

primrec rotate1 :: 'a list ⇒ 'a list where
rotate1 [] = []
rotate1 (x # xs) = xs @ [x]

definition rotate :: nat ⇒ 'a list ⇒ 'a list where
rotate n = rotate1 ^^ n

definition sublist :: 'a list ⇒ nat set ⇒ 'a list where
sublist xs A = map fst (filter (λp. snd p ∈ A) (zip xs [0..<size xs]))

primrec sublists :: 'a list ⇒ 'a list list where
sublists [] = [[]] 
sublists (x#xs) = (let xss = sublists xs in map (Cons x) xss @ xss)

primrec n-lists :: nat ⇒ 'a list ⇒ 'a list list where
n-lists 0 xs = [[]] 
(n-lists (Suc n) xs = concat (map (λys. map (λy. y # ys) xs) (n-lists n xs))

hide-const (open) n-lists

fun splice :: 'a list ⇒ 'a list ⇒ 'a list where
splice [] ys = ys
splice xs [] = xs
splice (x#xs) (y#ys) = x # y # splice xs ys

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient
<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[a, b] @ [c, d]</code></td>
<td><code>[a, b, c, d]</code></td>
</tr>
<tr>
<td><code>length [a, b, c]</code></td>
<td><code>3</code></td>
</tr>
<tr>
<td><code>set {a, b, c}</code></td>
<td><code>{a, b, c}</code></td>
</tr>
<tr>
<td><code>map f [a, b, c]</code></td>
<td><code>[f a, b, f c]</code></td>
</tr>
<tr>
<td><code>rev [a, b, c]</code></td>
<td><code>[c, b, a]</code></td>
</tr>
<tr>
<td><code>hd [a, b, c, d]</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>tl [a, b, c, d]</code></td>
<td><code>[b, c, d]</code></td>
</tr>
<tr>
<td><code>last [a, b, c, d]</code></td>
<td><code>d</code></td>
</tr>
<tr>
<td><code>butlast [a, b, c, d]</code></td>
<td><code>[a, b, c]</code></td>
</tr>
<tr>
<td><code>filter (λn::nat. n&lt;2) [0,2,1]</code></td>
<td><code>[0,1]</code></td>
</tr>
<tr>
<td><code>concat [[a, b], [c, d, e], [], [f]]</code></td>
<td><code>[a, b, c, d, e, f]</code></td>
</tr>
<tr>
<td><code>fold f [a, b, c] x = f c (f b (f a x))</code></td>
<td></td>
</tr>
<tr>
<td><code>foldr f [a, b, c] x = f a (f b (f c x))</code></td>
<td></td>
</tr>
<tr>
<td><code>foldl f x [a, b, c] = f (f (f x a) b) c</code></td>
<td></td>
</tr>
<tr>
<td><code>zip [a, b, c] [x, y, z]</code></td>
<td><code>[(a, x), (b, y), (c, z)]</code></td>
</tr>
<tr>
<td><code>enumerate 3 [a, b, c] = [(3, a), (4, b), (5, c)]</code></td>
<td></td>
</tr>
<tr>
<td><code>List.product [a, b] [c, d]</code></td>
<td><code>[[a, c], (a, d), (b, c), (b, d)]</code></td>
</tr>
<tr>
<td><code>product-lists [a, b], [c], [d, e]</code></td>
<td><code>[[a, c, d], [a, c, e], [b, c, d], [b, c, e]]</code></td>
</tr>
<tr>
<td><code>splice [a, b, c] [x, y, z]</code></td>
<td><code>[(a, x), b, y, (c, z)]</code></td>
</tr>
<tr>
<td><code>splice [a, b, c, d] [x, y]</code></td>
<td><code>[a, x, b, y, c, d]</code></td>
</tr>
<tr>
<td><code>take 2 [a, b, c, d]</code></td>
<td><code>[a, b]</code></td>
</tr>
<tr>
<td><code>take 6 [a, b, c, d]</code></td>
<td><code>[a, b, c, d]</code></td>
</tr>
<tr>
<td><code>drop 2 [a, b, c, d]</code></td>
<td><code>[c, d]</code></td>
</tr>
<tr>
<td><code>drop 6 [a, b, c, d]</code></td>
<td><code>[]</code></td>
</tr>
<tr>
<td><code>takeWhile (λn. n &lt; 3) [1, 2, 3, 0]</code></td>
<td><code>[1, 2]</code></td>
</tr>
<tr>
<td><code>dropWhile (λn. n &lt; 3) [1, 2, 3, 0]</code></td>
<td><code>[3, 0]</code></td>
</tr>
<tr>
<td><code>distinct [2, 0, 1]</code></td>
<td></td>
</tr>
<tr>
<td><code>remdups [2, 0, 2, 1, 2]</code></td>
<td><code>[0, 1, 2]</code></td>
</tr>
<tr>
<td><code>remdups-adj [2, 2, 3, 1, 1, 2, 1]</code></td>
<td><code>[2, 3, 1, 2, 1]</code></td>
</tr>
<tr>
<td><code>List.insert 2 [0, 1, 2]</code></td>
<td><code>[0, 1, 2]</code></td>
</tr>
<tr>
<td><code>List.insert 3 [0, 1, 2]</code></td>
<td><code>[3, 0, 1, 2]</code></td>
</tr>
<tr>
<td><code>List.union [2, 3, 4] [0, 1, 2]</code></td>
<td><code>[4, 3, 0, 1, 2]</code></td>
</tr>
<tr>
<td><code>List.find (op &lt; 0) [0, 0]</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>List.find (op &lt; 0) [0, 1, 0, 2]</code></td>
<td><code>Some 1</code></td>
</tr>
<tr>
<td><code>List.extract (op &lt; 0) [0, 0]</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>List.extract (op &lt; 0) [0, 1, 0, 2]</code></td>
<td><code>Some ([0], 1, [0, 2])</code></td>
</tr>
<tr>
<td><code>remove1 2 [2, 0, 2, 1, 2]</code></td>
<td><code>[0, 2, 1, 2]</code></td>
</tr>
<tr>
<td><code>removeAll 2 [2, 0, 2, 1, 2]</code></td>
<td><code>[0, 1]</code></td>
</tr>
<tr>
<td><code>[a, b, c, d] ! 2 = c</code></td>
<td></td>
</tr>
<tr>
<td><code>[a, b, c, d] ! 2 := x] = [a, b, x, d]</code></td>
<td></td>
</tr>
<tr>
<td><code>sublist [a, b, c, d, e] {0, 2, 3}</code></td>
<td><code>[a, c, d]</code></td>
</tr>
<tr>
<td><code>sublists [a, b] = [[a, b], [a], [b]]</code></td>
<td></td>
</tr>
<tr>
<td><code>List.n-lists 2 [a, b, c]</code></td>
<td><code>[[a, a], [b, a], [a, b], [b, b], [c, b], [a, c], [b, c], [c, c]]</code></td>
</tr>
<tr>
<td><code>rotate1 [a, b, c, d] = [b, c, d, a]</code></td>
<td></td>
</tr>
<tr>
<td><code>rotate 2 [a, b, c, d] = [a, b, c, d]</code></td>
<td></td>
</tr>
<tr>
<td><code>rotate 3 [a, b, c, d] = [d, a, b, c]</code></td>
<td></td>
</tr>
<tr>
<td><code>replicate 4 a = [a, a, a, a]</code></td>
<td></td>
</tr>
<tr>
<td><code>[2..&lt;5] = [2, 3, 4]</code></td>
<td></td>
</tr>
<tr>
<td><code>listsum [1, 2, 3] = 6</code></td>
<td></td>
</tr>
</tbody>
</table>
implementations.

context linorder

begin

inductive sorted :: 'a list ⇒ bool where
  Nil [iff]: sorted []
 | Cons: ∀ y∈set xs. x ≤ y ⇒ sorted xs ⇒ sorted (x # xs)

lemma sorted-single [iff]:
  sorted [x]
  by (rule sorted.Cons) auto

lemma sorted-many:
  x ≤ y ⇒ sorted (y # zs) ⇒ sorted (x # y # zs)
  by (rule sorted.Cons) (cases y # zs rule: sorted.cases, auto)

lemma sorted-many-eq [simp, code]:
  sorted (x # y # zs) ⊛ x ≤ y ∧ sorted (y # zs)
  by (auto intro: sorted-many elim: sorted.cases)

lemma [code]:
  sorted [] ⊛ True
  sorted [x] ⊛ True
  by simp-all

primrec insort-key :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list where
  insort-key f x [] = [x] |
  insort-key f x (y#ys) =
    (if f x ≤ f y then (x#y#ys) else y#(insort-key f x ys))

definition sort-key :: ('b ⇒ 'a) ⇒ 'b list ⇒ 'b list where
  sort-key f xs = foldr (insort-key f) xs []

definition insort-insert-key :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list where
  insort-insert-key f x xs =
    (if f x ∈ f ` set xs then xs else insort-key f x xs)

abbreviation sort ≡ sort-key (λx. x)
abbreviation insort ≡ insort-key (λx. x)
abbreviation insort-insert ≡ insort-insert-key (λx. x)

end

67.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: \[ [(x,y). x ← xs, y ← ys, x ≠ y] \], the list of all pairs of distinct elements from \( xs \) and \( ys \). The syntax is as in Haskell, except that \(|\) becomes
a dot (like in Isabelle’s set comprehension): \([ e. x \leftarrow xs, \ldots ]\) rather than 
[\{ e| x \leftarrow xs, \ldots \}].

The qualifiers after the dot are

**generators** \( p \leftarrow xs \), where \( p \) is a pattern and \( xs \) an expression of list type,
or

**guards** \( b \), where \( b \) is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of \([ e. x \leftarrow xs]\) is optimized to \( \text{map} \ (\lambda x. e) \ xs \).

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

**nonterminal** \( lc-qual \) and \( lc-quals \)

**syntax**

- \( \text{-listcompr} :: 'a \Rightarrow lc-qual \Rightarrow lc-quals \Rightarrow 'a \text{ list} \ (\ldots) \)
- \( \text{-lc-gen} :: 'a \Rightarrow 'a \text{ list} \Rightarrow lc-qual \ (- \leftarrow -) \)
- \( \text{-lc-test} :: \text{bool} \Rightarrow lc-qual \ (-) \)
- \( \text{-lc-end} :: lc-quals \ (\}) \)
- \( \text{-lc-quals} :: lc-qual \Rightarrow lc-quals \Rightarrow lc-quals \ (, \ldots) \)
- \( \text{-lc-abs} :: 'a \Rightarrow 'b \text{ list} \Rightarrow 'b \text{ list} \)

**syntax (xsymbols)**

- \( \text{-lc-gen} :: 'a \Rightarrow 'a \text{ list} \Rightarrow lc-qual \ (- \leftarrow -) \)

**syntax (HTML output)**

- \( \text{-lc-gen} :: 'a \Rightarrow 'a \text{ list} \Rightarrow lc-qual \ (- \leftarrow -) \)

**parse-translation** \( ⟨⟨ \)

\begin{verbatim}
let
  val NilC = Syntax.const @{const-syntax Nil};
  val ConsC = Syntax.const @{const-syntax Cons};
  val mapC = Syntax.const @{const-syntax map};
  val concatC = Syntax.const @{const-syntax concat};
  val IfC = Syntax.const @{const-syntax If};

  fun single x = ConsC $ x $ NilC;

  fun pat-tr ctxt p e opti = (**x. case p of e => e | - => [] *)
    let
      (* FIXME proper name context! *)
  \end{verbatim}
val x = Free (singleton (Name.variant-list (fold Term.add-free-names [p, e] [])) x, dummyT);
val e = if opti then single e else e;
val case1 = Syntax.const @{syntax-const -case1} $ p $ e;
val case2 = Syntax.const @{syntax-const -case1} $ Syntax.const @{const-syntax Pure.dummy-pattern} $ NilC;
val cs = Syntax.const @{syntax-const -case2} $ case1 $ case2;

in Syntax-Trans.abs-tr [x, Case-Translation.case-tr false ctxt] ] end

fun abs-tr ctxt p e opti =
  (case Term-Position.strip-positions p of
   Free (s, T) =>
     let
       val thy = Proof-Context.theory-of ctxt;
       val s′ = Proof-Context.intern-const ctxt s;
       in
         if Sign.declared-const thy s′
         then (pat-tr ctxt p e opti, false)
         else (Syntax-Trans.abs-tr [p, e], true)
       end
    | - => (pat-tr ctxt p e opti, false));

fun lc-tr ctxt [e, Const (@{syntax-const -lc-test}, _) $ b, qs] =
  let
    val res =
      (case qs of
       Const (@{syntax-const -lc-end}, _) =>> single e
       | Const (@{syntax-const -lc-quals}, _) $ q $ qs =>> lc-tr ctxt [e, q, qs]);
    in IfC $ b $ res $ NilC end
  | lc-tr ctxt [e, Const (@{syntax-const -lc-gen}, _) $ p $ es,
                         Const (@{syntax-const -lc-end}, _)] =
    (case abs-tr ctxt p e true of
     (f, true) =>> mapC $ f $ es
     | (f, false) =>> concatC $ (mapC $ f $ es))
  | lc-tr ctxt [e, Const (@{syntax-const -lc-gen}, _) $ p $ es,
                         Const (@{syntax-const -lc-quals}, _)] $ q $ qs] =
    let val e′ = lc-tr ctxt [e, q, qs];
    in concatC $ (mapC $ (fst (abs-tr ctxt p e′ false)) $ es) end;
  in [[@{syntax-const -listcompr}, lc-tr]] end
)

ML-val ⟨⟨
  let
    val read = Syntax.read-term @{context};
⟩⟩
fun check s1 s2 = read s1 aconv read s2 orelse error (Check failed:  ^ quote s1);

check [(x,y,z), b] if b then [(x, y, z)] else [];
check [(x,y,z), x→xs] map (λx. (x, y, z)) xs;
check [e x y, x←xs, y←ys] concat (map (λx. map (λy. e x y) ys) xs);
check [(x,y,z), x<a, x>b] if x < a then if b < x then [(x, y, z)] else [] else [];
check [(x,y,z), x←xs, x>b] concat (map (λx. if b < x then [(x, y, z)] else [])) xs;
check [(x,y,z), x<a, x←xs] if x < a then map (λx. (x, y, z)) xs else [];
check [(x,y), Cons True x ← xs]
  concat (map (λxa. case xa of [] ⇒ [] | True # x ⇒ [(x, y)] | False # x ⇒ [])) xs;
check [(x,y,z), Cons x [] ← xs]
  concat (map (λxa. case xa of [] ⇒ [] | [x] ⇒ [(x, y, z)] | x # aa # lista ⇒ [])) xs;
check [(x,y,z), x<a, x>b, x=d] if x < a then if b < x then if x = d then [(x, y, z)] else [] else [] else [];
check [(x,y,z), x<a, x>b, y←ys] if x < a then if b < x then map (λy. (x, y, z)) ys else [] else [];
check [(x,y,z), x<a, x←xs, y←ys] if x < a then concat (map (λx. if b < y then [(x, y, z)] else []) xs) else [];
check [(x,y,z), x<a, x←xs, y←ys] if x < a then concat (map (λx. map (λy. (x, y, z)) ys) xs) else [];
check [(x,y,z), x<a, x←xs, x>b, y<a] concat (map (λx. if b < x then if y < a then [(x, y, z)] else [] else []) xs);
check [(x,y,z), x←xs, x>b, y←ys] concat (map (λx. if b < x then map (λy. (x, y, z)) ys) [] xs);
check [(x,y,z), x←xs, y←ys,y>x] concat (map (λx. concat (map (λy. if x < y then [(x, y, z)] else [])) ys)) xs);
check [(x,y,z), x←xs, y←ys,z←zs] concat (map (λx. concat (map (λy. map (λz. (x, y, z)) zs) ys)) xs)
end;

ML ⟨⟨
(* Simproc for rewriting list comprehensions applied to List.set to set
  comprehension. *)

signature LIST-TO-SET-COMPREHENSION =
sig
  val simproc : Proof.context -> cterm -> thm option
end

structure List-to-Set-Comprehension : LIST-TO-SET-COMPREHENSION =
struct
fun all-exists-conv cv ctxt ct = 
  (case Thm.term_of ct of 
    Const (@{const-name HOL.Ex}, -) $ Abs _ => 
      Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct 
    | _ => cv ctxt ct)

fun all-but-last-exists-conv cv ctxt ct = 
  (case Thm.term_of ct of 
    Const (@{const-name HOL.Ex}, -) $ Abs (_, _, Const (@{const-name HOL.Ex}, _) => 
      Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct 
    | _ => cv ctxt ct)

fun Collect-conv cv ctxt ct = 
  (case Thm.term_of ct of 
    Const (@{const-name Set.Collect}, -) $ Abs _ => Conv.arg-conv (Conv.abs-conv cv ctxt) ct 
    | _ => raise CTERM (Collect-conv, [ct]))

fun rewr-conv' th = Conv.rewr-conv (mk-meta-eq th)

fun conjunct-assoc-conv ct = 
  Conv.try_conv 
  (rewr-conv' @{thm conj-assoc} then_conv HOLogic.conj-conv Conv.all-conv conjunct-assoc-conv) ct

fun right-hand-set-comprehension-conv conv ctxt = 
  HOLogic.Trueprop_conv (HOLogic.eq_conv Conv.all-conv 
    (Collect-conv (all-exists-conv conv o #2) ctxt))

datatype termlets = If | Case of (typ * int)

fun simproc ctxt redex = 
  let 
    val set-Nil-I = @{thm trans} OF [@{thm set-simps(1)}], @{thm empty-def} 
    val set-singleton = @{lemma set [a] = {x. x = a} by simp} 
    val inst-Collect-mem-eq = @{lemma set A = {x. x : set A} by simp} 
    val del-refl-eq = @{lemma t = t & P == P by simp} 
    fun mk-set T = Const (@{const-name List.set}, HOLogic.listT T) 
    fun dest-set (Const (@{const-name List.set}, -) $ xs) = xs 
    fun dest-singleton-list ( Const (@{const-name List.Cons}, -) $ t $ (Const (@{const-name List.Nil}, -))) = t 
    | dest-singleton-list t = raise TERM (dest-singleton-list, [t])
THEORY "List"

(* We check that one case returns a singleton list and all other cases
return [], and return the index of the one singleton list case *)

fun possible-index-of-singleton-case cases =
  let
    fun check (i, case-t) s =
      (case strip-abs-body case-t of
        (Const (@{const-name List.Nil}, -)) => s
      | - => (case s of SOME NONE => SOME (SOME i) | - => NONE))
    in
      fold-index check cases (SOME NONE | the-default NONE)
    end

(* returns (case-expr type index chosen-case constr-name) option *)

fun dest-case case-term =
  let
    val (case-const, args) = strip-comb case-term
    in
      (case try dest-Const case-const of
        SOME (c, T) =>
          (case Ctr-Sugar.ctr-sugar-of-case ctxt c of
            SOME {c, ...} =>
              (case possible-index-of-singleton-case (fst (split-last args)) of
                SOME i =>
                  let
                    val constr-names = map (fst o dest-Const) cts
                    val (Ts, -) = strip-type T
                    val T' = List.last Ts
                  in
                    SOME (List.last args, T', i, nth args i, nth constr-names i)
                  end
                | NONE => NONE)
            | NONE => NONE)
          | NONE => NONE)
    end

(* returns condition continuing term option *)

fun dest-if (Const (@{const-name If}, -) $ cond $ then-t $ Const (@{const-name Nil}, -)) =
  SOME (cond, then-t)
  | dest-if - = NONE

fun tac - [] = rtac set-singleton 1 ORELSE rtac inst-Collect-mem-eq 1
  | tac ctxt (If :: cont) =
    Splitter.split-tac [@{thm split-if}] 1
    THEN rtac @{thm conjI} 1
    THEN rtac @{thm impI} 1
    THEN Subgoal.FOCUS (fn {prems, context, ...} =>
      CONVERSION (right-hand-set-comprehension-conv (K
        (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm Eq-TrueI})) Conv.all-conv
        then-conv
        rewr-conv' @{lemma (True & P) = P by simp}) context)) 1) ctxt 1
    THEN tac ctxt cont
    THEN rtac @{thm impI} 1
THEN Subgoal.FOCUS (fn {prems, context, ...} =>
  CONVERSION (right-hand-set-comprehension-conv (K
   (HOLogic.conj-conv Conv.rewr-conv (List.last prems RS @{thm Eq-FalseI}))) Conv.all-conv
    then-conv rewr-conv' @{lemma (False & P) = False by simp})))
context 1) ctxt 1
  THEN rtac set-Nil-I 1
  | tac ctxt (Case (T, i) :: cont) =
    let
      val SOME {injects, distincts, case-thms, split, ...} =
        Ctr-Sugar.ctr-sugar-of ctxt (fst (dest-Type T))
    in
      (* do case distinction *)
      Splitter.split-tac [split] 1
      THEN EVERY
        (map-index (fn (i', -) =>
          (if i' < length case-thms - 1 then rtac @{thm conjI} 1 else all-tac)
          THEN REPEAT-DETERM (rtac @{thm allI} 1)
          THEN (if i' = i then (* continue recursively *)
            Subgoal.FOCUS (fn {prems, context, ...} =>
              CONVERSION (Thm.eta-conversion then-conv right-hand-set-comprehension-conv
               (K
                ((HOLogic.conj-conv
                  (HOLogic.eq-cong Conv.all-conv (rewr-conv' (List.last prems)))
                 Conv.all-conv
                 then-conv (Conv.try-conv (Conv.rewr-conv (map mk-meta-eq injects))))
               Conv.all-conv
               then-conv (Conv.try-conv (Conv.rewr-conv del-refl-eq))
               then-conv conjunct-assoc-conv) context
               then-conv (HOLogic.Trueprop-cong (HOLogic.eq-cong Conv.all-conv
                (Collect-conv (fn (_, ctxt) =>
                  Conv.repeat-conv
                  (all-but-last-exists-conv
                   (K (rewr-conv'
                     @[lemma (EX x. x = t & P x) = P t by simp}) ctxt))))
               context))))
            else
              Subgoal.FOCUS (fn {prems, context, ...} =>
                CONVERSION
                (right-hand-set-comprehension-conv (K
                 (HOLogic.conj-conv
                  ((HOLogic.eq-cong Conv.all-conv
                    (rewr-conv' (List.last prems)))
                   Conv.all-conv
                   (map fn th => th RS @{thm Eq-FalseI}))))
                distincts))
          context))
        THEN 1)
      ctxt 1
      THEN tac ctxt cont
    else
      Subgoal.FOCUS (fn {prems, context, ...} =>
        CONVERSION
        (right-hand-set-comprehension-conv (K
         (HOLogic.conj-conv
          ((HOLogic.eq-cong Conv.all-conv
            (rewr-conv' (List.last prems)))
           Conv.all-conv
           (map fn th => th RS @{thm Eq-FalseI}))))
        distincts))
      Conv.all-conv then-conv
      (rewr-conv' @{lemma (False & P) = False by simp}))) context
then-conv

HOLogic.Trueprop-conv
  (HOLogic.eq-conv Conv.all-conv
   (Collect-conv (fn (-, ctxt) =>
     Conv.repeat-conv
      (Conv.bottom-conv
       (K (rewr-conv'@
         (lemma (EX x. P) = P by simp)) ctxt)))
   context))

1) ctxt 1
   THEN rtac set-Nil-I 1
   end

fun make-inner-eqs bound-vs Tis eqs t =
  (case dest-case t of
   SOME (x, T, i, cont, constr-name) =>
     let
       val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
       val x' = incr-boundvars (length vs) x
       val eqs' = map (incr-boundvars (length vs)) eqs
       val constr-t =
         list-comb
         (Const (constr-name, map snd vs --- T), map Bound (((length vs) - 1) downto 0))
       val constr-eq = Const (@{const-name HOL.eq}, T --> T --> @{typ bool}) $ constr-t $ x'
       in
         make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
       end
   | NONE =>
     (case dest-if t of
      SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition :: eqs) cont
      | NONE =>
        if eqs = [] then NONE (* no rewriting, nothing to be done *)
        else
          let
            val Type (@{type-name List.list}, [rT]) = fastype-of1 (map snd bound-vs, t)
            val pat-eq =
              (case try dest-singleton-list t of
               SOME t' =>
                 Const (@{const-name HOL.eq}, rT --> rT --> @{typ bool})
               $ Bound (length bound-vs) $ t')
               | NONE =>
                 Const (@{const-name Set.member}, rT --> HOLogic.mk-set T rT --> @{typ bool})
               $ Bound (length bound-vs) $ (mk-set rT $ t))
            val reverse-bounds = curry subst-bounds
          end
        end)
   end

end
((map Bound ((length bound-vs − 1) downto 0)) @ [Bound (length bound-vs)]))

val eqs' = map reverse-bounds eqs
val pat-eq' = reverse-bounds pat-eq
val inner-t = fold (fn (-, T) => fn t => HOLogic.exists-const T $ absdummy T)

(rev bound-vs) (fold (curry HOLogic.mk-conj) eqs' pat-eq')

val rhs = HOLogic.mk-Collect (x, rT, inner-t)
val rewrite-rule-t = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))
in SOME (((Goal.prove ctxt [] [] rewrite-rule-t
  (fn {context, ...} => tac context (rev Tis)))) RS @{thm eq-reflection})
end

make-inner-eqs [] [] [] (dest-set (term-of redex))

end

end

simproc-setup list-to-set-comprehension (set xs) = ⟨⟨ K List-to-Set-Comprehension.simproc ⟩⟩

code-datatype set coset

hide-const (open) coset

67.1.2 [] and op #

lemma not-Cons-self [simp]:

xs ≠ x # xs

by (induct xs) auto

lemma not-Cons-self2 [simp]:

x ≠ xs ≠ x

by (rule not-Cons-self [symmetric])

lemma neq-Nil-conv: (xs ≠ []) = (∃y ys. xs = y # ys)

by (induct xs) auto

lemma tl-Nil: tl xs = [] ⟷ xs = [] ∨ (EX x. xs = [x])

by (cases xs) auto

lemma Nil-tl: [] = tl xs ⟷ xs = [] ∨ (EX x. xs = [x])

by (cases xs) auto
lemma length-induct:
\((\forall xs. \forall ys. \text{length } ys < \text{length } xs \rightarrow P ys \Longrightarrow P xs) \Longrightarrow P xs\)
by (fact measure-induct)

lemma list-nonempty-induct [consumes 1, case-names single cons]:
assumes \(xs \neq []\)
assumes single: \(\forall x. P [x]\)
assumes cons: \(\forall xs. xs \neq [] \Longrightarrow P xs \Longrightarrow P (x # xs)\)
shows \(P xs\)
using \(xs \neq []\)
proof (induct \(xs\))
case Nil then show \(?case\) by simp
next
case (Cons \(x xs\))
show \(?case\)
proof (cases \(xs\))
case Nil
with single show \(?thesis\) by simp
next
case Cons
show \(?thesis\)
proof (rule cons)
from Cons show \(xs \neq []\) by simp
with Cons.hyps show \(P xs\).
qed
qed

lemma inj-split-Cons: inj-on \((\lambda(xs, n). n#xs)\) \(X\)
by (auto intro!: inj-onI)

67.1.3 length

Needs to come before \(@\) because of theorem append-eq-append-conv.

lemma length-append [simp]: \(\text{length } (xs @ ys) = \text{length } xs + \text{length } ys\)
by (induct \(xs\)) auto

lemma length-map [simp]: \(\text{length } (\text{map } f xs) = \text{length } xs\)
by (induct \(xs\)) auto

lemma length-rev [simp]: \(\text{length } (\text{rev } xs) = \text{length } xs\)
by (induct \(xs\)) auto

lemma length_tl [simp]: \(\text{length } (\text{tl } xs) = \text{length } xs - 1\)
by (cases \(xs\)) auto

lemma length-0-conv [iff]: \((\text{length } xs = 0) = (xs = [])\)
by (induct \(xs\)) auto

lemma length_greater_0_conv [iff]: \((0 < \text{length } xs) = (xs \neq [])\)
by (induct xs) auto

lemma length-pos-if-in-set: x : set xs \implies length xs > 0
by auto

lemma length-Suc-conv:
(length xs = Suc n) = (∃ y ys. xs = y # ys ∧ length ys = n)
by (induct xs) auto

apply (induct xs, simp, simp)
apply blast
done

lemma impossible-Cons: length xs <= length ys ===> xs = x # ys = False
by (induct xs) auto

lemma list-induct2 [consumes 1, case-names Nil Cons]:
length xs = length ys \implies P [] [] \implies
(\forall x xs y ys. length xs = length ys \implies P xs ys \implies P (x#xs) (y#ys))
\implies P xs ys
proof (induct xs arbitrary: ys)
  case Nil then show ?case by simp
next
  case (Cons x xs ys) then show ?case by (cases ys) simp-all
qed

lemma list-induct3 [consumes 2, case-names Nil Cons]:
length xs = length ys \implies length ys = length zs \implies P [] [] [] \implies
(\forall x xs y ys z zs. length xs = length ys \implies P xs ys zs \implies P (x#xs) (y#ys) (z#zs))
\implies P xs ys zs
proof (induct xs arbitrary: ys zs)
  case Nil then show ?case by simp
next
  case (Cons x xs ys zs) then show ?case by (cases ys, simp-all)
    (cases zs, simp-all)
qed

lemma list-induct4 [consumes 3, case-names Nil Cons]:
length xs = length ys = length zs = length ws \implies
P [] [] [] [] \implies
(\forall x xs y ys z zs w ws. length xs = length ys \implies
length ys = length zs \implies length zs = length ws \implies P (x#xs) (y#ys) (z#zs) (w#ws)) \implies P xs ys zs ws
proof (induct xs arbitrary: ys zs ws)
  case Nil then show ?case by simp
next
  case (Cons x xs ys zs ws) then show ?case by ((cases ys, simp-all), (cases
lemma list-induct2':

\[ P [] \];
\( \forall x xs. P (x#xs) [] \);
\( \forall y ys. P [] (y#ys) \);
\( \forall x xs y ys. P xs ys \Rightarrow P (x#xs) (y#ys) \)
\( \Rightarrow P xs ys \)
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma list-all2-iff:

(list-all2 P xs ys \( \leftrightarrow \) length xs = length ys \( \land \) (\( \forall \) (x, y) \( \in \) set (zip xs ys). P x y))
by (induct xs ys rule: list-induct2') auto

lemma neq-if-length-neq:

length xs \( \neq \) length ys \( \Rightarrow \) (xs = ys) \( = \) False
by (rule Eq-FalseI) auto

simproc-setup list-neq ((xs::'a list) = ys) = \%
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (Const(@\{const-name Nil\},-)) acc = acc
| len (Const(@\{const-name Cons\},-) $ $ xs) (ts,n) = len xs (ts,n+1)
| len (Const(@\{const-name append\},-) $ xs $ ys) acc = len xs (len ys acc)
| len (Const(@\{const-name rev\},-) $ xs) acc = len xs acc
| len (Const(@\{const-name map\},-) $ $ xs) acc = len xs acc
| len t (ts,n) = (t:ts,n);

val ss = simpset-of @\{context\};

fun list-neq ctxt ct =
let
val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
val (ls,m) = len lhs ([]\,0) and (rs,n) = len rhs ([]\,0);
fun prove-neq() =
let
val Type(-,listT::-) = eqT;
val size = HOLogic.size-const listT;
val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
val thm = Goal.prove ctxt [] [ neq-len
\( \Rightarrow \) K (simp-tac (put-simpset ss ctxt) 1));
in SOME (thm RS @\{thm neq-if-length-neq\}) end
in
  if m < n andalso submultiset (op aconv) (ls,rs) orelse
  n < m andalso submultiset (op aconv) (rs,ls)
  then prove-neq() else NONE
end;
in K list-neq end;
⟩⟩

67.1.4 @ – append

lemma append-assoc [simp]: (xs @ ys) @ zs = xs @ (ys @ zs)
by (induct xs) auto

lemma append-Nil2 [simp]: xs @ [] = xs
by (induct xs) auto

lemma append-is-Nil-conv [iff]: (xs @ ys = []) = (xs = [] ∧ ys = [])
by (induct xs) auto

lemma Nil-is-append-conv [iff]: ([] = xs @ ys) = (xs = [] ∧ ys = [])
by (induct xs) auto

lemma append-self-conv [iff]: (xs @ ys = xs) = (ys = [])
by (induct xs) auto

lemma self-append-conv [iff]: (xs = xs @ ys) = (ys = [])
by (induct xs) auto

lemma append-eq-append-conv [simp];
length xs = length ys ∨ length us = length vs
===> (xs@us = ys@vs) = (xs=ys ∧ us=vs)
apply (induct xs arbitrary: ys)
apply (case-tac ys, simp, force)
apply (case-tac ys, force, simp)
done

lemma append-eq-append-conv2: (xs @ ys = zs @ ts) =
  (EX us. xs = zs @ us & us @ ys = ts | xs @ us = zs & ys = us@ ts)
apply (induct xs arbitrary: ys zs ts)
apply fastforce
apply(case-tac zs)
apply simp
apply fastforce
done

lemma same-append-eq [iff, induct-simp]: (xs @ ys = xs @ zs) = (ys = zs)
by simp

lemma append1-eq-conv [iff]: (xs @ [x] = ys @ [y]) = (xs = ys ∧ x = y)
by \textit{simp}

\textbf{lemma} \textit{append-same-eq} ([\textit{iff}, \textit{induct-simp}]: \((ys @ xs = zs @ xs) = (ys = zs)\)
by \textit{simp}

\textbf{lemma} \textit{append-self-conv2} ([\textit{iff}]: \((xs @ ys = ys) = (xs = [])\)
using \textit{append-same-eq} [of - - []] by \textit{auto}

\textbf{lemma} \textit{self-append-conv2} ([\textit{iff}]: \((ys = xs @ ys) = (xs = [])\)
using \textit{append-same-eq} [of []] by \textit{auto}

\textbf{lemma} \textit{hd-Cons-tl} ([\textit{simp}]: \(xs \neq [] \Rightarrow \text{hd} \; xs \# \text{tl} \; xs = xs\)
\textbf{by} (\textit{induct xs}) \textit{auto}

\textbf{lemma} \textit{hd-append}: \(\text{hd} (xs @ ys) = (if \; xs = [] \; \text{then} \; \text{hd} \; ys \; \text{else} \; \text{hd} \; xs)\)
\textbf{by} (\textit{induct xs}) \textit{auto}

\textbf{lemma} \textit{hd-append2} ([\textit{simp}]: \(xs \neq [] \Rightarrow \text{hd} \; (xs @ ys) = \text{hd} \; xs\)
\textbf{by} (\textit{simp add}: \textit{hd-append split: list.split})

\textbf{lemma} \textit{tl-append}: \(\text{tl} (xs @ ys) = (case \; xs \; of \; [] = > \; \text{tl} \; ys \mid z#zs = > zs @ ys)\)
\textbf{by} (\textit{simp split: list.split})

\textbf{lemma} \textit{tl-append2} ([\textit{simp}]: \(xs \neq [] \Rightarrow \text{tl} \; (xs @ ys) = \text{tl} \; xs @ ys\)
\textbf{by} (\textit{simp add}: \textit{tl-append split: list.split})

\textbf{lemma} \textit{Cons-eq-append-conv}: \(x#xs = ys@zs = \)
\((ys = [] \& x#xs = zs \mid (EX \; ys' \; x#ys' = ys \& xs = ys'@zs))\)
\textbf{by}(\textit{cases ys}) \textit{auto}

\textbf{lemma} \textit{append-eq-Cons-conv}: \((ys@zs = x#xs) = \)
\((ys = [] \& zs = x#xs \mid (EX \; ys' \; ys = x#ys' \& ys'@zs = xs))\)
\textbf{by}(\textit{cases ys}) \textit{auto}

Trivial rules for solving @-equations automatically.

\textbf{lemma} \textit{eq-Nil-appendI}: \(xs = ys ==> \text{xs = [] @ ys}\)
\textbf{by} \textit{simp}

\textbf{lemma} \textit{Cons-eq-appendI}: \(\text{[]} \; x \; # \; xs1 = ys; \; zs = xs1 @ zs \; \text{[]} \Rightarrow x \; # \; xs = ys @ zs\)
\textbf{by} (\textit{drule sym}) \textit{simp}

\textbf{lemma} \textit{append-eq-appendI}: \(\text{[]} \; xs @ xs1 = zs; \; ys = xs1 @ us \; \text{[]} \Rightarrow xs @ ys = zs @ us\)
\textbf{by} (\textit{drule sym}) \textit{simp}

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the
same list.

\begin{verbatim}

  simproc-setup list-eq ((xs::'a list) = ys) = ⟨
  let
    fun last (cons as Const (@{const-name Cons}, -) $ - $ xs) =
      (case xs of Const (@{const-name Nil}, -) => cons | - => last xs)
    | last (Const(@{const-name append},-)) $ - $ ys) = last ys
    | last t = t;

    fun list1 (Const(@{const-name Cons},-) $ - $ Const(@{const-name Nil},-)) = true
    | list1 = false;

    fun butlast ((cons as Const(@{const-name Cons},-) $ x) $ xs) =
      (case xs of Const (@{const-name Nil}, -) => xs | - => cons $ butlast xs)
    | butlast ((app as Const (@{const-name append},-) $ xs) $ ys) = app $ butlast ys
    | butlast xs = Const(@{const-name Nil}, fastype-of xs);

    val rearr-ss = simpset-of (put-simpset HOL-basic-ss @{context}
      addsimps [@{thm append-assoc}, @{thm append-Nil}, @{thm append-Cons}]);

    fun list-eq ctxt (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
      let
        val lastl = last lhs and lastr = last rhs;
        fun rearr conv =
          let
            val lhs1 = butlast lhs and rhs1 = butlast rhs;
            val Type(-,listT::-) = eqT
            val appT = [listT,listT] ---> listT
            val app = Const(@{const-name append},appT)
            val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
            val eq = HOLogic.mk_Trueprop (HOLogic.mk_eq (F,F2));
            val thm = Goal.prove ctxt [] [] eq
              (K (simp-tac (put-simpset rearr-ss ctxt) 1));
            in SOME ((conv RS (ths RS trans)) RS eq-reflection) end;
          in
            if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
            else if list1 aconv lastr then rearr @{thm append-same-eq}
            else NONE
          end;
        in fn - => fn ctxt => fn ct => list-eq ctxt (term-of ct) end;
      end
  end


end

\end{verbatim}

67.1.5  \textit{map}

\textbf{lemma} hd-map:

\[xs \neq [] \implies \text{hd} (\text{map} f xs) = f (\text{hd} xs)\]

\textbf{by} (cases \(xs\)) simp-all
lemma map-tl:
map f (tl xs) = tl (map f xs)
by (cases xs) simp-all

lemma map-ext: (!!!x. x : set xs --> f x = g x) ==> map f xs = map g xs
by (induct xs) simp-all

lemma map-ident [simp]: map (λx. x) = (λxs. xs)
by (rule ext, induct-tac xs) auto

lemma map-append [simp]: map f (xs @ ys) = map f xs @ map f ys
by (induct xs) auto

lemma map-map [simp]: map f (map g xs) = map (f o g) xs
by (induct xs) auto

lemma map-comp-map [simp]: (map f o map g) = map (f o g)
apply (rule ext)
apply (simp)
done

lemma rev-map: rev (map f xs) = map f (rev xs)
by (induct xs) auto

lemma map-eq-conv [simp]: (map f xs = map g xs) = (!x : set xs. f x = g x)
by (induct xs) auto

lemma map-cong [fundef-cong]:
xs = ys ==> (∀x. x ∈ set ys ==> f x = g x) ==> map f xs = map g ys
by simp

lemma map-is-Nil-conv [iff]: (map f xs = []) = (xs = [])
by (cases xs) auto

lemma Nil-is-map-conv [iff]: ([] = map f xs) = (xs = [])
by (cases ys) auto

lemma map-eq-Cons-conv:
(map f xs = y#ys) = (∃z zs. xs = z#zs ℓ f z = y ℓ map f zs = ys)
by (cases xs) auto

lemma Cons-eq-map-conv:
(x#xs = map f ys) = (∃z zs. ys = z#zs ℓ x = f z ℓ xs = map f zs)
by (cases ys) auto

lemmas map-eq-Cons-D = map-eq-Cons-conv [THEN iffD1]
lemmas Cons-eq-map-D = Cons-eq-map-conv [THEN iffD1]
declare map-eq-Cons-D [dest!] Cons-eq-map-D [dest!]
lemma ex-map-conv:
(EXPR xs. ys = map f xs) = (ALL y : set ys. EX x. y = f x)
by (induct ys, auto simp add: Cons-eq-map-conv)

lemma map-eq-imp-length-eq:
assumes map f xs = map g ys
shows length xs = length ys
using assms
proof (induct ys arbitrary: xs)
case Nil then show ?case by simp
next
case (Cons y ys) then obtain z zs where xs: xs = z # zs by auto
from Cons xs have map f zs = map g ys by simp
with Cons have length zs = length ys by blast
with xs show ?case by simp
qed

lemma map-inj-on:
\[
\begin{align*}
| \ & map f xs = map f ys; inj-on f (set xs Un set ys) | \\
\Rightarrow & xs = ys \\
\end{align*}
\]
apply (frule map-eq-imp-length-eq)
apply (rotate-tac -1)
apply (induct rule: list-induct2)
apply simp
apply simp
apply (blast intro: sym)
done

lemma inj-on-map-eq-map:
inj-on f (set xs Un set ys) \Rightarrow (map f xs = map f ys) = (xs = ys)
by (blast dest: map-inj-on)

lemma map-injective:
map f xs = map f ys \Rightarrow inj f \Rightarrow xs = ys
by (induct ys arbitrary: xs) (auto dest!: injD)

lemma inj-map-eq-map[simp]: inj f \Rightarrow (map f xs = map f ys) = (xs = ys)
by (blast dest: map-injective)

lemma inj-mapI: inj f \Rightarrow inj (map f)
by (iprover dest: map-injective injD intro: inj-onI)

lemma inj-mapD: inj (map f) \Rightarrow inj f
apply (unfold inj-on-def, clarify)
apply (erule-tac x = [x] in ballE)
apply (erule-tac x = [y] in ballE, simp, blast)
apply blast
done
lemma inj-map \[iff\]: inj (map f) = inj f
by (blast dest: inj-mapD intro: inj-mapI)

lemma inj-on-mapI: inj-on f (\(\bigcup (\text{set } A)\)) \(\implies\) inj-on (map f) A
apply (rule inj-onI)
apply (erule map-inj-on)
apply (blast intro: inj-onI dest: inj-onD)
done

lemma map-idI: \(\forall x. x \in \text{set } xs \implies f x = x\) \(\implies\) map f xs = xs
by (induct xs, auto)

lemma map-fun-upd [simp]: \(y \notin \text{set } xs\) \(\implies\) map (f(y:=v)) xs = map f xs
by (induct xs) auto

lemma map-fst-zip [simp]:
  length xs = length ys \(\implies\) map fst (zip xs ys) = xs
by (induct rule: list-induct2, simp-all)

lemma map-snd-zip [simp]:
  length xs = length ys \(\implies\) map snd (zip xs ys) = ys
by (induct rule: list-induct2, simp-all)

functor map: map
by (simp-all add: id-def)

declare map.id [simp]

67.1.6 \(\text{rev}\)

lemma rev-append [simp]: \(\text{rev } (xs @ ys) = \text{rev } ys @ \text{rev } xs\)
by (induct xs) auto

lemma rev-rev-ident [simp]: \(\text{rev } (\text{rev } xs) = xs\)
by (induct xs) auto

lemma rev-swap: \(\text{rev } (xs = ys) = (xs = \text{rev } ys)\)
by auto

lemma rev-is-Nil-conv [iff]: \(\text{rev } xs = []\) \(\implies\) \(\text{rev } xs = []\)
by (induct xs) auto

lemma Nil-is-rev-conv [iff]: \([], \text{rev } xs) = (xs = []\)
by (induct xs) auto

lemma rev-singleton-conv [simp]: \(\text{rev } xs = [x]\) \(\implies\) \(\text{rev } xs = [x]\)
by (cases xs) auto
lemma singleton-rev-conv [simp]: ([x] = rev xs) = (xs = [x])
by (cases xs) auto

lemma rev-is-rev-conv [iff]: (rev xs = rev ys) = (xs = ys)
apply (induct xs arbitrary: ys, force)
apply (case-tac ys, simp, force)
done

lemma inj-on-rev [iff]: inj-on rev A
by (simp add: inj-on-def)

lemma rev-induct [case-names Nil snoc]:
| P [] | !!xs. P xs ===> P (xs @ [x]) [] ===> P xs
apply (simplesubst rev-rev-ident [symmetric])
apply (rule-tac list = rev xs in list.induct, simp-all)
done

lemma rev-exhaust [case-names Nil snoc]:
(xs = [] ===> P) ===> (!!ys y. xs = ys @ [y] ===> P) ===> P
by (induct xs rule: rev-induct) auto

lemmas rev-cases = rev-exhaust

lemma rev-nonempty-induct [consumes 1, case-names single snoc]:
  assumes xs ≠ []
  and single: ∀x. P [x]
  and snoc': ∀xs. xs ≠ [] ==> P xs ==> P (xs@[x])
  shows P xs
using (xs ≠ []) proof (induct xs rule: rev-induct)
case (snoc x xs) then show ?case
proof (cases xs)
  case Nil thus ?thesis by (simp add: single)
next
case Cons with snoc show ?thesis by (fastforce intro!: snoc')
qed

lemma rev-eq-Cons-iff [iff]: (rev xs = y#ys) = (xs = rev ys @ [y])
by (rule rev-cases[of xs]) auto

67.1.7  set

lemma finite-set [iff]: finite (set xs)
by (induct xs) auto

lemma set-append [simp]: set (xs @ ys) = (set xs ∪ set ys)
by (induct xs) auto

lemma hd-in-set[simp]: xs ≠ [] ==> hd xs : set xs
by (cases xs) auto

lemma set-subset-Cons: set xs ⊆ set (x # xs)
by auto

lemma set-ConsD: y ∈ set (x # xs) ⇒ y = x ∨ y ∈ set xs
by auto

lemma set-empty [iff]: (set xs = {}) = (xs = [])
by (induct xs) auto

lemma set-empty2 [iff]: (xs = {}) = (set xs = [])
by (induct xs) auto

lemma set-rev [simp]: set (rev xs) = set xs
by (induct xs) auto

lemma set-map [simp]: set (map f xs) = f' (set xs)
by (induct xs) auto

lemma set-filter [simp]: set (filter P xs) = {x. x : set xs ∧ P x}
by (induct xs) auto

lemma set-upt [simp]: set[i..<j] = {i..<j}
by (induct j) auto

lemma split-list: x : set xs ⇒ ∃ ys zs. xs = ys @ x # zs
proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case by (auto intro: Cons-eq-appendI)
qed

lemma in-set-conv-decomp: x ∈ set xs ⟷ (∃ ys zs. xs = ys @ x # zs)
by (auto elim: split-list)

lemma split-list-first: x : set xs ⇒ ∃ ys zs. xs = ys @ x # zs ∧ x ∉ set ys
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons a xs)
  show ?case
  proof cases
    assume x = a thus ?case using Cons by fastforce
  next
    assume x ≠ a thus ?case using Cons by (fastforce intro!: Cons-eq-appendI)
  qed
qed

qed
lemma in-set-decomp-first:
\( (x : set xs) = (\exists ys zs. xs = ys @ x \# zs \land x \notin set ys) \)
by (auto dest!: split-list-first)

lemma split-list-last: \( x \in set xs \Longrightarrow \exists ys zs. xs = ys @ x \# zs \land x \notin set zs \)
proof (induct xs rule: rev-induct)
  case Nil thus ?case by simp
next
  case (snoc a xs)
  show ?case
  proof cases
    assume \( x = a \) thus ?case using snoc by (auto intro!: exI)
  next
    assume \( x \neq a \) thus ?case using snoc by fastforce
  qed
qed

lemma in-set-decomp-last:
\( (x : set xs) = (\exists ys zs. xs = ys @ x \# zs \land x \notin set zs) \)
by (auto dest!: split-list-last)

lemma split-list-prop:
\( \exists x \in set xs. P x \Longrightarrow \exists ys x zs. xs = ys @ x \# zs \land P x \)
proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case
  by (simp add: Bex_def) (metis append Cons append.simps (1))
qed

lemma split-list-propE:
assumes \( \exists x \in set xs. P x \)
obtains \( ys x zs \) where \( xs = ys @ x \# zs \land P x \)
using split-list-prop [OF assms] by blast

lemma split-list-first-prop:
\( \exists x \in set xs. P x \Longrightarrow \exists ys x zs. xs = ys @ x \# zs \land P x \land (\forall y \in set ys. \neg P y) \)
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  show ?case
  proof cases
    assume \( P x \)
    hence \( x \# xs = [] @ x \# xs \land P x \land (\forall y \in set[]. \neg P y) \) by simp
    thus ?thesis by fast
  next
    assume \( \neg P x \)
hence $\exists x \in \text{set } xs. \ P x$ using $\text{Cons}(2)$ by simp
thus $?thesis$ using $(\neg P x)$ $\text{Cons}(1)$ by (metis append-$\text{Cons}$ set-$\text{Cons}$D)
qed

lemma split-list-first-propE:
assumes $\exists x \in \text{set } xs. \ P x$
obegin{isabelle}obtains ys x zs where $xs = ys \# x \# zs$ and $P x$ and $\forall y \in \text{set } ys. \neg P y$
using split-list-first-prop [OF assms] by blast

lemma split-list-first-prop-iff:
$(\exists x \in \text{set } xs. \ P x) \iff (\exists ys x zs. \ xs = ys \# x \# zs \land P x \land (\forall y \in \text{set } ys. \neg P y))$
by (rule, erule split-list-first-prop) auto

lemma split-list-last-prop:
$\exists x \in \text{set } xs. \ P x \implies \exists ys x zs. \ xs = ys \# x \# zs \land P x \land (\forall z \in \text{set } zs. \neg P z)$
proof (induct xs rule: rev-induct)
case Nil thus $?case$ by simp
next
case (snoc x xs)
show $?case$
proof cases
  assume $P x$ thus $?thesis$ by (auto intro: exI)
next
  assume $\neg P x$
hence $\exists x \in \text{set } xs. \ P x$ using snoc [2]
thus $?thesis$ using $(\neg P x)$ snoc [1] by fastforce
qed

lemma split-list-last-propE:
assumes $\exists x \in \text{set } xs. \ P x$
obegin{isabelle}obtains ys x zs where $xs = ys \# x \# zs$ and $P x$ and $\forall z \in \text{set } zs. \neg P z$
using split-list-last-prop [OF assms] by blast

lemma split-list-last-prop-iff:
$\exists x \in \text{set } xs. \ P x \iff \exists ys x zs. \ xs = ys \# x \# zs \land P x \land (\forall z \in \text{set } zs. \neg P z)$
by (rule, erule split-list-last-prop, auto)

lemma finite-list: finite $A \implies \text{EX } xs. \ \text{set } xs = A$
by (erule finite-induct) (auto simp add: set-simps(2) [symmetric] simp del: set-simps(2))

lemma card-length: card $\text{(set } xs)$ $\leq$ length $xs$
by (induct xs) (auto simp add: card-insert-if)
lemma set-minus-filter-out:  
set xs - \{ y \} = set (filter (\lambda x. \neg (x = y)) xs)  
by (induct xs) auto

67.1.8  filter  
lemma filter-append [simp]: filter P (xs @ ys) = filter P xs @ filter P ys  
by (induct xs) auto

lemma rev-filter: rev (filter P xs) = filter P (rev xs)  
by (induct xs) simp-all

lemma filter-filter [simp]: filter P (filter Q xs) = filter (\lambda x. Q x \land P x) xs  
by (induct xs) auto

lemma length-filter-le [simp]: length (filter P xs) \leq length xs  
by (induct xs) (auto simp add: le-SucI)

lemma sum-length-filter-compl:  
length(filter P xs) + length(filter (%x. \neg P x) xs) = length xs  
by (induct xs) simp-all

lemma filter-True [simp]: \forall x \in set xs. P x ==> filter P xs = xs  
by (induct xs) auto

lemma filter-False [simp]: \forall x \in set xs. \neg P x ==> filter P xs = []  
by (induct xs) auto

lemma filter-empty-conv: (filter P xs = []) = (\forall x \in set xs. \neg P x)  
by (induct xs) simp-all

lemma filter-id-conv: (filter P xs = xs) = (\forall x \in set xs. P x)  
apply (induct xs)  
apply auto  
apply (cut-tac P=P and xs=xs in length-filter-le)  
apply simp  
done

lemma filter-map:  
filter P (map f xs) = map f (filter (P o f) xs)  
by (induct xs) simp-all

lemma length-filter-map[simp]:  
length (filter P (map f xs)) = length(filter (P o f) xs)  
by (simp add:filter-map)

lemma filter-is-subset [simp]: set (filter P xs) \leq set xs  
by auto
lemma length-filter-less:
[ \[ x : \text{set } xs; \sim P x \] ] \implies \text{length(filter } P \text{ } xs) < \text{length } xs
proof (induct xs)
  case Nil thus \case by simp
next
  case (Cons x xs) thus \case
    apply (auto split:split-if-asm)
    using length-filter-le[of P xs] apply arith
  done
qed

lemma length-filter-conv-card:
\text{length(filter } p \text{ } xs) = \text{card}\{ i. i < \text{length } xs \& p(xs!i)\}
proof (induct xs)
  case Nil thus \case by simp
next
  case (Cons x xs)
    let \?S = \{ i. i < \text{length } xs \& p(xs!i)\}
    have fin: finite \?S by (fast intro: bounded-nat-set-is-finite)
    show \case (is \?l = \text{card } \?S')
      proof (cases)
        assume p x
        hence eq: \?S' = insert 0 (Suc ' \?S)
          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
        have length (filter p (x # xs)) = Suc(\text{card } \?S)
          using Cons ⟨p x⟩ by simp
        also have \ldots = Suc(\text{card}(Suc ' \?S)) using fin
          by (simp add: card-image)
        also have \ldots = card \?S' using eq fin
          by (simp add:card-insert-if) (simp add:image-def)
      finally show \?thesis .
next
  assume \sim p x
  hence eq: \?S' = Suc ' \?S
    by (auto simp add: image-def split:nat.split elim:lessE)
  have length (filter p (x # xs)) = card \?S
    using Cons ⟨\sim p x⟩ by simp
  also have \ldots = card(Suc ' \?S) using fin
    by (simp add: card-image)
  also have \ldots = card \?S' using eq fin
    by (simp add:card-insert-if)
  finally show \?thesis .
  qed
qed

lemma Cons-eq-filterD:
x#xs = filter P ys \implies \exists us vs ys = us @ x # vs \& (\forall u \in \text{set } us. \sim P u) \& P x \& xs = filter P vs
(is - \implies \exists us vs \ P ys us vs)
proof (induct ys)
case Nil thus \ ?case by simp
next
case (Cons y ys)
show \ ?case (is \ \exists x \ \ ?Q x)
proof cases
assume Py: P y
show \ ?thesis
proof cases
assume \ x = y
with Py Cons.prems have \ ?Q [] by simp
then show \ ?thesis ..
next
assume \ x \neq y
with Py Cons.prems show \ ?thesis by simp
qed
next
assume \ \neg P y
with Cons obtain us vs where \ ?P \ (y\#ys) \ (y\#us) \ vs by fastforce
then have \ ?Q \ (y\#us) by simp
then show \ ?thesis ..
qed
qed

lemma filter-eq-ConsD:
filter P ys = x\#xs \ \implies \ \exists us vs \ ys = us \@ x \# vs \ \land \ \( \forall u \in \text{set} \ us. \ \neg P u \) \ \land \ P x \ \land \ xs = filter P vs
by (rule Cons-eq-filterD) simp

lemma filter-eq-Cons-iff:
(filter P ys = x\#xs) =
(\exists us vs. \ ys = us \@ x \# vs \ \land \ \( \forall u \in \text{set} \ us. \ \neg P u \) \ \land \ P x \ \land \ xs = filter P vs)
by (auto dest: filter-eq-ConsD)

lemma Cons-eq-filter-iff:
x\#xs = filter P ys =
(\exists us vs. \ ys = us \@ x \# vs \ \land \ \( \forall u \in \text{set} \ us. \ \neg P u \) \ \land \ P x \ \land \ xs = filter P vs)
by (auto dest: Cons-eq-filterD)

lemma filter-cong [fundef-cong]:
xs = ys \implies (\forall x. x \in \text{set} \ ys \implies P x = Q x) \implies filter P xs = filter Q ys
apply simp
apply (erule thin-rl)
by (induct ys) simp-all

67.1.9 List partitioning

primrec partition :: (\ 'a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list \times 'a list where
partition $P$ $\emptyset = ([], [])$
\[
\text{partition } P \ (x \neq \# \, xs) =
\]
\[
\begin{aligned}
\text{let } (\text{yes}, \text{no}) &= \text{partition } P \, xs \\
in \text{ if } P \, x \text{ then } (x \neq \# \, \text{yes}, \text{no}) \text{ else } (\text{yes}, x \neq \# \, \text{no})
\end{aligned}
\]

lemma partition-filter1:
\[
\text{fst } (\text{partition } P \, xs) = \text{filter } P \, xs
\]
by (induct $xs$) (auto simp add: Let-def split-def)

lemma partition-filter2:
\[
\text{snd } (\text{partition } P \, xs) = \text{filter } (\text{Not } o \, P) \, xs
\]
by (induct $xs$) (auto simp add: Let-def split-def)

lemma partition-P:
\[
\text{assumes } \text{partition } P \, xs = (\text{yes}, \text{no}) \\
\text{shows } (\forall p \in \text{set yes. } P \, p) \land (\forall p \in \text{set no. } \neg P \, p)
\]
proof –
\[
\text{from } \text{assms have } \text{yes} = \text{fst } (\text{partition } P \, xs) \text{ and } \text{no} = \text{snd } (\text{partition } P \, xs)
\]
\[
\text{by simp-all}
\]
then show ?thesis by (simp-all add: partition-filter1 partition-filter2)
qed

lemma partition-set:
\[
\text{assumes } \text{partition } P \, xs = (\text{yes}, \text{no}) \\
\text{shows } \text{set yes} \cup \text{set no} = \text{set } xs
\]
proof –
\[
\text{from } \text{assms have } \text{yes} = \text{fst } (\text{partition } P \, xs) \text{ and } \text{no} = \text{snd } (\text{partition } P \, xs)
\]
\[
\text{by simp-all}
\]
then show ?thesis by (auto simp add: partition-filter1 partition-filter2)
qed

lemma partition-filter-conv[simp]:
\[
\text{partition } f \, xs = (\text{filter } f \, xs, \text{filter } (\text{Not } o \, f) \, xs)
\]
unfolding partition-filter2[symmetric]
unfolding partition-filter1[symmetric] by simp

declare partition.simps[simp del]

67.1.10 concat

lemma concat-append [simp]: concat $\ (xs \ @ \ ys) = \text{concat } xs \ @ \ \text{concat } ys$
by (induct $xs$) auto

lemma concat-eq-Nil-conv [simp]: (concat $xs \ = \ [] ) = (\forall xs \in \text{set } xss. \, xs = [])$
by (induct $xss$) auto

lemma Nil-eq-concat-conv [simp]: ([] = concat $xss ) = (\forall xs \in \text{set } xss. \, xs = [] )$
by (induct $xss$) auto
lemma set-concat [simp]: set (concat xs) = (UN x:set xs. set x)
by (induct xs) auto

lemma concat-map-singleton[simp]: concat(map (%x. [f x]) xs) = map f xs
by (induct xs) auto

lemma map-concat: map f (concat xs) = concat (map (map f) xs)
by (induct xs) auto

lemma filter-concat: filter p (concat xs) = concat (map (filter p) xs)
by (induct xs) auto

lemma rev-concat: rev (concat xs) = concat (map rev (rev xs))
by (induct xs) auto

lemma concat-eq-concat-iff: ∀ (x, y) ∈ set (zip xs ys). length x = length y ==> length xs = length ys ==> (concat xs = concat ys) = (xs = ys)
proof (induct xs arbitrary: ys)
case (Cons x xs ys)
  thus ?case by (cases ys) auto
qed (auto)

lemma concat-injective: concat xs = concat ys ==> length xs = length ys ==> ∀ (x, y) ∈ set (zip xs ys). length x = length y ==> xs = ys
by (simp add: concat-eq-concat-iff)

67.1.11 op !

lemma nth-Cons-0 [simp, code]: (x # xs)!0 = x
by auto

lemma nth-Cons-Suc [simp, code]: (x # xs)!(Suc n) = xs!n
by auto

declare nth.simps [simp del]

lemma nth-Cons-pos[simp]: 0 < n ==> (x#xs)!n = xs ! (n - 1)
by(auto simp: Nat.gr0_conv-Suc)

lemma nth-append:
  (xs @ ys)!n = (if n < length xs then xs!n else ys!(n - length xs))
apply (induct xs arbitrary: n, simp)
apply (case_tac n, auto)
done

lemma nth-append-length [simp]: (xs @ x # ys) ! length xs = x
by (induct xs) auto

lemma nth-append-length-plus[simp]: (xs @ ys) ! (length xs + n) = ys ! n
by (induct \(xs\)) auto

lemma nth-map \([simp]\): \(n < \text{length} \; xs \Rightarrow (\text{map} \; f \; xs)!n = f(xs!n)\)
apply (induct \(xs\) arbitrary: \(n\), simp)
apply (case-tac \(n\), auto)
done

lemma nth-tl:
  assumes \(n < \text{length} \; (\text{tl} \; x)\) shows \(\text{tl} \; x ! n = x ! \text{Suc} \; n\)
using assms by (induct \(x\)) auto

lemma hd-conv-nth: \(xs \neq \text{[]} \Rightarrow \text{hd} \; xs = xs!0\)
by (cases \(xs\)) simp-all

lemma list-eq-iff-nth-eq:
  \((xs = ys) = (\text{length} \; xs = \text{length} \; ys \land (\text{ALL} \; i < \text{length} \; xs. \; xs!i = ys!i))\)
apply (induct \(xs\) arbitrary: \(ys\))
apply force
apply (case-tac \(ys\))
apply simp
apply (simp add: nth-Cons split:nat.split) apply blast
done

lemma set-conv-nth: \(\text{set} \; xs = \{xs!i \mid i. \; i < \text{length} \; xs\}\)
apply (induct \(xs\), simp, simp)
apply safe
apply (metis \(\text{nat.case}(1)\) nth.simps zero-less-Suc)
apply (metis \(\text{less-Suc-eq-0-disj} \; \text{nth-Cons-Suc}\))
apply (case-tac \(i\), simp)
apply (metis \(\text{diff-Suc-Suc} \; \text{nat.case}(2)\) nth.simps zero-less-diff)
done

lemma in-set-conv-nth: \((x \in \text{set} \; xs) = (\exists \; i < \text{length} \; xs. \; xs!i = x)\)
by (auto simp: set-conv-nth)

lemma nth-equal-first-eq:
  assumes \(x \notin \text{set} \; xs\)
  assumes \(n \leq \text{length} \; xs\)
  shows \((x \# \; xs) ! n = x \longleftrightarrow n = 0\) (is \(\text{lhs} \longleftrightarrow \text{rhs}\))
proof
  assume \(\text{lhs}\)
  show \(\text{rhs}\)
  proof (rule ccontr)
    assume \(n \neq 0\)
    then have \(n > 0\) by simp
    with \(\text{lhs}\) have \(xs ! (n - 1) = x\) by simp
    moreover from \(n > 0\), \(n \leq \text{length} \; xs\) have \(n - 1 < \text{length} \; xs\) by simp
    ultimately have \(\exists \; i < \text{length} \; xs. \; xs ! i = x\) by auto
with \( x \notin \text{set } xs \) in-set-conv-nth [of \( xs \)] show False by simp

qed

next

assume \(?rhs\) then show \(?lhs\) by simp

qed

lemma nth-non-equal-first-eq:
assumes \( x \neq y \)
shows \((x \# xs) ! n = y \leftrightarrow xs ! (n - 1) = y \land n > 0 \) (is \(?lhs \leftrightarrow ?rhs\))

proof
assume \(?lhs\) with \(\text{assms}\) have \(n > 0\) by (cases \(n\)) simp-all

with \(?lhs\) show \(?rhs\) by simp

next

assume \(?rhs\) then show \(?lhs\) by simp

qed

lemma list-ball-nth: \([| n < \text{length } xs; \! x : \text{set } xs. \ P x ||] \Longrightarrow P(xs!n)\)
by (auto simp add: set-conv-nth)

lemma nth-mem [simp]: \(n < \text{length } xs \Longrightarrow xs!n : \text{set } xs\)
by (auto simp add: set-conv-nth)

lemma all-nth-imp-all-set:
\([| \! i < \text{length } xs. \ P(xs!i); \ x : \text{set } xs ||] \Longrightarrow P x\)
by (auto simp add: set-conv-nth)

lemma all-set-conv-all-nth:
\((\forall x \in \text{set } xs. \ P x) = (\forall i. i < \text{length } xs \Longrightarrow P (xs ! i))\)
by (auto simp add: set-conv-nth)

lemma rev-nth:
\(n < \text{size } xs \Longrightarrow \text{rev } xs ! n = xs ! (\text{length } xs - \text{Suc } n)\)

proof (induct \(xs\) arbitrary: \(n\))

case Nil thus \(?case\) by simp

next
case (Cons \(x\) \(xs\))
hence \(n: n < \text{Suc } (\text{length } xs)\) by simp

moreover
{
assume \(n < \text{length } xs\)

with \(n\) obtain \(n'\) where \(n': \text{length } xs - n = \text{Suc } n'\)

by (cases \text{length } xs - n, auto)

moreover

from \(n'\) have \(\text{length } xs - \text{Suc } n = n'\) by simp

ultimately

have \(xs ! (\text{length } xs - \text{Suc } n) = (x \# xs) ! (\text{length } xs - n)\) by simp
}

ultimately

show \(?case\) by (clarsimp simp add: Cons nth-append)

qed
lemma Skolem-list-nth:
(\all\ i<k. \EX\ x. P\ i\ x) = (\EX\ xs. \all\ i<k. P\ i\ (xs\!\!i))
(is - = (\EX\ xs. \?P\ k\ xs))
proof(induct k)
case 0 show ?case by simp
next
case (Suc k)
show ?case (is \?L = \?R is - = (\EX\ xs. \?P'\ xs))
proof
  assume \?R thus \?L using Suc by auto
next
  assume \?L with Suc obtain x xs where \?P k xs & \?P'\ (xs\!\!0[x]) by (metis less-Suc-eq)
hence \?P'(xs\!\!\![i]) by(simp add: nth-append less-Suc-eq)
  thus \?R ..
qed
qed

67.1.12 list-update

lemma length-list-update [simp]: length(xs[i:=x]) = length xs
by (induct xs arbitrary: i) (auto split: nat.split)

lemma nth-list-update:
i < length xs=\Rightarrow (xs[i:=x])!j = (if i = j then x else xs\!\!j)
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma nth-list-update-eq [simp]: i < length xs =\Rightarrow (xs[i:=x])!i = x
by (simp add: nth-list-update)

lemma nth-list-update-neq [simp]: i \neq j =\Rightarrow xs[i:=x]!j = xs\!\!j
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma list-update-id [simp]: xs[i := x]!i = x
by (induct xs arbitrary: i) (simp-all split:nat.splits)

lemma list-update-beyond [simp]: length xs \leq i \Rightarrow xs[i:=x] = xs
apply (induct xs arbitrary: i)
apply simp
apply (case-tac i)
apply simp-all
done

lemma list-update-nonempty [simp]: xs[k:=x] = [] \iff xs=x=
by (simp only: length-0-conv[symmetric] length-list-update)

lemma list-update-same-conv:
i < length xs =\Rightarrow (xs[i := x] = xs) = (xs\!\!i = x)
THEORY "List"

by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append1:
i < size xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys
apply (induct xs arbitrary: i, simp)
apply(simp split:nat.split)
done

lemma list-update-append:
(xs @ ys)[n:=x] =
(if n < length xs then xs[n:=x] @ ys else xs @ (ys [n-length xs:= x]))
by (induct xs arbitrary: n) (auto split:nat.splits)

lemma list-update-length [simp]:
(xs @ x ≠ ys)[length xs := y] = (xs @ y ≠ ys)
by (induct xs, auto)

lemma map-update: map f (xs[k:= y]) = (map f xs)[k := f y]
by(induct xs arbitrary: k)(auto split:nat.splits)

lemma rev-update:
k < length xs \implies rev (xs[k:= y]) = (rev xs)[length xs - k - l := y]
by (induct xs arbitrary: k) (auto simp: list-update-append split:nat.splits)

lemma update-zip:
(zip xs ys)[i:=xy] = zip (xs[i:=fst xy]) (ys[i:=snd xy])
by (induct ys arbitrary: i xy xs) (auto, case-tac xs, auto split: nat.splits)

lemma set-update-subset-insert: set(xs[i:=x]) <= insert x (set xs)
by (induct xs arbitrary: i) (auto split: nat.split)

lemma set-update-subsetI: [] set xs <= A; x:A [] \implies set(xs[i := x]) <= A
by (blast dest!: set-update-subset-insert [THEN subsetD])

lemma set-update-memI: n < length xs \implies x ∈ set (xs[n := x])
by (induct xs arbitrary: n) (auto split:nat.splits)

lemma list-update-overwrite[simp]:
xs [i := x, i := y] = xs [i := y]
apply (induct xs arbitrary: i) apply simp
apply (case-tac i, simp-all)
done

lemma list-update-swap:
i ≠ i' \implies xs [i := x, i' := x'] = xs [i' := x', i := x]
apply (induct xs arbitrary: i i')
apply simp
apply (case-tac i, case-tac i')
apply auto
THEORY "List"

apply (case-tac i')
apply auto
done

lemma list-update-code [code]:
\[
[t][i := y] = []
\]
\[
(x \# xs)[0 := y] = y \# xs
\]
\[
(x \# xs)[Suc i := y] = x \# xs[i := y]
\]
by simp-all

67.1.13 last and butlast
lemma last-snoc [simp]: last (xs @ [x]) = x
by (induct xs) auto

lemma butlast-snoc [simp]: butlast (xs @ [x]) = xs
by (induct xs) auto

lemma last-ConsL: xs = [] \implies last(x\#xs) = x
by simp

lemma last-ConsR: xs \neq [] \implies last(x\#xs) = last xs
by simp

lemma last-append: last(xs @ ys) = (if ys = [] then last xs else last ys)
by (induct xs) (auto)

lemma last-appendL[simp]: ys = [] \implies last(xs @ ys) = last xs
by(simp add:last-append)

lemma last-appendR[simp]: ys \neq [] \implies last(xs @ ys) = last ys
by(simp add:last-append)

lemma last-tl: xs = [] \lor tl xs \neq [] \implies last (tl xs) = last xs
by (induct xs) simp-all

lemma butlast-tl: butlast (tl xs) = tl (butlast xs)
by (induct xs) simp-all

lemma hd-rev: xs \neq [] \implies hd(rev xs) = last xs
by(rule rev-exhaust[of xs]) simp-all

lemma last-rev: xs \neq [] \implies last(rev xs) = hd xs
by(cases xs) simp-all

lemma last-in-set[simp]: as \neq [] \implies last as \in set as
by (induct as) auto

lemma length-butlast [simp]: length (butlast xs) = length xs - 1
by (induct xs rule: rev-induct) auto

lemma butlast-append:
  butlast (xs @ ys) = (if ys = [] then butlast xs else xs @ butlast ys)
by (induct xs arbitrary: ys) auto

lemma append-butlast-last-id [simp]:
  xs ≠ [] ==> butlast xs @ [last xs] = xs
by (induct xs) auto

lemma in-set-butlastD: x : set (butlast xs) ==> x : set xs
by (induct xs) (auto split: split-if-asm)

lemma nth-butlast:
  assumes n < length (butlast xs) shows butlast xs ! n = xs ! n
proof (cases xs)
  case (Cons y ys)
  moreover from assms have butlast xs ! n = (butlast xs @ [last xs]) ! n
  by (simp add: nth-append)
  ultimately show ?thesis using append-butlast-last-id by simp
qed simp

lemma last-drop[simp]: n < length xs ==> last (drop n xs) = last xs
apply (induct xs arbitrary: n)
  apply simp
  apply (auto split: nat.split)
done

lemma butlast-conv-take:
  butlast xs = take (length xs - 1) xs
by (induct xs, simp, case-tac xs, simp-all)

lemma last-list-update:
  xs ≠ [] ==> last(xs[k:=x]) = (if k = size xs - 1 then x else last xs)
by (auto simp: last-list-update)

lemma butlast-list-update:
  butlast(xs[k:=x]) =
  (if k = size xs - 1 then butlast xs else (butlast xs)[k:=x])
apply (cases xs rule: rev-cases)
apply simp
apply (simp add: list-update-append split: nat.splits)
done
lemma last-map:  
xs \neq [] \implies \text{last} \ (\text{map} \ f \ xs) = f \ (\text{last} \ xs)  
by (cases \ xs \ rule: \ rev-cases) \ simp-all

lemma map-butlast:  
\text{map} \ f \ (\text{butlast} \ xs) = \text{butlast} \ (\text{map} \ f \ xs)  
by (induct \ xs) \ simp-all

lemma snoc-eq-iff-butlast:  
xs @ [x] = ys \iff (ys \neq [] \& \text{butlast} \ ys = xs \& \text{last} \ ys = x)  
by fastforce

67.1.14 take and drop  
lemma take-0 [simp]: take 0 xs = []  
by (induct \ xs) \ auto

lemma drop-0 [simp]: drop 0 xs = xs  
by (induct \ xs) \ auto

lemma take-Suc-Cons [simp]: take (Suc \ n) \ (x \# \ xs) = x \# \ take \ n \ xs  
by simp

lemma drop-Suc-Cons [simp]: drop (Suc \ n) \ (x \# \ xs) = drop \ n \ xs  
by simp

declare take-Cons [simp del] and drop-Cons [simp del]

lemma take-1-Cons [simp]: take 1 \ (x \# \ xs) = [x]  
unfolding One-nat-def by simp

lemma drop-1-Cons [simp]: drop 1 \ (x \# \ xs) = xs  
unfolding One-nat-def by simp

lemma take-Suc: xs \sim= [] \implies take (Suc \ n) \ xs = \text{hd} \ xs \# \ take \ n \ (\text{tl} \ xs)  
by(clarsimp simp add:neq-Nil-conv)

lemma drop-Suc: drop (Suc \ n) \ xs = drop \ n \ (\text{tl} \ xs)  
by(cases \ xs, simp-all)

lemma take-tl: take \ n \ (\text{tl} \ xs) = \text{tl} \ (\text{take} \ (Suc \ n) \ xs)  
by (induct \ xs \ arbitrary: \ n) \ simp-all

lemma drop-tl: drop \ n \ (\text{tl} \ xs) = \text{tl} \ (\text{drop} \ n \ xs)  
by(induct \ xs \ arbitrary: \ n, simp-all add:drop-Cons drop-Suc split:nat.split)

lemma tl-take: \text{tl} \ (\text{take} \ n \ xs) = \text{take} \ (n - 1) \ (\text{tl} \ xs)  
by (cases \ n, simp, cases \ xs, auto)
lemma tl-drop: tl (drop n xs) = drop n (tl xs)
by (simp only: drop-tl)

lemma nth-via-drop: drop n xs = y#ys \implies xs!n = y
apply (induct xs arbitrary: n, simp)
apply (simp add: drop-Cons nth-Cons split:nat.splits)
done

lemma take-Suc-conv-app-nth:
  i < length xs \implies take (Suc i) xs = take i xs @ [xs!i]
apply (induct xs arbitrary: i, simp)
apply (case-tac i, auto)
done

lemma drop-Suc-conv-tl:
  i < length xs \implies (xs!i) # (drop (Suc i) xs) = drop i xs
apply (induct xs arbitrary: i, simp)
apply (case-tac i, auto)
done

lemma length-take [simp]: length (take n xs) = min (length xs) n
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma length-drop [simp]: length (drop n xs) = (length xs - n)
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-all [simp]: length xs <= n \implies take n xs = xs
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma drop-all [simp]: length xs <= n \implies drop n xs = []
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-append [simp]:
take n (xs @ ys) = (take n xs @ take (n - length xs) ys)
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma drop-append [simp]:
drop n (xs @ ys) = drop n xs @ drop (n - length xs) ys
by (induct n arbitrary: xs) (auto, case-tac xs, auto)

lemma take-take [simp]: take n (take m xs) = take (min n m) xs
apply (induct m arbitrary: xs n, auto)
apply (case-tac xs, auto)
apply (case-tac n, auto)
done

lemma drop-drop [simp]: drop n (drop m xs) = drop (n + m) xs
apply (induct m arbitrary: xs, auto)
apply (case-tac xs, auto)
done

lemma take-drop: take n (drop m xs) = drop m (take (n + m) xs)
apply (induct m arbitrary: xs n, auto)
apply (case-tac xs, auto)
done

lemma drop-take: drop n (take m xs) = take (m − n) (drop n xs)
apply (induct xs arbitrary: m n)
apply simp
apply (simp add: take-Cons drop-Cons split:nat.split)
done

lemma append-take-drop-id [simp]: take n xs @ drop n xs = xs
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

lemma take-eq-Nil [simp]: (take n xs = []) = (n = 0 ∨ xs = [])
apply (induct xs arbitrary: n)
apply simp
apply (simp add: take-Cons split:nat.split)
done

lemma drop-eq-Nil [simp]: (drop n xs = []) = (length xs ≤ n)
apply (induct xs arbitrary: n)
apply simp
apply (simp add: drop-Cons split:nat.split)
done

lemma take-map: take n (map f xs) = map f (take n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

lemma drop-map: drop n (map f xs) = map f (drop n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

lemma rev-take: rev (take i xs) = drop (length xs − i) (rev xs)
apply (induct xs arbitrary: i, auto)
apply (case-tac i, auto)
done

lemma rev-drop: rev (drop i xs) = take (length xs − i) (rev xs)
apply (induct xs arbitrary: i, auto)
apply (case-tac i, auto)
done

lemma nth-take [simp]: \( i < n \implies (\text{take} \ n \ \text{xs})!i = \text{xs}!i \)
apply (induct xs arbitrary: i n, auto)
apply (case-tac n, blast)
apply (case-tac i, auto)
done

lemma nth-drop [simp]:
\( n + i \leq \text{length} \ \text{xs} \implies (\text{drop} \ n \ \text{xs})!i = \text{xs}!(n + i) \)
apply (induct n arbitrary: xs i, auto)
apply (case-tac xs, auto)
done

lemma butlast-take:
\( n < \text{length} \ \text{xs} \implies \text{butlast} \ (\text{take} \ n \ \text{xs}) = \text{take} \ (n - 1) \ \text{xs} \)
by (simp add: butlast-conv-take min.absorb1 min.absorb2)

lemma butlast-drop: \( \text{butlast} \ (\text{drop} \ n \ \text{xs}) = \text{drop} \ n \ (\text{butlast} \ \text{xs}) \)
by (simp add: butlast-conv-take drop-take ac-simps)

lemma take-butlast: \( n < \text{length} \ \text{xs} \implies \text{take} \ n \ (\text{butlast} \ \text{xs}) = \text{take} \ n \ \text{xs} \)
by (simp add: butlast-conv-take min.absorb1)

lemma drop-butlast: \( \text{drop} \ n \ (\text{butlast} \ \text{xs}) = \text{butlast} \ (\text{drop} \ n \ \text{xs}) \)
by (simp add: butlast-conv-take drop-take ac-simps)

lemma hd-drop-conv-nth:
\( n < \text{length} \ \text{xs} \implies \text{hd} \ (\text{drop} \ n \ \text{xs}) = \text{xs}!n \)
by (simp add: hd-conv-nth)

lemma set-take-subset-set-take:
\( m < n \implies \text{set} \ (\text{take} \ m \ \text{xs}) \subseteq \text{set} \ (\text{take} \ n \ \text{xs}) \)
apply (induct xs arbitrary: m n)
simp
apply (case-tac n)
apply (auto simp: take-Cons)
done

lemma set-take-subset: \( \text{set} \ (\text{take} \ n \ \text{xs}) \subseteq \text{set} \ \text{xs} \)
by (induct xs arbitrary: n) (auto simp: take-Cons split:nat.split)

lemma set-drop-subset: \( \text{set} \ (\text{drop} \ n \ \text{xs}) \subseteq \text{set} \ \text{xs} \)
by (induct xs arbitrary: n) (auto simp: drop-Cons split:nat.split)

lemma set-drop-subset-set-drop:
\( m \geq n \implies \text{set} \ (\text{drop} \ m \ \text{xs}) \subseteq \text{set} \ (\text{drop} \ n \ \text{xs}) \)
apply (induct xs arbitrary: m n)
apply (auto simp: drop-Cons split:nat.split)
by (metis set-drop-subset subset-iff)
lemma in-set-takeD: \( x : \text{set}(\text{take}\ n\ \text{xs}) \Rightarrow x : \text{set}\ \text{xs} \)
using set-take-subset by fast

lemma in-set-dropD: \( x : \text{set}(\text{drop}\ n\ \text{xs}) \Rightarrow x : \text{set}\ \text{xs} \)
using set-drop-subset by fast

lemma append-eq-conv-conj:
\((\text{xs} @ \text{ys} = \text{zs}) = (\text{xs} = \text{take}\ (\text{length}\ \text{xs})\ \text{zs} \land \text{ys} = \text{drop}\ (\text{length}\ \text{xs})\ \text{zs})\)
apply (induct \text{xs} arbitrary: \text{zs}, simp, clarsimp)
apply (case-tac \text{zs}, auto)
done

lemma take-add:
\(\text{take}\ (i+j)\ \text{xs} = \text{take}\ i\ \text{xs} @ \text{take}\ j\ (\text{drop}\ i\ \text{xs})\)
apply (induct \text{xs} arbitrary: \text{i}, auto)
apply (case-tac \text{i}, simp-all)
done

lemma append-eq-append-conv-if:
\((\text{xs}1 @ \text{xs}2 = \text{ys}1 @ \text{ys}2) =\)
\((\text{if size}\ \text{xs}1 \leq \text{size}\ \text{ys}1\) \then \(\text{xs}1 = \text{take}\ (\text{size}\ \text{xs}1)\ \text{ys}1 \land \text{xs}2 = \text{drop}\ (\text{size}\ \text{xs}1)\ \text{ys}1 @ \text{ys}2\) \else \(\text{take}\ (\text{size}\ \text{ys}1)\ \text{xs}1 = \text{ys}1 \land \text{drop}\ (\text{size}\ \text{ys}1)\ \text{xs}1 @ \text{xs}2 = \text{ys}2)\)
apply (induct \text{xs}1 arbitrary: \text{ys}1)
apply simp
apply (case-tac \text{ys}1)
apply simp-all
done

lemma take-hd-drop:
\(\text{n < length}\ \text{xs} \Rightarrow \text{take}\ n\ \text{xs} @ [\text{hd}\ (\text{drop}\ n\ \text{xs})] = \text{take}\ (\text{Suc}\ \text{n})\ \text{xs}\)
apply (induct \text{xs} arbitrary: \text{n})
apply simp
apply (simp add: drop-Cons split: nat.split)
done

lemma id-take-nth-drop:
\(\text{i < length}\ \text{xs} \Rightarrow \text{xs} = \text{take}\ i\ \text{xs} @ \text{xs}\!\#\ \text{drop}\ (\text{Suc}\ \text{i})\ \text{xs}\)
proof
–
assume \(\text{si:}\ \text{i < length}\ \text{xs}\)
hence \(\text{xs} = \text{take}\ (\text{Suc}\ \text{i})\ \text{xs} @ \text{drop}\ (\text{Suc}\ \text{i})\ \text{xs} \by auto\)
moreover
from \(\text{si}\) have \(\text{take}\ (\text{Suc}\ \text{i})\ \text{xs} = \text{take}\ i\ \text{xs} @ [\text{xs}\!\#]\)
apply (rule-tac take-Suc-cone-app-nth) \by arith
ultimately show \(\?\text{thesis}\ \by auto\)
qed

lemma upd-cone-take-nth-drop:
\[ i < \text{length } xs \implies xs[i:=a] = \text{take } i \text{ xs } \oplus a \# \text{ drop } \text{Suc } i \text{ xs} \]

**proof**

- **assume** \( i; i < \text{length } xs \)
- **have** \( xs[i:=a] = (\text{take } i \text{ xs } \oplus i \# \text{ drop } \text{Suc } i \text{ xs})[i:=a] \)
  by (rule \text{arg-cong} (OF \text{id-take-nth-drop} (OF i)])
- **also have** \( \ldots = \text{take } i \text{ xs } \oplus a \# \text{ drop } \text{Suc } i \text{ xs} \)
  using \( i \) by (simp add: \text{list-update-append})
- **finally show** \?thesis .

qed

**lemma** \( \text{nth-drop'} \):

\[ i < \text{length } xs \implies xs ! i \# \text{ drop } \text{Suc } i \text{ xs} = \text{drop } i \text{ xs} \]

**apply** (induct \( i \) \text{arbitrary: } xs)

**apply** (simp add: \text{neq-Nil-conv})

**apply** (erule \text{exE}+)

**apply** simp

**apply** (case_tac \( xs \))

**apply** simp-all

done

67.1.15  \text{takeWhile and dropWhile}

**lemma** \text{length-takeWhile-le}: \text{length } (\text{takeWhile } P \text{ xs}) \leq \text{length } xs

by (induct \( xs \)) auto

**lemma** \text{takeWhile-dropWhile-id} [simp]: \text{takeWhile } P \text{ xs } \oplus \text{dropWhile } P \text{ xs} = \text{xs}

by (induct \( xs \)) auto

**lemma** \text{takeWhile-append1} [simp]:

\[[i \text{:set } \text{xs}; \lnot P(x)] \implies \text{takeWhile } P \text{ (xs } \oplus y) = \text{takeWhile } P \text{ xs}\]

by (induct \( xs \)) auto

**lemma** \text{takeWhile-append2} [simp]:

\((\forall x. \text{set } \text{xs} \implies P(x)) \implies \text{takeWhile } P \text{ (xs } \oplus y) = \text{xs } \oplus \text{takeWhile } P \text{ ys}\)

by (induct \( xs \)) auto

**lemma** \text{takeWhile-tail}: \lnot P \text{ x } \implies \text{takeWhile } P \text{ (xs } \oplus (x\#l)) = \text{takeWhile } P \text{ xs}

by (induct \( xs \)) auto

**lemma** \text{takeWhile-nth}: \( j < \text{length } (\text{takeWhile } P \text{ xs}) \implies \text{takeWhile } P \text{ xs } ! j = \text{xs } ! j\)

**apply** (subst \( (3) \) \text{takeWhile-dropWhile-id}[symmetric]) unfolding \text{nth-append} by auto

**lemma** \text{dropWhile-nth}: \( j < \text{length } (\text{dropWhile } P \text{ xs}) \implies \text{dropWhile } P \text{ xs } ! j = \text{xs } ! (j + \text{length } (\text{takeWhile } P \text{ xs}))\)

**apply** (subst \( (3) \) \text{takeWhile-dropWhile-id}[symmetric]) unfolding \text{nth-append} by auto
lemma length-dropWhile-le: length (dropWhile P xs) ≤ length xs
by (induct xs) auto

lemma dropWhile-append1 [simp]:
[| x : set xs; ~P(x) |] ==> dropWhile P (xs @ ys) = (dropWhile P xs)@ys
by (induct xs) auto

lemma dropWhile-append2 [simp]:
(!x. x:set xs ==> P(x)) ==> dropWhile P (xs ys) = dropWhile P ys
by (induct xs) auto

lemma dropWhile-append3:
¬ P y ==> dropWhile P (xs @ y # ys) = dropWhile P ys
by (induct xs) auto

lemma dropWhile-last:
x ∈ set xs ==> ¬ P x ==> last (dropWhile P xs) = last xs
by (auto simp add: dropWhile-append3 in-set-conv-decomp)

lemma set-dropWhileD: x ∈ set (dropWhile P xs) ==> x ∈ set xs
by (induct xs) (auto split: split-if-asm)

lemma set-takeWhileD: x : set (takeWhile P xs) ==> x : set xs ∧ P x
by (induct xs) (auto split: split-if-asm)

lemma takeWhile-eq-all-conv[simp]:
(takeWhile P xs = xs) = (∀ x ∈ set xs. P x)
by (induct xs, auto)

lemma dropWhile-eq-Nil-conv[simp]:
(dropWhile P xs = []) = (∀ x ∈ set xs. P x)
by (induct xs, auto)

lemma dropWhile-eq-Cons-conv:
(dropWhile P xs = y#ys) = (xs = takeWhile P xs @ y # ys & ¬ P y)
by (induct xs, auto)

lemma distinct-takeWhile[simp]: distinct xs ==> distinct (takeWhile P xs)
by (induct xs) (auto dest: set-takeWhileD)

lemma distinct-dropWhile[simp]: distinct xs ==> distinct (dropWhile P xs)
by (induct xs) auto

lemma takeWhile-map: takeWhile P (map f xs) = map f (takeWhile (P o f) xs)
by (induct xs) auto

lemma dropWhile-map: dropWhile P (map f xs) = map f (dropWhile (P o f) xs)
by (induct xs) auto
**THEORY “List”**

**lemma** takeWhile-eq-take: \(\text{takeWhile } P \ x \cdot = \text{take} \ (\text{length} \ (\text{takeWhile } P \ x)) \ x\)

by (induct \(\cdot\)) auto

**lemma** dropWhile-eq-drop: \(\text{dropWhile } P \ x \cdot = \text{drop} \ (\text{length} \ (\text{takeWhile } P \ x)) \ x\)

by (induct \(\cdot\)) auto

**lemma** hd-dropWhile:

\(\text{dropWhile } P \ x \cdot \neq \text{[]} \implies \neg \ P \ (\text{hd} \ (\text{dropWhile } P \ x))\)

using assms by (induct \(\cdot\)) auto

**lemma** takeWhile-eq-filter:

assumes \(\forall x. \, x \in \text{set} \ (\text{dropWhile } P \ x) \implies \neg \ P \ x\)

shows \(\text{takeWhile } P \ x = \text{filter} \ P \ x\)

proof

\(-

have A: filter \ P \ x = \text{filter} \ P \ (\text{takeWhile } P \ x @ \text{dropWhile } P \ x)

by simp

have B: filter \ P \ (\text{dropWhile } P \ x) = []

unfolding filter-empty-conv using assms by blast

have filter \ P \ x = \text{takeWhile } P \ x

unfolding A filter-append B

by (auto simp add: filter-id-conv dest: set-takeWhileD)

thus \?thesis ..

qed

**lemma** takeWhile-eq-take-P-nth:

\([ \quad \forall i. \, [\ i < n \quad ; \ i < \text{length} \ x \quad ] \implies P \ (x ! i) \quad ; \ n < \text{length} \ x \implies \neg P \ (x ! n) \quad ] \quad \implies \text{takeWhile } P \ x = \text{take} \ n \ x\)

proof (induct \(\cdot\) arbitrary: \(n\))

case (Cons \(x\) \(\cdot\))

thus \?case

proof (cases \(n\))

case (Suc \(n\)) note this[simp]

have \(P \ x\) using Cons.prems(1)[of 0] by simp

moreover have \(\text{takeWhile } P \ x = \text{take} \ n \ x\) proof (rule Cons.hyps)

case goal1 thus \(P \ (x ! i)\) using Cons.prems(1)[of Suc \(i\)] by simp

next case goal2 thus \?case using Cons by auto

qed

ultimately show \?thesis by simp

qed simp

**lemma** nth-length-takeWhile:

\(\text{length} \ (\text{takeWhile } P \ x) < \text{length} \ x \implies \neg P \ (x ! \text{length} \ (\text{takeWhile } P \ x))\)

by (induct \(\cdot\)) auto

**lemma** length-takeWhile-less-P-nth:

assumes all: \(\forall i. \, i < j \implies P \ (x ! i)\) and \(j \leq \text{length} \ x\)

**lemma** takeWhile-eq-filter:
shows \( j \leq \text{length} \ (\text{takeWhile} \ P \ \text{xs}) \)

proof (rule classical)

assume \( \neg \ \text{thesis} \)

hence \( \text{length} \ (\text{takeWhile} \ P \ \text{xs}) < \text{length} \ \text{xs} \) using assms by simp 

thus \( \text{thesis} \) using all \( (\neg \ \text{thesis}) \) nth-length-takeWhile[of \ P \ \text{xs}] \ by \ auto 

qed 

The following two lemmas could be generalized to an arbitrary property.

lemma takeWhile-neq-rev: \[ \text{distinct} \ \text{xs}; \ x \in \text{set} \ \text{xs} \] = \[ \Rightarrow \] 

takeWhile \( (\lambda y. \ y \neq x) \) (\text{rev} \ \text{xs}) = \text{rev} \ (\text{tl} \ (\text{dropWhile} \ (\lambda y. \ y \neq x) \ \text{xs})) \)

by (induct \ \text{xs} \) (auto simp: takeWhile-tail[where \ l=[]])

lemma dropWhile-neq-rev: \[ \text{distinct} \ \text{xs}; \ x \in \text{set} \ \text{xs} \] = \[ \Rightarrow \] 

dropWhile \( (\lambda y. \ y \neq x) \) (\text{rev} \ \text{xs}) = x \# \text{rev} \ (\text{takeWhile} \ (\lambda y. \ y \neq x) \ \text{xs}) \)

apply (induct \ \text{xs} \)

apply simp

apply auto

apply (subst dropWhile-append2)

apply auto

done 

lemma takeWhile-not-last:

distinct \ \text{xs} = \[ \Rightarrow \] \text{takeWhile} \ (\lambda y. \ y \neq \text{last} \ \text{xs}) \ \text{xs} = \text{butlast} \ \text{xs} \)

apply (induct \ \text{xs} \)

apply simp

apply (case-tac \ \text{xs} \)

apply (auto)

done 

lemma takeWhile-cong \[ \text{fundef-cong} \]:

\[
\begin{array}{l}
\| \ \text{l} = \ \text{k}; \ !\! x. \ \text{x} : \text{set} \ \text{l} = \Rightarrow \ \text{P} \ \text{x} = \ \text{Q} \ \text{x} \ \|
\\
\Rightarrow \text{takeWhile} \ \text{P} \ \text{l} = \text{takeWhile} \ \text{Q} \ \text{k}
\end{array}
\]

by (induct \ \text{k} \ arbitrary: \ l) (simp-all)

lemma dropWhile-cong \[ \text{fundef-cong} \]:

\[
\begin{array}{l}
\| \ \text{l} = \ \text{k}; \ !\! x. \ \text{x} : \text{set} \ \text{l} = \Rightarrow \ \text{P} \ \text{x} = \ \text{Q} \ \text{x} \ \|
\\
\Rightarrow \text{dropWhile} \ \text{P} \ \text{l} = \text{dropWhile} \ \text{Q} \ \text{k}
\end{array}
\]

by (induct \ \text{k} \ arbitrary: \ l, simp-all)

lemma takeWhile-idem \[ \text{simp} \]:

takeWhile \ \text{P} \ (\text{takeWhile} \ \text{P} \ \text{xs}) = \text{takeWhile} \ \text{P} \ \text{xs}

by (induct \ \text{xs} \) auto 

lemma dropWhile-idem \[ \text{simp} \]:

dropWhile \ \text{P} \ (\text{dropWhile} \ \text{P} \ \text{xs}) = \text{dropWhile} \ \text{P} \ \text{xs}

by (induct \ \text{xs} \) auto
67.1.16  zip

lemma zip-Nil [simp]:  zip [] ys = []
by (induct ys) auto

lemma zip-Cons-Cons [simp]:  zip (x # xs) (y # ys) = (x, y) # zip xs ys
by simp

declare zip-Cons [simp del]

lemma [code]:
  zip [] ys = []
  zip xs [] = []
  zip (x # xs) (y # ys) = (x, y) # zip xs ys
by (fact zip-Nil zip.simps(1) zip-Cons-Cons)+

lemma zip-Cons1:
  zip (x # xs) ys = (case ys of [] ⇒ [] | y#ys ⇒ (x,y)#zip xs ys)
by (auto split: list.split)

lemma length-zip [simp]:
  length (zip xs ys) = min (length xs) (length ys)
by (induct xs ys rule: list-induct2' auto

lemma zip-obtain-same-length:
  assumes ∀zs ws n. length zs = length ws ⇒ n = min (length xs) (length ys)
  ⇒ zs = take n xs ⇒ ws = take n ys ⇒ P (zip zs ws)
  shows P (zip xs ys)
proof –
  let ?n = min (length xs) (length ys)
  have P (zip (take ?n xs) (take ?n ys))
    by (rule assms) simp-all
  moreover have zip xs ys = zip (take ?n xs) (take ?n ys)
  proof (induct xs arbitrary: ys)
    case Nil then show ?case by simp
  next
    case (Cons x xs) then show ?case by (cases ys) simp-all
  qed
  ultimately show ?thesis by simp
  qed

lemma zip-append1:
  zip (xs @ ys) zs =
  zip xs (take (length xs) zs) @ zip ys (drop (length xs) zs)
by (induct xs zs rule: list-induct2') auto

lemma zip-append2:
  zip xs (ys @ zs) =
  zip (take (length ys) xs) ys @ zip (drop (length ys) xs) zs
by (induct xs ys rule: list-induct2') auto
lemma zip-append [simp]:
\[ \text{length } xs = \text{length } us \implies \text{zip } (xs @ ys) (us @ vs) = \text{zip } xs us @ \text{zip } ys vs \]
by (simp add: zip-append1)

lemma zip-rev:
\[ \text{length } xs = \text{length } ys \implies \text{zip } (\text{rev } xs) (\text{rev } ys) = \text{rev } (\text{zip } xs ys) \]
by (induct rule: list-induct2, simp-all)

lemma zip-map-map:
\[ \text{zip } (\text{map } f xs) (\text{map } g ys) = \text{map } (\lambda (x, y). (f x, g y)) (\text{zip } xs ys) \]
proof (induct xs arbitrary: ys)
case \(\text{Cons } x xs\)
next
next
show \(?\)thesis
proof
show \(?\)thesis unfolding Cons using Cons-x-xs by simp
qed simp
qed simp

lemma zip-map1:
\[ \text{zip } (\text{map } f xs) ys = \text{map } (\lambda (x, y). (f x, y)) (\text{zip } xs ys) \]
using zip-map-map[of f xs \(\lambda x. x\) ys] by simp

lemma zip-map2:
\[ \text{zip } xs (\text{map } f ys) = \text{map } (\lambda (x, y). (x, f y)) (\text{zip } xs ys) \]
using zip-map-map[of \(\lambda x. x\) xs f ys] by simp

lemma map-zip-map:
\[ \text{map } f (\text{zip } (\text{map } g xs) ys) = \text{map } (\% (x, y). f (g x, y)) (\text{zip } xs ys) \]
unfolding zip-map1 by auto

lemma map-zip-map2:
\[ \text{map } f (\text{zip } xs (\text{map } g ys)) = \text{map } (\% (x, y). f (x, g y)) (\text{zip } xs ys) \]
unfolding zip-map2 by auto

Courtesy of Andreas Lochbihler:

lemma zip-same-conv-map:
\[ \text{zip } xs xs = \text{map } (\lambda x. (x, x)) xs \]
by (induct xs) auto

lemma nth-zip [simp]:
\[ \| i < \text{length } xs; i < \text{length } ys \| \implies (\text{zip } xs ys)!i = (xs!i, ys!i) \]
apply (induct ys arbitrary: i xs, simp)
apply (case-tac xs)
apply (simp-all add: nth.simps split: nat.split)
done

lemma set-zip:
set (zip xs ys) = \{(x$i$, y$i$) | $i$, $i < \text{min (length } xs \text{) (length } ys\})\}
by(simp add: set-conv-nth cong: rev-conj-cong)

lemma zip-same: ((a, b) ∈ set (zip xs xs)) = (a ∈ set xs ∧ a = b)
by(induct xs) auto

lemma zip-update:
zip (xs[i:=x]) (ys[i:=y]) = (zip xs ys)[i:=(x,y)]
by(rule sym, simp add: update-zip)

lemma zip-replicate [simp]:
zip (replicate i x) (replicate j y) = replicate (min i j) (x,y)
apply (induct i arbitrary: j, auto)
apply (case-tac j, auto)
done

lemma take-zip:
take n (zip xs ys) = zip (take n xs) (take n ys)
apply (induct n arbitrary: xs ys)
apply simp
apply (case-tac xs, simp)
apply (case-tac ys, simp-all)
done

lemma drop-zip:
drop n (zip xs ys) = zip (drop n xs) (drop n ys)
apply (induct n arbitrary: xs ys)
apply simp
apply (case-tac xs, simp)
apply (case-tac ys, simp-all)
done

lemma zip-takeWhile-fst: zip (takeWhile P xs) ys = takeWhile (P ◦ fst) (zip xs ys)
proof (induct xs arbitrary: ys)
  case (Cons x xs) thus ?case by (cases ys) auto
qed simp

lemma zip-takeWhile-snd: zip xs (takeWhile P ys) = takeWhile (P ◦ snd) (zip xs ys)
proof (induct xs arbitrary: ys)
  case (Cons x xs) thus ?case by (cases ys) auto
qed simp

lemma set-zip-leftD:
(x,y)∈ set (zip xs ys) ⟷ x ∈ set xs
by (induct xs ys rule:list-induct2' auto)

lemma set-zip-rightD:
\[(x,y) \in \text{set} (\text{zip} \text{ } xs \text{ } ys) \implies y \in \text{set} \text{ } ys\]

by \((\text{induct } xs \text{ } ys \text{ } \text{rule:} \text{list-induct2'}) \text{ } \text{auto}\)

\textbf{lemma in-set-zipE:}
\[(x,y) : \text{set} (\text{zip} \text{ } xs \text{ } ys) \implies ([x : \text{set} \text{ } xs; y : \text{set} \text{ } ys] \implies R) \implies R\]
by\((\text{blast dest:} \text{set-zip-leftD} \text{ } \text{set-zip-rightD})\)

\textbf{lemma zip-map-fst-snd:}
\[
\text{zip} (\text{map } \text{fst} \text{ } zs) (\text{map } \text{snd} \text{ } zs) = zs
\]
by \((\text{induct } zss) \text{ } \text{simp-all}\)

\textbf{lemma zip-eq-conv:}
\[
\text{length} \text{ } xs = \text{length} \text{ } ys \implies \text{zip} \text{ } xs \text{ } ys = zs \iff \text{map } \text{fst} \text{ } zs = xs \land \text{map } \text{snd} \text{ } zs = ys
\]
by \((\text{auto simp add:} \text{zip-map-fst-snd})\)

\textbf{lemma pair-list-eqI:}
\[
\text{assumes} \text{ } \text{map } \text{fst} \text{ } xs = \text{map } \text{fst} \text{ } ys \text{ } \text{and} \text{ } \text{map } \text{snd} \text{ } xs = \text{map } \text{snd} \text{ } ys
\text{shows} \text{ } xs = ys
\]
proof
–
from \text{assms}(1) \text{ have} \text{ length} \text{ } xs = \text{length} \text{ } ys \text{ by} \text{ (rule map-eq-imp-length-eq)}
from \text{this} \text{assms} \text{ show} \ \text{thesis}
by \((\text{induct } xs \text{ } ys \text{ rule:} \text{list-induct2}) \text{ (simp-all add:} \text{prod-eqI})\)
qed

\textbf{67.1.17 \ list-all2\)
\textbf{lemma list-all2-lengthD [intro?):}
\text{list-all2 } P \text{ } xs \text{ } ys \implies length \text{ } xs = length \text{ } ys
by \((\text{simp add:} \text{list-all2-iff})\)

\textbf{lemma list-all2-Nil [iff, code]: \text{list-all2 } P \text{ } [] \text{ } ys = (ys = [])}
by \((\text{simp add:} \text{list-all2-iff})\)

\textbf{lemma list-all2-Nil2 [iff, code]: \text{list-all2 } P \text{ } xs \text{ } [] = (xs = [])}
by \((\text{simp add:} \text{list-all2-iff})\)

\textbf{lemma list-all2-Cons [iff, code]:}
\[
\text{list-all2 } P \text{ } (x \# \text{ } xs) \text{ } (y \# \text{ } ys) = (P \text{ } x \text{ } y \lor \text{list-all2 } P \text{ } xs \text{ } ys)
\]
by \((\text{auto simp add:} \text{list-all2-iff})\)

\textbf{lemma list-all2-Cons1:}
\[
\text{list-all2 } P \text{ } (x \# \text{ } xs) \text{ } ys = (\exists z \text{ } zs. \text{ } ys = z \# \text{ } zs \land P \text{ } x \text{ } z \land \text{list-all2 } P \text{ } xs \text{ } zs)
\]
by \((\text{cases } ys) \text{ } \text{auto}\)
lemma list-all2-Cons2:
list-all2 P xs (y # ys) = (∃ z zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
by (cases xs) auto

lemma list-all2-induct
[consumes 1, case-names Nil Cons, induct set: list-all2]:
assumes P: list-all2 P xs ys
assumes Nil: R [] []
assumes Cons: ∀ x xs y ys. [P x y; list-all2 P xs ys; R xs ys] ⇒ R (x # xs) (y # ys)
shows R xs ys
using P
by (induct xs arbitrary: ys) (auto simp add: list-all2-Cons1 Nil Cons)

lemma list-all2-rev [iff]:
list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
by (simp add: list-all2-iff zip-rev cong: conj-cong)

lemma list-all2-rev1:
list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
by (subst list-all2-rev [symmetric]) simp

lemma list-all2-append1:
list-all2 P (xs @ ys) zs =
(EX us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
list-all2 P us ys ∧ list-all2 P vs)
apply (simp add: list-all2-iff zip-append1)
apply (rule iffI)
apply (rule-tac x = take (length xs) zs in ezI)
apply (rule-tac x = drop (length xs) zs in ezI)
apply (force split: nat-diff-split simp add: min-def, clarify)
apply (simp add: ball-Un)
done

lemma list-all2-append2:
list-all2 P xs (ys @ zs) =
(EX us vs. zs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
list-all2 P us ys ∧ list-all2 P vs)
apply (simp add: list-all2-iff zip-append2)
apply (rule iffI)
apply (rule-tac x = take (length ys) xs in ezI)
apply (rule-tac x = drop (length ys) xs in ezI)
apply (force split: nat-diff-split simp add: min-def, clarify)
apply (simp add: ball-Un)
done

lemma list-all2-append:
length xs = length ys =>
list-all2 P (xs @ us) (ys @ vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
THEORY "List"

by (induct rule:list-induct2, simp-all)

lemma list-all2-appendI [intro?, trans]:
  [ list-all2 P a b; list-all2 P c d ] \implies list-all2 P (a@b) (b@d)
by (simp add: list-all2-append list-all2-lengthD)

lemma list-all2-conv-all-nth:
list-all2 P xs ys =
(length xs = length ys \land (\forall i < length xs. P (xs!i) (ys!i)))
by (force simp add: list-all2-iff set-zip)

lemma list-all2-trans:
assumes tr: !!a b c. P1 a b == P2 b c == P3 a c
shows !!bs cs. list-all2 P1 as bs == list-all2 P2 bs cs == list-all2 P3 as cs
(is !!bs cs. PROP ?Q as bs cs)
proof (induct as)
  fix x xs bs assume I1: !!bs cs. PROP ?Q xs bs cs
  show !!cs. PROP ?Q (x # xs) bs cs
  proof (induct bs)
    fix y ys cs assume I2: !!cs. PROP ?Q (x # xs) (y # ys) cs
    show PROP ?Q (x # xs) (y # ys) cs
    by (induct cs) (auto intro: tr I1 I2)
  qed simp
qed simp

lemma list-all2-all-nthI [intro?):
length a = length b =
(\forall n < length a \implies P (a!n) (b!n)) = list-all2 P a b
by (simp add: list-all2-conv-all-nth)

lemma list-all2I:
\forall x \in set (zip a b), split P x \implies length a = length b \implies list-all2 P a b
by (simp add: list-all2-iff)

lemma list-all2-nthD:
[ list-all2 P xs ys; p < size xs ] \implies P (xs!p) (ys!p)
by (simp add: list-all2-conv-all-nth)

lemma list-all2-nthD2:
[ list-all2 P xs ys; p < size ys ] \implies P (xs!p) (ys!p)
by (frule list-all2-lengthD) (auto intro: list-all2-nthD)

lemma list-all2-map1:
list-all2 P (map f as) bs = list-all2 (\lambda x y. P (f x) y) as bs
by (simp add: list-all2-conv-all-nth)

lemma list-all2-map2:
list-all2 P as (map f bs) = list-all2 (\lambda x y. P x (f y)) as bs
by (auto simp add: list-all2-conv-all-nth)
lemma list-all2-refl [intro?):
(\x. P x x) \rightarrow list-all2 P xs xs
by (simp add: list-all2-conv-all-nth)

lemma list-all2-update-cong:
[ [ list-all2 P xs ys; P x y ] ] \rightarrow list-all2 P (xs[i:=x]) (ys[i:=y])
by (cases i < length ys) (auto simp add: list-all2-conv-all-nth nth-list-update)

lemma list-all2-takeI [simp, intro?]:
list-all2 P xs ys = \rightarrow list-all2 P (take n xs) (take n ys)
apply (induct xs arbitrary: n ys)
apply simp
apply (clarsimp simp add: list-all2-Cons1)
apply (case-tac n)
apply auto
done

lemma list-all2-dropI [simp, intro?]:
list-all2 P as bs = \rightarrow list-all2 P (drop n as) (drop n bs)
apply (induct as arbitrary: n bs, simp)
apply (clarsimp simp add: list-all2-Cons1)
apply (case-tac n, simp, simp)
done

lemma list-all2-mono [intro?):
list-all2 P xs ys = \rightarrow (\forall xs ys. P xs ys = \rightarrow Q xs ys) = \rightarrow list-all2 Q xs ys
apply (induct xs arbitrary: ys, simp)
apply (case-tac ys, auto)
done

lemma list-all2-eq:
xs = ys \leftrightarrow list-all2 (op =) xs ys
by (induct xs ys rule: list-induct2’) auto

lemma list-eq-iff-zip-eq:
xz = ys \leftrightarrow length xz = length ys \wedge (\forall (x,y) \in set (zip xz ys), x = y)
by(auto simp add: set-zip list-all2-eq list-all2-conv-all-nth cong: conj-cong)

lemma list-all2-same: list-all2 P xs xs \leftrightarrow (\forall x \in set xs. P x x)
by(auto simp add: list-all2-conv-all-nth set-conv-nth)

67.1.18  List.product and product-lists

lemma set-product[simp]:
set (List.product xs ys) = set xs \times set ys
by (induct xs) auto

lemma length-product [simp]:
length (List.product xs ys) = length xs \times length ys
by (induct xs) simp-all

lemma product-nth:
  assumes n < length xs * length ys
  shows List.product xs ys ! n = (xs ! (n div length ys), ys ! (n mod length ys))
  using assms proof (induct xs arbitrary: n)
  case Nil then show ?case by simp
  next
  case (Cons x xs n)
  then have length ys > 0 by auto
  with Cons show ?case
  by (auto simp add: nth-append not-less le-mod-geq le-div-geq)
qed

lemma in-set-product-lists-length:
  xs ∈ set (product-lists xss) ⟹ length xs = length xss
  by (induct xss arbitrary: xs) auto

lemma product-lists-set:
  set (product-lists xss) = {xs. list-all2 (λx ys. x ∈ set ys) xs xss}
  (is ?L = Collect ?R)
proof (intro equalityI subsetI, unfold mem-Collect-eq)
  fix xs assume xs ∈ ?L
  then have length xs = length xss by (rule in-set-product-lists-length)
  from this ⟨xs ∈ ?L⟩ show ?R xs by (induct xs xss rule: list-induct2) auto
next
  fix xs assume ?R xs
  then show xs ∈ ?L by induct auto
qed

67.1.19 fold with natural argument order

lemma fold-simps [code]: — eta-expanded variant for generated code – enables
tail-recursion optimisation in Scala
  fold f [] s = s
  fold f (x # xs) s = fold f xs (f x s)
  by simp-all

lemma fold-remove1-split:
  assumes f: ∀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ f x ∘ f y = f y ∘ f x
  and x: x ∈ set xs
  shows fold f xs = fold f (remove1 x xs) ∘ f x
  using assms by (induct xs) (auto simp add: comp-assoc)

lemma fold-cong [fundef-cong]:
  a = b ⟹ xs = ys ⟹ (∀x. x ∈ set xs ⟹ f x = g x)
  ⟹ fold f xs a = fold g ys b
  by (induct ys arbitrary: a b xs) simp-all
lemma fold-id:
assumes \( \forall x. x \in \text{set } xs \implies f x = id \)
shows \( \text{fold } f xs = id \)
using assms by (induct xs) simp-all

lemma fold-commute:
assumes \( \forall x. x \in \text{set } xs \implies h \circ g x = f x \circ h \)
shows \( h \circ \text{fold } g xs = \text{fold } f xs \circ h \)
using assms by (induct xs) (simp-all add: fun-eq-iff)

lemma fold-commute-apply:
assumes \( \forall x. x \in \text{set } xs \implies h \circ g x = f x \circ h \)
shows \( h (\text{fold } g xs s) = \text{fold } f xs (h s) \)
proof -
from assms have \( h \circ \text{fold } g xs = \text{fold } f xs \circ h \) by (rule fold-commute)
then show \( \text{thesis} \) by (simp add: fun-eq-iff)
qed

lemma fold-invariant:
assumes \( \forall x. x \in \text{set } xs \implies Q x \) and \( P s \)
and \( \forall x s. Q x = P s \implies P (f x s) \)
shows \( P (\text{fold } f xs s) \)
using assms by (induct xs arbitrary: s) simp-all

lemma fold-append [simp]:
\( \text{fold } f (xs @ ys) = \text{fold } f ys \circ \text{fold } f xs \)
by (induct xs) simp-all

lemma fold-map [code-unfold]:
\( \text{fold } g (\text{map } f xs) = \text{fold } (g \circ f) xs \)
by (induct xs) simp-all

lemma fold-rev:
assumes \( \forall x y. x \in \text{set } xs \implies y \in \text{set } xs \implies f y \circ f x = f x \circ f y \)
shows \( \text{fold } f (\text{rev } xs) = \text{fold } f xs \)
using assms by (induct xs) (simp-all add: fold-commute-apply fun-eq-iff)

lemma fold-Cons-rev:
\( \text{fold } \text{Cons } xs = \text{append } (\text{rev } xs) \)
by (induct xs) simp-all

lemma rev-conv-fold [code]:
\( \text{rev } xs = \text{fold } \text{Cons } xs \)[
by (simp add: fold-Cons-rev)

lemma fold-append-concat-rev:
\( \text{fold } \text{append } xss = \text{append } (\text{concat } (\text{rev } xss)) \)
by (induct xss) simp-all

Finite-Set.fold and fold
lemma (in comp-fun-commute) fold-set-fold-remdups:
  Finite-Set.fold f y (set xs) = fold f (remdups xs) y
by (rule sym, induct xs arbitrary: y) (simp-all add: fold-fun-left-comm insert-absorb)

lemma (in comp-fun-idem) fold-set-fold:
  Finite-Set.fold f y (set xs) = fold f xs y
by (rule sym, induct xs arbitrary: y) (simp-all add: fold-fun-left-comm)

lemma union-set-fold [code]:
  set xs ∪ A = fold Set.insert xs A
proof –
  interpret comp-fun-idem Set.insert
  by (fact comp-fun-idem-insert)
  show ?thesis by (simp add: union-fold-insert fold-set-fold)
qed

lemma union-coset-filter [code]:
  List.coset xs ∪ A = List.coset (List.filter (λx. x ∉ A) xs)
by auto

lemma minus-set-fold [code]:
  A − set xs = fold Set.remove xs A
proof –
  interpret comp-fun-idem Set.remove
  by (fact comp-fun-idem-remove)
qed

lemma minus-coset-filter [code]:
  A − List.coset xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-set-filter [code]:
  A ∩ set xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-coset-fold [code]:
  A ∩ List.coset xs = fold Set.remove xs A
by (simp add: Diff-eq [symmetric] minus-set-fold)

lemma (in semilattice-set) set-eq-fold [code]:
  F (set (x ≠ xs)) = fold f xs x
proof –
  interpret comp-fun-idem f
  by default (simp-all add: fun-eq-iff left-commute)
  show ?thesis by (simp add: eq-fold fold-set-fold)
qed
THEORY “List”

lemma (in complete-lattice) Inf-set-fold:
  \( \text{Inf} (\text{set} \ xs) = \text{fold} \ \text{inf} \ xs \ \text{top} \)

proof
  interpret \( \text{comp-fun-idem inf} :: 'a \Rightarrow 'a \Rightarrow 'a \)
    by (fact comp-fun-idem-inf)
  show \( \text{thesis} \) by (simp add: Inf-fold-inf fold-set-fold inf-commute)
qed

declare Inf-set-fold [where \( 'a = 'a \ \text{set} \), code]

lemma (in complete-lattice) Sup-set-fold:
  \( \text{Sup} (\text{set} \ xs) = \text{fold} \ \text{sup} \ xs \ \text{bot} \)

proof
  interpret \( \text{comp-fun-idem sup} :: 'a \Rightarrow 'a \Rightarrow 'a \)
    by (fact comp-fun-idem-sup)
  show \( \text{thesis} \) by (simp add: Sup-fold-sup fold-set-fold sup-commute)
qed

declare Sup-set-fold [where \( 'a = 'a \ \text{set} \), code]

lemma (in complete-lattice) INF-set-fold:
  \( \text{INFIMUM} (\text{set} \ xs) \ f = \text{fold} (\text{inf} \circ f) \ xs \ \text{top} \)

using Inf-set-fold [of map f xs] by (simp add: fold-map)

decallare INF-set-fold [code]

lemma (in complete-lattice) SUP-set-fold:
  \( \text{SUPREMUM} (\text{set} \ xs) \ f = \text{fold} (\text{sup} \circ f) \ xs \ \text{bot} \)

using Sup-set-fold [of map f xs] by (simp add: fold-map)

decallare SUP-set-fold [code]

67.1.20 Fold variants: foldr and foldl

Correspondence

lemma foldr-conv-fold [code-abbrev]:
  \( \text{foldr} \ f \ xs = \text{fold} \ f \ (\text{rev} \ xs) \)
  by (induct xs) simp-all

lemma foldl-conv-fold:
  \( \text{foldl} \ f \ s \ xs = \text{fold} \ (\lambda x s . f \ s \ x) \ xs \ s \)
  by (induct xs arbitrary: \ s \) simp-all

lemma foldr-conv-foldl: — The “Third Duality Theorem” in Bird & Wadler:
  \( \text{foldr} \ f \ s \ a = \text{foldl} \ (\lambda x y . f \ y \ x) \ a \ (\text{rev} \ xs) \)
  by (simp add: foldr-conv-fold foldl-conv-fold)

lemma foldl-conv-foldr:
  \( \text{foldl} \ f \ a \ xs = \text{foldr} \ (\lambda x y . f \ y \ x) \ (\text{rev} \ xs) \ a \)
by (simp add: foldr-conv-fold foldl-conv-fold)

lemma foldr-fold:
assumes \( \forall x y. x \in \text{set} \; xs \Rightarrow y \in \text{set} \; xs \Rightarrow f y \circ f x = f x \circ f y \)
shows \( \text{foldr} \; f \; xs = \text{fold} \; f \; xs \)
using assms unfolding foldr-conv-fold by (rule fold-rev)

lemma foldl-cong [fundef-cong]:
\( a = b \Rightarrow l = k \Rightarrow (\forall a \; x. \; x \in \text{set} \; l \Rightarrow f \; a \; x = g \; a \; x) \Rightarrow \text{foldl} \; f \; a \; l = \text{foldl} \; g \; b \; k \)
by (auto simp add: foldr-conv-fold intro: fold-cong)

lemma foldl-map [code-unfold]:
\( \text{foldl} \; g \; (\text{map} \; f \; xs) \; a = \text{foldl} \; (g \; o \; f) \; xs \; a \)
by (simp add: foldl-conv-fold)

lemma foldl-append [simp]:
\( \text{foldl} \; f \; a \; (xs @ ys) = \text{foldl} \; f \; (\text{foldl} \; f \; a \; xs) \; ys \)
by (simp add: foldl-conv-fold)

lemma concat-conv-foldr [code]:
\( \text{concat} \; xss = \text{foldr} \; \text{append} \; xss \; [] \)
by (simp add: foldr-conv-fold fold-append-concat-rev)

67.1.21 upt

lemma upt-rec [code]: 
\( [i..<j] = (\text{if} \; i < j \; \text{then} \; i \# [\text{Suc} \; i..<j] \; \text{else} \; []) \)
— simp does not terminate!
by (induct j) auto

lemmas upt-rec-numeral [simp] = upt-rec[of numeral m numeral n] for m n

lemma upt-conv-Nil [simp]: 
\( j <= i =\Rightarrow [i..<j] = [] \)
by (subst upt-Nil) simp

lemma upt-eq-Nil-conv [simp]: 
\( ([i..<j] = []) = (j = 0 \lor j <= i) \)
by (induct j) simp-all

lemma upt-eq-Cons-conv:
(i..<j) = x#xs = (i < j & i = x & [i+1..<j] = xs)
apply (induct j arbitrary: x xs)
  apply simp
apply (clarsimp simp add: append-eq-Cons-conv)
apply arith
done

lemma upt-Suc-append: i <= j ==> [i..<(Suc j)] = [i..<j]@[j]
— Only needed if upt-Suc is deleted from the simpset.
by simp

lemma upt-conv-Cons: i < j ==> [i..<j] = i # [Suc i..<j]
  by (simp add: upt-rec)

lemma upt-add-eq-append: i<=j ==> [i..<j+k] = [i..<j]@[j..<j+k]
— LOOPS as a simprule, since j <= j.
bys (induct k) auto

lemma length-upt [simp]: length [i..<j] = j - i
  by (induct j) (auto simp add: Suc-diff-le)

lemma nth-upt [simp]: i < j ==> [i..<j] ! k = i + k
apply (induct j)
apply (auto simp add: less-Suc-eq nth-append split: nat-diff-split)
done

lemma hd-upt [simp]: i < j ==> hd [i..<j] = i
by (simp add: upt-conv-Cons)

lemma last-upt [simp]: i < j ==> last [i..<j] = j - 1
apply (cases j)
apply simp
by (simp add: upt-Suc-append)

lemma take-upt [simp]: i+m <= n ==> take m [i..<n] = [i..<i+m]
apply (induct m arbitrary: i, simp)
apply (rule sym)
apply (subst upt-rec)
apply (simp del: upt.simps)
done

lemma drop-upt [simp]: drop m [i..<j] = [i+m..<j]
apply (induct j)
apply auto
done

**lemma** map-Suc-upt: map Suc [m..<n] = [Suc m..<Suc n]
by (induct n) auto

**lemma** map-add-upt: map (λi. i + n) [0..<m] = [n..<m + n]
by (induct m) simp-all

**lemma** nth-map-upt: i < n−m =⇒ (map f [m..<n]) ! i = f(m+i)
apply (induct n m arbitrary: i rule: diff-induct)
prefer 3 apply (subst map-Suc-upt [symmetric])
apply (auto simp add: less-diff-cone)
done

**lemma** nth-map-decr-upt:
map (λn. n − Suc 0) [Suc m..<Suc n] = [m..<n]
by (induct n) simp-all

**lemma** nth-take-lemma:
k < length xs =⇒ k <= length ys =⇒
(∀i. i < k =⇒ xs!i = ys!i) =⇒ take k xs = take k ys
apply (atomize, induct k arbitrary: xs ys)
apply (simp-all add: less-Suc-eq-0-disj all-conj-distrib, clarify)

Both lists must be non-empty
apply (case-tac xs, simp)
apply (case-tac ys, clarify)
apply (simp (no asm use))
apply clarify

prenexing’s needed, not miniscoping
apply (simp (no asm use) add: all-simps [symmetric] del: all-simps)
apply blast
done

**lemma** nth-equalityI:
[\[ length xs = length ys; ALL i < length xs. xs!i = ys!i \] =⇒ xs = ys]
by (frule nth-take-lemma [OF le-refl eq-imp-le]) simp-all

**lemma** map-nth:
map (λi. xs ! i) [0..<length xs] = xs
by (rule nth-equalityI, auto)

**lemma** list-all2-antisym:
[\[ (∀x y. [P x y; Q y x] =⇒ x = y); list-all2 P xs ys; list-all2 Q ys xs \]
=⇒ xs = ys]
apply (simp add: list-all2-conv-all-nth)
apply (rule nth-equalityI, blast, simp)
THEORY “List”

done

lemma take-equalityI: (∀ i. take i xs = take i ys) ==> xs = ys
— The famous take-lemma.
apply (drule-tac x = max (length xs) (length ys) in spec)
apply (simp add: le-max-iff-disj)
done

lemma take-Cons':
  take n (x # xs) = (if n = 0 then [] else x # take (n - 1) xs)
by (cases n) simp-all

lemma drop-Cons':
  drop n (x # xs) = (if n = 0 then x # xs else drop (n - 1) xs)
by (cases n) simp-all

lemma nth-Cons': (x # xs)!n = (if n = 0 then x else xs!(n - 1))
by (cases n) simp-all

lemma take-Cons-numeral [simp]:
  take (numeral v) (x # xs) = x # take (numeral v - 1) xs
by (simp add: take-Cons')

lemma drop-Cons-numeral [simp]:
  drop (numeral v) (x # xs) = drop (numeral v - 1) xs
by (simp add: drop-Cons')

lemma nth-Cons-numeral [simp]:
  (x # xs)!numeral v = xs!(numeral v - 1)
by (simp add: nth-Cons')

67.1.22 upto: interval-list on int

function upto :: int ⇒ int ⇒ int list ((1[−/])
upto i j = (if i ≤ j then i # [i+1..j] else [])
by auto
termination
by(relation measure(%(i::int,j). nat(j - i + 1))) auto

declare upto.simps[simp del]

lemmas upto-rec-numeral [simp] =
  upto.simps[of numeral m numeral n]
  upto.simps[of numeral m - numeral n]
  upto.simps[of - numeral m numeral n]
  upto.simps[of - numeral m - numeral n] for m n

lemma upto-empty[simp]: j < i ==> [i..j] = []
THEORY "List"

by(simp add: upto.simps)

lemma upto-rec1: \(i \leq j \implies [i..j] = i#[i+1..j]\)
by(simp add: upto.simps)

lemma upto-rec2: \(i \leq j \implies [i..j] = [i..j - 1]@[j]\)
proof(induct nat(j-i) arbitrary: i j)
case 0 thus ?case by(simp add: upto.simps)
next
case (Suc n)
  hence \(n = \text{nat}(j - (i + 1))\) \(i < j\) by linarith+
from this(2) Suc.hyps(1)[OF this(1)] Suc(2,3) upto-rec1 show ?case by simp
qed

lemma set-upto[simp]: \(\text{set}[i..j] = \{i..j\}\)
proof(induct i j rule:upto.induct)
case (1 i j)
  unfolding upto.simps[of i j] by auto
qed

Tail recursive version for code generation:
definition upto-aux :: int ⇒ int ⇒ int list ⇒ int list where
  upto-aux i j js = [i..j] @ js

lemma upto-aux-rec [code]:
  upto-aux i j js = (if j<i then js else upto-aux i (j - 1) (j#js))
by (simp add: upto-aux-def upto-rec2)

lemma upto-code[code]: [i..j] = upto-aux i j []
by(simp add: upto-aux-def)

67.1.23 distinct and remdups and remdups-adj

lemma distinct-tl:
distinct xs \implies distinct (tl xs)
by (cases xs) simp-all

lemma distinct-append [simp]:
distinct (xs @ ys) = (distinct xs ∧ distinct ys ∧ set xs ∩ set ys = {})
by (induct xs) auto

lemma distinct-rev[simp]: distinct(rev xs) = distinct xs
by(induct xs) auto

lemma set-remdups [simp]: set (remdups xs) = set xs
by (induct xs) (auto simp add: insert-absorb)

lemma distinct-remdups [iff]: distinct (remdups xs)
THEORY "List"

by (induct xs) auto

lemma distinct-remdups-id: distinct xs ==> remdups xs = xs
by (induct xs, auto)

lemma remdups-id iff distinct [simp]: remdups xs = xs <-> distinct xs
by (metis distinct-remdups distinct-remdups-id)

lemma finite-distinct-list: finite A ==> EX xs. set xs = A & distinct xs
by (metis distinct-remdups finite-list set-remdups)

lemma remdups-eq-nil iff [simp]: (remdups x = []) = (x = [])
by (induct x, auto)

lemma remdups-eq-nil-right iff [simp]: ([] = remdups x) = (x = [])
by (induct x, auto)

lemma length-remdups-leq [iff]: length(remdups xs) <= length xs
by (induct xs) auto

lemma length-remdups-eq [iff]:
  (length (remdups xs) = length xs) = (remdups xs = xs)
apply (induct xs)
apply auto
apply (subgoal-tac length (remdups xs) <= length xs)
apply arith
apply (rule length-remdups-leq)
done

lemma remdups-filter: remdups(filter P xs) = filter P (remdups xs)
apply (induct xs)
apply auto
done

lemma distinct-map:
  distinct(map f xs) = (distinct xs & inj-on f (set xs))
by (induct xs) auto

lemma distinct-filter [simp]: distinct xs ==> distinct (filter P xs)
by (induct xs) auto

lemma distinct-upt [simp]: distinct[i..<j]
by (induct j) auto

lemma distinct-upto [simp]: distinct[i..j]
apply (induct i j rule:upto.induct)
apply (subst upto.simps)
apply (simp)
done
lemma distinct-take[simp]: distinct xs \implies\ distinct (take i xs)
apply(induct xs arbitrary: i)
  apply simp
  apply (case-tac i)
  apply simp-all
  apply (blast dest: in-set-takeD)
done

lemma distinct-drop[simp]: distinct xs \implies\ distinct (drop i xs)
apply(induct xs arbitrary: i)
  apply simp
  apply (case-tac i)
  apply simp-all
done

lemma distinct-list-update:
assumes d: distinct xs and a: a \notin set xs - {xs!i}
shows distinct (xs[i:=a])
proof (cases i < length xs)
  case True
  with a have a \notin set (take i xs @ xs ! i \# drop (Suc i) xs) - {xs!i}
  apply (drule-tac id-take-nth-drop) by simp
  with d True show ?thesis
  apply (simp add: upd-conv-take-nth-drop)
  apply (drule subst [OF id-take-nth-drop]) apply assumption
  apply simp apply (cases a = xs!i) apply simp by blast
next
  case False with d show ?thesis by auto
qed

lemma distinct-concat:
assumes d: distinct xs and \( \forall y\, y \in set xs \implies\ distinct y\)
and \( \forall y\, z\, [ y \in set xs ; z \in set xs ; y \neq z ] \implies\ set y \cap set z = {} \)
shows distinct (concat xs)
using assms by (induct xs) auto

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma distinct-conv-nth:
distinct xs = (\forall i < size xs. \forall j < size xs. i \neq j \implies\ xs!i \neq xs!j)
apply (induct xs, simp, simp)
apply (rule iffI, clarsimp)
apply (case-tac i)
apply (case-tac j, simp)
apply (simp add: set-conv-nth)
apply (case-tac j)
apply (clarsimp simp add: set-conv-nth)
apply (rule conjI)
apply (clarsimp simp add: set-conv-nth)
apply (erule-tac x = 0 in allE, simp)
apply (erule-tac x = Suc i in allE, simp, clarsimp)
apply (erule-tac x = Suc i in allE, simp)
done

lemma nth-eq-iff-index-eq:
[ [ \text{distinct } xs; \ i < \text{ length } xs; \ j < \text{ length } xs ] ] \rightarrow\ (xs!i = xs!j) = (i = j)
by(auto simp: distinct-conv-nth)

lemma set-update-distinct: [ [ \text{distinct } xs; \ n < \text{ length } xs ] ] \rightarrow
set(xs[n := x]) = insert x (set xs - \{xs!n\})
by(auto simp: set-eq-iff in-set-conv-nth nth-list-update nth-eq-iff-index-eq)

lemma distinct-swap[simp]: [ [ \text{i < size } xs; \ j < \text{ size } xs ] ] \rightarrow
distinct(xs[i := xs!j, j := xs!i]) = distinct xs
apply (simp add: distinct-conv-nth nth-list-update)
apply safe
apply metis+
done

lemma set-swap[simp]:
[ [ \text{i < size } xs; \ j < \text{ size } xs ] ] \rightarrow set(xs[i := xs!j, j := xs!i]) = set xs
by_auto

lemma distinct-card: distinct xs ==> card (set xs) = size xs
by (induct xs) auto

lemma card-distinct: card (set xs) = size xs ==> distinct xs
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases x \in set xs)
    case False with Cons show ?thesis by simp
  next
    case True with Cons.prems
    have card (set xs) = Suc (length xs)
      by (simp add: card-insert-if split: split-if_asm)
    moreover have card (set xs) \leq length xs by (rule card-length)
    ultimately have False by simp
    thus ?thesis ..
  qed
  qed

lemma distinct-length-filter: distinct xs \rightarrow length (filter P xs) = card (\{x. P x\}
Int set xs)
by (induct xs) (auto)

lemma not-distinct-decomp: ∼ distinct ws ==⇒ EX xs ys zs y. ws = xs @ [y] @ ys @ [y] @ zs
apply (induct n == length ws arbitrary: ws) apply simp
apply (case-tac ws) apply simp
apply (simp split: split-if-asm)
apply (metis Cons-eq-appendI eq-Nil-appendI split-list)
done

lemma not-distinct-conv-prefix:
defines dec as xs y ys ≡ y ∈ set xs ∧ distinct xs ∧ as = xs @ y # ys
shows ∼distinct as ←→ (∃ xs y ys. dec as xs y ys) (is ?L = ?R)
proof
assume ?L then show ?R
proof (induct length as arbitrary: as rule: less-induct)
case less
obtain xs ys y where decomp: as = (xs @ y # ys) @ y # zs
using not-distinct-decomp[OF less.prems] by auto
show ?case
proof (cases distinct (xs @ y # ys))
case True
with decomp have dec as (xs @ y # ys) y zs by (simp add: dec-def)
then show ?thesis by blast
next
case False
with less decomp obtain xs’ y’ ys’ where decomp (xs @ y # ys) xs’ y’ ys’
by atomize-elim auto
with decomp have dec as xs’ y’ (ys’ @ y # zs) by (simp add: dec-def)
then show ?thesis by blast
qed
qed
qed (auto simp: dec-def)

lemma distinct-product:
distinct xs ==⇒ distinct ys ==⇒ distinct (List.product xs ys)
by (induct xs) (auto intro: inj-onI simp add: distinct-map)

lemma distinct-product-lists:
assumes ∀ xs ∈ set xss. distinct xs
shows distinct (product-lists xss)
using assms proof (induction xss)
case (Cons xs xss) note * = this
then show ?case
proof (cases product-lists xss)
case Nil then show ?thesis by (induct xs) simp-all
next
case (Cons ps pss) with * show ?thesis
by (auto intro!: inj-onI distinct-concat simp add: distinct-map)
qed
THEORY "List"

qed simp

lemma length-remdups-concat:
  length (remdups (concat xss)) = card (⋃xs∈set xss. set xs)
  by (simp add: distinct-card [symmetric])

lemma length-remdups-card-conv: length(remdups xs) = card(set xs)
proof –
  have xs: concat[xs] = xs by simp
  from length-remdups-concat[of xs] show thesis unfolding xs by simp
qed

lemma remdups-remdups:
  remdups (remdups xs) = remdups xs
  by (induct xs) simp-all

lemma distinct-butlast:
  assumes distinct xs
  shows distinct (butlast xs)
proof (cases xs)
  case False
  from ⟨xs ≠ []⟩ obtain ys y where xs = ys @ [y] by (cases xs rule: rev-cases)
  auto
  with ⟨distinct xs⟩ show thesis by simp
qed (auto)

lemma remdups-map-remdups:
  remdups (map f (remdups xs)) = remdups (map f xs)
  by (induct xs) simp-all

lemma distinct-zipI1:
  assumes distinct xs
  shows distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'a list and ys' :: 'b list and n
  assume length xs' = length ys'
  assume xs' = take n xs
  with assms have distinct xs' by simp
  with ⟨length xs' = length ys'⟩ show distinct (zip xs' ys')
    by (induct xs' ys' rule: list-induct2) (auto elim: in-set-zipE)
qed

lemma distinct-zipI2:
  assumes distinct ys
  shows distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs' :: 'b list and ys' :: 'a list and n
  assume length xs' = length ys'
  assume ys' = take n ys
with assms have distinct $y_1$ by simp
with ⟨length $x_1$ = length $y_1$⟩ show distinct (zip $x_1$ $y_1$)
  by (induct $x_1$ $y_1$ rule: list-induct2) (auto elim: in-set-zipE)
qed

lemma set-take-disj-set-drop-if-distinct:
  distinct $v$ $\implies$ $i \leq j \implies$ set (take $i$ $v$) $\cap$ set (drop $j$ $v$) = {}
by (auto simp: in-set-conv-nth distinct-conv-nth)

lemma distinct-singleton: distinct [a]
by simp

lemma distinct-length-2-or-more:
distinct (a # b # x) $\iff$ (a # b ∧ distinct (a # x) ∧ distinct (b # x))
by force

lemma remdups-adj-Cons: remdups-adj (a # x) =
  (case remdups-adj x of [] $\Rightarrow$ [a] | y # x $\Rightarrow$ if x = y then y # x else x # y # x)
by (induct x arbitrary: x) (auto split: list.splits)

lemma remdups-adj-rev[simp]: remdups-adj (rev x) = rev (remdups-adj x)
by (induct x rule: remdups-adj.induct, simp-all add: remdups-adj-append-two)

lemma remdups-adj-set[simp]: set (remdups-adj x) = x
by (induct x rule: remdups-adj.induct, simp-all)

lemma remdups-adj-Cons-alt[simp]: x # tl (remdups-adj (x # x)) = remdups-adj (x # x)
by (induct x rule: remdups-adj.induct, auto)

lemma remdups-adj-distinct: distinct x $\implies$ remdups-adj x = x
by (induct x rule: remdups-adj.induct, simp-all)
THEORY "List"

lemma remdups-adj-append:
  remdups-adj (xs1 @ x # xs2) = remdups-adj (xs1 @ [x]) @ tl (remdups-adj (x # xs2))
  by (induct xs1 rule: remdups-adj.induct, simp-all)

lemma remdups-adj-singleton:
  remdups-adj xs = [x] =⇒ xs = replicate (length xs) x
  by (induct xs rule: remdups-adj.induct, auto split: split-if-asm)

lemma remdups-adj-map-injective:
  assumes inj f
  shows remdups-adj (map f xs) = map f (remdups-adj xs)
  by (induct xs rule: remdups-adj.induct, auto simp add: injD [OF assms])

67.1.24 List summation: listsum and ∑

lemma (in monoid-add) listsum-simps [simp]:
  listsum [] = 0
  listsum (x # xs) = x + listsum xs
  by (simp-all add: listsum-def)

lemma (in monoid-add) listsum-append [simp]:
  listsum (xs @@ ys) = listsum xs + listsum ys
  by (induct xs) (simp-all add: add.assoc)

lemma (in comm-monoid-add) listsum-rev [simp]:
  listsum (rev xs) = listsum xs
  by (simp add: listsum-def foldr-fold fold-rev fun-eq-iff ac-simps)

lemma (in monoid-add) fold-plus-listsum-rev:
  fold plus xs = plus (listsum (rev xs))

proof
  fix x
  have fold plus xs x = fold plus xs (x + 0) by simp
  also have ... = fold plus (x # xs) 0 by simp
  also have ... = foldr plus (rev xs @ [x]) 0 by (simp add: foldr-conv-fold)
  also have ... = listsum (rev xs @ [x]) by (simp add: listsum-def)
  also have ... = listsum (rev xs) + listsum [x] by simp
  finally show fold plus xs x = listsum (rev xs) + x by simp
qed

Some syntactic sugar for summing a function over a list:

syntax
  -listsum :: pattrn => 'a list => 'b => 'b  ((3SUM -<- -.) [0, 51, 10] 10)
syntax (xsymbols)
  -listsum :: pattrn => 'a list => 'b => 'b  ((3∑ -<- -.) [0, 51, 10] 10)
syntax (HTML output)
  -listsum :: pattrn => 'a list => 'b => 'b  ((3∑ -<- -.) [0, 51, 10] 10)
translators — Beware of argument permutation!

\[ \sum_{x \leftarrow xs} b =\text{CONST listsum (CONST map (%x. b) xs)} \]

**lemma** (in comm-monoid-add) listsum-map-remove1:

\[ x \in \text{set xs} \implies \text{listsum (map f xs)} = f x + \text{listsum (map f (remove1 x xs))} \]

by (induct xs) (auto simp add: ac-simps)

**lemma** (in monoid-add) size-list-conv-listsum:

\[ \text{size-list f xs = listsum (map f xs) + size xs} \]

by (induct xs) auto

**lemma** (in monoid-add) length-concat:

\[ \text{length (concat xss) = listsum (map length xss)} \]

by (induct xss) simp-all

**lemma** (in monoid-add) length-product-lists:

\[ \text{length (product-lists xss) = foldr op * (map length xss)} \]

**proof** (induct xss)

\[ \text{case (Cons xs xss)} \text{ then show ?case by (induct xs) (auto simp: length-concat o-def)} \]

**qed** simp

**lemma** (in monoid-add) listsum-map-filter:

\[ \text{assumes } \forall x. x \in \text{set xs} \implies \neg P x \implies f x = 0 \]

\[ \text{shows} \ \text{listsum (map f (filter P xs)) = listsum (map f xs)} \]

**using** assms by (induct xs) auto

**lemma** (in comm-monoid-add) distinct-listsum-conv-Setsum:

\[ \text{distinct xs \implies listsum xs = Setsum (set xs)} \]

by (induct xs) simp-all

**lemma** listsum-eq-0-nat-iff-nat [simp]:

\[ \text{listsum ns = (0::nat) \iff (\forall n \in \text{set ns} \implies n = 0)} \]

by (induct ns) simp-all

**lemma** member-le-listsum-nat:

\[ (n :: nat) \in \text{set ns} \implies n \leq \text{listsum ns} \]

by (induct ns) auto

**lemma** elem-le-listsum-nat:

\[ k < \text{size ns} \implies \text{ns ! k \leq listsum (ns :: nat list)} \]

by (rule member-le-listsum-nat) simp

**lemma** listsum-update-nat:

\[ k < \text{size ns} \implies \text{listsum (ns[k := (n::nat)]) = listsum ns + n - ns ! k} \]

apply (induct ns arbitrary:k)
apply (auto split:nat.split)
apply (drule elem-le-listsum-nat)
apply arith
done

lemma (in monoid-add) listsum-triv:
  \( \sum_{x \leftarrow xs} r = \text{of-nat} (\text{length} xs) \ast r \)
  by (induct xs) (simp-all add: distrib-right)

lemma (in monoid-add) listsum-0 [simp]:
  \( \sum_{x \leftarrow xs} 0 = 0 \)
  by (induct xs) (simp-all add: distrib-right)

For non-Abelian groups \( xs \) needs to be reversed on one side:

lemma (in ab-group-add) uminus-listsum-map:
  \( - \text{listsum} (\text{map} f xs) = \text{listsum} (\text{map} (\text{uminus} \circ f) xs) \)
  by (induct xs) simp-all

lemma (in comm-monoid-add) listsum-addf:
  \( \sum_{x \leftarrow xs} f x + g x = \text{listsum} (\text{map} f xs) + \text{listsum} (\text{map} g xs) \)
  by (induct xs) (simp-all add: algebra-simps)

lemma (in ab-group-add) listsum-subtractf:
  \( \sum_{x \leftarrow xs} f x - g x = \text{listsum} (\text{map} f xs) - \text{listsum} (\text{map} g xs) \)
  by (induct xs) (simp-all add: algebra-simps)

lemma (in semiring-0) listsum-const-mult:
  \( \sum_{x \leftarrow xs} c \ast f x = c \ast (\sum_{x \leftarrow xs} f x) \)
  by (induct xs) (simp-all add: algebra-simps)

lemma (in semiring-0) listsum-mult-const:
  \( \sum_{x \leftarrow xs} f x \ast c = (\sum_{x \leftarrow xs} f x) \ast c \)
  by (induct xs) (simp-all add: algebra-simps)

lemma (in ordered-ab-group-add-ubs) listsum-abs:
  \( |\text{listsum} xs| \leq \text{listsum} (\text{map} \text{abs} xs) \)
  by (induct xs) (simp-all add: order-trans [OF abs-triangle-ineq])

lemma listsum-mono:
  fixes \( f, g :: \text{a} \Rightarrow \text{b} :: \{\text{monoid-add}, \text{ordered-ab-semigroup-add}\} \)
  shows \( (\lambda x. \ x \in \text{set} \ xs \Rightarrow \ f x \leq g x) \Rightarrow (\sum_{x \leftarrow xs} f x) \leq (\sum_{x \leftarrow xs} g x) \)
  by (induct xs) (simp, simp add: add-mono)

lemma (in monoid-add) listsum-distinct-conv-setsum-set:
  \( \text{distinct} \ xs \Rightarrow \text{listsum} (\text{map} f xs) = \text{setsum} f (\text{set} \ xs) \)
  by (induct xs) simp-all

lemma (in monoid-add) interv-listsum-cone-setsum-set-nat:
  \( \text{listsum} (\text{map} f [m..n]) = \text{setsum} f (\text{set} [m..n]) \)
  by (simp add: listsum-distinct-conv-setsum-set)
lemma (in monoid-add) inter-listsum-cone-setsum-set-int:
  listsum (map f [k..l]) = setsum f (set [k..l])
  by (simp add: listsum-distinct-cone-setsum-set)

General equivalence between listsum and setsum

lemma (in monoid-add) listsum-setsum-nth:
  listsum xs = (∑ i = 0..< length xs. xs ! i)
  using inter-listsum-cone-setsum-set-nat [of op ! xs 0 length xs] by (simp add: map-nth)

67.1.25 insert

lemma in-set-insert [simp]:
  x ∈ set xs ==> List.insert x xs = xs
  by (simp add: List.insert-def)

lemma not-in-set-insert [simp]:
  x /∈ set xs ==> List.insert x xs = x # xs
  by (simp add: List.insert-def)

lemma insert-Nil [simp]:
  List.insert x [] = [x]
  by simp

lemma set-insert [simp]:
  set (List.insert x xs) = insert x (set xs)
  by (auto simp add: List.insert-def)

lemma distinct-insert [simp]:
  distinct (List.insert x xs) = distinct xs
  by (simp add: List.insert-def)

lemma insert-remdups:
  List.insert x (remdups xs) = remdups (List.insert x xs)
  by (simp add: List.insert-def)

67.1.26 List.union

This is all one should need to know about union:

lemma set-union[simp]: set (List.union xs ys) = set xs ∪ set ys
  unfolding List.union-def
  by(induct xs arbitrary: ys) simp-all

lemma distinct-union[simp]: distinct(List.union xs ys) = distinct ys
  unfolding List.union-def
  by(induct xs arbitrary: ys) simp-all
67.1.27  List.find

lemma find-None-iff: List.find P xs = None ↔ ¬ (∃ x. x ∈ set xs ∧ P x)
proof (induction xs)
case Nil thus ?case by simp
next
case (Cons x xs) thus ?case by (fastforce split: if-splits)
qed

lemma find-Some-iff:
List.find P xs = Some x ⟷ (∃ i < length xs. P (xs!i) ∧ x = xs!i ∧ (∀ j < i. ¬ P (xs!j)))
proof (induction xs)
case Nil thus ?case by simp
next
case (Cons x xs) thus ?case
  apply (auto simp: nth-Cons' split: if-splits)
  using diff-Suc-1 [unfolded One-nat-def] less-Suc-eq-0-disj by fastforce
qed

lemma find-cong:
assumes xs = ys and (∀ x. x ∈ set ys =⇒ P x = Q x)
shows List.find P xs = List.find Q ys
proof (cases List.find P xs)
case None thus ?thesis using assms (metis find-None-iff assms)
next
case (Some x) hence List.find Q ys = Some x using assms
  by (auto simp add: find-Some-iff)
thus ?thesis using Some by auto
qed

lemma find-dropWhile:
List.find P xs = (case dropWhile (Not ◦ P) xs
  of [] ⇒ None
    | x # - ⇒ Some x)
by (induct xs) simp-all

67.1.28  List.extract

lemma extract-None-iff: List.extract P xs = None ↔ ¬ (∃ x ∈ set xs. P x)
by (auto simp: extract-def dropWhile-eq-Cons-conv split: list_splits)
  (metis in-set-conv-decomp)

lemma extract-SomeE:
List.extract P xs = Some (ys, y, zs) =⇒
xs = ys @ y # zs ∧ P y ∧ ¬ (∃ y ∈ set ys. P y)
by (auto simp: extract-def dropWhile-eq-Cons-conv split: list_splits)

lemma extract-Some-iff:
THEORY "List"

List.extract P xs = Some (ys, y, zs) 

xs = ys @ y # zs ∧ P y ∧ (∃ y ∈ set ys. P y)

by (auto simp: extract-def dropWhile-eq-Cons-conv dest: set-takeWhileD split: list.splits)

lemma extract-Nil-code[code]: List.extract P [] = None

by (simp add: extract-def)

lemma extract-Cons-code[code]:
List.extract P (x # xs) = (if P x then Some ([], x, xs) else
(case List.extract P xs of
  None ⇒ None |
  Some (ys, y, zs) ⇒ Some (x#ys, y, zs)))

by (auto simp add: extract-def comp-def split: list.splits)

(metis dropWhile-eq-Nil-conv list.distinct(1))

67.1.29 remove1

lemma remove1-append:
remove1 x (xs @ ys) =
(if x ∈ set xs then remove1 x xs @ ys else xs @ remove1 x ys)

by (induct xs) auto

lemma remove1-commute: remove1 x (remove1 y zs) = remove1 y (remove1 x zs)

by (induct zs) auto

lemma in-set-remove1 [simp]:
   a ≠ b ⇒ a : set(remove1 b xs) = (a : set xs)

apply (induct xs)
apply auto

done

lemma set-remove1-subset: set(remove1 x xs) ⊆ set xs

apply (induct xs)
apply simp
apply simp
apply blast

done

lemma set-remove1-eq [simp]: distinct xs ==> set(remove1 x xs) = set xs - {x}

apply (induct xs)
apply simp
apply simp
apply blast

done

lemma length-remove1:

length(remove1 x xs) = (if x : set xs then length xs - 1 else length xs)

apply (induct xs)
apply (auto dest!: length-pos-if-in-set)
done

lemma remove1-filter-not[simp]:
\neg P \, x \implies remove1 \, x \, (filter \, P \, xs) = filter \, P \, xs
by (induct xs) auto

lemma filter-remove1:
filter \, Q \, (remove1 \, x \, xs) = remove1 \, x \, (filter \, Q \, xs)
by (induct xs) auto

lemma notin-set-remove1[simp]: x \in\notin set \, xs == x \in\notin set \, (remove1 \, y \, xs)
apply (insert set-remove1-subset)
apply fast
done

lemma distinct-remove1[simp]: distinct \, xs \implies distinct \, (remove1 \, x \, xs)
by (induct xs) simp-all

lemma remove1-remdups:
distinct \, xs \implies remove1 \, x \, (remdups \, xs) = remdups \, (remove1 \, x \, xs)
by (induct xs) simp-all

lemma remove1-idem:
assumes x \notin set \, xs
shows remove1 \, x \, xs = xs
using assms by (induct xs) simp-all

67.1.30 removeAll
lemma removeAll-filter-not-eq:
removeAll \, x = filter \, (\lambda y. \, x \neq y)
proof
fix xs
show removeAll \, x \, xs = filter \, (\lambda y. \, x \neq y) \, xs
by (induct xs) auto
qed

lemma removeAll-append[simp]:
removeAll \, x \, (xs @ ys) = removeAll \, x \, xs @ removeAll \, x \, ys
by (induct xs) auto

lemma set-removeAll[simp]: set\, (removeAll \, x \, xs) = set \, xs \, - \, \{x\}
by (induct xs) auto

lemma removeAll-id[simp]: x \notin set \, xs \implies removeAll \, x \, xs = xs
by (induct xs) auto
lemma removeAll-filter-not[simp]:
\neg P x \implies removeAll x (filter P xs) = filter P xs
by (induct xs) auto

lemma distinct-removeAll:
distinct xs \implies distinct (removeAll x xs)
by (simp add: removeAll-filter-not-eq)

lemma distinct-remove1-removeAll:
distinct xs \implies remove1 x xs = removeAll x xs
by (induct xs) simp-all

lemma map-removeAll-inj-on:
inj-on f \((\text{insert } x \ (\text{set } xs))\) \implies
map f (removeAll x xs) = removeAll (f x) (map f xs)
by (induct xs) (simp-all add: inj-on-def)

lemma map-removeAll-inj:
inj f \implies
map f (removeAll x xs) = removeAll (f x) (map f xs)
by (rule map-removeAll-inj-on, erule subset-inj-on, rule subset-UNIV)

67.1.31 replicate

lemma length-replicate [simp]: length (replicate n x) = n
by (induct n) auto

lemma Ex-list-of-length: \exists xs. length xs = n
by (rule exI[of - replicate n undefined]) simp

lemma map-replicate [simp]: map f (replicate n x) = replicate n (f x)
by (induct n) auto

lemma map-replicate-const:
map (\lambda x. k) lst = replicate (length lst) k
by (induct lst) auto

lemma replicate-app-Cons-same:
(replicate n x) @ (x # xs) = x # replicate n x @ xs
by (induct n) auto

lemma rev-replicate [simp]: rev (replicate n x) = replicate n x
apply (induct n, simp)
apply (simp add: replicate-app-Cons-same)
done

lemma replicate-add: replicate \((n + m) x\) = replicate n x @ replicate m x
by (induct n) auto

Courtesy of Matthias Daum:

lemma append-replicate-commute:
replicate \(n\) \(x\) @ replicate \(k\) \(x\) = replicate \(k\) \(x\) @ replicate \(n\) \(x\)

apply (simp add: replicate-add [THEN sym])
apply (simp add: add.commute)
done

Courtesy of Andreas Lochbihler:

lemma filter-replicate:
filter \(P\) (replicate \(n\) \(x\)) = (if \(P\) \(x\) then replicate \(n\) \(x\) else [])
by (induct \(n\)) auto

lemma hd-replicate [simp]: \(n\) \(\neq\) 0 ==> \(hd\) (replicate \(n\) \(x\)) = \(x\)
by (induct \(n\)) auto

lemma tl-replicate [simp]: \(tl\) (replicate \(n\) \(x\)) = replicate \((n - 1)\) \(x\)
by (induct \(n\)) auto

lemma last-replicate [simp]: \(n\) \(\neq\) 0 ==> \(last\) (replicate \(n\) \(x\))! = \(x\)
by (atomize (full), induct \(n\)) auto

lemma nth-replicate [simp]: \(i\) < \(n\) ==> (replicate \(n\) \(x\))!\(i\) = \(x\)
apply (induct \(n\) arbitrary: \(i\), simp)
apply (simp add: nth-Cons split: nat.split)
done

Courtesy of Matthias Daum (2 lemmas):

lemma take-replicate [simp]: take \(i\) (replicate \(k\) \(x\)) = replicate \((\text{min} \(i\) \(k\))\) \(x\)
apply (case-tac \(k\) \(\leq\) \(i\))
apply (simp add: min-def)
apply (drule not-leE)
apply (simp add: min-def)
apply (subgoal-tac replicate \(k\) \(x\) = replicate \(i\) \(x\) @ replicate \((k - i)\) \(x\))
apply simp
apply (simp add: replicate-add [symmetric])
done

lemma drop-replicate [simp]: drop \(i\) (replicate \(k\) \(x\)) = replicate \((k - i)\) \(x\)
apply (induct \(k\) arbitrary: \(i\))
apply simp
apply clarsimp
apply (case-tac \(i\))
apply simp
apply clarsimp
done

lemma set-replicate-Suc: set (replicate \((\text{Suc} \(n\))\) \(x\)) = \(\{x\}\)
by (induct \(n\)) auto

lemma set-replicate [simp]: \(n\) \(\neq\) 0 ==> set (replicate \(n\) \(x\)) = \(\{x\}\)
by (fast dest!: not0-implies-Suc intro!: set-replicate-Suc)
lemma set-replicate-conv-if: set (replicate n x) = (if n = 0 then {} else {x})
by auto

lemma in-set-replicate[simp]: (x : set (replicate n y)) = (x = y & n ≠ 0)
by (simp add: set-replicate-conv-if)

lemma Ball-set-replicate[simp]:
(ALL x : set(replicate n a). P x) = (P a | n=0)
by (simp add: set-replicate-conv-if)

lemma Bex-set-replicate[simp]:
(EX x : set(replicate n a). P x) = (P a & n≠0)
by (simp add: set-replicate-conv-if)

lemma replicate-append-same:
replicate i x @ [x] = x # replicate i x
by (induct i) simp-all

lemma map-replicate-trivial:
map (λi. x) [0..<i] = replicate i x
by (induct i) (simp-all add: replicate-append-same)

lemma concat-replicate-trivial[simp]:
concat (replicate i []) = []
by (induct i) (auto simp add: map-replicate-const)

lemma replicate-empty[simp]: (replicate n x = []) ←→ n=0
by (induct n) auto

lemma empty-replicate[simp]: ([] = replicate n x) ←→ n=0
by (induct n) auto

lemma replicate-eq-replicate[simp]:
(replicate m x = replicate n y) ←→ (m=n & (m≠0 → x=y))
apply (induct m arbitrary: n)
apply simp
apply (induct-tac n)
apply auto
done

lemma replicate-length-filter:
replicate (length (filter (λy. x = y) xs)) x = filter (λy. x = y) xs
by (induct xs) auto

lemma comm-append-are-replicate:
fixes xs ys :: 'a list
assumes xs ≠ [] ys ≠ []
assumes xs @ ys = ys @ xs
shows \( \exists m \ n \ zs. \ \text{concat} (\text{replicate} \ m \ zs) = \text{xs} \land \text{concat} (\text{replicate} \ n \ zs) = \text{ys} \)

using assms

proof (induct length \((\text{xs} @ \text{ys})\) arbitrary; \text{xs} \ ys \ \text{rule:} \ \text{less-induct})

  case less

  def \text{xs}' \equiv \text{if} (\text{length} \text{xs} \leq \text{length} \text{ys}) \text{ then } \text{xs} \ \text{ else } \text{ys}
  and \text{ys}' \equiv \text{if} (\text{length} \text{xs} \leq \text{length} \text{ys}) \text{ then } \text{ys} \ \text{ else } \text{xs}

  then have \prems': \text{length} \text{xs}' \leq \text{length} \text{ys}'
    \text{xs}' @ \text{ys}' = \text{ys}' @ \text{xs}'
    and \text{xs}' \neq []
    and \text{len:} \text{length} (\text{xs} @ \text{ys}) = \text{length} (\text{xs}' @ \text{ys}')
  using less by (auto intro: less.hyps)

  from \prems'
  obtain \text{ws} \ \text{where} \ \text{ys}' = \text{xs}' @ \text{ws}
    by (auto simp: append-eq-append-conv2)

  have \( \exists m \ n \ zs. \ \text{concat} (\text{replicate} \ m \ zs) = \text{xs}' \land \text{concat} (\text{replicate} \ n \ zs) = \text{ys}' \)
  proof (cases \text{ws} = [])
    case True
    then have \text{concat} (\text{replicate} 1 \text{xs}') = \text{xs}'
      and \text{concat} (\text{replicate} 1 \text{xs}') = \text{ys}'
      using \text{ys}' = \text{xs}' @ \text{ws} \ \text{by} \ \text{auto}
    then show \?thesis \text{ by} \text{ blast}
  next
    case False
    from \( \text{ys}' = \text{xs}' @ \text{ws} \) \ and \( \text{zs}' @ \text{ys}' = \text{ys}' @ \text{xs}' \)
    have \( \text{xs}' @ \text{ws} = \text{us} @ \text{xs}' \) \ by simp
    then have \( \exists m \ n \ zs. \ \text{concat} (\text{replicate} \ m \ zs) = \text{xs}' \land \text{concat} (\text{replicate} \ n \ zs) = \text{ws} \)
      using False \ and \( \text{zs}' \neq [] \) \ and \( \text{ys}' = \text{xs}' @ \text{ws} \) \ and \text{len}
      by (intro less.hyps) \text{auto}
    then obtain \( m \ n \ zs \) \ \text{where} \( *: \ \text{concat} (\text{replicate} \ m \ zs) = \text{xs}' \)
      and \text{concat} (\text{replicate} \ n \ zs) = \text{ws} \ \text{by} \ \text{blast}
    then have \text{concat} (\text{replicate} (m + n) \ zs) = \text{ys}'
      using \( \text{ys}' = \text{xs}' @ \text{ws} \)
      by (simp add; \text{replicate-add})
    with \( * \) show \?thesis \text{ by} \text{ blast}
  qed
  then show \(?case\)
    using \text{xs}'-def \text{ys}'-def \ \text{by} \text{ meson}
  qed

lemma \text{comm-append-is-replicate:}
  fixes \text{xs} \ \text{ys} :: 'a list
  assumes \text{xs} \neq [] \ \text{ys} \neq []
  assumes \text{xs} @ \text{ys} = \text{ys} @ \text{xs}
  shows \( \exists n \ zs. \ n > 1 \land \text{concat} (\text{replicate} \ n \ zs) = \text{xs} @ \text{ys} \)
proof
obtain \( m \ n \ zs \) where \( \text{concat} \ (\text{replicate} \ m \ zs) = xs \)
and \( \text{concat} \ (\text{replicate} \ n \ zs) = ys \)
using \text{comm-append-are-replicate}[\text{of} \ xs \ ys, \ \text{OF} \ \text{assms}] \text{ by blast}
then have \( m + n > 1 \) and \( \text{concat} \ (\text{replicate} \ (m+n) \ zs) = xs @ ys \)
using \( \text{xs} \neq [] \) and \( \text{ys} \neq [] \)
by (auto \ simp: \text{replicate-add})
then show ?thesis by blast
qed

lemma \text{Cons-repeat-eq}:
x \# xs = \text{replicate} \ n \ y \longleftrightarrow x = y \land n > 0 \land xs = \text{replicate} \ (n - 1) \ x
by (induct n) auto

lemma \text{replicate-length-same}:
(\forall y \in \text{set} \ xs. \ y = x) \Longrightarrow \text{replicate} \ (\text{length} \ xs) \ x = xs
by (induct xs) simp-all

lemma \text{foldr-repeat} [simp]:
foldr f (\text{replicate} \ n \ x) = f x ^^ n
by (induct n) (simp-all)

lemma \text{fold-repeat} [simp]:
fold f (\text{replicate} \ n \ x) = f x ^^ n
by (subst \text{foldr-fold} [symmetric]) simp-all

67.1.32  \text{enumerate}

lemma \text{enumerate-simps} [simp, code]:
\text{enumerate} \ n [] = []
\text{enumerate} \ n \ (x \# xs) = (n, x) \# \text{enumerate} \ (\text{Suc} \ n) \ xs
apply (auto \ simp \ add: \text{enumerate-eq-zip} \ not-le)
apply (cases \ n < n + \text{length} \ xs)
apply (auto \ simp \ add: \text{upt-cone-Cons})
done

lemma \text{length-enumerate} [simp]:
\text{length} \ (\text{enumerate} \ n \ xs) = \text{length} \ xs
by (simp \ add: \text{enumerate-eq-zip})

lemma \text{map-fst-enumerate} [simp]:
\text{map} \ \text{fst} \ (\text{enumerate} \ n \ xs) = [n..<n + \text{length} \ xs]
by (simp \ add: \text{enumerate-eq-zip})

lemma \text{map-snd-enumerate} [simp]:
\text{map} \ \text{snd} \ (\text{enumerate} \ n \ xs) = xs
by (simp \ add: \text{enumerate-eq-zip})
lemma in-set-enumerate-eq:
\[ p \in \text{set} (\text{enumerate } n \text{ xs}) \iff n \leq \text{fst } p \land \text{fst } p < \text{length } \text{xs} + n \land \text{nth } \text{xs} (\text{fst } p - n) = \text{snd } p \]
proof -
{ fix m
  assume n \leq m
  moreover assume m < \text{length } \text{xs} + n
  ultimately have [n..<n + \text{length } \text{xs}] ![m - n] = m \land
  \text{xs} ![m - n] = \text{xs} ![m - n] \land m - n < \text{length } \text{xs} by auto
  then have \exists q. [n..<n + \text{length } \text{xs}] ![q] = m \land
  \text{xs} ![q] = \text{xs} ![m - n] \land q < \text{length } \text{xs} ..
} then show ?thesis by (cases p) (auto simp add: enumerate-eq-zip in-set-zip)
qed

lemma nth-enumerate-eq:
assumes m < \text{length } \text{xs}
shows \text{enumerate } n \text{ xs} ![m] = (n + m, \text{xs} ![m])
using assms by (simp add: enumerate-eq-zip)

lemma enumerate-replicate-eq:
\text{enumerate } n (\text{replicate } m a) = \text{map } (\lambda q. (q, a)) [n..<n + m]
by (rule pair-list-eqI)
  (simp-all add: enumerate-eq-zip comp-def map-replicate-const)

lemma enumerate-Suc-eq:
\text{enumerate } (\text{Suc } n) \text{ xs} = \text{map } (\text{apfst } \text{Suc}) (\text{enumerate } n \text{ xs})
by (rule pair-list-eqI)
  (simp-all add: not-le, simp del: map-map [simp del] add: map-Suc-upt map-map [symmetric])

lemma distinct-enumerate [simp]:
distinct (\text{enumerate } n \text{ xs})
by (simp add: enumerate-eq-zip distinct-zipI1)

67.1.33 \text{rotate\textbar{}I and rotate}

lemma rotate0 [simp]: rotate 0 = id
by (simp add: rotate-def)

lemma rotate-Suc [simp]: rotate (Suc n) \text{xs} = rotate1 (rotate \text{n xs})
by (simp add: rotate-def)

lemma rotate-add:
\text{rotate} (m + n) = \text{rotate} \text{m o rotate} \text{n}
by (simp add: rotate-def funpow-add)

lemma rotate-rotate: rotate \text{m} (\text{rotate } n \text{ xs}) = \text{rotate } (m + n) \text{xs}
by (simp add: rotate-add)
lemma rotate1-rotate-swap: rotate1 (rotate n xs) = rotate n (rotate1 xs)
by(simp add:rotate-def funpow-swap1)

lemma rotate1-length01[simp]: length xs <= 1 ==> rotate1 xs = xs
by(cases xs) simp-all

lemma rotate-length01[simp]: length xs <= 1 ==> rotate n xs = xs
apply(induct n)
apply simp
apply (simp add:rotate-def)
done

lemma rotate1-hd-tl: xs != [] ==> rotate1 xs = tl xs @ [hd xs]
by (cases xs) simp-all

lemma rotate-drop-take:
  rotate n xs = drop (n mod length xs) xs @ take (n mod length xs) xs
apply(induct n)
apply simp
apply(simp add:rotate-def)
apply (cases xs = [])
apply (simp)
apply(case-tac n mod length xs = 0)
apply(simp add:mod-Suc)
apply(simp add:rotate1-hd-tl drop-Suc take-Suc)
apply(simp add:mod-Suc rotate1-hd-tl drop-Suc[symmetric] drop-tl[symmetric]
      take-hd-drop linorder-not-le)
done

lemma rotate-conv-mod: rotate n xs = rotate (n mod length xs) xs
by(simp add:rotate-drop-take)

lemma rotate-id[simp]: n mod length xs = 0 ==> rotate n xs = xs
by(simp add:rotate-drop-take)

lemma length-rotate1[simp]: length(rotate1 xs) = length xs
by (cases xs) simp-all

lemma length-rotate[simp]: length(rotate n xs) = length xs
by (induct n arbitrary: xs) (simp-all add:rotate-def)

lemma distinct1-rotate[simp]: distinct(rotate1 xs) = distinct xs
by (cases xs) auto

lemma distinct-rotate[simp]: distinct(rotate n xs) = distinct xs
by (induct n) (simp-all add:rotate-def)

lemma rotate-map: rotate n (map f xs) = map f (rotate n xs)
by(simp add:rotate-drop-take take-map drop-map)
lemma `set-rotate1 [simp]`: `set(rotate1 xs) = set xs`  
by (cases xs) auto

lemma `set-rotate [simp]`: `set(rotate n xs) = set xs`  
by (induct n) (simp-all add:rotate-def)

lemma `rotate1-is-Nil-conv [simp]`: `(rotate1 xs = []) = (xs = [])`  
by (cases xs) auto

lemma `rotate-is-Nil-conv [simp]`: `(rotate n xs = []) = (xs = [])`  
by (induct n) (simp-all add:rotate-def)

lemma `rotate-rev`:  
\[ \text{rotate } n \text{ (}\text{rev } \text{xs}) = \text{rev } (\text{rotate } (\text{length } \text{xs} - (n \mod \text{length } \text{xs})) \text{ xs}) \]

apply (simp add: rotate-drop-take rev-drop rev-take)
apply (cases length xs = 0)
apply simp
apply (cases n mod length xs = 0)
apply simp
apply (simp add: rotate-drop-take rev-drop rev-take)
done

lemma `hd-rotate-conv-nth [simp]`: `xs \neq [] =\Rightarrow hd(rotate n xs) = \text{xs!}(n \mod \text{length } \text{xs})`  
apply (simp add: rotate-drop-take hd-append hd-drop-conv-nth hd-conv-nth)
apply (subgoal-tac length xs \neq 0)
prefer 2 apply simp
using mod-less-divisor[of length xs n] by arith

67.1.34  sublist — a generalization of `op !` to sets

lemma `sublist-empty [simp]`: `sublist xs {} = []`  
by (auto simp add: sublist-def)

lemma `sublist-nil [simp]`: `sublist [] A = []`  
by (auto simp add: sublist-def)

lemma `length-sublist`:  
\[ \text{length(sublist } \text{xs } \text{I}) = \text{card}\{i. i < \text{length } \text{xs} \land i : \text{I}\} \]

by (simp add: sublist-def length-filter-conv-card cong:conj-cong)

lemma `sublist-shift-lemma-Suc`:  
\[ \text{map fst } (\text{filter}(\%p. P(Suc(snd p)))) (\text{zip } \text{xs } \text{is})) = \text{map fst } (\text{filter}(\%p. P(snd p))) (\text{zip } \text{xs } (\text{map Suc } \text{is})) \]

apply (induct xs arbitrary: is)
apply simp
apply (case-tac is)
apply simp
apply simp
apply simp
done

lemma sublist-shift-lemma:
  map fst ([p< z: zip xs i.. i + length xs] . snd p : A =
  map fst ([p< z: zip xs [0.. i + length xs] . snd p + i : A]
by (induct xs rule: rev-induct) (simp-all add: add.commute)

lemma sublist-append:
  sublist (l @ l') A = (if A then [x] else []) @ sublist l {j. j + length l : A}
apply (induct l' rule: rev-induct)
apply (simp add: sublist-def)
apply (simp add: sublist-shift-lemma)
apply (simp add: add.commute)
done

lemma sublist-Cons:
  sublist (x # l) A = (if A then [x] else []) @ sublist l {j. Suc j : A}
apply (induct xs arbitrary: I)
apply (auto simp: sublist-Cons nth-Cons split:nat.split dest!: gr0_implies_Suc)
done

lemma set-sublist: set(sublist xs I) = {x! i. i < length xs ∧ i ∈ I}
apply (induct xs arbitrary: I)
apply (auto simp add: set-sublist)
apply (auto simp: sublist-Cons nth-Cons split:nat.split dest!: gr0_implies_Suc)
done

lemma set-sublist-subset: set(sublist xs I) ⊆ set xs
by (auto simp add: set-sublist)

lemma notin-set-sublistI [simp]: x ∉ set xs ⇒ x ∉ set(sublist xs I)
by (auto simp add: set-sublist)

lemma in-set-sublistD: x ∈ set(sublist xs I) ⇒ x ∈ set xs
by (auto simp add: set-sublist)

lemma sublist-singleton [simp]: sublist [x] A = (if A then [x] else [])
by (simp add: sublist-Cons)

lemma distinct-sublistI [simp]: distinct xs ⇒ distinct(sublist xs I)
apply (induct xs arbitrary: I)
apply simp
apply (auto simp add: sublist-Cons)
done

lemma sublist-upt-eq-take [simp]: sublist l {..<n} = take n l
apply (induct l rule: rev-induct, simp)
apply (simp split: nat-diff-split add: sublist-append)
done

lemma filter-in-sublist:
distinct xs \implies \text{filter}(\%, x \in \text{set}(\text{sublist} \ xs \ s)) \ xs = \text{sublist} \ xs \ s
proof (induct xs arbitrary: s)
  case Nil thus \text{thesis} by simp
next
case (Cons a xs)
  then have \exists. x: set \ xs \longrightarrow x \neq a \ by auto
  with Cons show \text{thesis} by (simp add: sublist-Cons cong:filter-cong)
qed

67.1.35 sublists and List.n-lists

lemma length-sublists:
length (sublists \ xs) = 2 \^ \ length \ xs
by (induct xs) (simp-all add: Let-def)

lemma sublists-powset:
set ' set (sublists \ xs) = \text{Pow} (set \ xs)
proof
  have aux: \\( \forall A. set ' Cons x ' A = \text{insert} ' x ' set ' A \)
    by (auto simp add: image-def)
  have set (map set (sublists \ xs)) = \text{Pow} (set \ xs)
    by (induct xs)
      (simp-all add: aux Let-def Pow-insert Un-commute comp-def del: map-map)
  then show \text{thesis} by simp
qed

lemma distinct-set-sublists:
assumes distinct \ xs
shows distinct (map set (sublists \ xs))
proof (rule card-distinct)
  have finite (set \ xs) by rule
  then have card (Pow (set \ xs)) = 2 \^ \ card (set \ xs) by (rule card-Pow)
    with assms distinct-card [of \ xs]
       have card (Pow (set \ xs)) = 2 \^ \ length \ xs by simp
    then show card (set (map set (sublists \ xs))) = length (map set (sublists \ xs))
      by (simp add: sublists-powset length-sublists)
qed

lemma n-lists-Nil [simp]: List.n-lists \ n [] = (if \ n = 0 then [] else [])
by (induct \ n) simp-all

lemma length-n-lists: length (List.n-lists \ n \ xs) = length \ xs \^ \ \ n
by (induct \ n) (auto simp add: length-concat o-def listsum-triv)

lemma length-n-lists-elem: \ ys \in set (List.n-lists \ n \ xs) \implies \ length \ ys = \ n
by (induct n arbitrary: ys) auto

lemma set-n-lists: set (List.n-lists n xs) = {ys. length ys = n ∧ set ys ⊆ set xs}
proof (rule set-eqI)
  fix ys :: 'a list
  show ys ∈ set (List.n-lists n xs) ←→ ys ∈ {ys. length ys = n ∧ set ys ⊆ set xs}
proof
  have ys ∈ set (List.n-lists n xs) ⟹ length ys = n 
    by (induct n arbitrary: ys) auto
  moreover have ∀x. ys ∈ set (List.n-lists n xs) ⟹ x ∈ set ys ⟹ x ∈ set xs 
    by (induct n arbitrary: ys) auto
  moreover have set ys ⊆ set xs ⟹ ys ∈ set (List.n-lists (length ys) xs) 
    by (induct ys) auto
  ultimately show thesis by auto
qed

lemma distinct-n-lists:
  assumes distinct xs
  shows distinct (List.n-lists n xs)
proof (rule card-distinct)
  from assms have card-length: card (set xs) = length xs by (rule distinct-card)
  have card (set (List.n-lists n xs)) = card (set xs) ^ n 
    proof (induct n)
      case 0 then show ?case by simp
    next
    case (Suc n)
    moreover have card (⋃ys∈set (List.n-lists n xs). (λy. y # ys) ' set xs) 
      = (∑ys∈set (List.n-lists n xs). card ((λy. y # ys) ' set xs)) 
      by (rule card-UN-disjoint) auto
    moreover have ∃ys. card ((λy. y # ys) ' set xs) = card (set xs) 
      by (rule card-image) (simp add: inj-on-def)
    ultimately show ?case by auto
    qed
  also have . . . = length xs ^ n by (simp add: card-length)
  finally show card (set (List.n-lists n xs)) = length (List.n-lists n xs) 
    by (simp add: length-n-lists)
  qed

67.1.36 splice

lemma splice-Nil2 [simp, code]: splice xs [] = xs
  by (cases xs) simp-all

declare splice.simps(1,3)[code]
declare splice.simps(2)[simp del]

lemma length-splice[simp]: length(splice xs ys) = length xs + length ys
  by (induct xs ys rule: splice.induct) auto
67.1.37 Transpose

function transpose where

\[
\begin{align*}
\text{transpose} \; [] &= [] \\
\text{transpose} \; ([\_] \# xss) &= \text{transpose} \; xss \\
\text{transpose} \; ((x \# \textit{xs}) \# xss) &= \\
(x \neq [\_]. (h#t) \leftarrow xss) \# \text{transpose} \; (xs \neq [t. (h#t) \leftarrow xss])
\end{align*}
\]

by pat-completeness auto

lemma transpose-aux-filter-head:

\[
\begin{align*}
\text{concat} \; (\text{map} \; (\text{case-list} \; [] \; (\lambda \textit{h t}. [\_])) \; xss) &= \\
\text{map} \; (\lambda \textit{xss} . \textit{ys} \neq [] \; [\_]) \; \text{ys} \leftarrow xss
\end{align*}
\]

by (induct xss) (auto split: list.split)

lemma transpose-aux-filter-tail:

\[
\begin{align*}
\text{concat} \; (\text{map} \; (\text{case-list} \; [] \; (\lambda \textit{h t}. [\_])) \; xss) &= \\
\text{map} \; (\lambda \textit{xss} . \textit{xs} \neq [] \; [\_]) \; \text{ys} \leftarrow xss
\end{align*}
\]

by (induct xss) (auto split: list.split)

lemma transpose-aux-max:

\[
\begin{align*}
\text{max} \; (\text{Suc} \; (\text{length} \; \textit{xs})) \; (\text{foldr} \; (\lambda \textit{xss} . \text{max} \; (\text{length} \; \textit{xs})) \; \textit{xs} \; 0) &= \\
\text{Suc} \; (\text{max} \; (\text{length} \; \textit{xs})) \; (\text{foldr} \; (\lambda \textit{xss} . \text{max} \; (\text{length} \; \textit{xs} \; - \; \text{Suc} \; 0)) \; [\_]) \; \text{ys} \leftarrow xss
\end{align*}
\]

proof: (cases \; \text{ys} \neq [] \; [\_] \; \Rightarrow \; [\_] )

\begin{itemize}
\item \text{case True} \; \\
\text{hence foldr} \; (\lambda \textit{xss} . \text{max} \; (\text{length} \; \textit{xs})) \; \textit{xs} \; 0 = 0
\item \text{proof (induct } \textit{xss}) \; \\
\text{case Cons x xs} \; \\
\text{then have } \textit{x} = [] \; \text{by (cases } \textit{x}) \; \text{auto} \; \\
\text{with Cons show } \text{?case by auto} \; \\
\text{qed simp} \; \\
\text{thus } \text{?thesis using True by simp}
\end{itemize}

next \text{case False}

\begin{itemize}
\item \text{have foldA: } \text{foldA} = \text{foldr} \; (\lambda \textit{xss} . \text{max} \; (\text{length} \; \textit{xs})) \; [\_] \; \text{ys} \neq [] \; 0 - 1
\item \text{by (induct } \textit{xss}) \; \text{auto}
\end{itemize}

\begin{itemize}
\item \text{have foldB: } \text{foldB} = \text{foldr} \; (\lambda \textit{xss} . \text{max} \; (\text{length} \; \textit{xs})) \; [\_] \; \text{ys} \neq [] \; 0
\item \text{by (induct } \textit{xss}) \; \text{auto}
\end{itemize}

\begin{itemize}
\item \text{have } 0 < ?\text{foldB}
\item \text{proof –}
\item \text{from False}
\item \text{obtain } z \; \text{zs} \; \text{where } \text{zs}: \; [\_] \; \text{ys} \neq [] \; \Rightarrow \; z \# \text{zs} \; \text{by (auto simp: neq-Nil-conv)}
\item \text{hence } z \in \; \text{set } ([\_] \; \text{ys} \neq []) \; \text{by auto}
\item \text{hence } z \neq [] \; \text{by auto}
\item \text{thus } \text{?thesis}
\item \text{unfolding foldB zs}
\item \text{by (auto simp: max-def intro: less-le-trans)}
\item \text{qed}
\end{itemize}
thus \( \text{thesis} \)

unfolding foldA foldB max-Suc-Suc\([\text{symmetric}]\)
by simp

qed

termination transpose
by (relation measure \((\lambda xs. \text{foldr} (\lambda xs. \text{max} (\text{length} xs)) xs 0 + \text{length} xs))\)
(auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max less-Suc-eq-le)

lemma transpose-empty: \((\text{transpose} \; xs = []) \iff (\forall x \in \text{set} \; xs. \; x = [])\)
by (induct rule: transpose.induct) simp-all

lemma length-transpose:
fixes \(xs :: 'a\) list list
shows \(\text{length} \; (\text{transpose} \; xs) = \text{foldr} (\lambda xs. \text{max} (\text{length} xs)) \; xs \; 0\)
by (induct rule: transpose.induct)
(auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max max-Suc-Suc\([\text{symmetric}]\) simp del: max-Suc-Suc)

lemma nth-transpose:
fixes \(xs :: 'a\) list list
assumes \(i < \text{length} \; (\text{transpose} \; xs)\)
shows \(\text{transpose} \; xs \; ![i] = \text{map} (\lambda xs. ![i]) \; (\text{filter} (\lambda ys. \; ys \neq []) \; xs)\)
using assms proof (induct arbitrary: \(i\) rule: transpose.induct)
  case \((3 \; x \; xs \; xss)\)
  def \(XS \equiv (x \# xs) \# xss\)
  hence \([\text{simp}]\): \(XS \neq []\) by auto
  thus \(\text{thesis}\) by simp
  next
  case \(0\)
  thus \(\text{thesis}\) by (simp add: transpose-aux-filter-head hd-conv-nth)

next
  case \((\text{Suc} \; j)\)
  have \(*\): \(\forall xs. \; xs \# \text{map} \; \text{tl} \; xss = \text{map} \; \text{tl} \; ((x\#xs)\#xss)\) by simp
  have \(**\): \(\forall xss. \; (x\#xs) \# \text{filter} (\lambda ys. \; ys \neq []) \; xss = \text{filter} (\lambda ys. \; ys \neq []) \; (x\#xs)\#xss\)
  by simp
  \{ fix \(x\) have \(\text{Suc} \; j < \text{length} \; x \iff x \neq [] \land j < \text{length} \; x - \text{Suc} \; 0\)
  by (cases \(x\)) simp-all
  \} note \(**\) = this

  have \(j\)-less: \(j < \text{length} \; (\text{transpose} \; (xs \# \text{concat} \; (\text{map} \; (\text{case-list} \; [] (\lambda h.\; [t])) \; xss)))\)
  using \(3.\)prems by (simp add: transpose-aux-filter-tail length-transpose Suc)

  show \(\text{thesis}\)
      unfolding transpose.sinps \((i = \text{Suc} \; j)\) nth-Cons-Suc \(3.\)hyps \((OF \; j\)-less\)
      apply (auto simp: transpose-aux-filter-tail filter-map comp-def length-transpose
* ** *** XS-def [symmetric])

apply (rule list.exhaust)
by auto
qed

lemma transpose-map-map:
transpose (map (map f) xs) = map (map f) (transpose xs)
proof (rule nth-equalityI, safe)
  have [simp]: length (transpose (map (map f) xs)) = length (transpose xs)
    by (simp add: length-transpose foldr-map comp-def)
  show length (transpose (map (map f) xs)) = length (map (map f) (transpose xs))
    by simp
next
  fix i assume i < length (transpose (map (map f) xs))
  thus transpose (map (map f) xs) ! i = map (map f) (transpose xs) ! i
    by (simp add: nth-transpose filter-map comp-def)
qed

67.1.38 (In)finiteness

lemma finite-maxlen:
finite (M :: 'a list set) ==> EX n. ALL s : M. size s < n
proof (induct rule: finite.induct)
  case emptyI show ?case by simp
next
  case (insertI M xs)
  then obtain n where [simp]: length xs < n by blast
  hence ALL s : insert xs M. size s < max n (size xs) + 1 by auto
  thus ?case ..
qed

lemma lists-length-Suc-eq:
{x. set xs ⊆ A ∧ length xs = Suc n} =
(λ(xs, n). n#xs) ` {x. set xs ⊆ A ∧ length xs = n} × A
by (auto simp: length-Suc-conv)

lemma assumes finite A
shows finite-lists-length-eq: finite {xs. set xs ⊆ A ∧ length xs = n}
and card-lists-length-eq: card {xs. set xs ⊆ A ∧ length xs = n} = (card A) ^ n
using (finite A)
by (induct n)
  (auto simp: card-image inj-split-Cons lists-length-Suc-eq cong: conj-cong)

lemma finite-lists-length-le:
assumes finite A shows finite {xs. set xs ⊆ A ∧ length xs ≤ n}
(is finite ?S)
proof
  have ?S = (⋃ n ∈ {0..n}. {xs. set xs ⊆ A ∧ length xs = n}) by auto
thus thesis by (auto intro!: finite-lists-length-eq[OF finite A] simp only:)
qed

lemma card-lists-length-le:
assumes finite A shows card {xs. set xs ⊆ A ∧ length xs ≤ n} = (∑ i≤n. card A ∧ i)
proof 
  have (∑ i≤n. card A ∧ i) = card (∪ i≤n. {xs. set xs ⊆ A ∧ length xs = i})
  using ⟨finite A⟩ by (auto simp add: card-UN-disjoint)
  also have (∪ i≤n. {xs. set xs ⊆ A ∧ length xs = i}) = {xs. set xs ⊆ A ∧ length xs ≤ n}
  by auto
  finally show thesis by simp
qed

lemma card-lists-distinct-length-eq:
assumes k < card A shows card {xs. length xs = k ∧ distinct xs ∧ set xs ⊆ A} = ∏ {card A − k + 1 .. card A}
proof (induct k)
case 0
then have {xs. length xs = 0 ∧ distinct xs ∧ set xs ⊆ A} = {} by auto
then show case by simp
next
case (Suc k)
let ?k-list = λk xs. length xs = k ∧ distinct xs ∧ set xs ⊆ A
have inj-Cons: ∨ A. inj-on (λ(xs, n). n # xs) A by (rule inj-onI) auto
from Suc have k < card A by simp
moreover have finite A using assms by (simp add: card-ge-0-finite)
moreover have finite {xs. ?k-list k xs}
  using finite-lists-length-eq[OF finite A, of k]
  by − (rule finite-subset, auto)
moreover have ∨ i j. i ≠ j −→ {i} × (A − set i) ∩ {j} × (A − set j) = {}
  by auto
moreover have ∨ i. i ∈ Collect (?k-list k) −→ card (A − set i) = card A − k
  by (simp add: card-Diff-subset distinct-card)
moreover have {xs. ?k-list (Suc k) xs} =
  (λ(xs, n). n # xs) · ∪((λxs. {xs} × (A − set xs)) · {xs. ?k-list k xs})
  by (auto simp: length-Suc-conv)
moreover have Suc (card A − Suc k) = card A − k using Suc.prems by simp
then have (card A − k) · ∏ {Suc (card A − k) .. card A} = ∏ {Suc (card A − Suc k) .. card A}
  by (subst setprod.insert[symmetric]) (simp add: atLeastAtMost-insertL)+
ultimately show ?case
by (simp add: card-image inj-Cons card-UN-disjoint Suc.hyps algebra-simps) qed

lemma infinite-UNIV-listI: \sim finite(UNIV::'a list set)
apply (rule notI)
apply (drule finite-maxlen)
apply clarsimp
apply (erule-tac x = replicate n undefined in allE)
by simp

67.2 Sorting
Currently it is not shown that sort returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context linorder
begin

lemma set-insort-key:
set (insort-key f x xs) = insert x (set xs)
by (induct xs) auto

lemma length-insort [simp]:
length (insort-key f x xs) = Suc (length xs)
by (induct xs) simp-all

lemma insort-key-left-comm:
assumes f x \neq f y
shows insort-key f y (insort-key f x xs) = insort-key f x (insort-key f y xs)
by (induct xs) (auto simp add: assms dest: antisym)

lemma insort-left-comm:
insort x (insort y xs) = insort y (insort x xs)
by (cases x = y) (auto intro: insort-key-left-comm)

lemma comp-fun-commute-insort:
comp-fun-commute insort
proof
qed (simp add: insort-left-comm fun-eq-iff)

lemma sort-key-simps [simp]:
sort-key f [] = []
sort-key f (x\#xs) = insort-key f x (sort-key f xs)
by (simp-all add: sort-key-def)

lemma (in linorder) sort-key-conv-fold:
assumes inj-on f (set xs)
shows sort-key f xs = fold (insort-key f) xs []
proof –
  have fold (insort-key f) (rev xs) = fold (insort-key f) xs
proof (rule fold-rev, rule ext)
    fix zs
    fix x y
  assume x ∈ set xs y ∈ set xs
  with assms have *: f y = f x ⟹ y = x by (auto dest: inj-onD)
  have **: x = y ⟷ y = x by auto
  show (insort-key f y ◦ insort-key f x) zs = (insort-key f x ◦ insort-key f y) zs
    by (induct zs) (auto intro: * simp add: **)  
qed
  then show ?thesis by (simp add: sort-key-def foldr-conv-fold)
qed

lemma (in linorder) sort-conv-fold:
  sort xs = fold insort xs []
by (rule sort-key-conv-fold simp)

lemma length-sort[simp]: length (sort-key f xs) = length xs
by (induct xs, auto)

lemma sorted-Cons: sorted (x#xs) = (sorted xs & (ALL y: set xs. x <= y))
apply(induct xs arbitrary: x) apply simp
by simp (blast intro: order-trans simp)

lemma sorted-tl:
  sorted xs ⟹ sorted (tl xs)
by (cases xs) (simp-all add: sorted-Cons)

lemma sorted-append:
  sorted (xs@ys) = (sorted xs & sorted ys & (ALL x: set xs. ALL y: set ys. x <= y))
by (induct xs) (auto simp add:sorted-Cons)

lemma sorted-nth-mono:
  sorted xs ⟹ i ≤ j ⟹ j < length xs ⟹ xs!i ≤ xs!j
by (induct xs arbitrary: i j) (auto simp:nth-Cons’ sorted-Cons)

lemma sorted-rev-nth-mono:
  sorted (rev xs) ⟹ i ≤ j ⟹ j < length xs ⟹ xs!j ≤ xs!i
using sorted-nth-mono[of rev xs length xs - j - 1 length xs - i - 1]
  rev-nth[of length xs - i - 1 xs] rev-nth[of length xs - j - 1 xs]
by auto

lemma sorted-nth-monoI:
  (∀ i. j ≤ i & j < length xs ⟹ xs!i ≤ xs!j) ⟹ sorted xs
proof (induct xs)
  case (Cons x zs)
have sorted xs
proof (rule Cons.hyps)
  fix i j assume i ≤ j and j < length xs
  with Cons.prems[of Suc i Suc j]
  show xs ! i ≤ xs ! j by auto
qed
moreover
{  
  fix y assume y ∈ set xs
  then obtain j where j < length xs and xs ! j = y
    unfolding in-set-cone-nth by blast
  with Cons.prems[of 0 Suc j]
  have x ≤ y
    by auto
}
ultimately
show ?case
  unfolding sorted-Cons by auto
qed simp

lemma sorted-equals-nth-mono:
  sorted xs = (∀ j < length xs. ∀ i ≤ j. xs ! i ≤ xs ! j)
by (auto intro: sorted-nth-mono1 sorted-nth-mono)

lemma set-insort: set(insort-key f x xs) = insert x (set xs)
by (induct xs) auto

lemma set-sort[simp]: set(sort-key f xs) = set xs
by (induct xs) (simp-all add: set-insort)

lemma distinct-insort: distinct (insort-key f x xs) = (x ∉ set xs ∧ distinct xs)
by (induct xs) (auto simp: set-insort)

lemma distinct-sort[simp]: distinct (sort-key f xs) = distinct xs
by (induct xs) (simp-all add: distinct-insort)

lemma sorted-insort-key: sorted (map f (insort-key f x xs)) = sorted (map f xs)
by (induct xs) (auto simp: sorted-Cons set-insort)

lemma sorted-insort: sorted (insort x xs) = sorted xs
using sorted-insort-key [where f=λx. x] by simp

theorem sorted-sort-key [simp]: sorted (map f (sort-key f xs))
by (induct xs) (auto simp: sorted-insort-key)

theorem sorted-sort [simp]: sorted (sort xs)
using sorted-sort-key [where f=λx. x] by simp

lemma sorted-butlast:
assumes $xs \neq []$ and sorted $xs$
shows sorted (butlast $xs$)
proof –
from $xs \neq []$ obtain $ys \ y$ where $xs = ys \ @ [y]$ by (cases $xs$ rule: rev-cases) auto
with (sorted $xs$) show ?thesis by (simp add: sorted-append)
qed

lemma insort-not-Nil [simp]:
insort-key $f \ a \ xs \neq []$
by (induct $xs$) simp-all

lemma insort-is-Cons: $\forall x \in \text{set} \ \ xs. \ f \ a \leq f \ x \Longrightarrow \ \text{insort-key} \ f \ a \ xs = a \ # \ xs$
by (cases $xs$) auto

lemma sorted-sort-id: sorted $xs \Longrightarrow \ \text{sort} \ \ xs = xs$
by (induct $xs$) (auto simp add: sorted-Cons insort-is-Cons)

lemma sorted-map-remove1:
\text{sorted} (\text{map} \ f \ xs) \Longrightarrow \ \text{sorted} (\text{map} \ (\text{remove1} \ x \ xs))
by (induct $xs$) (auto simp add: sorted-Cons)

lemma sorted-remove1: sorted $xs \Longrightarrow \ \text{sorted} \ (\text{remove1} \ a \ xs)$
using sorted-map-remove1 [of $\lambda x. \ x$] by simp

lemma insort-key-remove1:
assumes $a \in \text{set} \ \ xs$ and sorted $(\text{map} \ f \ xs)$ and $\text{hd} \ (\text{filter} \ (\lambda x. \ f \ a = f \ x) \ xs) = a$
shows insort-key $f \ a \ (\text{remove1} \ a \ xs) = xs$
using assms proof (induct $xs$)
case (Cons $x \ xs$)
then show ?case
proof (cases $x = a$)
case False
then have $f \ x \neq f \ a$ using Cons.prems by auto
then have $f \ x < f \ a$ using Cons.prems by (auto simp: sorted-Cons)
with $(f \ x \neq f \ a)$ show ?thesis using Cons by (auto simp: sorted-Cons)
qedauto simp: sorted-Cons insort-is-Cons
qedsimp

lemma insort-remove1:
assumes $a \in \text{set} \ \ xs$ and sorted $xs$
shows $\text{insort} \ a \ (\text{remove1} \ a \ xs) = xs$
proof (rule insort-key-remove1)
from $a \in \text{set} \ \ xs$ show $a \in \text{set} \ \ xs$.
from (sorted $xs$) show sorted $(\text{map} \ (\lambda \ x. \ x) \ xs)$ by simp
from $(a \in \text{set} \ \ xs)$ have $a \in \text{set} \ (\text{filter} \ (op = a) \ xs)$ by auto
then have set $(\text{filter} \ (op = a) \ xs) \neq {}$ by auto
then have \( \text{filter} \ (op = a) \ xs \neq [] \) by (auto simp only: set-empty)
then have \( \text{length} \ (\text{filter} \ (op = a) \ xs) > 0 \) by simp
then obtain \( n \) where \( \text{Suc} \ n = \text{length} \ (\text{filter} \ (op = a) \ xs) \)
  by (cases \( \text{length} \ (\text{filter} \ (op = a) \ xs) \)) simp-all
moreover have \( \text{replicate} \ (\text{Suc} \ n) \ a = \# \text{replicate} \ n \ a \)
  by simp
ultimately show \( \text{hd} \ (\text{filter} \ (op = a) \ xs) = a \) by (simp add: replicate-length-filter)
qed

lemma \( \text{sorted-remdups[simp]} \):
  \( \text{sorted} \ l \Rightarrow \text{sorted} \ (\text{remdups} \ l) \)
by (induct l) (auto simp: sorted-Cons)

lemma \( \text{sorted-remdups-adj[simp]} \):
  \( \text{sorted} \ xs \Rightarrow \text{sorted} \ (\text{remdups-adj} \ xs) \)
by (induct xs rule: remdups-adj.inuct, simp-all split: split-if-asm add: sorted-Cons)

lemma \( \text{sorted-distinct-set-unique} \):
assumes \( \text{sorted} \ xs \ \text{distinct} \ xs \ \text{sorted} \ ys \ \text{distinct} \ ys \ \text{set} \ xs = \text{set} \ ys \)
sows \( xs = ys \)
proof
  from assms have \( 1: \text{length} \ xs = \text{length} \ ys \) by (auto dest!: distinct-card)
  from assms show \( \text{thesis} \)
  proof (induct rule: list-induct2[OF \( 1 \)])
    case 1 show \( ?\text{case} \) by simp
  next
    case 2 thus \( ?\text{case} \) by (simp add: sorted-Cons)
    (metis Diff-insert-absorb antisym insertE insert-iff)
  qed
qed

lemma \( \text{map-sorted-distinct-set-unique} \):
assumes \( \text{inj-on} \ f \ (\text{set} \ xs \cup \text{set} \ ys) \)
assumes \( \text{sorted} \ (\text{map} \ f \ xs) \ \text{distinct} \ (\text{map} \ f \ xs) \)
  \( \text{sorted} \ (\text{map} \ f \ ys) \ \text{distinct} \ (\text{map} \ f \ ys) \)
assumes \( \text{set} \ xs = \text{set} \ ys \)
sows \( xs = ys \)
proof
  from assms have \( \text{map} \ f \ xs = \text{map} \ f \ ys \)
    by (simp add: sorted-distinct-set-unique)
  with (inj-on f (set xs \cup set ys)) show \( xs = ys \)
    by (blast intro: map-inj-on)
qed

lemma \( \text{finite-sorted-distinct-unique} \):
shows \( \text{finite} \ A \Rightarrow \exists! \ xs. \text{set} \ xs = A \ \& \ \text{sorted} \ xs \ \& \ \text{distinct} \ xs \)
apply (drule finite-distinct-list)
apply clarify
apply (rule_tac a=sort xs in ex1I)
apply (auto simp: sorted-distinct-set-unique)
done

lemma
assumes sorted xs
shows sorted-take: sorted (take n xs) and sorted-drop: sorted (drop n xs)
proof –
  from assms have sorted (take n xs @ drop n xs) by simp
  then show sorted (take n xs) and sorted (drop n xs)
    unfolding sorted-append by simp-all
qed

lemma sorted-dropWhile: sorted xs ⇒ sorted (dropWhile P xs)
  by (auto dest: sorted-drop simp add: dropWhile-eq-drop)

lemma sorted-takeWhile: sorted xs ⇒ sorted (takeWhile P xs)
  by (subst takeWhile-eq-take) (auto dest: sorted-take)

lemma sorted-filter:
  sorted (map f xs) =⇒ sorted (map f (filter P xs))
  by (induct xs) (simp-all add: sorted-Cons)

lemma foldr-max-sorted:
  assumes sorted (rev xs)
  shows foldr max xs y = (if xs = [] then y else max (xs ! 0) y)
  using assms
  proof (induct xs)
    case (Cons x xs)
    then have sorted (rev xs) using sorted-append by auto
    with Cons show ?case
      by (cases xs) (auto simp add: sorted-append max-def)
  qed simp

lemma filter-equals-takeWhile-sorted-rev:
  assumes sorted: sorted (rev (map f xs))
  shows [x ← xs. t < f x] = takeWhile (λ x. t < f x) xs
    (is filter ?P xs = ?tW)
  proof (rule takeWhile-eq-filter)[symmetric]
    let ?dW = dropWhile ?P xs
    fix x assume x ∈ set ?dW
    then obtain i where i: i < length ?dW and nth-i: x = ?dW ! i
    unfolding in-set-conv-nth by auto
    hence length ?tW + i < length (?tW @ ?dW)
      unfolding length-append by simp
    hence i': length (map f ?tW) + i < length (map f xs) by simp
    have (map f ?tW @ map f ?dW) ! (length (map f ?tW) + i) ≤
      (map f ?tW @ map f ?dW) ! (length (map f ?tW) + 0)
      using sorted-rev-nth-mono[OF sorted - i', of length ?tW]
unfolding map-append[symmetric] by simp  

hence \( f x \leq f \ (\ ?dW \ ! 0) \)

unfolding nth-append-length-plus nth-i  
using i preorder-class.le-less-trans[OF le0 i] by simp

also have \( \ldots \leq t \)
using hd-dropWhile[of \( ?P \) xs] le0[THEN preorder-class.le-less-trans, OF i]
using hd-cone-nth[of \( ?dW \)] by simp

finally show \( \neg t < f \ x \) by simp

qed

lemma insort-insert-key-triv:
\( f x \in f ^{'} \ \text{set} \ \text{xs} \Longrightarrow \text{insort-insert-key} \\ f \ x \ \text{xs} = \text{xs} \)
by (simp add: insort-insert-key-def)

lemma insort-insert-triv:
\( x \in \text{set} \ \text{xs} \Longrightarrow \text{insort-insert} \ \text{xs} = \text{xs} \)
using insort-insert-key-triv[of \( \lambda x. \ x \)] by simp

lemma insort-insert-insort-key:
\( f x \not\in f ^{'} \ \text{set} \ \text{xs} \Longrightarrow \text{insort-insert-key} \\ f \ x \ \text{xs} = \text{insort-key} \\ f \ x \ \text{xs} \)
by (simp add: insort-insert-key-def)

lemma insort-insert-insort:
\( x \not\in \text{set} \ \text{xs} \Longrightarrow \text{insort-insert} \ \text{xs} = \text{insort} \ x \ \text{xs} \)
using insort-insert-insort-key[of \( \lambda x. \ x \)] by simp

lemma set-insort-insert:
\( \text{set} \ (\text{insort-insert} \ x \ \text{xs}) = \text{insert} \ x \ (\text{set} \ \text{xs}) \)
by (auto simp add: insort-insert-key-def set-insort)

lemma distinct-insort-insert:
assumes \( \text{distinct} \ \text{xs} \)
shows \( \text{distinct} \ (\text{insort-insert-key} \ f \ x \ \text{xs}) \)
using assms by (induct xs) (auto simp add: insort-insert-key-def set-insort)

lemma sorted-insort-insert-key:
assumes \( \text{sorted} \ (\text{map} \ f \ \text{xs}) \)
shows \( \text{sorted} \ (\text{map} \ f \ (\text{insort-insert-key} \ f \ x \ \text{xs})) \)
using assms by (simp add: insort-insert-key-def sorted-insort-key)

lemma sorted-insort-insert:
assumes \( \text{sorted} \ \text{xs} \)
shows \( \text{sorted} \ (\text{insort-insert} \ x \ \text{xs}) \)
using assms sorted-insort-insert-key[of \( \lambda x. \ x \)] by simp

lemma filter-insort-triv:
\( \neg \ P \ x \Longrightarrow \text{filter} \ P \ (\text{insort-key} \ f \ x \ \text{xs}) = \text{filter} \ P \ \text{xs} \)
by (induct xs) simp-all
lemma filter-insort:
  \( \text{sorted} (\text{map } f \ x) \implies P x \implies \text{filter } P (\text{insort-key } f \ x) = \text{insort-key } f \ x (\text{filter } P \ x) \)
using assms by (induct xs)
(auto simp add: sorted-Cons, subst insort-is-Cons, auto)

lemma filter-sort:
  \( \text{filter } P (\text{sort-key } f \ x) = \text{sort-key } f (\text{filter } P \ x) \)
by (induct xs) (simp-all add: filter-insort-triv filter-insort)

lemma sorted-map-same:
  \( \text{sorted} (\lambda x \to (\text{map } f \ x) = g \ x) ) \)
proof (induct xs arbitrary: g)
case Nil then show ?case by simp
next
case (Cons x xs)
then have \( \text{sorted} (\lambda y \to (\text{map } f \ y = (\lambda x. f x) \ xs) ) \)
moreover from Cons have \( \text{sorted} (\lambda y \to (\text{map } f \ y = (g \circ \text{Cons } x) \ xs) ) \)
ultimately show ?case by (simp-all add: sorted-Cons)
qed

lemma sorted-same:
  \( \text{sorted} [x \to (\text{map } f \ x = g \ x) ] \)
using sorted-map-same[of \( \lambda x. x \)] by simp

lemma remove1-insort [simp]:
  \( \text{remove1 } x (\text{insort } x \ xs) = \xs \)
by (induct xs) simp-all

end

lemma sorted-upt [simp]: \( \text{sorted}[i..<j] \)
by (induct j) (simp-all add: sorted-append)

lemma sorted-upto [simp]: \( \text{sorted}[i..j] \)
apply (induct i j rule: upto.induct)
apply (subst upto.simps)
apply (simp add: sorted-Cons)
done

lemma sorted-find-Min:
assumes \( \text{sorted } \xs \)
assumes \( \exists x \in \text{set } \xs. \ P x \)
shows \( \text{List.find } P \ xs = \text{Some } (\text{Min } (x \in \text{set } \xs. \ P x )) \)
using assms proof (induct \( \xs \) rule: sorted.induct)
case Nil then show ?case by simp
next
case (Cons \( \xs \) \( x \)) show ?case proof (cases \( P x \))
case True with \( \text{Cons } \) show ?thesis by (auto intro: Min-eqI [symmetric])
next
  case False then have \{ y. (y = x \lor y \in \text{set} \, xs) \land P \, y \} = \{ y \in \text{set} \, xs. P \, y \}
    by auto
  with Cons False show \$\text{thesis}\$ by simp-all
qed

67.2.1 transpose on sorted lists

lemma sorted-transpose[simp]:
  shows sorted (rev (map length (transpose xs)))
  by (auto simp: sorted-equals-nth-mono rev-nth nth-transpose
      length-filter-conv-card intro: card-mono)

lemma transpose-max-length:
  foldr (\lambda xs. max (length xs)) (transpose xs) 0 = length \([x \leftarrow xs. x \neq []]\)
  (is \$L = ?R\$)
proof (cases transpose xs = [])
  case False
  have \$L = \text{foldr max (map length (transpose xs)) 0}\$
    by (simp add: foldr-map comp-def)
  also have ... = length (transpose xs ! 0)
    using False sorted-transpose by (simp add: foldr-max-sorted)
  finally show \$\text{thesis}\$
    using False by (simp add: nth-transpose)
next
  case True
  hence \([x \leftarrow xs. x \neq []]\) = []
    by (auto intro!: filter-False simp: transpose-empty)
  thus \$\text{thesis}\$ by (simp add: transpose-empty True)
qed

lemma length-transpose-sorted:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
  case False
  thus \$\text{thesis}\$
    using foldr-max-sorted[OF sorted] False
    unfolding length-transpose foldr-map comp-def
    by simp
qed simp

lemma nth-nth-transpose-sorted[simp]:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  and i: i < length (transpose xs)
  and j: j < length \([ys \leftarrow xs. i < length ys]\)
shows transpose xs \& j = xs \& i
using \( j \) filter-equals-takeWhile-sorted-rev[\( \text{OF sorted, of } i \)]
\( \text{nths-transpose[}\text{OF } i\text{]} \) \& \( \text{nths-map[}\text{OF } j\text{]} \)
by (simp add: takeWhile-nth)

lemma \( \text{transpose-column-length} \):
\begin{align*}
\text{fixes } & \text{xs :: } 'a \text{ list list} \\
\text{assumes } & \text{sorted: } \text{sorted } (\text{rev } (\text{map length } \text{xs})) \text{ and } i < \text{length } \text{xs} \\
\text{shows } & \text{length } (\text{filter } (\lambda \text{ys} . i < \text{length } \text{ys}) \text{ (transpose } \text{xs})) = \text{length } (\text{xs }\& i)
\end{align*}

proof –
\begin{align*}
\text{have } & \text{xs }\neq [] \text{ using } (i < \text{length } \text{xs}) \text{ by auto} \\
\text{note } & \text{filter-equals-takeWhile-sorted-rev[}\text{OF sorted, simp]} \\
\{ & \text{fix } j \text{ assume } j \leq i \\
& \text{note sorted-rev-nth-mono[}\text{OF sorted, of } j \text{ i, simplified, OF this } (i < \text{length } \text{xs})]\} \\
\text{note } & \text{sortedE } = \text{this[consumes 1]}
\end{align*}

\begin{align*}
\text{have } & \{j \& j < \text{length } (\text{transpose } \text{xs}) \land i < \text{length } (\text{transpose } \text{xs }\& j)\} \\
& = \{ \_. < \text{length } (\text{xs }\& i)\}
\end{align*}

proof safe

fix \( j \)
assume \( j < \text{length } (\text{transpose } \text{xs}) \text{ and } i < \text{length } (\text{transpose } \text{xs }\& j) \)
with this(2) \( \text{nths-transpose[}\text{OF this}(1)\] \)

\begin{align*}
\text{have } & i < \text{length } (\text{takeWhile } (\lambda \text{ys} . j < \text{length } \text{ys}) \text{ xs}) \text{ by simp} \\
\text{from } & \text{nths-mem[}\text{OF this} \text{ takeWhile-nth[}\text{OF this}]\} \\
\text{show } & j < \text{length } (\text{xs }\& i) \text{ by } (\text{auto dest: set-takeWhileD})
\end{align*}

next

fix \( j \)
assume \( j < \text{length } (\text{xs }\& i) \)

\begin{align*}
\text{thus } & j < \text{length } (\text{transpose } \text{xs}) \\
& \text{using foldr-max-sorted[}\text{OF sorted]} \text{ \( \text{xs }\neq []; \text{ sortedE[}\text{OF le0}\]} \\
& \text{by } (\text{auto simp: length-transpose comp-def foldr-map})
\end{align*}

\begin{align*}
\text{have } & \text{Suc } i \leq \text{length } (\text{takeWhile } (\lambda \text{ys} . j < \text{length } \text{ys}) \text{ xs}) \\
& \text{using } (i < \text{length } \text{xs}) \land j < \text{length } (\text{transpose } \text{xs}) \text{ by simp} \\
& \text{with nth-transpose[}\text{OF } (j < \text{length } (\text{transpose } \text{xs})\}] \\
\text{show } & i < \text{length } (\text{transpose } \text{xs }\& j) \text{ by simp}
\end{align*}

qed

thus \( ?\text{thesis} \) by (simp add: length-filter-cone-card)

qed

lemma \( \text{transpose-column} \):
\begin{align*}
\text{fixes } & \text{xs :: } 'a \text{ list list} \\
\text{assumes } & \text{sorted: } \text{sorted } (\text{rev } (\text{map length } \text{xs})) \text{ and } i < \text{length } \text{xs} \\
\text{shows } & \text{map } (\lambda \text{ys} . i < \text{length } \text{ys}) \text{ (filter } (\lambda \text{ys} . i < \text{length } \text{ys}) \text{ (transpose } \text{xs})) \\
& = \text{xs }\& i \text{ (is } ?\text{R }= \).)
\end{align*}

proof (rule nth-equalityI, safe)

\begin{align*}
\text{show } & \text{length } (\text{transpose } \text{xs }\& i) \\
& \text{using transpose-column-length[}\text{OF } \text{assms}] \text{ by simp}
\end{align*}
THEORY “List”

fix j assume j : j < length ?R
note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]
from j have j-less: j < length (xs ! i) using length by simp
have i-less-tW: Suc i ≤ length (takeWhile (λys. Suc j ≤ length ys) xs)
proof (rule length-takeWhile-less-P-nth)
  show Suc i ≤ length xs using (i < length xs) by simp
  fix k assume k < Suc i
  hence k ≤ i by auto
  with sorted-rev-nth-mono[OF sorted this] (i < length xs)
  have length (xs ! i) ≤ length (xs ! k) by simp
  thus Suc j ≤ length (xs ! k) using j-less by simp
qed

have i-less-filter: i < length [ys←xs . j < length ys]
  unfolding filter-equals-takeWhile-sorted-rev[OF sorted-transpose, of i]
  by (simp add: Suc-le-eq)
proof
  have len: length ?L = length ?R
    unfolding length-transpose transpose-max-length
    using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]
    by simp
    { fix i assume i < length ?R
      with less-le-trans[OF - length-takeWhile-le[of - xs]]
      have i < length xs by simp
    } note * = this
  show thesis
    by (rule nth-equalityI)
    (simp-all add: len nth-transpose transpose-column[OF sorted * i-less-filter])
qed

lemma transpose-transpose:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows transpose (transpose xs) = takeWhile (λx. x ≠ []) xs (is ?L = ?R)
proof –
  have sorted: sorted (rev (map length xs))
    by (auto simp: rev-nth rect intro!: sorted-nth-monoI)
  from foldr-max-sorted[OF this] assms
  show thesis
    by (rule nth-equalityI)
    (simp-all add: len nth-transpose transpose-column[OF sorted] * takeWhile-nth)
qed

theorem transpose-rectangle:
  assumes xs = [] ⇒ n = 0
  assumes rect: i. i < length xs ⇒ length (xs ! i) = n
  shows transpose xs = map (λ i. map (λ j. xs ! j ! i) [0..<length xs]) [0..<n]
    (is ?trans = ?map)
proof (rule nth-equalityI)
  have sorted: rev-nth rect intro!: sorted-nth-monoI
  from foldr-max-sorted[OF this] assms
  show thesis
    by (auto simp: rev-nth rect intro!: sorted-nth-monoI)
by (simp-all add: length-transpose foldr-map comp-def)
moreover
{ fix i assume i < n hence \[ys \leftarrow xs . \ i < \text{length } ys\] = xs
  using rect by (auto simp: in-set-conv-nth intro!: filter-True) }
ultimately show \(\forall \ i < \text{length } \ ?\text{trans} . \ ?\text{trans} ! \ i = \ ?\text{map} ! \ i\)
  by (auto simp: nth-transpose intro: nth-equalityI)
qed

67.2.2 sorted-list-of-set

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via set).

67.2.3 sorted-list-of-set

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via set).

context linorder begin

definition sorted-list-of-set :: \('a set \Rightarrow \ 'a list\ where
  \text{sorted-list-of-set} = \text{folding.F insort } []

sublocale sorted-list-of-set!: \text{folding insort Nil where}
  \text{folding.F insort } [] = \text{sorted-list-of-set}
proof -
  interpret \text{comp-fun-commute insort} by (fact comp-fun-commute-insort)
  show \text{folding insort} \text{by default (fact comp-fun-commute)}
  show \text{folding.F insort } [] = \text{sorted-list-of-set} by (simp only: \text{sorted-list-of-set-def})
qed

lemma sorted-list-of-set-empty: 
  \text{sorted-list-of-set} \{\} = []
  by (fact \text{sorted-list-of-set.empty})

lemma sorted-list-of-set-insert [simp]:
  \text{finite } A \implies \text{sorted-list-of-set} (\text{insert } x A) = \text{insort} x (\text{sorted-list-of-set} (A - \{x\}))
  by (fact \text{sorted-list-of-set.insert-remove})

lemma sorted-list-of-set-eq-Nil-iff [simp]:
  \text{finite } A \implies \text{sorted-list-of-set} A = [] \iff A = {} 
  by (auto simp: \text{sorted-list-of-set.remove})

lemma sorted-list-of-set [simp]:
finite $A \implies \text{set} \ (\text{sorted-list-of-set } A) = A \land \text{sorted} \ (\text{sorted-list-of-set } A) \land \text{distinct} \ (\text{sorted-list-of-set } A) \ 
\text{by (induct } A \text{ rule: finite-induct) (simp-all add: set-insort sorted-insort distinct-insort)}$

**lemma** distinct-sorted-list-of-set:

**using** sorted-list-of-set **by** (cases finite $A$) auto

**lemma** sorted-list-of-set-sort-remdups [code]:

sorted-list-of-set (set $xs$) = sort (remdups $xs$)

**proof** —

interpret comp-fun-commute insort **by** (fact comp-fun-commute-insort)

show ?thesis **by** (simp add: sorted-list-of-set eq-fold sort-conv-fold fold-set-fold-remdups)

qed

**lemma** sorted-list-of-set-remove:

assumes finite $A$

shows sorted-list-of-set ($A - \{x\}$) = remove1 $x$ (sorted-list-of-set $A$)

**proof** —

case False with assms have $x \notin \text{set} \ (\text{sorted-list-of-set } A)$ by simp

with False show ?thesis **by** (simp add: remove1-idem)

next
case True then obtain $B$ where $A = \text{insert } x B$ by (rule Set.set-insert)

with assms show ?thesis **by** simp

qed

end

**lemma** sorted-list-of-set-range [simp]:

sorted-list-of-set $\{m..<n\}$ = $\[m..<n\]$

by (rule sorted-distinct-set-unique) simp-all

67.2.4 lists: the list-forming operator over sets

**inductive-set**

lists :: 'a set => 'a list set

for $A$ :: 'a set

where

Nil [intro!, simp]: [] : lists $A$

| Cons [intro!, simp]: || a: $A$; l: lists $A$] ==> a#l : lists $A$

**inductive-cases** listsE [elim!]: $x#l$ : lists $A$

**inductive-cases** listspE [elim!]: listsp $A$ ($x \neq l$)

**inductive-simps** listsp-simps[code]:

listsp $A$ []

listsp $A$ ($x \neq xs$)

**lemma** listsp-mono [mono]: $A \leq B$ ==> listsp $A \leq listsp B
THEORY "List"

by (rule predicateI, erule listsp.induct, blast+)

lemmas lists-mono = listsp-mono [to-set]

lemma listsp-infI:
  assumes l: listsp A l shows listsp B l ==> listsp (inf A B) l using l
by induct blast+

lemmas lists-IntI = listsp-infI [to-set]

lemma listsp-inf-eq [simp]: listsp (inf A B) = inf (listsp A) (listsp B)
proof (rule mono-inf [where f=listsp, THEN order-antisym])
  show mono listsp by (simp add: mono-def listsp-mono)
  show inf (listsp A) (listsp B) <= listsp (inf A B) by (blast intro!: listsp-infI)
qed

lemmas listsp-conj-eq [simp] = listsp-inf-eq [simplified inf-fun-def inf-bool-def]

lemmas lists-Int-eq [simp] = listsp-inf-eq [to-set]

lemma Cons-in-lists-iff [simp]: x#xs : lists A <-> x:A ∧ xs : lists A
by auto

lemma append-in-listsp-conv [iff]:
  (listsp A (xs @ ys)) = (listsp A xs ∧ listsp A ys)
by (induct xs) auto


lemma in-listsp-conv-set: (listsp A xs) = (∀ x ∈ set xs. A x)
  — eliminate listsp in favour of set
by (induct xs) auto


lemma in-listspD [dest!]: listsp A xs ==> ∀ x ∈ set xs. A x
by (rule in-listsp-conv-set [THEN iffD1])

lemmas in-listsD [dest!] = in-listspD [to-set]

lemma in-listspI [intro!]: ∀ x ∈ set xs. A x ==> listsp A xs
by (rule in-listsp-conv-set [THEN iffD2])

lemmas in-listsI [intro!] = in-listspI [to-set]

lemma lists-eq-set: lists A = {xs. set xs <= A}
by auto

lemma lists-empty [simp]: lists {} = {{}}
by auto

lemma lists-UNIV [simp]: lists UNIV = UNIV
by auto

lemma lists-image: lists (f:A) = map f ' lists A
proof -
{ fix xs have \( \forall x \in \text{set } xs. x \in f \cdot A \implies xs \in \text{map } f \cdot \text{lists } A \)
  by (induct xs) (auto simp del: list.map simp add: list.map[symmetric] intro!: imageI) }
then show ?thesis by auto
qed

67.2.5 Inductive definition for membership

inductive ListMem :: '
'a ⇒ '
'a list ⇒ bool
where
elem: ListMem x (x # xs)
| insert: ListMem x xs \implies ListMem x (y # xs)

lemma ListMem-iff: (ListMem x xs) = (x \in \text{set } xs)
apply (rule iffI)
apply (induct set: ListMem)
apply auto
apply (induct xs)
apply (auto intro: ListMem.intros)
done

67.2.6 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from A and tail drawn from Xs.

definition set-Cons :: 'a set ⇒ 'a list set ⇒ 'a list set where
set-Cons A XS = {z. \exists x xs. z = x # xs \land x \in A \land xs \in XS}

lemma set-Cons-sing-Nil [simp]: set-Cons A {} = (%x. [x])'A
by (auto simp add: set-Cons-def)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec listset :: 'a set list ⇒ 'a list set where
listset [] = {[]}
listset (A # As) = set-Cons A (listset As)
67.3 Relations on Lists

67.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These orderings are not used in dictionaries.

```
primrec — The lexicographic ordering for lists of the specified length
lexn :: ('a × 'a) set ⇒ nat ⇒ ('a list × 'a list) set where
lexn r 0 = {}
lexn r (Suc n) =

(map-prod (%(x, xs), x#xs) (%(x, xs). x#xs) '(r <*lex*> lexn r n)) Int

{(xs, ys). length xs = Suc n ∧ length ys = Suc n}
```

```
definition lex :: ('a × 'a) set ⇒ ('a list × 'a list) set where
lex r = (∪ n. lexn r n) — Holds only between lists of the same length
```

```
definition lenlex :: ('a × 'a) set ⇒ ('a list × 'a list) set where
lenlex r = inv-image (less-than <∗lex∗> lex r) (λxs. (length xs, xs))
— Compares lists by their length and then lexicographically
```

```
lemma wf-lexn: wf r =⇒ wf (lexn r n)
apply (induct n, simp, simp)
apply(rule wf-subset)
prefer 2 apply (rule Int-lower1)
apply(rule wf-map-prod-image)
prefer 2 apply (rule inj-onI, auto)
done
```

```
lemma lexn-length:
(xs, ys) : lexn r n =⇒ length xs = n ∧ length ys = n
by (induct n arbitrary: xs ys) auto
```

```
lemma wf-lex [intro!]: wf r =⇒ wf (lex r)
apply (unfold lex-def)
apply (rule wf-UN)
apply (blast intro: wf-lexn, clarify)
apply (rename-tac m n)
apply (subgoal-tac m ≠ n)
prefer 2 apply blast
apply (blast dest: lexn-length not-sym)
done
```

```
lemma lexn-conv:
lexn r n =
{(xs,ys). length xs = n ∧ length ys = n ∧
(∃ yys y yys' ys'. xs = xys @ x#xs' ∧ ys = yys @ y # yys' ∧ (x, y):r)}
apply (induct n, simp)
apply (simp add: image-Collect lex-prod-def, safe, blast)
apply (rule-tac x = ab # yys in exI, simp)
done
```
apply (case-tac xys, simp-all, blast)
done

lemma lex-conv:
lex r = 
\{ (xs,ys). \text{length } xs = \text{length } ys \land \\
(\exists x y xs' ys'. xs = xs @ x \# xs' \land ys = ys @ y \# ys' \land (x, y):r) \} 
by (force simp add: lex-def lexn-conv)

lemma wf-lenlex [intro!]: \text{wf } r == \text{wf } (lenlex } r 
by (unfold lenlex-def) blast

lemma lenlex-conv:
lenlex r = \{ (xs,ys). \text{length } xs < \text{length } ys \mid \\
\text{length } xs = \text{length } ys \land (xs, ys) : r \} 
by (simp add: lenlex-def Id-on-def lex-prod-def inv-image-def)

lemma Nil-notin-lex [iff]: ([], ys) /\notin lex r 
by (simp add: lex-conv)

lemma Nil2-notin-lex [iff]: (xs, []) /\notin lex r 
by (simp add: lex-conv)

lemma Cons-in-lex [simp]:
((a#x, b#y) : lex r) = 
((x, y) : r \land \text{length } xs = \text{length } ys \mid x = y \land (xs, ys) : lex r) 
apply (simp add: lex-conv)
apply (rule iffI)
prefer 2 apply (blast intro: Cons-eq-appendI, clarify)
apply (case-tac xys, simp, simp)
apply blast
done

67.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" \(<"ab" \(<"b". This ordering does not preserve well-foundedness. Author: N. Voelker, March 2005.

definition lexord :: ('a set) set \Rightarrow ('a list set) set where
lexord r = { (x,y). \exists a v. y = x @ a \# v \lor \\
(\exists u a b v w. (a,b) \in r \land x = u @ (a \# v) \land y = u @ (b \# w))} 
lemma lexord-Nil-left[simp]: ([],y) \in lexord r = (\exists a x. y = a \# x) 
by (unfold lexord-def, induct-tac y, auto)

lemma lexord-Nil-right[simp]: (x,[]) \notin lexord r 
by (unfold lexord-def, induct-tac x, auto)

lemma lexord-cons-cons[simp]:
((a \# x, b \# y) \in lexord r) = ((a,b)\in r \mid (a = b \& (x,y) \in lexord r))
apply (unfold lexord-def, safe, simp-all)
apply (case-tac u, simp, simp)
apply (case-tac u, simp, clarsimp, blast, blast, clarsimp)
apply (erule-tac x=b # u in allE)
by force

lemmas lexord-simps = lexord-Nil-left lexord-Nil-right lexord-cons-cons

lemma lexord-append-rightI: \( \exists \, b, z. \, y = b \# z \Rightarrow (x, x @ y) \in \text{lexord} \, r \)
by (induct-tac x, auto)

lemma lexord-append-left-rightI: 
\( (a,b) \in r \Rightarrow (u @ a \# x, u @ b \# y) \in \text{lexord} \, r \)
by (induct-tac u, auto)

lemma lexord-append-leftI: 
\( (u,v) \in \text{lexord} \, r \Rightarrow (x @ u, x @ v) \in \text{lexord} \, r \)
by (induct x, auto)

lemma lexord-append-leftD: 
\[ \{ \, (x @ u, x @ v) \in \text{lexord} \, r; \, (! \, a. \, (a,a) \notin r) \} \Rightarrow (u,v) \in \text{lexord} \, r \]
by (erule rev-mp, induct-tac x, auto)

lemma lexord-take-index-conv:
\( (x,y) : \text{lexord} \, r \) =
\( (\text{length} \, x < \text{length} \, y \land \text{take} \, (\text{length} \, x) \, y = x) \lor \\
(\exists \, i. \, i < \text{min} \, (\text{length} \, x)(\text{length} \, y) \land \text{take} \, i \, x = \text{take} \, i \, y \land (x!i,y!i) \in r) \)
apply (unfold lexord-def Let-def, clarsimp)
apply (rule-tac f = (% a b. a \lor b) in arg-cong2)
apply auto
apply (rule-tac x=hd (drop (length x) y) in exI)
apply (rule-tac x=tl (drop (length x) y) in exI)
apply (erule subst, simp add: min-def)
apply (rule-tac x=\text{length} \, u in exI, simp)
apply (rule-tac x=\text{take} \, i \, x in exI)
apply (rule-tac x=x \# i in exI)
apply (rule-tac x=y \# i in exI, safe)
apply (rule-tac x=\text{drop} \, (\text{Suc} \, i) \, x in exI)
apply (drule sym, simp add: drop-Suc-conv-tl)
apply (rule-tac x=\text{drop} \, (\text{Suc} \, i) \, y in exI)
by (simp add: drop-Suc-conv-tl)

— lexord is extension of partial ordering List.lex

lemma lexord-lex: \( (x,y) \in \text{lex} \, r = ((x,y) \in \text{lexord} \, r \land \text{length} \, x = \text{length} \, y) \)
apply (rule-tac x=y in spec)
apply (induct-tac x, clarsimp)
by (clarify, case-tac x, simp, force)

lemma lexord-irreflexive: \( \forall \, x. \, (x,x) \notin r \Rightarrow (xs,xs) \notin \text{lexord} \, r \)
by (induct xs) auto
By René Thiemann:

**lemma** lexord-partial-trans:
\[
(\forall x y z. x \in \text{set } xs \implies (x,y) \in r \implies (y,z) \in r \implies (x,z) \in r)
\implies (xs,ys) \in \text{lexord } r \implies (ys,zs) \in \text{lexord } r \implies (xs,zs) \in \text{lexord } r
\]

**proof**: (induct xs arbitrary: ys zs)

**case** Nil

from Nil(3) show ?case unfolding lexord-def by (cases zs, auto)

**next**

**case** (Cons x xs yys zzs)

from Cons(3) obtain y ys where yys = y \# ys unfolding lexord-def

by (cases yys, auto)

note Cons = Cons[unfolded yys]

from Cons(4) have one: (x,y) \in r \lor x = y \land (xs,ys) \in \text{lexord } r by auto

from Cons(4) obtain z zs where zzs = z \# zs unfolding lexord-def

by (cases zzs, auto)

note Cons = Cons[unfolded zzs]

from Cons(4) have two: (y,z) \in r \lor y = z \land (ys,zs) \in \text{lexord } r by auto

{ assume (xs,ys) \in \text{lexord } r and (ys,zs) \in \text{lexord } r

from Cons(1)[OF - this] Cons(2)

have (xs,zs) \in \text{lexord } r by auto

} note ind1 = this

{ assume (x,y) \in r and (y,z) \in r

from Cons(2)[OF - this] have (x,z) \in r by auto

} note ind2 = this

from one two ind1 ind2

have (x,z) \in r \lor x = z \land (xs,zs) \in \text{lexord } r by blast

thus ?case unfolding zzs by auto

qed

**lemma** lexord-trans:
\[
[ (x, y) \in \text{lexord } r; (y, z) \in \text{lexord } r; \text{trans } r ] \implies (x, z) \in \text{lexord } r
\]

by (auto simp: trans-def intro:lexord-partial-trans)

**lemma** lexord-transI: \( \text{trans } r \implies \text{trans } (\text{lexord } r) \)

by (rule transI, drule lexord-trans, blast)

**lemma** lexord-linear: (! a. (a,b)\in r \implies a = b \implies (b,a) \in r) \implies (x,y) : \text{lexord } r \mid x = y \mid (y,x) : \text{lexord } r

apply (rule-tac x = y in spec)

apply (induct-tac x, rule allI)

apply (case-tac x, simp, simp)

apply (rule allI, case-tac x, simp, simp)

by blast

**lemma** lexord-irrefl:

irrefl R \implies irrefl (\text{lexord } R)

by (simp add: irrefl-def lexord-irreflexive)
lemma lexord-asym:
  assumes asym R
  shows asym (lexord R)
proof
  from assms obtain irrefl R by (blast elim: asym_cases)
  then show irrefl (lexord R) by (rule lexord-irrefl)
next
  fix xs ys
  assume (xs, ys) ∈ lexord R
  then show (ys, xs) /∈ lexord R
    proof (induct xs arbitrary: ys)
      case Nil
      then show ?case by simp
    next
      case (Cons x xs)
      then obtain z zs where ys: ys = z # zs by (cases ys) auto
      with assms Cons show ?case by (auto elim: asym_cases)
    qed
qed

lemma lexord-asymmetric:
  assumes asym R
  assumes hyp: ((a, b) ∈ lexord R)
  shows (b, a) /∈ lexord R
proof
  from (asym R) have asym (lexord R) by (rule lexord-asym)
  then show ?thesis by (rule asym_cases) (auto simp add: hyp)
qed

Predicate version of lexicographic order integrated with Isabelle’s order type classes. Author: Andreas Lochbihler

context ord begin

inductive lexordp :: 'a list ⇒ 'a list ⇒ bool
where
  Nil: lexordp [] (y # ys)
| Cons: x < y ⇒ lexordp (x # xs) (y # ys)
| Cons-eq:
  [¬ x < y; ¬ y < x; lexordp xs ys] ⇒ lexordp (x # xs) (y # ys)

lemma lexordp-simps [simp]:
  lexordp [] ys = (ys ≠ [])
  lexordp xs [] = False
  lexordp (x # xs) (y # ys) ⇔ x < y ∨ ¬ y < x ∧ lexordp xs ys
by(subst lexordp.simps, fastforce simp add: neg-nil-conv)+

inductive lexordp-eq :: 'a list ⇒ 'a list ⇒ bool where
  Nil: lexordp-eq [] ys
THEORY "List"

Cons: $x < y \implies \text{lexordp-eq} (x \# xs) (y \# ys)$
Cons-eq: $[\neg x < y; \neg y < x; \text{lexordp-eq} xs ys] \implies \text{lexordp-eq} (x \# xs) (y \# ys)$

lemma lexordp-eq-simps [simp]:
  lexordp-eq [] ys = True
  lexordp-eq xs [] \iff xs = []
  lexordp-eq (x # xs) [] = False
  lexordp-eq (x # xs) (y # ys) \iff x < y \lor \neg y < x \land \text{lexordp-eq} xs ys
by (subst lexordp-eq, simp, fastforce)+

lemma lexordp-append-rightI: ys \neq Nil \implies lexordp xs (xs @ ys)
by (induct xs) (auto simp add: neq-Nil-conv)

lemma lexordp-append-left-rightI: x < y \implies lexordp (us @ x # xs) (us @ y # ys)
by (induct us) auto

lemma lexordp-eq-refl:
  assumes irrefl: \forall x. \neg x < x
  shows \neg lexordp xs xs
proof
  assume lexordp xs xs
  thus False by (induct xs ys \equiv xs) (simp-all add: irrefl)
qed

lemma lexordp-into-lexordp-eq:
  assumes lexordp xs ys
  shows lexordp-eq xs ys
using assms by induct simp-all

end

declare ord.lexordp-simps [simp, code]
declare ord.lexordp-eq-simps [code, simp]

lemma lexord-code [code, code-unfold]: lexordp = ord.lexordp less
unfolding lexordp-def ord.lexordp-def ..
context order begin

lemma lexordp-antisym:
  assumes lexordp xs ys lexordp ys xs
  shows False
using assms by (induct auto)

lemma lexordp-irreflexive: \neg lexordp xs xs
by (rule lexordp-irreflexive) simp

end

context linorder begin

lemma lexordp-cases [consumes 1, case-names Nil Cons Cons-eq, cases pred: lexordp]:
  assumes lexordp xs ys
  obtains (Nil) y ys' where xs = [] ys = y # ys'
  | (Cons) x xs' y ys' where xs = x # xs' ys = y # ys' x < y
  | (Cons-eq) x xs' y ys' where xs = x # xs' ys = x # ys' lexordp xs' ys'
using assms by cases (fastforce simp add: not-less-iff-gr-or-eq)+

lemma lexordp-induct [consumes 1, case-names Nil Cons Cons-eq, induct pred: lexordp]:
  assumes major: lexordp xs ys
  and Nil: \And y ys. P [] (y # ys)
  and Cons: \And x xs y ys. x < y \Longrightarrow P (x # xs) (y # ys)
  and Cons-eq: \And x xs ys. \If lexordp xs ys; P xs ys \then P (x # xs) (x # ys)
  shows P xs ys
using major by (induct (simp-all add: Nil Cons not-less-iff-gr-or-eq Cons-eq)

lemma lexordp-iff:
  lexordp xs ys \iff (\exists x. vs. ys = xs @ x # vs) \lor (\exists us a b vs. us @ a # vs \land ys = us @ b # vs).
(is ?lhs = ?rhs)

proof
  assume ?lhs thus ?rhs

proof (induct)
  case Cons-eq thus ?case by simp (metis append.simps(2))
next
  assume ?rhs thus ?lhs
  by (auto intro: lexordp-append-leftI [where us=[], simplified] lexordp-append-leftI)

lemma lexordp-conv-lexord:
  lexordp xs ys \iff (xs, ys) \in lexord { (x, y). x < y }
by (simp add: lexordp-iff lexord-def)
lemma lexordp-eq-antisym:
  assumes lexordp-eq xs ys lexordp-eq ys xs
  shows xs = ys
  using assms by induct simp-all

lemma lexordp-eq-trans:
  assumes lexordp-eq xs ys and lexordp-eq ys zs
  shows lexordp-eq xs zs
  using assms
  apply (induct arbitrary: zs)
  apply (case-tac [2−3] zs)
  apply auto
  done

lemma lexordp-linear:
  assumes lexordp xs ys \lor \neg lexordp-eq ys xs
proof
  (induct xs arbitrary: ys)
  case Nil thus ?case by (cases ys) simp-all
next
  case Cons thus ?case by (cases ys) auto
  qed

lemma lexordp-conv-lexordp-eq:
  assumes lexordp-eq xs ys \leftrightarrow lexordp-eq(xs ys) \land \neg lexordp-eq ys xs
proof
  (induct arbitrary: ys)
  moreover hence \neg lexordp-eq ys xs by induct simp-all
  ultimately show ?rhs by (simp add: lexordp-into-lexordp-eq)
next
  assume ?rhs
  hence \neg lexordp-eq ys xs \land \neg lexordp-eq ys xs by simp-all
  thus ?lhs by induct simp-all
  qed

lemma lexordp-eq-linear:
  assumes lexordp-eq(xs ys) \leftrightarrow \neg lexordp-eq(xs ys)
proof
  (auto simp add: lexordp-conv-lexordp-eq lexordp-eq-refl dest: lexordp-eq-antisym)

lemma lexordp-eq-linear:
  assumes lexordp-eq(xs ys) \leftrightarrow \neg lexordp-eq(xs ys)
proof
  (auto simp add: lexordp-conv-lexordp-eq lexordp-eq-refl dest: lexordp-eq-antisym)
apply auto
done

lemma lexordp-linorder: 
  class.linorder lexordp-eq lexordp
by unfold-locales(auto simp add: lexordp-conv-lexordp-eq lexordp-eq-refl lexordp-eq-antisym
intro: lexordp-eq-trans del: disjCI intro: lexordp-eq-linear)
end

67.3.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition measures fs = inv-image (lex less-than) (%a. map (%f. f a) fs)

lemma uf-measures[simp]: uf (measures fs)
unfolding measures-def
by blast

lemma in-measures[simp]:
  (x, y) ∈ measures [] = False
  (x, y) ∈ measures (f # fs)
    = (f x < f y ∨ (f x = f y ∧ (x, y) ∈ measures fs))
unfolding measures-def
by auto

lemma measures-less: f x < f y ==⇒ (x, y) ∈ measures (f # fs)
by simp

lemma measures-leseq: f x ≤ f y ==⇒ (x, y) ∈ measures fs ==⇒ (x, y) ∈ measures (f # fs)
by auto

67.3.4 Lifting Relations to Lists: one element

definition listrel1 :: ('a × 'a) set ⇒ ('a list × 'a list) set where
listrel1 r = \{\{xs,ys\}.
  ∃ us z z' vs. xs = us @ z # vs ∧ (z,z') ∈ r ∧ ys = us @ z' # vs\}

lemma listrel1I:
  [ (x, y) ∈ r;  xs = us @ x # vs;  ys = us @ y # vs ] ==⇒
  (xs, ys) ∈ listrel1 r
unfolding listrel1-def by auto

lemma listrel1IE:
  [ (xs, ys) ∈ listrel1 r;
    ![x y us vs. [ (x, y) ∈ r;  xs = us @ x # vs;  ys = us @ y # vs ] ==⇒ P
  ] ==⇒ P
unfolding listrel1-def by auto
lemma not-Nil-listrel1 [iff]: (\[], \_\_\_) \notin listrel1 r
unfolding listrel1-def by blast

lemma not-listrel1-Nil [iff]: (\_, \_\[]\_) \notin listrel1 r
unfolding listrel1-def by blast

lemma Cons-listrel1-Cons [iff]:
(x \# xs, y \# ys) \in listrel1 r \iff (x, y) \in r \land xs = ys \lor x = y \land (xs, ys) \in listrel1 r
by (simp add: listrel1-def Cons-eq-append-conv) (blast)

lemma listrel1I1: (x, y) \in r =\Rightarrow (x \# xs, y \# ys) \in listrel1 r
by fast

lemma listrel1I2: (xs, ys) \in listrel1 r =\Rightarrow (x \# xs, x \# ys) \in listrel1 r
by fast

lemma append-listrel1I:
(xs, ys) \in listrel1 r \land us = vs \lor xs = ys \land (us, vs) \in listrel1 r
\iff (xs \# us, ys \# vs) \in listrel1 r
unfolding listrel1-def by auto

lemma Cons-listrel1E1 [elim!]:
assumes (x \# xs, y \# ys) \in listrel1 r
and \(\forall y.\) ys = y \# xs \Longrightarrow (x, y) \in r \Longrightarrow R
and \(\forall zs.\) xs = x \# zs \Longrightarrow (zs, ys) \in listrel1 r \Longrightarrow R
shows R
using assms by (cases ys) blast+

lemma Cons-listrel1E2 [elim!]:
assumes (xs, y \# ys) \in listrel1 r
and \(\forall x.\) xs = x \# ys \Longrightarrow (x, y) \in r \Longrightarrow R
and \(\forall zs.\) xs = y \# zs \Longrightarrow (zs, ys) \in listrel1 r \Longrightarrow R
shows R
using assms by (cases xs) blast+

lemma snoc-listrel1-snoc-iff:
(xs \# [x], ys \# [y]) \in listrel1 r
\iff (xs, ys) \in listrel1 r \land x = y \lor xs = ys \land (x, y) \in r (is \ ?L \iff ?R)

proof
assume ?L thus ?R
by (fastforce simp: listrel1-def snoc-eq-iff-butlast butlast-append)
next
assume ?R then show ?L unfolding listrel1-def by force
qed

lemma listrel1-eq-len: (xs,ys) \in listrel1 r \Longrightarrow length xs = length ys
unfolding listrel1-def by auto
lemma `listrel1-mono`:
\[ r \subseteq s \implies listrel1\ r \subseteq listrel1\ s \]
unfolding `listrel1-def` by blast

lemma `listrel1-converse`:
\[ listrel1\ (r^-1) = (listrel1\ r)^{-1} \]
unfolding `listrel1-def` by blast

lemma `in-listrel1-converse`:
\[ (x, y) : listrel1\ (r^-1) \iff (x, y) : (listrel1\ r)^{-1} \]
unfolding `listrel1-def` by blast

lemma `listrel1-iff-update`:
\[ (xs, ys) \in (\text{listrel1}\ r) \iff (\exists\ y\ n. (xs ! n, y) \in r \land n < \text{length}\ xs \land ys = xs[n:=y]) \text{ (is} \ ?L \iff \ ?R \text{)} \]
proof
assume \(?L\)
then obtain \(x\ y\ v\ u\ w\ z\) where \(xs = u \circ x \# v\ ys = u \circ y \# v\ (x, y) \in r\)
unfolding `listrel1-def` by auto
then have \(ys = xs[\text{length}\ u := y]\) and \(\text{length}\ u < \text{length}\ xs\)
and \((xs \# \text{length}\ u, y) \in r\) by auto
then show \(?R\) by auto
next
assume \(?R\)
then obtain \(x\ y\ n\) where \((xs!n, y) \in r \land n < \text{size}\ xs\ ys = xs[n:=y]\) \(x = xs!n\)
by auto
then obtain \(u\ v\) where \((xs!n, y) \in r \land n < \text{size}\ xs\ ys = xs[n:=y]\) \(x = xs!n\)
by (auto intro: upd-conv-take-nth-drop id-take-nth-drop)
then show \(?L\) by (auto simp: `listrel1-def`)
qed

Accessible part and wellfoundedness:

lemma `Cons-acc-listrel1I` [intro!]:
\[ x \in \text{Wellfounded.acc}\ r \implies xs \in \text{Wellfounded.acc}\ (\text{listrel1}\ r) \implies (x \# xs) \in \text{Wellfounded.acc}\ (\text{listrel1}\ r) \]
apply (induct arbitrary: xs set: Wellfounded.acc)
apply (erule thin-rl)
apply (erule acc-induct)
apply (rule accI)
apply (blast)
done

lemma `lists-accD`:
\[ xs \in \text{lists}\ (\text{Wellfounded.acc}\ r) \implies xs \in \text{Wellfounded.acc}\ (\text{listrel1}\ r) \]
apply (induct set: lists)
apply (rule accI)
apply simp
apply (rule accI)

apply (fast dest: acc-downward)
done

lemma lists-accI: \( x \in \text{Wellfounded.acc} \) \((\text{listrel1 r}) \implies x \in \text{lists (Wellfounded.acc)}\)
apply (induct set: \( \text{Wellfounded.acc} \))
apply clarify
apply (rule accI)
apply (fastforce dest!: in-set-conv-decomp THEN iffD1] simp: listrel1-def)
done

lemma wf-listrel1-iff [simp]: \( \text{wf (listrel1 r)} = \text{wf r} \)
by (auto simp: wf-acc-iff intro: lists-accD lists-accI [THEN Cons-in-lists-iff [THEN iffD1, THEN conjunct1]])

67.3.5 Lifting Relations to Lists: all elements

inductive-set
listrel :: ('a × 'b) set ⇒ ('a list × 'b list) set
for \( r :: ('a × 'b) set \)
where

\[ \begin{align*}
\text{Nil: } & (\[]\, \), (\[]\, \) \in \text{listrel r} \\
\text{Cons: } & (x,y) \in r; (xs,ys) \in \text{listrel r} \implies (x\#xs, y\#ys) \in \text{listrel r}
\end{align*} \]

inductive-cases listrel-Nil1 [elim!]: (\[],xs) \in \text{listrel r}
inductive-cases listrel-Nil2 [elim!]: (xs,\[]\) \in \text{listrel r}
inductive-cases listrel-Cons1 [elim!]: (y\#ys,xs) \in \text{listrel r}
inductive-cases listrel-Cons2 [elim!]: (xs,y\#ys) \in \text{listrel r}

lemma listrel-eq-len: \((xs, ys) \in \text{listrel r} \implies \text{length x} = \text{length y}\)
by (induct rule: listrel.induct) auto

lemma listrel-iff-zip [code-unfold]: \((xs,ys) : \text{listrel r} \implies \text{length x} = \text{length y} \& \forall (x,y) \in \text{set (zip xs ys)}. (x,y) \in r) \text{ (is ?L} \leftrightarrow \text{?R)}
proof
assume ?L thus ?R by induct (auto intro: listrel-eq-len)
next
assume ?R thus ?L

apply (clarify)
by (induct rule: list-induct2) (auto intro: listrel.intros)
qed

lemma listrel-iff-nth: \((xs,ys) : \text{listrel r} \implies \text{length x} = \text{length y} \& \forall n < \text{length x}. (xs!n, ys!n) \in r) \text{ (is ?L} \leftrightarrow \text{?R)}
by (auto simp add: all-set-cone-all-nth listrel-iff-zip)
lemma listrel-mono: $r \subseteq s \implies \text{listrel } r \subseteq \text{listrel } s$
apply clarify
apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

lemma listrel-subset: $r \subseteq A \times A \implies \text{listrel } r \subseteq \text{lists } A \times \text{lists } A$
apply clarify
apply (erule listrel.induct, auto)
done

lemma listrel-refl-on: refl-on $A$ $r \implies$ refl-on ($\text{lists } A$) (listrel $r$)
apply (simp add: refl-on-def listrel-subset Ball-def)
apply (rule allI)
apply (induct-tac $x$)
apply (auto intro: listrel.intros)
done

lemma listrel-sym: sym $r \implies$ sym (listrel $r$)
apply (auto simp add: sym-def)
apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

lemma listrel-trans: trans $r \implies$ trans (listrel $r$)
apply (simp add: trans-def)
apply (intro allI)
apply (rule impI)
apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

theorem equiv-listrel: equiv $A$ $r \implies$ equiv ($\text{lists } A$) (listrel $r$)
by (simp add: equiv-def listrel-refl-on listrel-sym listrel-trans)

lemma listrel-rtrancl-refl[iff]: $(xs,xs) : \text{listrel } (r^\ast)$
using listrel-refl-on[of UNIV, OF refl-rtrancl]
by (auto simp: refl-on-def)

lemma listrel-rtrancl-trans:
  $[(xs,ys) : \text{listrel } (r^\ast); \ (ys,zs) : \text{listrel } (r^\ast)] \implies (xs,zs) : \text{listrel } (r^\ast)$
by (metis listrel-trans trans-def trans-rtrancl)

lemma listrel-Nil [simp]: listrel $r \text{ " } [\ ] = [\ ]$
by (blast intro: listrel.intros)

lemma listrel-Cons:
THEORY “List”

listrel r " \{ x \# xs \} = set-Cons (r^-\{x\}) (listrel r " \{xs\})
by (auto simp add: set-Cons-def intro: listrel.intros)

Relating listrel1, listrel and closures:

lemma listrel1-rtrancl-subset-rtrancl-listrel1:
listrel1 (r^-*) ⊆ (listrel1 r)^*
proof (rule subrelI)
fix xs ys assume 1: (xs,ys) ∈ listrel1 (r^-*)
{ fix x y us vs
  have (x,y) : r^-* ⇒ (us @ x # vs, us @ y # vs) : (listrel1 r)^*
  proof (induct rule: rtrancl.induct)
    case rtrancl-refl show ?case by simp
  next
    case rtrancl-into-rtrancl thus ?case by (metis listrel1I rtrancl.rtrancl-into-rtrancl)
  qed }
thus (xs,ys) ∈ (listrel1 r)^* using 1 by (blast intro: listrel1E)
qed

lemma rtrancl-listrel1-eq-len: (x,y) ∈ (listrel1 r)^* ⇒ length x = length y
by (induct rule: rtrancl.induct) (auto intro: listrel1-eq-len)

lemma rtrancl-listrel1-ConsI1:
(x,y) ∈ (listrel1 r)^* ⇒ (x#xs, y#ys) ∈ (listrel1 r)^*
apply (induct rule: rtrancl.induct)
  apply simp
by (metis listrel1I2 rtrancl.rtrancl-into-rtrancl)

lemma rtrancl-listrel1-ConsI2:
(x,y) ∈ r^-* ⇒ (x#xs, y#ys) ∈ (listrel1 r)^*
⇒ (x # xs, y # ys) ∈ (listrel1 r)^*
by (blast intro: rtrancl-trans rtrancl-listrel1-ConsI1
  subsetD[OF listrel1-rtrancl-subset-rtrancl-listrel1 listrel1IIi])

lemma listrel1-subset-listrel:
  r ⊆ r' ⇒ refl r' ⇒ listrel1 r ⊆ listrel(r')
by (auto elim!: listrel1E simp add: listrel-iff-zip set-zip refl-on-def)

lemma listrel-reflcl-if-listrel1:
  (xs,ys) : listrel1 r ⇒ (xs,ys) : listrel(r^-*)
by (erule listrel1E) (auto simp add: listrel-iff-zip set-zip)

lemma listrel-rtrancl-eq-rtrancl-listrel1: listrel (r^-*) = (listrel1 r)^*
proof
{ fix x y assume (x,y) ∈ listrel (r^-*)
  then have (x,y) ∈ (listrel1 r)^*
    by (induct (auto intro: rtrancl-listrel1-ConsI2) )
  then show listrel (r^-*) ⊆ (listrel1 r)^*
  by (rule subrelI)
}
THEORY "List"

next
show listrel $\blacktriangleright^\ast \supseteq (\text{listrel1 } r)^\ast$
proof (rule subrelI)
fix xs ys assume (xs, ys) $\in (\text{listrel1 } r)^\ast$
then show (xs, ys) $\in \text{listrel } (r^\ast)$
proof induct
  case base show ?case by (auto simp add: listrel-iff-zip set-zip)
next
  case (step ys zs)
  thus ?case by (metis listrel-reflcl-if-listrel1 listrel-rtrancl-trans)
qed
qed

lemma rtrancl-listrel1-if-listrel:
(xs, ys) : listrel r $\Longrightarrow (xs, ys) : (\text{listrel1 } r)^\ast$
by (metis listrel-rtrancl-eq-rtrancl-listrel1 subsetD [OF listrel-mono] r-into-rtrancl subsetI)

lemma listrel-subset-rtrancl-listrel1: listrel r $\subseteq (\text{listrel1 } r)^\ast$
by (fast intro: rtrancl-listrel1-if-listrel)

67.4 Size function

lemma [measure-function]: is-measure $f \Longrightarrow$ is-measure $(\text{size-list } f)$
by (rule is-measure-trivial)

lemma [measure-function]: is-measure $f \Longrightarrow$ is-measure $(\text{size-option } f)$
by (rule is-measure-trivial)

lemma size-list-estimation[termination-simp]:
x $\in$ set xs $\Longrightarrow y < f x \Longrightarrow y < \text{size-list } f \text{ xs}$
by (induct xs) auto

lemma size-list-estimation'[termination-simp]:
x $\in$ set xs $\Longrightarrow y \leq f x \Longrightarrow y \leq \text{size-list } f \text{ xs}$
by (induct xs) auto

lemma size-list-map[simp]: size-list $f$ (map $g$ xs) = size-list $(f \circ g)$ xs
by (induct xs) auto

lemma size-list-append[simp]: size-list $f$ (xs @ ys) = size-list $f$ xs + size-list $f$ ys
by (induct xs, auto)

lemma size-list-pointwise[termination-simp]:
($\forall x \in$ set xs $\Longrightarrow f x \leq g x$) $\Longrightarrow$ size-list $f$ xs $\leq$ size-list $g$ xs
by (induct xs) force+
67.5 Monad operation

**definition** bind :: 'a list ⇒ ('a ⇒ 'b list) ⇒ 'b list where

\[ \text{bind } xs \ f = \text{concat} \ (\text{map } f \ xs) \]

**hide-const** (open) bind

**lemma** bind-simps [simp]:

\[ \begin{align*}
\text{List.bind } [] \ f &= [] \\
\text{List.bind } (x \ # \ xs) \ f &= f \ x \ @ \ \text{List.bind } xs \ f
\end{align*} \]

by (simp-all add: bind-def)

67.6 Transfer

**definition** embed-list :: nat list ⇒ int list where

\[ \text{embed-list } l = \text{map} \ \text{int} \ l \]

**definition** nat-list :: int list ⇒ bool where

\[ \text{nat-list } l = \text{nat-set} \ (\text{set} \ l) \]

**definition** return-list :: int list ⇒ nat list where

\[ \text{return-list } l = \text{map} \ \text{nat} \ l \]

**lemma** transfer-nat-int-list-return-embed: nat-list l ⇒ embed-list l

\[ \text{unfolding embed-list-def return-list-def nat-list-def nat-set-def} \]

\[ \text{apply } (\text{induct } l) \]

\[ \text{apply } \text{auto} \]

done

**lemma** transfer-nat-int-list-functions:

\[ l \ @ \ m = \text{return-list} \ (\text{embed-list } l \ @ \ \text{embed-list } m) \]

\[ [] = \text{return-list} [] \]

\[ \text{unfolding return-list-def embed-list-def} \]

\[ \text{apply } \text{auto} \]

\[ \text{apply } (\text{induct } l, \text{auto}) \]

\[ \text{apply } (\text{induct } m, \text{auto}) \]

done

67.7 Code generation

Optional tail recursive version of map. Can avoid stack overflow in some target languages.

**fun** map-tailrec-rev :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list ⇒ 'b list where

\[ \text{map-tailrec-rev } f \ [] \ bs = bs \ |
\]

\[ \text{map-tailrec-rev } f \ (a \ # \ as) \ bs = \text{map-tailrec-rev } f \ as \ (f \ a \ # \ bs) \]

**lemma** map-tailrec-rev:

\[ \text{map-tailrec-rev } f \ as \ bs = \text{rev} \ (\text{map} \ f \ as) \ @ \ bs \]
by (induction as arbitrary: bs) simp-all

**definition** map-tailrec :: "'a ⇒ 'b) ⇒ 'a list ⇒ 'b list where map-tailrec f as = rev (map-tailrec-rev f as [])

Code equation:

**lemma** map-eq-map-tailrec: map = map-tailrec

by (simp add: fun-eq-iff map-tailrec-def map-tailrec-rev)

### 67.7.1 Counterparts for set-related operations

**definition** member :: "'a list ⇒ 'a ⇒ bool where [code-abbrev]: member xs x ←→ x ∈ set xs

Use member only for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about.

**lemma** member-rec [code]:

member (x # xs) y ←→ (x = y ∨ member xs y)
member [] y ←→ False

by (auto simp add: member-def)

**lemma** in-set-member:

$x \in set\ xs$ ←→ member xs x

by (simp add: member-def)

**abbreviation** list-all == pred-list

**lemma** list-all-iff [code-abbrev]: list-all P xs ←→ Ball (set xs) P

unfolding pred-list-def ..

**definition** list-ex :: ("'a ⇒ bool) ⇒ 'a list ⇒ bool where list-ex iff [code-abbrev]: list-ex P xs ←→ Bex (set xs) P

**definition** list-ex1 :: ("'a ⇒ bool) ⇒ 'a list ⇒ bool where list-ex1-iff [code-abbrev]: list-ex1 P xs ←→ (∃! x. x ∈ set xs ∧ P x)

Usually you should prefer $\forall x ∈ set\ xs$, $\exists x ∈ set\ xs$ and $\exists! x. x ∈ set\ xs$ ∧ - over list-all, list-ex and list-ex1 in specifications.

**lemma** list-all-simps [code]:

list-all P (x # xs) ←→ P x ∧ list-all P xs
list-all P [] ←→ True

by (simp-all add: list-all-if)

**lemma** list-ex-simps [simp, code]:

list-ex P (x # xs) ←→ P x ∨ list-ex P xs
list-ex P [] ←→ False

by (simp-all add: list-ex-if)

**lemma** list-ex1-simps [simp, code]:
list-ex1 $P [] = \text{False}$
list-ex1 $P (x \# xs) = (\text{if } P x \text{ then } \text{list-all } (\lambda y. \neg P y \lor x = y) \text{ xs else list-ex1 } P \text{ xs})$
by (auto simp add: list-ex1-iff list-all-iff)

lemma Ball-set-list-all:
Ball (set xs) $P \iff$ list-all $P$ xs
by (simp add: list-all-iff)

lemma Bex-set-list-ex:
Bex (set xs) $P \iff$ list-ex $P$ xs
by (simp add: list-ex-iff)

lemma list-all-append [simp]:
list-all $P$ (xs @ ys) $\iff$ list-all $P$ xs $\land$ list-all $P$ ys
by (auto simp add: list-all-iff)

lemma list-ex-append [simp]:
list-ex $P$ (xs @ ys) $\iff$ list-ex $P$ xs $\lor$ list-ex $P$ ys
by (auto simp add: list-ex-iff)

lemma list-all-rev [simp]:
list-all $P$ (rev xs) $\iff$ list-all $P$ xs
by (simp add: list-all-iff)

lemma list-ex-rev [simp]:
list-ex $P$ (rev xs) $\iff$ list-ex $P$ xs
by (simp add: list-ex-iff)

lemma list-all-length:
list-all $P$ xs $\iff$ ($\forall n < \text{length } xs. P (xs ! n)$)
by (auto simp add: list-all-iff set-conv-nth)

lemma list-ex-length:
list-ex $P$ xs $\iff$ ($\exists n < \text{length } xs. P (xs ! n)$)
by (auto simp add: list-ex-iff set-conv-nth)

lemma list-all-cong [fundef-cong]:
x$s = y$s $\Longrightarrow$ ($\forall x. x \in \text{set } y$s $\Longrightarrow$ $f x = g x$) $\Longrightarrow$ list-all $f$ xs = list-all $g$ ys
by (simp add: list-all-iff)

lemma list-ex-cong [fundef-cong]:
x$s = y$s $\Longrightarrow$ ($\forall x. x \in \text{set } y$s $\Longrightarrow$ $f x = g x$) $\Longrightarrow$ list-ex $f$ xs = list-ex $g$ ys
by (simp add: list-ex-iff)

definition can-select :: ('a ⇒ bool) ⇒ 'a set ⇒ bool where
[code-abbrev]: can-select $P$ A = ($\exists x \in A. P x$)

lemma can-select-set-list-ex1 [code]:
can-select $P \ (\text{set } A) = \text{list-ex1 } P \ A$

by (simp add: list-ex1-iff can-select-def)

Executable checks for relations on sets

definition listrel1p :: \text{('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool} where
listrel1p $r$ $xs$ $ys$ = \((xs, ys) \in \text{listrel1} \ {(x, y), r x y}\)\

lemma [code-unfold]:
\((xs, ys) \in \text{listrel1} \ r \ = \ \text{listrel1p} \ (λ \ x y. \ (x, y) \in r) \ xs \ ys\)

unfolding listrel1p-def by auto

lemma [code]:
listrel1p $r \ [] \ xs \ = \ False$
listrel1p $r \ xs \ [] \ = \ False$
listrel1p $r \ (x \ # \ xs) \ (y \ # \ ys) \ \longleftrightarrow$
\hspace{1cm} $r x y \land xs = ys \lor x = y \land \text{listrel1p} \ r \ xs \ ys$
by (simp add: listrel1p-def)+

definition lexordp :: \text{('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool} where
lexordp $r$ $xs$ $ys$ = \((xs, ys) \in \text{lexord} \ {(x, y), r x y}\)\

lemma [code-unfold]:
\((xs, ys) \in \text{lexord} \ r \ = \ \text{lexordp} \ (λ \ x y. \ (x, y) \in r) \ xs \ ys\)

unfolding lexordp-def by auto

lemma [code]:
lexordp $r \ xs \ [] \ = \ False$
lexordp $r \ [] \ (y\#ys) \ = \ True$
lexordp $r \ (x \ # \ xs) \ (y \ # \ ys) \ = \ (r x y \ | \ (x = y \ & \ \text{lexordp} \ r \ xs \ ys))$

unfolding lexordp-def by auto

Bounded quantification and summation over nats.

lemma atMost-upto [code-unfold]:
\{..\ n\} \ = \ \text{set} \ [\theta..<\text{Suc } \ n]\nby auto

lemma atLeast-upt [code-unfold]:
\{..<\ n\} \ = \ \text{set} \ [\theta..<\ n]\nby auto

lemma greaterThanLessThan-upt [code-unfold]:
\{\ n..<\ m\} \ = \ \text{set} \ [\text{Suc } \ n..<\ m]\nby auto

lemmas atLeastLessThan-upt [code-unfold] = set-upt [symmetric]

lemma greaterThanAtMost-upt [code-unfold]:
\{\ n..<\ \text{Suc} \ m\} \ = \ \text{set} \ [\text{Suc } \ n..<\ \text{Suc} \ m]\nby auto
by auto

lemma atLeastAtMost-upt [code-unfold]:
\{n..m\} = set [n..<Suc m]
by auto

lemma all-nat-less-eq [code-unfold]:
(\forall m<n::nat. P m) \iff (\forall m \in \{0..<n\}. P m)
by auto

lemma ex-nat-less-eq [code-unfold]:
(\exists m<n::nat. P m) \iff (\exists m \in \{0..<n\}. P m)
by auto

lemma all-nat-less [code-unfold]:
(\forall m\leq n::nat. P m) \iff (\forall m \in \{0..n\}. P m)
by auto

lemma ex-nat-less [code-unfold]:
(\exists m\leq n::nat. P m) \iff (\exists m \in \{0..n\}. P m)
by auto

lemma setsum-set-upt-conv-listsum-nat [code-unfold]:
setsum f (set [m..<n]) = listsum (map f [m..<n])
by (simp add: interv-listsum-conv-setsum-set-nat)

Bounded LEAST operator:

definition Bleast S P = (LEAST x. x \in S \land P x)

definition abort-Bleast S P = (LEAST x. x \in S \land P x)

declare [[code abort: abort-Bleast]]

lemma Bleast-code [code]:
Bleast (set xs) P = (case filter P (sort xs) of
 x#zs \Rightarrow x |
 [] \Rightarrow abort-Bleast (set xs) P)

proof (cases filter P (sort xs))
case Nil thus ?thesis by (simp add: Bleast-def abort-Bleast-def)
next
case (Cons x ys)
have (LEAST x. x \in set xs \land P x) = x

proof (rule Least-equality)
show x \in set xs \land P x
  by (metis Cons Cons-eq-filter-iff in-set-comp-decomp set-sort)
next
fix y assume y : set xs \land P y
hence y : set (filter P xs) by auto
thus x \leq y
by (metis Cons eq-iff filter-sort set-ConsD set-sort sorted-Cons sorted-sort)
qed
thus ?thesis using Cons by (simp add: Bleast-def)
qed

declare Bleast-def[symmetric, code-unfold]

Summation over ints.

lemma greaterThanLessThan-upto [code-unfold]:
\{i..<j::int\} = set [i+1..j - 1]
by auto

lemma atLeastLessThan-upto [code-unfold]:
\{i..<j::int\} = set [i..j - 1]
by auto

lemma greaterThanAtMost-upto [code-unfold]:
\{i..<j::int\} = set [i+1..j]
by auto

lemmas atLeastAtMost-upto [code-unfold] = set-upto [symmetric]

lemma setsum-set-upto-conv-listsum-int [code-unfold]:
setsum f (set [i..j::int]) = listsum (map f [i..j])
by (simp add: interv-listsum-conv-setsum-set-int)

67.7.2 Optimizing by rewriting

definition null :: 'a list ⇒ bool where
[code-abbrev]: null xs ←→ xs = []

Efficient emptyness check is implemented by null.

lemma null-rec [code]:
null (x # xs) ←→ False
null [] ←→ True
by (simp-all add: null-def)

lemma eq-Nil-null:
xs = [] ←→ null xs
by (simp add: null-def)

lemma equal-Nil-null [code-unfold]:
HOL.equal xs [] ←→ null xs
HOL.equal [] = null
by (auto simp add: equal null-def)

definition maps :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list where
[code-abbrev]: maps f xs = concat (map f xs)
**THEORY “List”**

**Definition**

```
definition map-filter :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
  where
  [code-post]: map-filter f xs = map (the o f) (filter (λx. f x ≠ None) xs)
```

Operations `maps` and `map-filter` avoid intermediate lists on execution – do not use for proving.

**Lemma** `maps-simps` [code]:

```
maps f (x # xs) = f x @ maps f xs
maps f [] = []
by (simp-all add: maps-def)
```

**Lemma** `map-filter-simps` [code]:

```
map-filter f (x # xs) = (case f x of None ⇒ map-filter f xs | Some y ⇒ y #
  map-filter f xs)
map-filter f [] = []
by (simp-all add: map-filter-def split: option.split)
```

**Lemma** `concat-map-maps`:

```
concat (map f xs) = maps f xs
by (simp add: maps-def)
```

**Lemma** `map-filter-map-filter` [code-unfold]:

```
map f (filter P xs) = map-filter (λx. if P x then Some (f x) else None) xs
by (simp add: map-filter-def)
```

Optimized code for ∀ i∈{a..b::int} and ∀ n:{a..<b::nat} and similarly for ∃.

**Definition**

```
definition all-interval-nat :: (nat ⇒ bool) ⇒ nat ⇒ nat ⇒ bool
  where
  all-interval-nat P i j ←→ (∀ n ∈ {i..<j}. P n)
```

**Lemma** [code]:

```
all-interval-nat P i j ←→ i ≥ j ∨ P i ∧ all-interval-nat P (Suc i) j
proof –
have ∗: ∀ n. P i ⇒ ∀ n∈{Suc i..<j}. P n ⇒ i ≤ n ⇒ n < j ⇒ P n
proof –
fix n
assume P i ∨ n∈{Suc i..<j}. P n i ≤ n n < j
then show P n by (cases n = i) simp-all
qed
show ?thesis by (auto simp add: all-interval-nat-def intro: ∗)
qed
```

**Lemma** `list-all-iff-all-interval-nat` [code-unfold]:

```
list-all P [i..<j] ←→ all-interval-nat P i j
by (simp add: list-all-iff all-interval-nat-def)
```

**Lemma** `list-ex-iff-not-all-interval-nat` [code-unfold]:

```
list-ex P [i..<j] ←→ ¬ (all-interval-nat (Not o P) i j)
by (simp add: list-ex-iff all-interval-nat-def)
```
definition all-interval-int :: (int ⇒ bool) ⇒ int ⇒ int ⇒ bool where
all-interval-int P i j ←→ (∀ k ∈ {i..j}. P k)

lemma [code]:
all-interval-int P i j ←→ i > j ∨ P i ∧ all-interval-int P (i + 1) j
proof −
have *: ∃k. P i ⇒ ∀ k∈[i+1..j]. P k ⇒ i ≤ k ⇒ k ≤ j ⇒ P k
proof −
  assume P i ∀ k∈[i+1..j]. P k i ≤ k k ≤ j
  then show P k by (cases k = i) simp-all
qed
show ?thesis by (auto simp add: all-interval-int-def intro: *)
qed

lemma list-all-iff-all-interval-int [code-unfold]:
list-all P [i..j] ←→ all-interval-int P i j
by (simp add: list-all-iff all-interval-int-def)

lemma list-ex-iff-not-all-interval-int [code-unfold]:
list-ex P [i..j] ←→ ¬(all-interval-int (Not ◦ P) i j)
by (simp add: list-ex-iff all-interval-int-def)

optimized code (tail-recursive) for length
definition gen-length :: nat ⇒ 'a list ⇒ nat
where gen-length n xs = n + length xs

lemma gen-length-code [code]:
gen-length n [] = n
gen-length n (x # xs) = gen-length (Suc n) xs
by(simp-all add: gen-length-def)

declare list.size(3-4)[code del]

lemma length-code [code]: length = gen-length 0
by(simp add: gen-length-def fun-eq-iff)

hide-const (open) member null maps map-filter all-interval-nat all-interval-int

67.7.3 Pretty lists

ML ""
(* Code generation for list literals. *)

signature LIST-CODE =
sig val implode-list: Code-Thingol.iterm ⇒ Code-Thingol.iterm list option
  val default-list: int * string
THEORY “List”

→ (Code-Printer.fixity → Code-Thingol.ìterm → Pretty.T)
→ Code-Printer.fixity → Code-Thingol.ìterm → Code-Thingol.ìterm →
Pretty.T
val add-literal-list: string → theory → theory
end;
structure List-Code : LIST-CODE =

open Basic-Code-Thingol;

fun implode-list t =
  let
    fun dest-cons (IConst { sym = Code-Symbol.Constant @ (const-name Cons),
      ... } $ t1 $ t2) = SOME (t1, t2)
    | dest-cons - = NONE;
    val (ts, t') = Code-Thingol.unfoldr dest-cons t;
    in case t' of IConst { sym = Code-Symbol.Constant @ (const-name Nil), ... } => SOME ts
    | - => NONE
  end;

fun default-list (target-fxy, target-cons) pr fxy t1 t2 =
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy (pr (Code-Printer.INFX (target-fxy, Code-Printer.X)) t1,
  Code-Printer.str target-cons,
  pr (Code-Printer.INFX (target-fxy, Code-Printer.R)) t2);

fun add-literal-list target =
  let
    fun pretty literals pr - vars fxy [[(t1, -), (t2, -)]] =
      case Option.map (cons t1) (implode-list t2)
      of SOME ts =>
        Code-Printer.literal-list literals (map (pr vars Code-Printer.NOBR) ts)
      | NONE =>
        default-list (Code-Printer.infix-cons literals) (pr vars) fxy t1 t2;
    in
      Code-Target.set-printings (Code-Symbol.Constant @ (const-name Cons),
      [(target, SOME (Code-Printer.complex-const-syntax (2, pretty)))]))
    end
  end;

end

code-printing
type-constructor list →
  (SML) - list
THEORY “List”

and (OCaml) - list
and (Haskell) ![()]
and (Scala) List[(]

| constant Nil →
  (SML) []
  and (OCaml) []
  and (Haskell) []
  and (Scala) Nil
| class-instance list :: equal →
  (Haskell)
| constant HOL.equal :: 'a list ⇒ 'a list ⇒ bool →
  (Haskell) infix 4 ==

setup ⟨⟨ fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala] ⟩⟩

code-reserved SML
list

code-reserved OCaml
list

67.7.4 Use convenient predefined operations

code-printing
constant op @ →
  (SML) infixr 7 @
  and (OCaml) infixr 6 @
  and (Haskell) infixr 5 ++
  and (Scala) infixl 7 ++
| constant map →
  (Haskell) map
| constant filter →
  (Haskell) filter
| constant concat →
  (Haskell) concat
| constant List.maps →
  (Haskell) concatMap
| constant rev →
  (Haskell) reverse
| constant zip →
  (Haskell) zip
| constant List.null →
  (Haskell) null
| constant takeWhile →
  (Haskell) takeWhile
| constant dropWhile →
  (Haskell) dropWhile
| constant list-all →
  (Haskell) all
constant list-ex \rightarrow 
(Haskell) any

67.7.5 Implementation of sets by lists

lemma is-empty-set [code]:
Set.is-empty (set xs) \leftrightarrow List.null xs
by (simp add: Set.is-empty-def null-def)

lemma empty-set [code]:
\{\} = set []
by simp

lemma UNIV-coset [code]:
UNIV = List.coset []
by simp

lemma compl-set [code]:
- set xs = List.coset xs
by simp

lemma compl-coset [code]:
- List.coset xs = set xs
by simp

lemma [code]:
x \in set xs \leftrightarrow List.member xs x
x \in List.coset xs \leftrightarrow \neg List.member xs x
by (simp-all add: member-def)

lemma insert-code [code]:
insert x (set xs) = set (List.insert x xs)
insert x (List.coset xs) = List.coset (removeAll x xs)
by simp-all

lemma remove-code [code]:
Set.remove x (set xs) = set (removeAll x xs)
Set.remove x (List.coset xs) = List.coset (List.insert x xs)
by (simp-all add: remove-def Compl-insert)

lemma filter-set [code]:
Set.filter P (set xs) = set (filter P xs)
by auto

lemma image-set [code]:
image f (set xs) = set (map f xs)
by simp

lemma subset-code [code]:
set \(xs \subseteq B \iff (\forall x \in set \, xs. \, x \in B)\)
\(A \leq \text{List.coset} \, ys \iff (\forall y \in set \, ys. \, y \notin A)\)
\(\text{List.coset} \, [] \leq \text{set} \, [] \iff \text{False}\)
by auto

A frequent case – avoid intermediate sets

**lemma** [code-unfold]:
\(set \, xs \subseteq \text{set} \, ys \iff \text{list-all} \, (\lambda x. \, x \in \text{set} \, ys) \, xs\)
by (auto simp: list-all-iff)

**lemma** Ball-set [code]:
\(\text{Ball} \, (set \, xs) \, P \iff \text{list-all} \, P \, xs\)
by (simp add: list-all-iff)

**lemma** Bex-set [code]:
\(\text{Bex} \, (set \, xs) \, P \iff \text{list-ex} \, P \, xs\)
by (simp add: list-ex-iff)

**lemma** card-set [code]:
\(\text{card} \, (set \, xs) = \text{length} \, (\text{remdups} \, xs)\)

**proof** –
have \(\text{card} \, (set \, (\text{remdups} \, xs)) = \text{length} \, (\text{remdups} \, xs)\)
by (rule distinct-card) simp
then show ?thesis by simp
qed

**lemma** the-elem-set [code]:
\(\text{the-elem} \, (set \, [x]) = x\)
by simp

**lemma** Pow-set [code]:
\(\text{Pow} \, (set \, []) = \{\{\}\}\)
\(\text{Pow} \, (set \, (x \# \, xs)) = (\text{let} \, A = \text{Pow} \, (set \, xs) \, \text{in} \, A \cup \text{insert} \, x \, A)\)
by (simp-all add: Pow-insert Let-def)

**lemma** setsum-code [code]:
\(\text{setsum} \, f \, (set \, xs) = \text{listsum} \, (\text{map} \, f \, (\text{remdups} \, xs))\)
by (simp add: listsum-distinct-conv-setsum-set)

**definition** map-project :: ('a ⇒ 'b option) ⇒ 'a set ⇒ 'b set
where
\(\text{map-project} \, f \, A = \{b. \, \exists \, a \in A. \, f \, a = \text{Some} \, b\}\)

**lemma** [code]:
\(\text{map-project} \, f \, (set \, xs) = \text{set} \, (\text{List.map-filter} \, f \, xs)\)
by (auto simp add: map-project-def map-filter-def image-def)

**hide-const** (open) map-project

Operations on relations
lemma product-code [code]:
\[ \text{Product-Type.product (set } xs \text{) (set } ys \text{) } = \text{ set } [(x, y). x \leftarrow xs, y \leftarrow ys] \]
by (auto simp add: Product-Type.product-def)

lemma Id-on-set [code]:
\[ \text{Id-on (set } xs \text{) } = \text{ set } [(x, x). x \leftarrow xs] \]
by (auto simp add: Id-on-def)

lemma [code]:
\[ R S = \text{List.map-project } \%(x, y). \text{if } x : S \text{ then Some } y \text{ else None} \) R \]
unfolding map-project-def by (auto split: prod.split split-if-asm)

lemma trancl-set-ntrancl [code]:
\[ \text{trancl (set } xs \text{) } = \text{ntrancl (card (set } xs \text{) } - 1 \) (set } xs \text{)} \]
by (simp add: finite-trancl-ntranl)

lemma set-relcomp [code]:
\[ \text{set } xys \ O \text{ set } yzs \ = \text{ set } \{((\text{fst } xy, \text{snd } yz). xy \leftarrow xys, yz \leftarrow yzs, \text{snd } xy = \text{fst } yz)\} \]
by (auto simp add: Bex-def)

lemma wf-set [code]:
\[ \text{wf (set } xs \text{) } = \text{acyclic (set } xs \text{)} \]
by (simp add: wf-iff-acyclic-if-finite)

67.8 Setup for Lifting/Transfer

67.8.1 Transfer rules for the Transfer package

context
begin
interpretation lifting-syntax .

lemma Nil-transfer [transfer-rule]: (list-all2 A [][])
by simp

lemma Cons-transfer [transfer-rule]:
\[ (A \Longrightarrow list-all2 A \Longrightarrow list-all2 A) \Longrightarrow Cons Cons \]
unfolding rel-fun-def by simp

lemma case-list-transfer [transfer-rule]:
\[ (B \Longrightarrow (A \Longrightarrow list-all2 A \Longrightarrow B) \Longrightarrow list-all2 A \Longrightarrow B) \]
unfolding rel-fun-def by (simp split: list.split)

lemma rec-list-transfer [transfer-rule]:
\[ (B \Longrightarrow (A \Longrightarrow list-all2 A \Longrightarrow B) \Longrightarrow list-all2 A \Longrightarrow B) \]
unfolding rel-fun-def by (clarify, erule list-all2-induct, simp-all)
lemma tl-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) tl tl
unfolding tl-def[abs-def] by transfer-prover

lemma butlast-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) butlast butlast
by (rule rel-funI, erule list-all2-induct, auto)

lemma set-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) set set
unfolding set-rec[abs-def] by transfer-prover

lemma map-rec: map f xs = rec-list Nil (%x - y. Cons (f x) y) xs
by (induct xs) auto

lemma map-transfer [transfer-rule]:
  ((A ===> B) ===> list-all2 A ===> list-all2 B) map map
unfolding map-rec[abs-def] by transfer-prover

lemma append-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A ===> list-all2 A) append append
unfolding List.append-def by transfer-prover

lemma rev-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) rev rev
unfolding List.rev-def by transfer-prover

lemma filter-transfer [transfer-rule]:
  ((A ===> op =) ===> list-all2 A ===> list-all2 A) filter filter
unfolding List.filter-def by transfer-prover

lemma fold-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) fold fold
unfolding List.fold-def by transfer-prover

lemma foldr-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) foldr foldr
unfolding List.foldr-def by transfer-prover

lemma foldl-transfer [transfer-rule]:
  ((B ===> A ===> B) ===> list-all2 A ===> B) foldl foldl
unfolding List.foldl-def by transfer-prover

lemma concat-transfer [transfer-rule]:
  (list-all2 (list-all2 A) ===> list-all2 A) concat concat
unfolding List.concat-def by transfer-prover

lemma drop-transfer [transfer-rule]:
  (op ===> list-all2 A ===> list-all2 A) drop drop
unfolding \textit{List.drop-def} by \textit{transfer-prover}

\textbf{lemma} \textit{take-transfer} [\textit{transfer-rule}];
\begin{align*}
& (\text{op} = \text{==} > \text{list-all2} \ A \text{==} > \text{list-all2} \ A) \ \text{take} \text{take} \\
\end{align*}

unfolding \textit{List.take-def} by \textit{transfer-prover}

\textbf{lemma} \textit{list-update-transfer} [\textit{transfer-rule}];
\begin{align*}
& (\text{list-all2} \ A \text{==} > \text{op} = \text{==} > \text{A} \text{==} > \text{list-all2} \ A) \ \text{list-update} \text{list-update} \\
\end{align*}

unfolding \textit{list-update-def} by \textit{transfer-prover}

\textbf{lemma} \textit{takeWhile-transfer} [\textit{transfer-rule}];
\begin{align*}
& ((\text{A} = \text{==} > \text{op} =) = \text{==} > \text{list-all2} \ A = \text{==} > \text{list-all2} \ A) \ \text{takeWhile} \text{takeWhile} \\
\end{align*}

unfolding \textit{takeWhile-def} by \textit{transfer-prover}

\textbf{lemma} \textit{dropWhile-transfer} [\textit{transfer-rule}];
\begin{align*}
& ((\text{A} = \text{==} > \text{op} =) = \text{==} > \text{list-all2} \ A = \text{==} > \text{list-all2} \ A) \ \text{dropWhile} \text{dropWhile} \\
\end{align*}

unfolding \textit{dropWhile-def} by \textit{transfer-prover}

\textbf{lemma} \textit{zip-transfer} [\textit{transfer-rule}];
\begin{align*}
& (\text{list-all2} \ A = \text{==} > \text{list-all2} \ B = \text{==} > \text{list-all2} \ (\text{rel-prod} \ A \ B)) \ \text{zip} \text{zip} \\
\end{align*}

unfolding \textit{zip-def} by \textit{transfer-prover}

\textbf{lemma} \textit{product-transfer} [\textit{transfer-rule}];
\begin{align*}
& (\text{list-all2} \ A = \text{==} > \text{list-all2} \ B = \text{==} > \text{list-all2} \ (\text{rel-prod} \ A \ B)) \ \text{List.product} \text{List.product} \\
\end{align*}

unfolding \textit{List.product-def} by \textit{transfer-prover}

\textbf{lemma} \textit{product-lists-transfer} [\textit{transfer-rule}];
\begin{align*}
& (\text{list-all2} \ (\text{list-all2} \ A) = \text{==} > \text{list-all2} \ (\text{list-all2} \ A)) \ \text{product-lists} \text{product-lists} \\
\end{align*}

unfolding \textit{product-lists-def} by \textit{transfer-prover}

\textbf{lemma} \textit{insert-transfer} [\textit{transfer-rule}];
\begin{align*}
& \text{assumes} [\textit{transfer-rule}]: \text{bi-unique} \ A \\
& \text{shows} (A = \text{==} > \text{list-all2} \ A = \text{==} > \text{list-all2} \ A) \ \text{List.insert} \text{List.insert} \\
& \text{unfolding} \text{List.insert-def} [\textit{abs-def}] \text{by} \textit{transfer-prover} \\
\end{align*}

\textbf{lemma} \textit{find-transfer} [\textit{transfer-rule}];
\begin{align*}
& ((\text{A} = \text{==} > \text{op} =) = \text{==} > \text{list-all2} \ A = \text{==} > \text{rel-option} \ A) \ \text{List.find} \text{List.find} \\
& \text{unfolding} \text{List.find-def} \text{by} \textit{transfer-prover} \\
\end{align*}

\textbf{lemma} \textit{remove1-transfer} [\textit{transfer-rule}];
\begin{align*}
& \text{assumes} [\textit{transfer-rule}]: \text{bi-unique} \ A \\
& \text{shows} (A = \text{==} > \text{list-all2} \ A = \text{==} > \text{list-all2} \ A) \ \text{remove1} \text{remove1} \\
& \text{unfolding} \text{remove1-def} \text{by} \textit{transfer-prover} \\
\end{align*}

\textbf{lemma} \textit{removeAll-transfer} [\textit{transfer-rule}];
\begin{align*}
& \text{assumes} [\textit{transfer-rule}]: \text{bi-unique} \ A \\
& \text{shows} (A = \text{==} > \text{list-all2} \ A = \text{==} > \text{list-all2} \ A) \ \text{removeAll} \text{removeAll} \\
& \text{unfolding} \text{removeAll-def} \text{by} \textit{transfer-prover} \\
\end{align*}
lemma distinct-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> op =) distinct distinct
  unfolding distinct-def by transfer-prover

lemma remdups-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A) remdups remdups
  unfolding remdups-def by transfer-prover

lemma remdups-adj-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A) remdups-adj remdups-adj
  proof (rule rel-funI, erule list-all2-induct)
  qed (auto simp: remdups-adj-Cons assms [unfolded bi-unique-def] split: list.splits)

lemma replicate-transfer [transfer-rule]:
  (op = ===> A ===> list-all2 A) replicate replicate
  unfolding replicate-def by transfer-prover

lemma length-transfer [transfer-rule]:
  (list-all2 A ===> op =) length length
  unfolding size-list-overloaded-def size-list-def by transfer-prover

lemma rotate1-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) rotate1 rotate1
  unfolding rotate1-def by transfer-prover

lemma rotate-transfer [transfer-rule]:
  (op = ===> list-all2 A ===> list-all2 A) rotate rotate
  unfolding rotate-def [abs-def] by transfer-prover

lemma list-all2-transfer [transfer-rule]:
  ((A ===> B ===> op =) ===> list-all2 A ===> list-all2 B ===> op =)
  list-all2 list-all2
  apply (subst 4) list-all2-iff [abs-def])
  apply (subst 3) list-all2-iff [abs-def])
  apply transfer-prover
  done

lemma sublist-transfer [transfer-rule]:
  (list-all2 A ===> rel-set (op =) ===> list-all2 A) sublist sublist
  unfolding sublist-def [abs-def] by transfer-prover

lemma partition-transfer [transfer-rule]:
  ((A ===> op =) ===> list-all2 A ===> rel-prod (list-all2 A) (list-all2 A))
  partition
  unfolding partition-def by transfer-prover
lemma lists-transfer [transfer-rule]:
(rel-set A ===> rel-set (list-all2 A)) lists lists
apply (rule rel-funI, rule rel-setI)
apply (erule lists.induct, simp)
apply (simp only: rel-set-def list-all2.Cons1, metis lists.Cons)
apply (erule lists.induct, simp)
apply (simp only: rel-set-def list-all2.Cons2, metis lists.Cons)
done

lemma set-Cons-transfer [transfer-rule]:
(rel-set A ===> rel-set (list-all2 A) ===> rel-set (list-all2 A)) set-Cons set-Cons
unfolding rel-fun-def rel-set-def set-Cons-def
apply safe
apply (simp add: list-all2-Cons1, fast)
apply (simp add: list-all2-Cons2, fast)
done

lemma listset-transfer [transfer-rule]:
(list-all2 (rel-set A) ===> rel-set (list-all2 A)) listset listset
unfolding listset-def [abs-def] by transfer-prover

lemma null-transfer [transfer-rule]:
(list-all2 A ===> op =) List.null List.null
unfolding rel-fun-def List.null-def by auto

lemma list-all-transfer [transfer-rule]:
((A ===> op =) ===> list-all2 A ===> op =) list-all list-all
unfolding list-all-iff [abs-def] by transfer-prover

lemma list-ex-transfer [transfer-rule]:
((A ===> op =) ===> list-all2 A ===> op =) list-ex list-ex
unfolding list-ex-iff [abs-def] by transfer-prover

lemma splice-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A ===> list-all2 A) splice splice
apply (rule rel-funI, erule list-all2-induct, simp add: rel-fun-def, simp)
apply (rule rel-funI)
apply (erule-tac xs=x in list-all2-induct, simp, simp add: rel-fun-def)
done

lemma listsum-transfer[transfer-rule]:
assumes [transfer-rule]: A 0 0
assumes [transfer-rule]: (A ===> A ===> A) op + op +
shows (list-all2 A ===> A) listsum listsum
unfolding listsum-def [abs-def]
by transfer-prover

lemma rtrancl-parametric [transfer-rule]:
assumes [transfer-rule]: bi-unique A bi-total A
shows (rel-set (rel-prod A A) ===> rel-set (rel-prod A A)) rtrancl rtrancl
unfolding rtrancl-def by transfer-prover

end

end

68 Random: A HOL random engine

theory Random
imports List
begin

notation fcomp (infixl ◦ 60)
notation scomp (infixl ◦→ 60)

68.1 Auxiliary functions
fun log :: natural ⇒ natural ⇒ natural where
log b i = (if b ≤ 1 ∨ i < b then 1 else 1 + log b (i div b))
definition inc-shift :: natural ⇒ natural ⇒ natural where
inc-shift v k = (if v = k then 1 else k + 1)
definition minus-shift :: natural ⇒ natural ⇒ natural ⇒ natural where
minus-shift r k l = (if k < l then r + k − l else k − l)

68.2 Random seeds
type-synonym seed = natural × natural
primrec next :: seed ⇒ natural × seed where
next (v, w) = (let
   k = v div 53668;
   v' = minus-shift 2147483563 ((v mod 53668) * 40014) (k * 12211);
   l = w div 52774;
   w' = minus-shift 2147483399 ((w mod 52774) * 40692) (l * 3791);
   z = minus-shift 2147483562 v' (w' + 1) + 1
   in (z, (v', w')))
definition split-seed :: seed ⇒ seed × seed where
split-seed s = (let
   (v, w) = s;
   (v', w') = snd (next s);
   v'' = inc-shift 2147483562 v;
   w'' = inc-shift 2147483398 w
   in ((v'', w'), (v', w'')))
68.3 Base selectors

fun iterate :: \texttt{natural} \Rightarrow (\texttt{'}b \Rightarrow \texttt{'}a \Rightarrow \texttt{'}b 	imes \texttt{'}a) \Rightarrow \texttt{'}b \Rightarrow \texttt{'}a \Rightarrow \texttt{'}b 	imes \texttt{'}a) where
iterate \ k \ f \ x = (\text{if} \ k = 0 \ \text{then} \ \text{Pair} \ x \ \text{else} \ f \ x \ o\rightarrow \ \text{iterate} \ (k - 1) \ f)

definition range :: \texttt{natural} \Rightarrow \texttt{seed} \Rightarrow \texttt{natural} \times \texttt{seed} where
range \ k = \text{iterate} \ (\log 2147483561 \ k)
(\lambda \ l. \ \text{next} \ o\rightarrow (\lambda \ v. \ \text{Pair} \ (v + l \times 2147483561))) \ 1
o\rightarrow (\lambda \ v. \ \text{Pair} \ (v \ \text{mod} \ k))

lemma range:
\( k > 0 \Rightarrow \text{fst} \ (\text{range} \ k \ s) < k \)
by (simp add: \text{range-def} \ \text{split-def} \ \text{less-natural-def} \ \text{del: log.simps iterate.simps})

definition select :: \texttt{'}a \ \texttt{list} \Rightarrow \texttt{seed} \Rightarrow \texttt{'}a \times \texttt{seed} where
select \ xs = range \ (\text{natural-of-nat} \ (\text{length} \ xs))
o\rightarrow (\lambda \ k. \ \text{Pair} \ (\text{nth} \ xs \ (\text{nat-of-natural} \ k)))

lemma select:
assumes \ xs \ \neq \ []
shows \ \text{fst} \ (\text{select} \ xs \ s) \in \ \text{set} \ xs
proof –
from \ \text{assms} \ \text{have} \ \text{natural-of-nat} \ (\text{length} \ xs) \ > \ 0 \ \text{by} \ (\text{simp add: \text{less-natural-def}})
with \ \text{range} \ \text{have} \ \text{fst} \ (\text{range} \ (\text{natural-of-nat} \ (\text{length} \ xs))) \ s < \ \text{natural-of-nat} \ (\text{length} \ xs) \ \text{by \ best}
then \ \text{have} \ \text{nat-of-natural} \ (\text{fst} \ (\text{range} \ (\text{natural-of-nat} \ (\text{length} \ xs))) \ s) < \ \text{length} \ xs \ \text{by} \ (\text{simp add: \text{less-natural-def}})
then \ \text{show} \ \text{thesis}
by (simp add: \text{simp add: \text{split-beta select-def}})

qed

primrec pick :: (\texttt{natural} \times \texttt{'}a) \ \texttt{list} \Rightarrow \texttt{natural} \Rightarrow \texttt{'}a where
pick \ (x \ # \ xs) \ i = (\text{if} \ i < \ \text{fst} \ x \ \text{then} \ \text{snd} \ x \ \text{else} \ \text{pick} \ xs \ (i - \ \text{fst} \ x))

lemma pick-member:
i < \ \text{listsum} \ (\text{map} \ \text{fst} \ xs) \\implies \ \text{pick} \ xs \ i \in \ \text{set} \ (\text{map} \ \text{snd} \ xs)
by (induct \ xs \ \text{arbitrary:} \ i) \ (\text{simp-all add: \text{less-natural-def}})

lemma pick-drop-zero:
pick \ (\text{filter} \ (\lambda \ (k, -). \ k > 0) \ xs) = \text{pick} \ xs
by (induct \ xs) \ (\text{auto simp add: \text{fun-eq-iff less-natural-def minus-natural-def}})

lemma pick-same:
l < \ \text{length} \ xs \\implies \ \text{Random.pick} \ (\text{map} \ \text{Pair} \ 1) \ xs \ (\text{natural-of-nat} \ l) = \text{nth} \ xs \ l
proof (induct \ xs \ \text{arbitrary:} \ l)
case \ Nil \ \text{then show} \ ?\text{case by simp}
next
case \ (\text{Cons} \ x \ xs) \ \text{then show} \ ?\text{case by} \ (\text{cases} \ l) \ (\text{simp-all add: \text{less-natural-def}})

qed
definition select-weight :: (natural × 'a) list ⇒ seed ⇒ 'a × seed where
select-weight xs = range (listsum (map fst xs))
  o→ (λk. Pair (pick xs k))

lemma select-weight-member:
assumes 0 < listsum (map fst xs)
shows fst (select-weight xs s) ∈ set (map snd xs)
proof
  from range assms
  have fst (range (listsum (map fst xs))) s < listsum (map fst xs).
  with pick-member
  have pick xs (fst (range (listsum (map fst xs)))) s) ∈ set (map snd xs).
  then show thesis by (simp add: select-weight-def scomp-def split-def)
qed

lemma select-weight-cons-zero:
select-weight ((0, x) # xs) = select-weight xs
by (simp add: select-weight-def less-natural-def)

lemma select-weight-drop-zero:
select-weight (filter (λ(k, -). k > 0) xs) = select-weight xs
proof
  have listsum (map fst [(k, -)← xs. 0 < k]) = listsum (map fst xs)
    by (induct xs) simp-all
  then show thesis by (simp only: select-weight-def pick-drop-zero)
qed

lemma select-weight-select:
assumes xs ≠ []
shows select-weight (map (Pair 1) xs) = select xs
proof
  have less: ∀s. fst (range (natural-of-nat (length xs)) s) < natural-of-nat (length xs)
    using assms by (intro range) simp add: less-natural-def
  moreover have listsum (map fst (map (Pair 1) xs)) = natural-of-nat (length xs)
    by (induct xs) simp-all
  ultimately show thesis
    by (auto simp add: select-weight-def select-def scomp-def split-def
        fun-eq-iff pick-same [symmetric] less-natural-def)
qed

68.4  ML interface

code-reflect Random-Engine
functions range select select-weight

ML ¶
structure Random-Engine =

struct

open Random-Engine;

type seed = Code-Numeral.natural * Code-Numeral.natural;

local

val seed = Unsynchronized.ref

(let
val now = Time.toMilliseconds (Time.now ());
val (q, s1) = IntInf.divMod (now, 2147483562);
val s2 = q mod 2147483398;
in pairself Code-Numeral.natural-of-integer (s1 + 1, s2 + 1) end);

in

fun next-seed () =

(let
val (seed1, seed') = @{code split-seed} (! seed)
val _ = seed := seed'
in
seed1
end

fun run f =

(let
val (x, seed') = f (! seed);
val _ = seed := seed'
in x end;
end;
end;

))

hide-type (open) seed
hide-const (open) inc-shift minus-shift log next split-seed
iterate range select pick select-weight
hide-fact (open) range-def

no-notation fcomp (infixl ◦> 60)
no-notation scomp (infixl ◦→ 60)

end
69 Map: Maps

theory Map
imports List
begin

  type-synonym ('a,'b) map = 'a => 'b option (infixr "=>")

  type-notation (xsymbols)
  map (infixr "\rightarrow")

  abbreviation (xsymbols)
  empty :: 'a => 'b where
  empty == %x. None

  definition map-comp :: ('b => 'c) => ('a => 'b) => ('a => 'c) (infixl o' 55)
  where
  f o-m g = (\lambda k. case g k of None => None | Some v => f v)

  notation (xsymbols)
  map-comp (infixl \circ 55)

  definition map-add :: ('a => 'b) => ('a => 'b) => ('a => 'b) (infixl ++ 100)
  where
  m1 ++ m2 = (\lambda x. case m2 x of None => m1 x | Some y => Some y)

  definition restrict-map :: ('a => 'b) => 'a set => ('a => 'b) (infixl |\cdot 110)
  where
  m|\cdot A = (\lambda x. if x : A then m x else None)

  notation (latex output)
  restrict-map (\cdot | [111,110] 110)

  definition
  dom :: ('a => 'b) => 'a set where
  dom m = {a. m a => None}

  definition
  ran :: ('a => 'b) => 'b set where
  ran m = {b. \exists a. m a = Some b}

  definition
  map-le :: ('a => 'b) => ('a => 'b) => bool (infix \subseteq_m 50)
  where
  (m1 \subseteq_m m2) = (\forall a \in dom m1. m1 a = m2 a)

  nonterminal maplets and maplet
syntax
-\texttt{maplet} \::= \texttt{maplet} (\texttt{/} \rightarrow \texttt{/})
-\texttt{maplets} \::= \texttt{maplet} (\texttt{/}[\rightarrow] \rightarrow)
-\texttt{Maplets} \::= \texttt{maplet}, \texttt{maplets} (\rightarrow)

-\texttt{MapUpd} \::= \texttt{a} \rightarrow \texttt{b}, \texttt{maplets} \rightarrow (\rightarrow) [900,0][900]

translations
-\texttt{MapUpd} \::= \texttt{a} \rightarrow \texttt{b}, \texttt{maplets} \rightarrow (\rightarrow) ((1[\cdot]))

primrec
\texttt{map-of} \::= (\texttt{a} \times \texttt{b}) \texttt{list} \Rightarrow \texttt{a} \rightarrow \texttt{b} \texttt{where}
\begin{align*}
\texttt{map-of} \ [\] &= \texttt{empty} \\
\texttt{map-of} \ [p \# ps] &= (\texttt{map-of} ps)(\texttt{fst} p \mapsto \texttt{snd} p)
\end{align*}

definition
\texttt{map-upds} \::= (\texttt{a} \rightarrow \texttt{b}) \texttt{list} \Rightarrow \texttt{a} \texttt{list} \Rightarrow \texttt{b} \texttt{list} \Rightarrow \texttt{a} \rightarrow \texttt{b} \texttt{where}
\begin{align*}
\texttt{map-upds} \ [xs \ ys] &= \texttt{m} \mapsto \texttt{map-of} (\texttt{rev} (\texttt{zip} \texttt{xs} \texttt{ys}))
\end{align*}

translations
-\texttt{MapUpd} \ [\texttt{maplets} \ x \ y] \mapsto \texttt{CONST} \texttt{map-upds} \ [\texttt{m} \ x \ y]

lemma \texttt{map-of-Cons-code} \ [\texttt{code}]:
\begin{align*}
\texttt{map-of} \ [l] &= \texttt{None} \\
\texttt{map-of} \ [(l, v) \# ps] &= (\texttt{if} \ l = \texttt{k} \ \texttt{then} \ \texttt{Some} \ v \ \texttt{else} \ \texttt{map-of} \ ps \ k)
\end{align*}
by \texttt{simp-all}

69.1 \texttt{empty}

lemma \texttt{empty-upd-none} \ [\texttt{simp}]: \texttt{empty}(x := \texttt{None}) = \texttt{empty}
by \texttt{(rule ext) simp}

69.2 \texttt{map-upd}

lemma \texttt{map-upd-triv}: \texttt{t} \texttt{k} = \texttt{Some} \ x \mapsto \texttt{t(k)} \mapsto \texttt{x} = \texttt{t}
by \texttt{(rule ext) simp}

lemma \texttt{map-upd-nonempty} \ [\texttt{simp}]: \texttt{t(k)} \mapsto \texttt{x} = \texttt{empty}
proof
assume \texttt{t(k \mapsto x)} = \texttt{empty}
then \texttt{have} \texttt{t(k \mapsto x)} \mapsto \texttt{k} = \texttt{None} by \texttt{simp}
then show False by simp
qed

lemma map-upd-eqD1:
  assumes \( m(a \mapsto x) = n(a \mapsto y) \)
  shows \( x = y \)
proof -
  from assms have \( (m(a \mapsto x)) \ a \ = \ (n(a \mapsto y)) \ a \) by simp
  then show \( ?thesis \) by simp
qed

lemma map-upd-Some-unfold:
  \((m(a \mapsto b)) \ x \ = \ Some \ y) = (x = a \land b = y \lor x \neq a \land m \ x = Some \ y)\)
by auto

lemma image-map-upd [simp]: \( x \notin A \implies m(x \mapsto y) \ A = m \ A \)
by auto

lemma finite-range-updI:
  finite \( (range \ f) \) ==>
  finite \( (range \ (f(a \mapsto b))) \)
unfolding image-def
apply (simp (no-asn-use) add:full-SetCompr-eq)
apply (rule finite-subset)
prefer 2 apply assumption
apply (auto)
done

69.3 map-of

lemma map-of-eq-None-iff:
  \( (map \ of \ xys \ x = None) = (x \notin \text{fst} \ (\text{set} \ xys)) \)
by (induct xys) simp-all

lemma map-of-is-SomeD:
  map-of \ xys \ x = Some \ y \implies (x,y) \in \text{set} \ xys
apply (induct xys)
apply simp
apply (clarsimp split: if-splits)
done

lemma map-of-eq-Some-iff [simp]:
  \( \text{distinct}(map \ \text{fst} \ xys) \implies (map-of \ xys \ x = Some \ y) = ((x,y) \in \text{set} \ xys) \)
apply (induct xys)
apply simp
apply (auto simp: map-of-eq-None-iff [symmetric])
done

lemma Some-eq-map-of-iff [simp]:
  \( \text{distinct}(map \ \text{fst} \ xys) \implies (Some \ y = map-of \ xys \ x) = ((x,y) \in \text{set} \ xys) \)
by (auto simp del:map-of-eq-Some-iff simp add: map-of-eq-Some-iff [symmetric])
lemma map-of-is-SomeI [simp]: \[ \begin{align*} & \text{distinct(map fst xys); } (x,y) \in \text{set xys} ] \\
& \Longrightarrow \text{map-of xys x = Some y} \\
\end{align*} \]
apply (induct xys)
apply simp
apply force
done

lemma map-of-zip-is-None [simp]: 
\[ \begin{align*} & \text{length xs = length ys } \Longrightarrow \text{(map-of (zip xs ys) x = None) } = (x \notin \text{set xs}) \\
\end{align*} \]
by (induct rule: list-induct2) simp-all

lemma map-of-zip-is-Some: 
assumes \[ \begin{align*} & \text{length xs } = \text{length ys} \\
\end{align*} \]
shows \[ \begin{align*} & x \in \text{set xs } \longleftrightarrow (\exists y. \text{map-of (zip xs ys) x } = \text{Some y}) \\
\end{align*} \]
using assms
by (induct rule: list-induct2) simp-all

lemma map-of-zip-upd: 
fixes \[ \begin{align*} & x :: 'a and xs :: 'a list \text{ and } ys \text{ and } zs :: 'b list \\
\end{align*} \]
assumes \[ \begin{align*} & \text{length ys } = \text{length xs} \\
& \text{and length zs } = \text{length xs} \\
& \text{and } x \notin \text{set xs} \\
& \text{and map-of (zip xs ys)(x } \mapsto y) = \text{map-of (zip xs zs)(x } \mapsto z) \\
\end{align*} \]
shows \[ \begin{align*} & \text{map-of (zip xs ys) } = \text{map-of (zip xs zs)} \\
\end{align*} \]
proof
fix \[ \begin{align*} & x' :: 'a \\
\end{align*} \]
show \[ \begin{align*} & \text{map-of (zip xs ys) x' } = \text{map-of (zip xs zs) x'} \\
\end{align*} \]
proof (cases \[ \begin{align*} & x = x' \\
\end{align*} \])
  case True
  from assms True map-of-zip-is-None \[ \begin{align*} & \text{of xs ys x'} \\
\end{align*} \]
  have \[ \begin{align*} & \text{map-of (zip xs ys) x' } = \text{None} \text{ by simp} \\
\end{align*} \]
  moreover from assms True map-of-zip-is-None \[ \begin{align*} & \text{of xs zs x'} \\
\end{align*} \]
  have \[ \begin{align*} & \text{map-of (zip xs zs) x' } = \text{None} \text{ by simp} \\
\end{align*} \]
  ultimately show \[ \begin{align*} & \text{thesis by simp} \\
\end{align*} \]
next
  case False from assms
  have \[ \begin{align*} & \text{(map-of (zip xs ys)(x } \mapsto y)) x' = \text{(map-of (zip xs zs)(x } \mapsto z)) x'} \text{ by auto} \\
\end{align*} \]
  with False show \[ \begin{align*} & \text{thesis by simp} \\
\end{align*} \]
qed

lemma map-of-zip-inject: 
assumes \[ \begin{align*} & \text{length ys } = \text{length xs} \\
& \text{and length zs } = \text{length xs} \\
& \text{and dist: distinct xs} \\
& \text{and map-of: map-of (zip xs ys) } = \text{map-of (zip xs zs)} \\
\end{align*} \]
shows \[ \begin{align*} & ys = zs \\
\end{align*} \]
using assms(1) assms(2)[symmetric] using dist map-of proof (induct ys xs zs rule: list-induct3)
case Nil show ?case by simp

next
case (Cons y ys x xs z zs)
from (map-of (zip (x#xs) (y#ys))) = map-of (zip xs zs) by simp
have map-of: map-of (zip xs ys)(x ↦ y) = map-of (zip xs zs) by simp
from Cons have length ys = length xs and length zs = length xs
and x ∉ set xs by simp-all
then have map-of: map-of (zip xs ys) = map-of (zip xs zs) using map-of by (rule map-of-zip-upd)
with Cons.hyps (distinct (x # xs))
moreover from map-of have y = z by (rule map-upd-eqD1)
ultimately show ?case by simp
qed

lemma map-of-zip-map:
map-of (zip xs (map f xs)) = (λx. if x ∈ set xs then Some (f x) else None)
by (induct xs) (simp-all add: fun-eq-iff)

lemma finite-range-map-of: finite (range (map-of xs))
apply (induct xs)
apply (simp-all add: image-constant)
apply (rule finite-subset)
prefer 2 apply assumption
apply auto
done

lemma map-of-SomeD:
map-of xs k = Some y =⇒ (k, y) ∈ set xs
by (induct xs) (simp, atomize (full), auto)

lemma map-of-mapk-SomeI:
inj f =⇒ map-of t k = Some x =⇒
map-of (map (split (%k. Pair (f k))) t) (f k) = Some x
by (induct t) (auto simp add: inj-eq)

lemma weak-map-of-SomeI: (k, x) : set l =⇒ ∃x. map-of l k = Some x
by (induct l) auto

lemma map-of-filter-in:
map-of xs k = Some z =⇒ P k z =⇒ map-of (filter (split P) xs) k = Some z
by (induct xs) auto

lemma map-of-map:
map-of (map (λ(k, v). (k, f v)) xs) = map-option f ◦ map-of xs
by (induct xs) (auto simp add: fun-eq-iff)

lemma dom-map-option:
dom (λk. map-option (f k) (m k)) = dom m
by (simp add: dom-def)
lemma dom-map-option-comp [simp]:
  dom (map-option g ⪯ m) = dom m
using dom-map-option [of λ-. g m] by (simp add: comp-def)

69.4 map-option related

lemma map-option-o-empty [simp]: map-option f o empty = empty
by (rule ext) simp

lemma map-option-o-map-upd [simp]:
  map-option f o m(a|>b) = (map-option f o m)(a|>f b)
by (rule ext) simp

69.5 map-comp related

lemma map-comp-empty [simp]:
  m ⪯m empty = empty
  empty ⪯m m = empty
by (auto simp add: map-comp-def split: option.splits)

lemma map-comp-simps [simp]:
  m2 k = None ⟹ (m1 ⪯m m2) k = None
  m2 k = Some k' ⟹ (m1 ⪯m m2) k = m1 k'
by (auto simp add: map-comp-def)

lemma map-comp-Some-iff :
  (⋯ k' = Some v) = (∃ k'. m2 k = Some k' ∧ m1 k' = Some v)
by (auto simp add: map-comp-def split: option.splits)

lemma map-comp-None-iff :
  (⋯ k = None) = (m2 k = None ∨ (∃ k'. m2 k = Some k' ∧ m1 k' = None))
by (auto simp add: map-comp-def split: option.splits)

69.6 ++

lemma map-add-empty[simp]: m ++ empty = m
by(simp add: map-add-def)

lemma empty-map-add[simp]: empty ++ m = m
by (rule ext) (simp add: map-add-def split: option.split)

lemma map-add-assoc[simp]: m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3
by (rule ext) (simp add: map-add-def split: option.split)

lemma map-add-Some-iff :
  (⋯ k = Some x) = (n k = Some x ∨ n k = None & m k = Some x)
by (simp add: map-add-def split: option.split)

lemma map-add-SomeD [dest!]:
THEORY "Map"

\[(m ++ n) k = \text{Some } x \implies n k = \text{Some } x \lor n k = \text{None} \land m k = \text{Some } x\]

by (rule map-add-\text{Some-iff} \{\text{THEN } \text{iffD1}\})

\textbf{lemma} map-add-find-right [simp]: \!
\begin{align*}
!\text{xx}. n k &= \text{Some } xx \implies (m ++ n) k = \text{Some } xx \\
\text{by } \text{(subst map-add-\text{Some-iff}) fast}
\end{align*}

\textbf{lemma} map-add-\text{None} [iff]: \!
\begin{align*}
(m ++ n) k &= \text{None} \implies (n k = \text{None} \land m k = \text{None}) \\
\text{by } \text{(simp add: map-add-def split: option.split)}
\end{align*}

\textbf{lemma} map-add-upd[simp]: \!
\begin{align*}
f ++ g(x|\rightarrow y) &= (f ++ g)(x|\rightarrow y) \\
\text{by } \text{(rule ext) (simp add: map-add-def)}
\end{align*}

\textbf{lemma} map-add-upds[simp]: \!
\begin{align*}
\text{m1 ++ (m2(xs|\rightarrow ys))} &= (\text{m1 ++ m2})(xs|\rightarrow ys) \\
\text{by } \text{(simp add: map-upds-def)}
\end{align*}

\textbf{lemma} map-add-upd-left: \!
\begin{align*}
\text{m} \notin \text{dom } e2 \implies e1(m \mapsto u1) ++ e2 &= (e1 ++ e2)(m \mapsto u1) \\
\text{by } \text{(rule ext) (auto simp: map-add-def dom-def split: option.split)}
\end{align*}

\textbf{lemma} map-of-append[simp]: \!
\begin{align*}
\text{map-of } (xs @ ys) &= \text{map-of } ys ++ \text{map-of } xs \\
\text{unfolding map-add-def}
\end{align*}

\textbf{proof}\ \text{(rule sym, rule zip-obtain-same-length)}
\begin{align*}
\text{fix } ks :: \text{a list and } vs :: \text{b list}
\text{assume length } ks = \text{length } vs
\text{then show } \text{foldl } (\lambda m (k, v). m(k \mapsto v)) m (zip ks vs) = m ++ \text{map-of } (rev (zip ks vs))
\text{by (induct arbitrary: m rule: list-induct2 simp-all)}
\end{align*}

done

\textbf{lemma} finite-range-map-of-map-add:
\begin{align*}
\text{finite } (\text{range f}) \implies \text{finite } (\text{range } (f ++ \text{map-of } l)) \\
\text{apply (induct l)}
\end{align*}

\textbf{apply} (auto simp del: fun-upd-apply)

\textbf{apply} (erule finite-range-updI)

done

\textbf{lemma} inj-on-map-add-dom [iff]:
\begin{align*}
\text{inj-on } (m ++ m') (\text{dom } m') &= \text{inj-on } m (\text{dom } m') \\
\text{by } \text{(fastforce simp: map-add-def dom-def inj-on-def split: option.splits)}
\end{align*}

\textbf{lemma} map-upds-fold-map-upd:
\begin{align*}
m(ks|\rightarrow vs) &= \text{foldl } (\lambda m (k, v). m(k \mapsto v)) m (zip ks vs) \\
\text{unfolding map-upds-def proof (rule sym, rule zip-obtain-same-length)}
\end{align*}

\textbf{fix} ks :: \text{a list and vs :: b list}
\textbf{assume} length ks = length vs
\textbf{then show} \text{foldl } (\lambda m (k, v). m(k\mapsto v)) m (zip ks vs) = m ++ \text{map-of } (rev (zip ks vs))
\text{by (induct arbitrary: m rule: list-induct2 simp-all)}

qed
lemma map-add-map-of-foldr:
  \[ m +\map of ps = \text{foldr} \ (\lambda(k, v) \ m. \ m(k \map v)) \ ps \ m \]
  by (induct ps) (auto simp add: fun-eq-iff map-add-def)

69.7 restrict-map

lemma restrict-map-to-empty [simp]: \( m|\{} = \text{empty} \)
  by (simp add: restrict-map-def)

lemma restrict-map-insert: \( f \ |\ (\text{insert} \ a \ A) = (f |\ A)(a := f \ a) \)
  by (auto simp add: restrict-map-def)

lemma restrict-map-empty [simp]: \( \text{empty}|D = \text{empty} \)
  by (simp add: restrict-map-def)

lemma restrict-in [simp]: \( x \in A \implies (m|A) x = m x \)
  by (simp add: restrict-map-def)

lemma restrict-out [simp]: \( x \notin A \implies (m|A) x = \text{None} \)
  by (simp add: restrict-map-def)

lemma ran-restrictD: \( y \in \text{ran}(m|A) \implies \exists x \in A. \ m x = \text{Some} \ y \)
  by (auto simp: restrict-map-def ran-def split: split_if_asm)

lemma dom-restrict [simp]: \( \text{dom}(m|A) = \text{dom} m \cap A \)
  by (auto simp: restrict-map-def dom-def split: split_if_asm)

lemma restrict-upd-same [simp]: \( m(x \map y)|\D = m|\D \)
  by (rule ext) (auto simp: restrict-map-def)

lemma restrict-restrict [simp]: \( m|A|B = m|(A \cap B) \)
  by (rule ext) (auto simp: restrict-map-def)

lemma restrict-fun-upd [simp]:
  \( m(x := y)|\D = (if x \in D then (m|\D \setminus \{x\})(x := y) else m|\D) \)
  by (simp add: restrict-map-def fun-eq_iff)

lemma fun-upd-None-restrict [simp]:
  \( (m|\D)(x := \text{None}) = (if x \in D then m|\D \setminus \{x\}) else m|\D) \)
  by (simp add: restrict-map-def fun-eq_iff)

lemma fun-upd-restrict: \( (m|\D)(x := y) = (m|\D \setminus \{x\})(x := y) \)
  by (simp add: restrict-map-def fun-eq_iff)

lemma fun-upd-restrict-conv [simp]:
  \( x \in D \implies (m|\D)(x := y) = (m|\D \setminus \{x\})(x := y) \)
  by (simp add: restrict-map-def fun-eq_iff)
lemma map-of-map-restrict:
map-of (map (λ k. (k, f k)) ks) = (Some o f) | set ks
by (induct ks) (simp-all add: fun-eq-iff restrict-map-insert)

lemma restrict-complement-singleton-eq:
f | set (- {x}) = f(x := None)
by (simp add: restrict-map-def fun-eq-iff)

69.8 map-upds
lemma map-ups-Nil1 [simp]: m([]=|−> bs) = m
by (simp add: map-upds-def)

lemma map-ups-Nil2 [simp]: m(as [|−>] []) = m
by (simp add: map-upds-def)

lemma map-ups-Cons [simp]: m(a#as [|−>] b#bs) = (m(a|−b))(as|−>bs)
by (simp add: map-upds-def)

lemma map-ups-append1 [simp]: ∀ ys m. size xs < size ys ⇒
m(xs@[x] [|−>] ys) = m(xs [|−>] ys)(x|−> ys!size xs)
apply (induct xs)
apply (clarsimp simp add: neq-Nil-conv)
apply (case-tac ys)
apply simp
apply simp
done

lemma map-ups-list-update2-drop [simp]:
size xs ≤ i ⇒ m(xs[i:=y]) = m(xs [|−>] ys)
apply (induct xs arbitrary: m ys i)
apply simp
apply (case-tac ys)
apply simp
apply (simp split: nat.split)
done

lemma map-upd-upds-conv-if:
(f(x|−>y))(xs [|−>] ys) =
(if x : set(take (length ys) xs) then f(xs [|−>] ys) else f(xs [|−>] ys))(x|−>y)
apply (induct xs arbitrary: x y ys f)
apply simp
apply (case-tac ys)
apply (auto split: split-if simp: fun-upd-twist)
done

lemma map-ups-twist [simp]:
a − set as → m(a|−>b)(as|−>bs) = m(as|−>bs)(a|−>b)
using set-take-subset by (fastforce simp add: map-upd-upds-conv-if)

lemma map-ups-apply-nontin [simp]:
  x ∼: set xs ==⇒ (f(xs[−> ]ys)) x = f x
apply (induct xs arbitrary: ys)
  apply simp
  apply (case-tac ys)
  apply (auto simp: map-upd-upds-conv-if)
done

lemma fun-ups-append-drop [simp]:
  size xs = size ys ==⇒ m(xs@zs[−> ]ys) = m(xs[−> ]ys)
apply (induct xs arbitrary: m ys)
  apply simp
  apply (case-tac ys)
  apply simp-all
done

lemma fun-ups-append2-drop [simp]:
  size xs = size ys ==⇒ m(xs[−> ]ys@zs) = m(xs[−> ]ys)
apply (induct xs arbitrary: m ys)
  apply simp
  apply (case-tac ys)
  apply simp-all
done

lemma restrict-map-ups[simp]:
  [ length xs = length ys; set xs ⊆ D ]
  ==⇒ m(xs [−> ]ys)(D = (m |(D − set xs))(xs [−> ]ys)
apply (induct xs arbitrary: m ys)
  apply simp
  apply (case-tac ys)
  apply simp
  apply (simp add: Diff-insert [symmetric] insert-absorb)
  apply (simp add: map-upd-upds-conv-if)
done

69.9 dom

lemma dom-eq-empty-conv [simp]: dom f = {} ↔ f = empty
  by (auto simp: dom-def)

lemma domI: m a = Some b ==⇒ a : dom m
  by (simp add: dom-def)

lemma domD: a : dom m ==⇒ ∃ b. m a = Some b
  by (cases m a) (auto simp add: dom-def)
lemma `domIff [iff, simp del]: (a : dom m) = (m a =: None)
by(simp add:dom-def)

lemma `dom-empty [simp]: dom empty = {}
by(simp add:dom-def)

lemma `dom-fun-upd [simp]:
dom(f(x := y)) = (if y=:`None then dom f - {x} else insert x (dom f))
by(auto simp add:dom-def)

lemma `dom-if:
dom (λ x. if P x then f x else g x) = dom f ∩ {x. P x} ∪ dom g ∩ {x. ¬ P x}
by (auto split: if-splits)

lemma `dom-map-of-conv-image-fst:
dom (map-of xys) = fst ' set xys
by (induct xys) (auto simp add: dom-if)

lemma `dom-map-of-zip [simp]: length xs = length ys ==> dom (map-of (zip xs ys)) = set xs
by (induct rule: list-induct2) (auto simp add: dom-if)

lemma `finite-dom-map-of: finite (dom (map-of l))
by (induct l) (auto simp add: dom-def insert-Collect [symmetric])

lemma `dom-map-ups [simp]:
dom(m(xs||->ys)) = set(take (length ys) xs) Un dom m
apply (induct xs arbitrary: m ys)
apply simp
apply (case_tac ys)
apply auto
done

lemma `dom-map-add [simp]: dom(m++n) = dom n Un dom m
by(auto simp:dom-def)

lemma `dom-override-on [simp]:
dom(override-on f g A) =
   {dom f - {a. a : A - dom g}} Un {a. a : A Int dom g}
by(auto simp: dom-def override-on-def)

lemma `map-add-comm: dom m1 ∩ dom m2 = {} ==> m1++m2 = m2++m1
by (rule ext) (force simp: map-add-def dom-def split: option.split)

lemma `map-add-dom-app-simps:
  [ m∈dom l2 ] ==> (l1++l2) m = l2 m
  [ m∉dom l1 ] ==> (l1++l2) m = l1 m
  [ m∉dom l2 ] ==> (l1++l2) m = l1 m
lemma dom-const [simp]:
  \( \text{dom} (\lambda x. \text{Some (f x)}) = \text{UNIV} \)
  
  by auto

lemma finite-map-freshness:
  \( \text{finite (dom (f :: 'a \rightarrow 'b))} \implies \neg \text{finite (UNIV :: 'a set)} \implies \exists x. f x = \text{None} \)
  
  by (bestsimp dest:ex-new-if-finite)

lemma dom-minus:
  \( f x = \text{None} \implies \text{dom f} - \text{insert x A} = \text{dom f} - \text{A} \)
  
  unfolding dom-def by simp

lemma insert-dom:
  \( f x = \text{Some y} \implies \text{insert x (dom f)} = \text{dom f} \)
  
  unfolding dom-def by auto

lemma map-of-map-keys:
  \( \text{set xs = dom m} \implies \text{map-of (map (\lambda k. (k, the (m k))) xs)} = m \)
  
  by (rule ext) (auto simp add: map-of-map-restrict restrict-map-def)

lemma map-of-eqI:
  \underline{\text{assumes set-eq: set (map fst xs) = set (map fst ys)}}
  \underline{\text{assumes map-eq: \forall k\in set (map fst xs). map-of xs k = map-of ys k}}
  \underline{\text{shows map-of xs = map-of ys}}
  
  proof (rule ext)
  \( \text{fix k show map-of xs k = map-of ys k} \)
  
  proof (cases map-of xs k)
  \( \text{case None then have k \notin set (map fst xs) by (simp add: map-of-eq-None-iff)} \)
  \( \text{with set-eq have k \notin set (map fst ys) by simp} \)
  \( \text{then have map-of ys k = None by (simp add: map-of-eq-None-iff)} \)
  \( \text{with None show \?thesis by simp} \)
  
  next
  \( \text{case (Some v) then have k \in set (map fst xs) by (auto simp add: map-of-conv-image-fst [symmetric])} \)
  \( \text{with map-eq show \?thesis by auto} \)
  qed
  qed

lemma map-of-eq-dom:
  \underline{\text{assumes map-of xs = map-of ys}}
  \underline{\text{shows fst ' set xs = fst ' set ys}}
  
  proof
  \( \text{from assms have dom (map-of xs) = dom (map-of ys) by simp} \)
  \text{then show \?thesis by (simp add: map-of-conv-image-fst)}
  qed
lemma finite-set-of-finite-maps:
  assumes finite A finite B
  shows finite {m. dom m = A ∧ ran m ⊆ B} (is finite ?S)
  proof
  let ?S’ = {m. ∀x. (x ∈ A → m x ∈ Some ‘ B) ∧ (x /∈ A → m x = None)}
  have ?S = ?S’
  proof
    show ?S ⊆ ?S’ by(auto simp: dom-def ran-def image-def)
    show ?S’ ⊆ ?S
      proof
        fix m assume m ∈ ?S’
        hence 1: dom m = A by force
        hence 2: ran m ⊆ B using ⟨m ∈ ?S’⟩ by(auto simp: dom-def ran-def)
        from 1 2 show m ∈ ?S by blast
      qed
  qed
  with assms show ?thesis by(simp add: finite-set-of-finite-funs)
  qed

69.10  ran

lemma ranI: m a = Some b ==> b : ran m
  by(auto simp: ran-def)

lemma ran-empty [simp]: ran empty = {}
  by(auto simp: ran-def)

lemma ran-map-upd [simp]: m a = None ==> ran(m(a|->b)) = insert b (ran m)
  unfolding ran-def
  apply auto
  apply (subgoal-tac aa ~|= a)
  apply auto
  done

lemma ran-distinct:
  assumes dist: distinct (map fst al)
  shows ran (map-of al) = snd ‘ set al
  using assms proof (induct al)
    case Nil then show ?case by simp
  next
    case (Cons kv al)
    then have ran (map-of al) = snd ‘ set al by simp
    moreover from Cons.prems have map-of al (fst kv) = None
      by (simp add: map-of-eq-None-iff)
    ultimately show ?case by (simp only: map-of.simps ran-map-upd) simp
  qed
69.11 map-le

lemma map-le-empty [simp]: empty ⊆_m g
by (simp add: map-le-def)

lemma upd-None-map-le [simp]: f(x := None) ⊆_m f
by (force simp add: map-le-def)

lemma map-le-upd[simp]: f ⊆_m g ==> f(a := b) ⊆_m g(a := b)
by (fastforce simp add: map-le-def)

lemma map-le-upd-le [simp]: m1 ⊆_m m2 ==> m1(x := None) ⊆_m m2(x ↦→ y)
by (force simp add: map-le-def)

lemma map-le-imp-upd-le [simp]: m1 ⊆_m m2 ==> m1(x := None) ⊆_m m2(x ↦→ y)
by (force simp add: map-le-def)

lemma map-le-upds [simp]: f ⊆_m g ==> f(as ⊢→ bs) ⊆_m g(as ⊢→ bs)
apply (induct as arbitrary: f g bs)
apply simp
apply (case-tac bs)
apply auto
done

lemma map-le-implies-dom-le: (f ⊆_m g) ==> (dom f ⊆ dom g)
by (fastforce simp add: map-le-def dom-def)

lemma map-le-refl [simp]: f ⊆_m f
by (simp add: map-le-def)

lemma map-le-trans [trans]: [ m1 ⊆_m m2; m2 ⊆_m m3 ] ==> m1 ⊆_m m3
by (auto simp add: map-le-def dom-def)

lemma map-le-antisym: [ f ⊆_m g; g ⊆_m f ] ==> f = g
unfolding map-le-def
apply (rule ext)
apply (case-tac x ∈ dom f, simp)
apply (case-tac x ∈ dom g, simp, fastforce)
done

lemma map-le-map-add [simp]: f ⊆_m (g ++ f)
by (fastforce simp add: map-le-def)

lemma map-le-if-map-add-commute: (f ⊆_m f ++ g) = (f++g = g++f)
by (fastforce simp: map-add-def map-le-def fun-eq-iff split: option.splits)

lemma map-add-le-mapE: f++g ⊆_m h ==> g ⊆_m h
by (fastforce simp add: map-le-def map-add-def dom-def)

lemma map-add-le-mapI: [ f ⊆_m h; g ⊆_m h ] ==> f++g ⊆_m h
by (clarsimp simp add: map-le-def map-add-def dom-def split: option.splits)
THEORY “Map”

lemma dom-eq-singleton-conv: \( \text{dom } f = \{ x \} \leftrightarrow (\exists v. f = [x \mapsto v]) \)

proof (rule iffI)
  assume \( \exists v. f = [x \mapsto v] \)
  thus \( \text{dom } f = \{ x \} \) by (auto split: split-if-asm)
next
  assume \( \text{dom } f = \{ x \} \)
  then obtain \( v \) where \( f x = \text{Some } v \) by auto
  moreover have \( f \subseteq_m [x \mapsto v] \) using \( (\text{dom } f = \{ x \}) \)
    by (auto simp add: map-le-def)
  ultimately have \( f = [x \mapsto v] \) by -(rule map-le-antisym)
  thus \( \exists v. f = [x \mapsto v] \) by blast
qed

69.12 Various

lemma set-map-of-compr:
  assumes distinct: distinct \( (\text{map } \text{fst } xs) \)
  shows \( \text{set } xs = \{(k, v). \text{map-of } xs k = \text{Some } v\} \)
using assms proof (induct xs)
next
  case Nil then show \( ?\text{case} \) by simp
next
  case (Cons x xs)
  obtain \( k \) \( v \) where \( x = (k, v) \) by (cases x) blast
  with Cons.prems have \( k \notin \text{dom } (\text{map-of } xs) \)
    by (simp add: dom-map-of-cone-image-fst)
  then have \( *: \text{insert } (k, v) \{(k, v). \text{map-of } xs k = \text{Some } v\} = \{(k', v'). \text{map-of } xs(k \mapsto v)\} k' = \text{Some } v' \)
    by (auto split: if-splits)
  from Cons have \( \text{set } xs = \{(k, v). \text{map-of } xs k = \text{Some } v\} \)
    by simp
  with \( *: (x = (k, v)) \) show \( ?\text{case} \) by simp
qed

lemma map-of-inject-set:
  assumes distinct: distinct \( (\text{map } \text{fst } xs) \) distinct \( (\text{map } \text{fst } ys) \)
  shows \( \text{map-of } xs = \text{map-of } ys 
    \leftrightarrow \text{set } xs = \text{set } ys \) \( (\text{is } \?\text{lhs} \leftrightarrow \?\text{rhs}) \)
proof
  assume \( ?\text{lhs} \)
  moreover from \( \text{distinct } (\text{map } \text{fst } xs) \) have \( \text{set } xs = \{(k, v). \text{map-of } xs k = \text{Some } v\} \)
    by (rule set-map-of-compr)
  moreover from \( \text{distinct } (\text{map } \text{fst } ys) \) have \( \text{set } ys = \{(k, v). \text{map-of } ys k = \text{Some } v\} \)
    by (rule set-map-of-compr)
  ultimately show \( ?\text{rhs} \) by simp
next
  assume \( ?\text{rhs} \)
  show \( ?\text{lhs} \)
proof
fix $k$

show $\text{map-of } xs\ k = \text{map-of } ys\ k$ proof (cases $\text{map-of } xs\ k$)

  case None
  with ($\langle \text{rhs} \rangle$) have $\text{map-of } ys\ k = \text{None}$
    by (simp add: $\text{map-of-eq-None-iff}$)
  with None show $\langle \text{thesis} \rangle$ by simp

next
  case ($\langle \text{Some } v \rangle$)
  with distinct ($\langle \text{rhs} \rangle$) have $\text{map-of } ys\ k = \text{Some } v$
    by simp
  with Some show $\langle \text{thesis} \rangle$ by simp

qed

qed

end

70 Enum: Finite types as explicit enumerations

theory Enum
imports Map
begin

70.1 Class enum

class enum =
  fixes enum :: $'a\ \text{list}$
  fixes enum-all :: ($'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$
  fixes enum-ex :: ($'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$
  assumes UNIV-enum: $\text{UNIV} = \text{set enum}$
    and enum-distinct: distinct enum
  assumes enum-all-UNIV: $\text{enum-all } P \leftrightarrow \text{Ball } \text{UNIV } P$
  assumes enum-ex-UNIV: $\text{enum-ex } P \leftrightarrow \text{Bex } \text{UNIV } P$
  — tailored towards simple instantiation

begin

subclass finite proof
  qed (simp add: UNIV-enum)

lemma enum-UNIV:
  set enum = UNIV
  by (simp only: UNIV-enum)

lemma in-enum: $x \in \text{set enum}$
  by (simp add: enum-UNIV)

lemma enum-eq-I:
  assumes $\forall x. x \in \text{set xs}$
  shows $\text{set enum} = \text{set xs}$
proof 
  from assms UNIV-eq-I have UNIV = set xs by auto 
with enum-UNIV show ?thesis by simp 
qed 

lemma card-UNIV-length-enum: 
  card (UNIV :: 'a set) = length enum 
by (simp add: UNIV-enum distinct-card enum-distinct) 

lemma enum-all [simp]: 
  enum-all = HOL.All 
by (simp add: fun-eq-iff enum-all-UNIV) 

lemma enum-ex [simp]: 
  enum-ex = HOL.Ex 
by (simp add: fun-eq-iff enum-ex-UNIV) 

end 

70.2 Implementations using enum 

70.2.1 Unbounded operations and quantifiers 

lemma Collect-code [code]: 
  Collect P = set (filter P enum) 
by (simp add: enum-UNIV) 

lemma vimage-code [code]: 
  f -' B = set (filter (%x. f x : B) enum-class.enum) 
unfolding vimage-def Collect-code .. 

definition card-UNIV :: 'a itself ⇒ nat 
where 
  [code del]: card-UNIV TYPE('a) = card (UNIV :: 'a set) 

lemma [code]: 
  card-UNIV TYPE('a :: enum) = card (set (Enum.enum :: 'a list))) 
by (simp only: card-UNIV-def enum-UNIV) 

lemma all-code [code]: (∀ x. P x) ↔ enum-all P 
by simp 

lemma exists-code [code]: (∃ x. P x) ↔ enum-ex P 
by simp 

lemma exists1-code [code]: (∃! x. P x) ↔ list-ex1 P enum 
by (auto simp add: list-ex1-iff enum-UNIV)
70.2.2 An executable choice operator

definition
[code def]: enum-the = The

lemma [code]:
The P = (case filter P enum of [x] => x | - => enum-the P)
proof -
{ fix a
  assume filter-enum: filter P enum = [a]
  have The P = a
    proof (rule the-equality)
      fix x
      assume P x
      show x = a
        proof (rule ccontr)
          assume x ≠ a
          from filter-enum obtain us vs
            where enum-eq: enum = us @@ [a] @@ vs
            and ∀ x ∈ set us. ¬ P x
            and ∀ x ∈ set vs. ¬ P x
            and P a
            by (auto simp add: filter-eq-Cons-iff) (simp only: filter-empty-conv[symmetric])
            with (: P x) in-enun[of x, unfolded enum-eq] (x ≠ a) show False by auto
        qed
    next
    from filter-enum show P a by (auto simp add: filter-eq-Cons-iff)
    qed
} from this show {?thesis
  unfolding enum-the-def by (auto split: list.split)
  qed

declare [[code abort: enum-the]]

code-printing
  constant enum-the ⇒ (Eval) (fn ' =⇒ raise Match)

70.2.3 Equality and order on functions

instantiation fun :: (enum, equal) equal
begin

definition
  HOL.equal f g (⇒ (∀ x ∈ set enum. f x = g x))

instance proof
  qed (simp-all add: equal-fun-def fun-eq-iff enum-UNIV)
THEORY “Enum”

lemma [code]:
  HOL.equal f g ←→ enum-all (%x. f x = g x)
by (auto simp add: equal-fun-eq-iff)

lemma [code nbe]:
  HOL.equal (f :: - ⇒ -) f ←→ True
by (fact equal-refl)

lemma order-fun [code]:
  fixes 
  shows 
  by (simp-all add: fun-eq-iff le-fun-def order-less-le)

70.2.4 Operations on relations
lemma [code]:
  Id = image (λx. (x, x)) (set Enum.enum)
by (auto intro: imageI in-enumeration)

lemma tranclp-unfold [code]:
  tranclp r a b ←→ (a, b) ∈ trancl {x, y}.
by (simp add: trancl-def)

lemma rtranclp-rtrancl-eq [code]:
  rtranclp r x y ←→ (x, y) ∈ rtrancl
by (simp add: rtrancl-def)

lemma max-ext-eq [code]:
  max-ext R = {{X, Y}. finite X ∧ finite Y ∧ Y ≠ {} ∧ (∀x. x ∈ X → (∃xa ∈ Y. (x, xa) ∈ R))}
by (auto simp add: max-ext.simps)

lemma max-extp-eq [code]:
  max-extp r x y ←→ (x, y) ∈ max-ext
by (simp add: max-ext-def)

lemma mlex-eq [code]:
  f <∗mlex∗> R = {(x, y). f x < f y ∨ (f x ≤ f y ∧ (x, y) ∈ R)}
by (auto simp add: mlex-prod-def)

70.2.5 Bounded accessible part
primrec bacc :: ('a × 'a) set ⇒ nat ⇒ 'a set
where
  bacc r 0 = {x. ∀ y. (y, x) /∈ r}
| bacc r (Suc n) = (bacc r n ∪ {x. ∀ y. (y, x) ∈ r → y ∈ bacc r n})
lemma bacc-subseteq-acc:
bacc r n ⊆ Wellfounded.acc r
by (induct n) (auto intro: acc.intros)

lemma bacc-mono:
n ≤ m ⇒ bacc r n ⊆ bacc r m
by (induct rule: dec-induct) auto

lemma bacc-upper-bound:
bacc (r :: ('a × 'a) set) (card (UNIV :: 'a::finite set)) = (∪ n. bacc r n)
proof
have mono (bacc r) unfolding mono-def by (simp add: bacc-mono)
moreover have ∀ n. bacc r n = bacc r (Suc n) −→ bacc r (Suc (Suc n)) by auto
moreover have finite (range (bacc r)) by auto
ultimately show ?thesis
  by (intro finite-mono-strict-prefix-implies-finite-fixpoint)
  (auto intro: finite-mono-remains-stable-implies-strict-prefix)
qed

lemma acc-subseteq-bacc:
assumes finite r
shows Wellfounded.acc r ⊆ (∪ n. bacc r n)
proof
  fix x
  assume x : Wellfounded.acc r
  then have ∃ n. x : bacc r n
    proof (induct x arbitrary: rule: acc.induct)
      case (accI x)
      then have ∀ y. ∃ n. (y, x) ∈ r −→ y : bacc r n by simp
      from choice[OF this] obtain n where n y : r −→ y ∈ bacc r (n y)
      ..
      obtain n where ∀ y. (y, x) : r −→ y : bacc r n
    proof
      fix y assume y : (y, x) : r
      with n have y : bacc r (n y) by auto
      moreover have n y ≤ Max ((%(y, x). n y) :: r)
        using y (finite r) by (auto intro: Max-ge)
      note bacc-mono[OF this, of r]
      ultimately show y : bacc r (Max ((%(y, x). n y) :: r)) by auto
    qed
    then show ?case
      by (auto simp add: Let-def intro!: exI[of - Suc n])
  qed
  then show x : (∪ n. bacc r n) by auto
qed

lemma acc-bacc-eq:
  fixes A :: ('a :: finite × 'a) set
THEORY "Enum"

assumes finite A
shows Wellfounded.acc A = bacc A (card (UNIV :: 'a set))
using assms by (metis acc-subseteq-bacc bacc-subseteq-acc bacc-upper-bound order-eq-iff)

lemma [code]:
fixes xs :: ('a::finite × 'a) list
shows Wellfounded.acc (set xs) = bacc (set xs) (card-UNIV TYPE('a))
by (simp add: card-UNIV-def acc-bacc-eq)

70.3 Default instances for enum
lemma map-of-zip-enum-is-Some:
fixes xs ys :: 'b::enum list
assumes length: length xs = length (enum :: 'a::enum list)
length ys = length (enum :: 'a::enum list)
and map-of: the ∘ map-of (zip (enum :: 'a::enum list) xs) = the ∘ map-of (zip (enum :: 'a::enum list) ys)
shows xs = ys
proof
have map-of (zip (enum :: 'a list) xs) = map-of (zip (enum :: 'a list) ys)
proof
fix x :: 'a
from length map-of-zip-enum-is-Some obtain y1 y2
where map-of (zip (enum :: 'a list) xs) x = Some y1
and map-of (zip (enum :: 'a list) ys) x = Some y2 by blast
moreover from map-of
have the (map-of (zip (enum :: 'a::enum list) xs) x) = the (map-of (zip (enum :: 'a::enum list) ys) x)
by (auto dest: fun-cong)
ultimately show map-of (zip (enum :: 'a::enum list) xs) x = map-of (zip (enum :: 'a::enum list) ys) x
by simp
qed
with length enum-distinct show xs = ys by (rule map-of-zip-inject)
qed

definition all-n-lists :: ('a :: enum) list ⇒ bool ⇒ nat ⇒ bool
where
all-n-lists P n ⟷ (∀ xs ∈ set (List.n-lists n enum), P xs)
lemma [code]:
all-n-lists P n \leftrightarrow (if n = 0 then P \square else enum-all (%x. all-n-lists (%xs. P (x # xs)) (n - 1)))
unfolding all-n-lists-def enum-all
by (cases n) (auto simp add: enum-UNIV)

definition ex-n-lists :: (('a::enum) list \Rightarrow) nat \Rightarrow bool
where
ex-n-lists P n \leftrightarrow (\exists xs \in set (List.n-lists n enum). P xs)

lemma [code]:
ex-n-lists P n \leftrightarrow (if n = 0 then P \square else enum-ex (%x. ex-n-lists (%xs. P (x # xs)) (n - 1)))
unfolding ex-n-lists-def enum-ex
by (cases n) (auto simp add: enum-UNIV)

instantiation fun :: (enum, enum) enum
begin

definition enum = map (\lambda ys. the o map-of (zip (enum::'a list) ys)) (List.n-lists (length (enum::'a::enum list)) enum)

definition enum-all P = all-n-lists (\lambda bs. P (the o map-of (zip enum bs))) (length (enum :: 'a list))

definition enum-ex P = ex-n-lists (\lambda bs. P (the o map-of (zip enum bs))) (length (enum :: 'a list))

instance proof
show UNIV = set (enum :: ('a \Rightarrow 'b) list)
proof (rule UNIV-eq-I)
fix f :: 'a \Rightarrow 'b
have f = the o map-of (zip (enum :: 'a::enum list) (map f enum))
by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
then show f \in set enum
by (auto simp add: enum-fun-def set-n-lists intro: in-enum)
qed
next
from map-of-zip-enun-inject
show distinct (enum :: ('a \Rightarrow 'b) list)
by (auto intro!: inj-onI simp add: enum-fun-def
distinct-map distinct-n-lists enum-distinct set-n-lists)
next
fix P
show enum-all (P :: ('a \Rightarrow 'b) \Rightarrow bool) = Ball UNIV P
proof
  assume enum-all P
  show Ball UNIV P
  proof
    fix f :: 'a ⇒ 'b
    have f: f = the ◦ map-of (zip (enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
    from (enum-all P) have P (the ◦ map-of (zip enum (map f enum)))
      unfolding enum-all-fun-def all-n-lists-def
      apply (simp add: set-n-lists)
      apply (erule-tac x=map f enum in allE)
      apply (auto intro!: in-enum)
      done
    from this f show P f by auto
  qed
next
  assume Ball UNIV P
  from this show enum-all P
  unfolding enum-all-fun-def all-n-lists-def by auto
  qed
next
  fix P
  show enum-ex (P :: ('a ⇒ 'b ⇒ bool) ⇒ Bex UNIV P
  proof
    assume enum-ex P
    from this show Bex UNIV P
    unfolding enum-ex-fun-def ex-n-lists-def by auto
  next
    assume Bex UNIV P
    from this obtain f where P f ..
    have f: f = the ◦ map-of (zip (enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
    from (P f) this have P (the ◦ map-of (zip (enum :: 'a::enum list) (map f enum)))
      by auto
    from this show enum-ex P
    unfolding enum-ex-fun-def ex-n-lists-def
    apply (auto simp add: set-n-lists)
    apply (rule-tac x=map f enum in exI)
    apply (auto intro!: in-enum)
    done
  qed
  qed
end

lemma enum-fun-code [code]:
enum = (let enum-a = (enum :: 'a::{enum, equal} list)
in map (λys. the ◦ map-of (zip enum-a ys)) (List.n-lists (length enum-a) enum))
THEORY “Enum”

by (simp add: enum-fun-def Let-def)

lemma enum-all-fun-code [code]:
  enum-all P = (let enum-a = (enum :: 'a::{enum, equal} list)
in all-n-lists (λbs. P (the o map-of (zip enum-a bs))) (length enum-a))
by (simp only: enum-all-fun-def Let-def)

lemma enum-ex-fun-code [code]:
  enum-ex P = (let enum-a = (enum :: 'a::{enum, equal} list)
in ex-n-lists (λbs. P (the o map-of (zip enum-a bs))) (length enum-a))
by (simp only: enum-ex-fun-def Let-def)

instantiation set :: (enum) enum
begin

definition enum = map set (sublists enum)

definition enum-all P ←→ (∀ A∈set enum. P (A::'a set))

definition enum-ex P ←→ (∃ A∈set enum. P (A::'a set))

instance proof
qed (simp-all add: enum-set-def enum-all-set-def enum-ex-set-def sublists-powset
distinct-set-sublists enum-distinct enum-UNIV)

end

instantiation unit :: enum
begin

definition enum = [(])

definition enum-all P = P ()

definition enum-ex P = P ()

instance proof
qed (auto simp add: enum-unit-def enum-all-unit-def enum-ex-unit-def)

end

instantiation bool :: enum


begin

definition
enum = [False, True]

definition
definition enum-all P ↦ P False ∧ P True
definition enum-ex P ↦ P False ∨ P True

instance proof
qed (simp-all only: enum-bool-def enum-all-bool-def enum-ex-bool-def UNIV-bool, simp-all)

end

instantiation prod :: (enum, enum) enum
begin

definition
definition enum = List.product enum enum
definition enum-all P = enum-all (λx. enum-all (λy. P (x, y)))
definition enum-ex P = enum-ex (λx. enum-ex (λy. P (x, y)))

instance by default

end

instantiation sum :: (enum, enum) enum
begin

definition
definition enum = map Inl enum @ map Inr enum
definition enum-all P ↦ enum-all (λx. P (Inl x)) ∧ enum-all (λx. P (Inr x))
definition enum-ex P ↦ enum-ex (λx. P (Inl x)) ∨ enum-ex (λx. P (Inr x))

instance proof
qed (simp-all only: enum-sum-def enum-all-sum-def enum-ex-sum-def UNIV-sum,
auto simp add: enum-UNIV distinct-map enum-distinct)

end

instantiation option :: (enum) enum
begin

definition
enum = None # map Some enum

definition
enum-all P <-> P None & enum-all (x. P (Some x))

definition
enum-ex P <-> P None v enum-ex (x. P (Some x))

instance proof
qed (simp-all only: enum-option-def enum-all-option-def enum-ex-option-def UNIV-option-conv,
auto simp add: distinct-map enum-UNIV enum-distinct)

end

70.4 Small finite types

We define small finite types for the use in Quickcheck

datatype finite-1 = a_1

notation (output) a_1 (a_1)

lemma UNIV-finite-1:
UNIV = {a_1}
by (auto intro: finite-1.exhaust)

instantiation finite-1 :: enum
begin

definition
enum = [a_1]

definition
enum-all P = P a_1

definition
enum-ex P = P a_1

instance proof
qed (simp-all only: enum-finite-1-def enum-all-finite-1-def enum-ex-finite-1-def UNIV-finite-1,
simp-all)
THEORY "Enum"

end

instantiation finite-1 :: linorder
begin

definition less-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  x < (y :: finite-1) ⟷ False

definition less-eq-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  x ≤ (y :: finite-1) ⟷ True

instance
apply (intro-classes)
apply (auto simp add: less-finite-1-def less-eq-finite-1-def)
apply (metis finite-1.exhaust)
done

end

hide-const (open) a₁

datatype finite-2 = a₁ | a₂

notation (output) a₁ (a₁)
notation (output) a₂ (a₂)

lemma UNIV-finite-2:
  UNIV = {a₁, a₂}
  by (auto intro: finite-2.exhaust)

instantiation finite-2 :: enum
begin

definition enum = [a₁, a₂]

definition enum-all P ⟷ P a₁ ∧ P a₂

definition enum-ex P ⟷ P a₁ ∨ P a₂

instance proof
qed (simp-all only: enum-finite-2-def enum-all-finite-2-def enum-ex-finite-2-def UNIV-finite-2, simp-all)
THEORY "Enum"

end

instantiation finite-2 :: linorder
begin

definition less-finite-2 :: finite-2 ⇒ finite-2 ⇒ bool
where
\( x < y \iff x = a_1 \land y = a_2 \)

definition less-eq-finite-2 :: finite-2 ⇒ finite-2 ⇒ bool
where
\( x \leq y \iff x = y \lor x < (y :: finite-2) \)

instance
apply (intro-classes)
apply (auto simp add: less-finite-2_def less-eq-finite-2_def)
apply (metis finite-2.nchotomy)+
done

end

hide-const (open) \( a_1 \) \( a_2 \)

datatype finite-3 = \( a_1 \mid a_2 \mid a_3 \)

notation (output) \( a_1 \) (\( a_1 \))
notation (output) \( a_2 \) (\( a_2 \))
notation (output) \( a_3 \) (\( a_3 \))

lemma UNIV-finite-3:
\( \text{UNIV} = \{ a_1, a_2, a_3 \} \)
by (auto intro: finite-3.exhaust)

instantiation finite-3 :: enum
begin

definition enum = [\( a_1 \), \( a_2 \), \( a_3 \)]

definition enum-all \( P \) \iff \( P a_1 \land P a_2 \land P a_3 \)

definition enum-ex \( P \) \iff \( P a_1 \lor P a_2 \lor P a_3 \)

instance proof
qed (simp-all only: enum-finite-3-def enum-all-finite-3-def enum-ex-finite-3-def UNIV-finite-3, simp-all)
THEORY "Enum"

end

instantiation finite-3 :: linorder
begin

definition less-finite-3 :: finite-3 ⇒ finite-3 ⇒ bool
where
  \( x < y = \text{(case } x \text{ of } a_1 \Rightarrow y \neq a_1 | a_2 \Rightarrow y = a_3 | a_3 \Rightarrow \text{False)} \)

definition less-eq-finite-3 :: finite-3 ⇒ finite-3 ⇒ bool
where
  \( x \leq y \iff x = y \lor x < (y :: \text{finite-3}) \)

instance proof (intro-classes)
qed (auto simp add: less-finite-3-def less-eq-finite-3-def split: finite-3.split_asm)
end

hide-const (open) \( a_1 \ a_2 \ a_3 \)

datatype finite-4 = \( a_1 | a_2 | a_3 | a_4 \)

notation (output) \( a_1 \ (a_1) \)
notation (output) \( a_2 \ (a_2) \)
notation (output) \( a_3 \ (a_3) \)
notation (output) \( a_4 \ (a_4) \)

lemma UNIV-finite-4:
  \( \text{UNIV} = \{a_1, a_2, a_3, a_4\} \)
  by (auto intro: finite-4.exhaust)

instantiation finite-4 :: enum
begin

definition enum = \( [a_1, a_2, a_3, a_4] \)

definition enum-all \( P \) ≡ \( P \ a_1 \land P \ a_2 \land P \ a_3 \land P \ a_4 \)

definition enum-ex \( P \) ≡ \( P \ a_1 \lor P \ a_2 \lor P \ a_3 \lor P \ a_4 \)

instance proof
  qed (simp-all only: enum-finite-4-def enum-all-finite-4-def enum-ex-finite-4-def UNIV-finite-4, simp-all)
end
hide-const (open) $a_1 \ a_2 \ a_3 \ a_4$

datatype finite-5 = $a_1 \mid a_2 \mid a_3 \mid a_4 \mid a_5$

notation (output) $a_1 \ (a_1)$
notation (output) $a_2 \ (a_2)$
notation (output) $a_3 \ (a_3)$
notation (output) $a_4 \ (a_4)$
notation (output) $a_5 \ (a_5)$

lemma UNIV-finite-5:
UNIV = \{a_1, a_2, a_3, a_4, a_5\}
bym (auto intro: finite-5.exhaust)

instantiation finite-5 :: enum
begin

definition enum = [a_1, a_2, a_3, a_4, a_5]

definition enum-all $P \longleftrightarrow P \ a_1 \land P \ a_2 \land P \ a_3 \land P \ a_4 \land P \ a_5$

definition enum-ex $P \longleftrightarrow P \ a_1 \lor P \ a_2 \lor P \ a_3 \lor P \ a_4 \lor P \ a_5$

instance proof
qed (simp-all only: enum-finite-5-def enum-all-finite-5-def enum-ex-finite-5-def UNIV-finite-5, simp-all)

end

hide-const (open) $a_1 \ a_2 \ a_3 \ a_4 \ a_5$

70.5 Closing up

hide-type (open) finite-1 finite-2 finite-3 finite-4 finite-5
hide-const (open) enum enum-all enum-ex all-n-lists ex-n-lists ntrancl

end

71 String: Character and string types

theory String
imports Enum
begin
71.1 Characters and strings

datatype nibble =
  Nibble0 | Nibble1 | Nibble2 | Nibble3 | Nibble4 | Nibble5 | Nibble6 | Nibble7 |
  Nibble8 | Nibble9 | NibbleA | NibbleB | NibbleC | NibbleD | NibbleE | NibbleF

lemma UNIV-nibble:
  UNIV = {Nibble0, Nibble1, Nibble2, Nibble3, Nibble4, Nibble5, Nibble6, Nibble7,
           Nibble8, Nibble9, NibbleA, NibbleB, NibbleC, NibbleD, NibbleE, NibbleF} (is - = ?A)
proof (rule UNIV-eq-I)
  fix x show x ∈ ?A by (cases x) simp-all
qed

lemma size-nibble [code, simp]:
  size-nibble (x::nibble) = 0
  size (x::nibble) = 0
by (cases x, simp-all)+

instantiation nibble :: enum
begin

definition
  Enum.enum = [Nibble0, Nibble1, Nibble2, Nibble3, Nibble4, Nibble5, Nibble6,
                Nibble7,
                Nibble8, Nibble9, NibbleA, NibbleB, NibbleC, NibbleD, NibbleE, NibbleF]

definition
  Enum.enum-all P ←→ P Nibble0 ∧ P Nibble1 ∧ P Nibble2 ∧ P Nibble3 ∧ P
  Nibble4 ∧ P Nibble5 ∧ P Nibble6 ∧ P Nibble7 ∧ P Nibble8 ∧ P Nibble9 ∧ P NibbleA ∧ P
  NibbleB ∧ P NibbleC ∧ P NibbleD ∧ P NibbleE ∧ P NibbleF

definition
  Enum.enum-ex P ←→ P Nibble0 ∨ P Nibble1 ∨ P Nibble2 ∨ P Nibble3 ∨ P
  Nibble4 ∨ P Nibble5 ∨ P Nibble6 ∨ P Nibble7 ∨ P Nibble8 ∨ P Nibble9 ∨ P NibbleA ∨ P
  NibbleB ∨ P NibbleC ∨ P NibbleD ∨ P NibbleE ∨ P NibbleF

instance proof
qed (simp-all only: enum-nibble-def enum-all-nibble-def enum-ex-nibble-def UNIV-nibble,
  simp-all)
end

lemma card-UNIV-nibble:
  card (UNIV :: nibble set) = 16
  by (simp add: card-UNIV-length-enum enum-nibble-def)

primrec nat-of-nibble :: nibble ⇒ nat
where

nat-of-nibble Nibble0 = 0
| nat-of-nibble Nibble1 = 1
| nat-of-nibble Nibble2 = 2
| nat-of-nibble Nibble3 = 3
| nat-of-nibble Nibble4 = 4
| nat-of-nibble Nibble5 = 5
| nat-of-nibble Nibble6 = 6
| nat-of-nibble Nibble7 = 7
| nat-of-nibble Nibble8 = 8
| nat-of-nibble Nibble9 = 9
| nat-of-nibble NibbleA = 10
| nat-of-nibble NibbleB = 11
| nat-of-nibble NibbleC = 12
| nat-of-nibble NibbleD = 13
| nat-of-nibble NibbleE = 14
| nat-of-nibble NibbleF = 15

definition nibble-of-nat :: nat ⇒ nibble where
nibble-of-nat n = Enum.enum ! (n mod 16)

lemma nibble-of-nat-simps [simp]:
| nibble-of-nat 0 = Nibble0
| nibble-of-nat 1 = Nibble1
| nibble-of-nat 2 = Nibble2
| nibble-of-nat 3 = Nibble3
| nibble-of-nat 4 = Nibble4
| nibble-of-nat 5 = Nibble5
| nibble-of-nat 6 = Nibble6
| nibble-of-nat 7 = Nibble7
| nibble-of-nat 8 = Nibble8
| nibble-of-nat 9 = Nibble9
| nibble-of-nat 10 = NibbleA
| nibble-of-nat 11 = NibbleB
| nibble-of-nat 12 = NibbleC
| nibble-of-nat 13 = NibbleD
| nibble-of-nat 14 = NibbleE
| nibble-of-nat 15 = NibbleF

unfolding nibble-of-nat-def by (simp-all add: enum-nibble-def)

lemma nibble-of-nat-of-nibble [simp]:
| nibble-of-nat (nat-of-nibble x) = x
by (cases x) (simp-all add: nibble-of-nat-def enum-nibble-def)

lemma nat-of-nibble-of-nat [simp]:
| nat-of-nibble (nibble-of-nat n) = n mod 16
by (cases nibble-of-nat n)
| (simp-all add: nibble-of-nat-def enum-nibble-def nth-equal-first-eq nth-non-equal-first-eq, arith)
lemma inj-nat-of-nibble:
inj nat-of-nibble
by (rule inj-on-inverseI) (rule nibble-of-nat-of-nibble)

lemma nat-of-nibble-eq-iff:
nat-of-nibble x = nat-of-nibble y ⟷ x = y
by (rule inj-eq) (rule inj-nat-of-nibble)

lemma nat-of-nibble-less-16:
nat-of-nibble x < 16
by (cases x) auto

lemma nibble-of-nat-mod-16:
nibble-of-nat (n mod 16) = nibble-of-nat n
by (simp add: nibble-of-nat-def)

datatype char = Char nibble nibble
— Note: canonical order of character encoding coincides with standard term ordering

syntax
  -Char :: str-position ⇒ char  (CHR -)

type-synonym string = char list

syntax
  -String :: str-position ⇒ string  (-)

ML-file Tools/string-syntax.ML
setup String-Syntax.setup

lemma UNIV-char:
  UNIV = image (split Char) (UNIV × UNIV)
proof (rule UNIV-eq-I)
  fix x show x ∈ image (split Char) (UNIV × UNIV) by (cases x) auto
qed

lemma size-char [code, simp]:
  size-char (c::char) = 0
  size (c::char) = 0
  by (cases c, simp)+

instantiation char :: enum
begin

definition
  Enum.enum = [Char Nibble0 Nibble0, Char Nibble0 Nibble1, Char Nibble0 Nibble2,
THEORY “String”

Char Nibble0 Nibble3, Char Nibble0 Nibble4, Char Nibble0 Nibble5, Char Nibble0 Nibble6, Char Nibble0 Nibble7, Char Nibble0 Nibble8, Char Nibble0 Nibble9, CHR "[", Char Nibble0 NibbleB, Char Nibble0 NibbleC, Char Nibble0 NibbleD, Char Nibble0 NibbleE, Char Nibble0 NibbleF, Char Nibble1 Nibble0, Char Nibble1 Nibble1, Char Nibble1 Nibble2, Char Nibble1 Nibble3, Char Nibble1 Nibble4, Char Nibble1 Nibble5, Char Nibble1 Nibble6, Char Nibble1 Nibble7, Char Nibble1 Nibble8, Char Nibble1 Nibble9, Char Nibble1 NibbleA, Char Nibble1 NibbleB, Char Nibble1 NibbleC, Char Nibble1 NibbleD, Char Nibble1 NibbleE, Char Nibble1 NibbleF, CHR ":", CHR ";", Char Nibble2 Nibble2, CHR ";", CHR ";", Char Nibble2 Nibble5, CHR ";", CHR ";", Char Nibble2 Nibble7, CHR ";", CHR ";", Char Nibble2 Nibble8, Char Nibble2 Nibble9, Char Nibble2 Nibble3, Char Nibble2 Nibble4, Char Nibble2 Nibble5, Char Nibble2 Nibble6, Char Nibble2 Nibble7, Char Nibble2 Nibble8, Char Nibble2 Nibble9, Char Nibble2 NibbleA, Char Nibble2 NibbleB, Char Nibble2 NibbleC, Char Nibble2 NibbleD, Char Nibble2 NibbleE, Char Nibble2 NibbleF, Char Nibble3 Nibble0, Char Nibble3 Nibble1, Char Nibble3 Nibble2, Char Nibble3 Nibble3, Char Nibble3 Nibble4, Char Nibble3 Nibble5, Char Nibble3 Nibble6, Char Nibble3 Nibble7, Char Nibble3 Nibble8, Char Nibble3 Nibble9, Char Nibble3 NibbleA, Char Nibble3 NibbleB, Char Nibble3 NibbleC, Char Nibble3 NibbleD, Char Nibble3 NibbleE, Char Nibble3 NibbleF, Char Nibble4 Nibble0, Char Nibble4 Nibble1, Char Nibble4 Nibble2, Char Nibble4 Nibble3, Char Nibble4 Nibble4, Char Nibble4 Nibble5, Char Nibble4 Nibble6, Char Nibble4 Nibble7, Char Nibble4 Nibble8, Char Nibble4 Nibble9, Char Nibble4 NibbleA, Char Nibble4 NibbleB, Char Nibble4 NibbleC, Char Nibble4 NibbleD, Char Nibble4 NibbleE, Char Nibble4 NibbleF, Char Nibble5 Nibble0, Char Nibble5 Nibble1, Char Nibble5 Nibble2, Char Nibble5 Nibble3, Char Nibble5 Nibble4, Char Nibble5 Nibble5, Char Nibble5 Nibble6, Char Nibble5 Nibble7, Char Nibble5 Nibble8, Char Nibble5 Nibble9, Char Nibble5 NibbleA, Char Nibble5 NibbleB, Char Nibble5 NibbleC, Char Nibble5 NibbleD, Char Nibble5 NibbleE, Char Nibble5 NibbleF, Char Nibble6 Nibble0, Char Nibble6 Nibble1, Char Nibble6 Nibble2, Char Nibble6 Nibble3, Char Nibble6 Nibble4, Char Nibble6 Nibble5, Char Nibble6 Nibble6, Char Nibble6 Nibble7, Char Nibble6 Nibble8, Char Nibble6 Nibble9, Char Nibble6 NibbleA, Char Nibble6 NibbleB, Char Nibble6 NibbleC, Char Nibble6 NibbleD, Char Nibble6 NibbleE, Char Nibble6 NibbleF, Char Nibble7 Nibble0, Char Nibble7 Nibble1, Char Nibble7 Nibble2, Char Nibble7 Nibble3, Char Nibble7 Nibble4, Char Nibble7 Nibble5, Char Nibble7 Nibble6, Char Nibble7 Nibble7, Char Nibble7 Nibble8, Char Nibble7 Nibble9, Char Nibble7 NibbleA, Char Nibble7 NibbleB, Char Nibble7 NibbleC, Char Nibble7 NibbleD, Char Nibble7 NibbleE, Char Nibble7 NibbleF, Char Nibble8 Nibble0, Char Nibble8 Nibble1, Char Nibble8 Nibble2, Char Nibble8 Nibble3, Char Nibble8 Nibble4, Char Nibble8 Nibble5, Char Nibble8 Nibble6, Char Nibble8 Nibble7, Char Nibble8 Nibble8, Char Nibble8 Nibble9, Char Nibble8 NibbleA, Char Nibble8 NibbleB, Char Nibble8 NibbleC, Char Nibble8 NibbleD, Char Nibble8 NibbleE, Char Nibble8 NibbleF, Char Nibble9 Nibble0, Char Nibble9 Nibble1, Char Nibble9 Nibble2, Char Nibble9 Nibble3, Char Nibble9 Nibble4, Char Nibble9 Nibble5, Char Nibble9 Nibble6, Char Nibble9 Nibble7, Char Nibble9 Nibble8, Char Nibble9 Nibble9, Char Nibble9 NibbleA, Char Nibble9 NibbleB, Char Nibble9 NibbleC, Char Nibble9 NibbleD, Char Nibble9 NibbleE, Char Nibble9 NibbleF, Char NibbleA Nibble0, Char NibbleA Nibble1, Char NibbleA Nibble2, Char NibbleA Nibble3, Char NibbleA Nibble4, Char NibbleA Nibble5, Char NibbleA Nibble6, Char NibbleA Nibble7, Char NibbleA Nibble8, Char NibbleA Nibble9, Char NibbleA NibbleA, Char NibbleA NibbleB, Char NibbleA NibbleC, Char NibbleA NibbleD, Char NibbleA NibbleE, Char NibbleA NibbleF, Char NibbleB Nibble0, Char NibbleB Nibble1, Char NibbleB Nibble2, Char NibbleB Nibble3, Char NibbleB Nibble4, Char NibbleB Nibble5, Char NibbleB Nibble6, Char NibbleB Nibble7, Char NibbleB Nibble8, Char NibbleB Nibble9, Char NibbleB NibbleA, Char NibbleB NibbleB, Char NibbleB NibbleC, Char NibbleB NibbleD, Char NibbleB NibbleE, Char NibbleB NibbleF, Char NibbleC Nibble0, Char NibbleC Nibble1, Char NibbleC Nibble2, Char NibbleC Nibble3, Char NibbleC Nibble4, Char NibbleC Nibble5, Char NibbleC Nibble6, Char NibbleC Nibble7, Char NibbleC Nibble8, Char NibbleC Nibble9,
definition
Enum.enum-all P ⇔ list-all P (Enum.enum :: char list)
definition
Enum.enum-ex P ⇔ list-ex P (Enum.enum :: char list)

lemma enum-char-product-enum-nibble:
(Enum.enum :: char list) = map (split Char) (List.product Enum.enum Enum.enum)
by (simp add: enum-char-def enum-nibble-def)

instance proof
show UNIV: UNIV = set (Enum.enum :: char list)
  by (simp add: enum-char-product-enum-nibble UNIV-char enum-UNIV)
show distinct (Enum.enum :: char list)
  by (auto intro: inj-onI simp add: enum-char-product-enum-nibble distinct-map
   distinct-product enum-distinct)
show ∀P. Enum.enum-all P ⇔ Ball (UNIV :: char set) P
  by (simp add: UNIV enum-all-char-def list-all-iff)
show ∀P. Enum.enum-ex P ⇔ Bex (UNIV :: char set) P
  by (simp add: UNIV enum-ex-char-def list-ex-iff)
qed

end

lemma card-UNIV-char:
card (UNIV :: char set) = 256
by (simp add: card-UNIV-length enum-char-def)

definition nat-of-char :: char ⇒ nat
where
nat-of-char \( c \) = (case \( c \) of \( \text{Char} \) \( x \) \( y \) ⇒ \( \text{nat-of-nibble} \) \( x \) \(*\) \( 16 \) + \( \text{nat-of-nibble} \) \( y \))

**lemma** nat-of-char-Char:

\( \text{nat-of-char} \) \( (\text{Char} \) \( x \) \( y \)) = \( \text{nat-of-nibble} \) \( x \) \(*\) \( 16 \) + \( \text{nat-of-nibble} \) \( y \)

by (simp add: nat-of-char-def)

**setup** {
let
val nibbles = map-range (Thm.cterm_of @{theory} o HOLogic.mk-nibble) 16;
val simpset =
  put-simpset HOL-ss @{context}
  addsimps @{thms nat-of-nibble_simps mult-0 mult-1 add-0 add-0-right arith-simps numeral-plus-one};
fun mk-code-eqn \( x \) \( y \) =
  Drule.instantiate’ [[] [SOME \( x \), SOME \( y \)] @\{thm nat-of-char-Char\}]
  |> simplify simpset;
val code-eqns = map-product mk-code-eqn nibbles nibbles;
in
Global-Theory.note-thmss
  [[@{binding nat-of-char-simps}, []], [@{code-eqns, []}]]
  #> snd
end
}

declare nat-of-char-simps [code]

**lemma** nibble-of-nat-of-char-div-16:

\( \text{nibble-of-nat} \) \( (\text{nat-of-char} \ c \div 16) \) = (case \( c \) of \( \text{Char} \) \( x \) \( y \) ⇒ \( x \))

by (cases \( c \))
  (simp add: nat-of-char-def add.commute nat-of-nibble-less-16)

**lemma** nibble-of-nat-nat-of-char:

\( \text{nibble-of-nat} \) \( (\text{nat-of-char} \ c) \) = (case \( c \) of \( \text{Char} \) \( x \) \( y \) ⇒ \( y \))

**proof** (cases \( c \))
  case (Char \( x \) \( y \))
  then have \( * \): \( \text{nibble-of-nat} \) \( ((\text{nibble-of-nat} \ y + \text{nibble-of-nibble} \ x \* 16) \mod 16) \) = \( y \)
    by (simp add: nibble-of-nat-mod-16)
  then have \( \text{nibble-of-nat} \) \( (\text{nibble-of-nibble} \ y + \text{nibble-of-nibble} \ x \* 16) \) = \( y \)
    by (simp only: nibble-of-nat-mod-16)
  with Char show \( ?\)thesis by (simp add: nat-of-char-def add.commute)
qed

**code-datatype** Char — drop case certificate for char

**lemma** case-char-code [code]:

\( \text{case-char} \) \( f \) \( c \) = (let \( n = \text{nat-of-char} \ c \) in \( f \) \( (\text{nibble-of-nat} \ (n \div 16)) \) \( (\text{nibble-of-nat} \ n) \))

by (cases \( c \))
(simp only: Let-def nibble-of-nat-of-char-div-16 nibble-of-nat-nat-of-char char.case)

lemma [code]:
rec-char = case-char
by (simp add: fun-eq-iff split: char.split)

definition char-of-nat :: nat ⇒ char where
char-of-nat n = Enum.enum ! (n mod 256)

lemma char-of-nat-Char-nibbles:
char-of-nat n = Char (nibble-of-nat (n div 16)) (nibble-of-nat n)
proof –
from mod-mult2-eq [of n 16 16]
have n mod 256 = 16 * (n div 16 mod 16) + n mod 16 by simp
then show ?thesis
commute)
qed

lemma char-of-nat-of-char [simp]:
char-of-nat (nat-of-char c) = c
by (cases c)

lemma nat-of-char-of-nat [simp]:
nat-of-char (char-of-nat n) = n mod 256
proof –
have n mod 256 = n mod (16 * 16) by simp
then have n div 16 mod 16 * 16 + n mod 16 = n mod 256 by (simp only: mod-mult2-eq)
then show ?thesis
by (cases char-of-nat n) (auto simp add: nat-of-char-def char-of-nat-Char-nibbles)
qed

lemma inj-nat-of-char:
inj nat-of-char
by (rule inj-on-inverseI) (rule char-of-nat-of-char)

lemma nat-of-char-eq-iff:
nat-of-char c = nat-of-char d ↔ c = d
by (rule inj-eq) (rule inj-nat-of-char)

lemma nat-of-char-less-256:
nat-of-char c < 256
proof (cases c)
case (Char x y)
with nat-of-nibble-less-16 [of x] nat-of-nibble-less-16 [of y]
show ?thesis by (simp add: nat-of-char-def)
qed
THEORY "String"

lemma char-of-nat-mod-256:
\[ \text{char-of-nat} \ (n \mod 256) = \text{char-of-nat} \ n \]
proof -
  from nibble-of-nat-mod-16 \[ \text{of n mod 256} \]
  have nibble-of-nat \ (n \mod 256) = nibble-of-nat \ (n \mod 256 \mod 16) \ by simp
  with nibble-of-nat-mod-16 \[ \text{of n} \]
  have \( \cdot \): nibble-of-nat \ (n \mod 256) = nibble-of-nat \ n \ by (simp add: mod-mod-cancel)
  have \( \cdot \cdot \): \( n \mod 256 = \ n \mod (16 \ast 16) \) \ by simp
  then have \( \cdot \cdot \cdot \): \( n \mod 256 = \ n \div 16 \mod 16 \ast 16 + \ n \mod 16 \) \ by (simp only: mod-mult2-eq)
  show \(?thesis \)
    by (simp add: char-of-nat-Char-nibbles \( \cdot \cdot \cdot \))
    (simp add: div-add1-eq nibble-of-nat-mod-16 \[ \text{of n div 16} \] \( \cdot \cdot \cdot \))
qed

71.2 Strings as dedicated type

typedef literal = UNIV :: string set
morphisms explode STR ..

setup-lifting (no-code) type-definition-literal

definition implode :: string \Rightarrow String.literal
where
  implode = STR

instantiation literal :: size
begin

definition size-literal :: literal \Rightarrow nat
where
  [code]: size-literal \( s :: \text{literal} \) = 0

instance ..

end

instantiation literal :: equal
begin

lift-definition equal-literal :: literal \Rightarrow literal \Rightarrow bool is op = .

instance by intro-classes (transfer, simp)

end

declare equal-literal.rep-eq[code]
lemma [code nbe]:
  fixes s :: String.literal
  shows HOL.equal s s ←→ True
  by (simp add: equal)

lemma STR-inject' [simp]:
  STR xs = STR ys ←→ xs = ys
  by (simp add: STR-inject)

lifting-update literal.lifting
lifting-forget literal.lifting

71.3 Code generator

ML-file Tools/string-code.ML

code-reserved SML string
code-reserved OCaml string
code-reserved Scala string

code-printing
  type-constructor literal ↦
    (SML) string
    and (OCaml) string
    and (Haskell) String
    and (Scala) String

setup ⟨⟨ fold String-Code.add-literal-string [SML, OCaml, Haskell, Scala] ⟩⟩

code-printing
  class-instance literal :: equal ↦
    (Haskell) -
    | constant HOL.equal :: literal ⇒ literal ⇒ bool ⇒
      (SML) !((· : string) = ·)
    and (OCaml) !((· : string) = ·)
    and (Haskell) infix 4 ==
    and (Scala) infixl 5 ==

setup ⟨⟨ Sign.map-naming (Name-Space.mandatory-path Code) ⟩⟩

definition abort :: literal ⇒ (unit ⇒ 'a) ⇒ 'a
  where [simp, code del]: abort - f = f ()

lemma abort-cong: msg = msg' ==> Code.abort msg f = Code.abort msg' f
  by simp

setup ⟨⟨ Sign.map-naming Name-Space.parent-path ⟩⟩
setup \{ Code-Simp.map-ss (Simplifier.add-cong @\{thm Code.abort-cong\}) \}

code-printing constant Code.abort \to
  (SML) !(raise/ Fail/ -)
  and (OCaml) failwith
  and (Haskell) error
  and (Scala) !{/ sys.error((-));/ ((-).apply())/ }

hide-type (open) literal

hide-const (open) implode explode

end

72 Typerep: Reflecting Pure types into HOL

theory Typerep
imports String
begin

datatype typerep = Typerep String.l literal typerep list

class typerep =
  fixes typerep :: 'a itself \Rightarrow typerep
begin

definition typerep-of :: 'a \Rightarrow typerep where
  [simp]: typerep-of x = typerep TYPE('a)
end

syntax
  -TYPEREP :: type => logic ((1TYPEREP/(1(('))))

parse-translation \{
  let
    fun typerep-tr (*-TYPEREP*) [ty] =
      Syntax.const @\{const-syntax typerep\} $ Syntax.const @\{const-syntax
Pure.type\} $
      (Syntax.const @\{syntax-const -constrain\} $ Syntax.const @\{const-syntax


typed-print-translation \{
  let
    fun typerep-tr' ctxt (*typerep*)
fun add-typerep tyco thy = 
let
  val sorts = replicate (Sign.arity-number thy tyco) @{sort typerep};
  val vs = Name.invent-names Name.context 'a sorts;
  val ty = Type (tyco, map TFree vs);
  val lhs = Const (@{const-name typerep}, Term.itselfT ty --\> @{typ typerep})
    $ Free (T, Term.itselfT ty);
  val rhs = @{term Typerep} $ HOLogic.mk-list @{typ typerep}
    (map (HOLogic.mk-typerep o TFree) vs);
  val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
in
  thy
  |> Class.instantiation ([tyco], vs, @{sort typerep})
  |> 'fn thy => Syntax.check-term thy eq
  |-> (fn eq => Specification.definition (NONE, (Attrib.empty-binding, eq)))
  |> snd
  |> Class.prove-instantiation-exit (K (Class.intro-classes-tac []))
  end;

fun ensure-typerep tyco thy = 
if not (Sorts.has-instance (Sign.classes-of thy) tyco @{sort typerep})
  andalso Sorts.has-instance (Sign.classes-of thy) tyco @{sort type}
then add-typerep tyco thy else thy;
in
add-typerep @{type-name fun}
  |> Typedef.interpretation ensure-typerep
  |> Code.datatype-interpretation (ensure-typerep o fst)
  |> Code.abstype-interpretation (ensure-typerep o fst)
end

lemma [code]:
HOL.equal (Typerep tyco1 tys1) (Typerep tyco2 tys2) \leftrightarrow HOL.equal tyco1 tyco2 \\
\& list-all2 HOL.equal tys1 tys2
by (auto simp add: eq-equal [symmetric] list-all2-eq [symmetric])

lemma [code nbe]:
HOL.equal (x :: typerep) x ←→ True
by (fact equal-refl)

code-printing
  type-construct Typerep → (Eval) Term.typ
| constant Typerep → (Eval) Term.Type/ (-, -)

code-reserved Eval Term

hide-const (open) typerep Typerep

end

73 Predicate: Predicates as enumerations

theory Predicate
imports String
begin

73.1 The type of predicate enumerations (a monad)

datatype 'a pred = Pred 'a ⇒ bool

primrec eval :: 'a pred ⇒ 'a ⇒ bool where
eval-pred: eval (Pred f) = f

lemma Pred-eval [simp]:
Pred (eval x) = x
by (cases x) simp

lemma pred-eqI:
(∀w. eval P w ←→ eval Q w) ⇒ P = Q
by (cases P, cases Q) (auto simp add: fun-eq-iff)

lemma pred-eq-iff:
P = Q ⇒ (∀w. eval P w ←→ eval Q w)
by (simp add: pred-eqI)

instantiation pred :: (type) complete-lattice
begin

definition
P ≤ Q ←→ eval P ≤ eval Q

definition
P < Q ←→ eval P < eval Q
definition
$\bot = \text{Pred} \bot$

lemma eval-bot [simp]:
eval $\bot = \bot$
by (simp add: bot-pred-def)

definition
$\top = \text{Pred} \top$

lemma eval-top [simp]:
eval $\top = \top$
by (simp add: top-pred-def)

definition
$P \cap Q = \text{Pred} (\text{eval } P \cap \text{eval } Q)$

lemma eval-inf [simp]:
eval $(P \cap Q) = \text{eval } P \cap \text{eval } Q$
by (simp add: inf-pred-def)

definition
$P \cup Q = \text{Pred} (\text{eval } P \cup \text{eval } Q)$

lemma eval-sup [simp]:
eval $(P \cup Q) = \text{eval } P \cup \text{eval } Q$
by (simp add: sup-pred-def)

definition
$\bigcap A = \text{Pred} (\text{INFIMUM } A \text{ eval})$

lemma eval-Inf [simp]:
eval $(\bigcap A) = \text{INFIMUM } A \text{ eval}$
by (simp add: Inf-pred-def)

definition
$\bigcup A = \text{Pred} (\text{SUPREMUM } A \text{ eval})$

lemma eval-Sup [simp]:
eval $(\bigcup A) = \text{SUPREMUM } A \text{ eval}$
by (simp add: Sup-pred-def)

instance proof
qed (auto intro: pred-eqI simp add: less-eq-pred-def less-pred-def le-fun-def less-fun-def)
end

lemma eval-INF [simp]:
THEORY "Predicate"

eval (INFIMUM A f) = INFIMUM A (eval o f)
using eval-Inf [of f ' A] by simp

lemma eval-SUP [simp]:
eval (SUPREMUM A f) = SUPREMUM A (eval o f)
using eval-Sup [of f ' A] by simp

instantiation pred :: (type) complete-boolean-algebra
begin

definition − P = Pred (− eval P)

lemma eval-compl [simp]:
eval (− P) = − eval P
by (simp add: uminus-pred-def)

definition P − Q = Pred (eval P − eval Q)

lemma eval-minus [simp]:
eval (P − Q) = eval P − eval Q
by (simp add: minus-pred-def)

instance proof
qed (auto intro!: pred-eqI)
end

definition single :: 'a ⇒ 'a pred where
single x = Pred ((op =) x)

lemma eval-single [simp]:
eval (single x) = (op =) x
by (simp add: single-def)

definition bind :: 'a pred ⇒ ('a ⇒ 'b pred) ⇒ 'b pred (infixl ≫ = 70) where
P ≫ = f = (SUPREMUM {x. eval P x} f)

lemma eval-bind [simp]:
eval (P ≫ = f) = eval (SUPREMUM {x. eval P x} f)
by (simp add: bind-def)

lemma bind-bind:
(P ≫ = Q) ≫ = R = P ≫ = (λx. Q x ≫ = R)
by (rule pred-eqI) auto

lemma bind-single:
P ≫ = single = P
by (rule pred-eqI) auto

lemma single-bind:
single x \implies P = P x
by (rule pred-eqI) auto

lemma bottom-bind:
⊥ \implies P = ⊥
by (rule pred-eqI) auto

lemma sup-bind:
(P \sqcup Q) \implies R = P \implies R \sqcup Q \implies R
by (rule pred-eqI) auto

lemma Sup-bind:
(\bigsqcup A \implies f) = \bigsqcup ((\lambda x. \implies f) \ ' A)
by (rule pred-eqI) auto

lemma pred-iffI:
assumes \bigwedge x. \eval A x = \implies \eval B x
and \bigwedge x. \eval B x = \implies \eval A x
shows A = B
using assms by (auto intro: pred-eqI)

lemma singleI: \eval (single x) x
by simp

lemma singleI-unit: \eval (single ()) x
by simp

lemma singleE: \eval (single x) y \implies (y = x \implies P) \implies P
by simp

lemma singleE': \eval (single x) y \implies (x = y \implies P) \implies P
by simp

lemma bindI: \eval P x \implies \eval (Q x) y \implies \eval (P \implies Q) y
by auto

lemma bindE: \eval (R \implies Q) y \implies (\bigwedge x. \eval R x \implies \eval (Q x) y \implies P) \implies P
by auto

lemma botE: \eval \bot x \implies P
by auto

lemma supI1: \eval A x \implies \eval (A \sqcup B) x
by auto
lemma supI2: eval B x \implies eval (A \sqcup B) x 
by auto

lemma supE: eval (A \sqcup B) x \implies (eval A x \implies P) \implies (eval B x \implies P) \implies P 
by auto

lemma single-not-bot [simp]:
  single x \neq \bot
  by (auto simp add: single-def bot-pred-def fun-eq-iff)

lemma not-bot:
  assumes A \neq \bot
  obtains x where eval A x 
  using assms by (cases A) (auto simp add: bot-pred-def)

\subsection*{73.2 Emptiness check and definite choice}

definition is-empty :: 'a pred \Rightarrow bool where 
is-empty A \iff A = \bot

lemma is-empty-bot:
  is-empty \bot 
  by (simp add: is-empty-def)

lemma not-is-empty-single:
  \neg is-empty (single x) 
  by (auto simp add: is-empty-def single-def bot-pred-def fun-eq-iff)

lemma is-empty-sup:
  is-empty (A \sqcup B) \iff is-empty A \land is-empty B 
  by (auto simp add: is-empty-def)

definition singleton :: (unit \Rightarrow 'a) \Rightarrow 'a pred \Rightarrow 'a where 
singleton dfault A = (if \exists! x. eval A x then THE x. eval A x else dfault ())

lemma singleton-eqI:
  \exists! x. eval A x \implies eval A x \implies singleton dfault A = x 
  by (auto simp add: singleton-def)

lemma eval-singletonI:
  \exists! x. eval A x \implies eval A (singleton dfault A) 
proof
  assume asm: \exists! x. eval A x 
  then obtain x where x: eval A x .. 
  with asm have singleton dfault A = x by (rule singleton-eqI) 
  with x show \?thesis by simp 
qed

lemma single-singleton:
\[ \exists x. \text{eval } A x \implies \text{single } (\text{singleton dfault } A) = A \]

**proof**

- assume \( \text{assm: } \exists x. \text{eval } A x \)
- then have \( \text{eval } A (\text{singleton dfault } A) \)
  - by (rule eval-singletonI)
- moreover from \( \text{assm} \)
  - have \( \forall x. \text{eval } A x \implies \text{singleton dfault } A = x \)
  - by (rule singleton-eqI)
- ultimately have \( \text{eval } A (\text{singleton dfault } A) = \text{eval } A x \)
  - by (simp (no-asn-use) add: single-def fun-eq-iff blast)
- then have \( \forall x. \text{eval } (\text{single } (\text{singleton dfault } A)) x = \text{eval } A x \)
  - by simp
- then show \(?\text{thesis}\) by (rule pred-eqI)

qed

**lemma** singleton-undefinedI:

\( \neg (\exists x. \text{eval } A x) \implies \text{singleton dfault } A = \text{dfault } () \)

by (simp add: singleton-def)

**lemma** singleton-bot:

\( \text{singleton dfault } \bot = \text{dfault } () \)

by (auto simp add: bot-pred-def intro: singleton-undefinedI)

**lemma** singleton-single:

\( \text{singleton dfault } (\text{single } x) = x \)

by (auto simp add: intro: singleton-eqI singleI elim: singleE)

**lemma** singleton-sup-single-single:

\( \text{singleton dfault } (x \sqcup y) = (\text{if } x = y \text{ then } x \text{ else } \text{dfault } ()) \)

**proof**

- cases \( \exists x. \text{eval } A x \)
  - case True then show \(?\text{thesis}\) by (simp add: singleton-single)
  - next
  - case False
  - have \( \text{eval } (x \sqcup y) x \)
    - and \( \text{eval } (x \sqcup y) y \)
    - by (auto intro: supI1 supI2 singleI)
  - with \( \neg (\exists x. \text{eval } A x) \)
    - by blast
  - then have \( \text{singleton dfault } (x \sqcup y) = \text{dfault } () \)
    - by (rule singleton-undefinedI)
  - with \( \neg (\exists x. \text{eval } A x) \)
    - by simp
  - qed

**lemma** singleton-sup-aux:

\( \text{singleton dfault } (A \sqcup B) = (\text{if } A = \bot \text{ then } \text{singleton dfault } B \)
  - else if \( B = \bot \text{ then } \text{singleton dfault } A \)
  - else \( \text{singleton dfault } (\text{single } (\text{singleton dfault } A) \sqcup \text{single } (\text{singleton dfault } B)) \)

**proof**

- cases \( \exists x. \text{eval } A x \land \exists y. \text{eval } B y \)
  - case True then show \(?\text{thesis}\) by (simp add: singleton-single)
next
case False
from False have A-or-B:
  singleton dfault A = dfault () ∨ singleton dfault B = dfault ()
  by (auto intro!: singleton-undefinedI)
then have rhs: singleton dfault
  (single (singleton dfault A) ⊔ single (singleton dfault B)) = dfault ()
  by (auto simp add: singleton-sup-single-single singleton-single)
from False have not-unique:
  ¬ (∃!x. eval A x) ∨ ¬ (∃!y. eval B y) by simp
show ?thesis proof (cases A ≠ ⊥ ∧ B ≠ ⊥)
case True
then obtain a b where a: eval A a and b: eval B b
  by (blast elim: not-bot)
with True not-unique have ¬ (∃!x. eval (A ⊔ B) x)
  by (auto simp add: sup-pred-def bot-pred-def)
then have singleton dfault (A ⊔ B) = dfault () by (rule singleton-undefinedI)
  with True rhs show ?thesis by simp
next
case False then show ?thesis by auto
qed
qed

lemma singleton-sup:
  singleton dfault (A ⊔ B) = (if A = ⊥ then singleton dfault B
  else if B = ⊥ then singleton dfault A
  else if singleton dfault A = singleton dfault B then singleton dfault A else dfault ()
  ()
using singleton-sup-aux [of dfault A B] by (simp only: singleton-sup-single-single)

73.3 Derived operations

definition if-pred :: bool ⇒ unit pred where
  if-pred-eq: if-pred b = (if b then single () else ⊥)
definition holds :: unit pred ⇒ bool where
  holds-eq: holds P = eval P ()
definition not-pred :: unit pred ⇒ unit pred where
  not-pred-eq: not-pred P = (if eval P () then ⊥ else single ())
lemma if-predI: P ⇒ eval (if-pred P) ()
  unfolding if-pred-eq by (auto intro: singleI)
lemma if-predE: eval (if-pred b) x ⇒ (b ⇒ x = () ⇒ P) ⇒ P
  unfolding if-pred-eq by (cases b) (auto elim: botE)
lemma not-predI: ¬ P ⇒ eval (not-pred (Pred (λu. P))) ()
  unfolding not-pred-eq eval-pred by (auto intro: singleI)
lemma not-predI': \( \neg \text{eval } P () \Rightarrow \text{eval} (\neg \text{pred } P) () \)
unfolding not-pred-eq by (auto intro: singleI)

lemma not-predE': eval (\neg \text{pred} (\text{Pred} (\lambda u. P))) x \Rightarrow (\neg P \Rightarrow \text{thesis}) \Rightarrow \text{thesis}
unfolding not-pred-eq
by (auto split: split-if-asm elim: botE)

lemma f () = False \lor f () = True
by simp

lemma closure-of-bool-cases [no-atp]:
fixes f :: unit \Rightarrow bool
assumes f = (\lambda u. False) \Rightarrow P f
assumes f = (\lambda u. True) \Rightarrow P f
shows P f
proof -
  have f = (\lambda u. False) \lor f = (\lambda u. True)
    apply (cases f ()
    apply (rule disjI2)
    apply (rule ext)
    apply (simp add: unit-eq)
    apply (rule disjI1)
    apply (rule ext)
    apply (simp add: unit-eq)
    done
  from this assms show ?thesis by blast
qed

lemma unit-pred-cases:
assumes P \bot
assumes P (single ())
shows P Q
using assms unfolding bot-pred-def bot-fun-def bot-bool-def empty-def single-def
proof (cases Q)
  fix f
  assume P (Pred (\lambda u. False)) P (Pred (\lambda u. () = u))
  then have P (Pred f)
    by (cases - f rule: closure-of-bool-cases) simp-all
  moreover assume Q = Pred f
  ultimately show P Q by simp
qed

lemma holds-if-pred:
holds (if-pred b) = b
proof
  case Empty show \var{case}
  by (auto simp add: fun-eq-iff elim: botE)
next
  case Insert show \var{case}

73.4 Implementation

datatype \('a seq = Empty \mid Insert \ 'a \ 'a pred \mid Join \ 'a pred \ 'a seq

primrec pred-of-seq :: \('a seq \Rightarrow \ 'a pred
  where
  pred-of-seq Empty = ⊥
  | pred-of-seq (Insert x P) = single x ⊔ P
  | pred-of-seq (Join P xq) = P ⊔ pred-of-seq xq

definition Seq :: (unit ⇒ \('a seq) ⇒ \('a pred
  where
  Seq f = pred-of-seq (f ())

code-datatype Seq

primrec member :: \('a seq ⇒ \('a ⇒ bool
  where
  member Empty x \leftrightarrow False
  | member (Insert y P) x \leftrightarrow x = y \∨ eval P x
  | member (Join P xq) x \leftrightarrow eval P x \∨ member xq x

lemma eval-member:
  member xq = eval (pred-of-seq xq)
proof (induct xq)
  case Empty show \var{case}
  by (auto simp add: fun-eq-iff elim: botE)
next
  case Insert show \var{case}
by (auto simp add: fun-eq-iff elim: supE singleE intro: supI1 supI2 singleI)

next
  case Join then show ?case
  by (auto simp add: fun-eq-iff elim: supE intro: supI1 supI2)

qed

lemma eval-code [code]: eval (Seq f) = member (f ())
  unfolding Seq-def by (rule sym, rule eval-member)

lemma single-code [code]:
  single x = Seq (λu. Insert x ⊥)
  unfolding Seq-def by simp

primrec apply :: ('a ⇒ 'b pred) ⇒ 'a seq ⇒ 'b seq where
  apply f Empty = Empty
| apply f (Insert x P) = Join (f x) (Join (P ≻= f) Empty)
| apply f (Join P xq) = Join (P ≻= f) (apply f xq)

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq ≻= f
  proof (induct xq)
    case Empty show ?case
    by (simp add: bottom-bind)
  next
    case Insert show ?case
    by (simp add: single-bind sup-bind)
  next
    case Join then show ?case
    by (simp add: sup-bind)
  qed

lemma bind-code [code]:
  Seq g ≻= f = Seq (λu. apply f (g ()))
  unfolding Seq-def by (rule sym, rule apply-bind)

lemma bot-set-code [code]:
  ⊥ = Seq (λu. Empty)
  unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
  adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q ⊔ P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P ⊔ pred-of-seq xq
  by (induct xq) (simp-all add: sup-assoc sup-commute sup-left-commute)

lemma sup-code [code]:
THEORY "Predicate"

**Seq f ⊔ Seq g = Seq (λu. case f ()
of Empty ⇒ g ()
| Insert x P ⇒ Insert x (P ⊔ Seq g)
| Join P xq ⇒ adjunct (Seq g) (Join P xq))**

**proof** (cases f ())
case Empty
thus ?thesis
  unfolding Seq-def by (simp add: sup-commute [of ⊥])
next
case Insert
thus ?thesis
  unfolding Seq-def by (simp add: sup-assoc)
next
case Join
thus ?thesis
  unfolding Seq-def
  by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

**lemma** [code]:
  size (P :: 'a Predicate.pred) = 0 by (cases P) simp

**lemma** [code]:
  size-pred f P = 0 by (cases P) simp

**primrec** contained :: 'a seq ⇒ 'a pred ⇒ bool where
  contained Empty Q ←→ True
| contained (Insert x P) Q ←→ eval Q x ∧ P ≤ Q
| contained (Join P xq) Q ←→ P ≤ Q ∧ contained xq Q

**lemma** single-less-eq-eval:
  single x ≤ P ←→ eval P x
by (auto simp add: less-eq-pred-def le-fun-def)

**lemma** contained-less-eq:
  contained xq Q ←→ pred-of-seq xq ≤ Q
by (induct xq) (simp-all add: single-less-eq-eval)

**lemma** less-eq-pred-code [code]:
  Seq f ≤ Q = (case f ()
of Empty ⇒ True
| Insert x P ⇒ eval Q x ∧ P ≤ Q
| Join P xq ⇒ P ≤ Q ∧ contained xq Q)
by (cases f ())
  (simp-all add: Seq-def single-less-eq-eval contained-less-eq)

**lemma** eq-pred-code [code]:
  fixes P Q :: 'a pred
  shows HOL.equal P Q ←→ P ≤ Q ∧ Q ≤ P
by (auto simp add: equal)

lemma [code nbe]:
  HOL.equal (x :: 'a pred) x \iff True
by (fact equal-refl)

lemma [code]:
  case-pred f P = f (eval P)
by (cases P) simp

lemma [code]:
  rec-pred f P = f (eval P)
by (cases P) simp

inductive eq :: 'a \Rightarrow 'a \Rightarrow bool where eq x x

lemma eq-is-eq:
  eq x y \iff (x = y)
by (rule eq-reflection) (auto intro: eq.intros elim: eq.cases)

primrec null :: 'a seq \Rightarrow bool where
  null Empty \iff True
  | null (Insert x P) \iff False
  | null (Join P xq) \iff is-empty P \land null xq

lemma null-is-empty:
  null xq \iff is-empty (pred-of-seq xq)
by (induct xq) simp (simp add: is-empty-bot not-is-empty-single is-empty-sup)

lemma is-empty-code [code]:
  is-empty (Seq f) \iff null (f ( ))
by (simp add: null-is-empty Seq-def)

primrec the-only :: (unit \Rightarrow 'a) \Rightarrow 'a seq \Rightarrow 'a where
  [code del]: the-only dfault Empty = dfault ( )
  | the-only dfault (Insert x P) = (if is-empty P then x else let y = singleton dfault P in if x = y then x else dfault ( ))
  | the-only dfault (Join P xq) = (if is-empty P then the-only dfault xq else if null xq then singleton dfault P
  else let x = singleton dfault P; y = the-only dfault xq in if x = y then x else dfault ( ))

lemma the-only-singleton:
  the-only dfault xq = singleton dfault (pred-of-seq xq)
by (induct xq)
  (auto simp add: singleton-bot singleton-single is-empty-def null-is-empty Let-def singleton-sup)

lemma singleton-code [code]:
  singleton dfault (Seq f) = (case f ( )
of Empty ⇒ dfault ()
| Insert x P ⇒ if is-empty P then x
else let y = singleton dfault P in
if x = y then x else dfault ()
| Join P xq ⇒ if is-empty P then the-only dfault xq
else if null xq then singleton dfault P
else let x = singleton dfault P; y = the-only dfault xq in
if x = y then x else dfault ()

by (cases f ())
(auto simp add: Seq-def the-only-singleton is-empty-def
null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition the :: 'a pred ⇒ 'a where
the A = (THE x. eval A x)

lemma the-eqI:
(THE x. eval P x) = x =⇒ the P = x
by (simp add: the-def)

lemma the-eq [code]: the A = singleton (λx. Code.abort (STR "not-unique") (λ-. the A)) A
by (rule the-eqI) (simp add: singleton-def the-def)

code-reflect Predicate
datatypes pred = Seq and seq = Empty | Insert | Join

ML ⟨⟨
signature PREDICATE =
sig
  val anamorph: ('a ⇒ ('b ⇒ 'a option) ⇒ int ⇒ 'a ⇒ 'b list * 'a)
datatype 'a pred = Seq of (unit ⇒ 'a seq)
and 'a seq = Empty | Insert of ('a * 'a pred) | Join of 'a pred * 'a seq
val map: ('a ⇒ 'b) ⇒ 'a pred ⇒ 'b pred
val yield: 'a pred ⇒ ('a * 'a pred) option
val yieldn: int ⇒ 'a pred ⇒ 'a list * 'a pred
end;

structure Predicate : PREDICATE =
struct

fun anamorph f k x =
(if k = 0 then ([], x)
else case f x
  of NONE => ([], x)
| SOME (v, y) => let
val k' = k - 1;
val (vs, z) = anamorph f k' y
in (v :: vs, z) end);
THEORY "Predicate"

datatype pred = datatype Predicate.pred
datatype seq = datatype Predicate.seq

fun map f = @ {code Predicate.map} f;

fun yield (Seq f) = next (f ());
and next Empty = NONE
  | next (Insert (x, P)) = SOME (x, P)
  | next (Join (P, xq)) = (case yield P
          of NONE => next xq
          | SOME (x, Q) => SOME (x, Seq (fn => Join (Q, xq))));

fun yieldn k = anamorph yield k;

end;

Conversion from and to sets

definition pred-of-set :: 'a set ⇒ 'a pred
where pred-of-set = Pred ◦ (λA x. x ∈ A)

lemma eval-pred-of-set [simp]:
  eval (pred-of-set A) x ←→ x ∈ A
by (simp add: pred-of-set-def)

definition set-of-pred :: 'a pred ⇒ 'a set
where set-of-pred = Collect ◦ eval

lemma member-set-of-pred [simp]:
  x ∈ set-of-pred P ←→ Predicate.eval P x
by (simp add: set-of-pred-def)

definition set-of-seq :: 'a seq ⇒ 'a set
where set-of-seq = set-of-pred ◦ pred-of-seq

lemma member-set-of-seq [simp]:
  x ∈ set-of-seq xq = Predicate.member xq x
by (simp add: set-of-seq-def eval-member)

lemma of-pred-code [code]:
  set-of-pred (Predicate.Seq f) = (case f () of
      Predicate.Empty => {}
      | Predicate.Insert x P => insert x (set-of-pred P)
by (auto split: seq.split simp add: eval-code)

lemma of-seq-code [code]:
  set-of-seq Predicate.Empty = {}
set-of-seq (Predicate.Insert x P) = insert x (set-of-pred P)
set-of-seq (Predicate.Join P xq) = set-of-pred P \cup set-of-seq xq
by auto

Lazy Evaluation of an indexed function

function iterate-upto :: (natural \Rightarrow 'a) \Rightarrow natural \Rightarrow natural \Rightarrow 'a Predicate.pred
where
iterate-upto f n m =
  Predicate.Seq (%u. if n > m then Predicate.Empty
    else Predicate.Insert (f n) (iterate-upto f (n + 1) m))
by pat-completeness auto

termination by (relation measure (%(f, n, m). nat-of-natural (m + 1 - n)))
(auto simp add: less-natural-def)

Misc

declare Inf-set-fold [where 'a = 'a Predicate.pred, code]
declare Sup-set-fold [where 'a = 'a Predicate.pred, code]

lemma pred-of-set-fold-sup:
  assumes finite A
  shows pred-of-set A = Finite-Set.fold sup bot (Predicate.single ' A) (is ?lhs = ?rhs)
proof (rule sgm)
  interpret comp-fun-idem sup :: 'a Predicate.pred \Rightarrow 'a Predicate.pred \Rightarrow 'a Predicate.pred
  by (fact comp-fun-idem-sup)
  from ⟨finite A⟩ show ?thesis by (induct A) (auto intro!: pred-eqI)
qed

lemma pred-of-set-set-fold-sup:
  pred-of-set (set xs) = fold sup (List.map Predicate.single xs) bot
proof –
  interpret comp-fun-idem sup :: 'a Predicate.pred \Rightarrow 'a Predicate.pred \Rightarrow 'a Predicate.pred
  by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: pred-of-set-fold-sup fold-set-fold [symmetric])
qed

lemma pred-of-set-set-foldr-sup [code]:
  pred-of-set (set xs) = foldr sup (List.map Predicate.single xs) bot
by (simp add: pred-of-set-set-fold-sup ac-simps foldr-fold fun-eq-iff)

no-notation
bind (infixl >= 70)

hide-type (open) pred seq
hide-const (open) Pred eval single bind is-empty singleton if-pred not-pred holds
Lazy-Sequence: Lazy sequences

theory Lazy-Sequence
imports Predicate
begin

74.1 Type of lazy sequences
datatype 'a lazy-sequence = lazy-sequence-of-list 'a list

primrec list-of-lazy-sequence :: 'a lazy-sequence ⇒ 'a list
where
  list-of-lazy-sequence (lazy-sequence-of-list xs) = xs

lemma lazy-sequence-of-list-of-lazy-sequence [simp]:
  lazy-sequence-of-list (list-of-lazy-sequence xq) = xq
  by (cases xq) simp-all

lemma lazy-sequence-eqI:
  list-of-lazy-sequence xq = list-of-lazy-sequence yq ⇒ xq = yq
  by (cases xq, cases yq) simp

lemma lazy-sequence-eq-iff:
  xq = yq ⇐⇒ list-of-lazy-sequence xq = list-of-lazy-sequence yq
  by (auto intro: lazy-sequence-eqI)

lemma size-lazy-sequence-eq [code]:
  size-lazy-sequence f xq = Suc (size-list f (list-of-lazy-sequence xq))
  size (xq :: 'a lazy-sequence) = 0
  by (cases xq, simp)+

lemma case-lazy-sequence [simp]:
  case-lazy-sequence f xq = f (list-of-lazy-sequence xq)
  by (cases xq) auto

lemma rec-lazy-sequence [simp]:
  rec-lazy-sequence f xq = f (list-of-lazy-sequence xq)
  by (cases xq) auto

definition Lazy-Sequence :: (unit ⇒ ('a × 'a lazy-sequence) option) ⇒ 'a lazy-sequence
where
  Lazy-Sequence f = lazy-sequence-of-list (case f () of
    None ⇒ <>
THEORY "Lazy-Sequence"

| Some (x, xq) ⇒ x ≠ list-of-lazy-sequence xq)

code-datatype Lazy-Sequence

declare list-of-lazy-sequence.simps [code del]
declare lazy-sequence.case [code def]
declare lazy-sequence.rec [code def]

lemma list-of-Lazy-Sequence [simp]:
list-of-lazy-sequence (Lazy-Sequence f) = (case f () of
  None ⇒ []
| Some (x, xq) ⇒ x # list-of-lazy-sequence xq)
by (simp add: Lazy-Sequence-def)

definition yield :: 'a lazy-sequence ⇒ ('a × 'a lazy-sequence) option
where
  yield xq = (case list-of-lazy-sequence xq of
    [] ⇒ None
| x ≠ xs ⇒ Some (x, lazy-sequence-of-list xs))

lemma yield-Seq [simp, code]:
yield (Lazy-Sequence f) = f ()
by (cases f ()) (simp-all add: yield-def split-def)

lemma case-yield-eq [simp]: case-option g h (yield xq) =
case-list g (λx. curry h x o lazy-sequence-of-list) (list-of-lazy-sequence xq)
by (cases list-of-lazy-sequence xq) (simp-all add: yield-def)

lemma size-lazy-sequence-code [code]:
size-lazy-sequence s xq = (case yield xq of
  None ⇒ 1
| Some (x, xq′) ⇒ Suc (s x + size-lazy-sequence s xq'))
by (cases list-of-lazy-sequence xq) (simp-all add: size-lazy-sequence-eq)

lemma equal-lazy-sequence-code [code]:
HOL.equal xq yq = (case (yield xq, yield yq) of
  (None, None) ⇒ True
| (Some (x, xq'), Some (y, yq')) ⇒ HOL.equal x y ∧ HOL.equal xq yq
| _ ⇒ False)
by (simp-all add: lazy-sequence-eq-iff equal-eq split: list.splits)

lemma [code nbe]:
HOL.equal (x :: 'a lazy-sequence) x ←→ True
by (fact equal-refl)

definition empty :: 'a lazy-sequence
where
  empty = lazy-sequence-of-list []
THEORY "Lazy-Sequence"

lemma list-of-lazy-sequence-empty [simp]:
  list-of-lazy-sequence empty = []
by (simp add: empty-def)

lemma empty-code [code]:
  empty = Lazy-Sequence (λ-. None)
by (simp add: lazy-sequence-eq-iff)

definition single :: 'a ⇒ 'a lazy-sequence
where
  single x = lazy-sequence-of-list [x]

lemma list-of-lazy-sequence-single [simp]:
  list-of-lazy-sequence (single x) = [x]
by (simp add: single-def)

lemma single-code [code]:
  single x = Lazy-Sequence (λ-. Some (x, empty))
by (simp add: lazy-sequence-eq-iff)

definition append :: 'a lazy-sequence ⇒ 'a lazy-sequence ⇒ 'a lazy-sequence
where
  append xq yq = lazy-sequence-of-list (list-of-lazy-sequence xq @ list-of-lazy-sequence yq)

lemma list-of-lazy-sequence-append [simp]:
  list-of-lazy-sequence (append xq yq) = list-of-lazy-sequence xq @ list-of-lazy-sequence yq
by (simp add: append-def)

lemma append-code [code]:
  append xq yq = Lazy-Sequence (λ-. case yield xq of
    None ⇒ yield yq
    | Some (x, xq') ⇒ Some (x, append xq' yq))
by (simp-all add: lazy-sequence-eq-iff split: list.splits)

definition map :: ('a ⇒ 'b) ⇒ 'a lazy-sequence ⇒ 'b lazy-sequence
where
  map f xq = lazy-sequence-of-list (List.map f (list-of-lazy-sequence xq))

lemma list-of-lazy-sequence-map [simp]:
  list-of-lazy-sequence (map f xq) = List.map f (list-of-lazy-sequence xq)
by (simp add: map-def)

lemma map-code [code]:
  map f xq =
    Lazy-Sequence (λ-. map-option (λ(x, xq'). (f x, map f xq')) (yield xq))
by (simp-all add: lazy-sequence-eq-iff split: list.splits)
**THEORY “Lazy-Sequence”**

definition flat :: 'a lazy-sequence lazy-sequence ⇒ 'a lazy-sequence
  where
  flat xqq = lazy-sequence-of-list (concat (List.map list-of-lazy-sequence (list-of-lazy-sequence xqq)))

lemma list-of-lazy-sequence-flat [simp]:
  list-of-lazy-sequence (flat xqq) = concat (List.map list-of-lazy-sequence (list-of-lazy-sequence xqq))
  by (simp add: flat-def)

lemma flat-code [code]:
  flat xqq = Lazy-Sequence (λ-. case yield xqq of
    None ⇒ None
  | Some (xq, xqq') ⇒ yield (append xq (flat xqq')))
  by (simp add: lazy-sequence-eq-iff split: list.splits)

definition bind :: 'a lazy-sequence ⇒ ('a ⇒ 'b lazy-sequence) ⇒ 'b lazy-sequence
  where
  bind xq f = flat (map f xq)

definition if-seq :: bool ⇒ unit lazy-sequence
  where
  if-seq b = (if b then single () else empty)

definition those :: 'a option lazy-sequence ⇒ 'a lazy-sequence option
  where
  those xq = map-option lazy-sequence-of-list (List.those (list-of-lazy-sequence xq))

function iterate upto :: (natural ⇒ 'a ⇒ natural ⇒ natural ⇒ 'a lazy-sequence)
  where
  iterate upto f n m =
    Lazy-Sequence (λ-. if n > m then None else Some (f n, iterate upto f (n + 1) m))
  by pat-completeness auto

termination by (relation measure (λ(f, n, m). nat-of-natural (m + 1 − n)))
  (auto simp add: less-natural-def)

definition not-seq :: unit lazy-sequence ⇒ unit lazy-sequence
  where
  not-seq xq = (case yield xq of
    None ⇒ single ()
  | Some (()), xq ⇒ empty)

74.2 Code setup

code-reflect Lazy-Sequence
datatypes lazy-sequence = Lazy-Sequence
ML \ll
signature LAZY-SEQUENCE =
  sig
datatype 'a lazy-sequence = Lazy-Sequence of (unit -> ('a \times 'a Lazy-Sequence.lazy-sequence)
  option)
  val map: ('a -> 'b) -> 'a lazy-sequence -> 'b lazy-sequence
  val yield: 'a lazy-sequence -> ('a \times 'a lazy-sequence) option
  val yieldn: int -> 'a lazy-sequence -> 'a list * 'a lazy-sequence
end;
structure Lazy-Sequence : LAZY-SEQUENCE =
  struct
datatype lazy-sequence = datatype Lazy-Sequence.lazy-sequence;
  fun map f = @{code Lazy-Sequence.map} f;
  fun yield P = @{code Lazy-Sequence.yield} P;
  fun yieldn k = Predicate.anamorph yield k;
end
\rr

74.3 Generator Sequences

74.3.1 General lazy sequence operation

definition product :: 'a lazy-sequence \Rightarrow 'b lazy-sequence \Rightarrow ('a \times 'b) lazy-sequence
where
  product s1 s2 = bind s1 (\lambda a. bind s2 (\lambda b. single (a, b)))

74.3.2 Small lazy typeclasses

class small-lazy =
  fixes small-lazy :: natural \Rightarrow 'a lazy-sequence

instantiation unit :: small-lazy
begin
definition small-lazy d = single ()
instances ..
end

instantiation int :: small-lazy
begin
maybe optimise this expression - insecure append (single x) xs == cons x xs Per-
formance difference?

```plaintext
function small-lazy' :: int ⇒ int ⇒ int lazy-sequence
where
  small-lazy' d i = (if d < i then empty
    else append (single i) (small-lazy' d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (%(d, i). nat (d + 1 - i))) auto

definition
  small-lazy d = small-lazy' (int (nat-of-natural d)) (- (int (nat-of-natural d)))

instance ..
end

instatiation prod :: (small-lazy, small-lazy) small-lazy
begin

definition
  small-lazy d = product (small-lazy d) (small-lazy d)

instance ..
end

instatiation list :: (small-lazy) small-lazy
begin

fun small-lazy-list :: natural ⇒ 'a list lazy-sequence
where
  small-lazy-list d = append (single [])
    (if d > 0 then bind (product (small-lazy (d - 1)))
      (small-lazy (d - 1))) (λ(x, xs). single (x # xs)) else empty)

instance ..
end

74.4 With Hit Bound Value

assuming in negative context

type-synonym 'a hit-bound-lazy-sequence = 'a option lazy-sequence

definition hit-bound :: 'a hit-bound-lazy-sequence
where
  hit-bound = Lazy-Sequence (λ-. Some (None, empty))
```
lemma list-of-lazy-sequence-hit-bound [simp]:
  list-of-lazy-sequence hit-bound = [None]
by (simp add: hit-bound-def)

definition hb-single :: 'a ⇒ 'a hit-bound-lazy-sequence
where
  hb-single x = Lazy-Sequence (λ-. Some (Some x, empty))

definition hb-map :: ('a ⇒ 'b) ⇒ 'a hit-bound-lazy-sequence ⇒ 'b hit-bound-lazy-sequence
where
  hb-map f xq = map (map-option f) xq

lemma hb-map-code [code]:
  hb-map f xq =
  Lazy-Sequence (λ-. map-option (λ(x, xq'). (map-option f x, hb-map f xq'))) (yield xq)
using map-code [of map-option f xq] by (simp add: hb-map-def)

definition hb-flat :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence ⇒ 'a hit-bound-lazy-sequence
where
  hb-flat xqq = lazy-sequence-of-list (concat (List.map ((λx. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq)))

lemma list-of-lazy-sequence-hb-flat [simp]:
  list-of-lazy-sequence (hb-flat xqq) =
  concat (List.map ((λx. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq))
by (simp add: lazy-sequence-eq-iff split: list.splits option.splits)

lemma hb-flat-code [code]:
  hb-flat xqq = Lazy-Sequence (λ-. case yield xqq of
    None ⇒ None
    | Some (xq, xqq') ⇒ yield
    (append (case xq of None ⇒ hit-bound | Some xq ⇒ xq) (hb-flat xqq'))) 
by (simp add: lazy-sequence-eq-iff split: list.splits option.splits)

definition hb-bind :: 'a hit-bound-lazy-sequence ⇒ ('a ⇒ 'b hit-bound-lazy-sequence) ⇒ 'b hit-bound-lazy-sequence
where
  hb-bind xq f = hb-flat (hb-map f xq)

definition hb-if-seq :: bool ⇒ unit hit-bound-lazy-sequence
where
  hb-if-seq b = (if b then hb-single () else empty)

definition hb-not-seq :: unit hit-bound-lazy-sequence ⇒ unit lazy-sequence
where
hide-const (open) yield empty single append flat map bind
if-seq those iterate-upto not-seq product

hide-fact (open) yield-def empty-def single-def append-def flat-def map-def bind-def
if-seq-def those-def not-seq-def product-def

end

75 Limited-Sequence: Depth-Limited Sequences with failure element

theory Limited-Sequence imports Lazy-Sequence begin

75.1 Depth-Limited Sequence

type-synonym 'a dseq = natural ⇒ bool ⇒ 'a lazy-sequence option

definition empty :: 'a dseq
where
  empty = (λ- -. Some Lazy-Sequence.empty)

definition single :: 'a ⇒ 'a dseq
where
  single x = (λ- -. Some (Lazy-Sequence.single x))

definition eval :: 'a dseq ⇒ natural ⇒ bool ⇒ 'a lazy-sequence option
where
  [simp]: eval f i pol = f i pol

definition yield :: 'a dseq ⇒ natural ⇒ bool ⇒ ('a × 'a dseq) option
where
  yield f i pol = (case eval f i pol of
  None ⇒ None
  | Some s ⇒ (map-option o apsnd) (λx - -. Some r) (Lazy-Sequence.yield s))

definition map-seq :: ('a ⇒ 'b dseq) ⇒ 'a lazy-sequence ⇒ 'b dseq
where
  map-seq f xq i pol = map-option Lazy-Sequence.flat
  (Lazy-Sequence.those (Lazy-Sequence.map (λx. f x i pol) xq))

lemma map-seq-code [code]:
  map-seq f xq i pol = (case Lazy-Sequence.yield xq of
There are several definitions related to sequences in this theory.

- **Definition**: `bind :: 'a dseq ⇒ ('a ⇒ 'b dseq) ⇒ 'b dseq`
  - Where `bind x f = (λi pol.
    if i = 0 then
      (if pol then Some Lazy-Sequence.empty else None)
    else
      (case x (i - 1) pol of
        None ⇒ None
      | Some xq ⇒ map-seq f xq i pol))`

- **Definition**: `union :: 'a dseq ⇒ 'a dseq ⇒ 'a dseq`
  - Where `union x y = (λi pol. case (x i pol, y i pol) of
    (Some xq, Some yq) ⇒ Some (Lazy-Sequence.append xq yq)
  | - ⇒ None)`

- **Definition**: `if-seq :: bool ⇒ unit dseq`
  - Where `if-seq b = (if b then single () else empty)`

- **Definition**: `not-seq :: unit dseq ⇒ unit dseq`
  - Where `not-seq x = (λi pol. case x i (¬ pol) of
    None ⇒ Some Lazy-Sequence.empty
  | Some xq ⇒ (case Lazy-Sequence.yield xq of
    None ⇒ Some (Lazy-Sequence.single ()))
  | Some - ⇒ Some (Lazy-Sequence.empty)))`

- **Definition**: `map :: ('a ⇒ 'b) ⇒ 'a dseq ⇒ 'b dseq`
  - Where `map f g = (λi pol. case g i pol of
    None ⇒ None
  | Some xq ⇒ Some (Lazy-Sequence.map f xq))`

### 75.2 Positive Depth-Limited Sequence

**Type-Synonym** `'a pos-dseq = natural ⇒ 'a Lazy-Sequence.lazy-sequence`

**Definition** `pos-empty :: 'a pos-dseq`
Theory "Limited-Sequence"

where

\[ \text{pos-empty} = (\lambda i. \text{Lazy-Sequence}.\text{empty}) \]

definition \text{pos-single} :: 'a \Rightarrow 'a \text{ pos-dseq}
where

\[ \text{pos-single} \ x = (\lambda i. \text{Lazy-Sequence}.\text{single} \ x) \]

definition \text{pos-bind} :: 'a \text{ pos-dseq} \Rightarrow (\text{'a pos-dseq}) \Rightarrow 'b \text{ pos-dseq}
where

\[ \text{pos-bind} \ x \ f = (\lambda i. \text{Lazy-Sequence}.\text{bind} \ (x 

\text{i}) \ (\lambda a. f \ a \ i)) \]

definition \text{pos-decr-bind} :: 'a \text{ pos-dseq} \Rightarrow (\text{'a pos-dseq}) \Rightarrow 'b \text{ pos-dseq}
where

\[ \text{pos-decr-bind} \ x \ f = (\lambda i. \text{if } i = 0 \text{ then}

\text{Lazy-Sequence}.\text{empty}

\text{else}

\text{Lazy-Sequence}.\text{bind} \ (x \ (i - 1)) \ (\lambda a. f \ a \ i)) \]

definition \text{pos-union} :: 'a \text{ pos-dseq} \Rightarrow 'a \text{ pos-dseq} \Rightarrow 'a \text{ pos-dseq}
where

\[ \text{pos-union} \ xq \ yq = (\lambda i. \text{Lazy-Sequence}.\text{append} \ (xq \ i) \ (yq \ i)) \]

definition \text{pos-if-seq} :: \text{bool} \Rightarrow \text{unit pos-dseq}
where

\[ \text{pos-if-seq} \ b = (\text{if } b \text{ then pos-single} \ () \text{ else pos-empty}) \]

definition \text{pos-iterate-upto} :: \text{natural} \Rightarrow 'a \Rightarrow 'a \text{ pos-dseq}
where

\[ \text{pos-iterate-upto} \ f \ n \ m = (\lambda i. \text{Lazy-Sequence}.\text{iterate-upto} \ f \ n \ m) \]

definition \text{pos-map} :: (\text{'a} \Rightarrow 'b) \Rightarrow 'a \text{ pos-dseq} \Rightarrow 'b \text{ pos-dseq}
where

\[ \text{pos-map} \ f \ xq = (\lambda i. \text{Lazy-Sequence}.\text{map} \ f \ (xq \ i)) \]

75.3 Negative Depth-Limited Sequence

type-synonym 'a \text{ neg-dseq} = \text{natural} \Rightarrow 'a \text{ Lazy-Sequence}.\text{hit-bound-lazy-sequence}

definition \text{neg-empty} :: 'a \text{ neg-dseq}
where

\[ \text{neg-empty} = (\lambda i. \text{Lazy-Sequence}.\text{empty}) \]

definition \text{neg-single} :: 'a \Rightarrow 'a \text{ neg-dseq}
where

\[ \text{neg-single} \ x = (\lambda i. \text{Lazy-Sequence}.\text{hb-single} \ x) \]

definition \text{neg-bind} :: 'a \text{ neg-dseq} \Rightarrow ('a \Rightarrow 'b \text{ neg-dseq}) \Rightarrow 'b \text{ neg-dseq}
where
THEORY “Limited-Sequence”

neg-bind \( x f = (\lambda i. \ hb-bind \ (x \ i) \ (\lambda a. f \ a \ i)) \)

**definition** neg-decr-bind :: 'a neg-dseq \( \Rightarrow \) 'a neg-dseq \( \Rightarrow \) 'b neg-dseq
where
\[
\text{neg-decr-bind} \ x \ f = (\lambda i. \ \\
\text{if } i = 0 \text{ then } \text{Lazy-Sequence.hit-bound} \ \\
\text{else } \ hb-bind \ (x \ (i - 1)) \ (\lambda a. f \ a \ i))
\]

**definition** neg-union :: 'a neg-dseq \( \Rightarrow \) 'a neg-dseq \( \Rightarrow \) 'a neg-dseq
where
\[
\text{neg-union} \ x \ y = (\lambda i. \ \text{Lazy-Sequence.append} \ (x \ i) \ (y \ i))
\]

**definition** neg-if-seq :: bool \( \Rightarrow \) unit neg-dseq
where
\[
\text{neg-if-seq} \ b = (\text{if } b \text{ then } \text{neg-single} () \text{ else } \text{neg-empty})
\]

**definition** neg-iterate-upto
where
\[
\text{neg-iterate-upto} \ f \ n \ m = (\lambda i. \ \text{Lazy-Sequence.iterate-upto} \ (\lambda i. \ \text{Some} \ (f \ i)) \ n \ m)
\]

**definition** neg-map :: ('a \( \Rightarrow \) 'b) \( \Rightarrow \) 'a neg-dseq \( \Rightarrow \) 'b neg-dseq
where
\[
\text{neg-map} \ f \ xq = (\lambda i. \ \text{Lazy-Sequence.hb-map} \ f \ (xq \ i))
\]

### 75.4 Negation

**definition** pos-not-seq :: unit neg-dseq \( \Rightarrow \) unit pos-dseq
where
\[
\text{pos-not-seq} \ xq = (\lambda i. \ \text{Lazy-Sequence.hb-not-seq} \ (xq \ (3 \ast i)))
\]

**definition** neg-not-seq :: unit pos-dseq \( \Rightarrow \) unit neg-dseq
where
\[
\text{neg-not-seq} \ x = (\lambda i. \ \text{case Lazy-Sequence.yield} \ (x \ i) \text{ of} \ \\
\text{None } => \text{Lazy-Sequence.hb-single} () \ \\
| \text{Some} ((), \ xq) => \text{Lazy-Sequence.empty})
\]

ML

```ml
signature LIMITED-SEQUENCE =
  sig
  type 'a dseq = Code-Natural.natural \( \Rightarrow \) bool \( \Rightarrow \) 'a Lazy-Sequence.lazy-sequence
  option
  val map : ('a \( \Rightarrow \) 'b) \( \Rightarrow \) 'a dseq \( \Rightarrow \) 'b dseq
  val yield : 'a dseq \( \Rightarrow \) Code-Natural.natural \( \Rightarrow \) bool \( \Rightarrow \) ('a * 'a dseq) option
  val yieldn : int \( \Rightarrow \) 'a dseq \( \Rightarrow \) Code-Natural.natural \( \Rightarrow \) bool \( \Rightarrow \) 'a list * 'a dseq
end;
```
structure Limited-Sequence : LIMITED-SEQUENCE =

struct 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence

option

fun map f = @{code Limited-Sequence.map} f;
fun yield f = @{code Limited-Sequence.yield} f;

fun yieldn n f i pol = (case f i pol of
   NONE => ([], fn - => fn - => NONE)
| SOME s => let val (xs, s') = Lazy-Sequence.yieldn n s in (xs, fn - => fn - => SOME s') end);

end

⟩⟩

code-reserved Eval Limited-Sequence

hide-const (open) yield empty single eval map-seq bind union if-seq not-seq map
   pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto
   pos-not-seq pos-map
   neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-seq neg-iterate-upto
   neg-not-seq neg-map

hide-fact (open) yield-def empty-def single-def eval-def map-seq-def bind-def union-def
   if-seq-def not-seq-def map-def
   pos-not-seq-def pos-map-def
   neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-seq-def neg-iterate-upto-def
   neg-not-seq-def neg-map-def

end

76 Code-Evaluation: Term evaluation using the generic
code generator

theory Code-Evaluation
imports Typerep Limited-Sequence
keywords value :: diag
begin
76.1 Term representation

76.1.1 Terms and class term-of

datatype term = dummy-term

definition Const :: String.literal ⇒ typerep ⇒ term where
Const - - = dummy-term

definition App :: term ⇒ term ⇒ term where
App - - = dummy-term

definition Abs :: String.literal ⇒ typerep ⇒ term ⇒ term where
Abs - - - = dummy-term

definition Free :: String.literal ⇒ typerep ⇒ term where
Free - - = dummy-term

code-datatype Const App Abs Free

class term-of = typerep +
fixes term-of :: 'a ⇒ term

lemma term-of-anything: term-of x ≡ t
by (rule eq-reflection) (cases term-of x, cases t, simp)

definition valapp :: ('a ⇒ 'b) × (unit ⇒ term) ⇒ 'a × (unit ⇒ term) ⇒ 'b × (unit ⇒ term) where
valapp f x = (fst f (fst x), λu. App (snd f ()) (snd x ()))

lemma valapp-code [code, code-unfold]:
valapp (f, tf) (x, tx) = (f x, λu. App (tf ()) (tx ()))
by (simp only: valapp-def fst-conv snd-conv)

76.1.2 Syntax

definition termify :: 'a ⇒ term where
[code del]: termify x = dummy-term

abbreviation valtermify :: 'a ⇒ 'a × (unit ⇒ term) where
valtermify x ≡ (x, λu. termify x)

locale term-syntax
begin

notation App (infixl <::> 70)
and valapp (infixl {·} 70)

end
interpretation term-syntax.

no-notation $App$ (infixl $\leftarrow\rightarrow$ 70)
and $valapp$ (infixl $\{\cdot\}$ 70)

76.2 Tools setup and evaluation

lemma eq-eq-TrueD:
assumes $(x \equiv y) \equiv Trueprop$ True
shows $x \equiv y$
using assms by simp

code-printing
type-constructor term $\rightarrow$ (Eval) Term.term
| constant Const $\rightarrow$ (Eval) Term.Const/ ((-, -))
| constant App $\rightarrow$ (Eval) Term.$\{\cdot\}$/ ((-, -))
| constant Abs $\rightarrow$ (Eval) Term.Abs/ ((-, -), (-), (-))
| constant Free $\rightarrow$ (Eval) Term.Free/ ((-, -), (-))

ML-file Tools/code-evaluation.ML

code-reserved Eval Code-Evaluation

ML-file ~/src/HOL/Tools/value.ML

76.3 term-of instances

instantiation fun :: (typerep, typerep) term-of begin
definition $\text{term-of} \ (f : 'a \Rightarrow 'b) =$
Const (STR "Pure.dummy-pattern")
(Typerep.Typerep (STR "fun") [Typerep.typerep TYPE('a), Typerep.typerep TYPE('b)])

instance ..
end

instantiation String.literal :: term-of begin
definition $\text{term-of} \ s =$ $\text{App} \ (\text{Const} \ (\text{STR} "$\text{str}$")$
(Typerep.Typerep (STR "fun") [Typerep.Typerep (STR "list") [Typerep.Typerep
(STR "char") []],
Typerep.Typerep (STR "String.literal") []]) \ (\text{term-of} \ (\text{String.explode} \ s))$

instance ..
end

declare [[code drop: rec-term case-term size-term size term ⇒ - HOL.equal ::
  term ⇒ -
  term-of :: typerep ⇒ - term-of :: term ⇒ - term-of :: String.literal ⇒ -
  term-of :: - Predicate.pred ⇒ term term-of :: - Predicate.seq ⇒ term]]

lemma term-of-char [unfolded typerep-fun-def typerep-char-def typerep-nibble-def, code]:
  Code-Evaluation.term-of c = (case c of Char x y ⇒
  (Code-Evaluation.Const (STR "String.char.Char") (TYPEREPS(nibble ⇒ nibble ⇒ char))))
  (Code-Evaluation.term-of x)) (Code-Evaluation.term-of y))
by (subst term-of-anything) rule

code-printing
  constant term-of :: integer ⇒ term ⇒ (Eval) HOLLogic.mk'-number| HOLLogic.code'-integerT
  | constant term-of :: String.literal ⇒ term ⇒ (Eval) HOLLogic.mk'-literal

code-reserved Eval HOLLogic

76.4 Generic reification

ML-file ∼~/src/HOL/Tools/reification.ML

76.5 Diagnostic

definition tracing :: String.literal ⇒ 'a ⇒ 'a where
  [code del]: tracing s x = x

code-printing
  constant tracing :: String.literal => 'a => 'a => (Eval) Code'-Evaluation.tracing

hide-const dummy-term valapp
hide-const (open) Const App Abs Free termify valtermify term-of tracing

end

77 Quickcheck-Random: A simple counterexample generator performing random testing
notation \texttt{fcomp} \texttt{(infixl \texttt{\textasciicircum} 60)}
notation \texttt{scomp} \texttt{(infixl \texttt{\textasciicircum} \rightarrow 60)}

\textbf{setup} $\langle\langle \text{Code-Target.extend-target \ (Quickcheck, \ (Code/Runtime.target, \ I))} \rangle\rangle$

\section{Catching Match exceptions}

axiomatization \texttt{catch-match} :: \texttt{'a => 'a => 'a}

code-printing
constant \texttt{catch-match} \mapsto (Quickcheck) \ ((-) handle Match => -)

\section{The \texttt{random} class}

\texttt{class random = typerep +}
\texttt{fixes random :: natural \Rightarrow Random.seed \Rightarrow ('a \times (unit \Rightarrow \text{term})) \times Random.seed}

\section{Fundamental and numeric types}

\texttt{instantiation bool :: random}
begin

\texttt{definition random i = Random.range 2 \circ\rightarrow}
\texttt{(\lambda k. Pair (if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True))}

instance ..

end

\texttt{instantiation itself :: (typerep) random}
begin

\texttt{definition random-itself :: natural \Rightarrow Random.seed \Rightarrow ('a itself \times (unit \Rightarrow \text{term})) \times Random.seed}
where \texttt{random-itself - = Pair (Code-Evaluation.valtermify TYPE('a))}

instance ..

end

\texttt{instantiation char :: random}
begin

\texttt{definition random - = Random.select (Enum.enum :: char list) \circ\rightarrow (\lambda c. Pair (c, \lambda u. Code-Evaluation.term-of c))}
THEORY "Quickcheck-Random"

instance ..
end

instantiation String.literal :: random
begin
definition random = Pair (STR "", λu. Code-Evaluation.term-of (STR ""))
instance ..
end

instantiation nat :: random
begin
definition random-nat :: natural ⇒ Random.seed ⇒ (nat × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
random-nat i = Random.range (i + 1) ◦→ (λk. Pair (let n = nat-of-natural k
in (n, λ-. Code-Evaluation.term-of n)))
instance ..
end

instantiation int :: random
begin
definition random i = Random.range (2 * i + 1) ◦→ (λk. Pair (let j = (if k ≥ i then int (nat-of-natural (k - i)) else - (int (nat-of-natural (i - k)))))
in (j, λ-. Code-Evaluation.term-of j)))
instance ..
end

instantiation natural :: random
begin
definition random-natural :: natural ⇒ Random.seed ⇒ (natural × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
random-natural i = Random.range (i + 1) ◦→ (λn. Pair (n, λ-. Code-Evaluation.term-of n))
instance ..
end

instantiation integer :: random
begin
definition random-integer :: natural ⇒ Random.seed ⇒ (integer × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
random-integer i = Random.range (2 * i + 1) o→ (λk. Pair (let j = (if k ≥ i then integer-of-natural (k - i) else - (integer-of-natural (i - k))) in (j, λ-. Code-Evaluation.term-of j)))
instance ..
end

77.4 Complex generators
Towards ’a ⇒ ’b
axiomatization random-fun-aux :: typerep ⇒ typerep ⇒ (’a ⇒ ’a ⇒ bool) ⇒ (’a ⇒ term) ⇒ (Random.seed ⇒ (’b × (unit ⇒ term)) × Random.seed) ⇒ (Random.seed ⇒ (’a ⇒ ’b) × (unit ⇒ term)) × Random.seed
definition random-fun-lift :: (Random.seed ⇒ (’b × (unit ⇒ term)) × Random.seed)
⇒ Random.seed ⇒ ((’a::term-of ⇒ ’b::typerep) × (unit ⇒ term)) × Random.seed
where
random-fun-lift f =
random-fun-aux TYPEREPI’(a) TYPEREPI’(b) (op =) Code-Evaluation.term-of f Random.split-seed
instantiation fun :: ({equal, term-of}, random) random
begin
definition random-fun :: natural ⇒ Random.seed ⇒ ((’a ⇒ ’b) × (unit ⇒ term)) × Random.seed
where random i = random-fun-lift (random i)
instance ..
end
Towards type copies and datatypes

definition collapse :: ('a ⇒ ('a ⇒ 'b × 'a)) ⇒ 'a ⇒ 'b × 'a
  where collapse f = (f o→ id)

definition beyond :: natural ⇒ natural ⇒ natural
  where beyond k l = (if l > k then l else 0)

lemma beyond-zero: beyond k 0 = 0
  by (simp add: beyond-def)

definition (in term-syntax) [code-unfold]:
  valterm-insert x s = Code-Evaluation.valtermify insert x s

definition (in term-syntax) [code-unfold]:
  valtermify-insert x s = Code-Evaluation.valtermify-insert x s

instantiation set :: (random) random
begin

primrec random-aux-set
where
  random-aux-set 0 j = collapse (Random.select-weight [(1, Pair valterm-emptyset)])
| random-aux-set (Code-Numeral.Suc i) j =
  collapse (Random.select-weight [(1, Pair valterm-emptyset),
  (Code-Numeral.Suc i,
  random j o→ (%x. random-aux-set i j o→ (%s. Pair (valtermify-insert x s)))))

lemma [code]:
  random-aux-set i j =
  collapse (Random.select-weight [(1, Pair valterm-emptyset),
  (i, random j o→ (%x. random-aux-set (i − 1) j o→ (%s. Pair (valtermify-insert x s))))])

proof (induct i rule: natural.induct)
  case zero
  show ?case by (subst select-weight-drop-zero [symmetric])
    (simp add: random-aux-set.simps [simplified] less-natural-def)
  next
  case (Suc i)
  show ?case by (simp only: random-aux-set.simps(2) [of i] Suc-natural-minus-one)
qed

definition random-set i = random-aux-set i i

instance ..
lemma random-aux-rec:
  fixes random-aux :: natural ⇒ 'a
  assumes random-aux 0 = rhs 0
  and (k. random-aux (Code-Numerul.Suc k) = rhs (Code-Numerul.Suc k))
  shows random-aux k = rhs k
  using assms by (rule natural.induct)

77.5 Deriving random generators for datatypes

ML-file Tools/Quickcheck/quickcheck-common.ML
ML-file Tools/Quickcheck/random-generators.ML
setup Random-Generators.setup

77.6 Code setup

code-printing
  constant random-fun-aux ⇒ (Quickcheck) Random'-Generators.random'-fun
  — With enough criminal energy this can be abused to derive False; for this
  reason we use a distinguished target Quickcheck not spoiling the regular trusted
  code generation

code-reserved Quickcheck Random-Generators

no-notation fcomp (infixl ◦> 60)
no-notation scomp (infixl ◦→ 60)

hide-const (open) catch-match random collapse beyond random-fun-aux random-fun-lift

hide-fact (open) collapse-def beyond-def random-fun-lift-def

end

78 Random-Pred: The Random-Predicate Monad

theory Random-Pred
imports Quickcheck-Random
begin

fun iter' :: 'a itself ⇒ natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
where
  iter' T nrandom sz seed = (if nrandom = 0 then bot-class.bot else
    let ((x, -), seed') = Quickcheck-Random.random sz seed
    in Predicate.Seq (%a. Predicate.Insert x (iter' T (nrandom - 1) sz seed')))

definition iter :: natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
where
THEORY "Random-Pred"

iter nrandom sz seed = iter' (TYPE('a)) nrandom sz seed

lemma [code]:
iter nrandom sz seed = (if nrandom = 0 then bot-class.bot else
let ((x, -), seed') = Quickcheck-Random.random sz seed
in Predicate.Seq (%a. Predicate.Insert x (iter (nrandom - 1) sz seed')))
unfolding iter-def iter'.simps [af - nrandom] ..

type-synonym 'a random-pred = Random.seed ⇒ ('a Predicate.pred × Random.seed)

definition empty :: 'a random-pred
where empty = Pair bot

definition single :: 'a => 'a random-pred
where single x = Pair (Predicate.single x)

definition bind :: 'a random-pred ⇒ ('a ⇒ 'b random-pred) ⇒ 'b random-pred
where
bind R f = (λs. let
(P, s') = R s;
(s1, s2) = Random.split-seed s'
in (Predicate.bind P (%a. fst (f a s1)), s2))

definition union :: 'a random-pred ⇒ 'a random-pred ⇒ 'a random-pred
where
union R1 R2 = (λs. let
(P1, s') = R1 s; (P2, s'') = R2 s'
in (sup-class.sup P1 P2, s''))

definition if-randompred :: bool ⇒ unit random-pred
where
if-randompred b = (if b then single () else empty)

definition iterate-uppto :: (natural ⇒ 'a) => natural ⇒ natural ⇒ 'a random-pred
where
iterate-uppto f n m = Pair (Predicate.iterate-uppto f n m)

definition not-randompred :: unit random-pred ⇒ unit random-pred
where
not-randompred P = (λs. let
(P', s') = P s
in if Predicate.eval P' () then (Orderings.bot, s') else (Predicate.single (), s'))

definition Random :: (Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed) ⇒ 'a random-pred
where Random g = scomp g (Pair o (Predicate.single o fst))

definition map :: ('a ⇒ 'b) ⇒ 'a random-pred ⇒ 'b random-pred
where map f P = bind P (single o f)

hide-const (open) iter’ iter empty single bind union if-randompred
iterate-upto not-randompred Random map

hide-fact iter’.simp

hide-fact (open) iter-def empty-def single-def bind-def union-def
if-randompred-def iterate-upto-def not-randompred-def Random-def map-def

end

79 Random-Sequence: Various kind of sequences inside the random monad

theory Random-Sequence
imports Random-Pred
begin

type-synonym 'a random-dseq = natural ⇒ natural ⇒ Random.seed ⇒ ('a Limited-Sequence.dseq × Random.seed)

definition empty :: 'a random-dseq
where
  empty = (%nrandom size s. Pair (Limited-Sequence.empty))

definition single :: 'a ⇒ 'a random-dseq
where
  single x = (%nrandom size s. Pair (Limited-Sequence.single x))

definition bind :: 'a random-dseq ⇒ ('a ⇒ 'b random-dseq) ⇒ 'b random-dseq
where
  bind R f = (λnrandom size s. let
    (P, s') = R nrandom size s;
    (s1, s2) = Random.split-seed s'
in (Limited-Sequence.bind P (%a. fst (f a nrandom size s1)), s2))

definition union :: 'a random-dseq ⇒ 'a random-dseq ⇒ 'a random-dseq
where
  union R1 R2 = (λnrandom size s. let
    (S1, s') = R1 nrandom size s; (S2, s'') = R2 nrandom size s'
in (Limited-Sequence.union S1 S2, s''))

definition if-random-dseq :: bool ⇒ unit random-dseq
where
  if-random-dseq b = (if b then single () else empty)

definition not-random-dseq :: unit random-dseq ⇒ unit random-dseq
THEORY "Random-Sequence"

where
  not-random-dseq R = (\nnrandom size s. let
    (S, s') = R nrandom size s
    in (Limited-Sequence.not-seq S, s'))

definition map :: ('a => 'b) => 'a random-dseq => 'b random-dseq
where
  map f P = bind P (single o f)

fun Random :: (natural => Random.seed => (('a x (unit => term)) x Random.seed))
  => 'a random-dseq
where
  Random g nrandom = (\ns. if nrandom <= 0 then (Pair Limited-Sequence.empty) else
    (scomp (g s) (%r. scomp (Random g (nrandom - 1) s) (%rs. Pair
      (Limited-Sequence.union (Limited-Sequence.single (fst r)) rs)))))

type-synonym 'a pos-random-dseq = natural => natural => Random.seed => 'a
  Limited-Sequence.pos-dseq

definition pos-empty :: 'a pos-random-dseq
where
  pos-empty = (\nnrandom size seed. Limited-Sequence.pos-empty)

definition pos-single :: 'a => 'a pos-random-dseq
where
  pos-single x = (\nnrandom size seed. Limited-Sequence.pos-single x)

definition pos-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b
  pos-random-dseq
where
  pos-bind R f = (\nnrandom size seed. Limited-Sequence.pos-bind (R nrandom size
    seed) (%a. f a nrandom size seed))

definition pos-decr-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b
  pos-random-dseq
where
  pos-decr-bind R f = (\nnrandom size seed. Limited-Sequence.pos-decr-bind (R
    nrandom size seed) (%a. f a nrandom size seed))

definition pos-union :: 'a pos-random-dseq => 'a pos-random-dseq => 'a pos-random-dseq
where
  pos-union R1 R2 = (\nnrandom size seed. Limited-Sequence.pos-union (R1 nrandom
    size seed) (R2 nrandom size seed))

definition pos-if-random-dseq :: bool => unit pos-random-dseq
where
  pos-if-random-dseq b = (if b then pos-single () else pos-empty)
definition pos-iterate-upto :: \((\text{natural} \Rightarrow 'a) \Rightarrow \text{natural} \Rightarrow \text{natural} \Rightarrow 'a\) pos-random-dseq
where
pos-iterate-upto \(f \ n \ m\) = \((\lambda \text{random size seed}. \text{Limited-Sequence.pos-iterate-upto} f \ n \ m)\)

definition pos-map :: \(('a \Rightarrow 'b) \Rightarrow 'a\) pos-random-dseq => 'b pos-random-dseq
where
pos-map \(f \ P\) = pos-bind \(P\) \((\text{pos-single o f})\)

definition iter :: \((\text{Random.seed} \Rightarrow ('a \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed})\) \Rightarrow \text{natural} \Rightarrow \text{Random.seed} \Rightarrow 'a \text{Lazy-Sequence.lazy-sequence}
where
iter \(\text{random nrandom seed}\) =
\((\text{if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence} ('%u. let ((x, -), seed') = random seed in Some (x, iter random (nrandom - 1) seed')))\)

definition pos-Random :: \((\text{natural} \Rightarrow \text{Random.seed} \Rightarrow ('a \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed})\) \Rightarrow 'a pos-random-dseq
where
pos-Random \(g\) = (%\(nrandom size seed depth\). iter \((g size)\) \(nrandom seed\))

type-synonym 'a neg-random-dseq = \text{natural} \Rightarrow \text{natural} \Rightarrow \text{Random.seed} \Rightarrow 'a Limited-Sequence.neg-dseq

definition neg-empty :: 'a neg-random-dseq
where
neg-empty = (%\(nrandom size seed\). Limited-Sequence.neg-empty)

definition neg-single :: 'a => 'a neg-random-dseq
where
definition neg-single \(x\) = (%\(nrandom size seed\). Limited-Sequence.neg-single \(x\))

definition neg-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq
where
definition neg-bind \(R \ f\) = \((\lambda nrandom size seed. \text{Limited-Sequence.neg-bind} (R nrandom size seed)) ('a. f a nrandom size seed))\)

definition neg-decr-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq
where
definition neg-decr-bind \(R \ f\) = \((\lambda nrandom size seed. \text{Limited-Sequence.neg-decr-bind} (R nrandom size seed)) ('a. f a nrandom size seed))\)
definition neg-union :: 'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq
where
  neg-union R1 R2 = (\nnrandom size seed. Limited-Sequence.neg-union (R1 nrandom size seed) (R2 nrandom size seed))

definition neg-if-random-dseq :: bool => unit neg-random-dseq
where
  neg-if-random-dseq b = (if b then neg-single () else neg-empty)

definition neg-iterate-upto :: (natural => 'a) => natural => natural => 'a neg-random-dseq
where
  neg-iterate-upto f n m = (\nnrandom size seed. Limited-Sequence.neg-iterate-upto f n m)

definition neg-not-random-dseq :: unit pos-random-dseq => unit neg-random-dseq
where
  neg-not-random-dseq R = (\nnrandom size seed. Limited-Sequence.neg-not-seq (R nrandom size seed))

definition neg-map :: ('a => 'b) => 'a neg-random-dseq => 'b neg-random-dseq
where
  neg-map f P = neg-bind P (neg-single o f)

definition pos-not-random-dseq :: unit neg-random-dseq => unit pos-random-dseq
where
  pos-not-random-dseq R = (\nnrandom size seed. Limited-Sequence.pos-not-seq (R nrandom size seed))

hide-const (open)
  empty single bind union if-random-dseq not-random-dseq map Random
  pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto
  pos-not-random-dseq pos-map iter pos-Random
  neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto
  neg-not-random-dseq neg-map

hide-fact (open)
  empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def
  map-def Random.simps
  pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def
  neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def
  neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

end
library Quickcheck-Exhaustive

theory Quickcheck-Exhaustive
imports Quickcheck-Random
keywords quickcheck-generator :: thy-decl
begin

80 basic operations for exhaustive generators

definition orelse :: 'a option => 'a option => 'a option (infixl orelse 55)
where
  [code-unfold]: x orelse y = (case x of Some x' => Some x' | None => y)

80 exhaustive generator type classes

class exhaustive = term-of +
  fixes exhaustive :: ('a => (bool * term list) option) => natural => (bool * term list) option

class full-exhaustive = term-of +
  fixes full-exhaustive :: ('a * (unit => term) => bool * term list) option) => natural => (bool * term list) option

instantiation natural :: full-exhaustive
begin

function full-exhaustive-natural' :: (natural * (unit => term) = (bool * term list) option) =>
  natural => natural => (bool * term list) option
where full-exhaustive-natural' f d i =
  (if d < i then None
   else (f (i, %-. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d
   (i + 1)))
by pat-completeness auto

termination
by (relation measure (%(\_, d, i). nat-of-natural (d + 1 - i)))
  (auto simp add: less-natural-def)

definition full-exhaustive f d = full-exhaustive-natural' f d 0

instance ..

end

instantiation natural :: exhaustive
begin

function exhaustive-natural' :: (natural = (bool * term list) option) => natural

end
theory Quickcheck-Exhaustive

natural => (bool * term list) option

where exhaustive-natural' f d i =
  (if d < i then None
   else (f i orelse exhaustive-natural' f d (i + 1)))

by pat-completeness auto

termination
  by (relation measure (%(.-, d, i). nat-of-natural (d + 1 - i)))
    (auto simp add: less-natural-def)

definition exhaustive f d = exhaustive-natural' f d 0

instance ..

end

instantiation integer :: exhaustive
begin

function exhaustive-integer' :: (integer => (bool * term list) option) => integer
  => integer => (bool * term list) option
  where exhaustive-integer' f d i = (if d < i then None else (f i orelse exhaustive-integer'
    f d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (%(.-, d, i). nat-of-integer (d + 1 - i)))
    (auto simp add: less-integer-def nat-of-integer-def)

definition exhaustive f d = exhaustive-integer' f (integer-of-natural d) (- (integer-of-natural
d))

instance ..

end

instantiation integer :: full-exhaustive
begin

function full-exhaustive-integer' :: (integer * (unit => term) => (bool * term
  list) option) => integer => integer => (bool * term list) option
  where full-exhaustive-integer' f d i = (if d < i then None else (case f (i, %-.
    Code-Evaluation.term-of i) of Some t => Some t | None => full-exhaustive-integer'
    f d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (%(.-, d, i). nat-of-integer (d + 1 - i)))
    (auto simp add: less-integer-def nat-of-integer-def)
definition full-exhaustive $f \ d = \ \text{full-exhaustive-integer}' \ f \ (\text{integer-of-natural} \ d) \ (- \ (\text{integer-of-natural} \ d))$

instance ..
end

instantiation nat :: exhaustive
begin
definition exhaustive $f \ d = \ \text{exhaustive} \ (%x. \ f \ (\text{nat-of-natural} \ x)) \ d$
instance ..
end

instantiation nat :: full-exhaustive
begin
definition full-exhaustive $f \ d = \ \text{full-exhaustive} \ (%(x, \ xt). \ f \ (\text{nat-of-natural} \ x, \ %.- \ \text{Code-Evaluation}.\text{term-of} \ (\text{nat-of-natural} \ x))) \ d$
instance ..
end

instantiation int :: exhaustive
begin
function exhaustive-int' :: (int => (bool * term list) option) => int => int => (bool * term list) option
  where exhaustive-int' $f \ d \ i = (if \ d < i \ then \ None \ else \ (f \ i \ orelse \ exhaustive-int' \ f \ d \ (i + 1))))$
by pat-completeness auto

termination
  by (relation measure (%(-, \ d, \ i). \ nat \ (d + 1 - i))) auto

definition exhaustive $f \ d = \ \text{exhaustive-int'} \ f \ (\text{int-of-integer} \ (\text{integer-of-natural} \ d))$
  (- \ (\text{int-of-integer} \ (\text{integer-of-natural} \ d)))
instance ..
end

instantiation int :: full-exhaustive
begin
function full-exhaustive-int' :: (int * (unit => term) => (bool * term list) option) 
=> int => int => (bool * term list) option 
  where full-exhaustive-int' f d i = (if d < i then None else (case f (i, %-. Code-Evaluation.term-of i) of Some t => Some t | None => full-exhaustive-int' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (%(\_, d, i). nat (d + 1 - i))) auto

definition full-exhaustive f d = full-exhaustive-int' f (int-of-integer (integer-of-natural d))
  (\_ (int-of-integer (integer-of-natural d)))

instance ..
end

instantiation prod :: (exhaustive, exhaustive) exhaustive begin

definition
  exhaustive f d = exhaustive (\x. exhaustive (\y. f ((x, y))) d) d

instance ..
end

definition (in term-syntax) [code-unfold]: valtermify-pair x y = Code-Evaluation.valtermify (Pair :: \'a :: typerep => \'b :: typerep => \'a * \'b) {} x {} y

instantiation prod :: (full-exhaustive, full-exhaustive) full-exhaustive begin

definition
  full-exhaustive f d = full-exhaustive (\x. full-exhaustive (\y. f (valtermify-pair x y)) d) d

instance ..
end

instantiation set :: (exhaustive) exhaustive begin

fun exhaustive-set
  where
  exhaustive-set f i = (if i = 0 then None else (f {}) orelse exhaustive-set (\A. f A orelse exhaustive (\x. if x \in A then None else f (insert x A)) (i - 1)) (i - 1)))
instance ..

end

instantiation set :: (full-exhaustive) full-exhaustive begin

fun full-exhaustive-set where
  full-exhaustive-set f i = (if i = 0 then None else (f valterm-emptyset orelse
  full-exhaustive-set (%A. f A orelse Quickcheck-Exhaustive.full-exhaustive (%x. if
  fst x ∈ fst A then None else f (valtermify-insert x A)) (i - 1)) (i - 1)))

instance ..

end

instantiation fun :: ({equal, exhaustive}, exhaustive) exhaustive begin

fun exhaustive-fun' :: (('a => 'b) => (bool * term list) option) => natural =>
  natural => (bool * term list) option where
  exhaustive-fun' f i d = (exhaustive (%b. f (%- b)) d) orelse
  exhaustive-fun' (%g. exhaustive (%a. exhaustive (%b. f (g(a := b))) d) d) (i - 1) d else None

definition exhaustive-fun :: (('a => 'b) => (bool * term list) option) => natural =>
  (bool * term list) option where
  exhaustive-fun f d = exhaustive-fun' f d d

instance ..

end

definition [code-unfold]: valtermify-unsafedummy = (%(v, t). (%::'a. v, %::unit.
  Code-Evaluation.Abs (STR "x"') (Typerep.typerep TYPE('a::typerep)) (t ()')))
option) => natural => natural => (bool * term list) option
where
full-exhaustive-fun' f i d = (full-exhaustive (%v. f (valtermify-absdummy v)) d)
  orelse (if i > 1 then
  full-exhaustive-fun' (%g. full-exhaustive (%a. full-exhaustive (%b. f (valtermify-fun-upd g a b)) d) d) (i - 1) d else None)

definition full-exhaustive-fun :: (('a => 'b) * (unit => term) => (bool * term list) option) => natural => (bool * term list) option
where
full-exhaustive-fun f d = full-exhaustive-fun' f d d

instance .. end

80.2.1 A smarter enumeration scheme for functions over finite datatypes

class check-all = enum + term-of +
  fixes check-all :: ('a * (unit => term) => (bool * term list) option) => (bool * term list) option
  fixes enum-term-of :: 'a itself => unit => term list

fun check-all-n-lists :: ('a :: check-all) list * (unit => term list) => (bool * term list) option
where
check-all-n-lists f n =
  (if n = 0 then f ([], (%_. [])) else check-all (%(x, xt). check-all-n-lists (%(xs, xst). f ((x # xs), (%_. (xt () # xst ())))) (n - 1)))

definition (in term-syntax) [code-unfold]: termify-fun-upd g a b = (Code-Evaluation.termify (fun-upd :: ('a :: typerep => 'b :: typerep) => 'a => 'b => 'a => 'b) <- g <- a <> b)

definition mk-map-term :: (unit => typerep) => (unit => typerep) => (unit => term list) => (unit => term list) => unit => term
where
mk-map-term T1 T2 domm rng =
  (%_. let T1 = T1 ();
   T2 = T2 ();
   update-term = (%g (a, b),
   (Code-Evaluation.Const (STR "fun.fun-upd")
   (Typerep.Typerep (STR "fun")) [Typerep.Typerep (STR "fun") [T1, T2]],
   Typerep.Typerep (STR "fun") [T1, Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "fun") [T1, T2]]])))}
THEORY “Quickcheck-Exhaustive”

in
List.foldl update-term (Code-Evaluation.Abs (STR "x") T1 (Code-Evaluation.Const (STR "HOL.undefined") T2)) (zip (domm ()) (rng ()))

instantiation fun :: ({equal, check-all}, check-all) check-all
begin

definition check-all f =
(let
  mk-term = mk-map-term (%-. Typerep.typerep (TYPE('a)) (%-. Typerep.typerep (TYPE('b))) (enum-term-of (TYPE('a)));
  enum = (Enum.enum :: 'a list)
in check-all-n-lists (λ(ys, yst). f (the o map-of (zip enum ys), mk-term yst))
(natural-of-nat (length enum)))

definition enum-term-of-fun :: ('a => 'b) itself => unit => term list
where
  enum-term-of-fun = (%-. let
    enum-term-of-a = enum-term-of (TYPE('a));
    mk-term = mk-map-term (%-. Typerep.typerep (TYPE('a)) (%-. Typerep.typerep
    (TYPE('b))) enum-term-of-a
  in map (%ys. mk-term (%-. ys ()) (List.n-lists (length (enum-term-of-a ()))))
  (enum-term-of (TYPE('b) ()))))

instance ..

end

fun (in term-syntx) check-all-subsets :: ('a :: typerep) set * (unit => term) =>
  (bool * term list) option) => ('a * (unit => term)) list => (bool * term list)
  option
where
  check-all-subsets f [] = f valterm-emptyset
| check-all-subsets f (x # xs) = check-all-subsets (%s. case f s of Some ts =>
  Some ts | None => f (valtermify-insert x s)) xs

definition (in term-syntx) [code-unfold]: term-emptyset = Code-Evaluation.termify
  ({} :: ('a :: typerep) set)
definition (in term-syntx) [code-unfold]: termify-insert x s = Code-Evaluation.termify
  (insert :: ('a::typerep) => 'a set => 'a set) <\> x <\> s
definition (in term-syntx) setify :: ('a::typerep) itself => term list => term
where
  setify T ts = foldr (termify-insert T) ts (term-emptyset T)

instantiation set :: (check-all) check-all
begin

definition check-all-set f =
  check-all-subsets f (zip (Enum.enum :: 'a list) (map (%a. %u :: unit. a)
  (Quickcheck-Exhaustive.enum-term-of (TYPE ('a)) ())))

definition enum-term-of-set :: 'a set itself => unit => term list
where
  enum-term-of-set - - = map (setify (TYPE('a))) (sublists (Quickcheck-Exhaustive.enum-term-of
  (TYPE('a)) ())))

instance ..
end

instantiation unit :: check-all
begin

definition check-all f = f (Code-Evaluation.valtermify ())

definition enum-term-of-unit :: unit itself => unit => term list
where
  enum-term-of-unit = (%- -. [Code-Evaluation.term-of ()])

instance ..
end

instantiation bool :: check-all
begin

definition check-all f = (case f (Code-Evaluation.valtermify False) of Some x' => Some x'
  | None => f (Code-Evaluation.valtermify True))

definition enum-term-of-bool :: bool itself => unit => term list
where
  enum-term-of-bool = (%- -. map Code-Evaluation.term-of (Enum.enum :: bool
  list))

instance ..
end

definition (in term-syntax) [code-unfold]: termify-pair x y = Code-Evaluation.termify
  (Pair :: 'a :: typerep => 'b :: typerep => 'a * 'b) <-> x <-> y
THEORY "Quickcheck-Exhaustive"

instantiation prod :: (check-all, check-all) check-all
begin

definition
check-all f = check-all (%x. check-all (%y. f (valtermify-pair x y)))

definition enum-term-of-prod :: ('a * 'b) itself => unit => term list
where
enum-term-of-prod = (%- -. map (%(x, y). termify-pair TYPE('a) TYPE('b) x y)
(List.product (enum-term-of (TYPE('a))) (enum-term-of (TYPE('b))) ()))

instance ..
end

definition (in term-syntax) [code-unfold] valtermify-Inl x = Code-Evaluation.valtermify
(Inl :: 'a :: typerep => 'a + 'b :: typerep) {} x

definition (in term-syntax) [code-unfold] valtermify-Inr x = Code-Evaluation.valtermify
(Inr :: 'b :: typerep => 'a ::typerep + 'b) {} x

instantiation sum :: (check-all, check-all) check-all
begin

definition check-all f = check-all (%a. f (valtermify-Inl a)) orelse check-all (%b. f (valtermify-Inr b))

definition enum-term-of-sum :: ('a + 'b) itself => unit => term list
where
enum-term-of-sum = (%- -. let
T1 = (Typerep.typerep (TYPE('a)));
T2 = (Typerep.typerep (TYPE('b)))
in
(Typerep.Typerep (STR "fun") [T1, Typerep.Typerep (STR "Sum-Type.sum")
[T1, T2]]))
(enum-term-of (TYPE('a))) @
map (Code-Evaluation.App (Code-Evaluation.Const (STR "Sum-Type.Inr"))
(Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "Sum-Type.sum")
[T1, T2]]))
(enum-term-of (TYPE('b))) ()))

instance ..
end
instantiation nibble :: check-all
begin

definition check-all f =
  f (Code-Evaluation.valtermify Nibble0) orelse
  f (Code-Evaluation.valtermify Nibble1) orelse
  f (Code-Evaluation.valtermify Nibble2) orelse
  f (Code-Evaluation.valtermify Nibble3) orelse
  f (Code-Evaluation.valtermify Nibble4) orelse
  f (Code-Evaluation.valtermify Nibble5) orelse
  f (Code-Evaluation.valtermify Nibble6) orelse
  f (Code-Evaluation.valtermify Nibble7) orelse
  f (Code-Evaluation.valtermify Nibble8) orelse
  f (Code-Evaluation.valtermify Nibble9) orelse
  f (Code-Evaluation.valtermify NibbleA) orelse
  f (Code-Evaluation.valtermify NibbleB) orelse
  f (Code-Evaluation.valtermify NibbleC) orelse
  f (Code-Evaluation.valtermify NibbleD) orelse
  f (Code-Evaluation.valtermify NibbleE) orelse
  f (Code-Evaluation.valtermify NibbleF)

definition enum-term-of-nibble :: nibble itself => unit => term list
where
  enum-term-of-nibble = (%- -. map Code-Evaluation.term-of (Enum.enum :: nibble list))

instance ..
end

instantiation char :: check-all
begin

definition check-all f = check-all (%(x, t1). check-all (%(y, t2). f (Char x y, %- -. Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.term-of Char) t1 ()) (t2 ()))))

definition enum-term-of-char :: char itself => unit => term list
where
  enum-term-of-char = (%- -. map Code-Evaluation.term-of (Enum.enum :: char list))

instance ..
end
instantiation option :: (check-all) check-all
begin
definition check-all f = f (Code-Evaluation.valtermify (None :: 'a option)) orelse check-all (%(x, t). f (Some x, %- Code-Evaluation.App
                  (Code-Evaluation.Const (STR "'Option.option.Some'"))
               (Typerep.Typerep (STR "'fun'") [Typerep.Typerep TYPE('a), Typerep.Typerep
                  (STR "'Option.option'") [Typerep.Typerep TYPE('a)]]) (t ()))))
definition enum-term-of-option :: 'a option itself => unit => term list
where enum-term-of-option = (% - -. (Code-Evaluation.term-of (None :: 'a option)) #
                  (Typerep.Typerep (STR "'fun'") [Typerep.Typerep TYPE('a), Typerep.Typerep
                  (STR "'Option.option'") [Typerep.Typerep TYPE('a)]])
               (enum-term-of (TYPE('a))) ())))
instance ..
end

instantiation Enum.finite-1 :: check-all
begin
definition check-all f = f (Code-Evaluation.valtermify Enum.finite-1.a1)
definition enum-term-of-finite-1 :: Enum.finite-1 itself => unit => term list
where enum-term-of-finite-1 = (%- -. (Code-Evaluation.term-of Enum.finite-1.a1))
instance ..
end

instantiation Enum.finite-2 :: check-all
begin
definition check-all f = (f (Code-Evaluation.valtermify Enum.finite-2.a1)
               orelse f (Code-Evaluation.valtermify Enum.finite-2.a2))
definition enum-term-of-finite-2 :: Enum.finite-2 itself => unit => term list
where enum-term-of-finite-2 = (%- -. map Code-Evaluation.term-of (Enum.enum ::
               Enum.finite-2 list))
instance ..

end

instantiation Enum.finite-3 :: check-all
begin

definition check-all f =
  let
    a1 = Code-Evaluation.valtermify Enum.finite-3.a1
    a2 = Code-Evaluation.valtermify Enum.finite-3.a2
    a3 = Code-Evaluation.valtermify Enum.finite-3.a3
  in
    f a1 orelse f a2 orelse f a3

definition enum-term-of-finite-3 :: Enum.finite-3 itself => unit => term list
where
  enum-term-of-finite-3 = (% - -.
    map Code-Evaluation.term-of (Enum.enum :: Enum.finite-3 list))

instance ..

end

instantiation Enum.finite-4 :: check-all
begin

definition check-all f =
  let
    a1 = Code-Evaluation.valtermify Enum.finite-4.a1
    a2 = Code-Evaluation.valtermify Enum.finite-4.a2
    a3 = Code-Evaluation.valtermify Enum.finite-4.a3
    a4 = Code-Evaluation.valtermify Enum.finite-4.a4
  in
    f a1 orelse f a2 orelse f a3 orelse f a4

definition enum-term-of-finite-4 :: Enum.finite-4 itself => unit => term list
where
  enum-term-of-finite-4 = (% - -.
    map Code-Evaluation.term-of (Enum.enum :: Enum.finite-4 list))

instance ..

end

80.3 Bounded universal quantifiers

class bounded-forall =
  fixes bounded-forall :: ('a => bool) => natural => bool

80.4 Fast exhaustive combinators

class fast-exhaustive = term-of +
  fixes fast-exhaustive :: ('a => unit) => natural => unit
THEORY “Quickcheck-Exhaustive”

axiomatization throw-Counterexample :: term list => unit
axiomatization catch-Counterexample :: unit => term list option

code-printing
  constant throw-Counterexample =>
    (Quickcheck) raise (Exhaustive’.Generators.Counterexample -)
| constant catch-Counterexample =>
    (Quickcheck) (((-); NONE) handle Exhaustive’.Generators.Counterexample ts
  => SOME ts)

80.5 Continuation passing style functions as plus monad

type-synonym ’a cps = (’a => term list option) => term list option

definition cps-empty :: ’a cps
  where
cps-empty = (%cont. None)

definition cps-single :: ’a => ’a cps
  where
cps-single v = (%cont. cont v)

definition cps-bind :: ’a cps => (’a => ’b cps) => ’b cps
  where
cps-bind m f = (%cont. m (%a. (f a) cont))

definition cps-plus :: ’a cps => ’a cps => ’a cps
  where
cps-plus a b = (%c. case a c of None => b c | Some x => Some x)

definition cps-if :: bool => unit cps
  where
cps-if b = (if b then cps-single () else cps-empty)

definition cps-not :: unit cps => unit cps
  where
cps-not n = (%c. case n (%u. Some [])) of None => c () | Some - => None

type-synonym ’a pos-bound-cps = (’a => (bool * term list) option) => natural
  => (bool * term list) option

definition pos-bound-cps-empty :: ’a pos-bound-cps
  where
  pos-bound-cps-empty = (%cont i. None)

definition pos-bound-cps-single :: ’a => ’a pos-bound-cps
  where
  pos-bound-cps-single v = (%cont i. cont v)
definition pos-bound-cps-bind :: 'a pos-bound-cps => ('a => 'b pos-bound-cps)
  => 'b pos-bound-cps
where
  pos-bound-cps-bind m f = (%cont i. if i = 0 then None else (m (%a. (f a) cont i) (i - 1)))

definition pos-bound-cps-plus :: 'a pos-bound-cps => 'a pos-bound-cps => 'a pos-bound-cps
where
  pos-bound-cps-plus a b = (%c i. case a c i of None => b c i | Some x => Some x)

definition pos-bound-cps-if :: bool => unit pos-bound-cps
where
  pos-bound-cps-if b = (if b then pos-bound-cps-single () else pos-bound-cps-empty)

datatype 'a unknown = Unknown | Known 'a
datatype 'a three-valued = Unknown-value | Value 'a | No-value
type-synonym 'a neg-bound-cps = ('a unknown => term list three-valued) => term list three-valued

definition neg-bound-cps-empty :: 'a neg-bound-cps
where
  neg-bound-cps-empty = (%cont i. No-value)

definition neg-bound-cps-single :: 'a => 'a neg-bound-cps
where
  neg-bound-cps-single v = (%cont i. cont (Known v))

definition neg-bound-cps-bind :: 'a neg-bound-cps => ('a => 'b neg-bound-cps)
  => 'b neg-bound-cps
where
  neg-bound-cps-bind m f = (%cont i. if i = 0 then cont Unknown else m (%a. case a of Unknown => cont Unknown | Known a' => f a' cont i) (i - 1))

definition neg-bound-cps-plus :: 'a neg-bound-cps => 'a neg-bound-cps => 'a neg-bound-cps
where
  neg-bound-cps-plus a b = (%c i. case a c i of No-value => b c i | Value x => Value x | Unknown-value => (case b c i of No-value => Unknown-value | Value x => Value x | Unknown-value => Unknown-value))

definition neg-bound-cps-if :: bool => unit neg-bound-cps
where
  neg-bound-cps-if b = (if b then neg-bound-cps-single () else neg-bound-cps-empty)

definition neg-bound-cps-not :: unit pos-bound-cps => unit neg-bound-cps
where
neg-bound-cps-not \( n = \% c \ i. \ \text{case} \ n (\% u. \ \text{Some} \ (\text{True}, [])) \ i \ \text{of} \ \text{None} =\rightarrow c \) (Known (())), \( \text{Some} -\rightarrow \text{No-value} \)

definition pos-bound-cps-not :: unit \(\rightarrow\) unit \( \text{neg-bound-cps-not} \)
where
\( \text{pos-bound-cps-not} \ n = \% c \ i. \ \text{case} \ n (\% u. \ \text{Value} []) \ i \ \text{of} \ \text{No-value} =\rightarrow c \) (Known (())), \( \text{Value} -\rightarrow \text{None} | \text{Unknown-value} =\rightarrow \text{None} \)

80.6 Defining generators for any first-order data type

axiomatization unknown :: 'a

notation (output) unknown (?)

ML-file Tools/Quickcheck/exhaustive-generators.ML

setup ⟨⟨ Exhaustive-Generators.setup ⟩⟩

declare [[quickcheck-batch-tester = exhaustive]]

80.7 Defining generators for abstract types

ML-file Tools/Quickcheck/abstract-generators.ML

hide-fact (open) orelse-def
no-notation orelse (infixr orelse 55)

hide-fact
exhaustive-int′-def
exhaustive-integer′-def
exhaustive-natural′-def

hide-const valtermify-absdummy valtermify-fun-upd valtermify-emptyset valtermify-insert valtermify-pair
valtermify-Inl valtermify-Inr
termify-fun-upd term-emptyset termify-insert termify-pair setify

hide-const (open)
exhaustive full-exhaustive
exhaustive-int′ full-exhaustive-int′
exhaustive-integer′ full-exhaustive-integer′
exhaustive-natural′ full-exhaustive-natural′
throw-Counterexample catch-Counterexample
check-all enum-term-of
orelse unknown mk-map-term check-all-n-lists check-all-subsets

hide-type (open) cps pos-bound-cps neg-bound-cps unknown three-valued
hide-const (open) cps-empty cps-single cps-bind cps-plus cps-if cps-not
  pos-bound-cps-empty pos-bound-cps-single pos-bound-cps-bind pos-bound-cps-plus
  pos-bound-cps-if pos-bound-cps-not
neg-bound-cps-empty neg-bound-cps-single neg-bound-cps-bind neg-bound-cps-plus
neg-bound-cps-if neg-bound-cps-not
Unknown Known Unknown-value Value No-value

end

81 Predicate-Compile: A compiler for predicates defined by introduction rules

theory Predicate-Compile
imports Random-Sequence Quickcheck-Exhaustive
keywords code-pred :: thy-goal and values :: diag
begin

ML-file Tools/Predicate-Compile/predicate-compile-aux.ML
ML-file Tools/Predicate-Compile/predicate-compile-compilations.ML
ML-file Tools/Predicate-Compile/core-data.ML
ML-file Tools/Predicate-Compile/mode-inference.ML
ML-file Tools/Predicate-Compile/predicate-compile-proof.ML
ML-file Tools/Predicate-Compile/predicate-compile-core.ML
ML-file Tools/Predicate-Compile/predicate-compile-data.ML
ML-file Tools/Predicate-Compile/predicate-compile-fun.ML
ML-file Tools/Predicate-Compile/predicate-compile-pred.ML
ML-file Tools/Predicate-Compile/predicate-compile-specialisation.ML
ML-file Tools/Predicate-Compile/predicate-compile.ML

setup Predicate-Compile.setup

81.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

definition contains :: 'a set => 'a => bool
  where contains A x <-> x : A

definition contains-pred :: 'a set => 'a => unit Predicate.pred
  where contains-pred A x = (if x : A then Predicate.single () else bot)

lemma pred-of-setE:
  assumes Predicate.eval (pred-of-set A) x
  obtains contains A x
  using assms by (simp add: contains-def)

lemma pred-of-setI: contains A x ==> Predicate.eval (pred-of-set A) x
  by (simp add: contains-def)

lemma pred-of-set-eq: pred-of-set == \lambda A. Predicate.Pred (contains A)
  by (simp add: contains-def [abs-def] pred-of-set-def o-def)
lemma containsI: \( x \in A \Rightarrow \text{contains } A x \)
by (simp add: contains-def)

lemma containsE: assumes \( \text{contains } A x \)
obtains \( A' x' \) where \( A = A' x = x' x : A \)
using assms by (simp add: contains-def)

lemma contains-predI: \( \text{contains } A x \Rightarrow \text{Predicate.eval (contains-pred } A x) () \)
by (simp add: contains-pred-def contains-def)

lemma contains-predE:
  assumes \( \text{Predicate.eval (contains-pred } A x) y \)
obtains \( \text{contains } A x \)
using assms by (simp add: contains-pred-def contains-def split: split-if-asm)

lemma contains-pred-eq: \( \text{contains-pred } \equiv \lambda A x. \text{Predicate.Pred} (\lambda y. \text{contains } A x) \)
by (rule eq-reflection) (auto simp add: contains-pred-def function-names intro: pred-eqI)

lemma contains-pred-notI:
  \( \neg \text{contains } A x \Rightarrow \text{Predicate.eval (Predicate.not-pred (contains-pred } A x)) () \)
by (simp add: contains-pred-def contains-def not-pred-eq)

setup "let
  val Fun = Predicate-Compile-Aux.Fun
  val Input = Predicate-Compile-Aux.Input
  val Output = Predicate-Compile-Aux.Output
  val Bool = Predicate-Compile-Aux.Bool
  val io = Fun (Input, Fun (Output, Bool))
  val ii = Fun (Input, Fun (Input, Bool))
in
Core-Data.PredData.map (Graph.new-node
  (@ {const-name contains},
    Core-Data.PredData {
      pos = Position.thread-data (),
      intros = [(NONE, @ {thm containsI})],
      elim = SOME @ {thm containsE},
      preprocessed = true,
      function-names = [(Predicate-Compile-Aux.Pred,
        [(io, @ {const-name pred-of-set}), (ii, @ {const-name contains-pred})])],
      predfun-data = [
        (io, Core-Data.PredfunData {
          elim = @ {thm pred-of-setE}, intro = @ {thm pred-of-setI},
          neg-intro = NONE, definition = @ {thm pred-of-set-eq}
        } ),
        (ii, Core-Data.PredfunData {
      })])"
THEORY “Quickcheck-Narrowing”

elim = @{thm contains-predE}, intro = @{thm contains-predI},
neg-intro = SOME @{thm contains-pred-notI}, definition = @{thm contains-pred-eq}

end

hide-const (open) contains contains-pred
hide-fact (open) pred-of-setE pred-of-setI pred-of-set-eq
containsI containsE contains-predI contains-predE contains-pred-eq contains-pred-notI
end

82 Quickcheck-Narrowing: Counterexample generator performing narrowing-based testing

theory Quickcheck-Narrowing
imports Quickcheck-Random
keywords find-unused-assms :: diag
begin

82.1 Counterexample generator

82.1.1 Code generation setup

setup ⟨⟨ Code-Target.extend-target (Haskell-Quickcheck, (Code-Haskell.target, I)) ⟩⟩

code-printing
  code-module Typerep → (Haskell-Quickcheck) ⟨⟨
data Typerep = Typerep String [Typerep]
⟩⟩
| type-constructor typerep → (Haskell-Quickcheck) Typerep.Typerep
| constant Typerep.Typerep → (Haskell-Quickcheck) Typerep.Typerep
| type-constructor integer → (Haskell-Quickcheck) Prelude.Int

code-reserved Haskell-Quickcheck Typerep

82.1.2 Narrowing’s deep representation of types and terms

datatype narrowing-type = Narrowing-sum-of-products narrowing-type list list
datatype narrowing-term = Narrowing-variable integer list narrowing-type | Narrowing-constructor
integer narrowing-term list
datatype ′a narrowing-cons = Narrowing-cons narrowing-type (narrowing-term
list => ′a) list

primrec map-cons :: (′a => ′b) => ′a narrowing-cons => ′b narrowing-cons
where
map-cons f (Narrowing-cons ty cs) = Narrowing-cons ty (map (%c. f o c) cs)

82.1.3 From narrowing’s deep representation of terms to Code-Evaluation’s terms

class partial-term-of = typerep +
  fixes partial-term-of :: 'a itself => narrowing-term => Code-Evaluation.term

lemma partial-term-of-anything: partial-term-of x nt ≡ t
  by (rule eq-reflection) (cases partial-term-of x nt, cases t, simp)

82.1.4 Auxiliary functions for Narrowing

consts nth :: 'a list => integer => 'a

code-printing constant nth → (Haskell-Quickcheck) infixl 9 !!

consts error :: char list => 'a

code-printing constant error → (Haskell-Quickcheck) error

consts toEnum :: integer => char

code-printing constant toEnum → (Haskell-Quickcheck) Prelude.toEnum

consts marker :: char

code-printing constant marker → (Haskell-Quickcheck) '\0'

82.1.5 Narrowing’s basic operations

type-synonym 'a narrowing = integer => 'a narrowing-cons

definition empty :: 'a narrowing
  where
  empty d = Narrowing-cons (Narrowing-sum-of-products []) []

definition cons :: 'a => 'a narrowing
  where
  cons a d = (Narrowing-cons (Narrowing-sum-of-products [[]]) ([% a]))

fun conv :: (narrowing-term list => 'a) list => narrowing-term => 'a
  where
  conv cs (Narrowing-variable p -) = error (marker # map toEnum p)
  | conv cs (Narrowing-constructor i xs) = (nth cs i) xs

fun non-empty :: narrowing-type => bool
  where
    non-empty (Narrowing-sum-of-products ps) = (∼ (List.null ps))
definition apply :: ('a => 'b) narrowing => 'a narrowing => 'b narrowing
where
  apply f a d =
    (case f d of Narrowing-cons (Narrowing-sum-of-products ps) cfs =>>
      case a (d - 1) of Narrowing-cons ta cas =>>
        let
          shallow = (d > 0 ∧ non-empty ta);
          cs = (%xs. (case xs of [] => undefined | x # xs => cf xs (conv cas x))).
          shallow, cf <=- cfs
        in Narrowing-cons (Narrowing-sum-of-products [ta # p, shallow, p <=- ps])
      cs)

definition sum :: 'a narrowing => 'a narrowing => 'a narrowing
where
  sum a b d =
    (case a d of Narrowing-cons (Narrowing-sum-of-products ssa) ca =>>
      case b d of Narrowing-cons (Narrowing-sum-of-products ssb) cb =>>
      Narrowing-cons (Narrowing-sum-of-products (ssa @ ssb)) (ca @ cb))

lemma [fundef-cong]:
  assumes a d = a' d b d = b' d d = d'
  shows sum a b d = sum a' b' d'
  using assms unfolding sum-def by (auto split: narrowing-cons.split narrowing-type.split)

lemma [fundef-cong]:
  assumes f d = f' d (∀d'. 0 ≤ d' ∧ d' < d =>> a d' = a' d')
  assumes d = d'
  shows apply f a d = apply f' a' d'
  proof –
    note assms
    moreover have 0 < d' =>> 0 ≤ d' - 1
      by (simp add: less-integer-def less-eq-integer-def)
  ultimately show ?thesis
    by (auto simp add: apply-def Let-def
      split: narrowing-cons.split narrowing-type.split)
  qed

82.1.6 Narrowing generator type class

class narrowing =
  fixes narrowing :: integer => 'a narrowing-cons

datatype property = Universal narrowing-type (narrowing-term => property)

definition exists :: ('a :: {narrowing, partial-term-of} => property) => property
where
exists f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Existential ty (λ t. f (conv cs t)) (partial-term-of (TYPE('a))))

definition all :: ('a :: {narrowing, partial-term-of} => property) => property
where
  all f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Universal ty (λ t. f (conv cs t)) (partial-term-of (TYPE('a))))

82.1.7 class is-testable

The class is-testable ensures that all necessary type instances are generated.

class is-testable

instance bool :: is-testable ..

instance fun :: ({term-of, narrowing, partial-term-of}, is-testable) is-testable ..

definition ensure-testable :: 'a :: is-testable => 'a :: is-testable
where
  ensure-testable f = f

82.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype ('a, 'b) ffun = Constant 'b | Update 'a 'b ('a, 'b) ffun

primrec eval-ffun :: ('a, 'b) ffun => 'a => 'b
where
  eval-ffun (Constant c) x = c
  | eval-ffun (Update x y f) x = (if x = x' then y else eval-ffun f x)

hide-type (open) ffun
hide-const (open) Constant Update eval-ffun

datatype 'b cfun = Constant 'b

primrec eval-cfun :: 'b cfun => 'a => 'b
where
  eval-cfun (Constant c) y = c

hide-type (open) cfun
hide-const (open) Constant eval-cfun Abs-cfun Rep-cfun

82.1.9 Setting up the counterexample generator

ML-file Tools/Quickcheck/narrowing-generators.ML

setup ⟨⟨ Narrowing-Generators.setup ⟩⟩
definition narrowing-dummy-partial-term-of :: ('a :: partial-term-of) itself => narrowing-term => term
where
  narrowing-dummy-partial-term-of = partial-term-of

definition narrowing-dummy-narrowing :: integer => ('a :: narrowing) narrowing-cons
where
  narrowing-dummy-narrowing = narrowing

lemma [code]:
  ensure-testable f =
  (let
    x = narrowing-dummy-narrowing :: integer => bool narrowing-cons;
    y = narrowing-dummy-partial-term-of :: bool itself => narrowing-term =>
      term;
    z = (conv :: - => - => unit) in f)
unfolding Let-def ensure-testable-def ..

82.2 Narrowing for sets

instantiation set :: (narrowing) narrowing
begin

definition narrowing-set = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons
set) narrowing

instance ..

end

82.3 Narrowing for integers

definition drawn-from :: 'a list => 'a narrowing-cons
where
  drawn-from xs =
    Narrowing-cons (Narrowing-sum-of-products (map (λx. []) xs)) (map (λx -. x)
x)

function around-zero :: int => int list
where
  around-zero i = (if i < 0 then [] else (if i = 0 then [0] else around-zero (i - 1)
@ [i, -i]))
    by pat-completeness auto
termination by (relation measure nat) auto

declare around-zero.simps [simp del]

lemma length-around-zero:
  assumes i >= 0
  shows length (around-zero i) = 2 * nat i + 1
proof (induct rule: int-ge-induct [OF assms])

  case 1
  from 1 show ?case by (simp add: around-zero.simps)

next
  case (2 i)
  from 2 show ?case
    by (simp add: around-zero.simps [of i + 1])

qed

instantiation int :: narrowing
begin

  definition narrowing-int d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
       in drawn-from (around-zero i))

instance ..
end

lemma [code, code del]: partial-term-of (ty :: int itself) t ≡ undefined
  by (rule partial-term-of-anything)+

lemma [code]:
  partial-term-of (ty :: int itself) (Narrowing-variable p t) ≡
    Code-Evaluation.Free (STR "\ldots") (Typerep.Typerep (STR "Int.int") [])
  partial-term-of (ty :: int itself) (Narrowing-constructor i []) ≡
    (if i mod 2 = 0
     then Code-Evaluation.term-of (-(int-of-integer i) div 2)
     else Code-Evaluation.term-of ((int-of-integer i + 1) div 2))
  by (rule partial-term-of-anything)+

instantiation integer :: narrowing
begin

  definition narrowing-integer d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
       in drawn-from (map integer-of-int (around-zero i)))

instance ..
end

lemma [code, code del]: partial-term-of (ty :: integer itself) t ≡ undefined
  by (rule partial-term-of-anything)+

lemma [code]:
  partial-term-of (ty :: integer itself) (Narrowing-variable p t) ≡
    Code-Evaluation.Free (STR "\ldots") (Typerep.Typerep (STR "Code-Numeral.integer")
\begin{align*}
\text{partial-term-of } (ty :: \text{integer itself}) \text{ (Narrowing-constructor } i \text{ []}) \equiv \\
\text{if } i \bmod 2 = 0 \\
\text{then Code-Evaluation.} \text{term-of } (-i \bmod 2) \\
\text{else Code-Evaluation.} \text{term-of } ((i + 1) \div 2)) \\
\text{by (rule partial-term-of-anything)} +
\end{align*}

**code-printing constant** Code-Evaluation.\text{term-of} :: \text{integer} \Rightarrow \text{term} \quad \Rightarrow \text{(Haskell-Quickcheck)}

\begin{align*}
\text{(let } & \{ \ t = \text{Typerep.Code}\text{-Numeral.integer []}; \\
& \quad \text{mkFunT} \ s \ t = \text{Typerep.fun} \ [s, t]; \\
& \quad \text{numT} = \text{Typerep.Num.num []}; \\
& \quad \text{mkBit} \ 0 = \text{Generated.Code} \cdot \text{Const Num.num.Bit0} \ (\text{mkFunT} \ \text{numT} \ \text{numT}); \\
& \quad \text{mkBit} \ 1 = \text{Generated.Code} \cdot \text{Const Num.num.Bit1} \ (\text{mkFunT} \ \text{numT} \ \text{numT}); \\
& \quad \text{mkNumeral} \ i = \text{let} \{ \ q = i \ 'Prelude.div' 2; r = i \ 'Prelude.mod' 2 \} \\
& \quad \text{in} \ \text{Generated.Code} \cdot \text{App} \ (\text{mkBit} \ r) \ (\text{mkNumeral} \ q); \\
& \quad \text{mkNumber} \ 0 = \text{Generated.Code} \cdot \text{Const Groups.zero'\text{-class.zero \ t}}; \\
& \quad \text{mkNumber} \ 1 = \text{Generated.Code} \cdot \text{Const Groups.one'\text{-class.one \ t}}; \\
& \quad \text{mkNumber} \ i = \text{if } i > 0 \text{ then} \\
& \quad \quad \text{Generated.Code} \cdot \text{App} \\
& \quad \quad \quad \text{(Generated.Code} \cdot \text{Const Num.numeral'\text{-class.numeral}} \ (\text{mkFunT} \ \text{numT} \ t)) \\
& \quad \quad \quad \text{(mkNumeral} \ i) \\
& \quad \text{else} \\
& \quad \quad \text{Generated.Code} \cdot \text{App} \\
& \quad \quad \quad \text{(Generated.Code} \cdot \text{Const Groups.uminus'\text{-class.uminus} (mkFunT \ t \ t))} \\
& \quad \quad \quad \text{(mkNumber} (-i)); \} \in \text{mkNumber})
\end{align*}

82.4 **The find-unused-assms command**

ML-file Tools/Quickcheck/find-unused-assms.ML

82.5 **Closing up**

hide-type narrowing-type narrowing-term narrowing-cons property
hide-const map-cons nth error toEnum marker empty Narrowing-cons conv non-empty ensure-testable all exists drawn-from around-zero
hide-const (open) Narrowing-variable Narrowing-constructor apply sum cons
hide-fact empty-def cons-def conv.siims non-empty.siims apply-def sum-def ensure-testable-def all-def exists-def

end

83 **Extraction: Program extraction for HOL**

theory Extraction
imports Datatype Option
begin
ML-file Tools/rewrite-hol-proof.ML

83.1 Setup

setup ⟨⟨
Extraction.add-types
[[bool, [[]], NONE]] #>
Extraction.set-preprocessor (fn thy => Proofterm.rewrite-proof-notypes
[[[], RewriteruleProof.elim-cong :: ProofRewriteRules.rprocs true]) o
Proofterm.rewrite-proof thy
(RewriteruleProof.rews,
ProofRewriteRules.rprocs true @ [ProofRewriteRules.expand-of-class thy]) o
ProofRewriteRules.elim-vars (curry Const @ {const-name default}))
⟩⟩

lemmas [extraction-expand] =
meta-spec atomize-eq atomize-all atomize-imp atomize-conj
allE rew-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
induct-forall-eq induct-implies-eq induct-equal-eq
induct-atomize induct-atomize' induct-rulify induct-rulify'
induct-rulify-fallback induct-trueI
True-implies-equals TrueE

lemmas [extraction-expand-def] =
induct-forall-def induct-implies-def induct-equal-def
induct-true-def induct-false-def

datatype sumbool = Left | Right

83.2 Type of extracted program

extract-type

typeof (Trueprop P) ≡ typeof P
typeof P ≡ Type (TYPE (Null)) →typeof Q ≡ Type (TYPE (Q)) →
typeof (P → Q) ≡ Type (TYPE (Q))
typeof Q ≡ Type (TYPE (Null)) →typeof (P → Q) ≡ Type (TYPE (Null))
typeof P ≡ Type (TYPE (P)) →typeof (Q → Type (TYPE (Q)) →
typeof (P → Q) ≡ Type (TYPE (P → Q))

(λx. typeof (P x)) ≡ (λx. Type (TYPE (Null))) →typeof (∀ x. P x) ≡ Type (TYPE (Null))

(λx. typeof (P x)) ≡ (λx. Type (TYPE (P))) →typeof (∀ x: 'a. P x) ≡ Type (TYPE ('a → 'P))
(λx. typeof (P x)) ≡ (λx. Type (TYPE(Null))) \rightarrow
\quad \text{typeof} (\exists x::'a. P x) \equiv \text{Type} (\text{TYPE('a)})

(λx. typeof (P x)) ≡ (λx. Type (TYPE('P))) \rightarrow
\quad \text{typeof} (\exists x::'a. P x) \equiv \text{Type} (\text{TYPE('a × 'P)})

typeof P ≡ Type (TYPE(Null)) \rightarrow typeof Q ≡ Type (TYPE(Null)) \rightarrow
\quad typeof (P \lor Q) \equiv \text{Type} (\text{TYPE(\text{Null})})
\quad typeof (P \lor Q) \equiv \text{Type} (\text{TYPE(\text{'Q option})})

typeof P ≡ Type (TYPE('P)) \rightarrow typeof Q ≡ Type (TYPE(Null)) \rightarrow
\quad typeof (P \lor Q) \equiv \text{Type} (\text{TYPE('P option}))
\quad typeof (P \lor Q) \equiv \text{Type} (\text{TYPE('P + 'Q)})

typeof P ≡ Type (TYPE(Null)) \rightarrow typeof Q ≡ Type (TYPE('Q)) \rightarrow
\quad typeof (P \land Q) \equiv \text{Type} (\text{TYPE('Q)})
\quad typeof (P \land Q) \equiv \text{Type} (\text{TYPE('P)})
\quad typeof (P \land Q) \equiv \text{Type} (\text{TYPE('P × 'Q)})

\quad typeof (P = Q) \equiv typeof ((P \rightarrow Q) \land (Q \rightarrow P))

\quad typeof (x \in P) \equiv typeof P

\textbf{83.3 Realizability}

\textbf{realizability}
\quad (\text{realizes} t (\text{Trueprop} P)) \equiv (\text{Trueprop} (\text{realizes} t P))
\quad (\text{typeof} P) \equiv (\text{Type} (\text{TYPE(Null)})) \rightarrow
\quad \text{realizes} t (P \rightarrow Q) \equiv (\text{realizes} \text{Null} P \rightarrow \text{realizes} t Q)
\quad (\text{typeof} P) \equiv (\text{Type} (\text{TYPE('P)})) \rightarrow
\quad \text{realizes} t (P \rightarrow Q) \equiv (\forall x::'P. \text{realizes} x P \rightarrow \text{realizes} \text{Null} Q)
\quad (\text{realizes} t (P \rightarrow Q)) \equiv (\forall x. \text{realizes} x P \rightarrow \text{realizes} (t x) Q)
\quad (\lambda x. \text{typeof} (P x)) \equiv (\lambda x. \text{Type} (\text{TYPE(Null)})) \rightarrow
\quad (\text{realizes} t (\forall x. P x)) \equiv (\forall x. \text{realizes} \text{Null} (P x))
(realizes t (∀ x. P x)) ≡ (∀ x. realizes (t x) (P x))

(λx. typeof (P x)) ≡ (λx. Type (TYPE(Null)))

(realizes t (∃ x. P x)) ≡ (realizes Null (P t))

(realizes t (∃ x. P x)) ≡ (realizes (snd t) (P (fst t)))

(typeof P) ≡ (Type (TYPE(Null)))

(realizes t (P ∨ Q)) ≡ (case t of Left ⇒ realizes Null P | Right ⇒ realizes Null Q)

(realizes t (P ∨ Q)) ≡ (case t of Inl p ⇒ realizes Null P | Inr q ⇒ realizes q P)

(realizes t (P ∨ Q)) ≡ (case t of None ⇒ realizes Null Q | Some p ⇒ realizes p P)

(realizes t (P ∨ Q)) ≡ (case t of None ⇒ realizes Null Q | Some p ⇒ realizes p P)

(typeof Q) ≡ (Type (TYPE(Null)))

(realizes t (P ∧ Q)) ≡ (realizes t P ∧ realizes t Q)

(realizes t (P ∧ Q)) ≡ (realizes t P ∧ realizes Null Q)

(realizes t (P ∧ Q)) ≡ (realizes (fst t) P ∧ realizes (snd t) Q)

typeof P ≡ Type (TYPE(Null))

(realizes t (¬ P)) ≡ (¬ realizes t P)

typeof P ≡ Type (TYPE'(P))

(realizes t (¬ P)) ≡ (∀ x::'P. ¬ realizes t P)

typeof (P::bool) ≡ Type (TYPE(Null))

(realizes t (P = Q)) ≡ (realizes (P = Q) (P = Q) (P = Q))

83.4 Computational content of basic inference rules

**Theorem** disjE-realizer:

**Assumes** r: case x of Inl p ⇒ P p | Inr q ⇒ Q q

**And** r1: λp. P p ⇒⇒ R (f p) and r2: λg. Q q ⇒⇒ R (g q)

**Shows** R (case x of Inl p ⇒ f p | Inr q ⇒ g q)
proof (cases x)
  case Inl
  with r show ?thesis by simp (rule r1)
next
  case Inr
  with r show ?thesis by simp (rule r2)
qed

theorem disjE-realizer2:
  assumes r: case x of None ⇒ P | Some q ⇒ Q q
  and r1: P ⇒ R f and r2: ∀q. Q q ⇒ R (g q)
  shows R (case x of None ⇒ f | Some q ⇒ g q)
proof (cases x)
  case None
  with r show ?thesis by simp (rule r1)
next
  case Some
  with r show ?thesis by simp (rule r2)
qed

theorem disjE-realizer3:
  assumes r: case x of Left ⇒ P | Right ⇒ Q
  and r1: P ⇒ R f and r2: Q ⇒ R g
  shows R (case x of Left ⇒ f | Right ⇒ g)
proof (cases x)
  case Left
  with r show ?thesis by simp (rule r1)
next
  case Right
  with r show ?thesis by simp (rule r2)
qed

theorem conjI-realizer:
  P p ⇒ Q q ⇒ P (fst (p, q)) ∧ Q (snd (p, q))
  by simp

theorem exI-realizer:
  P y x ⇒ P (snd (x, y)) (fst (x, y)) by simp

theorem exE-realizer:
  P (snd p) (fst p) ⇒
  (∀x y. P y x ⇒ Q (f x y)) ⇒ Q (let (x, y) = p in f x y)
  by (cases p) (simp add: Let-def)

theorem exE-realizer':
  P (snd p) (fst p) ⇒
  (∀x y. P y x ⇒ Q) ⇒ Q by (cases p) simp

realizers
  impl (P, Q): λpq. pq
  λ(c: -) (d: -) P Q pq (h: -). allI · · · c · (λx. impl · · · (h · x))
impI (P): Null
  λc: (P Q (h: -)) allI · · · c · (λx. impI · · · · (h · x))

impI (Q): λq. q λc: (P Q q. impI · · ·)

impI: Null impI

mp (P, Q): λpq. pq
  λc: (d: -) P Q pq (h: -) p. mp · · · · (spec · · · p · c · h)

mp (P): Null
  λc: (P Q (h: -)) p. mp · · · · (spec · · · p · c · h)

mp (Q): λq. q λ(c: -) P Q q. mp · · ·

mp: Null mp

allI (P): λp. λc: (d: -) P (P: allI · · · d)

allI: Null allI

spec (P): λx p. p x λc: (P x (d: -)) p. spec · · · x · d

spec: Null spec

exI (P): λx p. (x, p) λ(c: -) P x (d: -) p. exI-realizer · P · p · x · c · d

exI: λx. x λP x (c: -) (h: -). h

exE (P, Q): λp pq. let (x, y) = p in pq x y
  λ(c: -) (d: -) P Q (e: -) p (h: -) pq. exE-realizer · P · p · Q · pq · c · e · d · h

exE (P): Null
  λc: (P Q (d: -)) p. exE-realizer′ · · · · · · c · d

exE (Q): λx pq. pq x
  λ(c: -) P Q (d: -) x (h1: -) pq (h2: -). h2 · x · h1

exE: Null
  λP Q (c: -) x (h1: -) (h2: -). h2 · x · h1

cconjI (P, Q): Pair
  λ(c: -) (d: -) P Q p (h: -) q. conjI-realizer · P · p · Q · q · c · d · h

cconjI (P): λp. p
  λ(c: -) P Q p. conjI · · ·

conjI (Q): λq. q
\[ \lambda (c: -) \ P \ Q \ (h: -) \ q. \ conjI \ \cdot \ \cdot \ \cdot \ h \]

\textit{conjI}: \text{Null conjI}

\textit{conjunct1} \ (P, Q): \textit{fst}
\[ \lambda (c: -) \ (d: -) \ P \ Q \ pq. \ conjunct1 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct1} \ (P): \lambda p. \ p
\[ \lambda (c: -) \ P \ Q \ p. \ conjunct1 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct1} \ (Q): \text{Null}
\[ \lambda (c: -) \ P \ Q \ q. \ conjunct1 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct1}: \text{Null conjunct1}

\textit{conjunct2} \ (P, Q): \textit{snd}
\[ \lambda (c: -) \ (d: -) \ P \ Q \ pq. \ conjunct2 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct2} \ (P): \text{Null}
\[ \lambda (c: -) \ P \ Q \ p. \ conjunct2 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct2} \ (Q): \lambda p. \ p
\[ \lambda (c: -) \ P \ Q \ p. \ conjunct2 \ \cdot \ \cdot \ \cdot \]

\textit{conjunct2}: \text{Null conjunct2}

\textit{disjI1} \ (P, Q): \textit{Inl}
\[ \lambda (c: -) \ (d: -) \ P \ Q \ p. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sumbool.case-1} \ \cdot \ P \ \cdot \ \cdot \ p \ \cdot \ \text{arity-type-bool} \ \cdot \ c \ \cdot \ d) \]

\textit{disjI1} \ (P): \text{Some}
\[ \lambda (c: -) \ P \ Q \ p. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option.case-2} \ \cdot \ \cdot \ P \ \cdot \ p \ \cdot \ \text{arity-type-bool} \ \cdot \ c) \]

\textit{disjI1} \ (Q): \text{None}
\[ \lambda (c: -) \ P \ Q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option.case-1} \ \cdot \ \cdot \ \cdot \ \text{arity-type-bool} \ \cdot \ c) \]

\textit{disjI1}: \text{Left}
\[ \lambda P \ Q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sumbool.case-1} \ \cdot \ \cdot \ \cdot \ \text{arity-type-bool}) \]

\textit{disjI2} \ (P, Q): \textit{Inr}
\[ \lambda (d: -) \ (c: -) \ Q \ P \ q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sum.case-2} \ \cdot \ Q \ \cdot \ q \ \cdot \ \text{arity-type-bool} \ \cdot \ c \ \cdot \ d) \]

\textit{disjI2} \ (P): \text{None}
\[ \lambda (c: -) \ Q \ P. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option.case-1} \ \cdot \ \cdot \ \cdot \ \text{arity-type-bool} \ \cdot \ c) \]

\textit{disjI2} \ (Q): \text{Some}
\[ \lambda (c: -) \ Q \ P \ q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option.case-2} \ \cdot \ Q \ \cdot \ q \ \cdot \ \text{arity-type-bool} \ \cdot \ c) \]
\textbf{THEORY “Extraction”}  

\textit{disjE} \hspace{1em} (P, Q, R): \lambda pq \pr qr.
\textit{disjE} \hspace{1em} \text{case } pq \text{ of } \text{Inl } p \Rightarrow \pr p \mid \text{Inr } q \Rightarrow \pr q
\displaymath
\lambda (c: \cdot) (d: \cdot) P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) qr.
\textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot \pr h1 \cdot h2
\displaymath
\textit{disjE} \hspace{1em} (Q, R): \lambda pq \pr qr.
\text{case } pq \text{ of } \text{None } \Rightarrow \pr \mid \text{Some } q \Rightarrow \pr q
\lambda (c: \cdot) (d: \cdot) P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) qr.
\textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot \pr h1 \cdot h2
\displaymath
\textit{disjE} \hspace{1em} (P, R): \lambda pq \pr qr.
\text{case } pq \text{ of } \text{None } \Rightarrow \pr \mid \text{Right } \Rightarrow \pr
\lambda (c: \cdot) P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) qr.
\textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot \pr h1 \cdot h2
\displaymath
\textit{disjE} \hspace{1em} (P, Q): \text{Null}
\lambda (c: \cdot) (d: \cdot) P \ Q \ R \ pq \ . \textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot (\lambda x. \ R) \cdot \cdot \cdot c \cdot d \\
\cdot \text{arity-type-bool}
\textit{disjE} \hspace{1em} (Q): \text{Null}
\lambda (c: \cdot) P \ Q \ R \ pq \ . \textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot (\lambda x. \ R) \cdot \cdot \cdot c \cdot \text{arity-type-bool}
\textit{disjE} \hspace{1em} (P): \text{Null}
\lambda (c: \cdot) P \ Q \ R \ pq \ (h1: \cdot) (h2: \cdot) (h3: \cdot).
\textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot (\lambda x. \ R) \cdot \cdot \cdot c \cdot \text{arity-type-bool} \cdot h1 \cdot h3 \cdot h2
\textit{disjE}\hspace{1em} \text{Null}
\lambda P \ Q \ R \ pq \ . \textit{disjE}\text{-realizer} \hspace{1em} \lambda \cdot \ P \ Q \ R \pq \cdot (\lambda x. \ R) \cdot \cdot \cdot \text{arity-type-bool}
\text{FalseE} \hspace{1em} (P): \text{default}
\lambda (c: \cdot) P \ . \text{FalseE} \cdot
\text{FalseE}\hspace{1em} \text{Null} \text{FalseE}
\textit{notI} \hspace{1em} (P): \text{Null}
\lambda (c: \cdot) P \ (h: \cdot). \text{allI} \cdot \cdot \cdot c \cdot (\lambda x. \ notI \cdot \cdot \cdot (h \cdot x))
\textit{notI}\hspace{1em} \text{Null} \text{notI}
\textit{notE} \hspace{1em} (P, R): \lambda p. \text{default}
\lambda (c: \cdot) (d: \cdot) P \ R \ (h: \cdot) p \ . \text{notE} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h)
\textbf{notE (P): Null}
\[ \lambda(c: \_ \_\_) P R (h: \_\_\_) p. \text{notE} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h) \]

\textbf{notE (R): default}
\[ \lambda(c: \_ \_\_) P R. \text{notE} \cdot \cdot \cdot \]

\textbf{notE: Null notE}

\textbf{subst (P): \( \lambda s t ps \).
\[ \lambda(c: \_ \_\_) s t P (d: \_\_\_) (h: \_\_\_) ps. \text{subst} \cdot s \cdot t \cdot P \cdot ps \cdot d \cdot h \]

\textbf{subst: Null subst}

\textbf{iffD1 (P, Q): fst}
\[ \lambda(d: \_\_\_) (c: \_\_\_) Q P pq (h: \_\_\_) p.
\quad \text{mp} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot d \cdot (\text{conjunct1} \cdot \cdot \cdot h)) \]

\textbf{iffD1 (P): \( \lambda p. p \)
\[ \lambda(c: \_\_\_) Q P p (h: \_\_\_). \text{mp} \cdot \cdot \cdot (\text{conjunct1} \cdot \cdot \cdot h) \]

\textbf{iffD1 (Q): Null}
\[ \lambda(c: \_\_\_) Q P q1 (h: \_\_\_) q2.
\quad \text{mp} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot q2 \cdot c \cdot (\text{conjunct1} \cdot \cdot \cdot h)) \]

\textbf{iffD1: Null iffD1}

\textbf{iffD2 (P, Q): snd}
\[ \lambda(c: \_\_\_) (d: \_\_\_) P Q pq (h: \_\_\_) q.
\quad \text{mp} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot q \cdot d \cdot (\text{conjunct2} \cdot \cdot \cdot h)) \]

\textbf{iffD2 (P): \( \lambda p. p \)
\[ \lambda(c: \_\_\_) P Q p (h: \_\_\_). \text{mp} \cdot \cdot \cdot (\text{conjunct2} \cdot \cdot \cdot h) \]

\textbf{iffD2 (Q): Null}
\[ \lambda(c: \_\_\_) P Q q1 (h: \_\_\_) q2.
\quad \text{mp} \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot q2 \cdot c \cdot (\text{conjunct2} \cdot \cdot \cdot h)) \]

\textbf{iffD2: Null iffD2}

\textbf{iffI (P, Q): Pair}
\[ \lambda(c: \_\_\_) (d: \_\_\_) P Q pq (h1: \_\_\_) q p (h2: \_\_\_). \text{conjI-realizer} \cdot
\quad (\lambda pq. \forall x. P x \rightarrow Q (pq x)) \cdot pq \cdot
\quad (\lambda pq. \forall x. Q x \rightarrow P (pq x)) \cdot qp \cdot
\quad (\text{arity-type-fun} \cdot c \cdot d) \cdot
\quad (\text{arity-type-fun} \cdot d \cdot c) \cdot
\quad (\text{allI} \cdot \cdot \cdot c \cdot (\lambda x. \text{impI} \cdot \cdot \cdot (h1 \cdot x))) \cdot
\quad (\text{allI} \cdot \cdot \cdot d \cdot (\lambda x. \text{impI} \cdot \cdot \cdot (h2 \cdot x))) \]
iffI (P): λp. p
    λ(c: -) P Q (h1 : -) p (h2 : -). conjI · · · ·
    (allI · · c · (λx. impI · · · · (h1 · x))) ·
    (impI · · · · h2)

iffI (Q): λq. q
    λ(c: -) P Q q (h1 : -) (h2 : -). conjI · · · ·
    (impI · · · · h1) ·
    (allI · · c · (λx. impI · · · · (h2 · x)))

iffI: Null iff

end

84 Lifting-Sum: Setup for Lifting/Transfer for the sum type

theory Lifting-Sum
imports Lifting Basic-BNFs
begin

lemma sum-pred-inject [simp]:
    pred-sum P1 P2 (Inl a) = P1 a and pred-sum P1 P2 (Inr a) = P2 a
unfolding pred-sum-def fun-eq-iff sum-set-simps by auto

84.1 Transfer rules for the Transfer package

context
begin
interpretation lifting-syntax .

lemma Inl-transfer [transfer-rule]: (A ===> rel-sum A B) Inl Inl
unfolding rel-fun-def by simp

lemma Inr-transfer [transfer-rule]: (B ===> rel-sum A B) Inr Inr
unfolding rel-fun-def by simp

lemma case-sum-transfer [transfer-rule]:
    ((A ===> C) ===> (B ===> C) ===> rel-sum A B ===> C) case-sum
unfolding rel-fun-def rel-sum-def by (simp split: sum.split)

end
85  Coinduction: Coinduction Method

theory Coinduction
imports Ctr-Sugar
begin

ML-file Tools/coinduction.ML

setup Coinduction.setup

end

86  Record: Extensible records with structural sub-typing

theory Record
imports Quickcheck-Exhaustive
keywords record :: thy-decl
begin

86.1  Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification \( \alpha (\beta\text{-update } f \ rec) = \alpha \ rec \) for distinct fields \( \alpha \) and \( \beta \) of some record \( \alpha \ rec \) with \( n \) fields. There are \( n^2 \) such theorems, which prohibits storage of all of them for large \( n \). The rules can be proved on the fly by case decomposition and simplification in \( O(n) \) time. By creating \( O(n) \) isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in \( O(\log(n)^2) \) time.

The \( O(n) \) cost of case decomposition is not because \( O(n) \) steps are taken, but rather because the resulting rule must contain \( O(n) \) new variables and an \( O(n) \) size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields \( 'a, 'b, 'c \) and \( 'd \) might be introduced as isomorphic to \( ('a \times ('b \times ('c \times 'd))) \). If we balance the tuple tree to \( ('a \times 'b) \times ('c \times 'd) \) then accessors can be defined by converting to the underlying type then using \( O(\log(n)) \) fst or snd operations. Updators can be defined similarly, if we introduce a \( \text{fst-update} \) and \( \text{snd-update} \) function. Furthermore, we can prove the access/update theorem in \( O(\log(n)) \) steps by using simple rewrites on \( \text{fst}, \text{snd}, \text{fst-update} \) and \( \text{snd-update} \).

The catch is that, although \( O(\log(n)) \) steps were taken, the underlying type we converted to is a tuple tree of size \( O(n) \). Processing this term type wastes performance. We avoid this for large \( n \) by taking each subtree of size \( K \) and
defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these \(\frac{O(n)}{K}\) new types, or, if \(n > K \cdot K\), we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant \(K\).

If we prove the access/update theorem on this type with the analogous steps to the tuple tree, we consume \(O(\log(n)^2)\) time as the intermediate terms are \(O(\log(n))\) in size and the types needed have size bounded by \(K\). To enable this analogous traversal, we define the functions seen below: iso-tuple-fst, iso-tuple-snd, iso-tuple-fst-update and iso-tuple-snd-update. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule isomorphic-tuple.intros rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

86.2 Operators and lemmas for types isomorphic to tuples

```plaintext
datatype (′a, ′b, ′c) tuple-isomorphism = 
  Tuple-Isomorphism ′a ⇒ ′b × ′b × ′c ⇒ ′a

primrec
  repr :: (′a, ′b, ′c) tuple-isomorphism ⇒ ′b × ′c where
  repr (Tuple-Isomorphism r) = r

primrec
  abst :: (′a, ′b, ′c) tuple-isomorphism ⇒ ′b × ′c ⇒ ′a where
  abst (Tuple-Isomorphism r) = a

definition
  iso-tuple-fst :: (′a, ′b, ′c) tuple-isomorphism ⇒ ′a ⇒ ′b where
  iso-tuple-fst isom = fst \circ\ repr isom

definition
  iso-tuple-snd :: (′a, ′b, ′c) tuple-isomorphism ⇒ ′a ⇒ ′c where
  iso-tuple-snd isom = snd \circ\ repr isom

definition
  iso-tuple-fst-update ::
    (′a, ′b, ′c) tuple-isomorphism ⇒ (′b ⇒ ′b) ⇒ (′a ⇒ ′a) where
  iso-tuple-fst-update isom f = abst isom \circ\ apfst f \circ\ repr isom
```

definition
iso-tuple-update ::
(′a, ′b, ′c) tuple-isomorphism ⇒ (′c ⇒ ′e) ⇒ (′a ⇒ ′a) where
iso-tuple-update isom f = abst isom o apsnd f o repr isom

definition
iso-tuple-update-accessor-cong-assist ::
(′a ⇒ ′b) ⇒ (′a ⇒ ′a) ⇒ (′b ⇒ ′b) ⇒ (′a ⇒ ′b) ⇒ ′a ⇒ ′b ⇒ bool
where
iso-tuple-update-accessor-cong-assist upd ac v f v' x ←→
(∀ f v. upd (λx. f (ac v)) v = upd f v) ∧ (∀ v. upd id v = v)

definition
iso-tuple-update-accessor-eq-assist ::
(′a ⇒ ′b) ⇒ (′a ⇒ ′a) ⇒ (′b ⇒ ′b) ⇒ (′a ⇒ ′b) ⇒ (′a ⇒ ′a) ⇒ (′b ⇒ ′b) ⇒ bool
where
iso-tuple-update-accessor-eq-assist upd ac v f v' x ←→
 upd f v = v' ∧ ac v = x ∧ iso-tuple-update-accessor-cong-assist upd ac

lemma update-accessor-congruence-foldE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
and r: r = r' and v: ac r' = v'
and f: (λv. v' = v ⇒ f v = f' v)
shows upd f r = upd f' r'
using uac r v [symmetric]
apply (subgoal-tac upd (λx. f (ac r')) r' = upd (λx. f' (ac r')) r')
apply (simp add: iso-tuple-update-accessor-cong-assist-def)
apply (simp add: f)
done

lemma update-accessor-congruence-unfoldE:
iso-tuple-update-accessor-cong-assist upd ac ☑
r = r' ⇒ ac r' = v' ⇒ (λv. v' = v ⇒ f v = f' v) ⇒
 upd f r = upd f' r'
apply (erule(2) update-accessor-congruence-foldE)
apply simp
done
lemma iso-tuple-update-accessor-cong-assist-id:
iso-tuple-update-accessor-cong-assist upd ac \implies upd id = id
by rule (simp add: iso-tuple-update-accessor-cong-assist-def)

lemma update-accessor-noopE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
and ac: f (ac x) = ac x
shows upd f x = x
using uac
by (simp add: ac iso-tuple-update-accessor-cong-assist-id [OF uac, unfolded id-def]
cong: update-accessor-congruence-unfoldE [OF uac])

lemma update-accessor-noop-compE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
and ac: f (ac x) = ac x
shows upd (g \circ f) x = upd g x
by (simp add: ac cong: update-accessor-congruence-unfoldE [OF uac])

lemma update-accessor-cong-assist-idI:
iso-tuple-update-accessor-cong-assist id id
by (simp add: iso-tuple-update-accessor-cong-assist-def)

lemma update-accessor-cong-assist-triv:
iso-tuple-update-accessor-cong-assist upd ac \implies
iso-tuple-update-accessor-cong-assist upd ac
by assumption

lemma update-accessor-accessor-eqE:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies ac v = x
by (simp add: iso-tuple-update-accessor-eq-assist-def)

lemma update-accessor-updator-eqE:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies upd f v = v'
by (simp add: iso-tuple-update-accessor-eq-assist-def)

lemma iso-tuple-update-accessor-eq-assist-idI:
v' = f v \implies iso-tuple-update-accessor-eq-assist id id v f v' v
by (simp add: iso-tuple-update-accessor-eq-assist-idI update-accessor-cong-assist-idI)

lemma iso-tuple-update-accessor-eq-assist-triv:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies iso-tuple-update-accessor-eq-assist upd ac v f v' x
by assumption

lemma iso-tuple-update-accessor-cong-from-eq:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \implies iso-tuple-update-accessor-cong-assist upd ac
by (simp add: iso-tuple-update-accessor-eq-assist-def)
lemma iso-tuple-surjective-proof-assistI:
  \[ f x = y \implies \text{iso-tuple-surjective-proof-assist } x \, y \, f \]
  by (simp add: iso-tuple-surjective-proof-assist-def)

lemma iso-tuple-surjective-proof-assist-idE:
  iso-tuple-surjective-proof-assist x \, y \, \text{id} \implies x = y
  by (simp add: iso-tuple-surjective-proof-assist-def)

locale isomorphic-tuple =
  fixes isom :: ('a', 'b', 'c) \text{tuple-isomorphism}
  assumes repr-inv: \( \forall x. \text{abst isom} (\text{repr isom } x) = x \)
  and abst-inv: \( \forall y. \text{repr isom} (\text{abst isom } y) = y \)

begin

lemma repr-inj: \( \text{repr isom } x = \text{repr isom } y \iff x = y \)
  by (auto dest: arg-cong [of \text{repr isom } x \text{repr isom } y \text{abst isom}]
                 simp add: repr-inv)

lemma abst-inj: \( \text{abst isom } x = \text{abst isom } y \iff x = y \)
  by (auto dest: arg-cong [of \text{abst isom } x \text{abst isom } y \text{repr isom}]
                 simp add: abst-inv)

lemmas simps = Let-def repr-inv abst-inv repr-inj abst-inj

lemma iso-tuple-access-update-fst-fst:
  \( f \circ h \circ g = j \circ f \implies (f \circ \text{iso-tuple-fst isom}) \circ (\text{iso-tuple-fst-update isom } o \text{h}) \circ g =\)
  \( j \circ (f \circ \text{iso-tuple-fst isom}) \)
  by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-fst-def simps
       fun-eq-iff)

lemma iso-tuple-access-update-snd-snd:
  \( f \circ h \circ g = j \circ f \implies (f \circ \text{iso-tuple-snd isom}) \circ (\text{iso-tuple-snd-update isom } o \text{h}) \circ g =\)
  \( j \circ (f \circ \text{iso-tuple-snd isom}) \)
  by (clarsimp simp: iso-tuple-snd-update-def iso-tuple-snd-def simps
       fun-eq-iff)

lemma iso-tuple-access-update-fst-snd:
  \( (f \circ \text{iso-tuple-fst isom}) \circ (\text{iso-tuple-snd-update isom } o \text{h}) \circ g =\)
  \( \text{id} \circ (f \circ \text{iso-tuple-fst isom}) \)
  by (clarsimp simp: iso-tuple-snd-update-def iso-tuple-fst-def simps
       fun-eq-iff)

lemma iso-tuple-access-update-snd-fst:
  \( (f \circ \text{iso-tuple-snd isom}) \circ (\text{iso-tuple-fst-update isom } o \text{h}) \circ g =\)
  \( \text{id} \circ (f \circ \text{iso-tuple-snd isom}) \)
  by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-def simps
       fun-eq-iff)
**THEORY** “Record”

**lemma** iso-tuple-update-swap-fst-fst:

h f o j g = j g o h f \implies

(iso-tuple-fst-update isom o h) f o (iso-tuple-fst-update isom o j) g =

(iso-tuple-fst-update isom o j) g o (iso-tuple-fst-update isom o h) f

*by* (clarsimp simp: iso-tuple-fst-update-def simps apfst-compose fun-eq-iff)

**lemma** iso-tuple-update-swap-snd-snd:

h f o j g = j g o h f \implies

(iso-tuple-snd-update isom o h) f o (iso-tuple-snd-update isom o j) g =

(iso-tuple-snd-update isom o j) g o (iso-tuple-snd-update isom o h) f

*by* (clarsimp simp: iso-tuple-snd-update-def simps apsnd-compose fun-eq-iff)

**lemma** iso-tuple-update-swap-fst-snd:

(iso-tuple-snd-update isom o h) f o (iso-tuple-snd-update isom o j) g =

(iso-tuple-snd-update isom o j) g o (iso-tuple-snd-update isom o h) f


**lemma** iso-tuple-update-compose-fst-fst:

h f o j g = k (f o g) \implies

(iso-tuple-fst-update isom o h) f o (iso-tuple-fst-update isom o j) g =

(iso-tuple-fst-update isom o k) (f o g)

*by* (clarsimp simp: iso-tuple-fst-update-def simps apfst-compose fun-eq-iff)

**lemma** iso-tuple-update-compose-snd-snd:

h f o j g = k (f o g) \implies

(iso-tuple-snd-update isom o h) f o (iso-tuple-snd-update isom o j) g =

(iso-tuple-snd-update isom o k) (f o g)

*by* (clarsimp simp: iso-tuple-snd-update-def simps apsnd-compose fun-eq-iff)

**lemma** iso-tuple-surjective-proof-assist-step:

iso-tuple-surjective-proof-assist v a (iso-tuple-fst isom o f) \implies

iso-tuple-surjective-proof-assist v b (iso-tuple-snd isom o f) \implies

iso-tuple-surjective-proof-assist v (iso-tuple-cons isom a b) f


**lemma** iso-tuple-fst-update-accessor-cong-assist:

*assumes* iso-tuple-update-accessor-cong-assist f g

*shows* iso-tuple-update-accessor-cong-assist

(iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)

*proof* –
from assms have \( f \ id = id \)
  by (rule iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis\)
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
    iso-tuple-fst-update-def iso-tuple-fst-def)

qed

lemma iso-tuple-snd-update-accessor-cong-assist:
assumes iso-tuple-update-accessor-cong-assist \( f \ g \)
shows iso-tuple-update-accessor-cong-assist
  (iso-tuple-snd-update isom o \( f \)) (g o iso-tuple-snd isom)

proof –
  from assms have \( f \ id = id \)
  by (rule iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis\)
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
    iso-tuple-snd-update-def iso-tuple-snd-def)

qed

lemma iso-tuple-fst-update-accessor-eq-assist:
assumes iso-tuple-update-accessor-eq-assist \( f \ g \ a \ a' \ v \)
shows iso-tuple-update-accessor-eq-assist
  (iso-tuple-fst-update isom o \( f \)) (g o iso-tuple-fst isom)
  (iso-tuple-cons isom \( a \) \( b \)) u (iso-tuple-cons isom \( a' \) \( b \)) v

proof –
  from assms have \( f \ id = id \)
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
    intro: iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis\)
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
    iso-tuple-fst-update-def iso-tuple-fst-def
    iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)

qed

lemma iso-tuple-snd-update-accessor-eq-assist:
assumes iso-tuple-update-accessor-eq-assist \( f \ g \ b \ b' \ v \)
shows iso-tuple-update-accessor-eq-assist
  (iso-tuple-snd-update isom \( a \)) (g o iso-tuple-snd isom)
  (iso-tuple-cons isom \( a \) \( b \)) u (iso-tuple-cons isom \( a' \) \( b' \)) v

proof –
  from assms have \( f \ id = id \)
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
    intro: iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis\)
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
    iso-tuple-snd-update-def iso-tuple-snd-def
    iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)

qed
lemma iso-tuple-cons-conj-eqI:
  \(a = c \land b = d \land P \iff Q\)
  iso-tuple-cons isom a b = iso-tuple-cons isom c d \land P \iff Q
  by (clarsimp simp: iso-tuple-cons-def simps)

lemmas intros =
  iso-tuple-access-update-fst-fst
  iso-tuple-access-update-snd-snd
  iso-tuple-access-update-fst-snd
  iso-tuple-update-swap-fst-fst
  iso-tuple-update-swap-snd-snd
  iso-tuple-update-swap-fst-snd
  iso-tuple-update-swap-snd-fst
  iso-tuple-update-compose-fst-fst
  iso-tuple-update-compose-snd-snd
  iso-tuple-surjective-proof-assist-step
  iso-tuple-fst-update-accessor-eq-assist
  iso-tuple-snd-update-accessor-eq-assist
  iso-tuple-update-compose-snd-snd
  iso-tuple-update-compose-fst-fst

end

lemma isomorphic-tuple-intro:
  fixes repr abst
  assumes repr-inj: \(\forall x y.\) repr x = repr y \iff x = y
  and abst-inv: \(\forall z.\) repr (abst z) = z
  and v: v \equiv Tuple-Isomorphism repr abst
  shows isomorphic-tuple v
proof
  fix x have repr (abst (repr x)) = repr x
  by (simp add: abst-inv)
  then show Record.abst v (Record.repr v x) = x
  by (simp add: v repr-inj)
next
  fix y
  show Record.repr v (Record.abst v y) = y
  by (simp add: v) (fact abst-inv)
qed

definition
tuple-iso-tuple \equiv Tuple-Isomorphism id id

lemma tuple-iso-tuple:
  isomorphic-tuple tuple-iso-tuple
  by (simp add: isomorphic-tuple-intro [OF - - reflexive] tuple-iso-tuple-def)
lemma refl-conj-eq: \( Q = R \iff P \land Q \iff P \land R \)
by simp

lemma iso-tuple-UNIV-I: \( x \in \text{UNIV} \equiv \text{True} \)
by simp

lemma iso-tuple-True-simp: \((\text{True} \implies \text{PROP } P) \equiv \text{PROP } P\)
by simp

lemma prop-subst: \( s = t \implies \text{PROP } P \, t \equiv \text{PROP } P \, s\)
by simp

lemma K-record-comp: \((\lambda x. \, c) \circ f = (\lambda x. \, c)\)
by (simp add: comp-def)

86.4 Concrete record syntax

nonterminal
ident and
field-type and
field-types and
field and
fields and
field-update and
field-updates

syntax
-constify :: id => ident (-)
-constify :: longid => ident (-)
-field-type :: ident => type => field-type ((2- :/ -))
:: field-type => field-types (-)
-field-types :: field-type => field-types => field-types (-/-)
-record-type :: field-types => type ((3'(| - '))
-record-type-scheme :: field-types => type => type ((3'(| -/ (.2... :/ -) |')))
-field :: ident => 'a => field ((2- =/ -))
:: field => fields (-)
-fields :: field => fields => fields (-/-)
-record :: fields => 'a ((3'(| - '))
-record-scheme :: fields => 'a => 'a ((3'(| -/ (.2... =/ -) |')))
-field-update :: ident => 'a => field-update ((2- :=/ -))
:: field-update => field-updates (-)
-field-updates :: field-update => field-updates => field-updates (-/-)
-record-update :: 'a => field-updates => 'b (-/(3'(| - ')) [900, 0] 900)

syntax (xsymbols)
-record-type :: field-types => type ((3'[-]))
 THEORY "Nitpick"  

-record-type-scheme :: field-types => type => type ((3|/- (2...::/-)))
-record :: fields => 'a ((3|/- (2...::/-)))
-record-scheme :: fields => 'a => 'a ((3|/- (2...::/-)))
-record-update :: 'a => field-updates => 'b (-/(3|) [900, 0] 900)

86.5 Record package

ML-file Tools/record.ML

hide-const (open) Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd
iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons
iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist
iso-tuple-update-accessor-eq-assist tuple-iso-tuple

end

87 Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory Nitpick
imports Record
keywords
  nitpick :: diag and
  nitpick-params :: thy-decl
begin

typedecl bisim-iterator

axiomatization unknown :: 'a
  and is-unknown :: 'a => bool
  and bisim :: bisim-iterator => 'a => 'a => bool
  and bisim-iterator-max :: bisim-iterator
  and Quot :: 'a => 'b
  and safe-The :: ('a => bool) => 'a

datatype ('a, 'b) fun-box = FunBox ('a => 'b)
datatype ('a, 'b) pair-box = PairBox 'a 'b

typedecl unsigned-bit
typedecl signed-bit

datatype 'a word = Word ('a set)

Alternative definitions.

lemma Ex1-unfold [nitpick-unfold]:
  Ex1 P ≡ ∃x. {x. P x} = {x}
apply (rule eq-reflection)
apply (simp add: Ex1-def set-eq-iff)


apply (rule iffI)
apply (erule exE)
apply (erule conjE)
apply (rule-tac x = x in exI)
apply (rule allI)
apply (rename-tac y)
apply (erule-tac x = y in allE)
by auto

lemma rtrancl-unfold [nitpick-unfold]:
r∗≡(r+)=
by (simp only: rtrancl-trancl-reflcl)

lemma rtranclp-unfold [nitpick-unfold]:
rtranclp r a b≡(a=b∨tranclp r a b)
by (rule eq-reflection) (auto dest: rtranclpD)

lemma tranclp-unfold [nitpick-unfold]:
tranclp r a b≡(a,b)∈trancl {(x,y). r x y}
by (simp add: trancl-def)

lemma [nitpick-simp]:
of-nat n= (if n=0 then 0 else 1+of-nat (n-1))
by (cases n) auto

definition prod :: 'a set ⇒ 'b set ⇒ ('a × 'b) set where
prod A B = {(a, b). a ∈ A ∧ b ∈ B}

definition refl' :: ('a × 'a) set ⇒ bool where
refl' r≡∀x. (x, x)∈r

definition wf' :: ('a × 'a) set ⇒ bool where
wf' r≡acyclic r ∧ (finite r ∨ unknown)

definition card' :: 'a set ⇒ nat where
card' A≡if finite A then length (SOME xs. set xs = A ∧ distinct xs) else 0

definition setsum' :: ('a ⇒ 'b::comm-monoid-add) ⇒ 'a set ⇒ 'b where
setsum' f A≡if finite A then listsum (map f (SOME xs. set xs = A ∧ distinct xs)) else 0

inductive fold-graph' :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool where
fold-graph' f z {} z |
[| x ∈ A; fold-graph' f z (A - {x}) y |] ⇒ fold-graph' f z A (f x y)

The following lemmas are not strictly necessary but they help the specialize optimization.

lemma The-psimp [nitpick-psimp]:
P = (op =) x⇒The P = x
by auto
lemma Eps-psimp [nitpick-psimp]:
\[ \text{Eps } P \, x \Rightarrow \text{Eps } P = y \Rightarrow \text{Eps } P = x \]
apply (cases P (Eps P))
apply auto
apply (erule contrapos-np)
by (rule someI)

lemma case-unit-unfold [nitpick-unfold]:
\[ \text{case-unit } x u \equiv x \]
apply (subgoal-tac u = ())
apply (simp only: unit_case)
by simp

declare unit_case [nitpick-simp del]

lemma case-nat-unfold [nitpick-unfold]:
\[ \text{case-nat } x f n \equiv \text{if } n = 0 \text{ then } x \text{ else } f (n - 1) \]
apply (rule eq-reflection)
by (cases n) auto

declare nat_case [nitpick-simp del]

lemma size-list-simp [nitpick-simp]:
\[ \text{size-list } f \, xs = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (f (\text{hd } xs) + \text{size-list } f (\text{tl } xs))) \]
size xs = (if xs = [] then 0 else Suc (size (tl xs)))
by (cases xs) auto

Auxiliary definitions used to provide an alternative representation for \textit{rat} and \textit{real}.

function nat-gcd :: nat \Rightarrow nat \Rightarrow nat where
[simp del]: nat-gcd x y = (if y = 0 then x else nat-gcd y (x mod y))
by auto
termination
apply (relation measure (λ(x, y). x + y + (if y > x then 1 else 0)))
apply auto
apply (metis mod-less-divisor xt1(9))
by (metis mod-mod-trivial mod-self nat-neq-iff xt1(10))

definition nat-lcm :: nat \Rightarrow nat \Rightarrow nat where
nat-lcm x y = x * y div (nat-gcd x y)

definition int-gcd :: int \Rightarrow int \Rightarrow int where
int-gcd x y = int (nat-gcd (nat (abs x)) (nat (abs y)))

definition int-lcm :: int \Rightarrow int \Rightarrow int where
int-lcm x y = int (nat-lcm (nat (abs x)) (nat (abs y)))

definition Frac :: int \times int \Rightarrow bool where
\( Frac \equiv \lambda(a, b). b > 0 \land \text{int-gcd } a b = 1 \)

axiomatization
\[
\begin{align*}
\text{Abs-Frac} &:: \text{int} \times \text{int} \Rightarrow 'a \text{ and} \\
\text{Rep-Frac} &:: 'a \Rightarrow \text{int} \times \text{int}
\end{align*}
\]

definition zero-frac :: 'a where
\( \text{zero-frac} \equiv \text{Abs-Frac} (0, 1) \)

definition one-frac :: 'a where
\( \text{one-frac} \equiv \text{Abs-Frac} (1, 1) \)

definition num :: 'a \Rightarrow \text{int} where
\( \text{num} \equiv \text{fst} \circ \text{Rep-Frac} \)

definition denom :: 'a \Rightarrow \text{int} where
\( \text{denom} \equiv \text{snd} \circ \text{Rep-Frac} \)

function norm-frac :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{int} where
\[
\begin{align*}
\text{simp del}: \text{norm-frac } a b &= (\text{if } b < 0 \text{ then } \text{norm-frac } (-a) (-b) \\
&\quad \text{else if } a = 0 \lor b = 0 \text{ then } (0, 1) \\
&\quad \text{else let } c = \text{int-gcd } a b \text{ in } (a \text{ div } c, b \text{ div } c))
\end{align*}
\]
by pat-completeness auto
termination by (relation measure (\(\lambda(\cdot, b). \text{if } b < 0 \text{ then } 1 \text{ else } 0\)) auto

definition frac :: \text{int} \Rightarrow \text{int} \Rightarrow 'a where
\( \text{frac } a b \equiv \text{Abs-Frac} (\text{norm-frac } a b) \)

definition plus-frac :: 'a \Rightarrow 'a \Rightarrow 'a where
\[
\begin{align*}
\text{nitpick-simp}: \text{plus-frac } q r &= (\text{let } d = \text{int-lcm } (\text{denom } q) (\text{denom } r) \text{ in} \\
&\quad \text{frac } (\text{num } q \ast (d \text{ div } \text{denom } q) + \text{num } r \ast (d \text{ div } \text{denom } r)) \ast d)
\end{align*}
\]

definition times-frac :: 'a \Rightarrow 'a \Rightarrow 'a where
\[
\begin{align*}
\text{nitpick-simp}: \text{times-frac } q r &= \text{frac } (\text{num } q \ast \text{num } r) (\text{denom } q \ast \text{denom } r)
\end{align*}
\]

definition uminus-frac :: 'a \Rightarrow 'a where
\( \text{uminus-frac } q \equiv \text{Abs-Frac } (\text{-num } q, \text{denom } q) \)

definition number-of-frac :: \text{int} \Rightarrow 'a where
\( \text{number-of-frac } n \equiv \text{Abs-Frac } (n, 1) \)

definition inverse-frac :: 'a \Rightarrow 'a where
\( \text{inverse-frac } q \equiv \text{frac } (\text{denom } q) (\text{num } q) \)

definition less-frac :: 'a \Rightarrow 'a \Rightarrow \text{bool} where
\[
\begin{align*}
\text{nitpick-simp}: \text{less-frac } q r &\iff \text{num } (\text{plus-frac } q (\text{uminus-frac } r)) < 0
\end{align*}
\]
**THEORY “Nitpick”**

**definition** less-eq-frac :: 'a ⇒ 'a ⇒ bool where

```
less-eq-frac q r ←→ num (plus-frac q (uminus-frac r)) ≤ 0
```

**definition** of-frac :: 'a ⇒ 'b :: {inverse, ring-1} where

```
of-frac q ≡ of-int (num q) / of-int (denom q)
```

**axiomatization** wf-wfrec :: ('a × 'a) set ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b

```
wf-wfrec 'R F x = F (cut (wf-wfrec 'R F) R x) x
```

**definition** wfrec' :: ('a × 'a) set ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b where

```
wfrec' 'R F x ≡ if wf 'R then wf-wfrec 'R F x
else THE y. wfrec-rel 'R (% f x. F (cut f 'R x) x) x y
```

**ML-file** Tools/Nitpick/kodkod.ML

**ML-file** Tools/Nitpick/kodkod-sat.ML

**ML-file** Tools/Nitpick/nitpick-util.ML

**ML-file** Tools/Nitpick/nitpick-hol.ML

**ML-file** Tools/Nitpick/nitpick-mono.ML

**ML-file** Tools/Nitpick/nitpick-preproc.ML

**ML-file** Tools/Nitpick/nitpick-scope.ML

**ML-file** Tools/Nitpick/nitpick-peephole.ML

**ML-file** Tools/Nitpick/nitpick-rep.ML

**ML-file** Tools/Nitpick/nitpick-nat.ML

**ML-file** Tools/Nitpick/nitpick-kodkod.ML

**ML-file** Tools/Nitpick/nitpick-model.ML

**ML-file** Tools/Nitpick/nitpick.ML

**ML-file** Tools/Nitpick/nitpick-commands.ML

**ML-file** Tools/Nitpick/nitpick-tests.ML

**setup** «

```
Nitpick-HOL.register-ersatz-global
[(@{const-name card}, @{const-name card'}),
 (@{const-name setsum}, @{const-name setsum'}),
 (@{const-name fold-graph}, @{const-name fold-graph'}),
 (@{const-name wf}, @{const-name wf'}),
 (@{const-name wf-wfrec}, @{const-name wf-wfrec'}),
 (@{const-name wfrec}, @{const-name wfrec'})]
```

**hide-const** (open) unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox ParBox Word prod refl' wf' card' setsum' fold-graph' nat-gcd nat-lcm int-gcd int-lcm Frac Abs-Frac Rep-Frac zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac number-of-frac inverse-frac less-frac less-eq-frac of-frac wf-wfrec wf-wfrec wfrec'
hide-type (open) bisim-iterator fun-box pair-box unsigned-bit signed-bit word

end

88  BNF-GFP: Greatest Fixed Point Operation on Bounded Natural Functors

theory BNF-GFP
imports BNF-FP-Base String
keywords
  codatatype :: thy-decl and
  primcorecursive :: thy-goal and
  primcorec :: thy-decl
begin

setup ⟨⟨ Sign.const-alias @{\{ binding proj \} @{\{ const-name Equiv-Relations.proj \} } } ⟩⟩

lemma one-pointE: \[ \forall x. s = x \implies P \] \implies P
by simp

lemma obj-sumE: \[ \forall x. s = Inl x \implies P; \forall x. s = Inr x \implies P \] \implies P
by (cases s) auto

lemma not-TrueE: \neg True \implies P
by (erule notE, rule TrueI)

lemma neq-eq-eq-contradict: \[ t \neq u; s = t; s = u \] \implies P
by fast

lemma case-sum-expand-Inr: \( f \circ \text{Inl} = g \implies f \circ x = \text{case-sum} g (f \circ \text{Inr}) x \)
by (auto split: sum.splits)

lemma case-sum-expand-Inr': \( f \circ \text{Inl} = g \implies h = f \circ \text{Inr} \longleftrightarrow \text{case-sum} g h = f \)
apply rule
apply (rule ext, force split: sum.split)
bys (rule ext, metis case-sum-o-inj(2))

lemma converse-Times: \((A \times B) \;
\neg 1 = B \times A\)
by fast

lemma equiv-proj:
  assumes e: equiv A R and z ∈ R
  shows (proj R o fst) z = (proj R o snd) z
proof
  from assms(2) have z: (fst z, snd z) ∈ R by auto
  with e have \( \forall x. (fst z, x) ∈ R \implies (snd z, x) ∈ R \implies (fst z, x) ∈ R \) by blast
  unfolding equiv-def sym-def trans-def by blast+
  then show ?thesis unfolding proj-def abs-def by auto
qed

definition image2 where image2 A f g = \{ (f a, g a) \mid a. a ∈ A \}

lemma Id-on-Gr: Id-on A = Gr A id
unfolding Id-on-def Gr-def by auto

lemma image2-eqI: \[ b = f x; c = g x; x ∈ A \] \implies (b, c) ∈ image2 A f g
unfolding image2-def by auto

lemma IdD: (a, b) ∈ Id \implies a = b
by auto

lemma image2-Gr: image2 A f g = (Gr A f) ^-1 O (Gr A g)
unfolding image2-def Gr-def by auto

lemma GrD1: (x, fx) ∈ Gr A f \implies x ∈ A
unfolding Gr-def by simp

lemma GrD2: (x, fx) ∈ Gr A f \implies f x = fx
unfolding Gr-def by simp

lemma Gr-incl: Gr A f ⊆ A <+> B \iff f ' A ⊆ B
unfolding Gr-def by auto

lemma subset-Collect-iff: B ⊆ A \implies (B ⊆ \{ x ∈ A. P x \}) = (\forall x ∈ B. P x)
by blast

lemma subset-CollectI: B ⊆ A \implies (\exists x, x ∈ B \implies Q x \implies P x) \implies (\{ x ∈ B. Q x \} ⊆ \{ x ∈ A. P x \})
by blast

lemma in-rel-Collect-split-eq: in-rel (Collect (split X)) = X
unfolding fun-eq-iff by auto

lemma Collect-split-in-rel-leI: X ⊆ Y \implies X ⊆ Collect (split (in-rel Y))
by auto
lemma Collect-split-in-rel-leE: \( X \subseteq \text{Collect} (\text{split} (\text{in-rel} Y)) \implies (X \subseteq Y \implies R) \implies R \) 
by force

lemma conversep-in-rel: \((\text{in-rel} R)^{-1}\) = \(\text{in-rel} (R^{-1})\) 
unfolding fun-eq-iff by auto

lemma relcompp-in-rel: \(\text{in-rel} R \circ \text{in-rel} S = \text{in-rel} (R \circ S)\) 
unfolding fun-eq-iff by auto

lemma in-rel-Gr: \(\text{in-rel} (\text{Gr} A f) = \text{Grp} A f\) 
unfolding Gr-def Grp-def fun-eq-iff by auto

definition relImage where
relImage R f ≡ \{ (f a1, f a2) | a1 a2. (a1,a2) ∈ R \}

definition relInvImage where
relInvImage A R f ≡ \{ (a1, a2) | a1 a2. a1 ∈ A ∧ a2 ∈ A ∧ (f a1, f a2) ∈ R \}

lemma relImage-Gr: \(\{R \subseteq A \times A\} \implies \text{relImage} R f = (\text{Gr} A f)^\sim O R O \text{Gr} A f\) 
unfolding relImage-def Gr-def relcomp-def by auto

lemma relInvImage-Gr: \(\{R \subseteq B \times B\} \implies \text{relInvImage} A R f = \text{Gr} A f O R O (\text{Gr} A f)^\sim\) 
unfolding Gr-def relcomp-def image-def relInvImage-def by auto

lemma relImage-mono: \(R_1 \subseteq R_2 \implies \text{relImage} R_1 f \subseteq \text{relImage} R_2 f\) 
unfolding relImage-def by auto

lemma relInvImage-mono: \(R_1 \subseteq R_2 \implies \text{relInvImage} A R_1 f \subseteq \text{relInvImage} A R_2 f\) 
unfolding relInvImage-def by auto

lemma relInvImage-Id-on: \((\forall a1 a2. f a1 = f a2 \leftrightarrow a1 = a2) \implies \text{relInvImage} A (\text{Id-on} B) f \subseteq \text{Id}\) 
unfolding relInvImage-def Id-on-def by auto

lemma relInvImage-UNIV-relImage: \(R \subseteq \text{relInvImage} \text{UNIV} (\text{relImage} R f) f\) 
unfolding relInvImage-def relImage-def by auto

lemma relImage-proj: assumes equiv A R 
shows \(\text{relImage} R (\text{proj} R) \subseteq \text{Id-on} (A|/R)\) 
unfolding relImage-def Id-on-def 
using proj-iff[OF assms] equiv-class-eq-iff[OF assms]
by (auto simp: proj-preserves)

lemma relImage-relInvImage:
assumes \( R \subseteq f \cdot A \leftrightarrow f \cdot A \)
shows relImage (relInvImage A R f) f = R
using assms unfolding relImage-def relInvImage-def by fast

lemma subst-Pair: \( P \cdot x \cdot y \Rightarrow a = (x, y) \Rightarrow P \cdot (\text{fst } a) \cdot (\text{snd } a) \)
by simp

lemma fst-diag-id: \( (\text{fst} \circ \% x. (x, x)) \circ \text{id} z ) = \text{id} z \)
by simp

lemma snd-diag-id: \( (\text{snd} \circ \% x. (x, x)) \circ \text{id} z ) = \text{id} z \)
by simp

lemma fst-diag-fst: \( \text{fst} \circ \% x. (x, x) \circ \text{fst} = \text{fst} \)
by auto

lemma snd-diag-fst: \( \text{snd} \circ \% x. (x, x) \circ \text{fst} = \text{fst} \)
by auto

lemma fst-diag-snd: \( \text{fst} \circ \% x. (x, x) \circ \text{snd} = \text{snd} \)
by auto

lemma snd-diag-snd: \( \text{snd} \circ \% x. (x, x) \circ \text{snd} = \text{snd} \)
by auto

definition Succ where \( \text{Succ } Kl kl = \{ k \cdot kl @ [k] \in Kl \} \)
definition Shift where \( \text{Shift } Kl k = \{ kl \cdot k # kl \in Kl \} \)
definition shift where \( \text{shift } lab k = (\lambda kl. lab (k # kl)) \)

lemma empty-Shift: \( [] \in Kl; k \in \text{Succ } Kl [] \Rightarrow [] \in \text{Shift } Kl k \)
unfolding Shift-def Succ-def by simp

lemma SuccD: \( k \in \text{Succ } Kl kl \Rightarrow kl @ [k] \in Kl \)
unfolding Succ-def by simp

lemmas SuccE = SuccD[elim-format]

lemma SuccI: \( kl @ [k] \in Kl \Rightarrow k \in \text{Succ } Kl kl \)
unfolding Succ-def by simp

lemma ShiftD: \( kl \in \text{Shift } Kl k \Rightarrow k # kl \in Kl \)
unfolding Shift-def by simp

lemma Succ-Shift: \( \text{Succ } (\text{Shift } Kl k) kl = \text{Succ } Kl (k # kl) \)
unfolding Succ-def Shift-def by auto

lemma length-Cons: \( \text{length } (x # xs) = \text{Suc } (\text{length } xs) \)
by simp

lemma length-append-singleton: \( \text{length } (xs @ [x]) = \text{Suc } (\text{length } xs) \)
by simp

definition toCard-pred A r f \equiv \text{inj-on } f A \land f ' A \subseteq \text{Field } r \land \text{Card-order } r
definition toCard A r \equiv \text{SOME } f. \text{toCard-pred } A r f
**THEORY “BNF-GFP”**

**lemma** ex-toCard-pred:
\[ |A| \preceq r; \text{Card-order } r] \implies \exists f. \text{toCard-pred } A r f \]

**unfolding** toCard-pred-def
**using** card-of-ordLeq[of A Field r]
ordLeq-ordIso-trans[of - card-of-unique[of Field r], of |A|]
by blast

**lemma** toCard-pred-toCard:
\[ |A| \preceq r; \text{Card-order } r] \implies \text{toCard-pred } A r (\text{toCard } A r) \]

**unfolding** toCard-def
**using** someI-ex[of ex-toCard-pred]

**lemma** toCard-inj:
\[ |A| \preceq r; \text{Card-order } r; x \in A; y \in A] \implies \toCard A r x = \toCard A r y \iff x = y \]
**using** toCard-pred-toCard unfolding inj-on-def toCard-pred-def by blast

**definition** fromCard A r k \equiv \text{SOME } b. b \in A \land \text{toCard } A r b = k

**lemma** fromCard-toCard:
\[ |A| \preceq r; \text{Card-order } r; b \in A] \implies \text{fromCard } A r (\text{toCard } A r b) = b \]
**unfolding** fromCard-def by (rule some-equality) (auto simp add: toCard-inj)

**lemma** Inl-Field-csum:
a \in \text{Field } r \implies \text{Inl } a \in \text{Field } (r + c s)
**unfolding** Field-card-of csum-def by auto

**lemma** Inr-Field-csum:
a \in \text{Field } s \implies \text{Inr } a \in \text{Field } (r + c s)
**unfolding** Field-card-of csum-def by auto

**lemma** rec-nat-0-imp:
f = rec-nat f1 (\% n rec. f2 n rec) \implies f 0 = f1
by auto

**lemma** rec-nat-Suc-imp:
f = rec-nat f1 (\% n rec. f2 n rec) \implies f (\text{Suc } n) = f2 n (f n)
by auto

**lemma** rec-list-Nil-imp:
f = rec-list f1 (\% xs rec. f2 x xs rec) \implies f [] = f1
by auto

**lemma** rec-list-Cons-imp:
f = rec-list f1 (\% xs rec. f2 x xs rec) \implies f (x # xs) = f2 x xs (f xs)
by auto

**lemma** not-arg-cong-Inr:
x \neq y \implies \text{Inr } x \neq \text{Inr } y
by simp

**lemma** Collect-splitD:
x \in \text{Collect } (\text{split } A) \implies A (fst x) (snd x)
by auto

**definition** image2p where
image2p f g R = (\lambda x y. \exists x' y'. R x' y' \land f x' = x \land g y' = y)
lemma image2pI: R x y --> image2p f g R (f x) (g y)
  unfolding image2p-def by blast

lemma image2pE: (image2p f g R (f x) (g y) --> P)
  unfolding image2p-def by blast

lemma rel-fun-iff-geq-image2p: rel-fun R S f g = (image2p f g R <= S)
  unfolding rel-fun-def image2p-def by auto

lemma rel-fun-image2p: rel-fun R (image2p f g R) f g
  unfolding rel-fun-def image2p-def by auto

88.1 Equivalence relations, quotients, and Hilbert’s choice

lemma equiv-Eps-in:
  [equiv A r; X ∈ A//r] ==> Eps (%. x ∈ X) ∈ X
  apply (rule some12-ex)
  using in-quotient-imp-non-empty by blast

lemma equiv-Eps-preserves:
  assumes ECH: equiv A r and X ∈ A//r
  shows Eps (%. x ∈ X) ∈ A
  apply (rule in-mono[rule-format])
  using assms apply (rule in-quotient-imp-subset)
  by (rule equiv-Eps-in) (rule assms)+

lemma proj-Eps:
  assumes equiv A r and X ∈ A//r
  shows proj r (Eps (%. x ∈ X)) = X
  unfolding proj-def proof auto
  fix x assume x: x ∈ X
  thus (Eps (%. x ∈ X), x) ∈ r using assms equiv-Eps-in in-quotient-imp-in-rel
  by fast
next
  fix x assume (Eps (%. x ∈ X),x) ∈ r
  thus x ∈ X using in-quotient-imp-closed[OF assms equiv-Eps-in[OF assms]] by fast
qed

definition univ where univ f X == f (Eps (%. x ∈ X))

lemma univ-commute:
  assumes ECH: equiv A r and RES: f respects r and x: x ∈ A
  shows (univ f) (proj r x) = f x
  unfolding univ-def proof --
  have prj: proj r x ∈ A//r using x proj-preserves by fast
  hence Eps (%. y ∈ proj r x) ∈ A using ECH equiv-Eps-preserves by fast
moreover have \( \text{proj } r (Eps (\% y. y \in \text{proj } r x)) = \text{proj } r x \) using ECH \text{proj-Eps by fast}

ultimately have \( (x, Eps (\% y. y \in \text{proj } r x)) \in r \) using \( x \) ECH \text{proj-iff by fast}

thus \( f (Eps (\% y. y \in \text{proj } r x)) = f x \) using \( RES \) unfolding \text{congruent-def by fastforce}

qed

lemma \text{univ-preserves}:
assumes \( ECH: \text{equiv } A r \) and \( RES: f \text{ respects } r \) and
\( PRES: \forall x \in A. f x \in B \)
shows \( \forall X \in A//r. \text{univ } f X \in B \)
proof
  fix \( X \) assume \( X \in A//r \)
  then obtain \( x \) where \( x \in A \) and \( X: X = \text{proj } r x \) using \( ECH \) \text{proj-image[of r A]} by blast
  hence \text{univ } f X = f x \) using \text{assms univ-commute by fastforce}
  thus \text{univ } f X \in B using \( x \) \( PRES \) by simp
qed

ML-file Tools/BNF/bnf-gfp-util.ML
ML-file Tools/BNF/bnf-gfp-tactics.ML
ML-file Tools/BNF/bnf-gfp.ML
ML-file Tools/BNF/bnf-gfp-rec-sugar-tactics.ML
ML-file Tools/BNF/bnf-gfp-rec-sugar.ML

end

89 SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers

theory SMT
imports Record
keywords smt-status :: diag
begin

ML-file Tools/SMT/smt-utils.ML
ML-file Tools/SMT/smt-failure.ML
ML-file Tools/SMT/smt-config.ML

89.1 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are
considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

\texttt{typedef\ pattern}

\texttt{consts}

\texttt{pat} :: \texttt{'}a \Rightarrow \texttt{pattern} \\
\texttt{nopat} :: \texttt{'}a \Rightarrow \texttt{pattern}

\texttt{definition trigger} :: \texttt{pattern\ list\ list} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool\ where} \texttt{trigger} - \texttt{P} = \texttt{P}

89.2 Quantifier weights

Weight annotations to quantifiers influence the priority of quantifier instantiations. They should be handled with care for solvers, which support them, because incorrect choices of weights might render a problem unsolvable.

\texttt{definition weight} :: \texttt{int} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool\ where} \texttt{weight} - \texttt{P} = \texttt{P}

Weights must be non-negative. The value \texttt{0} is equivalent to providing no weight at all.

Weights should only be used at quantifiers and only inside triggers (if the quantifier has triggers). Valid usages of weights are as follows:

- \( \forall x.\ \text{trigger} \ [[\text{pat} \ (P\ x)]] \ (\text{weight} \ 2 \ (P\ x)) \)
- \( \forall x.\ \text{weight} \ 3 \ (P\ x) \)

89.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

\texttt{definition fun-app\ where} \texttt{fun-app\ f} = \texttt{f}

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

\texttt{lemmas array-rules =} \texttt{ext\ fun-upd-apply\ fun-upd-same\ fun-upd-other\ fun-upd-upd\ fun-app-def}

89.4 First-order logic

Some SMT solvers only accept problems in first-order logic, i.e., where formulas and terms are syntactically separated. When translating higher-order
into first-order problems, all uninterpreted constants (those not built-in in the target solver) are treated as function symbols in the first-order sense. Their occurrences as head symbols in atoms (i.e., as predicate symbols) are turned into terms by logically equating such atoms with True. For technical reasons, True and False occurring inside terms are replaced by the following constants.

**definition** term-true where term-true = True
**definition** term-false where term-false = False

### 89.5 Integer division and modulo for Z3

**definition** z3div :: int ⇒ int ⇒ int where
z3div k l = (if 0 ≤ l then k div l else -(k div (-l)))

**definition** z3mod :: int ⇒ int ⇒ int where
z3mod k l = (if 0 ≤ l then k mod l else k mod (-l))

### 89.6 Setup

**ML-file** Tools/SMT/smt-builtin.ML
**ML-file** Tools/SMT/smt-datatypes.ML
**ML-file** Tools/SMT/smt-normalize.ML
**ML-file** Tools/SMT/smt-translate.ML
**ML-file** Tools/SMT/smt-solver.ML
**ML-file** Tools/SMT/smtlib-interface.ML
**ML-file** Tools/SMT/z3-interface.ML
**ML-file** Tools/SMT/z3-proof-parser.ML
**ML-file** Tools/SMT/z3-proof-tools.ML
**ML-file** Tools/SMT/z3-proof-literals.ML
**ML-file** Tools/SMT/z3-proof-methods.ML
**ML-file** Tools/SMT/z3-proof-reconstruction.ML
**ML-file** Tools/SMT/z3-model.ML
**ML-file** Tools/SMT/smt-setup-solvers.ML

```ml
setup ⟨⟨
  SMT-Config.setup #>
  SMT-Normalize.setup #>
  SMTLIB-Interface.setup #>
  Z3-Interface.setup #>
  Z3-Proof-Reconstruction.setup #>
  SMT-Setup-Solvers.setup
⟩⟩
```

**method-setup** smt = ⟨⟨
  Scan.optional Attrib.thms [] >>
  (fn thms => fn ctxt =>
    METHOD (fn facts => HEADGOAL (SMT-Solver.smt-tac ctxt (thms @ facts)))))
⟩⟩ apply an SMT solver to the current goal
89.7 Configuration

The current configuration can be printed by the command `smt-status`, which shows the values of most options.

89.8 General configuration options

The option `smt-solver` can be used to change the target SMT solver. The possible values can be obtained from the `smt-status` command. Due to licensing restrictions, Yices and Z3 are not installed/enabled by default. Z3 is free for non-commercial applications and can be enabled by setting Isabelle system option `z3-non-commercial` to `yes`.

```
declare [[ smt-solver = z3 ]]
```

Since SMT solvers are potentially non-terminating, there is a timeout (given in seconds) to restrict their runtime. A value greater than 120 (seconds) is in most cases not advisable.

```
declare [[ smt-timeout = 20 ]]  
```

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

```
declare [[ smt-random-seed = 1 ]]  
```

In general, the binding to SMT solvers runs as an oracle, i.e, the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently only implemented for Z3.

```
declare [[ smt-oracle = false ]]  
```

Each SMT solver provides several commandline options to tweak its behaviour. They can be passed to the solver by setting the following options.

```
declare [[ cvc3-options = ]]  
declare [[ yices-options = ]]  
declare [[ z3-options = ]]  
```

Enable the following option to use built-in support for datatypes and records. Currently, this is only implemented for Z3 running in oracle mode.

```
declare [[ smt-datatypes = false ]]  
```

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

```
declare [[ smt-infer-triggers = false ]]  
```
The SMT method monomorphizes the given facts, that is, it tries to instantiate all schematic type variables with fixed types occurring in the problem. This is a (possibly nonterminating) fixed-point construction whose cycles are limited by the following option.

```
declare [[ monomorph-max-rounds = 5 ]]```

In addition, the number of generated monomorphic instances is limited by the following option.

```
declare [[ monomorph-max-new-instances = 500 ]]```

### 89.9 Certificates

By setting the option `smt-certificates` to the name of a file, all following applications of an SMT solver a cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending `.certs` instead of `.thy`) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

```
declare [[ smt-certificates = ]]```

The option `smt-read-only-certificates` controls whether only stored certificates are should be used or invocation of an SMT solver is allowed. When set to `true`, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to `false` and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

```
declare [[ smt-read-only-certificates = false ]]```

### 89.10 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to `false`.

```
declare [[ smt-verbose = true ]]```

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option `smt-trace` should be set to `true`.

```
declare [[ smt-trace = false ]]```

From the set of assumptions given to the SMT solver, those assumptions used in the proof are traced when the following option is set to `true`. This
only works for Z3 when it runs in non-oracle mode (see options \texttt{smt-solver}
and \texttt{smt-oracle} above).

\texttt{declare \[ \texttt{smt-trace-used-facts} = \texttt{false} \]}

\section*{89.11 Schematic rules for Z3 proof reconstruction}

Several prof rules of Z3 are not very well documented. There are two lemma
groups which can turn failing Z3 proof reconstruction attempts into suc-
ceeding ones: the facts in \texttt{z3-rule} are tried prior to any implemented recon-
struction procedure for all uncertain Z3 proof rules; the facts in \texttt{z3-simp} are
only fed to invocations of the simplifier when reconstructing theory-specific
proof steps.

\begin{verbatim}
lemmas [z3-rule] =
    refl eq-commute conj-commute disj-commute simp-thms nnf-simps
    ring-distribs field-simps times-divide-eq-right times-divide-eq-left
    if-True if-False not-not

lemma [z3-rule]:
(P \land Q) = (\neg(\neg P \lor \neg Q))
(P \land Q) = (\neg(\neg Q \lor \neg P))
(\neg P \land Q) = (\neg(P \lor \neg Q))
(\neg P \land Q) = (\neg(Q \lor P))
(P \land \neg Q) = (\neg(\neg P \lor Q))
(\neg P \land \neg Q) = (\neg(Q \lor P))
by auto

lemma [z3-rule]:
(P \rightarrow Q) = (Q \lor \neg P)
(\neg P \rightarrow Q) = (P \lor Q)
(\neg P \rightarrow Q) = (Q \lor P)
(True \rightarrow P) = P
(P \rightarrow True) = True
(False \rightarrow P) = True
(P \rightarrow P) = True
by auto

lemma [z3-rule]:
((P = Q) \rightarrow R) = (R \ | \ (Q = (\neg P)))
by auto

lemma [z3-rule]:
(\neg True) = False
(\neg False) = True
(x = x) = True
(P = True) = P
(True = P) = P
\end{verbatim}
\[(P = False) = (\neg P)\]
\[(False = P) = (\neg P)\]
\[(\neg P = P) = False\]
\[(P = (\neg P)) = False\]
\[(\neg P = (\neg Q)) = (P = Q)\]
\[\neg(P = (\neg Q)) = (P = Q)\]
\[\neg((\neg P) = Q) = (P = Q)\]
\[(P \neq Q) = (Q = (\neg P))\]
\[(P = Q) = ((\neg P \lor Q) \land (P \lor \neg Q))\]
\[(P \neq Q) = ((\neg P \lor \neg Q) \land (P \lor Q))\]
by auto

lemma [z3-rule]:
\[
\begin{align*}
(\text{if } P \text{ then } P \text{ else } \neg P) &= \text{ True} \\
(\text{if } \neg P \text{ then } \neg P \text{ else } P) &= \text{ True} \\
(\text{if } P \text{ then } \text{True} \text{ else } \text{False}) &= P \\
(\text{if } P \text{ then } \text{False} \text{ else } \text{True}) &= (\neg P) \\
(\text{if } P \text{ then } \text{Q} \text{ else } \text{True}) &= ((\neg P) \lor Q) \\
(\text{if } P \text{ then } Q \text{ else } \text{True}) &= (Q \lor (\neg P)) \\
(\text{if } P \text{ then } Q \text{ else } \neg Q) &= (P = Q) \\
(\text{if } P \text{ then } \neg Q \text{ else } Q) &= (P = (\neg Q)) \\
(\text{if } P \text{ then } \neg Q \text{ else } Q) &= ((\neg Q) = P) \\
(\text{if } \neg P \text{ then } x \text{ else } y) &= (\text{if } P \text{ then } y \text{ else } x) \\
(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) &= (\text{if } P \text{ \wedge (\neg Q) then } y \text{ else } x) \\
(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } x) &= (\text{if } (\neg Q) \text{ \wedge } P \text{ then } y \text{ else } x) \\
(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) &= (\text{if } P \text{ \wedge Q then } x \text{ else } y) \\
(\text{if } P \text{ then } (\text{if } Q \text{ then } x \text{ else } y) \text{ else } y) &= (\text{if } Q \text{ \wedge P then } x \text{ else } y) \\
(\text{if } P \text{ then } x \text{ else if } P \text{ then } y \text{ else } z) &= (\text{if } P \text{ \text{then } x \text{ else } z}) \\
(\text{if } P \text{ then } x \text{ else if Q then } x \text{ else } y) &= (\text{if } P \text{ \text{then } Q \text{ \wedge x \text{ then } y \text{ else } y}}) \\
(\text{if } P \text{ then } x \text{ else if Q then } x \text{ else } y) &= (\text{if } Q \text{ \text{then } P \text{ \wedge x \text{ then } y \text{ else } y}}) \\
(\text{if } P \text{ then } x = y \text{ else } z = x) &= (x = (\text{if } P \text{ then } y \text{ else } z)) \\
(\text{if } P \text{ then } x = y \text{ else } z = y) &= (y = (\text{if } P \text{ then } x \text{ else } z)) \\
(\text{if } P \text{ then } x = y \text{ else } z = y) &= (y = (\text{if } P \text{ \text{then } x \text{ else } z})) \\
\end{align*}
\]
by auto

lemma [z3-rule]:
\[
\begin{align*}
\theta + (x::\text{int}) &= x \\
x + 0 &= x \\
x + x &= 2 * x \\
\theta * x &= 0 \\
1 * x &= x \\
x + y &= y + x \\
\end{align*}
\]
by auto

lemma [z3-rule]:
\[
\begin{align*}
P &= Q \lor P \lor Q \\
P &= Q \lor \neg P \lor \neg Q \\
(\neg P) &= Q \lor \neg P \lor Q \\
\end{align*}
\]
THEORY "Main"

(¬P) = Q ∨ P ∨ ¬Q
P = (¬Q) ∨ ¬P ∨ Q
P = (¬Q) ∨ P ∨ ¬Q
P ≠ Q ∨ P ∨ ¬Q
P ≠ Q ∨ ¬P ∨ Q
P ≠ (¬Q) ∨ P ∨ Q
(¬P) ≠ Q ∨ P ∨ Q
P ∨ Q ∨ P ≠ (¬Q)
P ∨ Q ∨ (¬P) ≠ Q
P ∨ ¬Q ∨ P ≠ Q
¬P ∨ Q ∨ ¬Q
¬P ∨ Q ∨ P ≠ Q
P ∨ y = (if P then x else y)
P ∨ (if P then x else y) = y
¬P ∨ x = (if P then x else y)
¬P ∨ (if P then x else y) = x
P ∨ R ∨ ¬(if P then Q else R)
¬P ∨ Q ∨ ¬(if P then Q else R)
¬(if P then Q else R) ∨ ¬P ∨ Q
(iP then Q else R) ∨ ¬P ∨ ¬Q
(iP then Q else R) ∨ P ∨ R
(iP then ¬Q else R) ∨ ¬P ∨ Q
(iP then Q else ¬R) ∨ P ∨ R
by auto

hide-type (open) pattern
hide-const fun-app term-true term-false z3div z3mod
hide-const (open) trigger pat nopat weight

end

90 Main: Main HOL

theory Main
imports Predicate-Compile Quickcheck-Narrowing Extraction Lifting-Sum Coinduction Nitpick BNF-GFP SMT
begin

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

See further [1]

no-notation
bot (⊥) and
top (⊤) and
inf (infixl ⊓ 70) and
sup (infixl ⊔ 65) and
Inf (⨆ [900] 900) and
Sup (⨆ [900] 900) and
THEORY "Fact"

ordLeq2 (infix <= o 50) and
ordLeq3 (infix <= o 50) and
ordLess2 (infix < o 50) and
ordIso2 (infix = o 50) and
card-of (|()-|) and
csum (infixr + c 65) and
cprod (infixr *c 80) and
cexp (infixr ^c 90) and
convol ((|(-/,-))

hide-const (open)
czero cinfinite cfinite csum cone ctwo Csum cprod cexp
image2 image2p vimage2p Gr Grp collect fsts snds setl setr
convol pick-middlep fstOp sndOp csquare relImage relInvImage
Succ Shift shift proj

no-syntax (xsymbols)
-INF1 :: pttrns ⇒ 'b ⇒ 'b ((3[|]/-)|[0,10]) 10
-INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3[|]-ε/-)|[0,0,10]) 10
-SUP1 :: pttrns ⇒ 'b ⇒ 'b ((3[|]/-)|[0,10]) 10
-SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3[|]-ε/-)|[0,0,10]) 10

end

91 Fact: Factorial Function

theory Fact
imports Main
begin

class fact =
  fixes fact :: 'a ⇒ 'a

instantiation nat :: fact
begin

fun
  fact-nat :: nat ⇒ nat
where
  fact-0-nat: fact-nat 0 = Suc 0
| fact-Suc: fact-nat (Suc x) = Suc x * fact x

instance ..
end

instantiation int :: fact
THEORY "Fact"

begin

definition
\textit{fact-int} :: \textit{int} \Rightarrow \textit{int}
where
\textit{fact-int} \; x = (\textit{if} \; x \geq 0 \; \text{then} \; \textit{int} \; (\textit{fact} \; (\textit{nat} \; x)) \; \text{else} \; 0)

instance proof qed

end

91.1 Set up Transfer

\textbf{lemma} transfer-nat-int-factorial:
\begin{itemize}
\item \textit{(x::int)} \geq 0 \implies \textit{fact} \; (\textit{nat} \; x) = \textit{nat} \; (\textit{fact} \; x)
\end{itemize}
\begin{description}
\item[unfolding] \textit{fact-int-def}
\item[by] \textit{auto}
\end{description}

\textbf{lemma} transfer-nat-int-factorial-closure:
\begin{itemize}
\item \textit{x} \geq (0::int) \implies \textit{fact} \; \textit{x} \geq 0
\end{itemize}
\begin{description}
\item[by] \textit{(auto simp add: fact-int-def)}
\end{description}

\textbf{declare} transfer-morphism-nat-int\[transfer add return:
\begin{itemize}
\item transfer-nat-int-factorial
\item transfer-nat-int-factorial-closure
\end{itemize}\]

\textbf{lemma} transfer-int-nat-factorial:
\begin{itemize}
\item \textit{fact} \; (\textit{int} \; x) = \textit{int} \; (\textit{fact} \; x)
\end{itemize}
\begin{description}
\item[unfolding] \textit{fact-int-def}
\item[by] \textit{auto}
\end{description}

\textbf{lemma} transfer-int-nat-factorial-closure:
\begin{itemize}
\item \textit{is-nat} \; x \implies \textit{fact} \; \textit{x} \geq 0
\end{itemize}
\begin{description}
\item[by] \textit{(auto simp add: fact-int-def)}
\end{description}

\textbf{declare} transfer-morphism-int-nat\[transfer add return:
\begin{itemize}
\item transfer-int-nat-factorial
\item transfer-int-nat-factorial-closure
\end{itemize}\]

91.2 Factorial

\textbf{lemma} fact-0-int [simp]: \textit{fact} \; (0::int) = 1
\begin{description}
\item[by] \textit{(simp add: fact-int-def)}
\end{description}

\textbf{lemma} fact-1-nat [simp]: \textit{fact} \; (1::nat) = 1
\begin{description}
\item[by] \textit{simp}
\end{description}

\textbf{lemma} fact-Suc-0-nat [simp]: \textit{fact} \; (Suc \; 0) = Suc \; 0
\begin{description}
\item[by] \textit{simp}
\end{description}

\textbf{lemma} fact-1-int [simp]: \textit{fact} \; (1::int) = 1
by (simp add: fact-int-def)

lemma fact-plus-one-nat: \text{fact} ((n::nat) + 1) = (n + 1) \ast \text{fact} n
  by simp

lemma fact-plus-one-int:
  assumes \( n \geq 0 \)
  shows \( \text{fact} ((n::int) + 1) = (n + 1) \ast \text{fact} n \)
  using assms unfolding fact-int-def
  by (simp add: nat-add-distrib algebra-simps int-mult)

lemma fact-reduce-nat: 
  \( n :: \text{nat} \) > 0 \implies \text{fact} n = n \ast \text{fact} (n - 1)
  apply (subgoal-tac n = Suc (n - 1))
  apply (erule ssubst)
  apply (subst fact-Suc)
  apply simp-all
  done

lemma fact-reduce-int: 
  \( n :: \text{int} \) > 0 \implies \text{fact} n = n \ast \text{fact} (n - 1)
  apply (subgoal-tac n = (n - 1) + 1)
  apply (erule ssubst)
  apply (subst fact-plus-one-int)
  apply simp-all
  done

lemma fact-nonzero-nat [simp]: \( \text{fact} (n::nat) \neq 0 \)
  apply (induct n)
  apply (auto simp add: fact-plus-one-nat)
  done

lemma fact-nonzero-int [simp]: \( n \geq 0 \implies \text{fact} (n::int) \sim 0 \)
  by (simp add: fact-int-def)

lemma fact-gt-zero-nat [simp]: \( \text{fact} (n :: \text{nat}) > 0 \)
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-gt-zero-int [simp]: \( n \geq 0 \implies \text{fact} (n :: \text{int}) > 0 \)
  by (auto simp add: fact-int-def)

lemma fact-ge-one-nat [simp]: \( \text{fact} (n :: \text{nat}) \geq 1 \)
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-ge-Suc-0-nat [simp]: \( \text{fact} (n :: \text{nat}) \geq \text{Suc} 0 \)
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-ge-one-int [simp]: \( n \geq 0 \implies \text{fact} (n :: \text{int}) \geq 1 \)
  apply (auto simp add: fact-int-def)
  apply (subgoal-tac 1 = int 1)
  apply (erule ssubst)
apply (subst zle-int)
apply auto
done

lemma dvd-fact-nat [rule-format]: \(1 \leq m \rightarrow m \leq n \rightarrow m \text{ dvd fact } (n::nat)\)
apply (induct n)
apply force
apply (auto simp only: fact-Suc)
apply (subgoal-tac m = Suc n)
apply (erule ssubst)
apply (rule dvd-triv-left)
apply auto
done

lemma dvd-fact-int [rule-format]: \(1 \leq m \rightarrow m \leq n \rightarrow m \text{ dvd fact } (n::int)\)
apply (case-tac 1 \leq n)
apply (induct n rule: int-ge-induct)
apply (auto simp add: fact-plus-one-int)
apply (subgoal-tac m = i +
apply auto
done

lemma interval-plus-one-nat: \(i::nat) \leq j + 1 \Rightarrow \{i..j+1\} = \{i..j\} \cup \{j+1\}\)
by auto

lemma interval-Suc: \(i \leq \text{Suc } j \Rightarrow \{i..\text{Suc } j\} = \{i..j\}\cup \{\text{Suc } j\}\)
by auto

lemma interval-plus-one-int: \(i::int) \leq j + 1 \Rightarrow \{i..j+1\} = \{i..j\}\cup \{j+1\}\)
by auto

lemma fact-altdef-nat: \(\text{fact } (n::nat) = \text{(PROD i:} \{1..n\}. i)\)
apply (induct n)
apply force
apply (subst fact-Suc)
apply (subst interval-Suc)
apply auto
done

lemma fact-altdef-int: \(n \geq 0 \Rightarrow \text{fact } (n::int) = \text{(PROD i:} \{1..n\}. i)\)
apply (induct n rule: int-ge-induct)
apply force
apply (subst fact-plus-one-int, assumption)
apply (subst interval-plus-one-int)
apply auto
done

lemma fact-dvd: \(n \leq m \Rightarrow \text{fact } n \text{ dvd fact } (m::nat)\)
THEORY "Fact"

by (auto simp add: fact-altdef-nat intro: setprod-dvd-setprod-subset)

lemma fact-mod: \( m \leq (n::nat) \Rightarrow \text{fact } n \mod \text{fact } m = 0 \)
  by (auto simp add: dvd-imp-mod-0 fact-dvd)

lemma fact-div-fact:
  assumes \( m \geq (n::nat) \)
  shows \( \text{fact } m \div \text{fact } n = \prod \{n + 1..m\} \)
proof
  from assms this have \( m = n + d \)
  have \( \text{fact } (n + d) \div (\text{fact } n) = \prod \{n + 1..n + d\} \)
  proof (induct d)
    case 0
    show ?case by simp
    next
    case (Suc d)
    have \( \text{fact } (n + Suc d) \div \text{fact } n = Suc (n + d) \star \text{fact } (n + d) \div \text{fact } n \)
      by simp
    also from Suc.hyps have \( \ldots = Suc (n + d) \star \prod \{n + 1..n + d\} \)
      unfolding div-mult1-eq[of - \text{fact } (n + d)] by (simp add: fact-mod)
    also have \( \ldots = \prod \{n + 1..n + Suc d\} \)
      by (simp add: atLeastAtMostSuc-conv setprod.insert)
    finally show ?case .
  qed
  from this \( \langle m = n + d \rangle \)
  show ?thesis by simp
qed

lemma fact-mono-nat: \((m::nat) \leq n \Rightarrow \text{fact } m \leq \text{fact } n\)
apply (drule le-imp-less-or-eq)
apply (auto dest: less-imp-Suc-add)
apply (induct-tac k, auto)
done

lemma fact-mono-int-aux [rule format]: \( k \geq (0::int) \Rightarrow (\text{fact } (m + k) \geq \text{fact } m) \)
apply (case-tac m \geq 0)
apply auto
apply (frule fact-gt-zero-int)
apply arith
done

lemma fact-mono-int [simp]: \( m < (0::int) \Rightarrow \text{fact } m = 0 \)
unfolding fact-int-def by auto

lemma fact-ge-zero-int [simp]: \( \text{fact } m \geq 0 \)
apply (case-tac \( m \geq 0 \))
apply auto
apply (frule fact-gt-zero-int)
apply arith
done
apply auto
apply (subst add_assoc [symmetric])
apply (subst fact-plus-one-int)
apply auto
apply (erule order-trans)
apply (subst mult_le_cancel_right1)
apply (subgoal_tac fact \(m + i\) \(\geq 0\))
apply arith
apply auto
done

lemma fact-mono-int: \((m::int) \leq n \Rightarrow fact m \leq fact n\)
apply (insert fact-mono-int-aux [of \(n - m\) \(m\)])
apply auto
done

Note that \(fact (0::'a) = fact (1::'a)\)

lemma fact-less-mono-nat: \[ (\theta::nat) < m; m < n \] \(\Rightarrow\) \(fact m < fact n\)
apply (drule_tac m = \(m\) in less-imp-Suc-add, auto)
apply (induct_tac \(k\), auto)
done

lemma fact-less-mono-int-aux: \(k \geq 0 \Rightarrow (\theta::int) < m \Rightarrow fact m < fact n\)
apply (induct k rule: int-ge-induct)
apply (simp add: fact-plus-one-int)
apply (subst (2) fact-reduce-int)
apply (auto simp add: ac-simps)
apply (erule order-less-le-trans)
apply auto
done

lemma fact-less-mono-int: \((\theta::int) < m \Rightarrow m < n \Rightarrow fact m < fact n\)
apply (insert fact-less-mono-int-aux [of \((m + 1)\) \(m\)])
apply auto
done

lemma fact-num-eq-if-nat: \(fact (m::nat) = \)
  \(\text{if } m = 0 \text{ then } 1 \text{ else } m \ast fact (m - 1)\)
by (cases \(m\)) auto

lemma fact-add-num-eq-if-nat:
  \(fact ((m::nat) + n) = \)
  \(\text{if } m + n = 0 \text{ then } 1 \text{ else } (m + n) \ast fact (m + n - 1)\)
by (cases \(m + n\)) auto

lemma fact-add-num-eq-if2-nat:
  \(fact ((m::nat) + n) = \)
  \(\text{if } m = 0 \text{ then } fact n \text{ else } (m + n) \ast fact ((m - 1) + n)\)
by (cases \(m\)) auto
lemma fact-le-power: fact n ≤ n ^ n
proof (induct n)
  case (Suc n)
  then have fact n ≤ Suc n ^ n by (rule le-trans) (simp add: power-mono)
  then show ?case by (simp add: add-le-mono)
qed simp

91.3  fact and of-nat

lemma of-nat-fact-not-zero [simp]: of-nat (fact n) ≠ (0 :: 'a::semiring-char-0)
by auto

lemma of-nat-fact-gt-zero [simp]: (0 :: 'a::linordered-semidom) < of-nat(fact n)
by auto

lemma of-nat-fact-ge-zero [simp]: (0 :: 'a::linordered-semidom) ≤ of-nat(fact n)
by simp

lemma inv-of-nat-fact-gt-zero [simp]: (0 :: 'a::linordered-field) < inverse (of-nat (fact n))
by (auto simp add: positive-imp-inverse-positive)

lemma inv-of-nat-fact-ge-zero [simp]: (0 :: 'a::linordered-field) ≤ inverse (of-nat (fact n))
by (auto intro: order-less-imp-le)

lemma fact-eq-rev-setprod-nat: fact (k :: nat) = (∏ i < k. k - i)
unfolding fact-altdef-nat
by (rule setprod.reindex-bij-witness[where i=λi. k - i and j=λi. k - i]) auto

lemma fact-div-fact-le-pow:
  assumes r ≤ n shows fact n div fact (n - r) ≤ n ^ r
proof
  have ∆r. r ≤ n ⇒ ∏ (n - r .. n) = (n - r) * ∏ {Suc (n - r) .. n}
  by (subst setprod.insert[symmetric]) (auto simp: atMostAtMostInsert)
  with assms show ?thesis
  by (induct r rule: nat.induct) (auto simp add: fact-div-fact Suc-diff-Suc mult-le-mono)
qed

lemma fact-numeral: — Evaluation for specific numerals
  fact (numeral k) = (numeral k) * (fact (pred-numeral k))
  by (simp add: numeral-eq-Suc)
end

92  Parity: Even and Odd for int and nat

theory Parity
imports Main

begin

class even-odd = semiring-div-parity
begin

definition even :: 'a ⇒ bool
where
  even-def [presburger]: even a ⟷ a mod 2 = 0

lemma even-iff-2-dvd [algebra]:
  even a ⟷ 2 dvd a
  by (simp add: even-def dvd-eq-mod-eq-0)

lemma even-zero [simp]:
  even 0
  by (simp add: even-def)

lemma even-times-anything:
  even a ⟹ even (a * b)
  by (simp add: even-iff-2-dvd)

lemma anything-times-even:
  even a ⟹ even (b * a)
  by (simp add: even-iff-2-dvd)

abbreviation odd :: 'a ⇒ bool
where
  odd a ≡ ¬ even a

lemma odd-times-odd:
  odd a ⟹ odd b ⟹ odd (a * b)
  by (auto simp add: even-def mod-mult-left-eq)

lemma even-product [simp, presburger]:
  even (a * b) ⟷ even a ∨ even b
  apply (auto simp add: even-times-anything anything-times-even)
  apply (rule contr)
  apply (auto simp add: odd-times-odd)
  done

end

instance nat and int :: even-odd ..

lemma even-nat-def [presburger]:
  even x ⟷ even (int x)
  by (auto simp add: even-def int-eq-iff int-mult nat-mult-distrib)
lemma transfer-int-nat-relations:
\[
even (\text{int } x) \leftrightarrow even x
\]
by (simp add: even-nat-def)

declare transfer-morphism-int-nat[transfer add return:

\[
\text{transfer-int-nat-relations}
\]
]

lemma odd-one-int [simp]:
odd (1::int)
by presburger

lemma odd-1-nat [simp]:
odd (1::nat)
by presburger

lemma even-numeral-int [simp]:
even (numeral (Num.Bit0 k) :: int)
unfolding even-def by simp

lemma odd-numeral-int [simp]:
odd (numeral (Num.Bit1 k) :: int)
unfolding even-def by simp

declare even-def [of − numeral v, simp] for v

lemma even-numeral-nat [simp]:
even (numeral (Num.Bit0 k) :: nat)
unfolding even-nat-def by simp

lemma odd-numeral-nat [simp]:
odd (numeral (Num.Bit1 k) :: nat)
unfolding even-nat-def by simp

92.1 Even and odd are mutually exclusive
92.2 Behavior under integer arithmetic operations

lemma even-plus-even: even (x::int) ==> even y ==> even (x + y)
by presburger

lemma even-plus-odd: even (x::int) ==> odd y ==> odd (x + y)
by presburger

lemma odd-plus-even: odd (x::int) ==> even y ==> odd (x + y)
by presburger

lemma odd-plus-odd: odd (x::int) ==> odd y ==> even (x + y) by presburger

lemma even-sum[simp,presburger]:
even ((x::int) + y) = ((even x & even y) | (odd x & odd y))
by presburger
lemma even-neg[simp,presburger,algebra]: even \(-(x::int)\) = even \(x\)
by presburger

lemma even-difference[simp]:
even \((x::int) - y\) = (even \(x\) & even \(y\)) | (odd \(x\) & odd \(y\))
by presburger

lemma even-power[simp,presburger]: even \((x::int) ^ n\) = (even \(x\) & \(n \neq 0\))
by (induct \(n\) auto)

lemma odd-pow: odd \(x\) == > odd\((x::int) ^ n\)
by simp

92.3 Equivalent definitions

lemma two-times-even-div-two: even \((x::int) \Longrightarrow 2 \ast (x \div 2) = x\)
by presburger

lemma two-times-odd-div-two-plus-one:
odd \((x::int) \Longrightarrow 2 \ast (x \div 2) + 1 = x\)
by presburger

lemma even-equiv-def: even \((x::int) = (EX y. x = 2 \ast y)\)
by presburger

lemma odd-equiv-def: odd \((x::int) = (EX y. x = 2 \ast y + 1)\)
by presburger

92.4 even and odd for nats

lemma pos-int-even-equiv-nat-even: \(0 \leq x \Longrightarrow even x = even (nat x)\)
by (simp add: even-nat-def)

lemma even-product-nat[simp,presburger,algebra]:
even\((x::nat) \ast y\) = (even \(x\) | even \(y\))
by (simp add: even-nat-def int-mult)

lemma even-sum-nat[simp,presburger,algebra]:
even \((x::nat) + y\) = ((even \(x\) & even \(y\)) | (odd \(x\) & odd \(y\)))
by presburger

lemma even-difference-nat[simp,presburger,algebra]:
even \((x::nat) - y\) = (x < y | (even \(x\) & even \(y\)) | (odd \(x\) & odd \(y\)))
by presburger

lemma even-Suc[simp,presburger,algebra]: even \((Suc x) = odd x\)
by presburger

lemma even-power-nat[simp,presburger,algebra]:
even \((x::nat) ^ y\) = (even \(x\) & \(0 < y\))
by (simp add: even-nat-def int-power)
92.5 Equivalent definitions

lemma even-nat-mod-two-eq-zero: even (x::nat) ==> x mod (Suc (Suc 0)) = 0
  by presburger

lemma odd-nat-mod-two-eq-one: odd (x::nat) ==> x mod (Suc (Suc 0)) = Suc 0
  by presburger

lemma even-nat-equiv-def: even (x::nat) = (x mod Suc (Suc 0) = 0)
  by presburger

lemma odd-nat-equiv-def: odd (x::nat) = (x mod Suc (Suc 0) = Suc 0)
  by presburger

lemma even-nat-div-two-times-two: even (x::nat) = Suc (Suc 0) * (x div Suc (Suc 0)) = x
  by presburger

lemma odd-nat-div-two-times-two-plus-one: odd (x::nat) = Suc (Suc (Suc 0) * (x div Suc (Suc 0))) = x
  by presburger

lemma even-nat-equiv-def2: even (x::nat) = (EX y. x = Suc (Suc 0) * y)
  by presburger

lemma odd-nat-equiv-def2: odd (x::nat) = (EX y. x = Suc (Suc (Suc 0) * y))
  by presburger

92.6 Parity and powers

lemma (in comm-ring-1) neg-power-if: 
  (- a) ^ n = (if even n then (a ^ n) else - (a ^ n))
  by (induct n) simp-all

lemma (in comm-ring-1) shows neg-one-even-power [simp]: even n ==> (- 1) ^ n = 1
  and neg-one-odd-power [simp]: odd n ==> (- 1) ^ n = - 1
  by (simp-all add: neg-power-if)

lemma zero-le-even-power: even n ==> 
  0 <= (x::a::{linordered-ring,monoid-mult}) ^ n
  apply (simp add: even-nat-equiv-def2)
  apply (erule exE)
  apply (erule ssubs
  apply (subst power-add)
  apply (rule zero-le-square)
  done

lemma zero-le-odd-power: odd n ==> 
  (0 <= (x::a::{linordered-idom}) ^ n) = (0 <= x)
  apply (auto simp: odd-nat-equiv-def2 power-add zero-le-mult-iff)
  apply (metis field-power-not-zero divisors-zero order-antisym-conv zero-le-square)
done

lemma zero-le-power-eq [presburger]: \((0 \leq (x::'a::{linordered-idom}) ^ n) = \) 
\[(\text{even } n \mid \text{odd } n \& 0 \leq x)\]
apply auto
apply (subst zero-le-odd-power [symmetric])
apply assumption+
apply (erule zero-le-even-power)
done

lemma zero-less-power-eq[presburger]: \((0 < (x::'a::{linordered-idom}) ^ n) = \) 
\[(n = 0 \mid \text{even } n \& x = 0) \mid \text{odd } n \& 0 < x)\]

unfolding order-less-le zero-le-power-eq by auto

lemma power-less-zero-eq[presburger]: \((x::'a::{linordered-idom}) ^ n < 0) = \)
\[(\text{odd } n \& x < 0)\]
apply (subst linorder-not-le [symmetric])+
apply (subst zero-le-power-eq)
apply auto
done

lemma power-le-zero-eq[presburger]: \((x::'a::{linordered-idom}) ^ n \leq 0) = \)
\[(n = 0 \mid \text{odd } n \& x <= 0) \mid \text{even } n \& x = 0)\]
apply (subst linorder-not-less [symmetric])+
apply (subst zero-less-power-eq)
apply auto
done

lemma power-even-abs: even n ==>
\((\text{abs } (x::'a::{linordered-idom})) ^ n = x ^ n\)
apply (subst power-abs [symmetric])
apply (simp add: zero-le-even-power)
done

lemma power-minus-even [simp]: even n ==>
\((- x) ^ n = (x ^ n::'a::{comm-ring-1})\)
apply (subst power-minus)
apply simp
done

lemma power-minus-odd [simp]: odd n ==>
\((- x) ^ n = - (x ^ n::'a::{comm-ring-1})\)
apply (subst power-minus)
apply simp
done

lemma power-mono-even: fixes x y :: 'a :: {linordered-idom}
assumes even n and |x| \leq |y|
shows $x^n \leq y^n$

proof
have $0 \leq |x|$ by auto
with $|x| \leq |y|$
have $|x|^n \leq |y|^n$ by (rule power-mono)
thus $?thesis$ unfolding power-even-abs[OF ⟨even n⟩].

qed

lemma odd-pos: odd $n$ $\Rightarrow$ $0 < n$ by presburger

lemma power-mono-odd: fixes $x$ $y$ :: int
assumes odd $n$ and $x \leq y$
sows $x^n \leq y^n$
proof (cases $y < 0$)
case True with $x \leq y$ have $-y \leq -x$ and $0 \leq -y$ by auto
hence $(-y)^n \leq (-x)^n$ by (rule power-mono)
thus $?thesis$ unfolding power-minus-odd[OF ⟨odd n⟩] by auto
next
case False
show $?thesis$
proof (cases $x < 0$)
case True hence $n \neq 0$ and $x \leq 0$ using (odd $n$)[THEN odd-pos] by auto
hence $x^n \leq 0$ unfolding power-le-zero-eq using (odd $n$) by auto
moreover
from ($\neg y < 0$) have $0 \leq y$ by auto
hence $0 \leq y^n$ by auto
ultimately show $?thesis$ by auto
next
case False hence $0 \leq x$ by auto
with $x \leq y$ show $?thesis$ using power-mono by auto
qed

qed

92.7 More Even/Odd Results

lemma even-mult-two-ex: even $n$ $\Rightarrow$ $\exists m :: \text{nat}. n = 2 \ast m$ by presburger
lemma odd-Suc-mult-two-ex: odd $n$ $\Rightarrow$ $\exists m :: \text{nat}. n = \text{Suc} \ (2 \ast m)$ by presburger
lemma even-add [simp]: even $m + n$ $\Rightarrow$ even $m$ = even $n$ by presburger
lemma odd-add [simp]: odd $m + n$ $\Rightarrow$ odd $m$ $\neq$ odd $n$ by presburger
lemma lemma-even-div2 [simp]: even $n$ $\Rightarrow$ $(n + 1) \div 2 = n \div 2$ by presburger
lemma lemma-not-even-div2 [simp]: $\neg$even $n$ $\Rightarrow$ $(n + 1) \div 2 = \text{Suc} \ (n \div 2)$ by presburger
lemma even-num-iff: $0 < n$ $\Rightarrow$ even $n = (\neg$ even $n - 1 :: \text{nat})$ by presburger
lemma even-even-mod-4-iff: even (n::nat) = even (n mod 4) by presburger

lemma lemma-odd-mod-4-div-2: n mod 4 = (3::nat) ==> odd((n - 1) div 2) by presburger

lemma lemma-even-mod-4-div-2: n mod 4 = (1::nat) ==> even ((n - 1) div 2) by presburger

Simplify, when the exponent is a numeral

lemmas zero-le-power-eq-numeral [simp] =
    zero-le-power-eq [of - numeral w] for w

lemmas zero-less-power-eq-numeral [simp] =
    zero-less-power-eq [of - numeral w] for w

lemmas power-le-zero-eq-numeral [simp] =
    power-le-zero-eq [of - numeral w] for w

lemmas power-less-zero-eq-numeral [simp] =
    power-less-zero-eq [of - numeral w] for w

lemmas zero-less-power-nat-eq-numeral [simp] =
    nat-zero-less-power-iff [of - numeral w] for w

lemmas power-eq-0-iff-numeral [simp] =
    power-eq-0-iff [of - numeral w] for w

lemmas power-even-abs-numeral [simp] =
    power-even-abs [of numeral w - ] for w

92.8 An Equivalence for 0 ≤ a ^ n

lemma zero-le-power-eq[presburger]:
    (0 ≤ a ^ n) = (0 ≤ (a::a::{linordered-idom}) | even n)

proof cases
    assume even: even n
    then obtain k where n = 2*k
        by (auto simp add: even-nat-equiv-def2 numeral-2-eq-2)
    thus ?thesis by (simp add: zero-le-even-power even)

next
    assume odd: odd n
    then obtain k where n = Suc(2*k)
        by (auto simp add: odd-nat-equiv-def2 numeral-2-eq-2)
    moreover have a ^ (2 * k) ≤ 0 ==> a = 0
        by (induct k) (auto simp add: zero-le-mult-iff mult-le-0-iff)
    ultimately show ?thesis
        by (auto simp add: zero-le-mult-iff zero-le-even-power)
qed
92.9 Miscellaneous

lemma \([\text{presburger}]: (x + 1) \div 2 = x \div 2 \iff \text{even } (x::\text{int}) \text{ by presburger}

lemma \([\text{presburger}]: (x + 1) \div 2 = x \div 2 + 1 \iff \text{odd } (x::\text{int}) \text{ by presburger}

lemma even-plus-one-div-two: \text{even } (x::\text{int}) \Rightarrow (x + 1) \div 2 = x \div 2 \text{ by presburger}

lemma odd-plus-one-div-two: \text{odd } (x::\text{int}) \Rightarrow (x + 1) \div 2 = x \div 2 + 1 \text{ by presburger}

lemma \([\text{presburger}]: (\text{Suc } x) \div \text{Suc } (\text{Suc } 0) = x \div \text{Suc } (\text{Suc } 0) \iff \text{even } x \text{ by presburger}

lemma even-nat-plus-one-div-two: \text{even } (x::\text{nat}) \Rightarrow (\text{Suc } x) \div \text{Suc } (\text{Suc } 0) = x \div \text{Suc } (\text{Suc } 0) \text{ by presburger}

lemma odd-nat-plus-one-div-two: \text{odd } (x::\text{nat}) \Rightarrow (\text{Suc } x) \div \text{Suc } (\text{Suc } 0) = \text{Suc } (x \div \text{Suc } (\text{Suc } 0)) \text{ by presburger}

end

93 GCD: Greatest common divisor and least common multiple

theory GCD
imports Fact Parity
begin

declare One-nat-def [simp del]

93.1 GCD and LCM definitions

class gcd = zero + one + dvd +
  fixes gcd :: 'a ⇒ 'a ⇒ 'a
  and lcm :: 'a ⇒ 'a ⇒ 'a
begin

abbreviation
  coprime :: 'a ⇒ 'a ⇒ bool
where
  coprime x y == (gcd x y = 1)
end

instantiation nat :: gcd
begin

fun
gcd-nat :: nat ⇒ nat ⇒ nat
where
gcd-nat x y =
fun gcd : int ⇒ int ⇒ int where
  gcd x y = int (gcd (nat (abs x)) (nat (abs y)))

fun lcm : int ⇒ int ⇒ int where
  lcm x y = int (lcm (nat (abs x)) (nat (abs y)))

lemma transfer-nat-int-gcd:
  (x:int) >= 0 ⇒ y >= 0 ⇒ gcd (nat x) (nat y) = nat (gcd x y)
  (x:int) >= 0 ⇒ y >= 0 ⇒ lcm (nat x) (nat y) = nat (lcm x y)
unfolding gcd-int-def lcm-int-def
by auto

lemma transfer-nat-int-gcd-closures:
x >= (0::int) ⇒ y >= 0 ⇒ gcd x y >= 0
x >= (0::int) ⇒ y >= 0 ⇒ lcm x y >= 0
by (auto simp add: gcd-int-def lcm-int-def)

declare transfer-morphism-nat-int[transfer add return:
  transfer-nat-int-gcd transfer-nat-int-gcd-closures]

lemma transfer-int-nat-gcd:
gcd (int x) (int y) = int (gcd x y)
lcm (int x) (int y) = int (lcm x y)
by (unfold gcd-int-def lcm-int-def, auto)
lemma transfer-int-nat-gcd-closures:
  is-nat x ⇒ is-nat y ⇒ gcd x y ≥ 0
  is-nat x ⇒ is-nat y ⇒ lcm x y ≥ 0
by (auto simp add: gcd-int-def lcm-int-def)

declare transfer-morphism-int-nat[transfer add return:
  transfer-int-nat-gcd transfer-int-nat-gcd-closures]

93.3 GCD properties

lemma gcd-nat-induct:
  fixes m n :: nat
  assumes ∀m. P m 0
  and ∀m n. 0 < n ⇒ P n (m mod n) ⇒ P m n.
  shows P m n
apply (rule gcd-nat.induct)
apply (case-tac y = 0)
using assms apply simp-all
done

lemma gcd-neg1-int [simp]: gcd (-x::int) y = gcd x y
by (simp add: gcd-int-def)

lemma gcd-neg2-int [simp]: gcd (x::int) (-y) = gcd x y
by (simp add: gcd-int-def)

lemma gcd-neg-numeral-1-int [simp]:
  gcd (- numeral n :: int) x = gcd (numeral n) x
by (fact gcd-neg1-int)

lemma gcd-neg-numeral-2-int [simp]:
  gcd x (- numeral n :: int) = gcd x (numeral n)
by (fact gcd-neg1-int)

lemma abs-gcd-int[simp]: abs(gcd (x::int) y) = gcd x y
by(simp add: gcd-int-def)

lemma gcd-abs-int: gcd (x::int) y = gcd (abs x) (abs y)
by (simp add: gcd-int-def)

lemma gcd-abs1-int[simp]: gcd (abs x) (y::int) = gcd x y
by (metis abs-idempotent gcd-abs-int)

lemma gcd-abs2-int[simp]: gcd (abs y::int) = gcd x y
by (metis abs-idempotent gcd-abs-int)

lemma gcd-cases-int:
fixes x :: int and y
assumes x >= 0 ==> y >= 0 ==> P (gcd x y)
    and x >= 0 ==> y <= 0 ==> P (gcd x (-y))
    and x <= 0 ==> y >= 0 ==> P (gcd (-x) y)
    and x <= 0 ==> y <= 0 ==> P (gcd (-x) (-y))
shows P (gcd x y)
by (insert assms, auto, arith)

lemma gcd-ge-0-int [simp]: gcd (x::int) y >= 0
by (simp add: gcd-int-def)

lemma lcm-neg1-int: lcm (-x::int) y = lcm x y
by (simp add: lcm-int-def)

lemma lcm-neg2-int: lcm (x::int) (-y) = lcm x y
by (simp add: lcm-int-def)

lemma lcm-abs-int: lcm (x::int) y = lcm (abs x) (abs y)
by (simp add: lcm-int-def)

lemma abs-lcm-int [simp]: abs (lcm i j::int) = lcm i j
by (simp add: lcm-int-def)

lemma lcm-abs1-int [simp]: lcm (abs x::int) y = lcm x y
by (metis abs-idempotent lcm-int-def)

lemma lcm-abs2-int [simp]: lcm x (abs y::int) = lcm x y
by (metis abs-idempotent lcm-int-def)

lemma lcm-cases-int:
fixes x :: int and y
assumes x >= 0 ==> y >= 0 ==> P (lcm x y)
    and x >= 0 ==> y <= 0 ==> P (lcm x (-y))
    and x <= 0 ==> y >= 0 ==> P (lcm (-x) y)
    and x <= 0 ==> y <= 0 ==> P (lcm (-x) (-y))
shows P (lcm x y)
using assms by (auto simp add: lcm-neg1-int lcm-neg2-int) arith

lemma lcm-ge-0-int [simp]: lcm (x::int) y >= 0
by (simp add: lcm-int-def)

lemma gcd-0-nat: gcd (x::nat) 0 = x
by simp

lemma gcd-0-int [simp]: gcd (x::int) 0 = abs x
by (unfold gcd-int-def, auto)
lemma gcd-0-left-nat: gcd 0 (x::nat) = x
  by simp

lemma gcd-0-left-int [simp]: gcd 0 (x::int) = abs x
  by (unfold gcd-int-def, auto)

lemma gcd-red-nat: gcd (x::nat) y = gcd y (x mod y)
  by (case_tac y = 0, auto)

lemma gcd-non-0-nat: y ~= (0::nat) ==> gcd (x::nat) y = gcd y (x mod y)
  by simp

lemma gcd-1-nat [simp]: gcd (m::nat) 1 = 1
  by simp

lemma gcd-1-int [simp]: gcd (m::int) 1 = 1
  by (simp add: One-int-def)

lemma gcd-idem-nat: gcd (x::nat) x = x
  by simp

lemma gcd-idem-int: gcd (x::int) x = abs x
  by (auto simp add: gcd-int-def)

declare gcd-nat.simps [simp del]

gcd m n divides m and n. The conjunctions don’t seem provable separately.

lemma gcd-dvd1-nat [iff]: (gcd (m::nat)) n dvd m
  and gcd-dvd2-nat [iff]: (gcd m n) dvd n
  by (induct m n rule: gcd-nat-induct)
  apply (simp-all add: gcd-non-0-nat gcd-0-nat)
  apply (blast dest: dvd-mod-imp-dvd)
done

lemma gcd-dvd1-int [iff]: gcd (x::int) y dvd x
  by (metis gcd-int-def int-dvd-iff gcd-dvd1-nat)

lemma gcd-dvd2-int [iff]: gcd (x::int) y dvd y
  by (metis gcd-int-def int-dvd-iff gcd-dvd2-nat)

lemma dvd-gcd-D1-nat: k dvd gcd m n ==> (k::nat) dvd m
  by (metis gcd-dvd1-nat dvd-trans)

lemma dvd-gcd-D2-nat: k dvd gcd m n ==> (k::nat) dvd n
by (metis gcd-dvd2-nat dvd-trans)

lemma dvd-gcd-D1-int: i dvd gcd m n \implies (i::int) dvd m
by (metis gcd-dvd1-int dvd-trans)

lemma dvd-gcd-D2-int: i dvd gcd m n \implies (i::int) dvd n
by (metis gcd-dvd2-int dvd-trans)

lemma gcd-le1-nat [simp]: a \neq 0 \implies gcd (a::nat) b \leq a
by (rule dvd-imp-le, auto)

lemma gcd-le2-nat [simp]: b \neq 0 \implies gcd (a::nat) b \leq b
by (rule dvd-imp-le, auto)

lemma gcd-le1-int [simp]: a > 0 \implies gcd (a::int) b \leq a
by (rule zdvd-imp-le, auto)

lemma gcd-le2-int [simp]: b > 0 \implies gcd (a::int) b \leq b
by (rule zdvd-imp-le, auto)

lemma gcd-greatest-nat: (k::nat) dvd m \implies k dvd gcd m n
by (induct m n rule: gcd-nat-induct) (simp-all add: gcd-non-0-nat dvd-mod gcd-0-nat)

lemma gcd-greatest-int: (k::int) dvd m \implies k dvd gcd m n
apply (subst gcd-abs-int)
apply (subst abs-dvd-iff [symmetric])
apply (rule gcd-greatest-nat [transferred])
apply auto
done

lemma gcd-greatest-iff-nat [iff]: (k dvd gcd (m::nat) n) = (k dvd m \& k dvd n)
by (blast intro!: gcd-greatest-int intro: dvd-trans)

lemma gcd-greatest-iff-int: ((k::int) dvd gcd m n) = (k dvd m \& k dvd n)
by (blast intro!: gcd-greatest-int intro: dvd-trans)

lemma gcd-zero-nat [simp]: (gcd (m::nat) n = 0) = (m = 0 \& n = 0)
by (simp only: dvd-0-left-iff [symmetric] gcd-greatest-iff-nat)

lemma gcd-zero-int [simp]: (gcd (m::int) n = 0) = (m = 0 \& n = 0)
by (auto simp add: gcd-int-def)

lemma gcd-pos-nat [simp]: (gcd (m::nat) n > 0) = (m > 0 \& n > 0)
by (insert gcd-zero-nat [of m n], arith)

lemma gcd-pos-int [simp]: (gcd (m::int) n > 0) = (m > 0 \& n > 0)
by (insert gcd-zero-int [of m n], insert gcd-ge-0-int [of m n], arith)
lemma gcd-unique-nat: (d::nat) dvd a ∧ d dvd b ∧
(∀ e. e dvd a ∧ e dvd b −→ e dvd d) ←→ d = gcd a b
apply auto
apply (rule dvd-antisym)
apply (erule (1) gcd-greatest-nat)
apply auto
done

lemma gcd-unique-int: d ≥ 0 & (d::int) dvd a ∧ d dvd b ∧
(∀ e. e dvd a ∧ e dvd b −→ e dvd d) ←→ d = gcd a b
apply (case-tac d = 0)
apply simp
apply (rule iffI)
apply (rule zdvd-antisym-nonneg)
apply (auto intro: gcd-greatest-int)
done

interpretation gcd-nat: abel-semigroup gcd :: nat ⇒ nat ⇒ nat
+ gcd-nat: semilattice-neutr-order gcd :: nat ⇒ nat ⇒ nat 0 op dvd (λm n. m dvd n ∧ ¬ n dvd m)
apply default
apply (auto intro: dvd-antisym dvd-trans)[4]
apply (metis dvd.dual-order.refl gcd-unique-nat)
apply (auto intro: dvdI elim: dvdE)
done

interpretation gcd-int: abel-semigroup gcd :: int ⇒ int ⇒ int
proof
qed (simp-all add: gcd-int-def gcd-nat.assoc gcd-nat.commute gcd-nat.left-commute)

lemmas gcd-assoc-nat = gcd-nat.assoc
lemmas gcd-commute-nat = gcd-nat.commute
lemmas gcd-left-commute-nat = gcd-nat.left-commute
lemmas gcd-assoc-int = gcd-int.assoc
lemmas gcd-commute-int = gcd-int.commute
lemmas gcd-left-commute-int = gcd-int.left-commute

lemmas gcd-ac-nat = gcd-assoc-nat gcd-commute-nat gcd-left-commute-nat
lemmas gcd-ac-int = gcd-assoc-int gcd-commute-int gcd-left-commute-int

lemma gcd-proj1-if-dvd-nat [simp]: (x::nat) dvd y −→ gcd x y = x
by (fact gcd-nat.absorb1)

lemma gcd-proj2-if-dvd-nat [simp]: (y::nat) dvd x −→ gcd x y = y
by (fact gcd-nat.absorb2)

lemma gcd-proj1-if-dvd-int [simp]: x dvd y −→ gcd (x::int) y = abs x
THEORY “GCD”

by (metis abs-dvd-iff gcd-0-left-int gcd-abs-int gcd-unique-int)

lemma gcd-proj2-if-dvd-int [simp]: y dvd x ⇒ gcd (x::int) y = abs y
  by (metis gcd-proj1-if-dvd-int gcd-commute-int)

Multiplication laws

lemma gcd-mult-distrib-nat: (k::nat) * gcd m n = gcd (k * m) (k * n)
  — [?, page 27]
  apply (induct m n rule: gcd-nat-induct)
  apply simp
  apply (case-tac k = 0)
  apply (simp-all add: gcd-non-0-nat)
  done

lemma gcd-mult-distrib-int: abs (k::int) * gcd m n = gcd (k * m) (k * n)
  apply (subst (1 2) gcd-abs-int)
  apply (subst (1 2) abs-mult)
  apply (rule gcd-mult-distrib-nat [transferred])
  apply auto
  done

lemma coprime-dvd-mult-nat: coprime (k::nat) n ⇒ k dvd m * n ⇒ k dvd m
  apply (insert gcd-mult-distrib-nat [of m k n])
  apply simp
  apply (erule-tac t = m in subst)
  apply simp
  done

lemma coprime-dvd-mult-int: coprime (k::int) n ⇒ k dvd m * n ⇒ k dvd m
  apply (subst abs-dvd-iff [symmetric])
  apply (subst dvd-abs-iff [symmetric])
  apply (subst (asm) gcd-abs-int)
  apply (rule coprime-dvd-mult-nat [transferred])
    prefer 4 apply assumption
    apply auto
  apply (subst abs-mult [symmetric], auto)
  done

lemma coprime-dvd-mult-iff-nat: coprime (k::nat) n ⇒ (k dvd m * n) = (k dvd m)
  by (auto intro: coprime-dvd-mult-nat)

lemma coprime-dvd-mult-iff-int: coprime (k::int) n ⇒ (k dvd m * n) = (k dvd m)
  by (auto intro: coprime-dvd-mult-int)

lemma gcd-mult-cancel-nat: coprime k n ⇒ gcd ((k::nat) * m) n = gcd m n
  apply (rule dvd-antisym)
apply (rule gcd-greatest-nat)
apply (rule_tac n = k in coprime-dvd-mult-nat)
apply (simp add: gcd-assoc-nat)
apply (simp add: gcd-commute-nat)
apply (simp-all add: mult.commute)
done

lemma gcd-mult-cancel-int:
coprime (k :: int) n ⇒ gcd (k * m) n = gcd m n
apply (subst (1 2) gcd-abs-int)
apply (subst abs-mult)
apply (rule gcd-mult-cancel-nat [transferred], auto)
done

lemma coprime-crossproduct-nat:
  fixes a b c d :: nat
  assumes coprime a d and coprime b c
  shows a * c = b * d ↔ a = b ∧ c = d (is ?lhs ↔ ?rhs)
proof
  assume ?rhs then show ?lhs by simp
next
  assume ?lhs
  from ⟨?lhs⟩ have a dvd b * d by (auto intro: dvdI dest: sym)
  with ⟨coprime a d⟩ have a dvd b by (simp add: coprime-dvd-mult-iff-nat)
  from ⟨?lhs⟩ have b dvd d * a by (auto intro: dvdI dest: sym)
  with ⟨coprime b c⟩ have b dvd a by (simp add: coprime-dvd-mult-iff-nat)
  from ⟨?lhs⟩ have c dvd d by (simp add: gcd-commute-nat)
  with ⟨coprime b c⟩ have c dvd d by (simp add: gcd-commute-nat)
  from ⟨?lhs⟩ have d dvd c * a by (auto intro: dvdI dest: sym simp add: mult.commute)
  with ⟨coprime a d⟩ have d dvd c by (simp add: gcd-commute-nat)
  from ⟨a dvd b ⟩ ⟨b dvd a ⟩ have a = b by (rule Nat.dvd.antisym)
  moreover from ⟨c dvd d ⟩ ⟨d dvd c ⟩ have c = d by (rule Nat.dvd.antisym)
  ultimately show ?rhs ..
qed

lemma coprime-crossproduct-int:
  fixes a b c d :: int
  assumes coprime a d and coprime b c
  shows |a * c| = |b * d| ↔ |a| = |b| ∧ |c| = |d|
  using assms by (intro coprime-crossproduct-nat [transferred]) auto

Addition laws

lemma gcd-add1-nat [simp]: gcd ((m::nat) + n) n = gcd m n
  apply (case-tac n = 0)
  apply (simp-all add: gcd-non-0-nat)
done

lemma gcd-add2-nat [simp]: gcd (m::nat) (m + n) = gcd m n
  apply (subst (1 2) gcd-commute-nat)
apply (subst add.commute)
apply simp
done

lemma gcd-diff1-nat: (m::nat) >= n ==> gcd (m - n) n = gcd m n
  by (subst gcd-add1-nat [symmetric], auto)

lemma gcd-diff2-nat: (n::nat) >= m ==> gcd (n - m) n = gcd m n
  apply (subst gcd-commute-nat)
  apply (subst gcd-diff1-nat [symmetric])
  apply auto
  apply (subst gcd-commute-nat)
  apply (subst gcd-diff1-nat)
  apply assumption
  apply (rule gcd-commute-nat)
done

lemma gcd-non-0-int: (y::int) > 0 ==> gcd x y = gcd y (x mod y)
  apply (frule-tac b = y and a = x in pos-mod-sign)
  apply (simp del: pos-mod-sign add: gcd-int-def abs-if nat-mod-distrib)
  apply (auto simp add: gcd-non-0-nat nat-mod-distrib [symmetric]
                     zmod-zminus1-eq-if)
  apply (frule-tac a = x in pos-mod-bound)
  apply (subst (1 2) gcd-commute-nat)
  apply (simp del: pos-mod-bound add: nat-diff-distrib gcd-diff2-nat
          nat-le-eq-zle)
done

lemma gcd-red-int: gcd (x::int) y = gcd y (x mod y)
  apply (case-tac y = 0)
  apply force
  apply (case-tac y > 0)
  apply (subst gcd-non-0-int, auto)
  apply (insert gcd-non-0-int [of -y -x])
  apply auto
done

lemma gcd-add1-int [simp]: gcd ((m::int) + n) n = gcd m n
  by (metis gcd-red-int mod-add-self1 add.commute)

lemma gcd-add2-int [simp]: gcd m ((m::int) + n) = gcd m n
  by (metis gcd-add1-int gcd-commute-int add.commute)

lemma gcd-add-mult-nat: gcd (m::nat) (k * m + n) = gcd m n
  by (metis mod-mult-self3 gcd-commute-nat gcd-red-nat)

lemma gcd-add-mult-int: gcd (m::int) (k * m + n) = gcd m n
by (metis gcd-commute-int gcd-red-int mod-mult1 add.commute)

lemma gcd-dvd-prod-nat [iff]: \( \text{gcd} (m::nat) \mid n \implies k \cdot n \)
using mult-dvd-mono [of 1] by auto

lemma finite-divisors-nat [simp]:
assumes (m::nat) \sim \emptyset
shows finite {d. d dvd m}
proof -
  have finite {d. d \leq m} by (blast intro: bounded-nat-set-is-finite)
  from finite-subset [OF - this] show \?thesis using assms
    by (bestsimp intro!: dvd-imp-le)
qed

lemma finite-divisors-int [simp]:
assumes (i::int) \sim \emptyset
shows finite {d. d dvd i}
proof -
  have {d. abs d \leq abs i} = {- abs i .. abs i} by (auto simp: abs-if)
  hence finite {d. abs d \leq abs i} by simp
  from finite-subset [OF - this] show \?thesis using assms
    by (bestsimp intro!: dvd-imp-le-int)
qed

lemma Max-divisors-self-nat [simp]:
n \neq 0 \implies \text{Max} {d::nat. d dvd n} = n
apply (rule antisym)
apply (fastforce intro: Max-le-iff [THEN iffD2] simp: dvd-imp-le)
apply simp
done

lemma Max-divisors-self-int [simp]:
n \neq 0 \implies \text{Max} {d::int. d dvd n} = \text{abs} n
apply (rule antisym)
apply (rule Max-le-iff [THEN iffD2])
apply (auto intro: abs-le-D1 dvd-imp-le-int)
done

lemma gcd-is-Max-divisors-nat:
m \sim \emptyset \implies n \sim \emptyset \implies \text{gcd} (m::nat) \mid n = (\text{Max} \{d. d dvd m \& d dvd n\})
apply (rule Max-eqI [THEN sym])
apply (metis finite-Collect-conj finite-divisors-nat)
apply simp
apply (metis Suc-diff-1 Suc-neq-Zero dvd-imp-le gcd-greatest-iff-nat gcd-pos-nat)
apply simp
done
lemma gcd-is-Max-divisors-int:
  \( m \sim 0 \implies n \sim 0 \implies \gcd(m::\text{int}) \ n = (\operatorname{Max}\ \{d.\ d \ \text{dvd} \ m \ \&\ \text{dvd} \ n\}) \)
apply(rule Max-cqI[THEN sym])
  apply (metis finite-Collect-conjI finite-divisors-int)
  apply simp
  apply (metis gcd-pos-int zdvd-imp-le)
apply simp
done

lemma gcd-code-int [code]:
  \( \gcd k l = \text{if } l = (0::\text{int}) \text{ then } k \text{ else } \gcd l (|k| \ \text{mod} \ |l|) \)
by (simp add: gcd-int-def nat-mod-distrib gcd-non-0-nat)

93.4 Coprimality

lemma div-gcd-coprime-nat:
  assumes nz: \((a::\text{nat}) \neq 0 \ \vee\ b \neq 0\)
  shows coprime \((a \ \text{div} \ gcd \ a \ b) \ (b \ \text{div} \ gcd \ a \ b)\)
proof –
  let ?g = gcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = gcd ?a' ?b'
  have dvdg: \(?g \ \text{dvd} \ a \ \&\ ?g \ \text{dvd} \ b\) by simp-all
  have dvdgp: \(?g' \ \text{dvd} \ ?a' \ ?g' \ \text{dvd} \ ?b'\) by simp-all
  from dvdg dvdgp obtain ka kb ka' kb' where
    kab: \(a = ?g \ \times \ ka \ \&\ b = ?g \ \times \ kb \ \&\ ?a' = ?g' \ \times \ ka' \ ?b' = ?g' \ \times \ kb'\)
  unfolding dvd-def by blast
  then have \(?g \ \times \ ?a' = (?g \ \times \ ?g') \ \times \ ka' \ ?g \ \times \ ?b' = (?g \ \times \ ?g') \ \times \ kb'\)
    by simp-all
  then have dvdgp': \(?g \ \times \ ?g' \ \text{dvd} \ a \ \&\ ?g \ \times \ ?g' \ \text{dvd} \ b\)
    by (auto simp add: dvd-mult-div-cancel[of dvdg1])
    dvd-mult-div-cancel[of dvdg2] dvd-def)
  have \(?g \neq 0\) using nz by simp
  then have gp: \(?g > 0\) by arith
  from gcd-greatest-nat[OF dvdgp'] have \(?g \ \times \ ?g' \ \text{dvd} \ ?g\)
    with dvd-mult-cancel1[of gp] show \(?g' = 1\) by simp
qed

lemma div-gcd-coprime-int:
  assumes nz: \((a::\text{int}) \neq 0 \ \vee\ b \neq 0\)
  shows coprime \((a \ \text{div} \ gcd \ a \ b) \ (b \ \text{div} \ gcd \ a \ b)\)
apply (subst (1 2 3) gcd-abs-int)
apply simp
apply simp
apply (subst (1 2) abs-div)
  apply simp
apply simp
apply (subst (1 2) abs-gcd-int)
apply (rule div-gcd-coprime-nat[transferred])
using nz apply (auto simp add: gcd-abs-int[symmetric])
THEORY "GCD"

done

lemma coprime-nat: coprime (a::nat) b ←→ (∀ d. d dvd a ∧ d dvd b ←→ d = 1)
  using gcd-unique-nat[of 1 a b, simplified] by auto

lemma coprime-Suc-0-nat:
  coprime (a::nat) b ←→ (∀ d. d dvd a ∧ d dvd b ←→ d = Suc 0)
  using coprime-nat by (simp add: One-nat-def)

lemma coprime-int: coprime (a::int) b ←→ (∀ d > = 0 ∧ d dvd a ∧ d dvd b ←→ d = 1)
  using gcd-unique-int[of 1 a b]
  apply clarsimp
  apply (erule subst)
  apply (rule iffI)
  apply force
  apply (drule-tac x = abs ?e in exI)
  apply force
  apply force
  done

lemma gcd-coprime-nat:
  assumes z: gcd (a::nat) b ≠ 0 and a: a = a' * gcd a b and
          b: b = b' * gcd a b
  shows  coprime a' b'
  apply (subgoal-tac a' = a div gcd a b)
  apply (erule ssubst)
  apply (subgoal-tac b' = b div gcd a b)
  apply (erule ssubst)
  apply (rule div-gcd-coprime-nat)
  using z apply force
  apply (subst (1) b)
  using z apply force
  apply (subst (1) a)
  using z apply force
  done

lemma gcd-coprime-int:
  assumes z: gcd (a::int) b ≠ 0 and a: a = a' * gcd a b and
          b: b = b' * gcd a b
  shows  coprime a' b'
  apply (subgoal-tac a' = a div gcd a b)
  apply (erule ssubst)
  apply (subgoal-tac b' = b div gcd a b)
  apply (erule ssubst)
  apply (rule div-gcd-coprime-int)
using z apply force
apply (subst (1) b)
using z apply force
apply (subst (1) a)
using z apply force
done

lemma coprime-mult-nat: assumes da: coprime (d::nat) a and db: coprime d b
  shows coprime d (a * b)
apply (subst gcd-commute-nat)
using da apply (subst gcd-mult-cancel-nat)
apply (subst gcd-commute-nat, assumption)
apply (subst gcd-commute-nat, rule db)
done

lemma coprime-mult-int: assumes da: coprime (d::int) a and db: coprime d b
  shows coprime d (a * b)
apply (subst gcd-commute-int)
using da apply (subst gcd-mult-cancel-int)
apply (subst gcd-commute-int, assumption)
apply (subst gcd-commute-int, rule db)
done

lemma coprime-lmult-nat: assumes dab: coprime (d::nat) (a * b)
  shows coprime d a
proof
  have gcd d a dvd gcd d (a * b)
    by (rule gcd-greatest-nat, auto)
with dab show ?thesis
  by auto
qed

lemma coprime-lmult-int: assumes coprime (d::int) (a * b)
  shows coprime d a
proof
  have gcd d a dvd gcd d (a * b)
    by (rule gcd-greatest-int, auto)
with assms show ?thesis
  by auto
qed

lemma coprime-rmult-nat: assumes coprime (d::nat) (a * b)
  shows coprime d b
proof
  have gcd d b dvd gcd d (a * b)
    by (rule gcd-greatest-nat, auto intro: ded-mult)
with assms show ?thesis
  by auto
qed
lemma coprime-rmult-int:
  assumes dab: coprime (d::int) (a * b) shows coprime d b
proof
  have gcd d b dvd gcd d (a * b)
    by (rule gcd-greatest-int, auto intro: dvd-mult)
  with dab show ?thesis
    by auto
qed

lemma coprime-mul-eq-nat: coprime (d::nat) (a * b) ←→
  coprime d a ∧ coprime d b
using coprime-rmult-nat[of d a b] coprime-lmult-nat[of d a b]
  coprime-mult-nat[of d a b]
by blast

lemma coprime-mul-eq-int: coprime (d::int) (a * b) ←→
  coprime d a ∧ coprime d b
using coprime-rmult-int[of d a b] coprime-lmult-int[of d a b]
  coprime-mult-int[of d a b]
by blast

lemma coprime-power-int:
  assumes 0 < n shows coprime (a :: int) (b ^ n) ←→ coprime a b
using assms
proof (induct n)
  case (Suc n) then show ?case
    by (cases n) (simp-all add: coprime-mul-eq-int)
qed simp

lemma gcd-coprime-exists-nat:
  assumes nz: gcd (a::nat) b ≠ 0
  shows ∃a' b'. a = a' * gcd a b ∧ b = b' * gcd a b ∧ coprime a' b'
apply (rule-tac x = a div gcd a b in exI)
apply (rule-tac x = b div gcd a b in exI)
using nz apply (auto simp add: div-gcd-coprime-nat dvd-div-mult)
done

lemma gcd-coprime-exists-int:
  assumes nz: gcd (a::int) b ≠ 0
  shows ∃a' b'. a = a' * gcd a b ∧ b = b' * gcd a b ∧ coprime a' b'
apply (rule-tac x = a div gcd a b in exI)
apply (rule-tac x = b div gcd a b in exI)
using nz apply (auto simp add: div-gcd-coprime-int dvd-div-mult-self)
done

lemma coprime-exp-nat: coprime (d::nat) a ⇒ coprime d (a ^ n)
  by (induct n, simp-all add: coprime-mult-nat)
lemma coprime-exp-int: coprime \((d::int)\) \(a \Rightarrow\) coprime \(d (a^n)\)
by (induct \(n\), simp-all add: coprime-mult-int)

lemma coprime-exp2-nat [intro]: coprime \((a::nat)\) \(b \Rightarrow\) coprime \((a^n) (b^m)\)
apply (rule coprime-exp-nat)
apply (subst gcd-commute-nat)
apply (rule coprime-exp-nat)
apply (subst gcd-commute-nat, assumption)
done

lemma coprime-exp2-int [intro]: coprime \((a::int)\) \(b \Rightarrow\) coprime \((a^n) (b^m)\)
apply (rule coprime-exp-int)
apply (subst gcd-commute-int)
apply (rule coprime-exp-int)
apply (subst gcd-commute-int, assumption)
done

lemma gcd-exp-nat: \(gcd ((a::nat)^n) (b^n) = (gcd a b)^n\)
proof (cases)
assume \(a = 0 \& b = 0\)
thus ?thesis by simp
next assume \(~(a = 0 \& b = 0)\)
hence coprime \(((a \ div \ gcd a b)^n) ((b \ div \ gcd a b)^n)\)
by (auto simp:div-gcd-coprime-nat)
hence \(gcd ((a \ div \ gcd a b)^n * (gcd a b)^n)\)
\(((b \ div \ gcd a b)^n * (gcd a b)^n) = (gcd a b)^n\)
apply (subst (1 2) mult.commute)
apply (subst gcd-mult-distrib-nat [symmetric])
apply simp
done
also have \((a \ div \ gcd a b)^n * (gcd a b)^n = a^n\)
apply (subst div-power)
apply auto
apply (rule dvd-div-mult-self)
apply (rule dvd-power-same)
apply auto
done
also have \((b \ div \ gcd a b)^n * (gcd a b)^n = b^n\)
apply (subst div-power)
apply auto
apply (rule dvd-div-mult-self)
apply (rule dvd-power-same)
apply auto
done
finally show ?thesis.
qed

lemma gcd-exp-int: \(gcd \((a::int)^n) (b^n) = (gcd a b)^n\)
apply (subst \((1 2)\) gcd-abs-int)
proof
apply (subst (1 2) power-abs)
apply (rule gcd-exp-nat [where \( n = n \), transferred])
apply auto
done

lemma division-decomp-nat: assumes \( dc\): \((a::nat)\) dvd \( b \cdot c\)
shows \( \exists b'\ c'.\ a = b' \cdot c' \land b' dvd b \land c' dvd c\)
proof–
let \(?g = gcd\ a\ b\)
{assume \(?g = 0\) with \( dc\) have \(?\)thesis by auto}
moreover
{assume \(?z\) ?\(g \neq 0\)
  from gcd-coprime-exists-nat[OF \( z\)]
  obtain \( a'\ b'\) where \( ab': a = a' * ?g b = b' * ?g\) coprime \( a'\ b'\)
  by blast
  have \( thb: ?g dvd b\) by auto
  from \( ab'(1)\) have \( a' dvd a\) unfolding dvd-def by blast
  with \( dc\) have \( th0: a' dvd b + c\) using dvd-trans[of \( a'\ a\ b + c\)] by simp
  from \( dc\ ab'(1,2)\) have \( a'*?g dvd (b'*?g) * c\) by auto
  hence \(?g*a' dvd ?g * (b' * c)\) by (simp add: mult.assoc)
  with \( z\) have \( th-1: a' dvd b' * c\) by auto
  from \( coprime-dvd-mult-nat[OF ab'(3)]\) \( th-1\)
  have \( thc: a' dvd c\) by (subst (asm) mult.commute, blast)
  from \( ab'\) have \( a = ?g*a'\) by algebra
  with \( thb\ thc\) have \(?\)thesis by blast }
ultimately show \(?\)thesis by blast
qed

lemma division-decomp-int: assumes \( dc\): \((a::int)\) dvd \( b \cdot c\)
shows \( \exists b'\ c'.\ a = b' \cdot c' \land b' dvd b \land c' dvd c\)
proof–
let \(?g = gcd\ a\ b\)
{assume \(?g = 0\) with \( dc\) have \(?\)thesis by auto}
moreover
{assume \(?z\) ?\(g \neq 0\)
  from gcd-coprime-exists-int[OF \( z\)]
  obtain \( a'\ b'\) where \( ab': a = a' * ?g b = b' * ?g\) coprime \( a'\ b'\)
  by blast
  have \( thb: ?g dvd b\) by auto
  from \( ab'(1)\) have \( a' dvd a\) unfolding dvd-def by blast
  with \( dc\) have \( th0: a' dvd b + c\)
    using dvd-trans[of \( a'\ a\ b + c\)] by simp
  from \( dc\ ab'(1,2)\) have \( a'*?g dvd (b'*?g) * c\) by auto
  hence \(?g*a' dvd ?g * (b' * c)\) by (simp add: mult.assoc)
  with \( z\) have \( th-1: a' dvd b' * c\) by auto
  from \( coprime-dvd-mult-int[OF ab'(3)]\) \( th-1\)
  have \( thc: a' dvd c\) by (subst (asm) mult.commute, blast)
  from \( ab'\) have \( a = ?g*a'\) by algebra
  with \( thb\ thc\) have \(?\)thesis by blast }
ultimately show \(?\)thesis by blast
qed
ultimately show ?thesis by blast

qed

lemma pow-divides-pow-nat:
  assumes \( ab : (a::nat)^n \vdots b^m \) and \( n : n \neq 0 \)
  shows \( a \vdots b \)
proof
  let \( ?g = \text{gcd} \ a \ b \)
  from \( n \) obtain \( m \) where \( m = \text{Suc} \ n \) by (cases \( n \), simp-all)
  \{\( \text{assume } ?g = 0 \ \text{with } ab \ \text{have } ?\text{thesis by auto} \} \)
moreover
  \{\( \text{assume } z : ?g \neq 0 \) \}
  hence \( zn : ?g \cdot n \neq 0 \) using \( n \) by simp
  from \( \text{gcd-coprime-exists-nat}(OF \ z) \)
  obtain \( a' b' \) where \( ab' : a = a' \cdot g \ b = b' \cdot g \) \( \text{coprime } a' b' \)
  by blast
  from \( ab \) have \( (a' \cdot g) \cdot n \vdots (b' \cdot g) \cdot n \)
  by (simp add: \( ab'(1,2)[\text{symmetric}] \))
  hence \( ?g^\cdot a' \cdot n \vdots ?g^\cdot b' \cdot n \)
  by (simp only: \( \text{power-mul} \- \text{distrib} \) \( \text{mult.commute} \))
with \( zn z n \) have \( \text{th0} : a'^\cdot n \vdots b'^\cdot n \) by auto
have \( a' \cdot g \ vdots a'\cdot n \) by (simp \( \text{add: } m \))
with \( \text{th0} \) have \( a' \cdot g \vdots b'^\cdot n \) using \( \text{dvd-trans}[of \ a' a'' n b'' n] \) by simp
hence \( th1 : a' \cdot g \vdots b'' m \cdot b' \cdot g \) by (simp \( \text{add: } m \text{ mult.commute} \))
from \( \text{coprime-dvd-mult-nat}(OF \ \text{coprime-exp-nat} \ [OF \ ab'(3), \ of \ m]) \) \( \text{th1} \)
have \( a' \cdot g \vdots b' \cdot g \) by (subst (asm) \( \text{mult.commute, blast} \))
  hence \( ?g \cdot a' \cdot n \vdots ?g \cdot b'' \cdot n \)
  by (simp only: \( \text{power-mul} \- \text{distrib} \) \( \text{mult.commute} \))
with \( zn z n \) have \( th0 : a'^\cdot n \vdots b'^\cdot n \) by auto
ultimately show ?thesis by blast

qed

lemma pow-divides-pow-int:
  assumes \( ab : (a::int)^n \vdots b^m \) and \( n : n \neq 0 \)
  shows \( a \vdots b \)
proof
  let \( ?g = \text{gcd} \ a \ b \)
  from \( n \) obtain \( m \) where \( m = \text{Suc} \ n \) by (cases \( n \), simp-all)
  \{\( \text{assume } ?g = 0 \ \text{with } ab \ \text{have } ?\text{thesis by auto} \} \)
moreover
  \{\( \text{assume } z : ?g \neq 0 \) \}
  hence \( zn : ?g \cdot n \neq 0 \) using \( n \) by simp
  from \( \text{gcd-coprime-exists-int}(OF \ z) \)
  obtain \( a' b' \) where \( ab' : a = a' \cdot g \ b = b' \cdot g \) \( \text{coprime } a' b' \)
  by blast
  from \( ab \) have \( (a' \cdot g) \cdot n \vdots (b' \cdot g) \cdot n \)
  by (simp add: \( ab'(1,2)[\text{symmetric}] \))
  hence \( ?g^\cdot n \cdot a' \cdot n \vdots ?g^\cdot n \cdot b'' \cdot n \)
  by (simp only: \( \text{power-mul} \- \text{distrib} \) \( \text{mult.commute} \))
with \( zn z n \) have \( \text{th0} : a'^\cdot n \vdots b'^\cdot n \) by auto
have 

have 

with 

using 

hence 

from 

hence 

with 

ultimately show 

qed

lemma 

lemma 

lemma 

lemma 

proof –

from 

unfolding 

from 

then obtain 

from 

show 

qed

lemma 

lemma 

lemma 

lemma 

apply 

apply 

apply 

done

lemma 

lemma 

using coprime-plus-one-nat by (simp add: One-nat-def)

lemma coprime-plus-one-int [simp]: coprime ((n::int) + 1) n
  apply (subgoal-tac gcd (n + 1) n dvd (n + 1 - n))
  apply force
  apply (rule dvd-diff)
  apply auto
done

lemma coprime-minus-one-nat: (n::nat) ≠ 0 ⇒ coprime (n - 1) n
  using coprime-plus-one-nat [of n - 1]
  gcd-commute-nat [of n - 1 n] by auto

lemma coprime-minus-one-int: coprime ((n::int) - 1) n
  using coprime-plus-one-int [of n - 1]
  gcd-commute-int [of n - 1 n] by auto

lemma setprod-coprime-nat [rule-format]:
  (ALL i: A. coprime (f i) (x::nat)) ---+ coprime (PROD i:A. f i) x
  apply (case-tac finite A)
  apply (induct set: finite)
  apply (auto simp add: gcd-mult-cancel-nat)
done

lemma setprod-coprime-int [rule-format]:
  (ALL i: A. coprime (f i) (x::int)) ---+ coprime (PROD i:A. f i) x
  apply (case-tac finite A)
  apply (induct set: finite)
  apply (auto simp add: gcd-mult-cancel-int)
done

lemma coprime-common-divisor-nat: coprime (a::nat) b ⇒ x dvd a ⇒
  x dvd b ⇒ x = 1
  apply (subgoal-tac x dvd gcd a b)
  apply simp
  apply (erule (1) gcd-greatest-nat)
done

lemma coprime-common-divisor-int: coprime (a::int) b ⇒ x dvd a ⇒
  x dvd b ⇒ abs x = 1
  apply (subgoal-tac x dvd gcd a b)
  apply simp
  apply (erule (1) gcd-greatest-int)
done

lemma coprime-divisors-nat: (d::int) dvd a ⇒ e dvd b ⇒ coprime a b ⇒
  coprime d e
  apply (auto simp add: dvd-def)
  apply (frule coprime-lmult-int)
apply (subst gcd-commute-int)
apply (subst (asm) (2) gcd-commute-int)
apply (erule coprime-lmult-int)
done

lemma invertible-coprime-nat: (\(x::\text{n}\at\) \(y \mod m = 1 \Rightarrow \text{coprime } x m\)
apply (metis coprime-lmult-nat gcd-1-nat gcd-commute-nat gcd-red-nat)
done

lemma invertible-coprime-int: (\(x::\text{n}\at\) \(y \mod m = 1 \Rightarrow \text{coprime } x m\)
apply (metis coprime-lmult-int gcd-1-int gcd-commute-int gcd-red-int)
done

93.5 Bezout’s theorem

fun \text{bezw} :: \text{n} \Rightarrow \text{n} \Rightarrow \text{n} \Rightarrow \text{n} \Rightarrow \text{n} \Rightarrow \text{n}
where \text{bezw} x y =
(if \(y = 0\) then \((1, 0)\) else
\((\text{snd} (\text{bezw} y (x \mod y)),\
\quad \text{fst} (\text{bezw} y (x \mod y)) - \text{snd} (\text{bezw} y (x \mod y)) * \text{int}(x \div y))\))

lemma bezw-0 [simp]: bezw x 0 = (1, 0) by simp

lemma bezw-non-0: \(y > 0 \Rightarrow \text{bezw} x y = (\text{snd} (\text{bezw} y (x \mod y)),\
\quad \text{fst} (\text{bezw} y (x \mod y)) - \text{snd} (\text{bezw} y (x \mod y)) * \text{int}(x \div y))\)
by simp

declare bezw.simps [simp del]

lemma bezw-aux [rule-format]:
\(\text{fst} (\text{bezw} x y) * \text{int} x + \text{snd} (\text{bezw} x y) * \text{int} y = \text{int} (\text{gcd} x y)\)
proof (induct x y rule: gcd-nat-induct)
fix m :: \text{n}
show \(\text{fst} (\text{bezw} m 0) * \text{int} m + \text{snd} (\text{bezw} m 0) * \text{int} 0 = \text{int} (\text{gcd} m 0)\)
by auto
next fix m :: \text{n} and n
assume ngt0: \(n > 0 \and\)
\(\text{ih: } \text{fst} (\text{bezw} n (m \mod n)) * \text{int} n +\
\text{snd} (\text{bezw} n (m \mod n)) * \text{int} (m \mod n) =\
\text{int} (\text{gcd} n (m \mod n))\)
thus \(\text{fst} (\text{bezw} m n) * \text{int} m + \text{snd} (\text{bezw} m n) * \text{int} n = \text{int} (\text{gcd} m n)\)
apply (simp add: bezw-non-0 gcd-non-0-nat)
apply (erule subst)
apply (simp add: field-simps)
apply (subst mod-div-equality [of m n, symmetric])
apply (simp only: field-simps of-nat-add of-nat-mult)
done
qed

lemma bezout-int:
  fixes x y
  shows EX u v. u * (x::int) + v * y = gcd x y
proof –
  have bezout-aux: !!x y. x ≥ (0::int) → y ≥ 0 →
    EX u v. u * x + v * y = gcd x y
    apply (rule-tac x = fst (bezw (nat x) (nat y)) in exI)
    apply (rule-tac x = snd (bezw (nat x) (nat y)) in exI)
    apply (unfold gcd-int-def)
    apply simp
    apply (subst bezw-aux [symmetric])
    apply auto
    done
  have (x ≥ 0 ∧ y ≥ 0) | (x ≥ 0 ∧ y ≤ 0) | (x ≤ 0 ∧ y ≥ 0) |
    (x ≤ 0 ∧ y ≤ 0)
  by auto
  moreover have x ≥ 0 → y ≥ 0 → thesis
    by (erule (1) bezout-aux)
  moreover have x ≥ 0 → y ≥ 0 → thesis
    apply (insert bezout-aux [of x − y])
    apply auto
    apply (rule-tac x = v in exI)
    apply (rule-tac x = −v in exI)
    apply (subst gcd-neg2-int [symmetric])
    apply auto
    done
  moreover have x ≤ 0 → y ≥ 0 → thesis
    apply (insert bezout-aux [of −x y])
    apply auto
    apply (rule-tac x = −u in exI)
    apply (rule-tac x = u in exI)
    apply (subst gcd-neg1-int [symmetric])
    apply auto
    done
  moreover have x ≤ 0 → y ≤ 0 → thesis
    apply (insert bezout-aux [of −x − y])
    apply auto
    apply (rule-tac x = −u in exI)
    apply (rule-tac x = −v in exI)
    apply (subst gcd-neg1-int [symmetric])
    apply (subst gcd-neg2-int [symmetric])
    apply auto
    done
  ultimately show thesis by blast
qed

versions of Bezout for nat, by Amine Chaieb
**THEORY “GCD”**

lemma **ind-euclid**:  
assumes \( c: \forall a b. P (a::nat) b \iff P b a \)  
and \( \text{add}: \forall a b. P a b \implies P a (a + b) \)  
shows \( P a b \)

**proof** (\text{induct \( a + b \) arbitrary; \( a b \) rule: \text{less-induct}})

\begin{itemize}
  \item **case** \text{less} \n  \begin{itemize}
    \item have \( a = b \lor a < b \lor b < a \) \text{ by arith}
  \end{itemize}
  \item moreover \{assume \( eq: a = b \)
    \begin{itemize}
      \item from \text{add[rule-format, OF z[rule-format, af a]]} have \( P a b \) \text{ using eq}
      \item by \text{simp}
    \end{itemize}
  \}
  \item moreover \{assume \( lt: a < b \)
    \begin{itemize}
      \item hence \( a + b - a < a + b \lor a = 0 \) \text{ by arith}
      \item moreover \{assume \( a = 0 \) with \text{z c} have \( P a b \) \text{ by blast} \}
      \item moreover \{assume \( a + b - a < a + b \)
        \begin{itemize}
          \item also have \( \text{th0: } a + b - a = a + (b - a) \) \text{ using \( lt \) by arith}
          \item finally have \( a + (b - a) < a + b \).
          \item then have \( P a (a + (b - a)) \) \text{ by \text{rule add[rule-format, OF less]}}
          \item then have \( P a b \) \text{ by \text{simp add: \text{th0[symmetric]}}}
          \item ultimately have \( P a b \) \text{ by blast} \}
        \}
    \end{itemize}
  \}
  \item moreover \{assume \( lt: a > b \)
    \begin{itemize}
      \item hence \( b + a - b < a + b \lor b = 0 \) \text{ by arith}
      \item moreover \{assume \( b = 0 \) with \text{z c} have \( P a b \) \text{ by blast} \}
      \item moreover \{assume \( b + a - b < a + b \)
        \begin{itemize}
          \item also have \( \text{th0: } b + a - b = b + (a - b) \) \text{ using \( lt \) by arith}
          \item finally have \( b + (a - b) < a + b \).
          \item then have \( P b (b + (a - b)) \) \text{ by \text{rule add[rule-format, OF less]}}
          \item then have \( P a b \) \text{ by \text{simp add: \text{th0[symmetric]}}}
          \item hence \( P a b \) \text{ using \( c \) by blast} \}
          \item ultimately have \( P a b \) \text{ by blast} \}
        \}
    \end{itemize}
  \}
\end{itemize}

ultimately show \( P a b \) \text{ by blast}

**qed**

lemma **bezout-lemma-nat**:  
assumes \( \text{ex}: \exists (d::nat) x y. d \text{ dvd } a \land d \text{ dvd } b \land \)
  \( (a * x = b * y + d \lor b * x = a * y + d) \)
shows \( \exists d x y. d \text{ dvd } a \land d \text{ dvd } a + b \land \)
  \( (a * x = (a + b) * y + d \lor (a + b) * x = a * y + d) \)
using \text{ex}

apply \text{clarsimp}
apply (\text{rule-tac } x=d \text{ in } \text{exI}, \text{ simp})
apply (\text{case-tac } a * x = b * y + d , \text{ simp-all})
apply (\text{rule-tac } x=x + y \text{ in } \text{exI})
apply (\text{rule-tac } x=y \text{ in } \text{exI})
apply algebra
apply (rule-tac \( x=x \) in exI)
apply (rule-tac \( x=x + y \) in exI)
apply algebra
done

lemma bezout-add-nat: \( \exists (d::nat) \) \( x \) \( y \). \( d \) dvd \( a \) \( \land \) \( d \) dvd \( b \) \( \land \)
\( (a \ast x = b \ast y + d) \) \( \lor \) \( b \ast x = a \ast y + d \)
apply (induct \( a \) \( b \) rule: ind-euclid)
apply blast
apply clarify
apply (rule-tac \( x=a \) in exI, simp)
apply clarsimp
apply (rule-tac \( x=d \) in exI)
apply (rule-tac \( x=x+y \) in exI)
apply algebra
apply (rule-tac \( x=x \) in exI)
apply (rule-tac \( x=x+y \) in exI)
apply algebra
done

lemma bezoutI-nat: \( \exists (d::nat) \) \( x \) \( y \). \( d \) dvd \( a \) \( \land \) \( d \) dvd \( b \)
\( (a \ast x - b \ast y = d) \) \( \lor \) \( b \ast x - a \ast y = d \)
using bezout-add-nat[of \( a \) \( b \)]
apply clarsimp
apply (rule-tac \( x=d \) in exI, simp)
apply (rule-tac \( x=x \) in exI)
apply (rule-tac \( x=y \) in exI)
apply auto
done

lemma bezout-add-strong-nat: assumes \( nz \): \( a \neq (0::nat) \)
shows \( \exists d \) \( x \) \( y \). \( d \) dvd \( a \) \( \land \) \( d \) dvd \( b \) \( \land \)
\( a \ast x = b \ast y + d \)
proof -
from \( nz \) have \( ap: a > 0 \) by simp
from bezout-add-nat[of \( a \) \( b \)]
have \( (\exists d \) \( x \) \( y \). \( d \) dvd \( a \) \( \land \) \( d \) dvd \( b \) \( \land \) \( a \ast x = b \ast y + d \)) \( \lor \)
\( (\exists d \) \( x \) \( y \). \( d \) dvd \( a \) \( \land \) \( d \) dvd \( b \) \( \land \) \( b \ast x = a \ast y + d \)) by blast
moreover
{ fix \( d \) \( x \) \( y \) assume \( H: d \) dvd \( a \) \( d \) dvd \( b \) \( a \ast x = b \ast y + d \)
from \( H \) have \( \)thesis by blast }
moreover
{ fix \( d \) \( x \) \( y \) assume \( H: d \) dvd \( a \) \( d \) dvd \( b \) \( b \ast x = a \ast y + d \)
{ assume \( b0: b = 0 \) with \( H \) have \( \)thesis by simp }
moreover
{ assume \( b: b \neq 0 \) hence \( bp: b > 0 \) by simp
from \( b \) dvd-imp-le [OF \( H(2) \)] have \( d < b \) \( \lor \) \( d = b \) }
proof
moreover
{assume db: d=b
with nz H have ?thesis apply simp
  apply (rule exI[where x=b], simp)
  apply (rule exI[where x=b])
  by (rule exI[where x=x−1], simp add: diff-mult-distrib2)
moreover
{assume db: d<b
  {assume x=0 hence ?thesis using nz H by simp }
moreover
{assume x0: x≠0 hence xp: x>0 by simp
  from db have d≤b−1 by simp
  hence d*b ≤ b*(b − 1) by simp
  with xp mult-mono[of 1 x d*b b*(b − 1)]
  have dble: d*b ≤ x*b*(b − 1) using bp by simp
  from H (3) have d + (b − 1) * (b*x) = d + (b − 1) * (a*y + d)
    by simp
  hence d + (b − 1) * a * y + (b − 1) * d = d + (b − 1) * b * x
    by (simp only: mult.assoc distrib-left)
  hence a * ((b − 1) * y) + d * (b − 1 + 1) = d + x*b*(b − 1)
    by algebra
  hence a * ((b − 1) * y) = d + x*b*(b − 1) − d*b using bp by simp
  hence a * ((b − 1) * y) = d + (x*b*(b − 1) − d*b)
    by (simp only: diff-add-assoc[OF dble, of d, symmetric])
  hence a * ((b − 1) * y) = b*(x*(b − 1) − d) + d
    by (simp only: diff-mult-distrib2 add.commute ac-simps)
  hence ?thesis using H(1,2)
apply −
  apply (rule exI[where x=d], simp)
  apply (rule exI[where x=(b − 1) * y])
  by (rule exI[where x=x*(b − 1) − d], simp))
ultimately have ?thesis by blast}
ultimately have ?thesis by blast}
ultimately have ?thesis by blast}
ultimately show ?thesis by blast
qed

lemma bezout-nat: assumes a: (a::nat) ≠ 0
shows ∃x y. a * x = b * y + gcd a b
proof −
let ?g = gcd a b
from bezout-add-strong-nat[OF a, of b]
  obtain d x y where d: d dvd a d dvd b a * x = b * y + d by blast
from d(1,2) have d dvd ?g by simp
then obtain k where k: ?g = d*k unfolding dvd-def by blast
from d(3) have a * x * k = (b * y + d) * k by auto
hence a * (x * k) = b * (y+k) + ?g by (algebra add: k)
thus ?thesis by blast
93.6 LCM properties

**lemma lcm-altdef-int** \[\text{code}: \text{lcm} \ (a::int) \ b = (\text{abs} \ a) \ast (\text{abs} \ b) \div \text{gcd} \ a \ b\]
*by* \(\text{(simp add: lcm-int-def lcm-nat-def zdiv-int of-nat-mult gcd-int-def)}\)

**lemma prod-gcd-lcm-nat** : \(m::nat) \ast n = \text{gcd} \ m \ n \ast \text{lcm} \ m \ n\)
*unfolding* lcm-nat-def
*by* \(\text{(simp add: dvd-mult-div-cancel [OF gcd-dvd-prod-nat])}\)

**lemma prod-gcd-lcm-int** :
abs(m::int) \ast abs n = \text{gcd} \ m \ n \ast \text{lcm} \ m \ n\)
*unfolding* lcm-int-def gcd-int-def
*apply* (subst int-mult [symmetric])
*apply* (subst prod-gcd-lcm-nat [symmetric])
*apply* (subst nat-abs-mult-distrib [symmetric])
*apply* (simp, simp add: abs-mult)
done

**lemma lcm-0-nat** \([simp]: \text{lcm} \ (m::nat) \ 0 = 0\)
*unfolding* lcm-nat-def *by* simp

**lemma lcm-0-int** \([simp]: \text{lcm} \ (m::int) \ 0 = 0\)
*unfolding* lcm-int-def *by* simp

**lemma lcm-0-left-nat** \([simp]: \text{lcm} \ (0::nat) \ n = 0\)
*unfolding* lcm-nat-def *by* simp

**lemma lcm-0-left-int** \([simp]: \text{lcm} \ (0::int) \ n = 0\)
*unfolding* lcm-int-def *by* simp

**lemma lcm-pos-nat:**
\((m::nat) > 0 \Longrightarrow \ n > 0 \Longrightarrow \text{lcm} \ m \ n > 0\)
*by* \(\text{(metis gr0I mult-is-0 prod-gcd-lcm-nat)}\)

**lemma lcm-pos-int:**
\((m::int) \sim 0 \Longrightarrow n \sim 0 \Longrightarrow \text{lcm} \ m \ n > 0\)
*apply* (subst lcm-abs-int)
*apply* (rule lcm-pos-nat [transferred])
*apply* auto
done

**lemma dvd-pos-nat:**
*fixes* \(n \ m :: \text{nat}\)
*assumes* \(n > 0 \ \text{and} \ m \ \text{dvd} \ n\)
*shows* \(m > 0\)
*using* assms *by* (cases m) auto
lemma lcm-least-nat:
  assumes (m::nat) dvd k and n dvd k
  shows lcm m n dvd k
proof (cases k)
  case 0 then show ?thesis by auto
next
  case (Suc -)
  then have pos-k: k > 0 by auto
  from assms dvd-pos-nat[of this] have pos-mn: m > 0 n > 0 by simp
  from assms obtain p where k-m: k = m * p using dvd-def by blast
  from assms obtain q where k-n: k = n * q using dvd-def by blast
  from pos-k k-m have pos-p: p > 0 by auto
  from pos-k k-n have pos-q: q > 0 by auto
  have k * k * gcd q p = k * gcd (k * q) (k * p)
    by (simp add: ac-simps gcd-mult-distrib-nat)
    also have ... = k * gcd (m * p * q) (n * q * p)
    by (simp add: k-m [symmetric] k-n [symmetric])
    also have ... = k * p * q * gcd m n
    by (simp add: ac-simps gcd-mult-distrib-nat)
    finally have (m * p) * (n * q) * gcd q p = k * p * q * gcd m n
      by (simp only: k-m [symmetric] k-n [symmetric])
    then have p * q * m * n * gcd q p = p * q * k * gcd m n
      by (simp add: ac-simps)
    with pos-p pos-q have m * n * gcd q p = k * gcd m n
      by simp
    with prod-gcd-lcm-nat[of m n]
    have lcm m n * gcd q p * gcd m n = k * gcd m n
      by (simp add: ac-simps)
    with pos-gcd have lcm m n * gcd q p = k by auto
    then show ?thesis using dvd-def by auto
qed

lemma lcm-least-int:
  (m::int) dvd k ==> n dvd k ==> lcm m n dvd k
apply (subst lcm-abs-int)
apply (rule dvd-trans)
apply (rule lcm-least-nat [transferred, of - abs k -])
apply auto
done

lemma lcm-dvd1-nat: (m::nat) dvd lcm m n
proof (cases m)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have mpos: m > 0 by simp
  show ?thesis
proof (cases n)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have npos: n > 0 by simp
  have gcd m n dvd n by simp
  then obtain k where n = gcd m n * k using dvd-def by auto
  then have m * n dvd gcd m n = m * (gcd m n * k) dvd gcd m n
    by (simp add: ac-simps)
  also have ... = m * k using mpos npos gcd-zero-nat by simp
  finally show ?thesis by (simp add: lcm-nat-def)
qed
qed

lemma lcm-dvd1-int: (m::int) dvd lcm m n
  apply (subst lcm-abs-int)
  apply (rule dvd-trans)
  prefer 2
  apply (rule lcm-dvd1-nat [transferred])
  apply auto
  done

lemma lcm-dvd2-nat: (n::nat) dvd lcm m n
  using lcm-dvd1-int [of n m] by (simp only: lcm-nat-def mult.commute gcd-nat.commute)

lemma lcm-dvd2-int: (n::int) dvd lcm m n
  using lcm-dvd1-int [of n m] by (simp only: lcm-int-def lcm-nat-def mult.commute gcd-nat.commute)

lemma dvd-lcm-I1-nat [simp]: (k::nat) dvd m =⇒ k dvd lcm m n
  by (metis lcm-dvd1-nat dvd-trans)

lemma dvd-lcm-I2-nat [simp]: (k::nat) dvd n =⇒ k dvd lcm m n
  by (metis lcm-dvd2-nat dvd-trans)

lemma dvd-lcm-I1-int [simp]: (i::int) dvd m =⇒ i dvd lcm m n
  by (metis lcm-dvd1-int dvd-trans)

lemma dvd-lcm-I2-int [simp]: (i::int) dvd n =⇒ i dvd lcm m n
  by (metis lcm-dvd2-int dvd-trans)

lemma lcm-unique-nat: (a::nat) dvd d ∧ b dvd d ∧
  (∀ e. a dvd e ∧ b dvd e −→ d dvd e) −→ d = lcm a b
  by (auto intro: dvd-antisym lcm-least-nat lcm-dvd1-nat lcm-dvd2-nat)

lemma lcm-unique-int: d ≥ 0 ∧ (a::int) dvd d ∧ b dvd d ∧
  (∀ e. a dvd e ∧ b dvd e −→ d dvd e) −→ d = lcm a b
  by (auto intro: dvd-antisym [transferred] lcm-least-int)

interpretation lcm-nat: abel-semigroup lcm :: nat ⇒ nat ⇒ nat
  + lcm-nat: semilattice-neutr lcm :: nat ⇒ nat ⇒ nat 1
proof
  fix \( n \), \( m \), \( p \) :: nat
  show \( \text{lcm} (\text{lcm} n m) p = \text{lcm} n (\text{lcm} m p) \)
    by (rule lcm-unique-nat [THEN iffD1]) (metis dvd.order-trans lcm-unique-nat)
  show \( \text{lcm} m n = \text{lcm} n m \)
    by (simp add: lcm-nat-def gcd-commute-nat field-simps)
  show \( \text{lcm} m m = m \)
    by (metis dvd.dual-order refl lcm-unique-nat)
  show \( \text{lcm} m 1 = m \)
    by (metis dvd refl lcm-unique-nat one-dvd)
qed

interpretation lcm-int: abel-semigroup \( \text{lcm} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \)
proof
  fix \( n \), \( m \), \( p \) :: int
  show \( \text{lcm} (\text{lcm} n m) p = \text{lcm} n (\text{lcm} m p) \)
    by (rule lcm-unique-int [THEN iffD1]) (metis dvd-trans lcm-unique-int)
  show \( \text{lcm} m n = \text{lcm} n m \)
    by (simp add: lcm-int-def lcm-nat commute)
qed

lemmas lcm-assoc-nat = lcm-nat.assoc
lemmas lcm-commute-nat = lcm-nat.commute
lemmas lcm-left-commute-nat = lcm-nat.left-commute
lemmas lcm-assoc-int = lcm-int.assoc
lemmas lcm-commute-int = lcm-int.commute
lemmas lcm-left-commute-int = lcm-int.left-commute

lemmas lcm-ac-nat = lcm-assoc-nat lcm-commute-nat lcm-left-commute-nat
lemmas lcm-ac-int = lcm-assoc-int lcm-commute-int lcm-left-commute-int

lemma lcm-proj2-if-dvd-nat [simp]: \( (x::nat) \) \( \text{dvd} y \Rightarrow \text{lcm} x y = y \)
  apply (rule sym)
  apply (subst lcm-unique-nat [symmetric])
  apply auto
  done

lemma lcm-proj2-if-dvd-int [simp]: \( (x::int) \) \( \text{dvd} y \Rightarrow \text{lcm} x y = \text{abs} y \)
  apply (rule sym)
  apply (subst lcm-unique-int [symmetric])
  apply auto
  done

lemma lcm-proj1-if-dvd-nat [simp]: \( (x::nat) \) \( \text{dvd} y \Rightarrow \text{lcm} y x = y \)
  by (subst lcm-commute-nat, erule lcm-proj2-if-dvd-nat)

lemma lcm-proj1-if-dvd-int [simp]: \( (x::int) \) \( \text{dvd} y \Rightarrow \text{lcm} y x = \text{abs} y \)
  by (subst lcm-commute-int, erule lcm-proj2-if-dvd-int)
lemma lcm-proj1-iff-nat[simp]: lcm m n = (m::nat) ↔ n dvd m
by (metis lcm-proj1-if-dvd-nat lcm-unique-nat)

lemma lcm-proj2-iff-nat[simp]: lcm m n = (n::nat) ↔ m dvd n
by (metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int)

lemma lcm-proj1-iff-int[simp]: lcm m n = \abs (m::int) ↔ n dvd m
by (metis gcd-unique-nat)

lemma lcm-proj2-iff-int[simp]: lcm m n = \abs (n::int) ↔ m dvd n
by (metis gcd-unique-nat)

lemma comp-fun-idem-gcd-nat: comp-fun-idem (gcd :: nat⇒ nat⇒ nat)
proof qed (auto simp add: gcd-ac-nat)

lemma comp-fun-idem-gcd-int: comp-fun-idem (gcd :: int⇒ int⇒ int)
proof qed (auto simp add: gcd-ac-int)

lemma comp-fun-idem-lcm-nat: comp-fun-idem (lcm :: nat⇒ nat⇒ nat)
proof qed (auto simp add: lcm-ac-nat)

lemma comp-fun-idem-lcm-int: comp-fun-idem (lcm :: int⇒ int⇒ int)
proof qed (auto simp add: lcm-ac-int)

lemma lcm-0-iff-nat[simp]: lcm (m::nat) n = 0 ↔ m=0 \or n=0
by (metis lcm-0-left-nat lcm-0-nat mult-is-0 prod-gcd-lcm-nat)

lemma lcm-0-iff-int[simp]: lcm (m::int) n = 0 ↔ m=0 \or n=0
by (metis lcm-0-int lcm-0-left-int lcm-pos-int less-le)

lemma lcm-1-iff-nat[simp]: lcm (m::nat) n = 1 ↔ m=1 \and n=1
by (metis lcm-unique-nat)

lemma lcm-1-iff-int[simp]: lcm (m::int) n = 1 ↔ \{m=1 \or m = -1\} \and \{n=1 \or n = -1\}
by (auto simp add: abs-mult-self trans [OF lcm-unique-int eq-commute, symmetric] 
zmult-eq-1-iff)

93.7 The complete divisibility lattice
interpretation gcd-semilattice-nat: semilattice-inf gcd op dvd (%m n::nat. m dvd n \& \sim n dvd m)
proof
  case goal3 thus ?case by (metis gcd-unique-nat)
qed auto
interpretation \textit{lcm-semilattice-nat}: \textit{semilattice-sup} lcm op dvd (%m n::nat. m dvd n & \sim n dvd m) \\
proof 
\begin{itemize}
  \item \textit{case} goal3 \textit{thus} ?case \textit{by} (metis \textit{lcm-unique-nat})
\end{itemize}
\textbf{qed} \textit{auto}

interpretation \textit{gcd-lcm-lattice-nat}: \textit{lattice gcd op dvd} (%m n::nat. m dvd n & \sim n dvd m) lcm ..

Lifting \textit{gcd} and \textit{lcm} to sets (\textit{Gcd/Lcm}). \textit{Gcd} is defined via \textit{lcm} to facilitate the proof that we have a complete lattice.

\textbf{class} \textit{Gcd} = \textit{gcd} + \\
\textbf{fixes} \textit{Gcd} :: 'a set \Rightarrow 'a \\
\textbf{fixes} \textit{Lcm} :: 'a set \Rightarrow 'a

\textbf{instantiation} nat :: \textit{Gcd} \\
\textbf{begin}

definition \textit{Lcm} (M::nat set) = (if finite M then \textit{semilattice-neutr-set}.F lcm 1 M else 0)

\textbf{interpretation} \textit{semilattice-neutr-set} lcm 1 ::nat ..

\textbf{lemma} \textit{Lcm-nat-infinite}: \\
\begin{equation}
\neg \text{finite } M \implies \textit{Lcm } M = (0::nat)
\end{equation} \\
\textit{by} (simp add: \textit{Lcm-nat-def})

\textbf{lemma} \textit{Lcm-nat-empty}: \\
\begin{equation}
\textit{Lcm } \{\} = (1::nat)
\end{equation} \\
\textit{by} (simp add: \textit{Lcm-nat-def})

\textbf{lemma} \textit{Lcm-nat-insert}: \\
\begin{equation}
\textit{Lcm } (\textit{insert } n M) = \textit{lcm } (n::nat) (\textit{Lcm } M)
\end{equation} \\
\textit{by} (cases finite M) (simp-all add: \textit{Lcm-nat-def} \textit{Lcm-nat-infinite})

definition \\
\textit{Gcd} (M::nat set) = \textit{Lcm } \{d. \forall m\in M. d \textit{ dvd } m\}

\textbf{instance} ..

end

\textbf{lemma} \textit{dvd-Lcm-nat} [simp]: \\
\textbf{fixes} M :: nat set \\
\textbf{assumes} m \in M \\
\textbf{shows} m \textit{ dvd } \textit{Lcm } M \\
\textbf{proof} (cases finite M) \\
\begin{itemize}
  \item \textit{case} False \textit{then} \textit{show} ?thesis \textit{by} (simp add: \textit{Lcm-nat-infinite})
\end{itemize}
\textbf{next}
case True then show \( \varphi \)thesis using assms by (induct M) (auto simp add: Lcm-nat-insert)
qed

lemma Lcm-dvd-nat [simp]:
  fixes M :: nat set
  assumes \( \forall m \in M . \ m \ dvd n \)
  shows Lcm M dvd n
proof (cases n = 0)
  assume n \neq 0
  hence finite \( \{d . \ d \ dvd n\} \) by (rule finite-divisors-nat)
  moreover have M \subseteq \{d . \ d \ dvd n\} using assms by fast
  ultimately have finite M by (rule rev-finite-subset)
  then show \( \varphi \)thesis using assms by (induct M) (simp-all add: Lcm-nat-empty Lcm-nat-insert)
qed simp

interpretation gcd-lcm-complete-lattice-nat:
  complete-lattice Gcd Lcm gcd Rings.dvd \( \lambda m . \ m \ dvd n \land \neg n \ dvd m \) lcm 1 \( \emptyset :: nat \)
where
  Inf.INFIMUM Gcd A f = Gcd (f \ A :: nat set)
  and Sup.SUPREMUM Lcm A f = Lcm (f \ A)
proof —
  show class.complete-lattice Gcd Lcm gcd Rings.dvd \( \lambda m . \ m \ dvd n \land \neg n \ dvd m \) lcm 1 \( \emptyset :: nat \)
  proof
    case goal1 thus \( \varphi \)case by (simp add: Gcd-nat-def)
  next
    case goal2 thus \( \varphi \)case by (simp add: Gcd-nat-def)
  next
    case goal5 show \( \varphi \)case by (simp add: Gcd-nat-def Lcm-nat-infinite)
  next
    case goal6 show \( \varphi \)case by (simp add: Lcm-nat-empty)
  next
    case goal3 thus \( \varphi \)case by simp
  next
    case goal4 thus \( \varphi \)case by simp
  qed
  then interpret gcd-lcm-complete-lattice-nat:
    complete-lattice Gcd Lcm gcd Rings.dvd \( \lambda m . \ m \ dvd n \land \neg n \ dvd m \) lcm 1 \( \emptyset :: nat \).
  from gcd-lcm-complete-lattice-nat.INF-def show Inf.INFIMUM Gcd A f = Gcd (f \ A) .
  from gcd-lcm-complete-lattice-nat.SUP-def show Sup.SUPREMUM Lcm A f = Lcm (f \ A) .
  qed

declare gcd-lcm-complete-lattice-nat.Inf-image-eq [simp del]
declare gcd-lcm-complete-lattice-nat.Sup-image-eq [simp del]
lemma Lcm-empty-nat: \( \text{Lcm } \{\} = (1::\text{nat}) \)
by (fact Lcm-nat-empty)

lemma Gcd-empty-nat: \( \text{Gcd } \{\} = (0::\text{nat}) \)
by (fact gcd-lcm-complete-lattice-nat.Inf-empty)

lemma Lcm-insert-nat [simp]:
shows \( \text{Lcm } (\text{insert } (n::\text{nat}) \text{ N}) = \text{lcm } n \text{ (Lcm } \text{ N}) \)
by (fact gcd-lcm-complete-lattice-nat.Sup-insert)

lemma Gcd-insert-nat [simp]:
shows \( \text{Gcd } (\text{insert } (n::\text{nat}) \text{ N}) = \text{gcd } n \text{ (Gcd } \text{ N}) \)
by (fact gcd-lcm-complete-lattice-nat.Inf-insert)

lemma Lcm0-iff [simp]:
finite \( \text{M} :: \text{nat set} \) = \( \text{⇒ M } \not= \{\} = \text{⇒ Lcm M } = \text{0 } \iff 0 : \text{M} \)
by (induct rule: finite-ne-induct) auto

lemma Lcm-eq-0 [simp]:
finite \( \text{M} :: \text{nat set} \) = \( \text{⇒ } 0 : \text{M } \iff \text{Lcm M } = \text{0} \)
by (metis Lcm0-iff empty-iff)

lemma Gcd-dvd-nat [simp]:
fixes \( \text{M} :: \text{nat set} \)
assumes \( m \in \text{M} \) shows \( \text{Gcd M } \text{dvd } m \)
using assms by (fact gcd-lcm-complete-lattice-nat.Inf-lower)

lemma dvd-Gcd-nat [simp]:
fixes \( \text{M} :: \text{nat set} \)
assumes \( \forall m \in \text{M}. \text{n dvd m} \) shows \( n \text{ dvd } \text{Gcd M} \)
using assms by (simp only: gcd-lcm-complete-lattice-nat.Inf-greatest)

Alternative characterizations of Gcd:

lemma Gcd-eq-Max: \( \text{finite (M::nat set) } \iff M \not= \{\} \iff \text{Lcm M } = 0 \iff 0 : M \)
apply (rule antisym)
apply (rule Max-ge)
apply (metis all-not-in-conv finite-divisors-nat finite-INT)
apply simp
apply (rule Max-le-iff [THEN iffD2])
apply (metis all-not-in-conv finite-divisors-nat finite-INT)
apply fastforce
apply clarsimp
apply (metis Gcd-dvd-nat Max-in dvd-0-left dvd-Gcd-nat dvd-imp-le linorder-antisym_conv3 not-less0)
done

lemma Gcd-remove0-nat: \( \text{finite M } \iff \text{Gcd M } = \text{Gcd } (\text{M } - \{0::\text{nat}\}) \)
apply (induct pred: finite)
apply simp
apply (case-tac x = 0)
apply simp
apply (subgoal-tac insert x F = {0} = insert x (F - {0}))
apply simp
apply blast
done

lemma Lcm-in-lcm-closed-set-nat:
finite M \implies M \neq \{\} \implies \forall m n :: \text{nat}. m:M \longrightarrow n:M \longrightarrow \text{lcm m n : M \longrightarrow Lcm M : M}
apply (induct rule:finite-linorder-min-induct)
apply simp
apply simp
apply (subgoal-tac \forall m n :: \text{nat}. m:A \longrightarrow n:A \longrightarrow \text{lcm m n : A})
apply simp
apply (case-tac A = \{\})
apply simp
apply simp
apply simp
apply (metis lcm-pos-nat lcm-unique-nat linorder-neq-iff nat-dvd-not-less not-less0)
done

lemma Lcm-eq-Max-nat:
finite M \implies M \neq \{\} \implies 0 \notin M \implies \forall m n :: \text{nat}. m:M \longrightarrow n:M \longrightarrow \text{lcm m n : M \longrightarrow Lcm M = \text{Max} M}
apply (rule antisym)
apply (rule Max-ge, assumption)
apply (erule (2) Lcm-in-lcm-closed-set-nat)
apply clarsimp
apply (metis Lcm0-iff dvd-Lcm-nat dvd-imp-le neq0-conv)
done

lemma Lcm-set-nat [code, code-unfold]:
Lcm (set ns) = fold lcm ns (1::nat)
by (fact gcd-lcm-complete-lattice-nat.Sup-set-fold)

lemma Gcd-set-nat [code, code-unfold]:
Gcd (set ns) = fold gcd ns (0::nat)
by (fact gcd-lcm-complete-lattice-nat.Inf-set-fold)

lemma mult-inj-if-coprime-nat:
inj-on f A \implies inj-on g B \implies \forall a:A. \forall b:B. \text{coprime (f a) (g b)}
\implies inj-on (%(a,b). f a * g b::nat) (A \times B)
apply (auto simp add: inj-on-def)
apply (metis coprime-dvd-mult-iff-nat dvdgneq-le-trans dvdtriv-left)
apply (metis gcd-semilattice-nat.inf-commute coprime-dvd-mult-iff-nat
dvdgneq-le-trans dvdtriv-right mult.commute)
done

Nitpick:
lemma gcd_eq_nitpick_gcd [nitpick-unfold]: \( \gcd x y = \text{Nitpick.nat-gcd} x y \)
by (induct \( x \) \( y \) rule: nat-gcd.induct)
  (simp add: gcd-nat.simps Nitpick.nat-gcd.simps)

lemma lcm_eq_nitpick_lcm [nitpick-unfold]: \( \lcm x y = \text{Nitpick.nat-lcm} x y \)
by (simp only: lcm-nat-def Nitpick.nat-lcm-def gcd_eq_nitpick_gcd)

93.7.1 Setwise gcd and lcm for integers

instantiation \( \text{int} \) :: \( \text{Gcd} \)
begin

definition \( \text{Lcm} \ M = \text{int} (\text{Lcm} (\text{nat} \cdot \text{abs} \cdot M)) \)

definition \( \text{Gcd} \ M = \text{int} (\text{Gcd} (\text{nat} \cdot \text{abs} \cdot M)) \)

instance ..
end

lemma Lcm_empty_int [simp]: \( \text{Lcm} \{} = (1 :: \text{int}) \)
by (simp add: Lcm_int_def)

lemma Gcd_empty_int [simp]: \( \text{Gcd} \{} = (0 :: \text{int}) \)
by (simp add: Gcd_int_def)

lemma Lcm_insert_int [simp]:
  shows \( \text{Lcm} (\text{insert} (n :: \text{int}) N) = \text{lcm} n (\text{Lcm} N) \)
by (simp add: Lcm_int_def lcm_int_def)

lemma Gcd_insert_int [simp]:
  shows \( \text{Gcd} (\text{insert} (n :: \text{int}) N) = \text{gcd} n (\text{Gcd} N) \)
by (simp add: Gcd_int_def gcd_int_def)

lemma dvd_int_iff: \( x \text{ dvd} y \longleftrightarrow \text{nat} (\text{abs} x) \text{ dvd} \text{nat} (\text{abs} y) \)
by (simp add: zdvd_int)

lemma dvd_Lcm_int [simp]:
  fixes \( M :: \text{int set} \)
  assumes \( m \in M \) shows \( m \text{ dvd} \text{Lcm} M \)
using assms by (simp add: Lcm_int_def dvd_int_iff)

lemma Lcm_dvd_int [simp]:
  fixes \( M :: \text{int set} \)
  assumes \( \forall m \in M. \text{ m dvd n} \) shows \( \text{Lcm} M \text{ dvd} n \)
using assms by (simp add: Lcm_int_def dvd_int_iff)

lemma Gcd_dvd_int [simp]:
  fixes \( M :: \text{int set} \)
assumes \( m \in M \) shows \( \gcd M \text{ dvd } m \) using assms by (simp add: Gcd-int-def dvd-int-iff)

lemma \( \text{dvd-Gcd-int[simp]} \):
fixes \( M :: \text{int set} \)
assumes \( \forall m \in M. \ n \text{ dvd } m \) shows \( n \text{ dvd } \gcd M \)
using assms by (simp add: Gcd-int-def dvd-int-iff)

lemma \( \text{Lcm-set-int [code, code-unfold]} \):
\( \text{Lcm \ (set xs) = fold lcm xs (1::int)} \)
by (induct xs rule: rev-induct) (simp-all add: lcm-commute-int)

lemma \( \text{Gcd-set-int [code, code-unfold]} \):
\( \text{Gcd \ (set xs) = fold gcd xs (0::int)} \)
by (induct xs rule: rev-induct) (simp-all add: gcd-commute-int)

end

94 Archimedean-Field: Archimedean Fields, Floor and Ceiling Functions

theory Archimedean-Field
imports Main
begin

94.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class archimedean-field = linordered-field +
  assumes \text{ex-le-of-int}: \( \exists z. x \leq \text{of-int } z \)

lemma \( \text{ex-less-of-int:} \):
fixes \( x :: 'a::archimedean-field \) shows \( \exists z. x < \text{of-int } z \)
proof –
  from \text{ex-le-of-int} obtain \( z \) where \( x \leq \text{of-int } z \) ..
  then have \( x < \text{of-int } (z + 1) \) by simp
  then show \(?thesis \) ..
qed

lemma \( \text{ex-of-int-less:} \):
fixes \( x :: 'a::archimedean-field \) shows \( \exists z. \text{of-int } z < x \)
proof –
  from \text{ex-less-of-int} obtain \( z \) where \( x < \text{of-int } z \) ..
  then have \( \text{of-int } (\neg z) < x \) by simp
  then show \(?thesis \) ..
qed

lemma \( \text{ex-less-of-nat:} \):
fixes $x :: 'a::archimedean-field$ shows $\exists n. x < \text{of-nat } n$
proof -
obtain $z$ where $x < \text{of-int } z$ using ex-less-of-int ..
also have $\ldots \leq \text{of-int } (\text{int } (\text{nat } z))$ by simp
also have $\ldots = \text{of-nat } (\text{nat } z)$ by (simp only: of-int-of-nat-eq)
finally show ?thesis ..
qed

lemma ex-le-of-nat:
fixes $x :: 'a::archimedean-field$ shows $\exists n. x \leq \text{of-nat } n$
proof -
obtain $n$ where $x < \text{of-nat } n$ using ex-less-of-nat ..
then have $x \leq \text{of-nat } n$ by simp
then show ?thesis ..
qed

Archimedean fields have no infinitesimal elements.

lemma ex-inverse-of-nat-Suc-less:
fixes $x :: 'a::archimedean-field$ assumes $0 < x$ shows $\exists n. \text{inverse } (\text{of-nat } (\text{Suc } n)) < x$
proof -
from $\langle 0 < x \rangle$ have $0 < \text{inverse } x$
by (rule positive-imp-inverse-positive)
obtain $n$ where $\text{inverse } x < \text{of-nat } n$
using ex-less-of-nat ..
then obtain $m$ where $\text{inverse } x < \text{of-nat } (\text{Suc } m)$
using $\langle 0 < \text{inverse } x \rangle$ by (cases $n$) (simp-all del: of-nat-Suc)
then have $\text{inverse } (\text{of-nat } (\text{Suc } m)) < \text{inverse } (\text{inverse } x)$
using $\langle 0 < \text{inverse } x \rangle$ by (rule less-imp-inverse-less)
then have $\text{inverse } (\text{of-nat } (\text{Suc } m)) < x$
using $\langle 0 < x \rangle$ by (simp add: nonzero-inverse-inverse-eq)
then show ?thesis ..
qed

lemma ex-inverse-of-nat-less:
fixes $x :: 'a::archimedean-field$ assumes $0 < x$ shows $\exists n > 0. \text{inverse } (\text{of-nat } n) < x$
using ex-inverse-of-nat-Suc-less [OF $\langle 0 < x \rangle$] by auto

lemma ex-less-of-nat-mult:
fixes $x :: 'a::archimedean-field$ assumes $0 < x$ shows $\exists n. y < \text{of-nat } n * x$
proof -
obtain $n$ where $y / x < \text{of-nat } n$ using ex-less-of-nat ..
with $\langle 0 < x \rangle$ have $y < \text{of-nat } n * x$ by (simp add: pos-divide-less-eq)
then show ?thesis ..
qed
94.2 Existence and uniqueness of floor function

**lemma** exists-least-lemma:

**assumes** \( \neg P \ 0 \) and \( \exists n. \ P \ n \)

**shows** \( \exists n. \ \neg P \ n \land P \ (Suc \ n) \)

**proof**

- from \( \exists n. \ P \ n \) have \( P \ (Least \ P) \) by (rule LeastI-ex)

- with \( \neg P \ 0 \) obtain \( n \) where \( Least \ P = Suc \ n \)

- by (cases Least P) auto

- then have \( n < Least \ P \) by simp

- then have \( \neg P \ n \) by (rule not-less-Least)

- then have \( \neg P \ n \land P \ (Suc \ n) \)

- using \( P \ (Least \ P) \) \( Least \ P = Suc \ n \) by simp

- then show \( \neg \)thesis ..

**qed**

**lemma** floor-exists:

**fixes** \( x :: \ a::archimedean-field \)

**shows** \( \exists z. \ of-int \ z \leq x \land x < of-int (z + 1) \)

**proof** (cases)

- assume \( 0 \leq x \)

- then have \( \neg x < of-nat \ 0 \) by simp

- then have \( \exists n. \ \neg x < of-nat n \land x < of-nat (Suc \ n) \)

- using ex-less-of-nat by (rule exists-least-lemma)

- then obtain \( n \) where \( \neg x < of-nat n \land x < of-nat (Suc \ n) \) ..

- then have \( of-int \ (int \ n) \leq x \land x < of-int \ (int \ n + 1) \) by simp

- then show \( \neg \)thesis ..

next

- assume \( \neg 0 \leq x \)

- then have \( \neg - x \leq of-nat \ 0 \) by simp

- then have \( \exists n. \ \neg - x \leq of-nat n \land - x \leq of-nat (Suc \ n) \)

- using ex-le-of-nat by (rule exists-least-lemma)

- then obtain \( n \) where \( \neg - x \leq of-nat n \land - x \leq of-nat (Suc \ n) \)..

- then have \( of-int \ (- int \ n - 1) \leq x \land x < of-int \ (- int \ n - 1 + 1) \) by simp

- then show \( \neg \)thesis ..

**qed**

**lemma** floor-exists1:

**fixes** \( x :: \ a::archimedean-field \)

**shows** \( \exists! z. \ of-int \ z \leq x \land x < of-int (z + 1) \)

**proof** (rule ex-ex1I)

- show \( \exists z. \ of-int \ z \leq x \land x < of-int (z + 1) \)

- by (rule floor-exists)

next

- fix \( y \ z \) assume

- \( of-int \ y \leq x \land x < of-int \ (y + 1) \)

- \( of-int \ z \leq x \land x < of-int \ (z + 1) \)

- with le-less-trans [of int y x of int (z + 1)]

- le-less-trans [of int z x of int (y + 1)]

- show \( y = z \) by (simp del: of-int-add)
94.3 Floor function

class floor-ceiling = archimedean-field +
  fixes floor :: 'a ⇒ int
  assumes floor-correct: of-int (floor x) ≤ x ∧ x < of-int (floor x + 1)

notation (xsymbols)
  floor (\floor{-})

notation (HTML output)
  floor (\floor{-})

lemma floor-unique: [of-int z ≤ x; x < of-int z + 1] \imp floor x = z
  using floor-correct [of x] floor-exists1 [of x] by auto

lemma of-int-floor-le: of-int (floor x) ≤ x
  using floor-correct ..

lemma le-floor-iff: z ≤ floor x \iff of-int z ≤ x
  proof
    assume z ≤ floor x
    then have (of-int z :: 'a) ≤ of-int (floor x) by simp
    also have of-int (floor x) ≤ x by (rule of-int-floor-le)
    finally show of-int z ≤ x .
  next
    assume of-int z ≤ x
    also have x < of-int (floor x + 1) using floor-correct ..
    finally show z ≤ floor x by (simp del: of-int-add)
  qed

lemma floor-less-cancel: floor x < floor y =⇒ x < y
  by (simp add: not-le [symmetric] le-floor-iff)

lemma floor- mono: assumes x ≤ y shows floor x ≤ floor y
  proof
    have of-int (floor x) ≤ x by (rule of-int-floor-le)
    also note (x ≤ y)
    finally show ?thesis by (simp add: le-floor-iff)
  qed

lemma floor-less-cancel: floor x < floor y =⇒ x < y
  by (simp add: not-le [symmetric] less-floor-iff)
by (auto simp add: not-le [symmetric] floor-mono)

lemma floor-of-int [simp]: floor (of-int z) = z
  by (rule floor-unique) simp-all

lemma floor-of-nat [simp]: floor (of-nat n) = int n
  using floor-of-int [of of-nat n] by simp

lemma le-floor-add: floor x + floor y ≤ floor (x + y)
  by (simp only: le-floor-iff of-int-add add-mono of-int-floor-le)

Floor with numerals

lemma floor-zero [simp]: floor 0 = 0
  using floor-of-int [of 0] by simp

lemma floor-one [simp]: floor 1 = 1
  using floor-of-int [of 1] by simp

lemma floor-numeral [simp]: floor (numeral v) = numeral v
  using floor-of-int [of numeral v] by simp

lemma floor-neg-numeral [simp]: floor (− numeral v) = − numeral v
  using floor-of-int [of − numeral v] by simp

lemma zero-le-floor [simp]: 0 ≤ floor x ↔ 0 ≤ x
  by (simp add: le-floor-iff)

lemma one-le-floor [simp]: 1 ≤ floor x ↔ 1 ≤ x
  by (simp add: le-floor-iff)

lemma numeral-le-floor [simp]:
  numeral v ≤ floor x ↔ numeral v ≤ x
  by (simp add: le-floor-iff)

lemma neg-numeral-le-floor [simp]:
  − numeral v ≤ floor x ↔ − numeral v ≤ x
  by (simp add: le-floor-iff)

lemma zero-less-floor [simp]: 0 < floor x ↔ 1 ≤ x
  by (simp add: less-floor-iff)

lemma one-less-floor [simp]: 1 < floor x ↔ 2 ≤ x
  by (simp add: less-floor-iff)

lemma numeral-less-floor [simp]:
  numeral v < floor x ↔ numeral v + 1 ≤ x
  by (simp add: less-floor-iff)

lemma neg-numeral-less-floor [simp]:
THEORY “Archimedean-Field”

− numeral v < ∑ x ←− numeral v + 1 ≤ x
  by (simp add: less-floor-iff)

lemma floor-le-zero [simp]: floor x ≤ 0 ←− x < 1
  by (simp add: floor-le-iff)

lemma floor-le-one [simp]: floor x ≤ 1 ←− x < 2
  by (simp add: floor-le-iff)

lemma floor-le-numeral [simp]:
  floor x ≤ numeral v ←− x < numeral v + 1
  by (simp add: floor-le-iff)

lemma floor-le-neg-numeral [simp]:
  floor x ≤ −numeral v ←− x < −numeral v + 1
  by (simp add: floor-le-iff)

lemma floor-less-zero [simp]: floor x < 0 ←− x < 0
  by (simp add: floor-less-iff)

lemma floor-less-one [simp]: floor x < 1 ←− x < 1
  by (simp add: floor-less-iff)

lemma floor-less-numeral [simp]:
  floor x < numeral v ←− x < numeral v
  by (simp add: floor-less-iff)

lemma floor-less-neg-numeral [simp]:
  floor x < −numeral v ←− x < −numeral v
  by (simp add: floor-less-iff)

Addition and subtraction of integers

lemma floor-add-of-int [simp]: floor (x + of-int z) = floor x + z
  using floor-correct [of x] by (simp add: floor-unique)

lemma floor-add-numeral [simp]:
  floor (x + numeral v) = floor x + numeral v
  using floor-add-of-int [of x numeral v] by simp

lemma floor-add-one [simp]: floor (x + 1) = floor x + 1
  using floor-add-of-int [of x 1] by simp

lemma floor-diff-of-int [simp]: floor (x − of-int z) = floor x − z
  using floor-add-of-int [of x − z] by (simp add: algebra-simps)

lemma floor-diff-numeral [simp]:
  floor (x − numeral v) = floor x − numeral v
  using floor-diff-of-int [of x numeral v] by simp
lemma floor-diff-one [simp]: floor (x - 1) = floor x - 1
using floor-diff-of-int [of x 1] by simp

94.4 Ceiling function

definition
  ceiling :: 'a::{floor, ceiling} ⇒ int where
  ceiling x = - floor (- x)

notation (xsymbols)
  ceiling ([\cdot\cdot])

notation (HTML output)
  ceiling ([\cdot\cdot])

lemma ceiling-correct: of-int (ceiling x) - 1 < x ∧ x ≤ of-int (ceiling x)
unfolding ceiling-def using floor-correct [of - x] by simp

lemma ceiling-unique: [of-int z - 1 < x; z ≤ of-int z] ⇒ ceiling x = z
unfolding ceiling-def using floor-unique [of - z - x] by simp

lemma le-of-int-ceiling: x ≤ of-int (ceiling x)
using ceiling-correct ..

lemma ceiling-le-iff: ceiling x ≤ z ←→ x ≤ of-int z
unfolding ceiling-def using le-floor-iff [of x z - 1] by simp

lemma less-ceiling-iff: z < ceiling x ←→ of-int z < x
by (simp add: not-le [symmetric] ceiling-le-iff)

lemma ceiling-less-iff: ceiling x < z ←→ x ≤ of-int z - 1
using ceiling-le-iff [of x z - 1] by simp

lemma le-ceiling-iff: z ≤ ceiling x ←→ of-int z - 1 < x
by (simp add: not-less [symmetric] ceiling-less-iff)

lemma ceiling-mono: x ≥ y ⇒ ceiling x ≥ ceiling y
unfolding ceiling-def by (simp add: floor-mono)

lemma ceiling-less-cancel: ceiling x < ceiling y ⇒ x < y
by (auto simp add: not-le [symmetric] ceiling-mono)

lemma ceiling-of-int [simp]: ceiling (of-int z) = z
by (rule ceiling-unique) simp-all

lemma ceiling-of-nat [simp]: ceiling (of-nat n) = int n
using ceiling-of-int [of of-nat n] by simp

lemma ceiling-add-le: ceiling (x + y) ≤ ceiling x + ceiling y
Ceiling with numerals

**Lemma ceiling-zero** [simp]: ceiling 0 = 0
  **Using** ceiling-of-int [of 0] **by simp**

**Lemma ceiling-one** [simp]: ceiling 1 = 1
  **Using** ceiling-of-int [of 1] **by simp**

**Lemma ceiling-numeral** [simp]: ceiling (numeral v) = numeral v
  **Using** ceiling-of-int [of numeral v] **by simp**

**Lemma ceiling-neg-numeral** [simp]: ceiling (− numeral v) = − numeral v
  **Using** ceiling-of-int [of − numeral v] **by simp**

**Lemma ceiling-le-zero** [simp]: ceiling x ≤ 0 ↔ x ≤ 0
  **By** (simp add: ceiling-le-iff)

**Lemma ceiling-le-one** [simp]: ceiling x ≤ 1 ↔ x ≤ 1
  **By** (simp add: ceiling-le-iff)

**Lemma ceiling-le-numeral** [simp]:
  ceiling x ≤ numeral v ↔ x ≤ numeral v
  **By** (simp add: ceiling-le-iff)

**Lemma ceiling-le-neg-numeral** [simp]:
  ceiling x ≤ − numeral v ↔ x ≤ − numeral v
  **By** (simp add: ceiling-le-iff)

**Lemma ceiling-less-zero** [simp]: ceiling x < 0 ↔ x ≤ −1
  **By** (simp add: ceiling-less-iff)

**Lemma ceiling-less-one** [simp]: ceiling x < 1 ↔ x ≤ 0
  **By** (simp add: ceiling-less-iff)

**Lemma ceiling-less-numeral** [simp]:
  ceiling x < numeral v ↔ x ≤ numeral v − 1
  **By** (simp add: ceiling-less-iff)

**Lemma ceiling-less-neg-numeral** [simp]:
  ceiling x < − numeral v ↔ x ≤ − numeral v − 1
  **By** (simp add: ceiling-less-iff)

**Lemma zero-le-ceiling** [simp]: 0 ≤ ceiling x ↔ −1 < x
  **By** (simp add: le-ceiling-iff)

**Lemma one-le-ceiling** [simp]: 1 ≤ ceiling x ↔ 0 < x
  **By** (simp add: le-ceiling-iff)
lemma numeral-le-ceiling [simp]:
numeral v ≤ ceiling x ←→ numeral v - 1 < x
by (simp add: le-ceiling-iff)

lemma neg-numeral-le-ceiling [simp]:
- numeral v ≤ ceiling x ←→ - numeral v - 1 < x
by (simp add: le-ceiling-iff)

lemma zero-less-ceiling [simp]:
0 < ceiling x ←→ 0 < x
by (simp add: less-ceiling-iff)

lemma one-less-ceiling [simp]:
1 < ceiling x ←→ 1 < x
by (simp add: less-ceiling-iff)

lemma numeral-less-ceiling [simp]:
umeral v < ceiling x ←→ numeral v < x
by (simp add: less-ceiling-iff)

lemma neg-numeral-less-ceiling [simp]:
- numeral v < ceiling x ←→ - numeral v < x
by (simp add: less-ceiling-iff)

Addition and subtraction of integers

lemma ceiling-add-of-int [simp]:
ceiling (x + of-int z) = ceiling x + z
using ceiling-correct [of x] by (simp add: ceiling-unique)

lemma ceiling-add-numeral [simp]:
ceiling (x + numeral v) = ceiling x + numeral v
using ceiling-add-of-int [of x numeral v] by simp

lemma ceiling-add-one [simp]:
ceiling (x + 1) = ceiling x + 1
using ceiling-add-of-int [of x 1] by simp

lemma ceiling-diff-of-int [simp]:
ceiling (x - of-int z) = ceiling x - z
using ceiling-diff-of-int [of x - z] by (simp add: algebra-simps)

lemma ceiling-diff-numeral [simp]:
ceiling (x - numeral v) = ceiling x - numeral v
using ceiling-diff-of-int [of x numeral v] by simp

lemma ceiling-diff-one [simp]:
ceiling (x - 1) = ceiling x - 1
using ceiling-diff-of-int [of x 1] by simp

lemma ceiling-diff-floor-le-1: ceiling x - floor x ≤ 1
proof
have of-int [x] - 1 < x
using ceiling-correct[of x] by simp
also have x < of-int [x] + 1
using floor-correct[of x] by simp-all
finally have of-int (\lceil x \rceil - \lfloor x \rfloor) < (of-int 2::'a)
  by simp
then show \?thesis
  unfolding of-int-less_iff by simp
qed

94.5 Negation

lemma floor-minus: floor (- x) = - ceiling x
  unfolding ceiling-def by simp

lemma ceiling-minus: ceiling (- x) = - floor x
  unfolding ceiling-def by simp

end

95 Rat: Rational numbers

theory Rat
imports GCD Archimedean-Field
begin

95.1 Rational numbers as quotient

95.1.1 Construction of the type of rational numbers

definition
  ratrel :: (int × int) ⇒ (int × int) ⇒ bool where
  ratrel = (λx y. snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y = fst y * snd x)

lemma ratrel-iff [simp]:
  ratrel x y ←→ snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y = fst y * snd x
  by (simp add: ratrel-def)

lemma exists-ratrel-refl: \exists x. ratrel x x
  by (auto intro!: one-neq-zero)

lemma symp-ratrel: symp ratrel
  by (simp add: ratrel-def symp-def)

lemma transp-ratrel: transp ratrel
proof (rule transpI, unfold split-paired-all)
  fix a b a' b' a'' b'' :: int
  assume A: ratrel (a, b) (a', b')
  assume B: ratrel (a'', b'') (a'', b'')
  have b' * (a * b'') = b'' * (a * b') by simp
  also from A have a * b' = a' * b by auto
  also have b'' * (a' * b) = b * (a' * b'') by simp
  also from B have a' * b'' = a'' * b' by auto
also have \( b \ast (a'' \ast b') = b' \ast (a'' \ast b) \) by simp
finally have \( b' \ast (a \ast b'') = b' \ast (a'' \ast b) \).
morerove from \( B \) have \( b' \neq 0 \) by auto
ultimately have \( a \ast b'' = a'' \ast b \) by simp
with \( A B \) show \( \text{ratrel} (a, b) (a'', b'') \) by auto
qed

**lemma** part-equivp-ratrel: part-equivp ratrel
by (rule part-equivpI [OF exists-ratrel-refl symp-ratrel transp-ratrel])

**quotient-type** rat = int \times int / partial: ratrel
**morphisms** Rep-Rat Abs-Rat
by (rule part-equivp-ratrel)

**lemma** Domainp-cr-rat [transfer-domain-rule]: Domainp pcr-rat = (\( \lambda x. \text{snd} x \neq 0 \))
by (simp add: rat.domain-eq)

95.1.2 Representation and basic operations

**lift-definition** Fract :: int \Rightarrow int \Rightarrow rat
is \( \lambda a b. \text{if } b = 0 \text{ then } (0, 1) \text{ else } (a, b) \)
by simp

**lemma** eq-rat:
shows \( \forall a b c d. b \neq 0 \implies d \neq 0 \implies \text{Fract} a b = \text{Fract} c d \iff a \ast d = c \ast b \)
and \( \forall a. \text{Fract} a 0 = \text{Fract} 0 1 \)
and \( \forall a c. \text{Fract} 0 a = \text{Fract} 0 c \)
by (transfer, simp)+

**lemma** Rat-cases [case-names Fract, cases type: rat]:
assumes \( \forall a b. q = \text{Fract} a b \implies b > 0 \implies \text{coprime} a b \implies C \)
shows C
proof –
obtain a b :: int where q = Fract a b and b \( \neq 0 \)
  by transfer simp
let ?a = a div gcd a b
let ?b = b div gcd a b
from \( \langle b \neq 0 \rangle \) have \( ?b \ast gcd a b = b \)
  by (simp add: dvd-div-mult-self)
with \( \langle b \neq 0 \rangle \) have \( ?b \neq 0 \) by auto
from \( q = \text{Fract} a b \langle b \neq 0 \rangle \langle ?b \neq 0 \rangle \) have q = Fract \( ?a ?b \)
  by (simp add: eq-rat dvd-div-mult mult.commute [of a])
from \( \langle b \neq 0 \rangle \) have \( \text{coprime} \?a \?b \)
  by (auto intro: dvd-gcd-coprime-int)
show C proof (cases b > 0)
case True
note assms
moreover note q
moreover from True have \(?b > 0\) by (simp add: nonneg1-imp-zdiv-pos-iff)
moreover note coprime
ultimately show \(C\).

next
case False
  note assms
moreover have \(q = \text{Fract} \ (-\ ?a) (-\ ?b)\) unfolding \(q\) by transfer simp
moreover from \(\text{False} : b \neq 0\) have \(-\ ?b > 0\) by (simp add: pos-imp-zdiv-neg-iff)
moreover from coprime have coprime \((-\ ?a) (-\ ?b)\) by simp
ultimately show \(C\).

qed

lemma Rat-induct [case_names Fract, induct type: rat]:
  assumes \(\land a \cdot b \cdot b > 0 \implies \text{coprime} a b \implies P (\text{Fract} a b)\)
  shows \(P q\)
  using assms by (cases \(q\)) simp

instantiation rat :: field-inverse-zero
begin

lift-definition zero-rat :: rat is \((0, 1)\)
by simp

lift-definition one-rat :: rat is \((1, 1)\)
by simp

lemma Zero-rat-def: \(0 = \text{Fract} 0 1\)
by transfer simp

lemma One-rat-def: \(1 = \text{Fract} 1 1\)
by transfer simp

lift-definition plus-rat :: rat \(\Rightarrow\) rat \(\Rightarrow\) rat
  is \(\lambda x \cdot y. ((\text{fst} x \cdot \text{snd} y + \text{fst} y \cdot \text{snd} x, \text{snd} x \cdot \text{snd} y))\)
by (clarsimp, simp add: distrib-right, simp add: ac-simps)

lemma add-rat [simp]:
    assumes \(b \neq 0\) and \(d \neq 0\)
  shows \(\text{Fract} a \cdot b + \text{Fract} c \cdot d = \text{Fract} (a \cdot d + c \cdot b) \cdot (b \cdot d)\)
  using assms by transfer simp

lift-definition uminus-rat :: rat \(\Rightarrow\) rat is \(\lambda x. (-\ \text{fst} x, \text{snd} x)\)
by simp

lemma minus-rat [simp]: \(-\ \text{Fract} a b = \text{Fract} (-\ a) b\)
by transfer simp

lemma minus-rat-cancel [simp]: \(\text{Fract} (- a) (- b) = \text{Fract} a b\)
by (cases $b = 0$) (simp-all add: eq-rat)

definition
diff-rat-def: $q - r = q + - (r::rat)$

lemma diff-rat [simp]:
  assumes $b \neq 0$ and $d \neq 0$
  shows Fract $a b - Fract c d = Fract (a * d - c * b) (b * d)$
  using assms by (simp add: diff-rat-def)

lift-definition times-rat :: rat $\Rightarrow$ rat $\Rightarrow$ rat
  is $\lambda x y. (fst x * fst y, snd x * snd y)$
  by (simp add: ac-simps)

lemma mult-rat [simp]: Fract $a b * Fract c d = Fract (a * c) (b * d)$
  by transfer simp

lemma mult-rat-cancel:
  assumes $c \neq 0$
  shows Fract $(c * a) (c * b) = Fract a b$
  using assms by transfer simp

lift-definition inverse-rat :: rat $\Rightarrow$ rat
  is $\lambda x. if fst x = 0 then (0, 1) else (snd x, fst x)$
  by (auto simp add: mult.commute)

lemma inverse-rat [simp]: inverse (Fract $a b$) = Fract $b a$
  by transfer simp

definition
divide-rat-def: $q / r = q * inverse (r::rat)$

lemma divide-rat [simp]: Fract $a b / Fract c d = Fract (a * d) (b * c)$
  by (simp add: divide-rat-def)

instance proof
  fix $q r s ::$ rat
  show $(q * r) * s = q * (r * s)$
    by transfer simp
  show $q * r = r * q$
    by transfer simp
  show $1 * q = q$
    by transfer simp
  show $q + r + s = q + (r + s)$
    by transfer (simp add: algebra-simps)
  show $q + r = r + q$
    by transfer simp
  show $0 + q = q$
    by transfer simp
show $-q + q = 0$
  by transfer simp
show $q - r = q + -r$
  by (fact diff-rat-def)
show $(q + r) * s = q * s + r * s$
  by transfer (simp add: algebra-simps)
show $(0::rat) \neq 1$
  by transfer simp
{  assume $q \neq 0$ thus $\inverse q * q = 1$
  by transfer simp }
show $q / r = q * \inverse r$
  by (fact divide-rat-def)
show $\inverse 0 = (0::rat)$
  by transfer simp
qed

end

lemma of-nat-rat: of-nat $k = \text{Fract}(\text{of-nat } k) 1$
  by (induct $k$) (simp-all add: Zero-rat-def One-rat-def)

lemma of-int-rat: of-int $k = \text{Fract} k 1$
  by (cases $k$ rule: int-diff-cases) (simp add: of-nat-rat)

lemma Fract-of-nat-eq: $\text{Fract}(\text{of-nat } k) 1 = \text{of-nat } k$
  by (rule of-nat-rat [symmetric])

lemma Fract-of-int-eq: $\text{Fract} k 1 = \text{of-int } k$
  by (rule of-int-rat [symmetric])

lemma rat-number-collapse:
  Fract 0 $k = 0$
  Fract 1 1 = 1
  Fract $(\text{numeral } w) 1 = \text{numeral } w$
  Fract $(\text{numeral } w) 1 = \text{numeral } w$
  Fract $(\text{numeral } w) 1 = \text{numeral } w$
  Fract $(\text{numeral } w) 1 = \text{numeral } w$
  Fract 0 0 = 0
  using Fract-of-int-eq [of numeral $w$]
  using Fract-of-int-eq [of numeral $w$]
  by (simp-all add: Zero-rat-def One-rat-def eq-rat)

lemma rat-number-expand:
  0 = Fract 0 1
  1 = Fract 1 1
  numeral $k = \text{Fract}(\text{numeral } k) 1$
  $-1 = \text{Fract}(\text{numeral } w) 1$
  $-\text{numeral } k = \text{Fract}(\text{numeral } w) 1$
  by (simp-all add: rat-number-collapse)
lemma Rat-cases-nonzero [case-names Fract 0]:
assumes Fract: \( \forall a \ b. \ q = \text{Fract} \ a \ b \implies b > 0 \implies a \neq 0 \implies \text{coprime} \ a \ b \implies C \)
assumes 0: \( q = 0 \implies C \)
shows C
proof (cases q = 0)
case True then show C using 0 by auto
next
case False then obtain a b where q = Fract a b and b > 0 and coprime a b by (cases q) auto
with False have 0 \neq Fract a b by simp
with (b > 0) have a \neq 0 by (simp add: Zero-rat-def eq-rat)
with Fract (q = Fract a b) (b > 0) (coprime a b) show C by blast
qed

95.1.3 Function normalize

lemma Fract-coprime: \( \text{Fract} \ (a \text{ div gcd} \ a \ b) \ (b \text{ div gcd} \ a \ b) = \text{Fract} \ a \ b \)
proof (cases b = 0)
case True then show ?thesis by (simp add: eq-rat)
next
case False moreover have \( b \text{ div gcd} \ a \ b \times \text{gcd} \ a \ b = b \)
by (rule dvd-div-mult-self) simp
ultimately have \( b \text{ div gcd} \ a \ b \neq 0 \) by auto
with False ?thesis by (simp add: eq-rat dvd-div-mult mult.commute [of a])
qed

definition normalize :: int \times int \Rightarrow int \times int where
normalize p = (if snd p > 0 then (let a = gcd (fst p) (snd p) in (fst p div a, snd p div a))
else if snd p = 0 then (0, 1)
else (let a = - gcd (fst p) (snd p) in (fst p div a, snd p div a)))

lemma normalize-crossproduct:
assumes q \neq 0 s \neq 0
assumes normalize (p, q) = normalize (r, s)
shows p \times s = r \times q
proof –
have aux: \( p \times \text{gcd} \ r \ s = \text{sgn} \ (q \times s) \times r \times \text{gcd} \ p \ q \implies q \times \text{gcd} \ r \ s = \text{sgn} \ (q \times s) \times s \times q \times r \times \text{gcd} \ p \ q \)
proof –
assume p \times \text{gcd} \ r \ s = \text{sgn} \ (q \times s) \times r \times \text{gcd} \ p \ q \text{ and } q \times \text{gcd} \ r \ s = \text{sgn} \ (q \times s) \times s \times \text{gcd} \ p \ q
\text{ then have } (p \times \text{gcd} \ r \ s) \times (\text{sgn} \ (q \times s) \times s \times \text{gcd} \ p \ q) = (q \times \text{gcd} \ r \ s) \times (\text{sgn} \ (q \times s) \times r \times \text{gcd} \ p \ q) \text{ by simp}
with assms show p \times s = q \times r by (auto simp add: ac-simps sgn-times sgn-0-0)
qed

from assms show ?thesis
  by (auto simp add: normalize-def Let-def dvd-div-div-eq-mult mult.commute
      sgn-times split: if-splits intro: aux)
qed

lemma normalize-eq: normalize \((a, b)\) = \((p, q)\) \(\implies\) \(\text{Fract } p q = \text{Fract } a b\)
  by (auto simp add: normalize-def Fract-coprime dvd-div-neg rat-number-collapse
      split:split-if-asm)

lemma normalize-denom-pos: normalize \(r = (p, q)\) \(\implies\) \(q > 0\)
  by (auto simp add: normalize-def Let-def dvd-div-neg pos-imp-zdiv-neg-iff nonneg1-imp-zdiv-pos-iff
      split:split-if-asm)

lemma normalize-coprime: normalize \(r = (p, q)\) \(\implies\) \(\text{coprime } p q\)
  by (auto simp add: normalize-def dvd-div-neg div-gcd-coprime-int
      split:split-if-asm)

lemma normalize-stable [simp]:
  \(q > 0 \implies \text{coprime } p q \implies \text{normalize } (p, q) = (p, q)\)
  by (simp add: normalize-def)

lemma normalize-denom-zero [simp]:
  \(\text{normalize } (p, 0) = (0, 1)\)
  by (simp add: normalize-def)

lemma normalize-negative [simp]:
  \(q < 0 \implies \text{normalize } (p, q) = \text{normalize } (-p, -q)\)
  by (simp add: normalize-def Let-def dvd-div-neg dvd-neg-div)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition quotient-of :: \(\text{rat} \Rightarrow \text{int} \times \text{int}\) where
  \(\text{quotient-of } x = (\text{THE pair. } x = \text{Fract } (\text{fst pair}) (\text{snd pair}) \&
      \text{snd pair} > 0 \& \text{coprime } (\text{fst pair}) (\text{snd pair}))\)

lemma quotient-of-unique:
  \(\exists ! p. \ r = \text{Fract } (\text{fst } p) (\text{snd } p) \& \text{snd } p > 0 \& \text{coprime } (\text{fst } p) (\text{snd } p)\)
proof (cases \(r\))
  case (Fract \(a b\))
  then have \(r = \text{Fract } (\text{fst } (a, b)) (\text{snd } (a, b)) \& \text{snd } (a, b) > 0 \& \text{coprime } (\text{fst } (a, b)) (\text{snd } (a, b))\) by auto
  then show ?thesis proof (rule ex1I)
    fix \(p\)
    obtain \(c d : \text{int}\) where \(p = (c, d)\) by (cases \(p\))
    assume \(r = \text{Fract } (\text{fst } p) (\text{snd } p) \& \text{snd } p > 0 \& \text{coprime } (\text{fst } p) (\text{snd } p)\)
    with \(p\) have \(\text{Fract'}: \ r = \text{Fract } c d d > 0 \text{ coprime } c d\) by simp-all
    have \(c = a \& d = b\)
    proof (cases \(a = 0\))
case True with Fract Fract' show ?thesis by (simp add: eq-rat)
next
case False
  with Fract Fract' have*: c * b = a * d and c ≠ 0 by (auto simp add: eq-rat)
  then have c * b > 0 ⟷ a * d > 0 by auto
  with (b > 0) (d > 0) have a > 0 ⟷ c > 0 by (simp add: zero-less-mult-iff)
  with (a ≠ 0) (c ≠ 0) have sgn: sgn a = sgn c by (auto simp add: not-less)
  from (coprime a b) (coprime c d) have |a| * |d| = |c| * |b| ⟷ |a| = |c| ∧ |d| = |b|
      by (simp add: coprime-crossproduct-int)
  with (b > 0) (d > 0) have |a| * d = |c| * b ⟷ |a| = |c| ∧ d = b by simp
  then have a * sgn a * d = c * sgn c * b ⟷ a * sgn a = c * sgn c ∧ d = b by (simp add: abs-sgn)
  with sgn * show ?thesis by (auto simp add: sgn-0-0)
qed
with p show p = (a, b) by simp
qed

lemma quotient-of-Fract [code]:
  quotient-of (Fract a b) = normalize (a, b)
proof –
  have Fract a b = Fract (fst (normalize (a, b))) (snd (normalize (a, b))) (is ?Fract)
      by (rule sgn) (auto intro: normalize-eq)
  moreover have 0 < snd (normalize (a, b)) (is ?denom-pos)
      by (cases normalize (a, b)) (rule normalize-denom-pos, simp)
  moreover have coprime (fst (normalize (a, b))) (snd (normalize (a, b))) (is ?coprime)
      by (rule normalize-coprime) simp
  ultimately have ?Fract ∧ ?denom-pos ∧ ?coprime by blast
  with quotient-of-unique have
    (THE p. Fract a b = Fract (fst p) (snd p) ∧ 0 < snd p ∧ coprime (fst p) (snd p)) = normalize (a, b)
        by (rule the1-equality)
  then show ?thesis by (simp add: quotient-of-def)
qed

lemma quotient-of-number [simp]:
  quotient-of 0 = (0, 1)
  quotient-of 1 = (1, 1)
  quotient-of (numeral k) = (numeral k, 1)
  quotient-of (- 1) = (- 1, 1)
  quotient-of (- numeral k) = (- numeral k, 1)
  by (simp-all add: rat-number-expand quotient-of-Fract)

lemma quotient-of-eq: quotient-of (Fract a b) = (p, q) ⟹ Fract p q = Fract a b
  by (simp add: quotient-of-Fract normalize-eq)
lemma quotient-of-denom-pos: quotient-of \( r = (p, q) \implies q > 0 \)
by (cases \( r \)) (simp add: quotient-of-Fract normalize-denom-pos)

lemma quotient-of-coprime: quotient-of \( r = (p, q) \implies \text{coprime } p q \)
by (cases \( r \)) (simp add: quotient-of-Fract normalize-coprime)

lemma quotient-of-inject:
assumes quotient-of \( a = \text{quotient-of } b \)
shows \( a = b \)
proof –
obtain \( p q r s \) where \( a = \text{Fract } p q \)
and \( b = \text{Fract } r s \)
and \( q > 0 \) and \( s > 0 \) by (cases \( a \), cases \( b \))
with assms show \( ?\text{thesis} \)
by (simp add: eq-rat quotient-of-Fract normalize-crossproduct)
qed

lemma quotient-of-inject-eq:
\( \text{quotient-of } a = \text{quotient-of } b \longleftrightarrow a = b \)
by (auto simp add: quotient-of-inject)

95.1.4 Various

lemma Fract-of-int-quotient: \( \text{Fract } k l = \text{of-int } k / \text{of-int } l \)
by (simp add: Fract-of-int-eq [symmetric])

lemma Fract-add-one: \( n \neq 0 \implies \text{Fract } (m + n) n = \text{Fract } m n + 1 \)
by (simp add: rat-number-expand)

lemma quotient-of-div:
assumes \( r: \text{quotient-of } r = (n,d) \)
shows \( r = \text{of-int } n / \text{of-int } d \)
proof –
from theI [OF quotient-of-unique[of \( r \)], unfolded \( r \)[unfolded quotient-of-def]]
have \( r = \text{Fract } n d \) by simp
thus ?thesis using Fract-of-int-quotient by simp
qed

95.1.5 The ordered field of rational numbers

lift-definition positive :: rat \( \Rightarrow \) bool
is \( \lambda x. \, 0 < \text{fst } x \times \text{snd } x \)

proof (clarsimp)
fix \( a \) \( b \) \( c \) \( d \) :: int
assume \( b \neq 0 \) and \( d \neq 0 \) and \( a * d = c * b \)
hence \( a * d * b * d = c * b * b * d \)
by simp
hence \( a * b * d^2 = c * d * b^2 \)
unfolding power2-eq-square by (simp add: ac-simps)
hence \( 0 < a * b * d^2 \longleftrightarrow 0 < c * d * b^2 \)
by simp
thus \( 0 < a \cdot b \iff 0 < c \cdot d \)
using \( \langle b \neq 0 \rangle \) and \( \langle d \neq 0 \rangle \).
by (simp add: zero-less-mult-iff)
qed

lemma positive-zero: \( \sim \text{positive} \ 0 \)
by transfer simp

lemma positive-add:
  \( \text{positive} \ x \implies \text{positive} \ y \implies \text{positive} \ (x + y) \)
apply transfer
apply (simp add: zero-less-mult-iff)
apply (elim disjE, simp-all add: add-pos-pos add-neg-neg
  mult-pos-neg mult-neg-pos mult-neg-neg)
done

lemma positive-mult:
  \( \text{positive} \ x \implies \text{positive} \ y \implies \text{positive} \ (x \ast y) \)
by transfer (drule (1) mult-pos-pos, simp add: ac-simps)

lemma positive-minus:
  \( \sim \text{positive} \ x \implies x \neq 0 \implies \text{positive} \ (-x) \)
by transfer (force simp: neq-iff zero-less-mult-iff mult-less-0-iff)

instantiation rat :: linordered-field-inverse-zero
begin

definition
  \( x < y \iff \text{positive} \ (y - x) \)

definition
  \( x \leq (y :: \text{rat}) \iff x < y \lor x = y \)

definition
  abs (a :: \text{rat}) = (if a < 0 then \(-a\) else a)

definition
  sgn (a :: \text{rat}) = (if a = 0 then 0 else if 0 < a then 1 else -1)

instance proof
fix a b c :: rat
show \( |a| = (\text{if} \ a < 0 \ \text{then} \ -a \ \text{else} \ a) \)
  by (rule abs-rat-def)
show \( a < b \iff a \leq b \land \sim b \leq a \)
  unfolding less-eq-rat-def less-rat-def
  by (auto, drule (1) positive-add, simp-all add: positive-zero)
show \( a \leq a \)
  unfolding less-eq-rat-def by simp
show $a \leq b \Rightarrow b \leq c \Rightarrow a \leq c$
  unfolding less-eq-rat-def less-rat-def
  by (auto, drule (1) positive-add, simp add: algebra-simps)

show $a \leq b \Rightarrow b \leq a \Rightarrow a = b$
  unfolding less-eq-rat-def less-rat-def
  by (auto, drule (1) positive-add, simp add: positive-zero)

show $a \leq b \Rightarrow c + a \leq c + b$
  unfolding less-eq-rat-def less-rat-def by auto

show $sgn a = (if \ a = 0 \ then \ 0 \ else \ if \ 0 < a \ then \ 1 \ else \ -1)$
  by (rule sgn-rat-def)

show $a \leq b \lor b \leq a$
  unfolding less-eq-rat-def less-rat-def
  by (auto dest: positive-minus)

show $a < b \Rightarrow 0 < c \Rightarrow c * a < c * b$
  unfolding less-rat-def
  by (drule (1) positive-mult, simp add: algebra-simps)

qed

end

instantiation rat :: distrib-lattice
begin

definition
  $(\text{inf} :: \text{rat} \Rightarrow \text{rat} \Rightarrow \text{rat}) = \text{min}$

definition
  $(\text{sup} :: \text{rat} \Rightarrow \text{rat} \Rightarrow \text{rat}) = \text{max}$

instance proof
  qed (auto simp add: inf-rat-def sup-rat-def max-min-distrib2)

end

lemma positive-rat: positive (Fract a b) $\longleftrightarrow$ $0 < a \cdot b$
  by transfer simp

lemma less-rat [simp]:
  assumes $b \neq 0$ and $d \neq 0$
  shows Fract a b < Fract c d $\longleftrightarrow$ $(a \cdot d) \cdot (b \cdot d) < (c \cdot b) \cdot (b \cdot d)$
  using assms unfolding less-rat-def
  by (simp add: positive-rat algebra-simps)

lemma le-rat [simp]:
  assumes $b \neq 0$ and $d \neq 0$
  shows Fract a b $\leq$ Fract c d $\longleftrightarrow$ $(a \cdot d) \cdot (b \cdot d) \leq (c \cdot b) \cdot (b \cdot d)$
  using assms unfolding le-less by (simp add: eq-rat)

lemma abs-rat [simp, code]: $|\text{Fract} a b| = \text{Fract} |a| |b|$
by (auto simp add: abs-rat-def zabs-def Zero-rat-def not-less le-less eq-rat zero-less-mult-iff)

lemma sgn-rat [simp, code]: \(\text{sgn} \ (\text{Fract} \ a \ b) = \text{of-int} \ (\text{sgn} \ a * \text{sgn} \ b)\)

unfolding Fract-of-int-eq
by (auto simp: zsgn-def sgn-rat-def Zero-rat-def eq-rat)

lemma Rat-induct-pos [case-names Fract, induct type: rat]:
assumes step: \(\forall a \ b. \ 0 < b \implies P \ (\text{Fract} \ a \ b)\)
shows P q
proof (cases q)
have step': \(\forall a \ b. \ b < 0 \implies P \ (\text{Fract} \ a \ b)\)
proof
fix a::int and b::int
assume b: b < 0
hence 0 < -b by simp
hence P (Fract (-a) (-b)) by (rule step)
thus P (Fract a b) by (simp add: order-less-imp-not-eq [OF b])
qed
case (Fract a b)
thus P q by (force simp add: linorder-neq-iff step step')
qed

lemma zero-less-Fract-iff:
\(\forall a \ b. \ 0 < b \implies 0 < \text{Fract} \ a \ b \iff 0 < a\)
by (simp add: Zero-rat-def zero-less-mult-iff)

lemma Fract-less-zero-iff:
\(\forall a \ b. \ 0 < b \implies \text{Fract} \ a \ b < 0 \iff a < 0\)
by (simp add: Zero-rat-def mult-less-0-iff)

lemma zero-le-Fract-iff:
\(\forall a \ b. \ 0 < b \implies 0 \leq \text{Fract} \ a \ b \iff 0 \leq a\)
by (simp add: Zero-rat-def mult-le-0-iff)

lemma Fract-le-zero-iff:
\(\forall a \ b. \ 0 < b \implies \text{Fract} \ a \ b \leq 0 \iff a \leq 0\)
by (simp add: Zero-rat-def zero-le-mult-iff)

lemma one-less-Fract-iff:
\(\forall a \ b. \ 0 < b \implies 1 < \text{Fract} \ a \ b \iff b < a\)
by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff:
\(\forall a \ b. \ 0 < b \implies \text{Fract} \ a \ b < 1 \iff a < b\)
by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma one-le-Fract-iff:
\(\forall a \ b. \ 0 < b \implies 1 \leq \text{Fract} \ a \ b \iff b \leq a\)
by (simp add: One-rat-def mult-le-cancel-right)

lemma Fract-le-one-iff:
\[ \begin{array}{c}
\emptyset < b \implies \frac{a}{b} \leq 1 \iff a \leq b \\
b \end{array} \]
by (simp add: One-rat-def mult-le-cancel-right)

95.1.6 Rationals are an Archimedean field

lemma rat-floor-lemma:
shows \( \lfloor \frac{a}{b} \rfloor \leq \frac{a}{b} \land \frac{a}{b} < \lfloor \frac{a}{b} + 1 \rfloor \)
proof -
  have \( \frac{a}{b} = \lfloor \frac{a}{b} \rfloor + \frac{a \mod b}{b} \)
  by (cases b = 0, simp, simp add: of-int-rat)
  moreover have \( 0 \leq \frac{a \mod b}{b} \land \frac{a \mod b}{b} < 1 \)
  unfolding Fract-of-int-quotient
  by (rule linorder-cases [of b 0]) (simp-all add: divide-nonpos-neg)
  ultimately show \(?thesis by simp\)
qed

instance rat :: archimedean-field
proof
  fix \( r :: \text{rat} \)
  show \( \exists z. r \leq \lfloor \frac{z}{1} \rfloor \)
  proof (induct r)
    case (Fract a b)
    have \( \frac{a}{b} \leq \lfloor \frac{a}{b} + 1 \rfloor \)
    using rat-floor-lemma [of a b] by simp
    then show \( \exists z. \frac{a}{b} \leq \lfloor \frac{z}{1} \rfloor \) ..
  qed
  qed

instantiation rat :: floor-ceiling
begin

definition [code del]:
\( \text{floor} (x :: \text{rat}) = (\text{THE} z. \lfloor \frac{z}{1} \rfloor \leq x \land x < \lfloor \frac{z}{1} + 1 \rfloor) \)

instance proof
  fix \( x :: \text{rat} \)
  show \( \lfloor \frac{\text{floor} x}{1} \rfloor \leq x \land x < \lfloor \frac{\text{floor} x + 1}{1} \rfloor \)
  unfolding floor-rat-def using floor-exists1 by (rule the1I')
  qed

end

lemma floor-Fract: \( \text{floor} (\frac{a}{b}) = a \div b \)
using rat-floor-lemma [of a b]
by (simp add: floor-unique)
95.2 Linear arithmetic setup

declaration ⟨⟨ K (Lin-Arith.add-inj-thms [@{thm of-nat-le-iff} RS iffD2, @{thm of-nat-eq-iff} RS iffD2] (* not needed because x < (y::nat) can be rewritten as Suc x <= y: of-nat-less-iff RS iffD2 *) #> Lin-Arith.add-inj-thms [@{thm of-int-le-iff} RS iffD2, @{thm of-int-eq-iff} RS iffD2] (* not needed because x < (y::int) can be rewritten as x + 1 <= y: of-int-less-iff RS iffD2 *) #> Lin-Arith.add-simps [@{thm neg-less-iff-less}, @{thm True-implies-equals}, @{thm distrib-left [where a = numeral v for v]}, @{thm distrib-left [where a = ~ numeral v for v]}, @{thm divide-1}, @{thm divide-zero-left}, @{thm times-divide-eq-right}, @{thm times-divide-eq-left}, @{thm minus-divide-left} RS sym, @{thm minus-divide-right} RS sym, @{thm of-int-minus}, @{thm of-int-diff}, @{thm of-int-of-nat-eq} ] #> Lin-Arith.add-simprocs Numerical-Simprocs.field-divide-cancel-numeral-factor #> Lin-Arith.add-inj-const (@{const-name of-nat}, @{typ nat = rat}) #> Lin-Arith.add-inj-const (@{const-name of-int}, @{typ int => rat}) ⟩⟩

95.3 Embedding from Rationals to other Fields

class field-char-0 = field + ring-char-0

subclass (in linordered-field) field-char-0 ..

context field-char-0

begin

lift-definition of-rat :: rat ⇒ 'a
  is λx. of-int (fst x) / of-int (snd x)

apply (clarsimp simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq)

apply (simp only: of-int-mult [symmetric])

done

end

lemma of-rat-rat: b ≠ 0 → of-rat (Fract a b) = of-int a / of-int b
by transfer simp

lemma of-rat-0 [simp]: of-rat 0 = 0
by transfer simp

lemma of-rat-1 [simp]: of-rat 1 = 1
by transfer simp
lemma of-rat-add: of-rat $(a + b) = of-rat a + of-rat b$
   by transfer (simp add: add-frac-eq)

lemma of-rat-minus: of-rat $(-a) = -of-rat a$
   by transfer simp

lemma of-rat-neg-one [simp]:
   of-rat $(-1) = -1$
   by (simp add: of-rat-minus)

lemma of-rat-diff: of-rat $(a - b) = of-rat a - of-rat b$
   using of-rat-add [of $a - b$] by (simp add: of-rat-minus)

lemma of-rat-mult: of-rat $(a * b) = of-rat a * of-rat b$
   apply transfer
   apply (simp add: divide-inverse nonzero-inverse-mult-distrib ac-simps)
done

lemma nonzero-of-rat-inverse:
   $a \neq 0 \Rightarrow of-rat (inverse a) = inverse (of-rat a)$
   apply (rule inverse-unique [symmetric])
   apply (simp add: of-rat-mult [symmetric])
done

lemma nonzero-of-rat-divide:
   $b \neq 0 \Rightarrow of-rat (a / b) = of-rat a / of-rat b$
   by (simp add: divide-inverse of-rat-mult nonzero-of-rat-inverse)

lemma of-rat-power:
   $(of-rat (a ^ n)::'a::{field-char-0, field-inverse-zero})$
   = of-rat a ^ of-rat n
   by (induct n) (simp-all add: of-rat-mult)

lemma of-rat-eq-iff [simp]: $(of-rat a = of-rat b) = (a = b)$
   apply transfer
   apply (simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq)
   apply (simp only: of-int-mult [symmetric] of-int-eq-iff)
done
lemma of-rat-eq-0-iff [simp]: \((\text{of-rat} \ a = 0) = (a = 0)\)
using of-rat-eq-iff [of - 0] by simp

lemma zero-eq-of-rat-iff [simp]: \((0 = \text{of-rat} \ a) = (0 = a)\)
using of-rat-eq-iff [of - 0] by simp

lemma of-rat-eq-1-iff [simp]: \((\text{of-rat} \ a = 1) = (a = 1)\)
using of-rat-eq-iff [of - 1] by simp

lemma one-eq-of-rat-iff [simp]: \((1 = \text{of-rat} \ a) = (1 = a)\)
by simp

lemma of-rat-less:
\((\text{of-rat} \ r :: 'a::linordered-field) < \text{of-rat} \ s \iff r < s\)
proof (induct r, induct s)
fix a b c d :: int
assume not-zero: \(b > 0\) \(d > 0\)
then have \(b * d > 0\) by simp
have of-int-divide-less-eq:
\((\text{of-int} \ a :: 'a) / \text{of-int} \ b < \text{of-int} \ c / \text{of-int} \ d\)
\iff \((\text{of-int} \ a :: 'a) * \text{of-int} \ d < \text{of-int} \ c * \text{of-int} \ b\)
using not-zero by (simp add: pos-less-divide-eq pos-divide-less-eq)
show \((\text{of-rat} \ (\text{Fract} \ a \ b)) :: 'a::linordered-field) < \text{of-rat} \ (\text{Fract} \ c \ d)\)
\iff \text{Fract} \ a \ b < \text{Fract} \ c \ d
using not-zero \((b * d > 0)\)
qed

lemma of-rat-less-eq:
\((\text{of-rat} \ r :: 'a::linordered-field) \leq \text{of-rat} \ s \iff r \leq s\)
unfolding le-less by (auto simp add: of-rat-less)

lemma of-rat-le-0-iff [simp]: \(((\text{of-rat} \ r :: 'a::linordered-field) \leq 0) = (r \leq 0)\)
using of-rat-less-eq [of r 0, where 'a='a] by simp

lemma zero-le-of-rat-iff [simp]: \((0 \leq (\text{of-rat} \ r :: 'a::linordered-field)) = (0 \leq r)\)
using of-rat-less-eq [of 0 r, where 'a='a] by simp

lemma of-rat-le-1-iff [simp]: \(((\text{of-rat} \ r :: 'a::linordered-field) \leq 1) = (r \leq 1)\)
using of-rat-less-eq [of r 1] by simp

lemma one-le-of-rat-iff [simp]: \(((\text{of-rat} \ r :: 'a::linordered-field) \leq 1) = (1 \leq r)\)
using of-rat-less-eq [of 1 r] by simp

lemma of-rat-less-0-iff [simp]: \(((\text{of-rat} \ r :: 'a::linordered-field) < 0) = (r < 0)\)
using of-rat-less [of r 0, where 'a='a] by simp

lemma zero-less-of-rat-iff [simp]: \((0 < (\text{of-rat} \ r :: 'a::linordered-field)) = (0 < r)\)
using of-rat-less [of 0 r, where 'a='a] by simp
lemma of-rat-less-1-iff [simp]: ((of-rat r :: 'a::linordered-field) < 1) = (r < 1)  
  using of-rat-less [of r 1] by simp

lemma one-less-of-rat-iff [simp]: (1 < (of-rat r :: 'a::linordered-field)) = (1 < r)  
  using of-rat-less [of 1 r] by simp

lemma of-rat-eq-id [simp]: of-rat = id
proof
  fix a
  show of-rat a = id a
  by (induct a)
    (simp add: of-rat-rat Fract-of-int-eq [symmetric])
qed

Collapse nested embeddings

lemma of-rat-of-nat-eq [simp]: of-rat (of-nat n) = of-nat n  
  by (induct n) (simp-all add: of-rat-add)

lemma of-rat-of-int-eq [simp]: of-rat (of-int z) = of-int z  
  by (cases z rule: int-diff-cases) (simp add: of-rat-diff)

lemma of-rat-numeral-eq [simp]:  
  of-rat (numeral w) = numeral w  
  using of-rat-of-int-eq [of numeral w] by simp

lemma of-rat-neg-numeral-eq [simp]:  
  of-rat (− numeral w) = − numeral w  
  using of-rat-of-int-eq [of − numeral w] by simp

lemmas zero-rat = Zero-rat-def
lemmas one-rat = One-rat-def

abbreviation  
  rat-of-nat :: nat ⇒ rat  
where  
  rat-of-nat ≡ of-nat

abbreviation  
  rat-of-int :: int ⇒ rat  
where  
  rat-of-int ≡ of-int

95.4 The Set of Rational Numbers
context field-char-0
begin

definition
THEORY "Rat"

\[ Rats :: 'a set \text{ where} \]
\[ Rats = \text{range of-rat} \]

notation (xsymbols)
\[ Rats (\mathbb{Q}) \]

end

lemma Rats-of-rat [simp]: \( \text{of-rat } r \in Rats \)
by (simp add: Rats-def)

lemma Rats-of-int [simp]: \( \text{of-int } z \in Rats \)
by (subst of-rat-of-int-eq [symmetric], rule Rats-of-rat)

lemma Rats-of-nat [simp]: \( \text{of-nat } n \in Rats \)
by (subst of-rat-of-nat-eq [symmetric], rule Rats-of-rat)

lemma Rats-number-of [simp]: \( \text{numeral } w \in Rats \)
by (subst of-rat-numeral-eq [symmetric], rule Rats-of-rat)

lemma Rats-0 [simp]: \( 0 \in Rats \)
apply (unfold Rats-def)
apply (rule range-eqI)
apply (rule of-rat-0 [symmetric])
done

lemma Rats-1 [simp]: \( 1 \in Rats \)
apply (unfold Rats-def)
apply (rule range-eqI)
apply (rule of-rat-1 [symmetric])
done

lemma Rats-add [simp]: \( [a \in Rats; b \in Rats] \implies a + b \in Rats \)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-add [symmetric])
done

lemma Rats-minus [simp]: \( a \in Rats \implies - a \in Rats \)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-minus [symmetric])
done

lemma Rats-diff [simp]: \( [a \in Rats; b \in Rats] \implies a - b \in Rats \)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-diff [symmetric])
done
lemma Rats-mult [simp]: \([a \in \text{Rats}; b \in \text{Rats}] \implies a \ast b \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-mult [symmetric])
done

lemma nonzero-Rats-inverse:
  fixes a :: 'a::{field-char-0, field-inverse-zero}
  shows \([a \in \text{Rats}; a \neq 0] \implies \text{inverse } a \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (erule nonzero-of-rat-inverse [symmetric])
done

lemma Rats-inverse [simp]:
  fixes a :: 'a::{field-char-0, field-inverse-zero}
  shows \([a \in \text{Rats}] \implies \text{inverse } a \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-inverse [symmetric])
done

lemma nonzero-Rats-divide:
  fixes a b :: 'a::{field-char-0}
  shows \([a \in \text{Rats}; b \in \text{Rats}; b \neq 0] \implies a / b \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (erule nonzero-of-rat-divide [symmetric])
done

lemma Rats-divide [simp]:
  fixes a b :: 'a::{field-char-0, field-inverse-zero}
  shows \([a \in \text{Rats}; b \in \text{Rats}] \implies a / b \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-divide [symmetric])
done

lemma Rats-power [simp]:
  fixes a :: 'a::{field-char-0}
  shows \(a \in \text{Rats} \implies a ^ n \in \text{Rats}\)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-power [symmetric])
done

lemma Rats-cases [cases set: Rats]:
  assumes \(q \in \mathbb{Q}\)
obtains \( (\text{of-rat}) \ r \) where \( q = \text{of-rat} \ r \)

proof –
from \( q \in \mathbb{Q} \) have \( q \in \text{range of-rat} \) unfolding \text{Rats-def}.
then obtain \( r \) where \( q = \text{of-rat} \ r \).
then show thesis.
qed

\text{lemma} \ \text{\texttt{Rats-induct}} [\text{case-names of-rat, induct set: Rats}]:
\qquad \quad q \in \mathbb{Q} \Rightarrow (\forall r. \ P \ (\text{of-rat} \ r)) \Rightarrow P \ q
by (\text{rule \texttt{Rats-cases}}) \text{ auto}

\text{lemma} \ \text{\texttt{Rats-infinite}}: \neg \text{finite} \ \mathbb{Q}
by (\text{auto dest!: finite-imageD simp: inj-on-def infinite-UNIV-char-0 \texttt{Rats-def}})

\section{Implementation of rational numbers as pairs of integers}

Formal constructor

\text{definition} \ \text{\texttt{Frct :: int \times int \Rightarrow rat where}} \quad 
[simp]: \texttt{Frct} \ p = \texttt{Fract} \ (\text{fst} \ p) \ (\text{snd} \ p)

\text{lemma} [\text{code abstype}]:
\qquad \texttt{Frct} \ (\text{quotient-of} \ q) = q
by (\text{cases} \ q) \ (\text{auto intro: quotient-of-eq})

Numerals

\text{declare} \ \texttt{quotient-of-Fract} [\text{code abstract}]

\text{definition} \ \text{\texttt{of-int :: int \Rightarrow rat where}} \quad 
where \quad \texttt{[code-abbrev]}: \texttt{of-int} = \texttt{Int.of-int}

\text{hide-const} (\text{open}) \ \texttt{of-int}

\text{lemma} \ \texttt{quotient-of-int} [\text{code abstract}]:
\qquad \texttt{quotient-of} \ (\texttt{Rat.of-int} \ a) = (a, 1)
by (\text{simp add: of-int-def of-int-rat quotient-of-Fract})

\text{lemma} [\text{code-unfold}]:
\qquad \texttt{numeral} \ k = \texttt{Rat.of-int} \ (\texttt{numeral} \ k)
by (\text{simp add: Rat.of-int-def})

\text{lemma} [\text{code-unfold}]:
\qquad \texttt{numeral} \ k = \texttt{Rat.of-int} \ (- \texttt{numeral} \ k)
by (\text{simp add: Rat.of-int-def})

\text{lemma} \ \texttt{Frct-code-post} [\text{code-post}]:
\qquad \texttt{Frct} \ (0, a) = 0
\qquad \texttt{Frct} \ (a, 0) = 0
\qquad \texttt{Frct} \ (1, 1) = 1
\qquad \texttt{Frct} \ (\texttt{numeral} \ k, 1) = \texttt{numeral} \ k
THEORY "Rat"

Frac\((1, \text{numeral } k) = 1 / \text{numeral } k\)
Frac\((\text{numeral } k, \text{numeral } l) = \text{numeral } k / \text{numeral } l\)
Frac\((- a, b) = - \text{Frac} (a, b)\)
Frac\((a, - b) = - \text{Frac} (a, b)\)
- \((- \text{Frac } q) = \text{Frac } q\)
by (simp-all add: Fract-of-int-quotient)

Operations

\textbf{lemma} \textit{rat-zero-code} [code abstract]:
\textit{quotient-of } 0 = (0, 1)
by (simp add: Zero-rat-def quotient-of-Fract normalize-def)

\textbf{lemma} \textit{rat-one-code} [code abstract]:
\textit{quotient-of } 1 = (1, 1)
by (simp add: One-rat-def quotient-of-Fract normalize-def)

\textbf{lemma} \textit{rat-plus-code} [code abstract]:
\textit{quotient-of } (p + q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q
in normalize \((a * d + b * c, c * d)\))
by (cases p, cases q) (simp add: quotient-of-Fract)

\textbf{lemma} \textit{rat-uminus-code} [code abstract]:
\textit{quotient-of } (- p) = (\text{let } (a, b) = \text{quotient-of } p \text{ in } (- a, b))
by (cases p) (simp add: quotient-of-Fract)

\textbf{lemma} \textit{rat-minus-code} [code abstract]:
\textit{quotient-of } (p - q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q
in normalize \((a * d - b * c, c * d)\))
by (cases p, cases q) (simp add: quotient-of-Fract)

\textbf{lemma} \textit{rat-times-code} [code abstract]:
\textit{quotient-of } (p * q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q
in normalize \((a * b, c * d)\))
by (cases p, cases q) (simp add: quotient-of-Fract)

\textbf{lemma} \textit{rat-inverse-code} [code abstract]:
\textit{quotient-of } (inverse p) = (\text{let } (a, b) = \text{quotient-of } p
in if a = 0 then (0, 1) else (sgn a * b, |a|))
proof (cases p)
case \text{Frac } a b then show ?thesis
by (cases 0::int a rule: linorder-cases) (simp-all add: quotient-of-Fract gcd-int.commute)
qed

\textbf{lemma} \textit{rat-divide-code} [code abstract]:
\textit{quotient-of } (p / q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q
in normalize \((a * d, c * b)\))
by (cases p, cases q) (simp add: quotient-of-Fract)

\textbf{lemma} \textit{rat-abs-code} [code abstract]:
quotient-of \( |p| = (\text{let } (a, b) = \text{quotient-of } p \text{ in } (|a|, b)) \) by (cases \( p \)) (simp add: quotient-of-Fract)

**lemma** rat-sgn-code [code abstract]:
\[
\text{quotient-of } (\text{sgn } p) = (\text{sgn } (\text{fst } \text{quotient-of } p)) \oplus 1
\]

**proof** (cases \( p \))
- case \( (\text{Fract } a \ b) \) then show \?thesis
  by (cases 0::int \( a \) rule: linorder-cases) (simp-all add: quotient-of-Fract)
qed

**lemma** rat-floor-code [code]:
\[
\text{floor } p = (\text{let } (a, b) = \text{quotient-of } p \text{ in } a \div b)
\]
by (cases \( p \)) (simp add: quotient-of-Fract floor-Fract)

**instantiation** \( \text{rat :: equal} \)
begin
**definition** [code]:
\[
\text{HOL.equal } a \ b \longleftrightarrow \text{quotient-of } a = \text{quotient-of } b
\]
instance proof qed (simp add: equal-rat-def quotient-of-inject-eq)

**lemma** rat-eq-refl [code nbe]:
\[
\text{HOL.equal } (r :: \text{rat}) \ r \longleftrightarrow \text{True}
\]
by (rule equal-refl)
end

**lemma** rat-less-eq-code [code]:
\[
p \leq q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a \cdot d \leq c \cdot b)
\]
by (cases \( p \), cases \( q \)) (simp add: quotient-of-Fract mult.commute)

**lemma** rat-less-code [code]:
\[
p < q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a \cdot d < c \cdot b)
\]
by (cases \( p \), cases \( q \)) (simp add: quotient-of-Fract mult.commute)

**lemma** [code]:
\[
of-rat p = (\text{let } (a, b) = \text{quotient-of } p \text{ in } \text{of-int } a \div \text{of-int } b)
\]
by (cases \( p \)) (simp add: quotient-of-Fract of-rat-rat)

Quickcheck

**definition** (in term-syntax)
\[
\text{valterm-fract} :: \text{int } \times (\text{unit } \Rightarrow \text{Code-Evaluation.term}) \Rightarrow \text{int } \times (\text{unit } \Rightarrow \text{Code-Evaluation.term}) \Rightarrow \text{rat } \times (\text{unit } \Rightarrow \text{Code-Evaluation.term}) \text{ where}
\]
\[
\text{[code-unfold]}: \text{valterm-fract } k \ l = \text{Code-Evaluation.valtermify Fract } \{\cdot\} \ k \ \{\cdot\} \ l
\]

**notation** \( fcomp \) (infix1 \( \circ \) \( >= \) 60)

**notation** \( scomp \) (infix1 \( \circ \rightarrow \) 60)
instantiation rat :: random
begin


instance ..
end

no-notation fcomp (infixl ◦> 60)
no-notation scomp (infixl ◦→ 60)

instantiation rat :: exhaustive
begin

definition exhaustive-rat f d = Quickcheck-Exhaustive.exhaustive (λl. Quickcheck-Exhaustive.exhaustive (λk. f (Fract k (int-of-integer (integer-of-natural l) + 1)))) d d

instance ..
end

instantiation rat :: full-exhaustive
begin

definition full-exhaustive-rat f d = Quickcheck-Exhaustive.full-exhaustive (%(l, -). Quickcheck-Exhaustive.full-exhaustive (%k. f (let j = int-of-integer (integer-of-natural l) + 1 in valterm-fract k (j, %-. Code-Evaluation.term-of j))) d) d

instance ..
end

instantiation rat :: partial-term-of
begin

instance ..
end
95.6 Setup for Nitpick

declaration ⟨⟨Nitpick-HOL.register-frac-type @{type-name rat}⟩⟩

lemmas [nitpick-unfold] = inverse-rat-inst.inverse-rat
one-rat-inst.one-rat ord-rat-inst.less-rat
ord-rat-inst.less-eq-rat plus-rat-inst.plus-rat times-rat-inst.times-rat
uminus-rat-inst.uminus-rat zero-rat-inst.zero-rat
95.7 Float syntax

Syntax -Float :: float-const ⇒ 'a

Parse-translation ⟨⟨
  let
    fun mk-number i =
      let
        fun mk 1 = Syntax.const @{const-syntax Num.One}
        | mk i =
          let
            val (q, r) = Integer.div-mod i 2;
            val bit = if r = 0 then @{const-syntax Num.Bit0} else @{const-syntax Num.Bit1};
            in Syntax.const bit $ (mk q) end;
        in
          if i = 0 then Syntax.const @{const-syntax Groups.zero}
        else if i > 0 then Syntax.const @{const-syntax Num.numeral} $ mk i
        else
          Syntax.const @{const-syntax Groups.uminus} $
          (Syntax.const @{const-syntax Num.numeral} $ mk (~ i))
        end;
    in Syntax.const bit $ (mk q) end;
    fun mk-frac str =
      let
        val {mant = i, exp = n} = Lexicon.read-float str;
        val exp = Syntax.const @{const-syntax Power.power};
        val ten = mk-number 10;
        val exp10 = if n = 1 then ten else exp $ ten $ mk-number n;
        in Syntax.const @{const-syntax divide} $ mk-number i $ exp10 end;
    in Syntax.const bit $ (mk q) end;
    fun float-tr [(c as Const (@{syntax-const -constrain}, -)) $ t $ u] = c $ float-tr [t] $ u
    | float-tr [t as Const (str, -)] = mk-frac str
    | float-tr ts = raise TERM (float-tr, ts);
    in
      [(@{syntax-const -Float}, K (float-tr)) end]
  ⟩⟩

Test:

Lemma 123.456 = \(-111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat)\)
  by simp

95.8 Hiding implementation details

Hide-const (open) normalize positive

Lifting-update rat.lifting
Lifting-forget rat.lifting

End
96  Conditionally-Complete-Lattices: Conditionally-complete Lattices

theory  Conditionally-Complete-Lattices
imports  Main
begin

lemma (in linorder) Sup-fin-eq-Max: finite X \implies X \neq \{} \implies Sup-fin X = Max X
by (induct X rule: finite-ne-induct) (simp-all add: sup-max)

lemma (in linorder) Inf-fin-eq-Min: finite X \implies X \neq \{} \implies Inf-fin X = Min X
by (induct X rule: finite-ne-induct) (simp-all add: inf-min)

context preorder
begin

definition bdd-above A \iff (\exists M. \forall x \in A. x \leq M)
definition bdd-below A \iff (\exists m. \forall x \in A. m \leq x)

lemma bdd-aboveI [intro]: (\forall x \in A \implies x \leq M) \implies bdd-above A
by (auto simp: bdd-above-def)

lemma bdd-belowI [intro]: (\forall x \in A \implies m \leq x) \implies bdd-below A
by (auto simp: bdd-below-def)

lemma bdd-aboveI2: (\forall x \in A \implies f x \leq M) \implies bdd-above (f' A)
by force

lemma bdd-belowI2: (\forall x \in A \implies m \leq f x) \implies bdd-below (f' A)
by force

lemma bdd-above-empty [simp, intro]: bdd-above \{}
unfolding bdd-above-def by auto

lemma bdd-below-empty [simp, intro]: bdd-below \{}
unfolding bdd-below-def by auto

lemma bdd-above-mono: bdd-above B \implies A \subseteq B \implies bdd-above A
by (metis (full-types) bdd-above-def order-class.le-neq-trans psubsetD)

lemma bdd-below-mono: bdd-below B \implies A \subseteq B \implies bdd-below A
by (metis bdd-below-def order-class.le-neq-trans psubsetD)

lemma bdd-above-Int1 [simp]: bdd-above A \implies bdd-above (A \cap B)
using bdd-above-mono by auto

lemma bdd-above-Int2 [simp]: bdd-above B \implies bdd-above (A \cap B)
using bdd-above-mono by auto
lemma bdd-below-Int1 [simp]: bdd-below A \implies bdd-below (A \cap B)
  using bdd-below-mono by auto

lemma bdd-below-Int2 [simp]: bdd-below B \implies bdd-below (A \cap B)
  using bdd-below-mono by auto

lemma bdd-above-Ioo [simp, intro]: bdd-above \{a <..< b\}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Ico [simp, intro]: bdd-above \{a ..< b\}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Ilo [simp, intro]: bdd-above \{..< b\}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Iic [simp, intro]: bdd-above \{a .. b\}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Iic [simp, intro]: bdd-above \{a .. b\}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Ico [simp, intro]: bdd-above \{a ..< b\}
  by (auto simp add: bdd-above-def intro!: exI[of - a] less-imp-le)

lemma bdd-above-Ilo [simp, intro]: bdd-above \{a <..< b\}
  by (auto simp add: bdd-above-def intro!: exI[of - a] less-imp-le)

lemma bdd-above-Ioi [simp, intro]: bdd-above \{a <.. b\}
  by (auto simp add: bdd-above-def intro!: exI[of - a] less-imp-le)

lemma bdd-above-Ici [simp, intro]: bdd-above \{a .. b\}
  by (auto simp add: bdd-above-def intro!: exI[of - a] less-imp-le)

end

lemma (in order-top) bdd-above-top[simp, intro!]: bdd-above A
  by (rule bdd-aboveI[of - top]) simp

lemma (in order-bot) bdd-above-bot[simp, intro!]: bdd-below A
by (rule bdd-belowI[of - bot]) simp

lemma bdd-above-uminus[simp]:
  fixes X :: 'a::ordered-ab-group-add set
  shows bdd-above (uminus ' X) ↔ bdd-below X
  by (auto simp: bdd-above-def bdd-below-def intro: le-imp-neg-le (metis le-imp-neg-le minus-minus))

lemma bdd-below-uminus[simp]:
  fixes X :: 'a::ordered-ab-group-add set
  shows bdd-below (uminus ' X) ↔ bdd-above X
  by (auto simp: bdd-above-def bdd-below-def intro: le-imp-neg-le (metis le-imp-neg-le minus-minus))

context lattice begin

lemma bdd-above-insert [simp]: bdd-above (insert a A) = bdd-above A
  by (auto simp: bdd-above-def intro: le-supI2 sup-ge1)

lemma bdd-below-insert [simp]: bdd-below (insert a A) = bdd-below A
  by (auto simp: bdd-below-def intro: le-infI2 inf-le1)

lemma bdd-finite [simp]:
  assumes finite A
  shows bdd-above-finite: bdd-above A and bdd-below-finite: bdd-below A
  using assms by (induct rule: finite-induct, auto)

lemma bdd-above-Un [simp]: bdd-above (A ∪ B) = (bdd-above A ∧ bdd-above B)
proof
  assume bdd-above (A ∪ B)
  thus bdd-above A ∧ bdd-above B unfolding bdd-above-def by auto
next
  assume bdd-above A ∧ bdd-above B
  then obtain a b where ∀ x∈A. x ≤ a ∀ x∈B. x ≤ b unfolding bdd-above-def by auto
  hence ∀ x ∈ A ∪ B. x ≤ sup a b by (auto intro: Un-iff le-supI1 le-supI2)
  thus bdd-above (A ∪ B) unfolding bdd-above-def ..
  qed

lemma bdd-below-Un [simp]: bdd-below (A ∪ B) = (bdd-below A ∧ bdd-below B)
proof
  assume bdd-below (A ∪ B)
  thus bdd-below A ∧ bdd-below B unfolding bdd-below-def by auto
next
  assume bdd-below A ∧ bdd-below B
  then obtain a b where ∀ x∈A. a ≤ x ∧ ∀ x∈B. b ≤ x unfolding bdd-below-def by auto
  hence ∀ x ∈ A ∪ B. inf a b ≤ x by (auto intro: Un-iff le-infI1 le-infI2)
thus \texttt{bdd-below} \((A \cup B)\) \texttt{unfolding} \texttt{bdd-below-def} ..

qed

\textbf{lemma} \texttt{bdd-above-sup}\([\texttt{simp}]: \texttt{bdd-above} ((\lambda x. \texttt{sup} (f x) (g x)) \ ('A)) \iff \texttt{bdd-above} (f' A) \land \texttt{bdd-above} (g' A)

by (auto \texttt{simp} \(\texttt{bdd-above-def intro: le-supI1 le-supI2})

\textbf{lemma} \texttt{bdd-below-inf}\([\texttt{simp}]: \texttt{bdd-below} ((\lambda x. \texttt{inf} (f x) (g x)) \ ('A)) \iff \texttt{bdd-below} (f' A) \land \texttt{bdd-below} (g' A)

by (auto \texttt{simp} \(\texttt{bdd-below-def intro: le-infI1 le-infI2})

end

To avoid name classes with the \texttt{complete-lattice}-class we prefix \texttt{Sup} and \texttt{Inf} in theorem names with \(c\).

\textbf{class} \texttt{conditionally-complete-lattice} = \texttt{lattice} + \texttt{Sup} + \texttt{Inf} +

\texttt{assumes } \texttt{cInf-lower}: \(x \in X = \rightarrow \texttt{bdd-below} X = \rightarrow \texttt{Inf} X \leq x\)

\texttt{and } \texttt{cInf-greatest}: \(X \neq \emptyset = \rightarrow (\forall x. x \in X = \rightarrow z \leq x) = \rightarrow z \leq \texttt{Inf} X\)

\texttt{assumes } \texttt{cSup-upper}: \(x \in X = \rightarrow \texttt{bdd-above} X = \rightarrow x \leq \texttt{Sup} X\)

\texttt{and } \texttt{cSup-least}: \(X \neq \emptyset = \rightarrow (\forall x. x \in X = \rightarrow x \leq z) = \rightarrow \texttt{Sup} X \leq z\)

begin

\textbf{lemma} \texttt{cSup-upper2}: \(x \in X = \rightarrow y \leq x = \rightarrow \texttt{bdd-above} X = \rightarrow y \leq \texttt{Sup} X\)

by (metis \texttt{cSup-upper} \texttt{order-trans})

\textbf{lemma} \texttt{cInf-lower2}: \(x \in X = \rightarrow x \leq y = \rightarrow \texttt{bdd-below} X = \rightarrow \texttt{Inf} X \leq y\)

by (metis \texttt{cInf-lower} \texttt{order-trans})

\textbf{lemma} \texttt{cSup-mono}: \(B \neq \emptyset = \rightarrow \texttt{bdd-above} A = \rightarrow (\forall b. b \in B = \rightarrow \exists a \in A. b \leq a) = \rightarrow \texttt{Sup} B \leq \texttt{Sup} A\)

by (metis \texttt{cSup-least} \texttt{cSup-upper2})

\textbf{lemma} \texttt{cInf-mono}: \(B \neq \emptyset = \rightarrow \texttt{bdd-below} A = \rightarrow (\forall b. b \in B = \rightarrow \exists a \in A. a \leq b) = \rightarrow \texttt{Inf} A \leq \texttt{Inf} B\)

by (metis \texttt{cInf-greatest} \texttt{cInf-lower2})

\textbf{lemma} \texttt{cSup-subset-mono}: \(A \neq \emptyset = \rightarrow \texttt{bdd-above} B = \rightarrow A \subseteq B = \rightarrow \texttt{Sup} A \leq \texttt{Sup} B\)

by (metis \texttt{cSup-least} \texttt{cSup-upper subsetD})

\textbf{lemma} \texttt{cInf-superset-mono}: \(A \neq \emptyset = \rightarrow \texttt{bdd-below} B = \rightarrow A \subseteq B = \rightarrow \texttt{Inf} B \leq \texttt{Inf} A\)

by (metis \texttt{cInf-greatest} \texttt{cInf-lower subsetD})

\textbf{lemma} \texttt{cSup-eq-maximum}: \(z \in X = \rightarrow (\forall x. x \in X = \rightarrow x \leq z) = \rightarrow \texttt{Sup} X = z\)

by (intro antisym \texttt{cSup-upper}[of \(z X\)] \texttt{cSup-least}[of \(X z\)]) \texttt{auto}

\textbf{lemma} \texttt{cInf-eq-minimum}: \(z \in X = \rightarrow (\forall x. x \in X = \rightarrow z \leq x) = \rightarrow \texttt{Inf} X = z\)

by (intro antisym \texttt{cInf-lower}[of \(z X\)] \texttt{cInf-greatest}[of \(X z\)]) \texttt{auto}
lemma cSup-le-iff: \( S \neq \{ \} \implies \text{bdd-above } S \implies \sup S \leq a \iff (\forall x \in S. \ x \leq a) \)
by (metis order-trans cSup-upper cSup-least)

lemma le-cInf-iff: \( S \neq \{ \} \implies \text{bdd-below } S \implies a \leq \inf S \iff (\forall x \in S. \ a \leq x) \)
by (metis order-trans cInf-lower cInf-greatest)

lemma cSup-eq-non-empty:
assumes 1: \( X \neq \{ \} \)
assumes 2: \( \forall x. \ x \in X \implies x \leq a \)
assumes 3: \( \forall y. \ (\forall x \in X \implies x \leq y) \implies y \leq a \)
shows \( \sup X = a \)
by (intro 3 1 antisym cSup-least) (auto intro: 2 1 cSup-upper)

lemma cInf-eq-non-empty:
assumes 1: \( X \neq \{ \} \)
assumes 2: \( \forall x. \ x \in X \implies a \leq x \)
assumes 3: \( \forall y. \ (\forall x \in X \implies y \leq x) \implies y \leq a \)
shows \( \inf X = a \)
by (intro 3 1 antisym cInf-greatest) (auto intro: 2 1 cInf-lower)

lemma cInf-cSup:
assumes \( S \neq \{ \} \implies \text{bdd-below } S \implies \inf S = \sup \{ x. \ \forall s \in S. \ x \leq s \} \)
by (rule cInf-eq-non-empty) (auto intro!: cSup-upper cSup-least simp: bdd-below-def)

lemma cSup-cInf:
assumes \( S \neq \{ \} \implies \text{bdd-above } S \implies \sup S = \inf \{ x. \ \forall s \in S. \ s \leq x \} \)
by (rule cSup-eq-non-empty) (auto intro!: cInf-lower cInf-greatest simp: bdd-above-def)

lemma cSup-insert: \( X \neq \{ \} \implies \text{bdd-above } X \implies \sup (\text{insert } a X) = \sup a \ (\sup X) \)
by (intro cSup-eq-non-empty) (auto intro: le-supI2 cSup-upper cSup-least)

lemma cInf-insert: \( X \neq \{ \} \implies \text{bdd-below } X \implies \inf (\text{insert } a X) = \inf a \ (\inf X) \)
by (intro cInf-eq-non-empty) (auto intro: le-infI2 cInf-lower cInf-greatest)

lemma cSup-singleton [simp]: \( \sup \{ x \} = x \)
by (intro cSup-eq-maximum) auto

lemma cInf-singleton [simp]: \( \inf \{ x \} = x \)
by (intro cInf-eq-minimum) auto

lemma cSup-insert-If: \( \text{bdd-above } X \implies \sup (\text{insert } a X) = (\text{if } X = \{ \} \text{ then } a \text{ else } \sup a \ (\sup X)) \)
using cSup-insert[of X] by simp

lemma cInf-insert-If: \( \text{bdd-below } X \implies \inf (\text{insert } a X) = (\text{if } X = \{ \} \text{ then } a \text{ else } \inf a \ (\inf X)) \)
using cInf-insert[of X] by simp
**THEORY** “Conditionally-Complete-Lattices”

**lemma** le-cSup-finite: finite X \implies x \in X \implies x \leq \Sup X

**proof** (induct X arbitrary: x rule: finite-induct)
- case (insert x X y) then show ?case
  - by (cases X = {}) (auto simp: cSup-insert intro: le-supI2)

**qed simp**

**lemma** cInf-le-finite: finite X \implies x \in X \implies \Inf X \leq x

**proof** (induct X arbitrary: x rule: finite-induct)
- case (insert x X y) then show ?case
  - by (cases X = {}) (auto simp: cInf-insert intro: le-infI2)

**qed simp**

**lemma** cSup-eq-Sup-fin: finite X \neq {} \implies \Sup X = \Sup-fin X

**by** (induct X rule: finite-ne-induct) (simp-all add: cSup-insert)

**lemma** cInf-eq-Inf-fin: finite X \neq {} \implies \Inf X = \Inf-fin X

**by** (induct X rule: finite-ne-induct) (simp-all add: cInf-insert)

**lemma** cSup-atMost[simp]: Sup {..x} = x

**by** (auto intro!: cSup-eq-maximum)

**lemma** cSup-greaterThanAtMost[simp]: y < x \implies Sup {y..<x} = x

**by** (auto intro!: cSup-eq-maximum)

**lemma** cSup-atLeastAtMost[simp]: y \leq x \implies Sup {y..x} = x

**by** (auto intro!: cSup-eq-maximum)

**lemma** cINF-lower: bdd-below (f ' A) \implies x \in A \implies INFIMUM A f \leq f x

**using** cInf-lower [of - f ' A] by simp

**lemma** cINF-greatest: A \neq {} \implies (\forall x. x \in A \implies m \leq f x) \implies m \leq INFIMUM A f

**using** cInf-greatest [of f ' A] by auto

**lemma** cSUP-upper: x \in A \implies bdd-above (f ' A) \implies f x \leq SUPREMUM A f

**using** cSup-upper [of - f ' A] by simp

**lemma** cSUP-least: A \neq {} \implies (\forall x. x \in A \implies f x \leq M) \implies SUPREMUM A f \leq M

**using** cSup-least [of f ' A] by auto
**THEORY** “Conditionally-Complete-Lattices”

---

**lemma** cINF-lower2: bdd-below \((f^A)\) \(\Rightarrow \) \(x \in A \Rightarrow f x \leq u \Rightarrow \) \(\) INFIMUM \(A f\)
\(f \leq u\)
by (auto intro: cINF-lower assms order-trans)

**lemma** cSUP-upper2: bdd-above \((f^A)\) \(\Rightarrow \) \(x \in A \Rightarrow u \leq f x \Rightarrow u \leq \) SUPREMUM \(A f\)
by (auto intro: cSUP-upper assms order-trans)

**lemma** cSUP-const: \(A \neq \{\} \Rightarrow (SUP x:A. c) = c\)
by (intro antisym cSUP-least) (auto intro: cSUP-upper assms order-trans)

**lemma** cINF-const: \(A \neq \{\} \Rightarrow (INF x:A. c) = c\)
by (intro antisym cINF-greatest) (auto intro: cINF-lower assms order-trans)

**lemma** le-cINF-iff: \(A \neq \{\} \Rightarrow bdd-below (f^A) \Rightarrow u \leq \) INFIMUM \(A f\) \(\iff\)
\(\forall x \in A. u \leq f x\)
by (metis cINF-greatest cINF-lower assms order-trans)

**lemma** cSUP-le-iff: \(A \neq \{\} \Rightarrow bdd-above (f^A) \Rightarrow SUPREMUM A f \leq u\) \(\iff\)
\(\forall x \in A. f x \leq u\)
by (metis cSUP-least cSUP-upper assms order-trans)

**lemma** less-cINF-D: \(bdd-below (f^A) \Rightarrow y < (INF i:A. f i) \Rightarrow i \in A \Rightarrow y < f i\)
by (metis cINF-lower less-le-trans)

**lemma** cSUP-lessD: \(bdd-above (f^A) \Rightarrow (SUP i:A. f i) < y \Rightarrow i \in A \Rightarrow f i < y\)
by (metis cSUP-upper le-less-trans)

**lemma** cINF-insert: \(A \neq \{\} \Rightarrow bdd-below (f^A) \Rightarrow \) INFIMUM \((\) insert \(a\) \((A f)\) \(\) \(\) = inf \((f a)\) (INFIMUM \(A f)\)
by (metis cINF-insert Inf-image-eq image-insert image-is-empty)

**lemma** cSUP-insert: \(A \neq \{\} \Rightarrow bdd-above (f^A) \Rightarrow \) SUPREMUM \((\) insert \(a\) \((A f)\) \(\) \(\) = sup \((f a)\) (SUPREMUM \(A f)\)
by (metis cSUP-insert Sup-image-eq image-insert image-is-empty)

**lemma** cINF-mono: \(B \neq \{\} \Rightarrow bdd-below (f^A) \Rightarrow (\forall m. m \in B \Rightarrow \exists n \in A. f n \leq g m) \Rightarrow \) INFIMUM \((\) \(A f) \leq \) INFIMUM \((\) \(B g)\)
using cINF-monotonicity [of g f B] by auto

**lemma** cSUP-mono: \(A \neq \{\} \Rightarrow bdd-above (g^B) \Rightarrow (\forall n. n \in A \Rightarrow \exists m \in B. f n \leq g m) \Rightarrow \) SUPREMUM \((\) \(A f) \leq \) SUPREMUM \((\) \(B g)\)
using cSUP-monotonicity [of f g A B] by auto

**lemma** cINF-superset-mono: \(A \neq \{\} \Rightarrow bdd-below (g^B) \Rightarrow A \subseteq B \Rightarrow (\forall x. x \in B \Rightarrow g x \leq f x) \Rightarrow \) INFIMUM \((\) \(B g) \leq \) INFIMUM \((\) \(A f)\)
theory "Conditionally-Complete-Lattices" 1388

by (rule \textit{cInf-mono}) auto

\textbf{lemma} \textit{cSUP-subset-mono}: \(A \neq \{\} \Rightarrow \text{bdd-above } (g \cdot B) \Rightarrow A \subseteq B \Rightarrow (\forall x. x \in B \Rightarrow f x \leq g x) \Rightarrow \text{SUPREMUM } A f \leq \text{SUPREMUM } B g\)
by (rule \textit{cSUP-mono}) auto

\textbf{lemma} \textit{less-eq-cInf-inter}: \(\text{bdd-below } A \Rightarrow \text{bdd-below } B \Rightarrow A \cap B \neq \{\} \Rightarrow \inf \(\lnf A\) (\lnf B) \leq \lnf (A \cap B)\)
by (metis \textit{cInf-superset-mono} lattice-class.inf-sup-ord(1) le-infI1)

\textbf{lemma} \textit{cSup-inter-less-eq}: \(\text{bdd-above } A \Rightarrow \text{bdd-above } B \Rightarrow A \cap B \neq \{\} \Rightarrow \text{Sup } (A \cap B) \leq \text{sup } (\text{Sup } A) (\text{Sup } B)\)
by (metis \textit{cSup-subset-mono} lattice-class.inf-sup-ord(1) le-supI1)

\textbf{lemma} \textit{cInf-union-distrib}: \(A \neq \{\} \Rightarrow \text{bdd-below } A \Rightarrow B \neq \{\} \Rightarrow \text{bdd-below } B \Rightarrow \lnf (A \cup B) = \lnf (\lnf A) (\lnf B)\)
by (rule cSup-union-distrib \[\text{of } f \cdot A f \cdot B\] by simp add: image-Un \[\text{symmetric}\])

\textbf{lemma} \textit{cSup-union}: \(A \neq \{\} \Rightarrow \text{bdd-above } A \Rightarrow B \neq \{\} \Rightarrow \text{bdd-above } B \Rightarrow \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)\)
by (intro antisym le-supI \textit{cSup-least} \textit{cSup-upper}) (auto intro: le-supI1 le-supI2 cSup-upper)

\textbf{lemma} \textit{cSUP-union}: \(A \neq \{\} \Rightarrow \text{bdd-above } (f' A) \Rightarrow B \neq \{\} \Rightarrow \text{bdd-above } (f'B) \Rightarrow \text{SUPREMUM } (A \cup B) f = \text{sup } (\text{SUPREMUM } A f) (\text{SUPREMUM } B f)\)
using \textit{cSup-union-distrib} \[\text{of } f \cdot A f \cdot B\] by (simp add: image-Un \[\text{symmetric}\])

\textbf{lemma} \textit{cINF-inf-distrib}: \(A \neq \{\} \Rightarrow \text{bdd-below } (fA) \Rightarrow \text{bdd-below } (g'A) \Rightarrow \inf \(\inf \text{INFIMUM } A f) (\text{INFIMUM } A g) = (\inf a: A. \inf (f a) (g a))\)
by (intro antisym le-infI \textit{cINF-greatest} \textit{cINF-lower2})
(auto intro: le-infI1 le-infI2 cINF-greatest \textit{cINF-lower} le-infI)

\textbf{lemma} \textit{SUP-sup-distrib}: \(A \neq \{\} \Rightarrow \text{bdd-above } (fA) \Rightarrow \text{bdd-above } (g'A) \Rightarrow \sup \(\text{SUPREMUM } A f) (\text{SUPREMUM } A g) = (\sup a: A. \sup (f a) (g a))\)
by (intro antisym le-supI \textit{cSUP-least} \textit{cSUP-upper2})
(auto intro: le-supI1 le-supI2 cSUP-least cSUP-upper le-supI)

\textbf{lemma} \textit{cInf-le-cSup}: \(A \neq \{\} \Rightarrow \text{bdd-above } A \Rightarrow \text{bdd-below } A \Rightarrow \text{Inf } A \leq \text{Sup } A\)
by (auto intro!: \textit{cSup-upper2} \[\text{of } \text{SOME } a. a \in A\] intro: someI \textit{cInf-lower})

end

\textbf{instance} \textit{complete-lattice} \(\subseteq\) \textit{conditionally-complete-lattice}
by default (auto intro: Sup-upper Sup-least Inf-lower Inf-greatest)

lemma cSup-eq:
  fixes a :: 'a :: {conditionally-complete-lattice, no-bot}
  assumes upper: \( \forall x. \ x \in X \implies x \leq a \)
  assumes least: \( \forall y. (\forall x. \ x \in X \implies x \leq y) \implies a \leq y \)
  shows Sup X = a
proof cases
  assume X = {} with lt-ex [of a] least ?thesis by (auto simp: less-le-not-le)
qed (intro cSup-eq-non-empty assms)

lemma cInf-eq:
  fixes a :: 'a :: {conditionally-complete-lattice, no-top}
  assumes upper: \( \forall x. \ x \in X \implies a \leq x \)
  assumes least: \( \forall y. (\forall x. \ x \in X \implies y \leq x) \implies y \leq a \)
  shows Inf X = a
proof cases
  assume X = {} with gt-ex [of a] least ?thesis by (auto simp: less-le-not-le)
qed (intro cInf-eq-non-empty assms)

class conditionally-complete-linorder = conditionally-complete-lattice + linorder
begin

lemma less-cSup_iff :
  \( X \neq \{} \implies \text{bdd-above } X \implies y < \text{Sup } X \iff (\exists x \in X. \ y < x) \)
  by (rule iffI) (metis cSup-least assms not-le that)

lemma cInf_less_iff: \( X \neq \{} \implies \text{bdd-below } X \implies \text{Inf } X < y \iff (\exists x \in X. \ x < y) \)
  by (rule iffI) (metis cInf-greatest not-less, metis cSup-upper less-le-trans)

lemma cINF_less_iff: \( A \neq \{} \implies \text{bdd-below } (f\ A) \implies (\text{INF } i:A. \ f \ i) < a \iff (\exists x \in A. \ f \ x < a) \)
  using cInf_less_iff [of f\ A] by auto

lemma less-cSUP_iff: \( A \neq \{} \implies \text{bdd-above } (f\ A) \implies a < (\text{SUP } i:A. \ f \ i) \iff (\exists x \in A. \ a < f \ x) \)
  using less-cSupE [of f\ A] by auto

lemma less-cSupE:
  assumes \( y < \text{Sup } X \neq \{} \) obtains x where \( x \in X \ \ y < x \)
  by (metis cSup-least assms not-le that)

lemma less-cSupD:
  \( X \neq \{} \implies z < \text{Sup } X \implies (\exists x \in X. \ z < x) \)
  by (metis less-cSupE not-leE bdd-above-def)

lemma cInf-lessD:
  \( X \neq \{} \implies \text{Inf } X < z \implies (\exists x \in X. \ x < z) \)
THEORY "Conditionally-Complete-Lattices"

by (metis cInf-less-iff not-leE bdd-below-def)

lemma complete-interval:
  assumes a < b and P a and ¬ P b
  shows ∃ c. a ≤ c ∧ c ≤ b ∧ (∀ x. a ≤ x ∧ x < c −→ P x) ∧
  (∀ d. (∀ x. a ≤ x ∧ x < d −→ P x) −→ d ≤ c)
proof (rule exI [where x = Sup {d. ∀ c. a ≤ c ∧ c < d −→ P c}], auto)
  show a ≤ Sup {d. ∀ c. a ≤ c ∧ c < d −→ P c}
    by (rule cSup-upper, auto simp: bdd-above-def)
  next
    show Sup {d. ∀ c. a ≤ c ∧ c < d −→ P c} ≤ b
      apply (rule cSup-least)
      apply auto
      apply (metis less-le-not-le)
      done
next
  fix x
  assume x: a ≤ x and lt: x < Sup {d. ∀ c. a ≤ c ∧ c < d −→ P c}
  show P x
    apply (rule less-cSupE [OF lt], auto)
    apply (metis less-le-not-le)
    done
next
  fix d
  assume 0: ∀ x. a ≤ x ∧ x < d −→ P x
  thus d ≤ Sup {d. ∀ c. a ≤ c ∧ c < d −→ P c}
    by (rule-tac cSup-upper, auto simp: bdd-above-def)
  (metis (a < b) (¬ P b) linear less-le)
qed

end

lemma cSup-eq-Max: finite (X::'a::conditionally-complete-linorder set) ⇒ X ≠ ∅ ⇒ Sup X = Max X
using cSup-eq-Sup-fin[of X] Sup-fin-eq-Max[of X] by simp

lemma cInf-eq-Min: finite (X::'a::conditionally-complete-linorder set) ⇒ X ≠ ∅ ⇒ Inf X = Min X
using cInf-eq-Inf-fin[of X] Inf-fin-eq-Min[of X] by simp

lemma cSup-lessThan[simp]: Sup {..<x::'a::conditionally-complete-linorder, no-bot, dense-linorder} = x
  by (auto intro!: cSup-eq-non-empty intro: dense-le)

lemma cSup-greaterThanLessThan[simp]: y < x ⇒ Sup {y..<..<x::'a::conditionally-complete-linorder, dense-linorder} = x
THEORY "Conditionally-Complete-Lattices"

by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)

lemma cSup-atLeastLessThan[simp]: \( y < x \implies \operatorname{Sup}\{y..<x\::\{\text{conditionally-complete-linorder, dense-linorder}\}\} = x \)
by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)

lemma cInf-greaterThan[simp]: \( \operatorname{Inf}\{x::\{\text{conditionally-complete-linorder, no-top, dense-linorder}\}\}<..\} = x \)
by (auto intro!: cInf-eq-non-empty intro: dense-ge)

lemma cInf-greaterThanAtMost[simp]: \( y < x \implies \operatorname{Inf}\{y..<x\::\{\text{conditionally-complete-linorder, dense-linorder}\}\} = y \)
by (auto intro!: cInf-eq-non-empty intro: dense-ge-bounded)

lemma cInf-greaterThanLessThan[simp]: \( y < x \implies \operatorname{Inf}\{y..<..<x\::\{\text{conditionally-complete-linorder, dense-linorder}\}\} = y \)
by (auto intro!: cInf-eq-non-empty intro: dense-ge-bounded)

class linear-continuum = conditionally-complete-linorder + dense-linorder +
assumes UNIV-not-singleton: \( \exists a b::a \neq b \)
begin

lemma ex-gt-or-lt: \( \exists b. a < b \lor b < a \)
by (metis UNIV-not-singleton neq-iff)

end

instantiation nat :: conditionally-complete-linorder
begin

definition Sup (X::nat set) = Max X
definition Inf (X::nat set) = (LEAST n. n \in X)

lemma bdd-above-nat: \( \text{bdd-above } X \iff \text{finite } (X::nat set) \)
proof
assume \( \text{bdd-above } X \)
then obtain z where \( X \subseteq \{.. z\}\)
  by (auto simp: bdd-above-def)
then show \( \text{finite } X \)
  by (rule finite-subset simp)
qed simp

instance
proof
  fix x :: nat and X :: nat set
  { assume x \in X bdd-below X then show Inf X \leq x
    by (simp add: Inf-nat-def Least-le)
  }
  { assume X \not= {} \\\ \\ (\\ \ \ y \in X \implies x \leq y then show x \leq Inf X
    unfolding Inf-nat-def ex-in-cone[symmetric] by (rule LeastI2-ex)
  }

end
\begin{verbatim}
{ assume \( x \in X \) \text{bdd-above} \( X \) then show \( x \leq \text{Sup} \ X \)
  by (simp add: \text{Sup-nat-def} \text{bdd-above-nat}) }
{ assume \( X \neq {} \) \text{bdd-above} \( X \)
  moreover then have \text{bdd-above} \( X \)
  by (auto simp: \text{bdd-above-def})
  ultimately show \( \text{Sup} \ X \leq x \)
  by (simp add: \text{Sup-nat-def} \text{bdd-above-nat}) }
qed

\end{verbatim}
then have \( \operatorname{Sup} X \in X \land (\forall y \in X. y \leq \operatorname{Sup} X) \)
unfolding \( \operatorname{Sup-int-def} \) by (rule theI')

note \( \operatorname{Sup-int} = \) this

{ fix \( x :: \text{int} \) and \( X :: \text{int set} \)
  assume \( x \in X \quad \text{bdd-above} \quad \text{X} \)
  then show \( x \leq \operatorname{Sup} X \)
  using \( \operatorname{Sup-int[of X]} \) by auto }

note \( \text{le-Sup} = \) this

{ fix \( x :: \text{int} \) and \( X :: \text{int set} \)
  assume \( X \neq \{\} \land y. y \in X \implies y \leq x \)
  then show \( \operatorname{Sup} X \leq x \)
  using \( \operatorname{Sup-int[of X]} \) by (auto simp: \text{bdd-above-def}) }

note \( \text{Sup-le} = \) this

{ fix \( x :: \text{int} \) and \( X :: \text{int set} \)
  assume \( x \in X \quad \text{bdd-below} \quad \text{X} \)
  then show \( x \leq \operatorname{Inf} X \)
  using \( \text{le-Sup[of uminus ' X]} \) by (force simp: \text{Inf-int-def}) }

\text{lemma interval-cases:}

\text{fixes} \( S :: 'a :: \text{conditionally-complete-linorder set} \)
\text{assumes} \( \text{ivl:} \quad \land x. a \in S \implies b \in S \quad a \leq x \quad x \leq b \implies x \in S \)
\text{shows} \( \exists a. b. S = \{\} \lor S = \text{UNIV} \lor S = \{..b\} \lor S = \{..\} \lor S = \{a..<b\} \lor S = \{a..<\} \lor S = \{a..<b\} \lor S = \{a..<\} \lor S = \{a..b\} \)
\text{proof}

\text{def} \( \text{lower} \equiv \{x. \exists s \in S. s \leq x\} \quad \text{and} \quad \text{upper} \equiv \{x. \exists s \in S. x \leq s\} \)
\text{with} \( \text{ivl have} \quad S = \text{lower} \cap \text{upper} \quad \text{by auto} \)

\text{moreover have} \( \exists a. \text{upper} = \text{UNIV} \lor \text{upper} = \{\} \lor \text{upper} = \{..a\} \lor \text{upper} = \{..<a\} \)

\text{proof cases}

\text{assume} \( *: \text{bdd-above} \quad S \quad \text{S} \neq \{\} \)
\text{from} \( * \) have \( \text{upper} \subseteq \{.. \text{Sup} S\} \)
\text{by (auto simp: upper-def intro: cSup-upper2)}

\text{moreover from} \( * \) have \( \{..< \text{Sup} S\} \subseteq \text{upper} \)
\text{by (force simp add: less-cSup-iff upper-def subset-eq Ball-def)}

\text{ultimately have} \( \text{upper} = \{.. \text{Sup} S\} \lor \text{upper} = \{..< \text{Sup} S\} \)

\text{unfolding} \( \text{ivl-disj-un}[2] \) by auto
then show \( ?\text{thesis} \) by auto

next
assume \( \neg (\text{bdd-above } S \land S \neq \{\}) \)
then have upper = UNIV \lor upper = \{\}
  by (auto simp: upper-def bdd-above-def not-le dest: less-imp-le)
then show \( ?\text{thesis} \) by auto
qed

moreover have \( \exists b. \text{lower} = \text{UNIV} \lor \text{lower} = \{\} \lor \text{lower} = \{b \ldots\} \lor \text{lower} = \{b < \ldots\} \)
proof cases
assume \( \ast : \text{bdd-below } S \land S \neq \{\} \)
from \( \ast \) have lower \( \subseteq \{\Inf S \ldots\} \)
  by (auto simp: lower-def intro: cInf-lower2)
moreover from \( \ast \) have \( \{\Inf S \ldots\} \subseteq \text{lower} \)
  by (force simp add: cInf-less-iff lower-def subset-eq Ball-def)
ultimately have lower = \( \{\Inf S \ldots\} \lor \text{lower} = \{\Inf S \ldots\} \)
  unfolding iel-disj-un(I)[symmetric] by auto
then show \( ?\text{thesis} \) by auto
next
assume \( \neg (\text{bdd-below } S \land S \neq \{\}) \)
then have lower = UNIV \lor lower = \{\}
  by (auto simp: lower-def bdd-below-def not-le dest: less-imp-le)
then show \( ?\text{thesis} \) by auto
qed

ultimately show \( ?\text{thesis} \)
unfolding greaterThanAtMost-def greaterThanLessThan-def atLeastAtMost-def
atLeastLessThan-def
  by (elim exE disjE) auto
qed

end

97 Real: Development of the Reals using Cauchy Sequences

theory Real
imports Rat Conditionally-Complete-Lattices
begin

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See 
``~\( \text{src/HOL/ex/Dedekind_Real.thy} \)`` for an alternative construction using Dedekind cuts.

97.1 Preliminary lemmas

lemma add-diff-add:
fixes $a$ $b$ $c$ $d$ :: 'a::ab-group-add
shows $(a + c) - (b + d) = (a - b) + (c - d)$
by simp

lemma minus-diff-minus:
fixes $a$ $b$ :: 'a::group-add
shows $-a - -b = -(a - b)$
by simp

lemma mult-diff-mult:
fixes $x$ $y$ $a$ $b$ :: 'a::ring
shows $(x * y - a * b) = x * (y - b) + (x - a) * b$
by (simp add: algebra-simps)

lemma inverse-diff-inverse:
fixes $a$ $b$ :: 'a::division-ring
assumes $a \neq 0$ and $b \neq 0$
shows $inverse a - inverse b = -(inverse a * (a - b) * inverse b)$
using assms by (simp add: algebra-simps)

lemma obtain-pos-sum:
fixes $r$ :: 'a::rat
assumes $r > 0$
obtains $s$ $t$ where $0 < s$ and $0 < t$ and $r = s + t$
proof
  from $r$ show $0 < r/2$ by simp
  from $r$ show $0 < r/2$ by simp
  show $r = r/2 + r/2$ by simp
qed

97.2 Sequences that converge to zero

definition vanishes :: (nat ⇒ 'a::real) ⇒ bool
where
vanishes $X = (\forall r > 0. \exists k. \forall n \geq k. |X n| < r)$

lemma vanishesI: $\forall r > 0. \Rightarrow \exists k. \forall n \geq k. |X n| < r)$ ⇒ vanishes $X$
unfolding vanishes-def by simp

lemma vanishesD: [vanishes $X; 0 < r$] ⇒ \exists k. \forall n \geq k. |X n| < r
unfolding vanishes-def by simp

lemma vanishes-const [simp]: vanishes $(\lambda n. c) \iff c = 0$
unfolding vanishes-def
apply (cases $c = 0$, auto)
apply (rule exI [where $x=|c|$, auto])
done

lemma vanishes-minus: vanishes $X$ ⇒ vanishes $(\lambda n. - X n)$
unfolding \texttt{vanishes-def} \texttt{by simp}

\textbf{lemma \texttt{vanishes-add}:}
\begin{itemize}
\item \textbf{assumes} \texttt{X}: \texttt{vanishes X} \texttt{and Y}: \texttt{vanishes Y}
\item \textbf{shows} \texttt{vanishes (\lambda n. X n + Y n)}
\end{itemize}
\textbf{proof} \texttt{(rule \texttt{vanishesI})}
\begin{itemize}
\item \texttt{fix r :: rat assume 0 < r}
\item \texttt{then obtain s t where s: 0 < s and t: 0 < t and r = s + t}
\item \texttt{by (rule \texttt{obtain-pos-sum})}
\item \texttt{obtain i where i: \forall n \geq i. |X n| < s}
\item \texttt{using \texttt{vanishesD [OF X s] ..}}
\item \texttt{obtain j where j: \forall n \geq j. |Y n| < t}
\item \texttt{using \texttt{vanishesD [OF Y t] ..}}
\item \texttt{have \forall n \geq max i j. |X n + Y n| < r}
\item \texttt{proof (clarsimp)}
\item \texttt{fix n assume n: i \leq n j \leq n}
\item \texttt{have |X n + Y n| \leq |X n| + |Y n| by (rule \texttt{abs-triangle-ineq})}
\item \texttt{also have \ldots < s + t by (simp add: \texttt{add-strict-mono i j n})}
\item \texttt{finally show |X n + Y n| < r unfolding r .}
\item \texttt{qed}
\item \texttt{thus \exists k. \forall n \geq k. |X n + Y n| < r ..}
\item \texttt{qed}
\end{itemize}

\textbf{lemma \texttt{vanishes-diff}:}
\begin{itemize}
\item \textbf{assumes} \texttt{X}: \texttt{vanishes X} \texttt{and Y}: \texttt{vanishes Y}
\item \textbf{shows} \texttt{vanishes (\lambda n. X n - Y n)}
\end{itemize}
\textbf{proof} \texttt{(rule \texttt{vanishesI})}
\begin{itemize}
\item \texttt{fix r :: \texttt{rat}} \texttt{assume r: 0 < r}
\item \texttt{obtain a where a: 0 < a \forall n. |X n| < a}
\item \texttt{using X by fast}
\item \texttt{obtain b where b: 0 < b r = a * b}
\item \texttt{proof}
\item \texttt{show 0 < r / a using r a by simp}
\item \texttt{show r = a * (r / a) using a by simp}
\item \texttt{qed}
\item \texttt{obtain k where k: \forall n \geq k. |Y n| < b}
\item \texttt{using \texttt{vanishesD [OF Y b(1)] ..}}
\item \texttt{have \forall n \geq k. |X n * Y n| < r}
\item \texttt{by (simp add: b(2) \texttt{abs-mult \texttt{mult-strict-mono'}} a k)}
\item \texttt{thus \exists k. \forall n \geq k. |X n * Y n| < r ..}
\item \texttt{qed}
\end{itemize}
97.3 Cauchy sequences

definition
cauchy :: (nat ⇒ rat) ⇒ bool

where
cauchy X ←→ (∀ r>0. ∃ k. ∀ m≥k. ∀ n≥k. |X m − X n| < r)

lemma cauchyl:
(∀ r. 0 < r ⇒ ∃ k. ∀ m≥k. ∀ n≥k. |X m − X n| < r) ⇒ cauchy X

unfolding cauchy-def by simp

lemma cauchyD:
[ cauchy X; 0 < r ] ⇒ ∃ k. ∀ m≥k. ∀ n≥k. |X m − X n| < r

unfolding cauchy-def by simp

lemma cauchy-const [simp]: cauchy (λn. x)

unfolding cauchy-def by simp

lemma cauchy-add [simp]:
assumes X: cauchy X and Y: cauchy Y
shows cauchy (λn. X n + Y n)
proof (rule cauchyl)
fix r :: rat assume 0 < r
then obtain s t where s: 0 < s and t: 0 < t and r: r = s + t
by (rule obtain-pos-sum)

obtain i where i: ∀ m≥i. ∀ n≥i. |X m − X n| < s
using cauchyD [OF X s] ..

obtain j where j: ∀ m≥j. ∀ n≥j. |Y m − Y n| < t
using cauchyD [OF Y t] ..

have ∀ m≥max i j. ∀ n≥max i j. |(X m + Y m) − (X n + Y n)| < r
proof (clarsimp)
fix m n assume *: i ≤ m j ≤ m i ≤ n j ≤ n
have |(X m + Y m) − (X n + Y n)| ≤ |X m − X n| + |Y m − Y n|
unfolding add-diff-add by (rule abs-triangle-ineq)
also have ... < s + t
by (rule add-strict-mono, simp-all add: i j *)
finally show |(X m + Y m) − (X n + Y n)| < r unfolding r .
qed
thus ∃ k. ∀ m≥k. ∀ n≥k. |(X m + Y m) − (X n + Y n)| < r ..
qed

lemma cauchy-minus [simp]:
assumes X: cauchy X
shows cauchy (λn. − X n)
using assms unfolding cauchy-def
unfolding minus-diff-minus abs-minus-cancel .

lemma cauchy-diff [simp]:
assumes X: cauchy X and Y: cauchy Y
shows cauchy (λn. X n − Y n)
using assms unfolding diff-conv-add-uminus by (simp del: add-uminus-conv-diff)

lemma cauchy-imp-bounded:
assumes cauchy X shows \( \exists b > 0. \forall n. |X n| < b \)
proof
  obtain k where k: \( \forall m \geq k. \forall n \geq k. |X m - X n| < 1 \)
  using cauchyD [OF assms zero-less-one] ..
  show \( \exists b > 0. \forall n. |X n| < b \)
  proof (intro exI conjI allI)
  have 0 \( \leq |X 0| \) by simp
  also have \( |X 0| \leq \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} \) by simp
  finally have 0 \( \leq \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} \).
  thus 0 \( < \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} + 1 \) by simp
next
  fix n :: nat
  show \( |X n| < \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} + 1 \)
  proof (rule linorder-le-cases)
    assume n \( \leq k \)
    hence \( |X n| \leq \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} \) by simp
    thus \( |X n| < \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} + 1 \) by simp
  next
  assume k \( \leq n \)
  have \( |X n| = |X k + (X n - X k)| \) by simp
  also have \( |X k + (X n - X k)| \leq |X k| + |X n - X k| \)
    by (rule abs-triangle-ineq)
  finally have \( \ldots < \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} + 1 \)
    by (rule add-le-less-mono, simp, simp add: k ⟨k \( \leq n\)⟩)
  finally show \( |X n| < \operatorname{Max} \operatorname{abs} \{ X \_{..k} \} + 1 \).
  qed
qed

lemma cauchy-mult [simp]:
assumes X: cauchy X and Y: cauchy Y
shows cauchy (\( \lambda n. X n \ast Y n \) )
proof (rule cauchyI)
  fix r :: rat assume \( \theta < r \)
  then obtain u v where u: \( \theta < u \) and v: \( \theta < v \) and \( r = u + v \)
    by (rule obtain-pos-sum)
  obtain a where a: \( \theta < a \) \( \forall n. |X n| < a \)
    using cauchy-imp-bounded [OF X] by fast
  obtain b where b: \( \theta < b \) \( \forall n. |Y n| < b \)
    using cauchy-imp-bounded [OF Y] by fast
  obtain s t where s: \( \theta < s \) and t: \( \theta < t \) and \( r = a \ast t + s \ast b \)
  proof
    show \( \theta < v/b \) using v b(1) by simp
    show \( \theta < u/a \) using u a(1) by simp
    show \( r = a \ast (u/a) + (v/b) \ast b \)
      using a(1) b(1) \( \forall r = a + v \) by simp
theory "Real"

|

proof

- cauchy-not-vanishes-cases

qed

lemma cauchy-not-vanishes-cases:

assumes X: cauchy X

assumes nz: ~ vanishes X

shows \( \exists b > 0. \exists k. (\forall n \geq k. b \leq X n) \lor (\forall n \geq k. b < X n) \)

proof

- obtain r where \( \theta < r \) and \( r: \forall n \geq k. r \leq |X n| \)

  using nz unfolding vanishes-def by (auto simp add: not-less)

- obtain s t where s: \( \theta < s \) and \( t: \theta < t \) and \( r = s + t \)

  using \( \theta < r \) by (rule obtain-pos-sum)

- obtain i where i: \( \forall m \geq i. \forall n \geq i. |X m - X n| < s \)

  using cauchyD [OF X s] ..

- obtain k where i \leq k and \( r \leq |X k| \)

  using r by fast

  have k: \( \forall n \geq k. |X n - X k| < s \)

  using i \leq k by auto

  have X k \leq - r \lor r \leq X k

  using \( r \leq |X k| \) by auto

  hence \( (\forall n \geq k. t < - X n) \lor (\forall n \geq k. t < X n) \)

  unfolding \( r = s + t \) using k by auto

  hence \( \exists k. (\forall n \geq k. t < - X n) \lor (\forall n \geq k. t < X n) \)

  thus \( \exists t > 0. \exists k. (\forall n \geq k. t < - X n) \lor (\forall n \geq k. t < X n) \)

  using t by auto

qed

lemma cauchy-not-vanishes:

assumes X: cauchy X

assumes nz: ~ vanishes X
shows $\exists b > 0. \exists k. \forall n \geq k. b < |X n|$
using cauchy-not-vanishes-cases [OF assms]
by clarify (rule exI,erule conjI,rule-tac x=k in exI,auto)

lemma cauchy-inverse [simp]:
  assumes X: cauchy X
  assumes nz: ~ vanishes X
  shows cauchy ($\lambda n. \text{inverse} (X n)$)
proof (rule cauchyI)
  fix r :: rat assume 0 < r
  obtain b i where b: 0 < b and i: $\forall n \geq i. b < |X n|$
    using cauchy-not-vanishes [OF X nz] by fast
  from b i have nz: $\forall n \geq i. X n \neq 0$ by auto
  obtain s where s: 0 < s and r: r = inverse b * s * inverse b
proof
  show 0 < b * r * b by (simp add: inverse : OP assms)
  show r = inverse b * (b * r * b) * inverse b
    using b by simp
qed

obtain j where j: $\forall m \geq j. \forall n \geq j. |X m - X n| < s$
  using cauchyD [OF X s] ..
  have $\forall m \geq \max i j. \forall n \geq \max i j. |\text{inverse} (X m) - \text{inverse} (X n)| < r$
proof (clarsimp)
  fix m n assume *: i \leq m j \leq m i \leq n j \leq n
  have $|\text{inverse} (X m) - \text{inverse} (X n)| =$
    inverse $|X m| * |X m - X n| * \text{inverse} |X n|$
    by (simp add: inverse-diff-inverse nz \* abs mult)
  also have $\ldots < \text{inverse} b * s * \text{inverse} b$
    by (simp add: mult-strict-monotone less_imp_inverse_less
      i j b \* s)
  finally show $|\text{inverse} (X m) - \text{inverse} (X n)| < r$ unfolding r ..
qed

thus $\exists k. \forall m \geq k. \forall n \geq k. |\text{inverse} (X m) - \text{inverse} (X n)| < r$. ..
qed

lemma vanishes-diff-inverse:
  assumes X: cauchy X ~ vanishes X
  assumes Y: cauchy Y ~ vanishes Y
  assumes XY: vanishes ($\lambda n. X n - Y n$)
  shows vanishes ($\lambda n. \text{inverse} (X n) - \text{inverse} (Y n)$)
proof (rule vanishesI)
  fix r :: rat assume r: 0 < r
  obtain a i where a: 0 < a and i: $\forall n \geq i. a < |X n|$
    using cauchy-not-vanishes [OF X] by fast
  obtain b j where b: 0 < b and j: $\forall n \geq j. b < |Y n|$
    using cauchy-not-vanishes [OF Y] by fast
  obtain s where s: 0 < s and inverse a * s * inverse b = r
proof
  show 0 < a * r * b
using 
  a r b 
  by simp 
  show 
  inverse a * (a * r * b) * inverse b = r 
  using 
  a r b 
  by simp 
  qed 

obtain 
  k 
  where 
  k. ∀ n ≥ k. |X n − Y n| < s 
  using 
  vanishesD [OF XY s] .. 
  have 
  ∀ n ≥ max (max i j) k. |inverse (X n) − inverse (Y n)| < r 
  proof (clarsimp) 
    fix 
    n 
    assume 
    n: i ≤ n j ≤ n k ≤ n 
    have 
    X n ≠ 0 
    and 
    Y n ≠ 0 
    using 
    i j a b n 
    by auto 
    hence 
    |inverse (X n) − inverse (Y n)| = 
    inverse |X n| * |X n − Y n| * inverse |Y n| 
    by (simp add: inverse-diff-inverse abs-mult) 
    also have 
    ... < inverse a * s * inverse b 
    apply (intro mult-strict-mono' less-imp-inverse-less) 
    apply (simp-all add: a b i j k n) 
    done 
    also note 
    (inverse a * s * inverse b = r) 
    finally show 
    |inverse (X n) − inverse (Y n)| < r .. 
  qed 

thus 
  ∃ k. ∀ n ≥ k. |inverse (X n) − inverse (Y n)| < r .. 
  qed 

97.4  Equivalence relation on Cauchy sequences 

definition realrel :: (nat ⇒ rat) ⇒ (nat ⇒ rat) ⇒ bool 
  where 
  realrel = (λ X Y. cauchy X ∧ cauchy Y ∧ vanishes (λ n. X n − Y n)) 

lemma realrelI [intro?]: 
  assumes 
  cauchy X and 
  cauchy Y and 
  vanishes (λ n. X n − Y n) 
  shows 
  realrel X Y 
  using 
  assms 
  unfolding realrel-def by simp 

lemma realrel-refl: cauchy X ⇒ realrel X X 
  unfolding realrel-def by simp 

lemma symp-realrel: symp realrel 
  unfolding realrel-def 
  by (rule sympI, clarify, drule vanishes-minus, simp) 

lemma transp-realrel: transp realrel 
  unfolding realrel-def 
  apply (rule transpI, clarify) 
  apply (drule (1) vanishes-add) 
  apply (simp add: algebra-simps) 
  done 

lemma part-equivp-realrel: part-equivp realrel
by (fast intro: part-equivI symp-realrel transp-realrel
  realrel-refl cauchy-const)

97.5 The field of real numbers

quotient-type real = nat ⇒ rat / partial: realrel
  morphisms rep-real Real
  by (rule part-equivp-realrel)

lemma cr-real-eq: pcr-real = (λx y. cauchy x ∧ Real x = y)
  unfolding real.pcr-cr-eq cr-real-def realrel-def by auto

lemma Real-induct [induct type: real]:
  assumes ⋀X. cauchy X ⇒ P (Real X) shows P x
  proof (induct x)
    case (1 X)
    hence cauchy X by (simp add: realrel-def)
    thus P (Real X) by (rule assms)
  qed

lemma eq-Real: cauchy X ⇒ cauchy Y ⇒ Real X = Real Y ←→ vanishes (λn. X n − Y n)
  using real.rel-eq-transfer
  unfolding real.pcr-cr-eq cr-real-def rel-fun-def realrel-def by simp

lemma Domainp-pcr-real [transfer-domain-rule]: Domainp pcr-real = cauchy
  by (simp add: real.domain-eq realrel-def)

instantiation real :: field-inverse-zero
begin

lift-definition zero-real :: real is λn. 0
  by (simp add: realrel-refl)

lift-definition one-real :: real is λn. 1
  by (simp add: realrel-refl)

lift-definition plus-real :: real ⇒ real ⇒ real is λX Y n. X n + Y n
  unfolding realrel-def add-diff-add
  by (simp only: cauchy-add vanishes-add simp-thms)

lift-definition uminus-real :: real ⇒ real is λX n. − X n
  unfolding realrel-def minus-diff-minus
  by (simp only: cauchy-minus vanishes-minus simp-thms)

lift-definition times-real :: real ⇒ real ⇒ real is λX Y n. X n * Y n
  unfolding realrel-def mult-diff-mult
  by (subst (4) mult.commute, simp only: cauchy-mult vanishes-add
  vanishes-mult-bounded cauchy-imp-bounded simp-thms)
lift-definition inverse-real :: real ⇒ real
is λX. if vanishes X then (λn. 0) else (λn. inverse (X n))

proof –
fix X Y assume realrel X Y
hence X: cauchy X and Y: cauchy Y and XY: vanishes (λn. X n − Y n)
  unfolding realrel-def by simp-all
have vanishes X ←→ vanishes Y
proof
  assume vanishes X
  from vanishes-diff [OF this XY] show vanishes Y by simp
next
  assume vanishes Y
  from vanishes-add [OF this XY] show vanishes X by simp
qed
thus ?thesis X Y
  unfolding realrel-def
  by (simp add: vanishes-diff-inverse X Y XY)
qed

definition x − y = (x::real) + − y

definition x / y = (x::real) * inverse y

lemma add-Real:
  assumes X: cauchy X and Y: cauchy Y
  shows Real X + Real Y = Real (λn. X n + Y n)
  using assms plus-real.transfer
  unfolding cr-real-eq rel-fun-def by simp

lemma minus-Real:
  assumes X: cauchy X
  shows − Real X = Real (λn. − X n)
  using assms uminus-real.transfer
  unfolding cr-real-eq rel-fun-def by simp

lemma diff-Real:
  assumes X: cauchy X and Y: cauchy Y
  shows Real X − Real Y = Real (λn. X n − Y n)
  unfolding minus-real-def
  by (simp add: minus-Real add-Real X Y)

lemma mult-Real:
  assumes X: cauchy X and Y: cauchy Y
  shows Real X * Real Y = Real (λn. X n * Y n)
  using assms times-real.transfer
  unfolding cr-real-eq rel-fun-def by simp
lemma inverse-Real:
  assumes X: cauchy X
  shows inverse (Real X) = 
    (if vanishes X then 0 else Real (λn. inverse (X n)))
using assms inverse-real.transfer zero-real.transfer
unfolding cr-real-eq rel-fun-def by (simp split: split-if-asm, metis)

instance proof
  fix a b c :: real
  show a + b = b + a
    by transfer (simp add: ac-simps realrel-def)
  show (a + b) + c = a + (b + c)
    by transfer (simp add: ac-simps realrel-def)
  show 0 + a = a
    by transfer (simp add: realrel-def)
  show − a + a = 0
    by transfer (simp add: realrel-def)
  show a − b = a + − b
    by (rule minus-real-def)
  show (a * b) * c = a * (b * c)
    by transfer (simp add: ac-simps realrel-def)
  show a * b = b * a
    by transfer (simp add: ac-simps realrel-def)
  show 1 * a = a
    by transfer (simp add: ac-simps realrel-def)
  show (a + b) * c = a * c + b * c
    by transfer (simp add: distrib-right realrel-def)
  show (0::real) ≠ (1::real)
    by transfer (simp add: realrel-def)
  show a ≠ 0 ⇒ inverse a * a = 1
    apply transfer
    apply (simp add: realrel-def)
    apply (rule vanishesI)
    apply (frule (1) cauchy-not-vanishes, clarify)
    apply (rule-tac x=k in exI, clarify)
    apply (drule-tac x=n in spec, simp)
    done
  show a / b = a * inverse b
    by (rule divide-real-def)
  show inverse (0::real) = 0
    by transfer (simp add: realrel-def)
qed

end

97.6 Positive reals

lift-definition positive :: real ⇒ bool
is \( \lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X n \)

proof

\[
\begin{aligned}
&\text{fix } X Y \\
&\text{assume } \text{realrel } X Y \\
&\text{hence } XY: \text{vanishes } (\lambda n. X n - Y n) \\
&\text{unfolding } \text{realrel-def by simp-all} \\
&\text{assume } \exists r > 0. \exists k. \forall n \geq k. r < X n \\
&\text{then obtain } r i \text{ where } 0 < r \text{ and } i: \forall n \geq i. r < X n \\
&\text{by fast} \\
&\text{obtain } s t \text{ where } s: 0 < s \text{ and } t: 0 < t \text{ and } r = s + t \\
&\text{using } (0 < r) \text{ by (rule obtain-pos-sum)} \\
&\text{obtain } j \text{ where } j: \forall n \geq j. |X n - Y n| < s \\
&\text{using } \text{vanishesD } [OF XY s] .. \\
&\text{have } \forall n \geq \max i j. t < Y n \\
&\text{proof (clarsimp)} \\
&\text{fix } n \text{ assume } n: i \leq n \text{ and } j \leq n \\
&\text{have } |X n - Y n| < s \text{ and } r < X n \\
&\text{using } i j n \text{ by simp-all} \\
&\text{thus } t < Y n \text{ unfolding } r \text{ by simp} \\
&\text{qed} \\
&\text{hence } \exists r > 0. \exists k. \forall n \geq k. r < Y n \text{ using } t \text{ by fast} \\
\end{aligned}
\]

\} note 1 = this

\begin{aligned}
&\text{fix } X Y \text{ assume } \text{realrel } X Y \\
&\text{hence } \text{realrel } X Y \text{ and } \text{realrel } Y X \\
&\text{using } \text{ symlink-realrel } \text{ unfolding } \text{symp-def by auto} \\
&\text{thus } ?\text{thesis } X Y \\
&\text{by (safe elim!: 1)} \\
&\text{qed}
\end{aligned}

\begin{aligned}
\text{lemma } \text{positive-Real:} \\
&\text{assumes } X: \text{cauchy } X \\
&\text{shows } \text{positive } (\text{Real } X) \iff (\exists r > 0. \exists k. \forall n \geq k. r < X n) \\
&\text{using } \text{assms positive-transfer} \\
&\text{unfolding } \text{cr-real-eq rel-fun-def by simp}
\end{aligned}

\begin{aligned}
\text{lemma } \text{positive-zero: } \neg \text{positive } 0 \\
&\text{by transfer auto}
\end{aligned}

\begin{aligned}
\text{lemma } \text{positive-add:} \\
&\text{positive } x \implies \text{positive } y \implies \text{positive } (x + y) \\
&\text{apply transfer} \\
&\text{apply (clarify, rename-tac a b i j)} \\
&\text{apply (rule-tac } x = a + b \text{ in exI, simp)} \\
&\text{apply (rule-tac } x = \text{max i j in exI, clarsimp)} \\
&\text{apply (simp add: add-strict-mono)} \\
&\text{done}
\end{aligned}

\begin{aligned}
\text{lemma } \text{positive-mult:} \\
&\text{positive } x \implies \text{positive } y \implies \text{positive } (x * y)
\end{aligned}
apply transfer
apply (clarify, rename-tac a b i j)
apply (rule-tac $x = a * b$ in \textit{exI}, simp)
apply (rule-tac $x = \max i j$ in \textit{exI}, clarsimp)
apply (rule mult-strict-mono, auto)
done

lemma positive-minus:
\[ \neg \text{positive } x \implies x \neq 0 \implies \text{positive } (-x) \]
apply transfer
apply (simp add: realrel-def)
apply (drule (1) cauchy-not-vanishes-cases, safe, fast, fast)
done

instantiation real :: linordered-field-inverse-zero
begin

definition $x < y \iff \text{positive } (y - x)$

definition $x \leq (y :: \text{real}) \iff x < y \lor x = y$

definition $\text{abs } (a :: \text{real}) = (\text{if } a < 0 \text{ then } -a \text{ else } a)$

definition $\text{sgn } (a :: \text{real}) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)$

instance proof
fix $a, b, c :: \text{real}$
show $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$
by (rule abs-real-def)
show $a < b \iff a \leq b \land \neg b \leq a$
unfolding less-eq-real-def less-real-def
by (auto, drule (1) positive-add, simp-all add: positive-zero)
show $a < a$
unfolding less-eq-real-def by simp
show $a \leq b \implies b \leq c \implies a \leq c$
unfolding less-eq-real-def less-real-def
by (auto, drule (1) positive-add, simp add: algebra-simps)
show $a \leq b \implies b \leq a \implies a = b$
unfolding less-eq-real-def less-real-def
by (auto, drule (1) positive-add, simp add: positive-zero)
show $a \leq b \implies c + a \leq c + b$
unfolding less-eq-real-def less-real-def by auto

show $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)$
by (rule sgn-real-def)
show \(a \leq b \lor b \leq a\)
unfolding less-eq-real-def less-real-def
by (auto dest!: positive-minus)
show \(a < b \implies 0 < c \implies c + a < c + b\)
unfolding less-real-def
by (drule (1) positive-mult, simp add: algebra-simps)
qed

instantiation real :: distrib-lattice
begin

definition \(\text{inf} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}\) = min

definition \(\text{sup} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}\) = max

instance proof
qed (auto simp add: inf-real-def sup-real-def max-min-distrib2)

end

lemma of-nat-Real: of-nat \(x\) = Real \((\lambda n. \text{of-nat } x)\)
apply (induct \(x\))
apply (simp add: zero-real-def)
apply (simp add: one-real-def add-Real)
done

lemma of-int-Real: of-int \(x\) = Real \((\lambda n. \text{of-int } x)\)
apply (cases \(x\) rule: int-diff-cases)
apply (simp add: of-nat-Real diff-Real)
done

lemma of-rat-Real: of-rat \(x\) = Real \((\lambda n. \text{of-rat } x)\)
apply (induct \(x\))
apply (simp add: Fract-of-int-quotient of-rat-divide)
apply (simp add: of-int-Real divide-inverse)
apply (simp add: inverse-Real mult-Real)
done

instance real :: archimedean-field
proof
fix \(x\) :: real
show \(\exists z. x \leq \text{of-int } z\)
apply (induct \(x\))
apply (frule cauchy-imp-bounded, clarify)

qed
apply (rule-tac \(x=\text{ceiling } b + 1\) in exI)
apply (rule less-imp-le)
apply (simp add: of-int-Real less-real-def diff-Real positive-Real)
apply (rule-tac \(x=1\) in exI, simp add: algebra-simps)
apply (rule-tac \(x=0\) in exI, clarsimp)
apply (rule le-less-trans [OF abs-ge-self])
apply (rule less-le-trans [OF - le-of-int-ceiling])
apply simp
done

qed

instantiation real :: floor-ceiling
begin

definition [code del]:
floor (x :: real) = (THE z. of-int z \(\le\) x \(\land\) x < of-int (z + 1))

instance proof
  fix x :: real
  show of-int (floor x) \(\le\) x \(\land\) x < of-int (floor x + 1)
    unfolding floor-real-def using floor-exists1 by (rule theI')
qed

end

97.7 Completeness

lemma not-positive-Real:
  assumes X: cauchy X
  shows \(\neg\) positive (Real X) \iff\ (\(\forall r>0. \exists k. \forall n\ge k. X n \le r\))
unfolding positive-Real [OF X]
apply (auto, unfold not-less)
apply (erule obtain-pos-sum)
apply (drule-tac \(x=s\) in spec, simp)
apply (drule-tac \(r=t\) in cauchyD [OF X], clarify)
apply (drule-tac \(x=k\) in spec, clarsimp)
apply (rule-tac \(x=n\) in exI, clarify, rename-tac m)
apply (drule-tac \(x=m\) in spec, simp)
apply (drule-tac \(x=n\) in spec, simp)
apply (drule spec, drule (1) mp, clarify, rename-tac i)
apply (rule-tac \(x=max i k\) in exI, simp)
done

lemma le-Real:
  assumes X: cauchy X and Y: cauchy Y
  shows Real X \(\le\) Real Y = (\(\forall r>0. \exists k. \forall n\ge k. X n \le Y n + r\))
unfolding not-less [symmetric, where 'a=real] less-real-def
apply (simp add: diff-Real not-positive-Real X Y)
apply (simp add: diff-le-eq ac-simps)
done

lemma le-RealI:
  assumes Y: cauchy Y
  shows ∀ n. x ≤ of-rat (Y n) ⟷ x ≤ Real Y
proof (induct x)
  fix X assume X: cauchy X and ∀ n. Real X ≤ of-rat (Y n)
hence le: ∀ m r. 0 < r ⟷ ∃ k. ∀ n≥k. X n ≤ Y m + r
  by (simp add: of-rat-Real le-Real)
{  
  fix r :: rat assume 0 < r
  then obtain s t where s: 0 < s and t: 0 < t and r: r = s + t
    by (rule obtain-pos-sum)
  obtain i where i: ∀ m≥i. ∀ n≥i. |Y m − Y n| < s
    using cauchyD [OF Y s] ..
  obtain j where j: ∀ n≥j. X n ≤ Y i + t
    using le [OF t] ..
  have ∀ n≥max i j. X n ≤ Y n + r
  proof (clarsimp)
    fix n assume n: i ≤ n j ≤ n
    have X n ≤ Y i + t using n j by simp
    moreover have |Y i − Y n| < s using n i by simp
    ultimately show X n ≤ Y n + r unfolding r by simp
  qed
  hence ∃ k. ∀ n≥k. X n ≤ Y n + r ..
}
thus Real X ≤ Real Y
  by (simp add: of-rat-Real le-Real X Y)
qed

lemma Real-leI:
  assumes X: cauchy X
  assumes le: ∀ n. of-rat (X n) ≤ y
  shows Real X ≤ y
proof –
  have − y ≤ − Real X
    by (simp add: minus-Real X le-RealI of-rat-minus le)
  thus thesis by simp
qed

lemma less-RealD:
  assumes Y: cauchy Y
  shows x < Real Y ⟷ ∃ n. x < of-rat (Y n)
by (erule contrapos-pp, simp add: not-less, erule Real-leI [OF Y])

lemma of-nat-less-two-power:
  of-nat n < (2 :: a::linordered-idom) ^ n
apply (induct n)
apply simp
apply (subgoal-tac (1::'a) ≤ 2 ^ n)
apply (drule (1) add-le-less-mono, simp)
apply simp
done

lemma complete-real:
  fixes S :: real set
  assumes ∃x. x ∈ S and ∃y. ∀x∈S. x ≤ y
  shows ∃z. (∀x∈S. x ≤ y) ∧ (∀z. (∀x∈S. x ≤ z) → y ≤ z)
proof
  obtain x where x: x ∈ S using assms(1) ..
  obtain z where z: ∀x∈S. x ≤ z using assms(2) ..

  def P ≡ λx. ∀y∈S. y ≤ of-rat x
  obtain a where a: ¬ P a
  proof
    have of-int (floor (x - 1)) ≤ x - 1 by (rule of-int-floor-le)
    also have x - 1 < x by simp
    finally have of-int (floor (x - 1)) < x .
    hence ¬ x ≤ of-int (floor (x - 1)) by (simp only: not-le)
    then show ¬ P (of-int (floor (x - 1)))
      unfolding P-def of-rat-of-int-eq using x by fast
  qed
  obtain b where b: P b
  proof
    show P (of-int (ceiling z))
    unfolding P-def of-rat-of-int-eq
    proof
      fix y assume y: y ∈ S
      hence y ≤ z using z by simp
      also have z ≤ of-int (ceiling z) by (rule le-of-int-ceiling)
      finally show y ≤ of-int (ceiling z) .
    qed
  qed

  def avg ≡ λx y :: rat. x/2 + y/2
  def bisect ≡ λ(x, y). if P (avg x y) then (x, avg x y) else (avg x y, y)
  def A ≡ λn. fst ((bisect ^^ n) (a, b))
  def B ≡ λn. snd ((bisect ^^ n) (a, b))
  def C ≡ λn. avg (A n) (B n)
  have A-0 [simp]: A 0 = a unfolding A-def by simp
  have B-0 [simp]: B 0 = b unfolding B-def by simp
  have A-Suc [simp]: ∀n. A (Suc n) = (if P (C n) then A n else C n)
    unfolding A-def B-def C-def bisect-def split-def by simp
  have B-Suc [simp]: ∀n. B (Suc n) = (if P (C n) then C n else B n)
    unfolding A-def B-def C-def bisect-def split-def by simp
  have width: ∀n. B n - A n = (b - a) / 2 ^ n
  apply (simp add: eq-divide-eq)
apply (induct-tac n, simp)
apply (simp add: C-def avg-def algebra-simps)
done

have \( \forall y \in \mathbb{R}. 0 < r \Rightarrow \exists n. y / 2^n < r \)
  apply (simp add: divide-less-eq)
  apply (subst mult.commute)
apply (frule_tac \( y = y \) in ex-less-of-nat-mult)
apply clarify
apply (rule_tac \( x = n \) in exI)
apply (erule less-trans)
apply (rule mult-strict-right-mono)
apply (rule le-less-trans [OF of-nat-less-two-power])
apply simp
apply assumption
done

have \( \forall n. \neg P (A n) \)
  by (induct-tac n, simp-all add: a)
have \( \forall n. P (B n) \)
  by (induct-tac n, simp-all add: b)

have \( a < b \)
  using a b unfolding P-def
apply (clarsimp simp add: not-le)
apply (erule (1) bspec)
apply (erule (1) less-le-trans)
apply (simp add: of-rat-less)
done

have \( \forall n. A n < B n \)
  by (induct-tac n, simp add: ab, simp add: C-def avg-def)

have \( \forall i j. i \leq j \Rightarrow A i \leq A j \)
  apply (auto simp add: le-less [where \( 'a=nat \)])
apply (erule less-Suc-induct)
apply (clarsimp simp add: C-def avg-def)
apply (simp add: add-divide-distrib [symmetric])
apply (rule AB [THEN less-imp-le])
apply simp
done

have \( \forall i j. i \leq j \Rightarrow B j \leq B i \)
  apply (auto simp add: le-less [where \( 'a=nat \)])
apply (erule less-Suc-induct)
apply (clarsimp simp add: C-def avg-def)
apply (simp add: add-divide-distrib [symmetric])
apply (rule AB [THEN less-imp-le])
apply simp
done

have cauchy-lemma:
  \[ \forall X. \forall i \geq n. A n \leq X i \land X i \leq B n \Rightarrow \text{cauchy } X \]
apply (rule cauchyI)
apply (drule twos [where \(\text{y} = b - a\)])
apply (erule exE)
apply (rule-tac \(x = n\) in \(\text{exI}\), clarify, rename-tac i j)
apply (rule-tac \(\text{y} = B n - A n\) in \(\text{le-less-trans}\)) defer
apply (simp add: width)
apply (drule-tac \(x = n\) in \(\text{spec}\))
apply (frule-tac \(x = i\) in \(\text{spec}\), drule \(1\) mp)
apply (frule-tac \(x = j\) in \(\text{spec}\), drule \(1\) mp)
apply (frule \(A\)-mono, drule \(B\)-mono)
apply (frule \(A\)-mono, drule \(B\)-mono)
apply arith
done
have \(\text{cauchy A}\)
apply (rule cauchy-lemma [rule-format])
apply (simp add: \(A\)-mono)
apply (erule order-trans [OF \(\text{less-imp-le}\) [OF \(\text{AB}\) \(B\)-mono]])
done
have \(\text{cauchy B}\)
apply (rule cauchy-lemma [rule-format])
apply (simp add: \(B\)-mono)
apply (erule order-trans [OF \(\text{A\-mono \text{less-imp-le}}\) [OF \(\text{AB}\)]])
done
have \(1\): \(\forall x \in S. \ x \leq \text{Real B}\)
proof
fix \(x\) assume \(x \in S\)
then show \(x \leq \text{Real B}\)
using \(\text{PB [unfolded P-def]}\) (cauchy \(B\))
by (simp add: le-RealI)
qed
have \(2\): \(\forall z. (\forall x \in S. \ x \leq z) \rightarrow \text{Real A} \leq z\)
apply clarify
apply (erule contrapos-pp)
apply (simp add: not-le)
apply (drule less-RealD [OF \(\text{cauchy A}\) [unf.]], clarify)
apply (subgoal-tac \(\neg P (A n)\))
apply (simp add: P-def not-le, clarify)
apply (erule rev-bexI)
apply (erule \(1\) less-trans)
apply (simp add: PA)
done
have \(\text{vanishes}\) (\(\lambda n. (b - a) / 2 \sim n\))
proof (rule vanishesL)
fix \(r\) :: rat assume \(\theta < r\)
then obtain \(k\) where \(k: |b - a| / 2 \sim k < r\)
using \(\text{twos}\) by fast
have \(\forall n \geq k. |(b - a) / 2 \sim n| < r\)
proof (clarify)
fix \(n\) assume \(n: k \leq n\)
have \(|(b - a) / 2 \sim n| = |b - a| / 2 \sim n||
by simp
also have \( \ldots \leq |b - a| / 2^k \)
using \( n \) by (simp add: divide-left-mono)
also note \( k \)
finally show \( |(b - a) / 2^n| < r \).
\[ \text{qed} \]
thus \( \exists k. \forall n \geq k. |(b - a) / 2^n| < r \).
\[ \text{qed} \]
hence
3. Real \( B = \text{Real A} \)
by (simp add: eq-Real [cauchy A] [cauchy B])
show \( \exists y. (\forall x \in S. x \leq y) \land (\forall z. (\forall x \in S. x \leq z) \rightarrow y \leq z) \)
using 1 2 3 by (rule-tac x=Real B in exI, simp)
\[ \text{qed} \]

instantiation real :: linear-continuum
begin

97.8 Supremum of a set of reals

definition \( \text{Sup } X = (\text{LEAST } z::\text{real}. \forall x\in X. x \leq z) \)
definition \( \text{Inf } (X::\text{real set}) = - \text{Sup } (-X) \)

instance
proof
\{ fix \( x \) :: real and \( X \) :: real set
assume \( x: x \in X \text{ bdd-above } X \)
then obtain \( s \) where \( s: \forall y\in X. y \leq s \land \forall y\in X. y \leq z \implies s \leq z \)
using complete-real[of X] unfolding bdd-above-def by blast
then show \( x \leq \text{Sup } X \)
unfolding Sup-real-def by (rule LeastI2-order) (auto simp: x)
\}
note Sup-upper = this

\{ fix \( z \) :: real and \( X \) :: real set
assume \( x: X \neq \{\} \) and \( z: \forall x. x \in X \implies x \leq z \)
then obtain \( s \) where \( s: \forall y\in X. y \leq s \land \forall y\in X. y \leq z \implies s \leq z \)
using complete-real[of X] by blast
then have \( \text{Sup } X = s \)
unfolding Sup-real-def by (best intro: Least-equality)
also from \( s \) have \( \ldots \leq z \)
by blast
finally show \( \text{Sup } X \leq z \).
\}
note Sup-least = this

\{ fix \( x \) :: real and \( X \) :: real set assume \( x: x \in X \text{ bdd-below } X \) then show \( \text{Inf } X \leq x \)
using Sup-upper[of \#-X uminus ' X] by (auto simp: Inf-real-def)
\}
\{ fix \( z \) :: real and \( X \) :: real set assume \( X \neq \{\} \) and \( x: x \in X \implies z \leq x \) then show \( z \leq \text{Inf } X \)
using Sup-least[of uminus ' X - z] by (force simp: Inf-real-def)
\}
show $\exists a\ b::\text{real}.\ a \neq b$
  using zero-neq-one by blast
qed
end

97.9 Hiding implementation details

hide-const (open) vanishes cauchy positive Real

declare Real-induct [induct del]
declare Abs-real-induct [induct del]
declare Abs-real-cases [cases del]

lifting-update real.lifting
lifting-forget real.lifting

97.10 More Lemmas

BH: These lemmas should not be necessary; they should be covered by existing simp rules and simplification procedures.

lemma real-mult-less-iff1 [simp]: $(0::\text{real}) < z ==> (x*z < y*z) = (x < y)$
  by simp

lemma real-mult-le-cancel-iff1 [simp]: $(0::\text{real}) < z ==> (x*z \leq y*z) = (x \leq y)$
  by simp

lemma real-mult-le-cancel-iff2 [simp]: $(0::\text{real}) < z ==> (z*x \leq z*y) = (x \leq y)$
  by simp

97.11 Embedding numbers into the Reals

abbreviation
  real-of-nat :: nat $\Rightarrow$ real
where
  real-of-nat $\equiv$ of-nat

abbreviation
  real-of-int :: int $\Rightarrow$ real
where
  real-of-int $\equiv$ of-int

abbreviation
  real-of-rat :: rat $\Rightarrow$ real
where
  real-of-rat $\equiv$ of-rat

consts

  real :: 'a $\Rightarrow$ real
THEORY “Real”

defs (overloaded)
  real-of-nat-def [code-unfold]: real == real-of-nat
  real-of-int-def [code-unfold]: real == real-of-int

declare [[coercion-enabled]]
declare [[coercion real::nat⇒real]]
declare [[coercion real::int⇒real]]
declare [[coercion int]]
declare [[coercion-map map]]
declare [[coercion-map % f g h x. g (h (f x))]]
declare [[coercion-map % f g (x,y). (f x, g y)]]

lemma real-eq-of-nat: real = of-nat
  unfolding real-of-nat-def ..

lemma real-eq-of-int: real = of-int
  unfolding real-of-int-def ..

lemma real-of-int-zero [simp]: real (0::int) = 0
  by (simp add: real-of-int-def)

lemma real-of-one [simp]: real (1::int) = (1::real)
  by (simp add: real-of-int-def)

lemma real-of-int-add [simp]: real(x + y) = real(x::int) + real y
  by (simp add: real-of-int-def)

lemma real-of-int-minus [simp]: real(−x) = −real(x::int)
  by (simp add: real-of-int-def)

lemma real-of-int-diff [simp]: real(x − y) = real(x::int) − real y
  by (simp add: real-of-int-def)

lemma real-of-int-mult [simp]: real(x * y) = real(x::int) * real y
  by (simp add: real-of-int-def)

lemma real-of-int-power [simp]: real(x ^ n) = real(x::int) ^ n
  by (simp add: real-of-int-def real-of-int-power)

lemmas power-real-of-int = real-of-int-power [symmetric]

lemma real-of-int-setsum [simp]: real ((SUM x:A. f x)::int) = (SUM x:A. real(f x))
  apply (subst real-eq-of-int)+
  apply (rule of-int-setsum)
  done
lemma real-of-int-setprod [simp]: real ((\(\prod\)) \(\times A. f x\)) = real ((\(\prod\)) \(\times A.\) real(f x))
apply (subst real-eq-of-int)+
apply (rule of-int-setprod)
done

lemma real-of-int-zero-cancel [simp, algebra, presburger]: (real \(x = 0\)) = \((x = (0::\text{int}))\)
by (simp add: real-of-int-def)

lemma real-of-int-inject [iff, algebra, presburger]: \((\text{real} (x::\text{int}) = \text{real} y)\) = \((x = y)\)
by (simp add: real-of-int-def)

lemma real-of-int-less-iff [iff, presburger]: \((\text{real} (x::\text{int}) < \text{real} y)\) = \((x < y)\)
by (simp add: real-of-int-def)

lemma real-of-int-le-iff [simp, presburger]: \((\text{real} (x::\text{int}) \leq \text{real} y)\) = \((x \leq y)\)
by (simp add: real-of-int-def)

lemma real-of-int-gt-zero-cancel-iff [simp, presburger]: \((0 < \text{real} (n::\text{int}))\) = \((0 < n)\)
by (simp add: real-of-int-def)

lemma real-of-int-ge-zero-cancel-iff [simp, presburger]: \((0 \leq \text{real} (n::\text{int}))\) = \((0 \leq n)\)
by (simp add: real-of-int-def)

lemma real-of-int-lt-zero-cancel-iff [simp, presburger]: \((\text{real} (n::\text{int}) < 0)\) = \((n < 0)\)
by (simp add: real-of-int-def)

lemma real-of-int-le-zero-cancel-iff [simp, presburger]: \((\text{real} (n::\text{int}) \leq 0)\) = \((n \leq 0)\)
by (simp add: real-of-int-def)

lemma one-less-real-of-int-cancel-iff: \(1 < \text{real} (i :: \text{int})\) = \(1 < i\)
unfolding real-of-one[symmetric] real-of-int-less-iff ..

lemma one-le-real-of-int-cancel-iff: \(1 \leq \text{real} (i :: \text{int})\) = \(1 \leq i\)
unfolding real-of-one[symmetric] real-of-int-le-iff ..

lemma real-of-int-less-one-cancel-iff: \(\text{real} (i :: \text{int}) < 1\) = \(i < 1\)
unfolding real-of-one[symmetric] real-of-int-less-iff ..

lemma real-of-int-le-one-cancel-iff: \(\text{real} (i :: \text{int}) \leq 1\) = \(i \leq 1\)
unfolding real-of-one[symmetric] real-of-int-le-iff ..

lemma real-of-int-abs [simp]: real (abs x) = abs(real (x::int))
by (auto simp add: abs-if)

lemma int-less-real-le: ((n::int) < m) = (real n + 1 <= real m)
  apply (subgoal-tac real n + 1 = real (n + 1))
  apply (simp del: real-of-int-add)
  apply auto
  done

lemma int-le-real-less: ((n::int) <= m) = (real n < real m + 1)
  apply (subgoal-tac real m + 1 = real (m + 1))
  apply (simp del: real-of-int-add)
  apply simp
  done

lemma real-of-int-div-aux: (real (x::int)) / (real d) =
  real (x div d) + (real (x mod d)) / (real d)
proof
  have x = (x div d) * d + x mod d
    by auto
  then have real x = real (x div d) * real d + real(x mod d)
    by (simp only: real-of-int-mult [THEN sym] real-of-int-add [THEN sym])
  then have real x / real d = ... / real d
    by simp
  then show ?thesis
    by (auto simp add: add-divide-distrib algebra-simps)
qed

lemma real-of-int-div: (d :: int) dvd n ==> 
  real(n div d) = real n / real d
  apply (subst real-of-int-div-aux)
  apply simp
  apply (simp add: dvd-eq-mod-eq-0)
  done

lemma real-of-int-div2:
  0 <= real (n::int) / real (x) - real (n div x)
  apply (case-tac x = 0)
  apply simp
  apply (case-tac 0 < x)
  apply (simp add: algebra-simps)
  apply (subst real-of-int-div-aux)
  apply simp
  apply (simp add: algebra-simps)
  apply (subst real-of-int-div-aux)
  apply simp
  apply (simp add: algebra-simps)
  apply (subst real-of-int-div-aux)
  apply simp
  apply (subst zero-le-divide-iff)
  apply auto
  done
lemma real-of-int-div3:
  real (n::int) / real (x) = real (n div x) <= 1
apply (simp add: algebra-simps)
apply (subst real-of-int-div-aux)
apply (auto simp add: divide-le-eq intro: order_less_imp_le)
done

lemma real-of-int-div4:
  real (n div x) <= real (n::int) / real x
by (insert real-of-int-div2 [of n x], simp)

lemma Ints-real-of-int [simp]: real (x::int) ∈ Ints
unfolding real-of-int-def by (rule Ints-of-int)

97.12 Embedding the Naturals into the Reals

lemma real-of-nat-zero [simp]: real (0::nat) = 0
by (simp add: real-of-nat-def)

lemma real-of-nat-1 [simp]: real (1::nat) = 1
by (simp add: real-of-nat-def)

lemma real-of-nat-one [simp]: real (Suc 0) = (1::real)
by (simp add: real-of-nat-def)

lemma real-of-nat-add [simp]: real (m + n) = real (m::nat) + real n
by (simp add: real-of-nat-def)

lemma real-of-nat-Suc: real (Suc n) = real n + (1::real)
by (simp add: real-of-nat-def)

lemma real-of-nat-less-iff [iff]:
  (real (n::nat) < real m) = (n < m)
by (simp add: real-of-nat-def)

lemma real-of-nat-le-iff [iff]: (real (n::nat) ≤ real m) = (n ≤ m)
by (simp add: real-of-nat-def)

lemma real-of-nat-ge-zero [iff]: 0 ≤ real (n::nat)
by (simp add: real-of-nat-def)

lemma real-of-nat-Suc-gt-zero: 0 < real (Suc n)
by (simp add: real-of-nat-def del: of-nat-Suc)

lemma real-of-nat-mult [simp]: real (m * n) = real (m::nat) * real n
by (simp add: real-of-nat-def of-nat-mult)

lemma real-of-nat-power [simp]: real (m ^ n) = real (m::nat) ^ n
by (simp add: real-of-nat-def of-nat-power)
lemmas \text{power-real-of-nat} = \text{real-of-nat-power} \ [\text{symmetric}]

lemma \text{real-of-nat-setsum} [simp]: real \ (\sum x:A. f x)::nat) = \
(\sum x:A. real(f x)) 
apply (subst real-eq-of-nat)+
apply (rule of-nat-setsum)
done

lemma \text{real-of-nat-setprod} [simp]: real \ (\prod x:A. f x)::nat) = \
(\prod x:A. real(f x)) 
apply (subst real-eq-of-nat)+
apply (rule of-nat-setprod)
done

lemma \text{real-of-card}: real \ (\text{card} A) = \text{setsum} \ (\%x.1) A 
apply (subst \text{card-eq-setsum})
apply (subst \text{real-of-nat-setsum})
apply simp
done

lemma \text{real-of-nat-inject} [iff]: (real \ n::nat) = real m = (n = m) 
by (simp add: real-of-nat-def)

lemma \text{real-of-nat-zero-iff} [iff]: (real \ n::nat) = 0 = (n = 0) 
by (simp add: real-of-nat-def)

lemma \text{real-of-nat-diff}: n \leq m ==> real \ (m - n) = real \ m - real n 
by (simp add: add: real-of-nat-def of-nat-diff)

lemma \text{real-of-nat-gt-zero-cancel-iff} [simp]: (0 < real \ n::nat) = (0 < n) 
by (auto simp: real-of-nat-def)

lemma \text{real-of-nat-le-zero-cancel-iff} [simp]: (real \ n::nat) \leq 0 = (n = 0) 
by (simp add: add: real-of-nat-def)

lemma \text{not-real-of-nat-less-zero} [simp]: \sim real \ n::nat) < 0 
by (simp add: add: real-of-nat-def)

lemma \text{nat-less-real-le}: (\text{nat} < m) = (\text{real} n + 1 <= \text{real} m) 
apply (subgoal-tac real n + 1 = real \ Suc n))
apply simp
apply (auto simp: real-of-nat-Suc)
done

lemma \text{nat-le-real-less}: (\text{nat} \leq m) = (\text{real} n < \text{real} m + 1) 
apply (subgoal-tac real m + 1 = real \ Suc m))
apply (simp add: less-Suc-eq-le)
apply (simp add: real-of-nat-Suc)
done

lemma real-of-nat-div-aux: (real (x::nat)) / (real d) =
  real (x div d) + (real (x mod d)) / (real d)
proof -
  have x = (x div d) * d + x mod d
    by auto
  then have real x = real (x div d) * real d + real(x mod d)
    by (simp only: real-of-nat-mult [THEN sym] real-of-nat-add [THEN sym])
  then have real x / real d = ... / real d
    by simp
  then show ?thesis
    by (auto simp add: add-divide-distrib algebra-simps)
qed

lemma real-of-nat-div: (d :: nat) dvd n ==> real(n div d) = real n / real d
proof (subst real-of-nat-div-aux)
  (auto simp add: dvd-eq-mod-eq-0 [symmetric])

lemma real-of-nat-div2:
  0 <= real (n::nat) / real (x) - real (n div x)
proof (simp add: algebra-simps)
apply (subst real-of-nat-div-aux)
apply simp
done

lemma real-of-nat-div3:
  real (n::nat) / real (x) - real (n div x) <= 1
proof (case-tac x = 0)
apply (simp)
apply (simp add: algebra-simps)
apply (subst real-of-nat-div-aux)
apply simp
done

lemma real-of-nat-div4: real (n div x) <= real (n::nat) / real x
proof (insert real-of-nat-div2 [of n x], simp)

lemma real-of-int-of-nat-eq [simp]: real (of-nat n :: int) = real n
proof (simp add: real-of-int-def real-of-nat-def)

lemma real-nat-eq-real [simp]: 0 <= x ==> real(nat x) = real x
proof (subgoal_tac real(int(nat x)) = real(nat x))
apply force
apply (simp only: real-of-int-of-nat-eq)
done

lemma Nats-real-of-nat [simp]: real (n::nat) ∈ Nats
unfolding real-of-nat-def by (rule of-nat-in-Nats)

**Lemma** Ints-real-of-nat [simp]: real (n::nat) ∈ Ints
unfolding real-of-nat-def by (rule Ints-of-nat)

### 97.13 The Archimedean Property of the Reals

**Theorem** reals-Archimedean:
assumes x-pos: 0 < x
shows ∃ n. inverse (real (Suc n)) < x
unfolding real-of-nat-def using x-pos
by (rule ex-inverse-of-nat-Suc-less)

**Lemma** reals-Archimedean2: ∃ n. (x::real) < real (n::nat)
unfolding real-of-nat-def by (rule ex-less-of-nat)

**Lemma** reals-Archimedean3:
assumes x-greater-zero: 0 < x
shows ∀ (y::real). ∃ (n::nat). y < real n * x
unfolding real-of-nat-def using (0 < x)
by (auto intro: ex-less-of-nat-mult)

### 97.14 Rationals

**Lemma** Rats-real-nat [simp]: real (n::nat) ∈ Q
by (simp add: real-eq-of-nat)

**Lemma** Rats-eq-int-div-int:
Q = { real(i::int)/real(j::int) | i j ≠ 0} (is ::= ?S)
proof
show Q ⊆ ?S
proof
  fix x::real assume x : Q
  then obtain r where x = of-rat r unfolding Rats-def ..
  have of-rat r : ?S
    by (cases r)(auto simp add:of-rat-rat real-eq-of-int)
  thus x : ?S using (x = of-rat r) by simp
qed
next
show ?S ⊆ Q
proof(auto simp:Rats-def)
  fix i j :: int assume j ≠ 0
  hence real i / real j = of-rat(Fract i j)
    by (simp add:of-rat-rat real-eq-of-int)
  thus real i / real j ∈ range of-rat by blast
qed
qed

**Lemma** Rats-eq-int-div-nat:
Q = { real(i::int)/real(n::nat) | i n. n ≠ 0}
proof(auto simp:Rats-eq-int-div-int)
fix i j::int assume j ≠ 0
show EX (i':int) (n::nat). real i/real j = real i'/real n ∧ 0<n
proof cases
assume j>0
hence real i/real j = real i/(real(nat j) ∧ 0<nat j)
  by (simp add: real-eq-of-int real-eq-of-nat of-nat-nat)
thus ?thesis by blast
next
assume ~ j>0
hence real i/real j = real(-i)/(real(nat(-j)) ∧ 0<nat(-j) using ⟨j≠0⟩)
  by (simp add: real-eq-of-int real-eq-of-nat of-nat-nat)
thus ?thesis by blast
qed
next
fix i::int and n::nat assume 0 < n
hence real i/real n = real i/real(int n) ∧ int n ≠ 0 by simp
thus ∃(i':int) j::int. real i/real n = real i'/real j ∧ j ≠ 0 by blast
qed

lemma Rats-abs-nat-div-natE:
  assumes x ∈ ℚ
  obtains m n :: nat where n ≠ 0 and |x| = real m / real n and gcd m n = 1
proof -
  from ⟨x ∈ ℚ⟩ obtain i::int and n::nat where n ≠ 0 and x = real i / real n
    by(auto simp add: Rats-eq-int-div-nat)
hence |x| = real(nat(abs i)) / real n by simp
then obtain m :: nat where x-rat: |x| = real m / real n by blast
let ?gcd = gcd m n
from ⟨n≠0⟩ have gcd: ?gcd ≠ 0 by simp
let ?k = m div ?gcd
let ?l = n div ?gcd
let ?gcd' = gcd ?k ?l
have ?gcd dvd m .. then have gcd-k: ?gcd * ?k = m
  by (rule ded-mult-div-cancel)
have ?gcd dvd n .. then have gcd-l: ?gcd * ?l = n
  by (rule ded-mult-div-cancel)
from ⟨n≠0⟩ and gcd-l have ?l ≠ 0 by (auto iff del: neq0-conv)
moreover
have |x| = real ?k / real ?l
proof -
  from gcd have real ?k / real ?l =
    real (gcd * ?k) / real (gcd * ?l) by simp
  also from gcd-k and gcd-l have ... = real m / real n by simp
  also from x-rat have ... = |x| ..
  finally show ?thesis ..
qed
moreover
97.15 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is Real Analysis by H.L. Royden. It employs the Archimedean property of the reals.

lemma Rats-dense-in-real:
  fixes x :: real
  assumes x < y shows ∃r ∈ Q. x < r ∧ r < y
proof –
  from ⟨x < y⟩ have 0 < y − x by simp
  with reals-Archimedean obtain q :: nat
    where q: inverse (real q) < y − x and 0 < q by auto
  def p ≡ ceiling (y * real q) − 1
  def r ≡ of-int p / real q
  from q have x < y − inverse (real q) by simp
  also have y − inverse (real q) ≤ r
    unfolding r-def p-def
    by (simp add: le-divide-eq left-diff-distrib le-of-int-ceiling ⟨0 < q⟩)
  finally have x < r .
  moreover have r < y
    unfolding r-def p-def
    by (simp add: divide-less-eq diff-less-eq ⟨0 < q⟩
      less-ceiling-iff [symmetric])
  moreover from r-def have r ∈ Q by simp
  ultimately show ?thesis by fast
qed

lemma of-rat-dense:
  fixes x y :: real
  assumes x < y
  shows ∃q :: rat. x < of-rat q ∧ of-rat q < y
using Rats-dense-in-real [OF ⟨x < y⟩]
by (auto elim: Rats-cases)

97.16 Numerals and Arithmetic

lemma [code-abbrev]:
  real-of-int (numeral k) = numeral k
  real-of-int (− numeral k) = − numeral k
by simp-all

Collapse applications of real to numeral

lemma real-numeral [simp]:
  real (numeral v :: int) = numeral v
  real (- numeral v :: int) = - numeral v
by (simp-all add: real-of-int-def)

lemma real-of-nat-numeral [simp]:
  real (numeral v :: nat) = numeral v
by (simp add: real-of-int-def)

declaration ⟨⟨
  K (Lin-Arith.add-inj-thms [@{thm real-of-nat-le-iff} RS iffD2, @{thm real-of-nat-inject} RS iffD2]
  (* not needed because x < (y::nat) can be rewritten as Suc x <= y: real-of-nat-less-iff
  RS iffD2 *)
  #> Lin-Arith.add-inj-thms [@{thm real-of-int-le-iff} RS iffD2, @{thm real-of-int-inject} RS iffD2]
  (* not needed because x < (y::int) can be rewritten as x + 1 <= y: real-of-int-less-iff
  RS iffD2 *)
  @{thm real-of-int-add}, @{thm real-of-int-minus}, @{thm real-of-int-diff},
  @{thm real-of-int-mult}, @{thm real-of-int-eq},
  @{thm real-of-nat-numeral}, @{thm real-numeral(1)}, @{thm real-numeral(2)}]
  #> Lin-Arith.add-inj-const (@{const-name real}, @{typ nat => real})
  #> Lin-Arith.add-inj-const (@{const-name real}, @{typ int => real})⟩⟩

97.17 Simprules combining x+y and 0: ARE THEY NEEDED?

lemma real-add-minus-iff [simp]: (x + - a = (0::real)) = (x=a)
by arith

FIXME: redundant with add-eq-0-iff below

lemma real-add-eq-0-iff: (x+y = (0::real)) = (y = -x)
by auto

lemma real-add-less-0-iff: (x+y < (0::real)) = (y < -x)
by auto

lemma real-0-less-add-iff: ((0::real) < x+y) = (-x < y)
by auto

lemma real-add-le-0-iff: (x+y ≤ (0::real)) = (y ≤ -x)
by auto
lemma real-0-le-add-iff: \((0::\text{real}) \leq x+y) = (−x \leq y)
by auto

97.18 Lemmas about powers

FIXME: declare this in Rings.thy or not at all
declare abs-mult-self [simp]

lemma two-realpow-ge-one: \((1::\text{real}) \leq 2 ^ n)
by simp

lemma two-realpow-gt [simp]: \(\text{real} \ (n::\text{nat}) < 2 ^ n)
apply (induct n)
apply (auto simp add: real-of-nat-Suc)
apply (subst mult-2)
apply (erule add-less-le-mono)
apply (rule two-realpow-ge-one)
done

TODO: no longer real-specific; rename and move elsewhere
lemma realpow-Suc-le-self:
  fixes \(r :: 'a::\text{linordered-semidom}
  shows \[0 \leq r; r \leq 1\] \equiv r ^ Suc n \leq r
by (insert power-decreasing [of 1 Suc n r], simp)

TODO: no longer real-specific; rename and move elsewhere
lemma realpow-minus-mult:
  fixes \(x :: 'a::\text{monoid-mult}
  shows 0 < n \Rightarrow x ^ (n-1) * x = x ^ n
by (simp add: power-commutes split add: nat-diff-split)

FIXME: declare this [simp] for all types, or not at all
lemma real-two-squares-add-zero-iff [simp]:
\((x * x + y * y = 0) = ((x::\text{real}) = 0 \land y = 0)
by (rule sum-squares-eq-zero-iff)

FIXME: declare this [simp] for all types, or not at all
lemma realpow-two-sum-zero-iff [simp]:
\((x^2 + y^2 = (0::\text{real})) = (x = 0 \& y = 0)
by (rule sum-power2-eq-zero-iff)

lemma real-minus-mult-self-le [simp]: −(u * u) \leq (x * (x::\text{real}))
by (rule-tac y = 0 in order-trans, auto)

lemma realpow-square-minus-le [simp]: −u ^ 2 \leq (x::\text{real}) ^ 2
by (auto simp add: power2-eq-square)
lemma numeral-power-le-real-of-nat-cancel-iff [simp]:
(numeral x::real) ^ n <= real a <-> (numeral x::nat) ^ n <= a
unfolding real-of-nat-le-iff[symmetric] by simp

lemma real-of-nat-le-numeral-power-cancel-iff [simp]:
real a <= (numeral x::real) ^ n <-> a <= (numeral x::nat) ^ n
unfolding real-of-nat-le-iff[symmetric] by simp

lemma numeral-power-le-real-of-int-cancel-iff [simp]:
(numeral x::real) ^ n <= real a <-> (numeral x::int) ^ n <= a
unfolding real-of-int-le-iff[symmetric] by simp

lemma real-of-int-le-numeral-power-cancel-iff [simp]:
real a <= (numeral x::real) ^ n <-> a <= (numeral x::int) ^ n
unfolding real-of-int-le-iff[symmetric] by simp

lemma neg-numeral-power-le-real-of-int-cancel-iff [simp]:
(- numeral x::real) ^ n <= real a <-> (- numeral x::int) ^ n <= a
unfolding real-of-int-le-iff[symmetric] by simp

lemma real-of-int-le-neg-numeral-power-cancel-iff [simp]:
real a <= (- numeral x::real) ^ n <-> a <= (- numeral x::int) ^ n
unfolding real-of-int-le-iff[symmetric] by simp

97.19 Density of the Reals

lemma real-lbound-gt-zero:
|- |0::real| < d1; 0 < d2 |===> \exists e. 0 < e & e < d1 & e < d2
apply (rule-tac x = (min d1 d2) /2 in exI)
apply (simp add: min-def)
done

Similar results are proved in Fields

lemma real-less-half-sum: x < y ==> x < (x+y) / (2::real)
by auto

lemma real-gt-half-sum: x < y ==> (x+y)/(2::real) < y
by auto

lemma real-sum-of-halves: x/2 + x/2 = (x::real)
by simp

97.20 Absolute Value Function for the Reals

lemma abs-minus-add-cancel: abs(x + (-y)) = abs (y + (-x::real))
by (simp add: abs-iff)
lemma abs-le-interval-iff: \((\text{abs } x \leq r) = (-r \leq x \& x \leq (r :: \text{real}))\)
by (force simp add: abs-le-iff)

lemma abs-add-one-gt-zero: \((0 :: \text{real}) < 1 + \text{abs}(x)\)
by (simp add: abs-if)

lemma abs-real-of-nat-cancel [simp]: \(\text{abs}(\text{real } x) = \text{real}(x :: \text{nat})\)
by (rule abs-of-nonneg [OF real-of-nat-ge-zero])

lemma abs-add-one-not-less-self: \(\neg \text{abs}(x) + (1 :: \text{real}) < x\)
by simp

lemma abs-sum-triangle-ineq: \(\text{abs}((x :: \text{real}) + y + (-1 - m)) \leq \text{abs}(x + l) + \text{abs}(y + -m)\)
by simp

97.21 Floor and Ceiling Functions from the Reals to the Integers

lemma numeral-less-real-of-int-iff [simp]:
\(((\text{numeral } n) < \text{real}(m :: \text{int})) = (\text{numeral } n < m)\)
apply auto
apply (rule real-of-int-less-iff [THEN iffD1])
apply (drule-tac [2] real-of-int-less-iff [THEN iffD2], auto)
done

lemma numeral-less-real-of-int-iff2 [simp]:
\((\text{real}(m :: \text{int}) < (\text{numeral } n)) = (m < \text{numeral } n)\)
apply auto
apply (rule real-of-int-less-iff [THEN iffD1])
apply (drule-tac [2] real-of-int-less-iff [THEN iffD2], auto)
done

lemma real-of-nat-less-numeral-iff [simp]:
\(\text{real}(n :: \text{nat}) < \text{numeral } w \longleftrightarrow n < \text{numeral } w\)
using real-of-nat-less-iff[of n numeral w] by simp

lemma numeral-less-real-of-nat-iff [simp]:
\(\text{numeral } w < \text{real}(n :: \text{nat}) \longleftrightarrow \text{numeral } w < n\)
using real-of-nat-less-iff[of numeral w n] by simp

lemma numeral-le-real-of-int-iff [simp]:
\(((\text{numeral } n) \leq \text{real}(m :: \text{int})) = (\text{numeral } n \leq m)\)
by (simp add: linorder-not-less [symmetric])

lemma numeral-le-real-of-int-iff2 [simp]:
\((\text{real}(m :: \text{int}) \leq (\text{numeral } n)) = (m \leq \text{numeral } n)\)
by (simp add: linorder-not-less [symmetric])
lemma floor-real-of-nat [simp]: floor (real (n::nat)) = int n
unfolding real-of-nat-def by simp

lemma floor-minus-real-of-nat [simp]: floor (− real (n::nat)) = − int n
unfolding real-of-nat-def by (simp add: floor-minus)

lemma floor-real-of-int [simp]: floor (real (n::int)) = n
unfolding real-of-int-def by simp

lemma floor-minus-real-of-int [simp]: floor (− real (n::int)) = − n
unfolding real-of-int-def by (simp add: floor-minus)

lemma real-lb-ub-int: ∃ n::int. real n ≤ r & r < real (n+1)
unfolding real-of-int-def by (rule floor-exists)

lemma lemma-floor:
asumes a1: real m ≤ r and a2: r < real n + 1
shows m ≤ (n::int)
proof −
have real m < real n + 1 using a1 a2 by (rule order-le-less-trans)
also have ... = real (n + 1) by simp
finally have m < n + 1 by (simp only: real-of-int-less-iff)
thus thesis by arith
qed

lemma real-of-int-floor-le [simp]: real (floor r) ≤ r
unfolding real-of-int-def by (rule of-int-floor-le)

lemma lemma-floor2: real n < real (x::int) + 1 ==> n ≤ x
by (auto intro: lemma-floor)

lemma real-of-int-floor-cancel [simp]:
(real (floor x) = x) = (∃ n::int. x = real n)
using floor-real-of-int by metis

lemma floor-eq: [| real n < x; x < real n + 1 |] ==> floor x = n
unfolding real-of-int-def using floor-unique [of n x] by simp

lemma floor-eq2: [| real n ≤ x; x < real n + 1 |] ==> floor x = n
unfolding real-of-int-def by (rule rule-finite)

lemma floor-eq3: [| real n < x; x < real (Suc n) |] ==> nat(floor x) = n
apply (rule inj-int [THEN injD])
apply (simp add: real-of-nat-Suc)
apply (simp add: real-of-nat-Suc floor-eq floor-eq [where n = int n])
done

lemma floor-eq4: [| real n ≤ x; x < real (Suc n) |] ==> nat(floor x) = n
apply (erule order-imp-less-or-eq)
apply (auto intro: floor-eq3)
done

lemma real-of-int-floor-ge-diff-one [simp]: \( r - 1 \leq \text{real}(\text{floor } r) \)
unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-gt-diff-one [simp]: \( r - 1 < \text{real}(\text{floor } r) \)
unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-add-one-ge [simp]: \( r \leq \text{real}(\text{floor } r) + 1 \)
unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-add-one-gt [simp]: \( r < \text{real}(\text{floor } r) + 1 \)
unfolding real-of-int-def using floor-correct [of r] by simp

lemma le-floor: real a \leq x ==> a \leq floor x
unfolding real-of-int-def by (simp add: le-floor-iff)

lemma real-le-floor: a \leq floor x ==> real a \leq x
unfolding real-of-int-def by (simp add: le-floor-iff)

lemma le-floor-eq: (a \leq floor x) = (real a \leq x)
unfolding real-of-int-def by (rule le-floor-iff)

lemma floor-less-eq: (floor x < a) = (x < real a)
unfolding real-of-int-def by (rule floor-less-iff)

lemma less-floor-eq: (a < floor x) = (real a + 1 \leq x)
unfolding real-of-int-def by (rule less-floor-iff)

lemma floor-le-eq: (floor x <= a) = (x < real a + 1)
unfolding real-of-int-def by (rule floor-le-iff)

lemma floor-add [simp]: floor (x + real a) = floor x + a
unfolding real-of-int-def by (rule floor-add-of-int)

lemma floor-subtract [simp]: floor (x - real a) = floor x - a
unfolding real-of-int-def by (rule floor-diff-of-int)

lemma le-mult-floor:
  assumes \( 0 \leq (a :: \text{real}) \) and \( 0 \leq b \)
  shows floor a * floor b \leq floor (a * b)

proof
  have real (floor a) \leq a
    and real (floor b) \leq b by auto
  hence real (floor a * floor b) \leq a * b
    using assms by (auto intro!: mult-mono)
  also have a * b < real (floor (a * b) + 1) by auto
  finally show \(?thesis unfolding real-of-int-less-iff by simp\)
lemma floor-divide-eq-div: 
floor (real a / real b) = a div b

proof cases
  assume b ≠ 0 ∨ b dvd a
  with real-of-int-div3[of a b] show ?thesis
    (metis add-left-cancel zero-neq-one real-of-int-div-aux real-of-int-inject
     real-of-int-zero-cancel right-inverse-eq div-self mod-div-trivial))

qed (auto simp: real-of-int-div)

lemma ceiling-real-of-nat [simp]: ceiling (real (n::nat)) = int n
  unfolding real-of-nat-def by simp

lemma real-of-int-ceiling-ge [simp]: r ≤ real (ceiling r)
  unfolding real-of-int-def by (rule le-of-int-ceiling)

lemma ceiling-real-of-int [simp]: ceiling (real (n::int)) = n
  unfolding real-of-int-def by simp

lemma real-of-int-ceiling-cancel [simp]:
  (real (ceiling x) = x) = (∃ n::int. x = real n)
  using ceiling-real-of-int by metis

lemma ceiling-eq: [| real n < x; x < real n + 1 |] ==> ceiling x = n + 1
  unfolding real-of-int-def using ceiling-unique [of n + 1 x] by simp

lemma ceiling-eq2: [| real n < x; x ≤ real n + 1 |] ==> ceiling x = n + 1
  unfolding real-of-int-def using ceiling-unique [of n + 1 x] by simp

lemma ceiling-eq3: [| real n - 1 < x; x ≤ real n |] ==> ceiling x = n
  unfolding real-of-int-def using ceiling-unique [of n x] by simp

lemma real-of-int-ceiling-diff-one-le [simp]: real (ceiling r) - 1 ≤ r
  unfolding real-of-int-def using ceiling-correct [of r] by simp

lemma real-of-int-ceiling-le-add-one [simp]: real (ceiling r) ≤ r + 1
  unfolding real-of-int-def using ceiling-correct [of r] by simp

lemma ceiling-le: x <= real a ==> ceiling x <= a
  unfolding real-of-int-def by (simp add: ceiling-le-iff)

lemma ceiling-le-real: ceiling x <= a ==> x <= real a
  unfolding real-of-int-def by (simp add: ceiling-le-iff)

lemma ceiling-le-eq: (ceiling x <= a) = (x <= real a)
  unfolding real-of-int-def by (rule ceiling-le-iff)
lemma less-ceiling-eq: \((a < \text{ceiling } x) = (\text{real } a < x)\)
unfolding real-of-int-def by (rule less-ceiling-iff)

lemma ceiling-less-eq: \((\text{ceiling } x < a) = (x \leq \text{real } a - 1)\)
unfolding real-of-int-def by (rule ceiling-less-iff)

lemma le-ceiling-eq: \((a \leq \text{ceiling } x) = (\text{real } a - 1 < x)\)
unfolding real-of-int-def by (rule le-ceiling-iff)

lemma ceiling-add [simp]: \(\text{ceiling } (x + \text{real } a) = \text{ceiling } x + a\)
unfolding real-of-int-def by (rule ceiling-add-of-int)

lemma ceiling-subtract [simp]: \(\text{ceiling } (x - \text{real } a) = \text{ceiling } x - a\)
unfolding real-of-int-def by (rule ceiling-diff-of-int)

97.21.1 Versions for the natural numbers

definition
natfloor :: real \(\Rightarrow\) nat where
natfloor \(x = \text{nat}(\text{floor } x)\)

definition
natceiling :: real \(\Rightarrow\) nat where
natceiling \(x = \text{nat}(\text{ceiling } x)\)

lemma natfloor-zero [simp]: natfloor \(0 = 0\)
by (unfold natfloor-def, simp)

lemma natfloor-one [simp]: natfloor \(1 = 1\)
by (unfold natfloor-def, simp)

lemma zero-le-natfloor [simp]: \(0 \leq \text{natfloor } x\)
by (unfold natfloor-def, simp)

lemma natfloor-numeral-eq [simp]: natfloor \((\text{numeral } n) = \text{numeral } n\)
by (unfold natfloor-def, simp)

lemma natfloor-real-of-nat [simp]: natfloor(\text{real } n) = n
by (unfold natfloor-def, simp)

lemma real-natfloor-le: \(0 \leq x \implies \text{real}(\text{natfloor } x) \leq x\)
by (unfold natfloor-def, simp)

lemma natfloor-neg: \(x \leq 0 \implies \text{natfloor } x = 0\)
unfolding natfloor-def by simp

lemma natfloor-mono: \(x \leq y \implies \text{natfloor } x \leq \text{natfloor } y\)
unfolding natfloor-def by (intro nat-mono floor-mono)
lemma le-natfloor: real x <= a ==> x <= natfloor a
  apply (unfold natfloor-def)
  apply (subst nat-int [THEN sym])
  apply (rule nat-mono)
  apply (rule le-floor)
  apply simp
  done

lemma natfloor-less-iff: 0 <= x ==> natfloor x < n <-> x < real n
  unfolding natfloor-def real-of-nat-def
  by (simp add: nat-less-iff floor-less-iff)

lemma less-natfloor:
  assumes 0 <= x and x < real (n :: nat)
  shows natfloor x < n
  using assms by (simp add: natfloor-less-iff)

lemma le-natfloor-eq: 0 <= x ==> (a <= natfloor x) = (real a <= x)
  apply (rule iffI)
  apply (rule order-trans)
  prefer 2
  apply (erule real-natfloor-le)
  apply (rule natfloor-less-iff)
  apply assumption
  apply (erule le-natfloor)
  done

lemma le-natfloor-eq-numeral [simp]:
  ~ neg((numeral n)::int) ==> 0 <= x ==> (numeral n <= natfloor x) = (numeral n <= x)
  apply (subgoal_tac le-natfloor-eq, assumption)
  apply (rule natfloor-less-iff)
  done

lemma le-natfloor-eq-one [simp]: (1 <= natfloor x) = (1 <= x)
  apply (case-tac 0 <= x)
  apply (rule le-natfloor-eq, assumption, simp)
  apply (rule iffI)
  apply (subgoal-tac natfloor x <= natfloor 0)
  apply simp
  apply (rule natfloor-mono)
  apply simp
  apply simp
  done

lemma natfloor-eq: real n <= x ==> x < real n + 1 ==> natfloor x = n
  unfolding natfloor-def by (simp add: floor-eq2 [where n=int n])

lemma real-natfloor-add-one-gt: x < real(natfloor x) + 1
apply (case-tac 0 <= x)
apply (unfold natfloor-def)
apply simp
apply simp-all
done

lemma real-natfloor-gt-diff-one:  x - 1 < real(natfloor x)
using real-natfloor-add-one-gt by (simp add: algebra-simps)

lemma ge-natfloor-plus-one-imp-gt:  natfloor z + 1 <= n == z < real n
apply (subgoal-tac z < real(natfloor z) + 1)
apply arith
apply (rule real-natfloor-add-one-gt)
done

lemma natfloor-add [simp]:  0 <= x ==> natfloor (x + real a) = natfloor x + a
unfolding natfloor-def
unfolding real-of-int-of-nat-eq [symmetric] floor-add
by (simp add: nat-add-distrib)

lemma natfloor-add-numeral [simp]:
  ~neg ((numeral n)::int) ==> 0 <= x ==>
  natfloor (x + numeral n) = natfloor x + numeral n
by (simp add: natfloor-add [symmetric])

lemma natfloor-add-one:  0 <= x ==> natfloor(x + 1) = natfloor x + 1
by (simp add: natfloor-add [symmetric] del: One-nat-def)

lemma natfloor-subtract [simp]:
  natfloor(x - real a) = natfloor x - a
unfolding natfloor-def real-of-int-of-nat-eq [symmetric] floor-subtract
by simp

lemma natfloor-div-nat:
  assumes 1 <= x and y > 0
  shows natfloor (x / real y) = natfloor x div y
proof (rule natfloor-eq)
  have (natfloor x) div y * y <= natfloor x
    by (rule add-leD1 [where k=natfloor x mod y], simp)
  thus real (natfloor x div y) <= x / real y
    using assms by (simp add: le-divide-eq le-natfloor-eq)
  have natfloor x < (natfloor x) div y * y + y
    apply (subst mod-div-equality [symmetric])
    apply (rule add-strict-left-mono)
    apply (rule mod-less-divisor)
    apply fact
    done
  thus x / real y < real (natfloor x div y) + 1
    using assms

by (simp add: divide-less-eq natfloor-less-iff distrib-right)

qed

lemma le-mult-natfloor:
  shows natfloor a * natfloor b ≤ natfloor (a * b)
  by (cases 0 <= a & 0 <= b)
    (auto simp add: le-natfloor-eq mult-mono' real-natfloor-le natfloor-neg)

lemma natceiling-zero [simp]: natceiling 0 = 0
  by (unfold natceiling-def, simp)

lemma natceiling-one [simp]: natceiling 1 = 1
  by (unfold natceiling-def, simp)

lemma zero-le-natceiling [simp]: 0 <= natceiling x
  by (unfold natceiling-def, simp)

lemma natceiling-numeral-eq [simp]: natceiling (numeral n) = numeral n
  by (unfold natceiling-def, simp)

lemma natceiling-real-of-nat [simp]: natceiling(real n) = n
  by (unfold natceiling-def, simp)

lemma real-natceiling-ge: x <= real(natceiling x)
  unfolding natceiling-def by (cases x < 0, simp-all)

lemma natceiling-neg: x <= 0 ==> natceiling x = 0
  unfolding natceiling-def by simp

lemma natceiling-mono: x <= y ==> natceiling x <= natceiling y
  unfolding natceiling-def by (intro nat-mono ceiling-mono)

lemma natceiling-le: x <= real a ==> natceiling x <= a
  unfolding natceiling-def real-of-nat-def
  by (simp add: nat-le-iff ceiling-le-iff)

lemma natceiling-le-eq: (natceiling x <= a) = (x <= real a)
  unfolding natceiling-def real-of-nat-def
  by (simp add: nat-le-iff ceiling-le-iff)

lemma natceiling-le-eq-numeral [simp]:
  ~ neg((numeral n)::int) ==> (natceiling x <= numeral n) = (x <= numeral n)
  by (simp add: natceiling-le-eq)

lemma natceiling-le-eq-one: (natceiling x <= 1) = (x <= 1)
  unfolding natceiling-def
  by (simp add: nat-le-iff ceiling-le-iff)
lemma natceiling-eq: real n < x ==> x <= real n + 1 ==> natceiling x = n + 1
  unfolding natceiling-def
  by (simp add: ceiling-eq [where n=int n])

lemma natceiling-add [simp]: 0 <= x ==> natceiling (x + real a) = natceiling x + a
  unfolding natceiling-def
  unfolding real-of-int-of-nat-eq [symmetric] ceiling-add
  by (simp add: nat-add-distrib)

lemma natceiling-add-numeral [simp]:
  ~ neg ((numeral n)::int) ==> 0 <= x ==> natceiling (x + numeral n) = natceiling x + numeral n
  by (simp add: natceiling-add [symmetric])

lemma natceiling-add-one: 0 <= x ==> natceiling(x + 1) = natceiling x + 1
  by (simp add: natceiling-add [symmetric] del: One-nat-def)

lemma natceiling-subtract [simp]: natceiling(x - real a) = natceiling x - a
  unfolding natceiling-def real-of-int-of-nat-eq [symmetric] ceiling-subtract
  by simp

lemma Rats-no-top-le: \exists q \in \mathbb{Q}. (x :: real) <= q
  by (auto intro!: bexI [of - of-nat (natceiling x)]
      (metis real-natceiling-ge real-of-nat-def))

lemma Rats-no-bot-less: \exists q \in \mathbb{Q}. q < (x :: real)
  apply (auto intro!: bexI [of - of-int (floor x - 1)])
  apply (rule less-le-trans[OF - of-int-floor-le])
  apply simp
  done

97.22 Exponentiation with floor

lemma floor-power:
  assumes x = real (floor x)
  shows floor (x ^ n) = floor x ^ n
proof
  from assms have *: x ^ n = real (floor x ^ n)
    using assms by (induct n arbitrary: x) simp-all
  show ?thesis unfolding real-of-int-inject [symmetric]
    unfolding * floor-real-of-int ..
qed

lemma natfloor-power:
  assumes x = real (natfloor x)
  shows natfloor (x ^ n) = natfloor x ^ n
proof
  from assms have 0 <= floor x by auto
97.23 Implementation of rational real numbers

Formal constructor

definition Ratreal :: rat ⇒ real where
  [code-abbrev, simp]: Ratreal = of-rat

code-datatype Ratreal

Numerals

lemma [code-abbrev]:
  (of-rat (of-int a) :: real) = of-int a
  by simp

lemma [code-abbrev]:
  (of-rat 0 :: real) = 0
  by simp

lemma [code-abbrev]:
  (of-rat 1 :: real) = 1
  by simp

lemma [code-abbrev]:
  (of-rat (numeral k) :: real) = numeral k
  by simp

lemma [code-abbrev]:
  (of-rat (− numeral k) :: real) = − numeral k
  by simp

lemma [code-post]:
  (of-rat (0 / of rat) :: real) = 0
  (of-rat (r / 0) :: real) = 0
  (of-rat (1 / 1) :: real) = 1
  (of-rat (numeral k / 1) :: real) = numeral k
  (of-rat (− numeral k / 1) :: real) = − numeral k
  (of-rat (1 / numeral k) :: real) = 1 / numeral k
  (of-rat (1 / − numeral k) :: real) = 1 / − numeral k
  (of-rat (numeral k / numeral l) :: real) = numeral k / numeral l
  (of-rat (− numeral k / numeral l) :: real) = − numeral k / numeral l
  (of-rat (− numeral k / − numeral l) :: real) = − numeral k / − numeral l
  by (simp-all add: of-rat-divide of-rat-minus)
Operations

lemma zero-real-code [code]:
  0 = Ratreal 0
by simp

lemma one-real-code [code]:
  1 = Ratreal 1
by simp

instantiation real :: equal
begin

definition HOL.equal (x::real) y ←→ x – y = 0

instance proof
  qed (simp add: equal-real-def)

lemma real-equal-code [code]:
  HOL.equal (Ratreal x) (Ratreal y) ←→ HOL.equal x y
by (simp add: equal-real-def equal)

lemma [code nbe]:
  HOL.equal (x::real) x ←→ True
by (rule equal-refl)

end

lemma real-less-eq-code [code]: Ratreal x ≤ Ratreal y ←→ x ≤ y
by (simp add: of-rat-less-eq)

lemma real-less-code [code]: Ratreal x < Ratreal y ←→ x < y
by (simp add: of-rat-less)

lemma real-plus-code [code]: Ratreal x + Ratreal y = Ratreal (x + y)
by (simp add: of-rat-add)

lemma real-times-code [code]: Ratreal x * Ratreal y = Ratreal (x * y)
by (simp add: of-rat-mult)

lemma real-uminus-code [code]: – Ratreal x = Ratreal (– x)
by (simp add: of-rat-minus)

lemma real-minus-code [code]: Ratreal x – Ratreal y = Ratreal (x – y)
by (simp add: of-rat-diff)

lemma real-inverse-code [code]: inverse (Ratreal x) = Ratreal (inverse x)
by (simp add: of-rat-inverse)

lemma real-divide-code [code]: Ratreal x / Ratreal y = Ratreal (x / y)
by (simp add: of-rat-divide)

lemma real-floor-code [code]: floor (Ratreal x) = floor x
  by (metis Ratreal-def floor-le-iff floor-unique le-floor-iff of-int-floor-le of-rat-of-int-eq
       real-less-eq-code)

Quickcheck

definition (in term-syntax)
  term)
  where
    [code-unfold]: valterm-ratreal k = Code-Evaluation.valtermify Ratreal {·} k

notation fcomp (infixl ◦ 60)
notation scomp (infixl ◦→ 60)

instantiation real :: random
begin

definition
  Quickcheck-Random.random i = Quickcheck-Random.random i ◦→ (λr. Pair (valterm-ratreal r))

instance ..

end

no-notation fcomp (infixl ◦ 60)
note-notation scomp (infixl ◦→ 60)

instantiation real :: exhaustive
begin

definition
  exhaustive-real f d = Quickcheck-Exhaustive.exhaustive (%r. f (Ratreal r)) d

instance ..

end

instantiation real :: full-exhaustive
begin

definition
  full-exhaustive-real f d = Quickcheck-Exhaustive.full-exhaustive (%r. f (valterm-ratreal r)) d

instance ..

end
instantiation real :: narrowing
begin

definition
  narrowing = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons Ratreal) narrowing

instance ..
end

97.24 Setup for Nitpick

declaration ⟨⟨ Nitpick-HOL.register-frac-type @{type-name real}
  [[@{const-name zero-real-inst.zero-real}, @{const-name Nitpick.zero-frac}],
   (@{const-name one-real-inst.one-real}, @{const-name Nitpick.one-frac}),
   (@{const-name plus-real-inst.plus-real}, @{const-name Nitpick.plus-frac}),
   (@{const-name times-real-inst.times-real}, @{const-name Nitpick.times-frac}),
   (@{const-name uminus-real-inst.uminus-real}, @{const-name Nitpick.uminus-frac}),
   (@{const-name inverse-real-inst.inverse-real}, @{const-name Nitpick.inverse-frac}),
   (@{const-name ord-real-inst.less-real}, @{const-name Nitpick.less-frac}),
   (@{const-name ord-real-inst.less-eq-real}, @{const-name Nitpick.less-eq-frac})]
⟩⟩

lemmas [nitpick-unfold] = inverse-real-inst.inverse-real one-real-inst.one-real
  ord-real-inst.less-real ord-real-inst.less-eq-real plus-real-inst.plus-real
  times-real-inst.times-real uminus-real-inst.uminus-real
  zero-real-inst.zero-real

97.25 Setup for SMT

ML-file Tools/SMT/smt-real.ML
setup SMT-Real.setup
ML-file Tools/SMT2/smt2-real.ML
ML-file Tools/SMT2/z3-new-real.ML

lemma [z3-new-rule]:
  0 + (x::real) = x
  x + 0 = x
  0 * x = 0
  1 * x = x
  x + y = y + x
  by auto
98  Topological-Spaces: Topological Spaces

theory Topological-Spaces
imports Main Conditionally-Complete-Lattices
begin

ML ⟨⟨
structure Continuous-Intros = Named-Thms
(  val name = @{binding continuous-intros}
   val description = Structural introduction rules for continuity
)
⟩⟩

setup Continuous-Intros.

98.1  Topological-Space

class open =
  fixes open :: 'a set ⇒ bool

class topological-space = open +
  assumes open-UNIV [simp, intro]: open UNIV
  assumes open-Int [intro]: open S ⇒ open T ⇒ open (S ∩ T)
  assumes open-Union [intro]: ∀ S∈K. open S ⇒ open (∪ K)
begin

definition
  closed :: 'a set ⇒ bool where
  closed S ⇔ open (¬ S)

lemma open-empty [continuous-intros, intro, simp]: open {}
  using open-Union [of {}] by simp

lemma open-Un [continuous-intros, intro]: open S ⇒ open T ⇒ open (S ∪ T)
  using open-Union [of {S, T}] by simp

lemma open-UN [continuous-intros, intro]: ∀ x∈A. open (B x) ⇒ open (∪ x∈A. B x)
  using open-Union [of B ' A] by simp

lemma open-Inter [continuous-intros, intro]: finite S ⇒ ∀ T∈S. open T ⇒ open (∩ S)
  by (induct set: finite) auto

lemma open-INT [continuous-intros, intro]: finite A ⇒ ∀ x∈A. open (B x) ⇒
open (∩ x∈A. B x)
  using open-Inter [of B ' A] by simp
lemma openI:  
assumes \( \forall x. x \in S \Rightarrow \exists T. \text{open } T \land x \in T \land T \subseteq S \)  
shows open S  
proof –  
have open \( \bigcup \{ T. \text{open } T \land T \subseteq S \} \) by auto  
moreover have \( \bigcup \{ T. \text{open } T \land T \subseteq S \} = S \) by (auto dest: assms)  
ultimately show open S by simp  
qed  

lemma closed-empty [continuous-intros, intro, simp]: closed {}  
unfolding closed-def by simp  

lemma closed-Un [continuous-intros, intro]: closed S \( \Rightarrow \) closed T \( \Rightarrow \) closed (S \( \cup \) T)  
unfolding closed-def by auto  

lemma closed-UNIV [continuous-intros, intro, simp]: closed UNIV  
unfolding closed-def by simp  

lemma closed-Int [continuous-intros, intro]: closed S \( \Rightarrow \) closed T \( \Rightarrow \) closed (S \( \cap \) T)  
unfolding closed-def by auto  

lemma closed-INT [continuous-intros, intro]: \( \forall x \in A. \) closed (B x) \( \Rightarrow \) closed (\( \bigcap \) x \( \in \) A. B x)  
unfolding closed-def by auto  

lemma closed-Inter [continuous-intros, intro]: \( \forall S \in K. \) closed S \( \Rightarrow \) closed (\( \bigcap \) K)  
unfolding closed-def aminus-Inf by auto  

lemma closed-Union [continuous-intros, intro]: finite S \( \Rightarrow \) \( \forall T \in S. \) closed T \( \Rightarrow \) closed (\( \bigcup \) S)  
by (induct set: finite) auto  

lemma closed-UN [continuous-intros, intro]: finite A \( \Rightarrow \) \( \forall x \in A. \) closed (B x) \( \Rightarrow \) closed (\( \bigcup \) x \( \in \) A. B x)  
using closed-Union [of B ' A] by simp  

lemma open-closed: open S \( \Longleftrightarrow \) closed (\( \neg \) S)  
unfolding closed-def by simp  

lemma closed-open: closed S \( \Longleftrightarrow \) open (\( \neg \) S)  
unfolding closed-def by simp  

lemma open-Diff [continuous-intros, intro]: open S \( \Rightarrow \) closed T \( \Rightarrow \) open (S \( \setminus \) T)  
unfolding closed-open Diff-eq by (rule open-Int)


THEORY “Topological-Spaces”
lemma closed-Diff \text{[continuous-intros, intro]}: \text{closed } S \implies \text{open } T \implies \text{closed } (S - T)
unfolding \text{open-closed Diff-eq by (rule closed-Int)}

lemma open-Compl \text{[continuous-intros, intro]}: \text{closed } S \implies \text{open } (\neg S)
unfolding \text{closed-open}.

lemma closed-Compl \text{[continuous-intros, intra]}: \text{open } S \implies \text{closed } (\neg S)
unfolding \text{open-closed}.

lemma open-Collect-neg: \text{closed } \{ x. P x \} \implies \text{open } \{ x. \neg P x \}
unfolding \text{Collect-neg-eq by (rule closed-Compl)}

lemma open-Collect-conj: \text{assumes open } \{ x. P x \} \text{ open } \{ x. Q x \} \text{ shows open } \{ x. P x \land Q x \}
using \text{open-Int[OF assms] by (simp add: Int-def)}

lemma open-Collect-disj: \text{assumes open } \{ x. P x \} \text{ open } \{ x. Q x \} \text{ shows open } \{ x. P x \lor Q x \}
using \text{open-Un[OF assms] by (simp add: Un-def)}

lemma open-Collect-ex: \text{\{\land i. open \{ x. P i x \}\} \implies open \{ x. \exists i. P i x \}}
using \text{open-UN[of UNIV \lambda i. \{ x. P i x \}] unfolding Collect-ex-eq by simp}

lemma open-Collect-imp: \text{closed } \{ x. P x \} \implies \text{open } \{ x. Q x \} \implies \text{open } \{ x. P x \rightarrow Q x \}
unfolding \text{imp-conv-disj by (intro open-Collect-disj open-Collect-neg)}

lemma open-Collect-const: \text{open } \{ x. P \}
by (cases P) auto

lemma closed-Collect-neg: \text{open } \{ x. P x \} \implies \text{closed } \{ x. \neg P x \}
unfolding \text{Collect-neg-eq by (rule closed-Compl)}

lemma closed-Collect-conj: \text{assumes closed } \{ x. P x \} \text{ closed } \{ x. Q x \} \text{ shows closed } \{ x. P x \land Q x \}
using \text{closed-Int[OF assms] by (simp add: Int-def)}

lemma closed-Collect-disj: \text{assumes closed } \{ x. P x \} \text{ closed } \{ x. Q x \} \text{ shows closed } \{ x. P x \lor Q x \}
using \text{closed-Un[OF assms] by (simp add: Un-def)}

lemma closed-Collect-all: \text{\{\land i. closed \{ x. P i x \}\} \rightarrow closed \{ \forall i. P i x \}}
using \text{closed-INT[of UNIV \lambda i. \{ x. P i x \}] unfolding Collect-all-eq by simp}

lemma closed-Collect-imp: \text{open } \{ x. P x \} \implies \text{closed } \{ x. Q x \} \implies \text{closed } \{ x. P x \rightarrow Q x \}
unfolding \text{imp-conv-disj by (intro closed-Collect-disj closed-Collect-neg)}
lemma closed-Collect-const: closed \{x. P\}
  by (cases P) auto

end

98.2 Hausdorff and other separation properties

class t0-space = topological-space +
  assumes t0-space: x \neq y \implies \exists U. open U \land \neg (x \in U \iff y \in U)

class t1-space = topological-space +
  assumes t1-space: x \neq y \implies \exists U. open U \land x \in U \land y \notin U

instance t1-space \subseteq t0-space
proof qed (fast dest: t1-space)

lemma separation-t1:
  fixes x y :: 'a::t1-space
  shows x \neq y \iff (\exists U. open U \land x \in U \land y \notin U)
  using t1-space[of x y] by blast

lemma closed-singleton:
  fixes a :: 'a::t1-space
  shows closed \{a\}
  proof
    let ?T = \bigcup \{S. open S \land a \notin S\}
    have open ?T by (simp add: open-Union)
    also have ?T = \{a\}
      by (simp add: set-eq-iff separation-t1, auto)
    finally show closed ?T unfolding closed-def .
  qed

lemma closed-insert [continuous-intros, simp]:
  fixes a :: 'a::t1-space
  assumes closed S
  shows closed (insert a S)
  proof
    from closed-singleton assms
    have closed (\{a\} \cup S) by (rule closed-Un)
    thus closed (insert a S) by simp
  qed

lemma finite-imp-closed:
  fixes S :: 'a::t1-space set
  shows finite S \implies closed S
  by (induct set: finite, simp-all)

T2 spaces are also known as Hausdorff spaces.

class t2-space = topological-space +
  assumes hausdorff: x \neq y \implies \exists U V. open U \land open V \land x \in U \land y \in V \land
\[ U \cap V = \{\} \]

**instance** \( t_2\)-space \( \subseteq t_1\)-space

**proof** *qed* (\( \text{fast dest: hausdorff} \))

**lemma** separation-t2:

**fixes** \( x \ y :: 'a::t2\)-space

**shows** \( x \neq y \longleftrightarrow (\exists U \ V. \ \text{open} \ U \land \text{open} \ V \land x \in U \land y \in V \land U \cap V = \{\}) \)

**using** \( \text{hausdorff[of x y]} \) \( \text{by blast} \)

**lemma** separation-t0:

**fixes** \( x \ y :: 'a::t0\)-space

**shows** \( x \neq y \longleftrightarrow (\exists U. \ \text{open} \ U \land \sim (x \in U \longleftrightarrow y \in U)) \)

**using** \( \text{t0-space[of x y]} \) \( \text{by blast} \)

A perfect space is a topological space with no isolated points.

**class** perfect-space = topological-space +

**assumes** not-open-singleton: \( \neg \text{open} \{x\} \)

### 98.3 Generators for topologies

**inductive** generate-topology for \( S \) where

- **UNIV**: \( \text{generate-topology} S \ \text{UNIV} \)
- **Int**: \( \text{generate-topology} S \ a \ \Rightarrow \ \text{generate-topology} S \ b \ \Rightarrow \ \text{generate-topology} S \ (a \cap b) \)
- **UN**: \( (\forall k. \ k \in K \ \Rightarrow \ \text{generate-topology} S \ k) \ \Rightarrow \ \text{generate-topology} S \ (\bigcup K) \)
- **Basis**: \( s \in S \ \Rightarrow \ \text{generate-topology} S \ s \)

**hide-fact** (\( \text{open} \)) \( \text{UNIV Int UN Basis} \)

**lemma** generate-topology-Union:

\( (\forall k. \ k \in I \ \Rightarrow \ \text{generate-topology} S \ (K k)) \ \Rightarrow \ \text{generate-topology} S \ (\bigcup k \in I. \ K k) \)

**using** \( \text{generate-topology.UN[of K ' I]} \) \( \text{by auto} \)

**lemma** topological-space-generate-topology:

**class** topological-space (\( \text{generate-topology} S \))

**by** default (\( \text{auto intro: generate-topology.intros} \))

### 98.4 Order topologies

**class** order-topology = order + open +

**assumes** open-generated-order: \( \text{open} = \text{generate-topology} \ (\lambda a. \ \{..< a\}) \cup \text{range} \ (\lambda a. \ \{a <.\}) \)

**begin**

**subclass** topological-space

**unfolding** open-generated-order

**by** (\( \text{rule topological-space-generate-topology} \))
lemma open-greaterThan [continuous-intros, simp]: open \{ a <.. \}
  unfolding open-generated-order by (auto intro: generate-topology.Basis)

lemma open-lessThan [continuous-intros, simp]: open \{ ..< a \}
  unfolding open-generated-order by (auto intro: generate-topology.Basis)

lemma open-greaterThanLessThan [continuous-intros, simp]: open \{ a <..< b \}
  unfolding greaterThanLessThan-eq by (simp add: open-Int)

end

class linorder-topology = linorder + order-topology

lemma closed-atMost [continuous-intros, simp]: closed \{ .. a ::.. \}
  by (simp add: closed-open)

lemma closed-atLeast [continuous-intros, simp]: closed \{ a ::.. .. \}
  by (simp add: closed-open)

lemma closed-atLeastAtMost [continuous-intros, simp]: closed \{ a ::.. b \}
  proof –
    have \{ a .. b \} = \{ a .. \} ∩ \{ .. b \}
    by auto
    then show ?thesis
    by (simp add: closed-Int)
  qed

lemma (in linorder) less-separate:
  assumes x < y
  shows ∃ a b. x ∈ \{ ..< a \} ∧ y ∈ \{ b <.. \} ∧ \{ ..< a \} ∩ \{ b <.. \} = \{
proof (cases \b x < z \wedge z < y
  case True
  then obtain z where x < z ∧ z < y ..
  then have x ∈ \{ ..< z \} ∧ y ∈ \{ z <.. \} ∧ \{ z <.. \} ∩ \{ ..< z \} = \{
  by auto
  then show ?thesis by blast
next
  case False
  with \b x < y \ have x ∈ \{ ..< y \} ∧ y ∈ \{ x <.. \} ∧ \{ x <.. \} ∩ \{ ..< y \} = \{
  by auto
  then show ?thesis by blast
qed

instance linorder-topology ⊆ t2-space
proof
  fix x y :: 'a
  from less-separate[of x y] less-separate[of y x]
show \( x \neq y \implies \exists U \ V. \ open U \land open V \land x \in U \land y \in V \land U \cap V = \{\} \)

by (elim \( \text{neqE} \)) (metis open-lessThan open-greaterThan Int-commute)+

qed

lemma (in linorder-topology) open-right:
assumes open \( S \ x \in S \) and \( \text{gt-ex: } x < y \) shows \( \exists b > x. \ \{x .. < b\} \subseteq S \)

using assms unfolding open-generated-order

proof induction
  case (Int \( A \ B \))
  then obtain \( a \ b \) where \( a > x \ \{x .. < a\} \subseteq A \ b > x \ \{x .. < b\} \subseteq B \) by auto
  then show \( ?\text{case} \) by (auto intro: exI ![of - min a b])
next
  case (Basis \( S \))
  then show \( ?\text{case} \) by (fastforce intro: exI ![of - y] gt-ex)

qed blast+

lemma (in linorder-topology) open-left:
assumes open \( S \ x \in S \) and \( \text{lt-ex: } y < x \) shows \( \exists b < x. \ \{b <.. x\} \subseteq S \)

using assms unfolding open-generated-order

proof induction
  case (Int \( A \ B \))
  then obtain \( a \ b \) where \( a < x \ \{a .. < x\} \subseteq A \ b < x \ \{b <.. x\} \subseteq B \) by auto
  then show \( ?\text{case} \) by (auto intro: exI ![of - max a b])
next
  case (Basis \( S \))
  then show \( ?\text{case} \) by (fastforce intro: exI ![of - y] lt-ex)

qed blast+

98.5 Filters

This definition also allows non-proper filters.

locale is-filter =
  fixes \( F :: (\ 'a \Rightarrow \text{bool}) \Rightarrow \text{bool} \)
  assumes True: \( F \ (\lambda x. \ \text{True}) \)
  assumes conj: \( F \ (\lambda x. \ P \ x) \Longrightarrow F \ (\lambda x. \ Q \ x) \Longrightarrow F \ (\lambda x. \ P \ x \land Q \ x) \)
  assumes mono: \( \forall x. \ P x \Longrightarrow Q x \Longrightarrow F \ (\lambda x. \ P x) \Longrightarrow F \ (\lambda x. \ Q x) \)

typedef 'a filter = \{F :: (\ 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}. \ is-filter F\}

proof
  show \( (\lambda x. \ \text{True}) \in \ ?\text{filter} \) by (auto intro: is-filter.intro)

qed

lemma is-filter-Rep-filter: is-filter (Rep-filter \( F \))
  using Rep-filter ![of \( F \)] by simp

lemma Abs-filter-inverse':
assumes is-filter \( F \) shows \( \text{Rep-filter} \ (\text{Abs-filter} \ F) = F \)
  using assms by (simp add: Abs-filter-inverse)
98.5.1 Eventually

**definition** eventually :: ('a ⇒ bool) ⇒ 'a filter ⇒ bool

**where** eventually P F ←→ Rep-filter F P

**lemma** eventually-Abs-filter:
  assumes is-filter F
  shows eventually P (Abs-filter F) = F P

**unfolding** eventually-def using assms by (simp add: Abs-filter-inverse)

**lemma** filter-eq-iff:
  shows F = F' ←→ (∀P. eventually P F = eventually P F')

**unfolding** Rep-filter-inject [symmetric] fun-eq-iff eventually-def ..

**lemma** eventually-True [simp]: eventually (λx. True) F

**unfolding** eventually-def
by (rule is-filter.True [OF is-filter-Rep-filter])

**lemma** always-eventually: ∀x. P x ⇒ eventually P F

**proof** –
  assume ∀x. P x hence P = (λx. True) by (simp add: ext)
  thus eventually P F by simp

**qed**

**lemma** eventually-mono:
  (∀x. P x −→ Q x) ⇒ eventually P F −→ eventually Q F

**unfolding** eventually-def
by (rule is-filter.mono [OF is-filter-Rep-filter])

**lemma** eventually-conj:
  assumes P: eventually (λx. P x) F
  assumes Q: eventually (λx. Q x) F
  shows eventually (λx. P x ∧ Q x) F

**using** assms unfolding eventually-def
by (rule is-filter.conj [OF is-filter-Rep-filter])

**lemma** eventually-Ball-finite:
  assumes finite A and ∀y∈A. eventually (λx. P x y) net
  shows eventually (λx. ∀y∈A. P x y) net

**using** assms by (induct set: finite, simp, simp add: eventually-conj)

**lemma** eventually-all-finite:
  fixes P :: 'a ⇒ 'b::{finite ⇒ bool
  assumes ∃y. eventually (λx. P x y) net
  shows eventually (λx. ∀y∈A. P x y) net

**using** eventually-Ball-finite [of UNIV P] assms by simp

**lemma** eventually-mp:
  assumes eventually (λx. P x −→ Q x) F
  assumes eventually (λx. P x) F
  shows eventually (λx. Q x) F
proof (rule eventually-mono)
  show ∀ x. (P x −→ Q x) ∧ P x −→ Q x by simp
  using assms by (rule eventually-conj)
qed

lemma eventually-rev-mp:
  assumes eventually (λx. P x) F
  assumes eventually (λx. P x −→ Q x) F
  shows eventually (λx. Q x) F
  using assms(2) assms(1) by (rule eventually-mp)

lemma eventually-conj-iff:
  eventually (λx. P x ∧ Q x) F ←→ eventually P F ∧ eventually Q F
  by (auto intro: eventually-conj elim: eventually-rev-mp)

lemma eventually-elim1:
  assumes eventually (λi. P i) F
  assumes P i =⇒ Q i
  shows eventually (λi. Q i) F
  using assms by (auto elim!: eventually-rev-mp)

lemma eventually-elim2:
  assumes eventually (λi. P i) F
  assumes eventually (λi. Q i) F
  assumes P i =⇒ Q i =⇒ R i
  shows eventually (λi. R i) F
  using assms by (auto elim!: eventually-rev-mp)

lemma not-eventually-impl: eventually P F =⇒ ¬ eventually Q F =⇒ ¬ eventually (λx. P x −→ Q x) F
  by (auto intro: eventually-mp)

lemma not-eventuallyD: ¬ eventually P F =⇒ ∃ x. ¬ P x
  by (metis always-eventually)

lemma eventually-subst:
  assumes eventually (λn. P n = Q n) F
  shows eventually P F = eventually Q F (is ?L = ?R)
proof –
  from assms have eventually (λx. P x −→ Q x) F
    and eventually (λx. Q x −→ P x) F
    by (auto elim: eventually-elim1)
  then show ?thesis by (auto elim: eventually-elim2)
qed

ML ⟨⟨
  fun eventually-elim-tac ctxt thms = SUBGOAL-CASES (fn (_, _, st) =>
    let

```
THEORY "Topological-Spaces"

val thy = Proof-Context.theory-of ctxt
val mp-thms = thms RL [@{thm eventually-rev-mp}]
val raw-elim-thm = (@{thm allI} RS @{thm always-eventually})
  |> fold (fn thm1 => fn thm2 => thm2 RS thm1) mp-thms
  |> fold (fn - => fn thm => @{thm impI} RS thm) thms
val cases-prop = prop-of (raw-elim-thm RS st)
val cases = (Rule-Cases.make-common (thy, cases-prop) [((elim, []), [])])
in
  CASES cases (rtac raw-elim-thm 1)
end

method-setup eventually-elim = ⟨⟨
  Scan.succeed (fn ctxt => METHOD-CASES (eventually-elim-tac ctxt))⟩⟩

} elimination of eventually quantifiers

98.5.2 Finer-than relation

$F \leq F'$ means that filter $F$ is finer than filter $F'$.

instantiation filter :: (type) complete-lattice
begin

definition le-filter-def:
$F \leq F' \iff (\forall P. \text{eventually } P F' \rightarrow \text{eventually } P F)$

definition
$(F :: \text{a filter}) < F' \iff F \leq F' \land \neg F' \leq F$

definition
$\text{top} = \text{Abs-filter} (\lambda P. \forall x. P x)$

definition
$\text{bot} = \text{Abs-filter} (\lambda P. \text{True})$

definition
$\text{sup } F F' = \text{Abs-filter} (\lambda P. \text{eventually } P F \land \text{eventually } P F')$

definition
$\text{inf } F F' = \text{Abs-filter}$
$(\lambda P. \exists Q R. \text{eventually } Q F \land \text{eventually } R F' \land (\forall x. Q x \land R x \rightarrow P x))$

definition
$\text{Sup } S = \text{Abs-filter} (\lambda P. \forall F \in S. \text{eventually } P F)$

definition
$\text{Inf } S = \text{Sup } \{ F :: \text{a filter}. \forall F' \in S. F \leq F' \}$

lemma eventually-top [simp]: eventually $P \text{top} \iff (\forall x. P x)$
unfolding  top-filter-def
by (rule eventually-Abs-filter, rule is-filter.intro, auto)

lemma eventually-bot [simp]: eventually P bot
unfolding  bot-filter-def
by (subst eventually-Abs-filter, rule is-filter.intro, auto)

lemma eventually-sup:
eventually P (sup F F')  \iff  eventually P F  \land  eventually P F'
unfolding  sup-filter-def
by (rule eventually-Abs-filter, rule is-filter.intro)
(auto elim!: eventually-rev-mp)

lemma eventually-inf:
eventually P (inf F F')  \iff
(\exists Q R. \ eventually Q F  \land  \ eventually R F'  \land  (\forall x. Q x \land R x  \rightarrow  P x))
unfolding  inf-filter-def
apply (rule eventually-Abs-filter, rule is-filter.intro)
apply (fast intro: eventually-True)
apply clarify
apply (intro exI conjI)
apply (erule (1) eventually-conj)
apply (erule (1) eventually-conj)
apply simp
apply auto
done

lemma eventually-Sup:
eventually P (Sup S)  \iff
(\forall F \in S. \ eventually P F)
unfolding  Sup-filter-def
apply (rule eventually-Abs-filter, rule is-filter.intro)
apply (auto intro: eventually-conj elim!: eventually-rev-mp)
done

instance proof
fix F F' F'' : 'a filter and S : 'a filter set
{ show F < F'  \iff  F \leq F' \land  \neg F' \leq F
  by (rule less-filter-def) }
{ show F \leq F'
  unfolding le-filter-def by simp }
{ assume F' \leq F'' and F' \leq F thus F = F''
  unfolding le-filter-def by simp }
{ assume F \leq F' and F \leq F'' thus F = F'
  unfolding le-filter-def filter-eq-iff by fast }
{ show inf F F' \leq F and inf F F' \leq F'
  unfolding le-filter-def eventually-inf by (auto intro: eventually-True) }
{ assume F \leq F' and F \leq F'' thus F \leq inf F' F''
  unfolding le-filter-def eventually-inf
  by (auto elim!: eventually-mono intro: eventually-conj) }


{ show $F \leq \sup F F'$ and $F' \leq \sup F F'$
  unfolding le-filter-def eventually-sup by simp-all }
{ assume $F \leq F''$ and $F' \leq F''$ thus $\sup F F' \leq F''$
  unfolding le-filter-def eventually-sup by simp }
{ assume $F'' \in S$ thus $\inf S \leq F''$
  unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
{ assume $\forall F. F \in S$ thus $\inf S \leq F$
  unfolding le-filter-def eventually-Sup Ball-def by simp }
{ show $\inf \{\} = (\top ::'a filter)$
  by (auto simp: top-filter-def Inf-filter-def Sup-filter-def)
  (metis (full-types) top-filter-def always-eventually eventually-top) }
{ show $\sup \{\} = (\bot ::'a filter)$
  by (auto simp: bot-filter-def Sup-filter-def)
qed }

end

lemma filter-leD:
  $F \leq F' \Longrightarrow \text{eventually } P F' \Longrightarrow \text{eventually } P F$
  unfolding le-filter-def by simp

lemma filter-leI:
  $(\forall P. \text{eventually } P F' \Longrightarrow \text{eventually } P F) \Longrightarrow F \leq F'$
  unfolding le-filter-def eventually-Sup Ball-def by simp

lemma eventually-False:
  $\text{eventually } (\lambda x. \text{False}) F \leftrightarrow F = \bot$
  unfolding filter-eq-iff by (auto elim: eventually-rev-mp)

abbreviation (input) trivial-limit :: 'a filter ⇒ bool
  where trivial-limit $F \equiv F = \bot$

lemma trivial-limit-def: trivial-limit $F \leftrightarrow \text{eventually } (\lambda x. \text{False}) F$
  by (rule eventually-False [symmetric])

lemma eventually-const: $\neg$ trivial-limit net $\Longrightarrow \text{eventually } (\lambda x. P) \text{ net} \leftrightarrow P$
  by (cases P) (simp-all add: eventually-False)

lemma eventually-Inf: $\text{eventually } P \ (\text{Inf } B) \leftrightarrow (\exists X \subseteq B. \text{finite } X \land \text{eventually } P \ (\text{Inf } X))$
proof
  let $?F = \lambda P. \exists X \subseteq B. \text{finite } X \land \text{eventually } P \ (\text{Inf } X)$

  { fix P have $\text{eventually } P \ (\text{Abs-filter } ?F) \leftrightarrow ?F P$
    proof (rule eventually-Abs-filter is-filter.intro)+
  }
show ?F (\lambda x. True)
  by (rule exI[of - {}]) (simp add: le-fun-def)

next
  fix P Q
  assume ?F P then guess X ..
  moreover assume ?F Q then guess Y ..
  ultimately show ?F (\lambda x. P x & Q x)
    by (intro exI[of - X \cup Y])
      (auto simp: Inf-union-distrib eventually-inf)

next
  fix P Q
  assume ?F P then guess X ..
  moreover assume \forall x. P x \longrightarrow Q x
  ultimately show ?F Q
    by (intro exI[of - X]) (auto elim: eventually-elim1)

qed

note eventually-F = this

have Inf B = Abs-filter ?F
proof (intro antisym Inf-greatest)
  show Inf B \leq Abs-filter ?F
    by (auto simp: le-filter-def eventually-F dest: Inf-superset-mono)

next
  fix F assume F \in B then show Abs-filter ?F \leq F
    by (auto simp add: le-filter-def eventually-F intro: exI[of - {F}])

then show \?thesis
  by (simp add: eventually-F)

qed

lemma eventually-INF: eventually P (\INF b:B. F b) \iff \exists X \subseteq B. finite X \land

  eventually P (\INF b:X. F b))
unfolding INDEF[of B] \qquad eventually-INF[of P F'B]
by (metis Inf-image-eq finite-imageI image-mono finite-subset-image)

lemma Inf-filter-not-bot:
fixes B :: 'a filter set
shows (\bigwedge X. X \subseteq B \then finite X \then Inf X \neq bot) \then Inf B \neq bot
unfolding trivial-limit-def eventually-Inf[of - B]
  bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma INF-filter-not-bot:
fixes F :: 'i \Rightarrow 'a filter
shows (\bigwedge X. X \subseteq B \then finite X \then (INF b:X. F b) \neq bot) \then (INF b:B. F)
  \neq bot
unfolding trivial-limit-def eventually-INF[of - B]
  bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp
lemma \textit{eventually-Inf-base}: \begin{align*}
\text{assumes } & B \neq \{\} \text{ and } \text{base}: \forall F, G, F \in B \implies G \in B \implies \exists x \in B. \ x \leq \inf F \ G \\
\text{shows } & \text{eventually } P \ (\text{Inf } B) \iff (\exists b \in B. \ \text{eventually } P \ b) \\
\text{proof } & (\text{subst eventually-Inf, safe}) \\
\text{fix } & X \text{ assume } \text{finite } X \ X \subseteq B \\
\text{then have } & \exists b \in B. \ \forall x \in X. \ b \leq x \\
\text{proof } & \text{induct} \\
\text{case empty then show } & \text{?case} \\
\text{using } & (\text{?case}) \\
\text{next} \\
\text{case } & \text{insert } x \ X \\
\text{then obtain } & b \text{ where } b \in B \ \land x \in X \implies b \leq x \ \text{by auto} \\
\text{with } & (\text{?case}) \text{ show } ?\text{case} \\
\text{by } & (\text{auto intro: order-trans}) \\
\text{qed} \\
\text{then obtain } & b \text{ where } b \in B \ b \leq \text{Inf } X \ \text{by (auto simp: le-Inf-iff)} \\
\text{then show } & \text{eventually } P \ (\text{Inf } X) \implies \text{Bex } B \ (\text{eventually } P) \\
\text{by } & (\text{intro bexI [of - b]) (auto simp: le-filter-def}) \\
\text{qed} & (\text{auto intro!: exI [of - \{x\} for x]}) \\
\end{align*}

lemma \textit{eventually-INF-base}: \begin{align*}
B \neq \{\} \implies \left( \forall a, b \in B \implies \exists x \in B. \ F x \leq \inf (F \ a) \ (F \ b) \implies \text{eventually } P \ (\text{Inf } b:B. \ F \ b) \iff (\exists b \in B. \ \text{eventually } P \ (F \ b)) \right)
\text{unfolding INF-def by (subst eventually-INF-base) auto}
\end{align*}

98.5.3 Map function for filters

\begin{itemize}
\item definition \textit{filtermap} :: \('a \Rightarrow \text{filter} \Rightarrow 'b \text{ filter} \\
\item where \textit{filtermap} f F \ = \ \text{Abs-filter} \ (\lambda P. \ \text{eventually} \ (\lambda x. \ P \ (f \ x)) \ F) \\
\item lemma \textit{eventually-filtermap}: \begin{align*}
& \text{eventually } P \ (\text{filtermap } f \ F) \ = \ \text{eventually} \ (\lambda x. \ P \ (f \ x)) \ F \\
& \text{unfolding filtermap-def} \\
& \text{apply (rule eventually-Abs-filter)} \\
& \text{apply (rule is-filter.intro)} \\
& \text{apply (auto elim!: eventually-rev-mp)} \\
& \text{done} \\
\item lemma \textit{filtermap-ident}: \text{filtermap} \ (\lambda x. \ x) \ F \ = \ F \\
& \text{by (simp add: filter-eq-iff eventually-filtermap)} \\
\item lemma \textit{filtermap-filtermap}: \begin{align*}
& \text{filtermap } f \ (\text{filtermap } g \ F) \ = \ \text{filtermap} \ (\lambda x. \ f \ (g \ x)) \ F \\
& \text{by (simp add: filter-eq-iff eventually-filtermap)} \\
\item lemma \textit{filtermap-mono}: F \leq F' \implies \text{filtermap} f F \leq \text{filtermap} f F' \\
& \text{unfolding le-filter-def eventually-filtermap by simp}
\end{align*}
\end{itemize}
lemma filtermap-bot [simp]: filtermap f bot = bot
  by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-sup: filtermap f (sup F1 F2) = sup (filtermap f F1) (filtermap f F2)
  by (auto simp: filter-eq-iff eventually-filtermap eventually-sup)

lemma filtermap-inf: filtermap f (inf F1 F2) ≤ inf (filtermap f F1) (filtermap f F2)
  by (auto simp: le-filter-def eventually-filtermap eventually-inf)

lemma filtermap-INF: filtermap f (INF b: B. F b) ≤ (INF b: B. filtermap f (F b))
proof –
  { fix X :: 'c set assume finite X
    then have filtermap f (INFIMUM X F) ≤ (INF b: X. filtermap f (F b))
      proof (induct)
        case (insert x X)
        have filtermap f (INF a: insert x X. F a) ≤ inf (filtermap f (F x)) (filtermap f (INF a: X. F a))
          by (rule order-trans[OF - filtermap-inf]) simp
        also have ... ≤ inf (filtermap f (F x)) (INF a: X. filtermap f (F a))
          by (intro inf-mono insert order-refl)
        finally show ?case
          by simp
      qed simp }
  then show ?thesis
    unfolding le-filter-def eventually-filtermap
    by (subst (1 2) eventually-INF) auto
qed

98.5.4 Standard filters

definition principal :: 'a set ⇒ 'a filter where
  principal S = Abs-filter (λP. ∀x∈S. P x)

lemma eventually-principal: eventually P (principal S) iff (∀x∈S. P x)
  unfolding principal-def
  by (rule eventually-Abs-filter, rule is-filter.intro) auto

lemma eventually-inf-principal: eventually P (inf F (principal s)) iff eventually (∀x. x ∈ s → P x) F
  unfolding eventually-inf eventually-principal
  by (auto elim: eventually-elim1)

lemma principal-UNIV[simp]: principal UNIV = top
  by (auto simp: filter-eq-iff eventually-principal)

lemma principal-empty[simp]: principal {} = bot
  by (auto simp: filter-eq-iff eventually-principal)
lemma principal-eq-bot-iff: principal X = bot ←→ X = {}
  by (auto simp add: filter-eq-iff eventually-principal)

lemma principal-le-iff: principal A ≤ principal B ←→ A ⊆ B
  by (auto simp: le-filter-def eventually-principal)

lemma principal-le-iff: principal A ≤ principal B ←→ A ⊆ B
  by (auto simp: le-filter-def eventually-principal)

lemma le-principal: F ≤ principal A ←→ eventually (λx. x ∈ A) F
  unfolding le-filter-def eventually-principal
  apply safe
  apply (erule-tac x=x in allE)
  apply (auto elim: eventually-elim1)
  done

lemma principal-inject: principal A = principal B ←→ A = B
  unfolding eq-iff by simp

lemma sup-principal[simp]: sup (principal A) (principal B) = principal (A ∪ B)
  unfolding filter-eq-iff eventually-sup eventually-principal by auto

lemma inf-principal[simp]: inf (principal A) (principal B) = principal (A ∩ B)
  unfolding filter-eq-iff eventually-inf eventually-principal
  by (auto intro: exI [of - λx. x ∈ A]
                   exI [of - λx. x ∈ B])

lemma SUP-principal[simp]: (SUP i : I. principal (A i)) = principal (∪ i∈I. A i)
  unfolding filter-eq-iff eventually-Sup SUP-def by (auto simp: eventually-principal)

lemma INF-principal-finite: finite X =⇒ (INF x:X. principal (f x)) = principal (∩ x∈X. f x)
  by (induct X rule: finite-induct)

lemma filtermap-principal[simp]: filtermap f (principal A) = principal (λx. f x)
  unfolding filter-eq-iff eventually-filtermap eventually-principal by simp

98.5.5 Order filters

definition at-top : ('a::order) filter
  where at-top = (INF k. principal {k ..})

lemma at-top-sub: at-top = (INF k:{c::'a::linorder..}. principal {k ..})
  by (auto intro!: INF-eq max.cobounded1 max.cobounded2 simp: at-top-def)

lemma eventually-at-top-linorder: eventually P at-top =⇒ (∃ N::'a::linorder. ∀ n≥N. P n)
  unfolding at-top-def
  by (subst eventually-INF-base) (auto simp: eventually-principal intro: max.cobounded1 max.cobounded2)
lemma eventually-ge-at-top:
  eventually (λx. (c::linorder) ≤ x at-top
  unfolding eventually-at-top-linorder by auto

lemma eventually-at-top-dense: eventually P at-top ↔ (∃ N::'a::{no-top, linorder}. ∀ n>N. P n)
  proof –
   have eventually P (INF k. principal {k <..}) ↔ (∃ N::'a. ∀ n>N. P n)
     by (subst eventually-INF-base) (auto simp: eventually-principal intro: max.cobounded1 max.cobounded2)
   also have (INF k. principal {k::'a <..}) = at-top
     unfolding at-top-def
     by (intro INF-eq) (auto intro: less-imp-le simp: Ici-subset-Ioi-iff lt-ex)
   finally show ?thesis .
  qed

lemma eventually-gt-at-top:
  eventually (λx. (c::unbounded-dense-linorder) < x) at-top
  unfolding eventually-at-top-dense by auto

definition at-bot :: ('a::order) filter
  where at-bot = (INF k. principal {.. k})

lemma at-bot-sub: at-bot = (INF k:{.. c::'a::linorder}. principal {.. k})
  by (auto intro!: INF-eq min.cobounded1 min.cobounded2 simp: at-bot-def)

lemma eventually-at-bot-linorder:
  fixes P :: 'a::linorder ⇒ bool shows eventually P at-bot ↔ (∃ N. ∀ n≤N. P n)
  unfolding at-bot-def
  by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)

lemma eventually-le-at-bot:
  eventually (λx. x ≤ (c::linorder)) at-bot
  unfolding eventually-at-bot-linorder by auto

lemma eventually-at-bot-dense: eventually P at-bot ↔ (∃ N::'a::{no-bot, linorder}. ∀ n<N. P n)
  proof –
   have eventually P (INF k. principal {..< k}) ↔ (∃ N::'a. ∀ n<N. P n)
     by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)
   also have (INF k. principal {..< k::'a}) = at-bot
     unfolding at-bot-def
     by (intro INF-eq) (auto intro: less-imp-le simp: Iic-subset-Ioi-iff lt-ex)
   finally show ?thesis .
  qed

lemma eventually-gt-at-bot:
eventually ($\lambda x. \ x < (c::unbounded-dense-linorder)$) at-bot
unfolding eventually-at-bot-dense by auto

lemma trivial-limit-at-bot-linorder: ¬ trivial-limit (at-bot ::('a::linorder) filter)
unfolding trivial-limit-def
by (metis eventually-at-bot-linorder order-refl)

lemma trivial-limit-at-top-linorder: ¬ trivial-limit (at-top ::('a::linorder) filter)
unfolding trivial-limit-def
by (metis eventually-at-top-linorder order-refl)

98.6 Sequentially

abbreviation sequentially :: nat filter
where sequentially ≡ at-top

lemma eventually-sequentially:
  eventually P sequentially ←→ ($\exists N. \ \forall n\geq N. \ P n$)
by (rule eventually-at-top-linorder)

lemma sequentially-bot [simp, intro]: sequentially ≠ bot
unfolding filter-eq-iff eventually-sequentially by auto

lemmas trivial-limit-sequentially = sequentially-bot

lemma eventually-False-sequentially [simp]:
  ¬ eventually ($\lambda n. \ False$) sequentially
by (simp add: eventually-False)

lemma le-sequentially:
  $F \leq$ sequentially ←→ ($\forall N. \ eventually (\lambda n. \ N \leq n) \ F$)
by (simp add: at-top-def le-INF-iff le-principal)

lemma eventually-sequentiallyI:
  assumes $\forall x. \ c \leq x \Longrightarrow P x$
  shows eventually P sequentially
using assms by (auto simp: eventually-sequentially)

lemma eventually-sequentially-seq:
  eventually ($\lambda n. \ P (n + k)$) sequentially ←→ eventually P sequentially
unfolding eventually-sequentially
apply safe
  apply (rule-tac $x=N + k$ in exI)
  apply rule
  apply (erule-tac $x=n - k$ in allE)
  apply auto []
apply (rule-tac $x=N$ in exI)
apply auto []
done
98.6.1 Topological filters

**Definition (in topological-space)**

\( \text{nhds} :: 'a \Rightarrow 'a \text{ filter} \)

where \( \text{nhds} a = (\INF S:\{S. \text{open } S \land a \in S\}. \text{principal } S) \)

**Definition (in topological-space)**

\( \text{at-within} :: 'a \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ filter} \)

where \( \text{at} a \text{ within } s = \text{inf} (\text{nhds } a) (\text{principal } (s - \{a\})) \)

**Abbreviation (in topological-space)**

\( \text{at} :: 'a \Rightarrow 'a \text{ filter} \)

where \( \text{at } x \equiv \text{at } x \text{ within } (\text{CONST } \text{UNIV}) \)

**Abbreviation (in order-topology)**

\( \text{at-right} :: 'a \Rightarrow 'a \text{ filter} \)

where \( \text{at-right } x \equiv \text{at } x \text{ within } \{x <..\} \)

**Abbreviation (in order-topology)**

\( \text{at-left} :: 'a \Rightarrow 'a \text{ filter} \)

where \( \text{at-left } x \equiv \text{at } x \text{ within } \{..< x\} \)

**Lemma (in topological-space)**

\( \text{nhds-generated-topology:} \)

\( \text{open } = \text{generate-topology } T \implies \text{nhds } x = (\INF S:\{S \in T. x \in S\}. \text{principal } S) \)

**Unfolding** \( \text{nhds-def} \)

**Proof (safe intro! antisym INF-greatest)**

fix \( S \)

assume \( \text{generate-topology } T \ S \in S \)

then show \( (\INF S:\{S \in T. x \in S\}. \text{principal } S) \leq \text{principal } S \)

by induction

(auto intro: INF-lower intro: generate-topology.intros)

**Qed (auto intro!: INF-lower intro: generate-topology.intros)**

**Lemma (in topological-space)**

\( \text{eventually-nhds:} \)

\( \text{eventually } P (\text{nhds } a) \iff (\exists S. \text{open } S \land a \in S \land (\forall x \in S. P x)) \)

**Unfolding** \( \text{nhds-def by (subst eventually-INF-base) (auto simp: eventually-principal)} \)

**Lemma** \( \text{nhds-neq-bot [simp]: nhds } a \neq \text{ bot} \)

**Unfolding** \( \text{trivial-limit-def eventually-nhds by simp} \)

**Lemma** \( \text{at-within-eq:} \)

\( \text{at } x \text{ within } s = (\INF S:\{S \in T. x \in S\}. \text{principal } (S \cap s - \{x\})) \)

**Unfolding** \( \text{nhds-def at-within-def by (subst INF-inf-const2[symmetric]) (auto simp add: Diff-Int-distrib)} \)

**Lemma** \( \text{eventually-at-filter:} \)

\( \text{eventually } P (\text{at } a \text{ within } s) \iff (\exists x. x \neq a \rightarrow x \in s \rightarrow P x) \) \( \text{nhds } a \)

**Unfolding** \( \text{at-within-def eventually-inf-principal by (simp add: imp-conjL[symmetric] conj-commute)} \)

**Lemma** \( \text{at-le:} s \subseteq t \implies \text{at } x \text{ within } s \leq \text{at } x \text{ within } t \)

**Unfolding** \( \text{at-within-def by (intro inf-mono) auto} \)
lemma eventually-at-topological:
  eventually P (at a within s) ←→ (∃ S. open S ∧ a ∈ S ∧ (∀ x ∈ S. x ≠ a → x ∈ s → P x))

unfolding eventually-nhds eventually-at-filter by simp

lemma at-within-open: a ∈ S ⇒ open S ⇒ at a within S = at a
unfolding filter-eq-iff eventually-at-topological by (metis open-Int Int-iff UNIV-I)

lemma at-within-empty [simp]: at a within {} = bot
unfolding at-within-def by simp

lemma at-within-union:
at x within (S ∪ T) = sup (at x within S) (at x within T)
unfolding filter-eq-iff eventually-sup eventually-at-filter by (auto elim!: eventually-rev-mp)

lemma at-eq-bot-iff:
at a = bot ←→ open {a}
unfolding trivial-limit-def eventually-at-topological by (safe, case-tac S = {a}, simp, fast, fast)

lemma at-neq-bot [simp]: at (a :: ′a::perfect-space) ≠ bot
by (simp add: at-eq-bot-iff not-open-singleton)

lemma (in order-topology) nhds-order: nhds x = inf (INF a:{..<}. principal {..< a}) (INF a:{..<}. principal {..< a})
proof -
  have 1: {S ∈ range lessThan ∪ range greaterThan. x ∈ S} =
    (λa. {..< a}) ∪ {x <..} ∪ (λa. {..< a}) ∪ {..< x}
    by auto
  show ?thesis
    unfolding nhds-generated-topology[OF open-generated-order] INF-union 1 INF-image comp-def ..
qed

lemma (in linorder-topology) at-within-order: UNIV ≠ {x} ⇒
at x within s = inf (INF a:{..<}. principal {..< a} ∩ s − {x}))
  (INF a:{..< x}. principal {..< x} ∩ s − {x}))
proof (cases {x <..} = {} {..< x} = {} rule: case-split case-split case-split case-split)
  assume UNIV ≠ {x} {x <..} = {} {..< x} = {}
  moreover have UNIV = {..< x} ∪ {x} ∪ {x <..}
    by auto
  ultimately show ?thesis
    by auto
qed (auto simp: at-within-def nhds-order Int-Diff inf-principal[symmetric] INF-inf-const2
  inf-sup-aci where 'a='a filter
  simp del: inf-principal)

lemma (in linorder-topology) at-left-eq:
y < x ⇒ at-left x = (INF a:{..< x}. principal {..< x})
by ( subst at-within-order)
(auto simp: greaterThan-Int-greaterThan greaterThanLessThan-eq[symmetric]
min.absorb2 INF-constant
intro!: INF-lower2 inf-absorb2)

lemma (in linorder-topology) eventually-at-left:
y < x ⇒ eventually P (at-left x) ←→ (∃ b < x. ∀ y > b. y < x → P y)
unfolding at-left-eq by ( subst eventually-INF-base) (auto simp: eventually-principal Ball-def)

lemma (in linorder-topology) at-right-eq:
x < y =⇒ at-right x = (INF a:{x<..}. principal {x<..<a})
by (subst at-within-order)
(auto simp: lessThan-Int-lessThan greaterThanLessThan-eq[symmetric] max.absorb2
INF-constant Int-commute
intro!: INF-lower2 inf-absorb1)

lemma (in linorder-topology) eventually-at-right:
x < y =⇒ eventually P (at-right x) ←→ (∃ b > x. ∀ y > x. y < b → P y)
unfolding at-right-eq by ( subst eventually-INF-base) (auto simp: eventually-principal Ball-def)

lemma trivial-limit-at-right-top: at-right (top::::{order-top, linorder-topology}) = bot
unfolding filter-eq-iff eventually-at-topological by auto

lemma trivial-limit-at-left-bot: at-left (bot::::{order-bot, linorder-topology}) = bot
unfolding filter-eq-iff eventually-at-topological by auto

lemma trivial-limit-at-left-real [simp]:
¬ trivial-limit (at-left (x:::a::{no-bot, dense-order, linorder-topology})))
using lt-ex[of x]
by safe (auto simp add: trivial-limit-def eventually-at-left dest: dense)

lemma trivial-limit-at-right-real [simp]:
¬ trivial-limit (at-right (x:::a::{no-top, dense-order, linorder-topology})))
using gt-ex[of x]
by safe (auto simp add: trivial-limit-def eventually-at-right dest: dense)

lemma at-eq-sup-left-right: at (x:::a::linorder-topology) = sup (at-left x) (at-right x)
by (auto simp: eventually-at-filter filter-eq-iff eventually-sup
elim: eventually-elim2 eventually-elim1)

lemma eventually-at-split:
eventually P (at (x:::a::linorder-topology)) ↔ eventually P (at-left x) ∧ eventually P (at-right x)
by (subst at-eq-sup-left-right) (simp add: eventually-sup)
98.7 Limits

definition filterlim :: (‘a ⇒ ‘b) ⇒ ‘b filter ⇒ ‘a filter ⇒ bool where
  filterlim f F2 F1 ←→ filtermap f F1 ≤ F2

syntax
  -LIM :: pattrns ⇒ ‘a ⇒ ‘b ⇒ ‘a ⇒ bool ((3LIM (-)/ (-)/ (-) ⇒ (-)) [1000, 10, 0, 10] 10)

translations
  LIM x F1. f :- F2 == CONST filterlim (%x. f) F2 F1

lemma filterlim-iff:
  (LIM x F1. f x :- F2) ←→ (∀ P. eventually P F2 → eventually (λx. P (f x)) F1)
  unfolding filterlim-def le-filter-def eventually-filtermap ..

lemma filterlim-compose:
  filterlim g F3 F2 ⇒ filterlim f F2 F1 ⇒ filterlim (λx. g (f x)) F3 F1
  unfolding filterlim-def filtermap-filtermap[symmetric] by (metis filtermap-mono order-trans)

lemma filterlim-mono:
  filterlim f F2 F1 ⇒ F2 ≤ F2′ ⇒ F1′ ≤ F1 ⇒ filterlim f F2′ F1′
  unfolding filterlim-def by (metis filtermap-mono order-trans)

lemma filterlim-ident:
  LIM x F. x :- F
  by (simp add: filterlim-def filtermap-ident)

lemma filterlim-cong:
  F1 = F1′ ⇒ F2 = F2′ ⇒ eventually (λx. f x = g x) F2 ⇒ filterlim f F1 F2 = filterlim g F1′ F2′
  by (auto simp: filterlim-def le-filter-def eventually-filtermap elim: eventually-elim2)

lemma filterlim-mono-eventually:
  assumes filterlim f F G and ord: F ≤ F′ G′ ≤ G
  assumes eq: eventually (λx. f x = f′ x) G′
  shows filterlim f′ F′ G′
  apply (rule filterlim-cong[OF refl refl eq, THEN iffD1])
  apply (rule filterlim-mono[OF - ord])
  apply fact
  done

lemma filtermap-mono-strong: inj f ⇒ filtermap f F ≤ filtermap f G ←→ F ≤ G
  apply (auto intro!: filtermap-mono) []
  apply (auto simp: le-filter-def eventually-filtermap)
  apply (erule-tac x=λx. P (inv f x) in allE)
  apply auto
  done
lemma filtermap-eq-strong: inj f \implies filtermap f F = filtermap f G \iff F = G
by (simp add: filtermap-mono-strong eq-iff)

lemma filterlim-principal:
(LIM x F. f x :> principal S) \iff (eventually (\lambda x. f x \in S) F)
unfolding filterlim-def eventually-filtermap le-principal ..

lemma filterlim-inf:
(LIM x F1. f x :> inf F2 F3) \iff ((LIM x F1. f x :> F2) \land (LIM x F1. f x :> F3))
unfolding filterlim-def by simp

lemma filterlim-INF:
(LIM x F. f x :> (INF b:B. LIM x F. f x :> G b)) \iff (\forall b\in B. LIM x F. f x :> G b)
unfolding filterlim-def le-INF-iff ..

lemma filterlim-INF-INF:
(\forall j\in J. \exists i\in I. \forall x\in F i. f x :> G j) \iff
(LIM x INF i: I. principal (F i)). f x :> (INF j: J. principal (G j))
proof (subst eventually-INF-base)
fix i j assume i \in I j \in I
with chain[OF this] show \exists x\in I. principal (F x) \leq inf (principal (F i)) (principal (F j))
by auto
qed (auto simp: eventually-principal \( I \neq \{\}\))

lemma filterlim-filtermap: filterlim f F1 (filtermap g F2) = filterlim (\lambda x. f (g x)) F1 F2
unfolding filterlim-def filtermap-filtermap ..

lemma filterlim-sup:
filterlim f F F1 \implies filterlim f F F2 \implies filterlim f F (sup F1 F2)
unfolding filterlim-def filtermap-sup by auto

lemma eventually-sequentially-Suc: eventually (\lambda i. P (Suc i)) sequentially \iff
eventually \( P \) sequentially
unfolding eventually-sequentially by (metis Suc-le-D Suc-le-mono le-Suc-eq)

lemma filterlim-sequentially-Suc:
\[ (\text{LIM} \ x \ \text{sequentially}, \ f (\text{Suc} \ x) :> F) \leftrightarrow (\text{LIM} \ x \ \text{sequentially}, \ f \ x :> F) \]
unfolding filterlim-iff by (subst eventually-sequentially-Suc) simp

lemma filterlim-Suc:
filterlim Suc sequentially sequentially
by (simp add: filterlim-iff eventually-sequentially) (metis le-Suc-eq)

98.7.1 Tendsto
abbreviation (in topological-space)
tendsto :: \( ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{filter} \Rightarrow \text{bool} \) (infixr \( ---> \)) where
\( (f ---> l) \ F \equiv \text{filterlim} \ f \ (\text{nhds} \ l) \ F \)

definition (in t2-space) Lim :: \( 'f \Rightarrow \text{filter} \Rightarrow ('a \Rightarrow \text{filter}) \Rightarrow \text{bool} \) where
\( \text{Lim} \ A \ f = (\text{THE} \ l. \ (f ---> l) \ A) \)

lemma tendsto-eq-rhs: \( (f ---> x) \ F = \Rightarrow x = y = \Rightarrow (f ---> y) \ F \)
by simp

ML \( \langle\langle \text{structure Tendsto-Intros } = \text{Named-Thms} \) \( \\
\text{val name } = @\{\text{binding tendsto-intros}\} \) \( \\
\text{val description } = \text{introduction rules for tendsto} \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \)

setup \( \langle\langle \text{Tendsto-Intros.setup } \#> \) \( \\
\text{Global-Theory.add-thms-dynamic } (@\{\text{binding tendsto-eq-intros}\}, \) \( \\
\text{map-filter } (\text{try } (\text{fn thm } = > @\{\text{thm tendsto-eq-rhs}\} \ \text{OF } [\text{thm}]) \ o \ \text{Tendsto-Intros.get} \ o \ \text{Context.proof-of}); \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \) \( \\
\text{\} \)

lemma (in topological-space) tendsto-def:
\( (f ---> l) \ F \leftrightarrow (\forall S. \ \text{open} \ S \rightarrow l \in S \rightarrow \text{eventually} \ (\lambda x. \ f \ x \in S) \ F) \)
unfolding nhds-def filterlim-INF filterlim-principal by auto

lemma tendsto-mono: \( F \leq F' \Rightarrow (f ---> l) \ F' \Rightarrow (f ---> l) \ F \)
unfolding tendsto-def le-filter-def by fast

lemma tendsto-within-subset: \( (f ---> l) \ (at \ x \ \text{within} \ S) \Rightarrow T \subseteq S \Rightarrow (f ---> l) \ (at \ x \ \text{within} \ T) \)
by (blast intro: tendsto-mono at-le)
lemma \texttt{filterlim-at}:
$$\text{filterlim-at} : (\text{LIM } x. f(x) > at\ b\ \text{within}\ s) \iff (\text{eventually } (\lambda x. f(x) \in s \land f(x) \neq b) \land (f \dashv b) F)$$

by (simp add: at-within-def filterlim-inf filterlim-principal conj-commute)

lemma \texttt{(in topological-space) topological-tendstoI}:
$$(\forall S. \text{open } S \Rightarrow l \in S \Rightarrow \text{eventually } (\lambda x. f(x) \in S) F) \Rightarrow (f \dashv l) F$$

unfolding tendsto-def by auto

lemma \texttt{(in topological-space) topological-tendstoD}:
$$(f \dashv l) F \Rightarrow \text{open } S \Rightarrow l \in S \Rightarrow \text{eventually } (\lambda x. f(x) \in S) F$$

unfolding tendsto-def by auto

lemma \texttt{(in order-topology) order-tendsto-iff}:
$$(f \dashv x) F \iff (\forall l < x. \text{eventually } (\lambda x. l < f(x)) F) \land (\forall u > x. \text{eventually } (\lambda x. f(x) < u) F)$$

unfolding nhds-order filterlim-inf filterlim-INF filterlim-principal by auto

lemma \texttt{(in order-topology) order-tendstoI}:
$$(\forall a. a < y \Rightarrow \text{eventually } (\lambda x. a < f(x)) F) \Rightarrow (\forall a. y < a \Rightarrow \text{eventually } (\lambda x. f(x) < a) F)$$

unfolding order-tendsto-iff by auto

lemma \texttt{tendsto-bot [simp]}: 
$$(f \dashv a) \bot$$

unfolding tendsto-def by simp

lemma \texttt{(in linorder-topology) tendsto-max}:
assumes \texttt{X: (X \dashv x) net}
assumes \texttt{Y: (Y \dashv y) net}
shows \texttt{((\lambda x. \max (X x) (Y x)) \dashv max x y) net}

proof (rule order-tendsto)

fix a assume \texttt{a \in max x y}
then show \texttt{eventually } (\lambda x. a < \max (X x) (Y x)) \texttt{ net}
  using order-tendstoD\(1)[OF X, of a] order-tendstoD\(1)[OF Y, of a]
  by (auto simp: less-max-iff-disj elim: eventually-elim1)

next

fix a assume \texttt{max x y < a}
then show \texttt{eventually } (\lambda x. \max (X x) (Y x) < a) \texttt{ net}
  using order-tendstoD\(2)[OF X, of a] order-tendstoD\(2)[OF Y, of a]
  by (auto simp: eventually-conj-iff)

qed
lemma (in linorder-topology) tendsto-min:
assumes X: (X ----> x) net
assumes Y: (Y ----> y) net
shows ((λx. min (X x) (Y x)) ----> min x y) net
proof (rule order-tendstoI)
  fix a assume a < min x y
  then show eventually (λx. a < min (X x) (Y x)) net
    using order-tendstoD(1)[OF X, of a] order-tendstoD(1)[OF Y, of a]
    by (auto simp: eventually-conj-iff)
next
  fix a assume min x y < a
  then show eventually (λx. min (X x) (Y x) < a) net
    using order-tendstoD(2)[OF X, of a] order-tendstoD(2)[OF Y, of a]
    by (auto simp: min-less-iff-disj elim: eventually-elim1)
qed

lemma tendsto-ident-at [tendsto-intros]: ((λx. x) ----> a) (at a within s)
unfolding tendsto-def eventually-at-topological by auto

lemma (in topological-space) tendsto-const [tendsto-intros]: ((λx. k) ----> k) F
by (simp add: tendsto-def)

lemma (in t2-space) tendsto-unique:
assumes F ≠ bot and (f ----> a) F and (f ----> b) F
shows a = b
proof (rule ccontr)
  assume a ≠ b
  obtain U V where open U open V a ∈ U b ∈ V U ∩ V = {}
    using hausdorff [OF ‹a ≠ b›] by fast
  have eventually (λx. f x ∈ U) F
    using (f ----> a) F [open U] ‹a ∈ U› by (rule topological-tendstoD)
  moreover
  have eventually (λx. f x ∈ V) F
    using (f ----> b) F [open V] ‹b ∈ V› by (rule topological-tendstoD)
  ultimately
  have eventually (λx. False) F
  proof eventually-elim
    case (elim x)
    hence f x ∈ U ∩ V by simp
      with (U ∩ V = {}) show ?case by simp
  qed
  with (¬ trivial-limit F) show False
    by (simp add: trivial-limit-def)
  qed

lemma (in t2-space) tendsto-const-iff:
assumes ¬ trivial-limit F shows ((λx. a :: 'a) ----> b) F ⇔ a = b
by (safe intro!: tendsto-const tendsto-unique [OF assms tendsto-const])
lemma increasing-tendsto:
  fixes f :: \( \cdot \rightarrow \cdot \)
  assumes bdd: eventually (\( \lambda n. \; f \; n < l \)) \( F \)
  and en: \( \forall n. \; x < l \Rightarrow \) eventually (\( \lambda n. \; x < f \; n \)) \( F \)
  shows \( (f \; \longrightarrow \; l) \)
  using assms by (intro order-tendstoI) (auto elim: eventually-elim1)

lemma decreasing-tendsto:
  fixes f :: \( \cdot \rightarrow \cdot \)
  assumes bdd: eventually (\( \lambda n. \; l \leq f \; n \)) \( F \)
  and en: \( \forall x. \; l < x \Rightarrow \) eventually (\( \lambda n. \; f \; n < x \)) \( F \)
  shows \( (f \; \longrightarrow \; l) \)
  using assms by (intro order-tendstoI) (auto elim: eventually-elim1)

lemma tendsto-sandwich:
  fixes f g h :: \( \cdot \Rightarrow \cdot \)
  assumes ev: eventually (\( \lambda n. \; f \; n \leq g \; n \)) \( \text{net} \)
  assumes lim : \( (f \; \longrightarrow c) \) \( \text{net} \)
  shows \( (g \; \longrightarrow c) \)
  proof (rule order-tendstoI)
    assume \( c < a \Rightarrow \) eventually (\( \lambda x. \; a < g \; x \)) \( \text{net} \)
    using order-tendstoD[OF \( \text{lim} (1) \), of a] \( \text{ev} \)
    by (auto elim: eventually-elim2)
  next
    assume \( c < a \Rightarrow \) eventually (\( \lambda x. \; g \; x < a \)) \( \text{net} \)
    using order-tendstoD[OF \( \text{lim} (2) \), of a] \( \text{ev} \)
    by (auto elim: eventually-elim2)
  qed

lemma tendsto-le:
  fixes f g :: \( \cdot \Rightarrow \cdot \)
  assumes F: \( \neg \; \text{trivial-limit} \; F \)
  shows \( y \leq x \)
  proof (rule ccontr)
    assume \( \neg y \leq x \)
    with less-separate[of \( x \; y \)] obtain a b where xe: \( x < a \; b < y \) \( \{ <a \} \cap \{ b <. \} \)
      = \( \{ \} \)
      by (auto simp: not-le)
    then have eventually (\( \lambda x. \; f \; x < a \)) \( F \)
      eventually (\( \lambda x. \; b < g \; x \)) \( F \)
      using xe \( \text{by} \) (auto intro: order-tendstoD)
      by eventually-elim (insert xe, fastforce)
    show False
      by (simp add: eventually-False)
  qed

lemma tendsto-le-const:
  fixes f :: \( \cdot \Rightarrow \cdot \)

assumes $F$: ¬ trivial-limit $F$
assumes $x$: $(f \longrightarrow x) F$ and $a$: eventually $(\lambda i. a \leq f i) F$
shows $a \leq x$
using $F x \text{ tendsto-const } a$ by (rule tendsto-le)

lemma tendsto-ge-const:
fixes $f :: 'a \Rightarrow 'b::linorder-topology$
assumes $F$: ¬ trivial-limit $F$
assumes $x$: $(f \longrightarrow x) F$ and $a$: eventually $(\lambda i. a \geq f i) F$
shows $a \geq x$
by (rule tendsto-le [OF $F$ tendsto-const $x$ $a$])

#### 98.7.2 Rules about Lim

lemma tendsto-Lim:
\[ \neg(\text{trivial-limit net}) \implies (f \longrightarrow l) \text{ net } \implies \text{Lim net } f = l \]
unfolding Lim-def using tendsto-unique [of net $f$] by auto

lemma Lim-ident-at: ¬ trivial-limit $(at x within s)$ $\implies$ Lim $(at x within s) (\lambda x. x) = x$
by (rule tendsto-Lim [OF - tendsto-ident-at]) auto

#### 98.8 Limits to at-top and at-bot

lemma filterlim-at-top:
fixes $f :: 'a \Rightarrow ('b::linorder)$
sshows $(\text{LIM x } F. f x :> \text{at-top}) \iff (\forall Z. \text{eventually } (\lambda x. Z \leq f x) F)$
by (auto simp: filterlim-iff eventually-at-top-linorder elim: eventually-elim1)

lemma filterlim-at-top-mono:
$\text{LIM x } F. f x :> \text{at-top } \implies \text{eventually } (\lambda x. f x \leq (g x ::'a::linorder)) F \implies \text{LIM x } F. g x :> \text{at-top}$
by (auto simp: filterlim-at-top elim: eventually-elim2 intro: order-trans)

lemma filterlim-at-top-dense:
fixes $f :: 'a \Rightarrow (b::unbounded-dense-linorder)$
sshows $(\text{LIM x } F. f x :> \text{at-top}) \iff (\forall Z. \text{eventually } (\lambda x. Z < f x) F)$
by (metis eventually-elim1 [of F] filterlim-at-top mono eventually-at-top-linorder elim: eventually-elim2 intro: order-trans)

lemma filterlim-at-top-ge:
fixes $f :: 'a \Rightarrow (b::linorder)$ and $c :: 'b$
sshows $(\text{LIM x } F. f x :> \text{at-top}) \iff (\forall Z \geq c. \text{eventually } (\lambda x. Z \leq f x) F)$
unfolding at-top-sub[of $c$] filterlim-INF by (auto simp add: filterlim-principal)

lemma filterlim-at-top-at-top:
fixes $f :: 'a::linorder \Rightarrow 'b::linorder$
assumes mono: $\bigwedge x y. Q x \Longrightarrow Q y \Longrightarrow x \leq y \Longrightarrow f x \leq f y$
assumes bij: $\bigwedge x. P x \Longrightarrow f (g x) = x$ $\bigwedge x. P x \Longrightarrow Q (g x)$
assumes $Q$: eventually $Q$ at-top
assumes $P$: eventually $P$ at-top
shows filterlim $f$ at-top at-top
proof
from $P$ obtain $x$ where $x: \forall y. x \leq y \implies P y$
unfolding eventually-at-top-linorder by auto
show ?thesis
proof (intro filterlim-at-top-ge [THEN iffD2] allI impI)
fix $z$
assume $x \leq z$
with $x$
have $P z$ by auto
have eventually $(\lambda x. g z \leq x)$ at-top
by (rule eventually-ge-at-top)
with $Q$
show eventually $(\lambda x. z \leq f x)$ at-top
by eventually-elim (metis mono bij $P z$
qed
qed

lemma filterlim-at-top-gt:
fixes $f :: 'a \Rightarrow ('b::{unbounded-dense-linorder})$
and $c :: 'b$
shows $(\text{LIM } x F. f x > \text{at-top}) \iff (\forall Z > c. \text{eventually } (\lambda x. Z \leq f x) F)$
by (metis filterlim-at-top order-less-le-trans gt-ex filterlim-at-top-ge)

lemma filterlim-at-bot:
fixes $f :: 'a \Rightarrow ('b::{linorder})$
and $c :: 'b$
shows $(\text{LIM } x F. f x > \text{at-bot}) \iff (\forall Z. \text{eventually } (\lambda x. f x \leq Z) F)$
by (auto simp add: filterlim-at-bot dense at-top-le)

lemma filterlim-at-bot-dense:
fixes $f :: 'a \Rightarrow ('b::{linorder, no-bot})$
and $c :: 'b$
shows $(\text{LIM } x F. f x > \text{at-bot}) \iff (\forall Z. \text{eventually } (\lambda x. f x < Z) F)$
proof (auto simp add: filterlim-at-bot [of f F])
fix $Z :: 'b$
from lt-ex [of $Z$] obtain $Z'$ where $I: Z' < Z$
assume $\forall Z. \text{eventually } (\lambda x. f x \leq Z) F$
hence eventually $(\lambda x. f x \leq Z') F$ by auto
thus eventually $(\lambda x. f x < Z) F$
apply (rule eventually-mono [rotated])
using $I$ by auto
next
fix $Z :: 'b$
show $\forall Z. \text{eventually } (\lambda x. f x \leq Z) F \implies \text{eventually } (\lambda x. f x < Z) F$
by (drule spec [of $- Z$], erule eventually-mono [rotated], auto simp add: less-imp-le)
qed

lemma filterlim-at-bot-le:
fixes $f :: 'a \Rightarrow ('b::linorder)$
and $c :: 'b$
shows $(\text{LIM } x F. f x > \text{at-bot}) \iff (\forall Z \leq c. \text{eventually } (\lambda x. Z \geq f x) F)$
unfolding filterlim-at-bot
proof safe
THEORY "Topological-Spaces"

lemma filterlim-at-bot-at-right:
  assumes mono: \( \forall x. \, Q x \implies Q y \implies x \leq y \implies f x \leq f y \)
  assumes bij: \( \forall x. \, P x \implies f (g x) = x \land x. \, P x \implies Q (g x) \)
  assumes Q: eventually \( Q \) (at-right \( a \)) and bound: \( \forall b. \, Q b \implies a < b \)
  assumes P: eventually \( P \) at-bot
  shows filterlim \( f \) at-bot (at-right \( a \))
proof -
  from \( P \) obtain \( x \) where \( x. \, \forall y. \, y \leq x \implies P y \)
  unfolding eventually-at-bot-linorder by auto
  show \( \? \)thesis
  proof (intro filterlim-at-bot-le[THEN iffD2] allI implI)
    fix \( z \) assume \( z \leq x \)
    with \( x \) have \( P z \) by auto
    have eventually \( (\lambda x. \, x \leq g z) \) (at-right \( a \))
      using bound[OF bij(2)[OF \( P z \)]]
    unfolding eventually-at-right[OF bound[OF bij(2)[OF \( P z \)]]] by (auto intro!::
      ext[of \(- g z\)])
    with \( Q \) show eventually \( (\lambda x. \, f x \leq z) \) (at-right \( a \))
      by eventually-elim (metis bij[OF \( P z \); mono])
  qed

lemma filterlim-at-top-at-left:
  fixes \( f \) :: \('a::linorder-topology \Rightarrow 'b::linorder\)
  assumes mono: \( \forall x. \, Q x \implies Q y \implies x \leq y \implies f x \leq f y \)
  assumes bij: \( \forall x. \, P x \implies f (g x) = x \land x. \, P x \implies Q (g x) \)
  assumes Q: eventually \( Q \) (at-left \( a \)) and bound: \( \forall b. \, Q b \implies b < a \)
  assumes P: eventually \( P \) at-top
  shows filterlim \( f \) at-top (at-left \( a \))
proof -
  from \( P \) obtain \( x \) where \( x. \, \forall y. \, x \leq y \implies P y \)
  unfolding eventually-at-top-linorder by auto
  show \( \? \)thesis
  proof (intro filterlim-at-top-le[THEN iffD2] allI implI)
    fix \( z \) assume \( x \leq z \)
    with \( z \) have \( P z \) by auto
    have eventually \( (\lambda x. \, g z \leq x) \) (at-left \( a \))
      using bound[OF bij(2)[OF \( P z \)]]
THEORY “Topological-Spaces”

unfolding eventually-at-left[OF bound[OF bij(2)][OF ⟨P z⟩]] by (auto intro!: exI[of - g z])
with Q show eventually (λx. z ≤ f x) (at-left a)
  by eventually-elim (metis bij ⟨P z⟩ mono)
qed

lemma filterlim-split-at:
filterlim f F (at-left x) =⇒ filterlim f F (at-right x) =⇒ filterlim f F (at x::'a::linorder-topology))
by (subst at-eq-sup-left-right) (rule filterlim-sup)

lemma filterlim-at-split:
filterlim f F (at x::'a::linorder-topology)) ←→ filterlim f F (at-left x) ∧ filterlim f F (at-right x)
by (subst at-eq-sup-left-right) (simp add: filterlim-def filtermap-sup)

lemma eventually-nhds-top:
fixes P :: 'a::{order-top, linorder-topology} ⇒ bool
assumes (b::'a) < top
shows eventually P (nhds top) ←→ (∃ b<top. (∀ z. b < z → P z))
unfolding eventually-nhds
proof safe
fix S :: 'a set
assume open S top ∈ S
note open-left[OF this ⟨b<top⟩]
moreover assume ∀ s∈S. P s
ultimately show ∃ b<top. ∀ z>b. P z
  by (auto simp: subset-eq Ball-def)
next
fix b assume b < top ∀ z>b. P z
then show ∃ S. open S ∧ top ∈ S ∧ (∀ xa∈S. P xa)
  by (intro exI[of - {b<..}]) auto
qed

lemma tendsto-at-within-iff-tendsto-nhds:
(g −−−→ g l) (at l within S) ←→ (g −−−→ g l) (inf (nhds l) (principal S))
unfolding tendsto-def eventually-at-filter eventually-inf-principal
by (intro ext all-cong imp-cong) (auto elim!: eventually-elim1)

98.9 Limits on sequences
abbreviation (in topological-space)
LIMSEQ :: [nat ⇒ 'a] ⇒ bool
((⟨-⟩/ −−−→ ⟨-⟩) [60, 60] 60) where
X −−−→ L ≡ (X −−−→ L) sequentially

abbreviation (in t2-space) lim :: (nat ⇒ 'a) ⇒ 'a where
lim X ≡ Lim sequentially X

definition (in topological-space) convergent :: (nat ⇒ 'a) ⇒ bool where
convergent \( X = (\exists L. X \longrightarrow L) \)

**lemma** \( \text{lim-def: lim} X = (\text{THE} L. X \longrightarrow L) \)

**unfolding** \( \text{lim-def} \) ..

### 98.9.1 Monotone sequences and subsequences

**definition**

\[ \text{monoseq :: (nat \Rightarrow 'a::order) \Rightarrow bool where} \]

— Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

\[ \text{monoseq} X = ((\forall m. \forall n \geq m. X m \leq X n) \lor (\forall m. \forall n \geq m. X n \leq X m)) \]

**abbreviation** \( \text{incseq :: (nat \Rightarrow 'a::order) \Rightarrow bool where} \)

\[ \text{incseq} X \equiv \text{mono} X \]

**lemma** \( \text{incseq-def: incseq} X \iff (\forall m. \forall n \geq m. X n \geq X m) \)

**unfolding** \( \text{mono-def} \) ..

**abbreviation** \( \text{decseq :: (nat \Rightarrow 'a::order) \Rightarrow bool where} \)

\[ \text{decseq} X \equiv \text{antimono} X \]

**lemma** \( \text{decseq-def: decseq} X \iff (\forall m. \forall n \geq m. X n \leq X m) \)

**unfolding** \( \text{antimono-def} \) ..

**definition**

\[ \text{subseq :: (nat \Rightarrow nat) \Rightarrow bool where} \]

— Definition of subsequence

\[ \text{subseq} f \iff (\forall m. \forall n \gt m. f m \lt f n) \]

**lemma** \( \text{incseq-SucI:} \)

\[ (\forall n. X n \leq X (\text{Suc} n)) \implies \text{incseq} X \]

**using** \( \text{lift-Suc-mono-le[of} X] \)

**by** \( \text{(auto simp: incseq-def)} \)

**lemma** \( \text{incseqD:} \)

\[ (\forall i j. \text{incseq} f \implies i \leq j \implies f i \leq f j) \]

**by** \( \text{(auto simp: incseq-def)} \)

**lemma** \( \text{incseq-SucD:} \)

\[ \text{incseq} A \implies A i \leq A (\text{Suc} i) \]

**using** \( \text{incseqD[of} A i \text{Suc} i] \) by \text{auto}

**lemma** \( \text{incseq-Suc iff:} \)

\[ \text{incseq} f \iff (\forall n. f n \leq f (\text{Suc} n)) \]

**by** \( \text{(auto intro: incseq-SucI dest: incseq-SucD)} \)

**lemma** \( \text{incseq-const[simp, intro]:} \)

\[ \text{incseq} (\lambda x. k) \]

**unfolding** \( \text{incseq-def by auto} \)

**lemma** \( \text{decseq-SucI:} \)
\[ (\forall n. X (\text{Suc } n) \leq X n) \implies \text{decseq } X \]

using \texttt{order.lift-Suc-mono-le}\{OF dual-order, of X\]

by \texttt{(auto simp: decseq-def)}

\begin{itemize}
  \item \textbf{lemma \texttt{decseqD}}: \( \forall i j. \text{decseq } f \implies i \leq j \implies f j \leq f i \)
  \begin{itemize}
    \item \texttt{(auto simp: decseq-def)}
  \end{itemize}
  \item \textbf{lemma \texttt{decseq-SucD}}: \( \text{decseq } A \implies A (\text{Suc } i) \leq A i \)
  \begin{itemize}
    \item using \texttt{decseqD[of A i Suc i] by auto}
  \end{itemize}
  \item \textbf{lemma \texttt{decseq-Suc-iff}}: \( \text{decseq } f \iff (\forall n. f (\text{Suc } n) \leq f n) \)
      \begin{itemize}
        \item \texttt{by (auto intro: decseq-SucI dest: decseq-SucD)}
      \end{itemize}
  \item \textbf{lemma \texttt{monoseq-iff}}: \( \text{monoseq } X \iff \text{incseq } X \lor \text{decseq } X \)
      \begin{itemize}
        \item \texttt{unfolding monoseq-def incseq-def decseq-def} 
      \end{itemize}
  \item \textbf{lemma \texttt{monoseq-Suc}}:
      \begin{itemize}
        \item \( \text{monoseq } X \iff (\forall n. X n \leq X (\text{Suc } n)) \lor (\forall n. X (\text{Suc } n) \leq X n) \)
        \item \texttt{unfolding monoseq-iff incseq-Suc-iff decseq-Suc-iff} 
      \end{itemize}
  \item \textbf{lemma \texttt{monoI1}}: \( \forall m. \forall n \geq m. X m \leq X n \implies \text{monoseq } X \)
      \begin{itemize}
        \item \texttt{by simp add: monoseq-def} 
      \end{itemize}
  \item \textbf{lemma \texttt{monoI2}}: \( \forall m. \forall n \geq m. X n \leq X m \implies \text{monoseq } X \)
      \begin{itemize}
        \item \texttt{by simp add: monoseq-def} 
      \end{itemize}
  \item \textbf{lemma \texttt{mono-SucI1}}: \( \forall n. X n \leq X (\text{Suc } n) \implies \text{monoseq } X \)
      \begin{itemize}
        \item \texttt{by simp add: monoseq-Suc} 
      \end{itemize}
  \item \textbf{lemma \texttt{mono-SucI2}}: \( \forall n. X (\text{Suc } n) \leq X n \implies \text{monoseq } X \)
      \begin{itemize}
        \item \texttt{by simp add: monoseq-Suc} 
      \end{itemize}
  \item \textbf{lemma \texttt{monoseq-minus}}:
      \begin{itemize}
        \item \texttt{fixes} \( a :: \text{nat} \rightarrow 'a::ordered-ab-group-add \)
        \item \texttt{assumes} \( \text{monoseq } a \)
        \item \texttt{shows} \( \text{monoseq } (\lambda n. - a n) \)
        \item \texttt{proof} \( \texttt{(cases } \forall m. \forall n \geq m. a m \leq a n) \)
          \begin{itemize}
            \item \texttt{case True}
              \begin{itemize}
                \item \texttt{hence } \forall m. \forall n \geq m. - a n \leq - a m \texttt{ by auto}
              \end{itemize}
            \item \texttt{thus } ?thesis \texttt{ by (rule monoI2)}
          \end{itemize}
          \begin{itemize}
            \item \texttt{next}
              \begin{itemize}
                \item \texttt{case False}
                  \begin{itemize}
                    \item \texttt{hence } \forall m. \forall n \geq m. - a m \leq - a n \texttt{ using monoseq a[unfolded monoseq-def]}
                    \item \texttt{by auto}
                    \item \texttt{thus } ?thesis \texttt{ by (rule monoI1)}
                  \end{itemize}
              \end{itemize}
          \end{itemize}
          \begin{itemize}
            \item \texttt{qed}
          \end{itemize}
      \end{itemize}
\end{itemize}
Subsequence (alternative definition, e.g. Hoskins)

**lemma** subseq-Suc-iff: subseq f = (∀ n. (f n) < (f (Suc n)))

**apply** (simp add: subseq-def)

**apply** (auto dest!: less-imp-Suc-add)

**apply** (induct-tac k)

**apply** (auto intro: less-trans)

**done**

for any sequence, there is a monotonic subsequence

**lemma** seq-monosub:

fixes s :: nat =⇒ 'a::linorder

shows ∃ f. subseq f ∧ monoseq (λ n. (s (f n)))

**proof** cases

assume (∀ n. ∃ p>n. ∀ m≥p. s m ≤ s p)

then have ∃ f. ∀ n. (∀ m≥f n. s m ≤ s (f n)) ∧ f n < f (Suc n)

by (intro dependent-nat-choice) (auto simp: conj-commute)

then obtain f where subseq f and mono: ∀ n m. f n ≤ m ⇒ s m ≤ s (f n)

moreover then have incseq f

unfolding subseq-Suc-iff incseq-Suc-iff by (auto intro: less-imp-le)

then have monoseq (λ n. s (f n))

by (auto simp add: incseq-def intro!: mono monoI2)

ultimately show ?thesis

by auto

next

assume ¬ (∀ n. ∃ p>n. (∀ m≥p. s m ≤ s p))

then obtain N where N: ∀ p > N ⇒ ∃ m≥p. s p < s m by (force simp: not-le le-_less)

have ∃ f. ∀ n. N < f n ∧ f n < f (Suc n) ∧ s (f n) ≤ s (f (Suc n))

proof (intro dependent-nat-choice)

fix x assume N < x with N[of x] show ∃ y>N. x < y ∧ s x ≤ s y

by (auto intro: less-trans)

qed auto

then show ?thesis

by (auto simp: monoseq-iff incseq-Suc-iff subseq-Suc-iff)

qed

**lemma** seq-suble: assumes sf: subseq f shows n ≤ f n

**proof** (induct n)

**case** 0 thus ?case by simp

**next**

**case** (Suc n)

from sf[unfolded subseq-Suc-iff, rule-format, of n] Suc.hyps

have n < f (Suc n) by arith

thus ?case by arith

qed

**lemma** eventually-subseq:
subseq r ↦ eventually P sequentially ↦ eventually (λn. P (r n)) sequentially
unfolding eventually-sequentially by (metis seq-suble le-trans)

lemma not-eventually-sequentiallyD:
assumes P: ¬ eventually P sequentially
shows ∃ r. subseq r ∧ (∀ n. ¬ P (r n))
proof
  from P have ∀ n. ∃ m ≥ n. ¬ P m
  unfolding eventually-sequentially by (simp add: not-less)
  then obtain r where ∃ n. r n ≥ n ∧ n. ¬ P (r n)
  by (auto simp: choice-iff)
  then show thesis by (auto intro!: exI
                      of -λn. r ((Suc ◦ r) ^^ Suc n) 0)
 (simp: less-eq-Suc-le subseq-Suc-iff)
qed

lemma filterlim-subseq: subseq f ⇒ filterlim f sequentially sequentially
unfolding filterlim-iff by (metis eventually-subseq)

lemma subseq-o: subseq r ⇒ subseq s ⇒ subseq (r ◦ s)
unfolding subseq-def by simp

lemma subseq-mono: assumes subseq r m < n shows r m < r n
using asms by (auto simp: subseq-def)

lemma incseq-imp-monoseq: incseq X ⇒ monoseq X
by (simp add: incseq-def monoseq-def)

lemma decseq-imp-monoseq: decseq X ⇒ monoseq X
by (simp add: decseq-def monoseq-def)

lemma decseq-eq-incseq:
fixes X :: nat ⇒ 'a::ordered-ab-group-add
shows decseq X = incseq (λn. - X n)
by (simp add: decseq-def incseq-def)

lemma INT-decseq-offset:
assumes decseq F
shows (⋂ i. F i) = (⋂ i∈{n..}. F i)
proof safe
fix i assume x: x ∈ (⋂ i∈{n..}. F i)
show x ∈ F i
proof cases
  from x have x ∈ F n by auto
  also assume i ≤ n with (decseq F) have F n ⊆ F i
  unfolding decseq-def by simp
finally show thesis.
qed (insert x, simp)
qed auto
lemma **LIMSEQ-const-iff**:  
fixes \( k \) \( l \) :: 'a::t2-space  
shows \((\lambda n. \, k) \longorder{\longrightarrow} l \iff k = l\)  
using trivial-limit-sequentially by (rule tendsto-const-iff)

lemma **LIMSEQ-SUP**:  
incseq X \longorder{\implies} X \longorder{\longrightarrow} (SUP i. \, X i :: 'a :: {complete-linorder, linorder-topology})  
by (intro increasing-tendsto)  
(auto simp : SUP-upper less-SUP-iff incseq-def eventually-sequentially intro: less-le-trans)

lemma **LIMSEQ-INF**:  
decseq X \longorder{\implies} X \longorder{\longrightarrow} (INF i. \, X i :: 'a :: {complete-linorder, linorder-topology})  
by (intro decreasing-tendsto)  
(auto simp : INF-lower INF-less-iff decseq-def eventually-sequentially intro: le-less-trans)

lemma **LIMSEQ-ignore-initial-segment**:  
f \longorder{\longrightarrow} a \implies (\lambda n. \, f \,(n + k)) \longorder{\longrightarrow} a  
unfolding tendsto-def  
by (subst eventually-sequentially-seg[where \( k=k \)])

lemma **LIMSEQ-offset**:  
(\lambda n. \, f \,(n + k)) \longorder{\longrightarrow} a \implies f \longorder{\longrightarrow} a  
unfolding tendsto-def  
by (subst (asm) eventually-sequentially-seg[where \( k=k \)])

lemma **LIMSEQ-Suc**:  
f \longorder{\longrightarrow} l \implies (\lambda n. \, f \,(Suc n)) \longorder{\longrightarrow} l  
by (drule-tac \( k=Suc \, 0 \) in LIMSEQ-ignore-initial-segment, simp)

lemma **LIMSEQ-imp-Suc**:  
(\lambda n. \, f \,(Suc n)) \longorder{\longrightarrow} l \implies f \longorder{\longrightarrow} l  
by (rule-tac \( k=Suc \, 0 \) in LIMSEQ-offset, simp)

lemma **LIMSEQ-Suc-iff**:  
(\lambda n. \, f \,(Suc n)) \longorder{\longrightarrow} l = f \longorder{\longrightarrow} l  
by (blast intro: LIMSEQ-imp-Suc LIMSEQ-Suc)

lemma **LIMSEQ-unique**:  
fixes \( a \) \( b \) :: 'a::t2-space  
shows [\( X \longorder{\longrightarrow} a \); \( X \longorder{\longrightarrow} b \)] \implies a = b  
using trivial-limit-sequentially by (rule tendsto-unique)

lemma **LIMSEQ-le-const**:  
[\( X \longorder{\longrightarrow} x; \, x :: 'a::linorder-topology \); \( \exists \, N. \, \forall \, n \geq N. \, a \leq X \, n \)] \implies a \leq x  
using tendsto-le-const[of sequentially \( X \, x \)] by (simp add: eventually-sequentially)

lemma **LIMSEQ-le**:  
[\( X \longorder{\longrightarrow} x; \, Y \longorder{\longrightarrow} y; \, \exists \, N. \, \forall \, n \geq N. \, X \, n \leq Y \, n \)] \implies x \leq (y :: 'a::linorder-topology)  
using tendsto-le[of sequentially \( Y \, Y \, X \, x \)] by (simp add: eventually-sequentially)
lemma LIMSEQ-le-const2: 
\[ [X \longrightarrow (x::'a::linorder-topology); \exists N. \forall n \geq N. X n \leq a] \implies x \leq a \]
by (rule LIMSEQ-le[of X x \lambda n. a]) (auto simp: tendsto-const)

lemma convergentD: convergent X \implies \exists L. (X \longrightarrow L)
by (simp add: convergent_def)

lemma convergentI: (X \longrightarrow L) \implies convergent X
by (auto simp add: convergent_def)

lemma convergent-LIMSEQ-iff: convergent X = (X \longrightarrow \lim X)
by (auto intro: theI LIMSEQ_unique simp add: convergent_def lim_def)

lemma convergent-const: convergent (\lambda n. c)
by (rule convergentI, rule tendsto-const)

lemma monoseq-le:
monoseq a \implies a \longrightarrow (x::'a::linorder-topology) \implies
(\forall n. a n \leq x) \land (\forall m. \forall n \geq m. a m \leq a n) \lor (\forall n. x \leq a n) \land (\forall m.
\forall n \geq m. a n \leq a m))
by (metis LIMSEQ-le-const LIMSEQ-le-const2 decseq-def incseq-def monoseq-iff)

lemma LIMSEQ-subseq-LIMSEQ:
\[ [X \longrightarrow L; \text{subseq } f ] \implies (X o f) \longrightarrow L \]
unfolding comp-def by (rule filterlim-compose[of X, OF - filterlim-subseq])

lemma convergent-subseq-convergent:
\[ [\text{convergent } X; \text{subseq } f ] \implies \text{convergent } (X o f) \]
unfolding convergent-def by (auto intro: LIMSEQ-subseq-LIMSEQ)

lemma limI: X \longrightarrow L \implies \lim X = L
by (rule tendsto-Lim) (rule trivial-limit-sequentially)

lemma lim-le: convergent f \implies (\forall n. f n \leq (x::'a::linorder-topology)) \implies \lim f \leq x
using LIMSEQ-le-const2[of f \lim f x] by (simp add: convergent-LIMSEQ-iff)

98.9.2 Increasing and Decreasing Series

lemma incseq-le: incseq X \implies X \longrightarrow L \implies X n \leq (L::'a::linorder-topology)
by (metis incseq-def LIMSEQ-le-const)

lemma decseq-le: decseq X \implies X \longrightarrow L \implies (L::'a::linorder-topology) \leq X
by (metis decseq-def LIMSEQ-le-const2)

98.10 First countable topologies

class first-countable-topology = topological-space +
assumes first-countable-basis:
\[ \exists A :: \text{nat} \Rightarrow 'a \text{ set} \ (\forall i. \ x \in A \ i \land \text{open} (A \ i)) \land (\forall S. \ \text{open} S \land x \in S \longrightarrow (\exists i. \ A \ i \subseteq S)) \]

lemma (in first-countable-topology) countable-basis-at-decseq:
\[ \text{obtains } A :: \text{nat} \Rightarrow 'a \text{ set where} \]
\[ \land i. \ \text{open} (A \ i) \land i. \ x \in (A \ i) \]
\[ \land S. \ \text{open} S \Longrightarrow x \in S \Longrightarrow \text{eventually} (\lambda i. \ A \ i \subseteq S) \text{ sequentially} \]

proof atomize-elim

from first-countable-basis[of x] obtain A :: nat \Rightarrow 'a set where
\[ \land i. \ \text{open} (A \ i) \land i. \ x \in A \ i \]
and incl: \[ \land S. \ \text{open} S \Longrightarrow x \in S \Longrightarrow \exists i. \ A \ i \subseteq S \text{ by auto} \]
def F \equiv \lambda n. \bigcap i \leq n. \ A \ i

show \[ \exists A. (\forall i. \ \text{open} (A \ i)) \land (\forall i. \ x \in A \ i) \land \]
\[ (\forall S. \ \text{open} S \longrightarrow x \in S \longrightarrow \text{eventually} (\lambda i. \ A \ i \subseteq S) \text{ sequentially}) \]

proof (safe intro!: exI[of - F])

fix i

show open (F i) using nhds(1) by (auto simp: F-def)

show x \in F i using nhds(2) by (auto simp: F-def)

next

fix S assume open S x \in S

from incl[OF this] obtain i where F i \subseteq S unfolding F-def by auto

moreover have \[ \land j. \ i \leq j \Longrightarrow F \ j \subseteq F \ i \]
by (auto simp: F-def)

ultimately show eventually (\lambda i. F i \subseteq S) sequentially
by (auto simp: eventually-sequentially)

qed

qed

lemma (in first-countable-topology) nhds-countable:

obtains X :: nat \Rightarrow 'a set
where decseq X \[ \land n. \ \text{open} (X \ n) \land n. \ x \in X \ n \ \text{nhds} x = (\text{INF} n. \ \text{principal} (X \ n)) \]

proof

from first-countable-basis obtain A :: nat \Rightarrow 'a set
where A: \[ \land n. \ x \in A \ n \land n. \ \text{open} (A \ n) \land S. \ \text{open} S \Longrightarrow x \in S \Longrightarrow \exists i. \ A \ i \subseteq S \]

by metis

show thesis

proof

show decseq (\lambda n. \bigcap i \leq n. A \ i)
by (auto simp: decseq-def)

show \[ \land n. \ x \in (\bigcap i \leq n. A \ i) \land n. \ \text{open} (\bigcap i \leq n. A \ i) \]
using A by auto

show nhds x = (\text{INF} n. \ \text{principal} (\bigcap i \leq n. A \ i))
using A unfolding nhds-def
apply (intro INF-eq)
apply simp-all
apply force
apply (intro exI[of - Int i≤n. A i for n] conjI open-INT)
apply auto
done
qed
qed

lemma (in first-countable-topology) countable-basis:
  obtains A :: nat ⇒ 'a set where
  ∀ i. open (A i) ∧ i ∈ A i
  ∀ F. (∀ n. F n ∈ A n) ⇒ F ----→ x
proof atomize-elim
  obtain A :: nat ⇒ 'a set where
    A: ∀ i. open (A i)
    ∀ i. x ∈ A i
    ∀ S. open S ⇒ x ∈ S ⇒ eventually (λi. A i ⊆ S) sequentially
    by (rule countable-basis-at-decseq) blast
  { fix S assume ∀ n. F n ∈ A n open S x ∈ S
    with A(3)[of S] have eventually (λn. F n ∈ S) sequentially
      by (auto elim: eventually-elim1 simp: subset-eq)
  }
  with A show ∃ A. (∀ i. open (A i)) ∧ (∀ i. x ∈ A i) ∧ (∀ F. (∀ n. F n ∈ A n)
    ⇒ F ----→ x)
    by (intro exI[of - A]) (auto simp: tendsto-def)
qed

lemma (in first-countable-topology) sequentially-imp-eventually-nhds-within:
  assumes ∀ f. (∀ n. f n ∈ s) ∧ f ----→ a ⇒ eventually (λn. P (f n)) sequentially
  shows eventually P (inf (nhds a) (principal s))
proof (rule ccontr)
  obtain A :: nat ⇒ 'a set where
    A: ∀ i. open (A i)
    ∀ i. A i ∈ A i
    ∀ F. ∀ n. F n ∈ A n ⇒ F ----→ a
    by (rule countable-basis) blast
  assume ¬ thesis
  with A have P: ∃ F. ∀ n. F n ∈ s ∧ F n ∈ A n ∧ ¬ P (F n)
    unfolding eventually-inf-principal eventually-nhds by (intro choice) fastforce
  then obtain F where F0: ∀ n. F n ∈ s and F2: ∀ n. F n ∈ A n and F3: ∀ n.
    ¬ P (F n)
    by blast
  with A have F ----→ a by auto
  hence eventually (λn. P (F n)) sequentially
    using asms F0 by simp
  thus False by (simp add: F3)
qed

lemma (in first-countable-topology) eventually-nhds-within-iff-sequentially:

eventually $P \left( \inf(\text{nhds } a) (\text{principal } s) \right) \leftrightarrow$

$(\forall f \quad \forall n \quad f_n \in s) \quad \forall \lambda n. \quad f_n \in s \quad \Rightarrow\quad f \longrightarrow a \quad \Rightarrow\quad \text{eventually} \quad (\lambda n. \quad P (f_n)) \quad \text{sequentially}$

**proof** (safe intro; sequentially-impl-eventually-nhds-within)

**assume** eventually $P \left( \inf(\text{nhds } a) (\text{principal } s) \right)$

then obtain $S$ where $\text{open } a \in S \forall x \in S. \quad x \in s \longrightarrow P x$

by (auto simp: eventually-inf-principal eventually-nhds)

moreover fix $f$

**assume** $\forall n. \quad f_n \in s \quad \Rightarrow\quad f \longrightarrow a \quad \Rightarrow\quad \text{eventually} \quad (\lambda n. \quad P (f_n)) \quad \text{sequentially}$

by (auto dest!: topological-tendstoD elim: eventually-elim1)

qed

**lemma** (in first-countable-topology) eventually-nhds-iff-sequentially:

eventually $P \left( \text{nhds } a \right)$ $\leftrightarrow$ $(\forall f \quad \forall n. \quad f_n \in s \quad \Rightarrow\quad f \longrightarrow a \quad \Rightarrow\quad \text{eventually} \quad (\lambda n. \quad P (f_n)) \quad \text{sequentially})$

**using** eventually-nhds-within-iff-sequentially[of $P \ a \ \text{UNIV}$] by simp

**lemma** tendsto-at-iff-sequentially:

fixes $f :: a :: \text{first-countable-topology}$

shows $(f \longrightarrow a) \ (\text{at } x \ \text{within } s) \leftrightarrow (\forall X. \ (\forall i. \ X_i \in s \quad \Rightarrow\quad X \longrightarrow x \quad \Rightarrow\quad (f \circ X) \longrightarrow a))$

unfolding filterlim-def[of - nhds $a$] le-filter-def eventually-filtermap at-within-def eventually-nhds-within-iff-sequentially comp-def

by metis

98.11 Function limit at a point

**abbreviation**

$LIM :: \ ('a::\text{topological-space} \Rightarrow \ 'b::\text{topological-space}) \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow \ bool$

$\quad$ $f \quad \text{is} \quad L \equiv (f \longrightarrow L) \ (\text{at } a)$

**lemma** tendsto-within-open: $a \in S \Rightarrow \text{open } S \Rightarrow (f \longrightarrow l) \ (\text{at } a \ \text{within } S)$

$\leftrightarrow (f \quad \text{is} \quad l) \quad \text{at-within-open}[\text{where } S=S]$

**lemma** $LIM\text{-const-not-eq}[\text{tendsto-intros]}$

fixes $a :: \ 'a::\text{perfect-space}$

fixes $k L :: \ 'b::\text{t2-space}$

shows $k \neq L \Rightarrow \ (\lambda x. \ k) \quad \text{is} \quad L$

by (simp add: tendsto-const-iff)

**lemmas** $LIM\text{-const-not-zero} = LIM\text{-const-not-eq} \ [\text{where } \ L=0]$

**lemma** $LIM\text{-const-eq}$

fixes $a :: \ 'a::\text{perfect-space}$

fixes $k L :: \ 'b::\text{t2-space}$

shows $(\lambda x. \ k) \quad \text{is} \quad L \quad \Rightarrow\quad k = L$

by (simp add: tendsto-const-iff)
lemma LIM-unique:
fixes a :: "a::perfect-space" and L M :: "'b::t2-space"
shows \( f \dashv a \dashv \rightarrow L \Longrightarrow f \dashv a \dashv \rightarrow M \Longrightarrow L = M \)
using at-neq-bot by (rule tendsto-unique)

Limits are equal for functions equal except at limit point

lemma LIM-equal: \( \forall x \cdot x \neq a \dashv \rightarrow (f x = g x) =\Longrightarrow (f \dashv a \dashv \rightarrow l) \iff (g \dashv a \dashv \rightarrow l) \)

unfolding tendsto-def eventually-at-topological by simp

lemma LIM-cong-limit: \( f \dashv x \dashv \rightarrow L = L = \Longrightarrow (f \dashv x \dashv \rightarrow K) \iff (g \dashv b \dashv \rightarrow m) \)
by (simp add: LIM-equal)

lemma tendsto-at-iff-tendsto-nhds:
\( g \dashv l \dashv \rightarrow g l \iff (g \dashv \rightarrow g l) \) (nhds l)

unfolding tendsto-def eventually-at-filter
by (intro ext all-cong imp-cong) (auto elim!: eventually-elim1)

lemma tendsto-compose:
\( g \dashv l \dashv \rightarrow g l \Longrightarrow (f \dashv \rightarrow l) F = (\lambda x \cdot g (f x)) \dashv \rightarrow g l \) F

unfolding tendsto-at-iff-tendsto-nhds by (rule filterlim-compose[of g])

lemma LIM-o: \( [g \dashv l \dashv \rightarrow g l; f \dashv a \dashv \rightarrow l] = (g \circ f) \dashv a \dashv \rightarrow g l \)
unfolding o-def by (rule tendsto-compose)

lemma tendsto-compose-eventually:
\( g \dashv l \dashv \rightarrow m \Longrightarrow (f \dashv \rightarrow l) F \Longrightarrow eventually (\lambda x \cdot f x \neq l) F \Longrightarrow (\lambda x \cdot g (f x)) \dashv \rightarrow m \) F
by (rule filterlim-compose[of g - at l]) (auto simp add: filterlim-at)

lemma LIM-compose-eventually:
assumes \( f: f \dashv a \dashv \rightarrow b \)
assumes \( g: g \dashv b \dashv \rightarrow c \)
assumes inj: eventually \( (\lambda x \cdot f x \neq b) \) (at a)
shows \( (\lambda x \cdot g (f x)) \dashv a \dashv \rightarrow c \)
using \( g \circ f \) inj by (rule tendsto-compose-eventually)

lemma tendsto-compose-filtermap: \( ((g \circ f) \dashv \rightarrow T) F \iff (g \dashv \rightarrow T) \) (filtermap \( f F \))
by (simp add: filterlim-def filtermap-filtermap comp-def)

98.11.1 Relation of LIM and LIMSEQ

lemma (in first-countable-topology) sequentially-imp-eventually-within:
\( (\forall f. (\forall n. f \in s \land f n \neq a) \land f \dashv \rightarrow a \Longrightarrow eventually (\lambda n. P (f n)) \)
lemma (in first-countable-topology) sequentially-imp-eventually-at:
$(\forall f. (\forall n. f \neq a) \land f \longrightarrow a \longrightarrow (\lambda n. P (f n))) \quad\Longrightarrow\quad (\forall n. P (a))$

using assms sequentially-imp-eventually-within [where $s=\text{UNIV}$] by simp

lemma LIMSEQ-SEQ-conv1:
fixes $f :: \text{'a::topological-space \Rightarrow 'b::topological-space}$
assumes $f : f \quad\longrightarrow\quad a \quad\longrightarrow\quad l$
shows $(\\forall S. (\forall n. S \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. f (S n))) \longrightarrow l$
using tendsto-compose-eventually [OF $f$, where $F=\text{sequentially}$] by simp

lemma LIMSEQ-SEQ-conv2:
fixes $f :: \text{'a::first-countable-topology \Rightarrow 'b::topological-space}$
assumes $f : f \quad\longrightarrow\quad a \quad\longrightarrow\quad l$
shows $(\\forall S. (\forall n. S \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. f (S n))) \longrightarrow l$
using LIMSEQ-SEQ-conv2 LIMSEQ-SEQ-conv1..

lemma sequentially-imp-eventually-at-left:
fixes $a :: \text{'a::dense-linorder \land \text{linorder-topology \land first-countable-topology}}$
assumes $b[simp]: b < a$
assumes $*: (\\forall n. b < f n) \implies (\\forall n. f n < a) \implies \text{incseq} f \quad\longrightarrow\quad a$
shows $(\\forall n. P (f n))$ sequentially

proof (safe intro!: sequentially-imp-eventually-within)
fix $X$ assume $X : (\forall n. X \in \{..<a\} \land X \neq a \land X \longrightarrow a$
show $(\\forall n. P (X n))$ sequentially

proof (rule contr)
  assume neg: $\neg$ eventually $(\lambda n. P (X n))$ sequentially
  have $(\exists s. (\forall n. (\neg P (X (s n)) \land b < X (s n)) \land (X (s n) \leq X (s \text{Suc n})) \land $ $\text{Suc (s n)} \leq s (\text{Suc n}))$

proof (rule dependent-nat-choice)
  have $\neg$ eventually $(\lambda n. b < X n)$ sequentially
  by (intro not-eventually-implnegorder-tendstoD1) [OF $X (2)$ $b$]
  then show $(\exists x. \neg P (X x) \land b < X x$
  by (auto dest!: not-eventuallyD)
next
fix $x n$
have $\neg$ eventually $(\lambda n. \text{Suc x} \leq n \longrightarrow b < X n \longrightarrow X x < X n \longrightarrow P (X n)$
n)) sequentially
using X by (intro not-eventually-impI order-tendstoD(1)[OF X(2)] eventually-ge-at-top
neg) auto
  then show \( \exists n. (\neg P \ (X \ n) \land b < X \ n) \land (X \ x \leq X \ n \land Suc \ x \leq n) \)
    by (auto dest!: not-eventuallyD)
  qed
then guess s ..
then have \( \bigwedge n. \ b < X \ (s \ n) \land X \ (s \ n) < a \ \text{incseq} \ \ (\lambda n. \ X \ (s \ n)) \ (\lambda n. \ X \ (s n)) \quad \longrightarrow \quad a \ \neg P \ (X \ (s \ n)) \)
  using X by (auto simp: subseq-Suc-iff Suc-le-eq incseq-Suc-iff intro: LIMSEQ-subseq-LIMSEQ[OF
  \ (X \ ---- \ a; \ unfolded \ comp-def)\])
from *[OF this(1,2,3,4)] this(5) show False by auto
qed

lemma tendsto-at-left-sequentially:
fixes a :: :: \{dense-linorder, linorder-topology, first-countable-topology\}
assumes b < a
assumes S: \( \forall \ n. \ S \ n < a \implies \bigwedge \ n. \ S \ n < a \implies \text{incseq} \ \ (\lambda n. \ X \ (S \ n)) \ ---- \ L \)
shows (X ---- L) (at-left a)
using assms unfolding tendsto_def [where l=L]
by (simp add: sequentially-imp-eventually-at-left)

lemma sequentially-imp-eventually-at-right:
fixes a :: :: \{dense-linorder, linorder-topology, first-countable-topology\}
assumes b[simp]: a < b
assumes f: \( \forall \ n. \ a < f \ n \implies \bigwedge \ n. \ f \ n < b \implies \text{decseq} \ f \implies f \ ---- \ a \implies \text{eventually} \ \ (\lambda n. \ P \ (f \ n)) \ \text{sequentially} \)
shows eventually P (at-right a)
proof (safe intro!: sequentially-imp-eventually-within)
  fix X assume X: \( \forall \ n. \ X \ n \in \{a <..\} \land X \ n \neq a \quad X \ ---- \ a \)
  show eventually (\lambda n. \ P \ (X \ n)) sequentially
  proof (rule econtr)
    assume neg: \( \neg \ \text{eventually} \ \ (\lambda n. \ P \ (X \ n)) \ \text{sequentially} \)
    have 3.s. \( \forall \ n. \ \neg (P \ (X \ (s \ n)) \land X \ (s \ n) < b) \land (X \ (s \ (Suc \ n)) \leq X \ (s \ n) \land Suc \ (s \ n) \leq s \ (Suc \ n)) \)
    proof (rule dependent-nat-choice)
      have \( \neg \ \text{eventually} \ \ (\lambda n. \ X \ n < b \implies P \ (X \ n)) \ \text{sequentially} \)
        by (intro not-eventually-impI order-tendstoD(2) [OF X(2) b])
      then show \( \exists x. \neg P \ (X \ x) \land X \ x < b \)
        by (auto dest!: not-eventuallyD)
    next
      fix x n
      have \( \neg \text{eventually} \ (\lambda n. \ Suc \ x \leq n \implies X \ x < b \implies X \ n < X \ x \implies P \ (X \ n)) \)
        sequentially
        using X by (intro not-eventually-impI order-tendstoD(2)[OF X(2)] eventually-ge-at-top
        neg) auto
      then show \( \exists n. \ (\neg P \ (X \ n) \land X \ n < b) \land (X \ n \leq X \ x \land Suc \ x \leq n) \)
THEORY "Topological-Spaces"

by (auto dest!: not-eventuallyD)
qed

then have \( \bigwedge n. a < X (s n) \land n. X (s n) < b \) decseq \((\land n. X (s n)) \land n. X (s n)) \rightarrow a \land n. \neg P (X (s n)) \)

using \( X \) by (auto simp: subseq-Suc-iff Suc-le-eq decseq-Suc-iff intro!: LIMSEQ-subseq-LIMSEQ[OF \( \langle X \rightarrow a \rangle \))

then guess \( s \) using \( X \) by (auto)

then have \( \forall \) \( B \) \( \exists \) \( x \) \( f \) \( x \in x \in f \) \( \exists A. open A \land x \in A \land (\forall y \in s. y \in A \rightleftharpoons f y \in B)) \)

unfolding continuous-on-def tendsto-def eventually-at-topological by metis

lemma continuous-on-open-invariant:
continuous-on s f \( \leftarrow \rightarrow (\forall B. open B \rightarrow (\exists A. open A \land A \cap s = f \cap B \cap s)) \)

proof safe
fix \( B \) :: \( ' \) set assume continuous-on s f open B
then have \( \forall x \in f \cap B \cap s. (\exists A. open A \land x \in A \land s \cap A \subseteq f \cap B \cap s)) \)

by (auto simp: continuous-on-topological subset-eq Ball-def imp-conjL)

then obtain A where \( \forall x \in f \cap B \cap s. (A x) \land x \in A x \land s \cap A x \subseteq f \cap B \cap s \)

unfolding bchoice-iff ..
then show \( \exists A. open A \land A \cap s = f \cap B \cap s \)

by (intro exI[of - \bigcup x \in f \cap B \cap s. A x]) auto

98.12 Continuity

98.12.1 Continuity on a set

definition continuous-on :: 'a set \( \Rightarrow \) ('a :: topological-space) \( \Rightarrow \) bool where
continuous-on s f \( \leftarrow \rightarrow (\forall x \in s. (f \rightarrow f x) \land (x \in x \in s)) \)

lemma continuous-on-cong cong:
\( s = t \rightarrow (\bigwedge x. x \in t \rightarrow f x = g x) \rightarrow continuous-on s f \leftarrow continuous-on t g \)

unfolding continuous-on-topological by (intro ball-cong filterlim-cong) (auto simp: eventually-at-filter)

lemma continuous-on-topological:
continuous-on s f \( \leftarrow \rightarrow (\forall x \in s. (\forall B. open B \rightarrow f x \in B \rightarrow (\exists A. open A \land x \in A \land (\forall y \in s. y \in A \rightarrow f y \in B)))) \)

unfolding continuous-on-def tendsto-def eventually-at-topological by metis

lemma continuous-on-open-invariant:
continuous-on s f \( \leftarrow \rightarrow (\forall B. open B \rightarrow (\exists A. open A \land A \cap s = f \cap B \cap s)) \)

proof safe
fix B :: \( ' \) set assume continuous-on s f open B
then have \( \forall x \in f \cap B \cap s. (\exists A. open A \land x \in A \land s \cap A \subseteq f \cap B \cap s)) \)

by (auto simp: continuous-on-topological subset-eq Ball-def imp-conjL)

then obtain A where \( \forall x \in f \cap B \cap s. (A x) \land x \in A x \land s \cap A x \subseteq f \cap B \cap s \)

unfolding bchoice-iff ..
then show \( \exists A. open A \land A \cap s = f \cap B \cap s \)

by (intro exI[of - \bigcup x \in f \cap B \cap s. A x]) auto
next
  assume \( B : \forall B. \text{open } B \longrightarrow (\exists A. \text{open } A \land A \cap s = f \setminus B \cap s) \)
  show continuous-on s f
    unfolding continuous-on-topological
    proof safe
      fix x B assume x \( \in \) s open B \( f x \in \) B
      with B obtain A where A: open A \( A \cap s = f \setminus B \cap s \) by auto
      with \( \langle x \in s \rangle \langle f x \in B \rangle \) show \( \exists A. \text{open } A \land x \in A \land (\forall y \in s. y \in A \longrightarrow f y \in B) \)
        by (intro \( \text{exI[of - A]} \)) auto
      qed
    qed

lemma continuous-on-open-vimage:
  open s \( \iff \) continuous-on s f \( \iff \) (\( \forall B. \text{open } B \longrightarrow \text{open } (f \setminus B \cap s) \))
  unfolding continuous-on-open-invariant
  by (metis open-Int Int-absorb Int-commute[of s] Int-assoc[of - - s])

corollary continuous-imp-open-vimage:
  assumes continuous-on s f open s open B \( f \setminus B \subseteq s \)
  shows open \( f \setminus B \)
  by (metis assms continuous-on-open-vimage le-iff-inf)

corollary open-vimage[continuous-intros]:
  assumes open s and continuous-on UNIV f
  shows open \( f \setminus s \)
  using assms unfolding continuous-on-open-vimage [OF open-UNIV]
  by simp

lemma continuous-on-closed-invariant:
  continuous-on s f \( \iff \) (\( \forall B. \text{closed } B \longrightarrow (\exists A. \text{closed } A \land A \cap s = f \setminus B \cap s) \))
  proof –
    have \( * \): \( \forall P : b \text{ set} \Rightarrow \text{bool}. (\forall A. P A \iff Q (\neg A)) \Rightarrow (\forall A. P A) \iff (\forall A. Q A) \)
      by (metis double-compl)
    show ?thesis
      unfolding continuous-on-open-invariant by (intro \( * \)) (auto simp: open-closed[symmetric])
    qed

lemma continuous-on-closed-vimage:
  closed s \( \iff \) continuous-on s f \( \iff \) (\( \forall B. \text{closed } B \longrightarrow \text{closed } (f \setminus B \cap s) \))
  unfolding continuous-on-closed-invariant
  by (metis closed-Int Int-absorb Int-commute[of s] Int-assoc[of - - s])

corollary closed-vimage[continuous-intros]:
  assumes closed s and continuous-on UNIV f
  shows closed \( f \setminus s \)
  using assms unfolding continuous-on-closed-vimage [OF closed-UNIV]
by simp

lemma continuous-on-open-Union:
(∀s. s ∈ Simplies open s)implies (∀s. s ∈ Simplies continuous-on s f)implies continuous-on (∪S) f
unfolding continuous-on-def by safe (metis open-Union at-within-open UnionI)

lemma continuous-on-open-UN:
(∀s. s ∈ Simplies open (A s))implies (∀s. s ∈ Simplies continuous-on (A s) f)implies continuous-on (∪s∈S. A s) f
unfolding Union-image-eq[symmetric] by (rule continuous-on-open-Union) auto

lemma continuous-on-closed-Un:
closed simplies closed timplies continuous-on s fimplies continuous-on t fimplies continuous-on (s ∪ t) f
by (auto simp add: continuous-on-closed-vimage closed-Un Int-Un-distrib)

lemma continuous-on-If:
assumes closed: closed s closed t and cont: continuous-on s f continuous-on t g and P: ∀x. x ∈ simplies ¬ P ximplies f x = g x ∀x. x ∈ timplies P ximplies f x = g x
shows continuous-on (s ∪ t) (λx. if P x then f x else g x) (is continuous-on - ?h)
proof –
  from P have ∀x∈s. f x = ?h x ∀x∈t. g x = ?h x
    by auto
  with cont have continuous-on s ?h continuous-on t ?h
    by simp-all
  with closed show ?thesis
    by (rule continuous-on-closed-Un)
qed

lemma continuous-on-id[continuous-intros]: continuous-on s (λx. x)
unfolding continuous-on-def by (fast intro: tendsto-ident-at)

lemma continuous-on-const[continuous-intros]: continuous-on s (λx. c)
unfolding continuous-on-def by (auto intro: tendsto-const)

lemma continuous-on-compose[continuous-intros]:
continuous-on s fimplies continuous-on (f' s) gimplies continuous-on s (g o f)
unfolding continuous-on-topological by simp metis

lemma continuous-on-compose2:
continuous-on t gimplies continuous-on s fimplies t = f' simplies continuous-on s (λx. g (f x))
using continuous-on-compose[of s f g] by (simp add: comp-def)
98.12.2 Continuity at a point

definition continuous :: 'a::t2-space filter ⇒ ('a ⇒ 'b::topological-space) ⇒ bool
where
  continuous F f ≡ (f ---> f (Lim F (λx. x))) F

lemma continuous-bot[continuous-intros, simp]: continuous bot f
  unfolding continuous-def by auto

lemma continuous-trivial-limit: trivial-limit net ⇒ continuous net f
  by simp

lemma continuous-within: continuous (at x within s) f ---> (f ---> f x) (at x within s)
  by (cases trivial-limit (at x within s)) (auto simp add: Lim-ident-at continuous-def)

lemma continuous-within-topological:
  continuous (at x within s) f ---> (∀ B. open B ---> f x ∈ B ---> (∃ A. open A ∧ x ∈ A ∧ (∀ y ∈ s. y ∈ A ---> f y ∈ B)))
  unfolding continuous-within tendsto-def eventually-at-topological by metis

lemma continuous-within-compose[continuous-intros]:
  continuous (at x within s) f ---/> continuous (at (f x) within f ' s) g ---/> continuous (at x within s) (g o f)
  by (simp add: continuous-within-topological) metis

lemma continuous-within-compose2:
  continuous (at x within s) f ---/> continuous (at (f x) within f ' s) g ---/> continuous (at x within s) (λx. g (f x))
  using continuous-within-compose[of x s f g] by (simp add: comp-def)

lemma continuous-at: continuous (at x) f ---/> f -- x ---/> f x
  using continuous-within[of x UNIV f] by simp

lemma continuous-ident[continuous-intros, simp]: continuous (at x within S) (λx. x)
  unfolding continuous-within by (rule tendsto-ident-at)

lemma continuous-const[continuous-intros, simp]: continuous F (λx. c)
  unfolding continuous-def by (rule tendsto-const)

lemma continuous-on-eq-continuous-within:
  continuous-on s f ---> (∀ x ∈ s. continuous (at x within s) f)
  unfolding continuous-on-def continuous-within ..

abbreviation isCont :: ('a::t2-space ⇒ 'b::topological-space) ⇒ 'a ⇒ bool
where
  isCont f a ≡ continuous (at a) f

lemma isCont-def: isCont f a ---> f -- a ---/> f a
by (rule continuous-at)

lemma continuous-at-within: isCont f x \implies \text{continuous} (at x within s) f
  by (auto intro: tendsto-mono at-le simp: continuous-at continuous-within)

lemma continuous-on-eq-continuous-at: open s \implies \text{continuous-on s} f \iff (\forall x \in s. \text{isCont f x})
  by (simp add: continuous-on-def continuous-at within-open[of - s])

lemma isContI-continuous: continuous (at x within UNIV) f \implies isCont f x
  by simp

lemma isCont-ident: isCont (\lambda x. x) a
  using continuous-ident by (rule isContI-continuous)

lemmas isCont-const = continuous-const

lemma isCont-o2: isCont f a \implies isCont g (f a) \implies isCont (\lambda x. g (f x)) a
  unfolding isCont-def by (rule tendsto-compose)

lemma isCont-o[continuous-intros]: isCont f a \implies isCont g (f a) \implies isCont (g \circ f) a
  unfolding o-def by (rule isCont-o2)

lemma isCont-tendsto-compose: isCont g l \implies (f \dashv\dashv l) F \implies ((\lambda x. g (f x)) \dashv\dashv g l) F
  unfolding isCont-def by (rule tendsto-compose)

lemma continuous-within-compose3:
  isCont g (f x) \implies \text{continuous} (at x within s) f \implies \text{continuous} (at x within s)
  \(\lambda x. g (f x)\)
  using continuous-within-compose2[of x s f g] by (simp add: continuous-at-within)

lemma filtermap-nhds-open-map:
  assumes cont: isCont f a and open-map: \(\forall S. \text{open } S \implies \text{open } (f S)\)
  shows filtermap f (nhds a) = nhds (f a)
  unfolding filter-eq-iff
  proof (safe)
    fix P assume eventually P (filtermap f (nhds a))
    then guess \(S\) unfolding eventually-filtermap eventually-nhds ..
    then show eventually P (nhds (f a))
      unfolding eventually-nhds by (intro exI[of - f S]) (auto intro!: open-map)
  qed (metis filterlim-iff tendsto-at-iff-tendsto-nhds isCont-def eventually-filtermap)
lemma continuous-at-split:
  continuous (at (x::'a::linorder-topology)) f = (continuous (at-left x) f ∧ continuous (at-right x) f)
  by (simp add: continuous-within filterlim-at-split)

98.12.3 Open-cover compactness

context topological-space
begin

definition compact :: 'a set ⇒ bool where
  compact-eq-heine-borel: — This name is used for backwards compatibility
  compact S ←→ (∀C. (∀c∈C. open c) ∧ S ⊆ ∪C → (∃D⊆C. finite D ∧ S ⊆ ∪D))

lemma compactI:
  assumes (∀C. ∀t∈C. open t ⇒ s ⊆ ∪C =⇒ ∃C'. C' ⊆ C ∧ finite C' ∧ s ⊆ ∪C')
  shows compact s
  unfolding compact-eq-heine-borel using assms by metis

lemma compact-empty[simp]: compact {}
  by (auto intro!: compactI)

lemma compactE:
  assumes compact s and ∀t∈C. open (f t) and s ⊆ (⋃c∈C. f c)
  obtains C' where C' ⊆ C and finite C' and s ⊆ (⋃c∈C'. f c)
  using assms unfolding compact-eq-heine-borel by metis

lemma compactE-image:
  assumes compact s and ∀t∈C. open f (C ∪ {−t}) and s ⊆ (⋃c∈C. f c)
  obtains C' where C' ⊆ C and finite C' and s ⊆ (⋃c∈C'. f c)
  using assms unfolding ball-simps[symmetric] SUP-def
  by (metis (lifting) finite-subset-image compact-eq-heine-borel[of s])

lemma compact-inter-closed [intro]:
  assumes compact s and closed t
  shows compact (s ∩ t)
  proof (rule compactI)
    fix C assume C: ∀c∈C. open c and cover: s ∩ t ⊆ ∪C
    from C (closed t) have ∀c∈C ∪ {−t}. open c by auto
    moreover from cover have s ⊆ (⋃c∈C ∪ {−t}) by auto
    ultimately have ∃D⊆C ∪ {−t}. finite D ∧ s ⊆ ∪D
      using (compact s) unfolding compact-eq-heine-borel by auto
    then obtain D where D ⊆ C ∪ {−t} ∧ finite D ∧ s ⊆ ∪D ..
    then show ∃D⊆C. finite D ∧ s ∩ t ⊆ ∪D
      by (intro ext[of - D ∩ {−t}]) auto
THEORY "Topological-Spaces"

qed

lemma inj-setminus: inj-on uminus (A::'a set set)
  by (auto simp: inj-on-def)

lemma compact-fip:
  compact U a set/set
  (\forall A. (\forall a \in A. closed a) \rightarrow (\forall B \subseteq A. finite B \rightarrow U \cap \bigcap B \neq \{\})) \rightarrow U \neq \{\}
  (is - \rightarrow ?R)
  proof (safe intro!: compact-eq-heine-borel[THEN iffD2])
    fix A
    assume compact U
    then have (\forall a \in uminus'A. open a) \land U \subseteq \bigcup (uminus'A)
      by auto
    with \{compact U\} obtain B where B \subseteq A finite (uminus'B) \neq \{\}
      unfolding compact-eq-heine-borel by (metis subset-image-iff)
    then show False
      by (auto dest: finite-imageD intro: inj-setminus)
  next
    fix A
    assume ?R
    then have (\forall a \in A. open a \subseteq U \cap \bigcup A = \{\})
      by auto
    with ?R obtain B where B \subseteq A finite (uminus'B) \subseteq \bigcup (uminus'B)
      unfolding compact-eq-heine-borel by (metis subset-image-iff)
    then show \exists T \subseteq A. finite T \land U \subseteq \bigcup T
      by (auto intro!: exI[of - B] inj-setminus dest: finite-imageD)
  qed

lemma compact-imp-fip-image:
  assumes compact s
    and P: \A i. i \in I \rightarrow closed (f i)
    and Q: \A I'. finite I' \rightarrow I' \subseteq I \rightarrow (s \cap (\bigcap i \in I'. f i) \neq \{\})
  shows s \cap (\bigcap i \in I. f i) \neq \{\}
  proof
    note (compact s)
    moreover from P have \A i \in f \cdot I. closed i by blast
    moreover have \A. finite A \land A \subseteq f \cdot I \rightarrow (s \cap (\bigcap A) \neq \{\})
    proof (rule, rule, erule conjE)
fix $A :: 'a set$
assume finite $A$
moreover assume $A \subseteq f \cdot I$
ultimately obtain $B$ where $B \subseteq I$ and finite $B$ and $A = f \cdot B$
using finite-subset-image [of $A f I$] by blast
with $Q$ [of $B$] show $s \cap \bigcap A \neq \emptyset$ by simp
qed
ultimately have $s \cap (\bigcap (f \cdot I)) \neq \emptyset$ by (rule compact-imp-fip)
then show $\text{thesis}$ by simp
qed

end

lemma (in t2-space) compact-imp-closed:
assumes compact $s$
shows closed $s$
unfolding closed-def
proof (rule openI)
fix $y$
assume $y \in - s$
let $?C = \bigcup x \in s. \{ u. open u \land x \in u \land eventually (\lambda y. y \notin u) (nhds y) \}$
note (compact $s$)
moreover have $\forall u \in ?C. open u$ by simp
moreover have $s \subseteq \bigcup ?C$
proof
fix $x$
assume $x \in s$
with $y \in - s$ have $x \neq y$ by clarsimp
hence $\exists u. v. open u \land open v \land x \in u \land y \in v \land u \cap v = \{ \}$
by (rule hausdorff)
with $x \in s$ show $x \in \bigcup ?C$
unfolding eventually-nhds by auto
qed
ultimately obtain $D$ where $D \subseteq ?C$ and finite $D$ and $s \subseteq \bigcup D$
by (rule compactE)
from $D \subseteq ?C$ have $\forall x \in D. eventually (\lambda y. y \notin x) (nhds y)$ by auto
with $\{finite D\}$ have $\forall u. v. open u \land open v \land u \cap v = \{ \}$
by (simp add: eventually-Ball-finite)
with $\{s \subseteq \bigcup D\}$ have $\forall u. v. open u \land open v \land u \cap v = \{ \}$
by (auto elim!: eventually-mono [rotated])
thus $\exists t. open t \land y \in t \land t \subseteq - s$
by (simp add: eventually-nhds subset-eq)
qed

lemma compact-continuous-image:
assumes $f$: continuous-on $s$ and $s$ compact $s$
shows compact $(f \cdot s)$
proof (rule compactI)
fix $C$ assume $\forall c \in C. open c$ and cover: $f's \subseteq \bigcup C$
with $f$ have $\forall c \in C. \exists A. open A \land A \cap s = f - c \cap s$
unfolding continuous-on-open-invariant by blast
then obtain $A$ where $A: \forall c \in C. open (A c) \land A c \cap s = f - c \cap s$
unfolding bchoice_iff ..

with cover have ∀c∈C. open (A c) s ⊆ (⋃c∈C. A c) by (fastforce simp add: subset-eq set-eq-iff)+

from compactE-image[OF s this] obtain D where D ⊆ C finite D s ⊆ (⋃c∈D. A c) .

with A show ∃D ⊆ C. finite D ∧ f's ⊆ ⋃D

by (intro exI[of - D]) (fastforce simp add: subset-eq set-eq-iff)+

qed

lemma continuous-on-inv:

fixes f :: ′a::topological-space ⇒ ′b::t2-space

assumes continuous-on s f compact s ∀x∈s. g (f x) = x

shows continuous-on (f ' s) g

unfolding continuous-on-topological

proof (clarsimp simp add: assms(3))

fix x :: ′a and B :: ′a set

assume x ∈ s and open B and x ∈ B

have 1: ∀x∈s. f x ∈ f ' (s - B) ⟷ x ∈ s - B

using assms(3) by (auto, metis)

have continuous-on (s - B) f

using (continuous-on-on s f) Diff-subset

by (rule continuous-on-on-subset)

moreover have compact (s - B)

using (open B) and (compact s)

unfolding Diff-eq by (intro compact-inter-closed closed-Compl)

ultimately have compact (f ' (s - B))

by (rule compact-continuous-image)

hence closed (f ' (s - B))

by (rule compact-imp-closed)

hence open (− f ' (s - B))

by (rule open-Compl)

moreover have f x ∈ − f ' (s - B)

using (x ∈ s) and (x ∈ B); by (simp add: 1)

moreover have ∀y∈s. f y ∈ − f ' (s - B) ⟷ y ∈ B

by (simp add: 1)

ultimately show ∃A. open A ∧ f x ∈ A ∧ (∀y∈s. f y ∈ A ⟷ y ∈ B)

by fast

qed

lemma continuous-on-inv-into:

fixes f :: ′a::topological-space ⇒ ′b::t2-space

assumes s: continuous-on s f compact s and f: inj-on f s

shows continuous-on (f ' s) (the-inv-into s f)

by (rule continuous-on-inv[OF s]) (auto simp: the-inv-into-f f[OF f])

lemma (in linorder-topology) compact-attains-sup:

assumes compact S S ≠ {} 

shows ∃s∈S. ∀t∈S. t ≤ s

proof (rule classical)
proof

lemma (in linorder-topology) compact-attains-inf:
  assumes compact S \( S \neq \{\} \)
  shows \( \exists s \in S. \forall t \in S. s \leq t \)
proof (rule classical)
  assume \( \neg (\exists s \in S. \forall t \in S. s \leq t) \)
  then obtain \( t \) where \( t: \forall s \in S. \forall t \in S \) and \( \forall s \in S. s < t \)
    by (metis not-le)
  then have \( \forall s \in S. \) open \( \{..< t s\} \) \( S \subseteq (\bigcup s \in S. \{..< t s\}) \)
    by auto
  with (compact S) obtain \( C \) where \( C \subseteq S \) finite \( C \) and \( C: S \subseteq (\bigcup s \in C. \{..< t s\}) \)
    by (erule compactE-image)
  with \( S \neq \{\} \) have Max: Max \( (t'C) \in t'C \) and \( \forall s \in t'C. s \leq Max \ (t'C) \)
    by (auto intro!: Max-in)
  with \( C \) have \( S \subseteq \{..< Max \ (t'C)\} \)
    by (auto intro: less-le-trans simp: subset-eq)
  with \( t \) Max \( (C \subseteq S) \) show \( \exists \)thesis
    by fastforce
qed

lemma (in linorder-topology) compact-attains-sup:
  fixes f :: 'a::topological-space \Rightarrow 'b::linorder-topology
  shows compact s \( \Rightarrow s \neq \{\} \) \( \Rightarrow \) continuous-on s f \( \Rightarrow (\exists x \in s. \forall y \in s. f y \leq f x) \)
  using compact-attains-sup[of f ' s] compact-continuous-image[of s f] by auto

lemma compact-attains-inf:
  fixes f :: 'a::topological-space \Rightarrow 'b::linorder-topology
  shows compact s \( \Rightarrow s \neq \{\} \) \( \Rightarrow \) continuous-on s f \( \Rightarrow (\exists x \in s. \forall y \in s. f x \leq f y) \)
  using compact-attains-inf[of f ' s] compact-continuous-image[of s f] by auto
THEORY "Topological-Spaces"

98.13 Connectedness

context topological-space

begin

definition connected S ←→ ¬ (∃ A B. open A ∩ open B ∧ S ⊆ A ∪ B ∧ A ∩ B ∩ S = {} ∧ A ∩ S ≠ {} ∧ B ∩ S ≠ {}})

lemma connectedI:
(⋀ A B. open A = ⇒ open B = ⇒ A ∩ U ≠ {} = ⇒ B ∩ U ≠ {} = ⇒ A ∩ B ∩ U = {} = ⇒ connected U
by (auto simp: connected-def)

lemma connected-empty[simp]: connected {}
by (auto intro!: connectedI)

lemma connectedD:
connected A = ⇒ open U = ⇒ open V = ⇒ U ∩ V ∩ A = {} = ⇒ A ⊆ U ∪ V
⇒ U ∩ A = {} = ⇒ V ∩ A = {}
by (auto simp: connected-def)

end

lemma connected-local-const:
assumes connected A a ∈ A b ∈ A
assumes *: ∀ a∈A. eventually (λb a = f b) (at a within A)
shows f a = f b
proof −
obtain S where S: ⋀a. a ∈ A =⇒ a ∈ S a ⋀a. a ∈ A =⇒ open (S a)
⋀a x. a ∈ A =⇒ x ∈ S a =⇒ x ∈ A =⇒ f a = f x
using * unfolding eventually-at-topological by metis

let ?P = ⋃b∈{b∈A. f a = f b}. S b and ?N = ⋃b∈{b∈A. f a ≠ f b}. S b
have ?P ∩ A = {} = ?N ∩ A = {}
using ‹connected A› S ‹a∈A›
by (intro connectedD) (auto, metis)
then show f a = f b
proof
assume ?N ∩ A = {}
then have ‹∀ x∈A. f a = f x›
using S(1) by auto
with ‹b∈A› show ?thesis by auto
next
assume ?P ∩ A = {}
then show ?thesis
using ‹a ∈ A› S(1)[of a] by auto
qed

qed
lemma (in linorder-topology) connectedD-interval:
assumes connected U and xy: x ∈ U y ∈ U and x ≤ z z ≤ y
shows z ∈ U
proof –
  have eq: {..<z} ∪ {z<..} = − {z}
    by auto
  { assume z /∈ U x < z z < y
    with xy have ¬ connected U
      unfolding connected-def simp-thms
      apply (rule-tac exI [of - {..<z}])
      apply (rule-tac exI [of - {z<..}])
      apply (auto simp add: eq)
      done }
  with assms show z ∈ U
    by (metis less-le)
qed

lemma connected-continuous-image:
assumes ∗: continuous-on s f
assumes connected s
shows connected (f ' s)
proof (rule connectedI)
  fix A B assume A: open A A ∩ f ' s ≠ {} and B: open B B ∩ f ' s ≠ {} and
  AB: A ∩ B ∩ f ' s = {} f ' s ⊆ A ∪ B
  obtain A' where A': open A' f − ' A ∩ s = A' ∩ s
    using ∗ ⟨open A'⟩ unfolding continuous-on-open-invariant by metis
  obtain B' where B': open B' f − ' B ∩ s = B' ∩ s
    using ∗ ⟨open B'⟩ unfolding continuous-on-open-invariant by metis
  have ∃ A B. open A ∧ open B ∧ s ⊆ A ∪ B ∧ A ∩ B ∩ s = {} ∧ A ∩ s ≠ {} ∧ B ∩ s ≠ {}
    proof (rule exI [of A'], rule exI [of B'], intro conjI)
      have s ⊆ (f − ' A ∩ s) ∪ (f − ' B ∩ s) using AB by auto
      then show s ⊆ A' ∪ B' using A' B' by auto
    next
      have (f − ' A ∩ s) ∩ (f − ' B ∩ s) = {} using AB by auto
      then show A' ∩ B' ∩ s = {} using A' B' by auto
    qed (insert A' B' A B, auto)
  with ⟨connected s⟩ show False
    unfolding connected-def by blast
qed

99 Connectedness

class linear-continuum-topology = linorder-topology + linear-continuum
begin

lemma Inf-notin-open:
  assumes A: open A and bmd: ∀ a ∈ A. x < a
shows $\inf A \notin A$

proof

assume $\inf A \in A$
then obtain $b$ where $b < \inf A \{b < \inf A\} \subseteq A$
using $\text{open-left}[\text{of} A \inf A x]$ assms by auto
with $\text{dense}[\text{of} b \inf A]$ obtain $c$ where $c < \inf A c \in A$
by (auto simp: subset-eq)
then show $\text{False}$
using $c\inf\text{-lower}[\text{OF} \langle c \in A \rangle] \text{bnd}$ by (metis $\text{less-imp-le bdd\text{-}\text{-belowI}$)
qed

lemma $\text{Sup\text{-notin\text{-open}}}$:

assumes $A$: $\text{open A}$ and $\text{bnd}$: $\forall a \in A. a < x$
shows $\text{Sup A} \notin A$

proof

assume $\text{Sup A} \in A$
then obtain $b$ where $\text{Sup A} < b \{\text{Sup A} .. < b\} \subseteq A$
using $\text{open-right}[\text{of} A \sup A x]$ assms by auto
with $\text{dense}[\text{of} \text{Sup A} b]$ obtain $c$ where $\text{Sup A} < c c \in A$
by (auto simp: subset-eq)
then show $\text{False}$
using $c\sup\text{-upper}[\text{OF} \langle c \in A \rangle] \text{bnd}$ by (metis $\text{less-imp-le not-\text{-le bdd\text{-aboveI}$)
qed

end

instance $\text{linear\text{-continuum\text{-topology}} \subseteq perfect\text{-space}$

proof

fix $x :: 'a$
obtain $y$ where $x < y \lor y < x$
using $\text{ex-gt-or-lt} \ [\text{of} x] ..$
with $\text{Inf\text{-notin\text{-open}}}[\text{of} \{x\} y] \text{Sup\text{-notin\text{-open}}}[\text{of} \{x\} y]$
show $\neg \text{open} \{x\}$
by auto
qed

lemma $\text{connectedI\text{-interval}}$:

fixes $U :: 'a :: \text{linear\text{-continuum\text{-topology set}$
assumes $*: \bigwedge x y z. x \in U \implies y \in U \implies x \leq z \implies z \leq y \implies z \in U$
shows $\text{connected} U$

proof (rule $\text{connectedI}$)

{ fix $A B$ assume $\text{open A open B} A \cap B \cap U = \{\} U \subseteq A \cup B$
fix $x y$ assume $x < y x \in A y \in B x \in U y \in U$

let $\?z = \inf (B \cap \{x <..\})$

have $x \leq \?z \?z \leq y$
using $(y \in B) \langle x < y \rangle$ by (auto intro: $\inf\text{-lower} \inf\text{-greatest}$)
with $(x \in U) \langle y \in U \rangle$ have $\?z \in U$}
by (rule *)
moreover have \( \exists z \notin B \cap \{x <..\} \)
using (open B, by (intro Inf-notin-open)) auto
ultimately have \( \exists z \in A \)
using \( x \leq \exists z \cap A \cap B \cap U = \{\} \) \( \forall x \in A \) \( U \subseteq A \cup B \) by auto

\{ assume \( \exists z < y \)
obtain a where \( \exists z < a \{a..\} \subseteq A \)
using open-right[OF (open A) \( \exists z \in A \) \( \exists z < y \)] by auto
moreover obtain b where \( b \in B x < b b < \min \ a \ y \)
using cInf-less-iff[of B \( \{x <..\} \) \( \min a \ y \) \( \exists z < a \) \( \exists z < y \) \( x < y \) \( y \in B \) by (auto intro: less-imp-le)
moreover have \( \exists z \leq b \)
using \( \{b \in B \} x < b \) by (intro cInf-lower) auto
moreover have \( b \in U \)
using \( x \leq \exists z \) \( \exists z \leq b \) \( \{b < \min \ a \ y \) by (intro *(OF \( \{x \in U \} \) \( \{y \in U \} \)) (auto simp: less-imp-le)
ultimately have \( \exists b \in B. b \in A \land b \in U \)
by (intro bex[of - b]) auto
then have \( \text{False} \)
using \( \{\exists z \leq y \} \) \( \{\exists z \in A \} \) \( y \in B \) \( y \in U \) \( A \cap B \cap U = \{\} \) unfolding le-less
by blast \}

\text{note not-disjoint = this}

fix \( A \ b \) assume \( AB: \text{open} \ A \text{ open} \ B \ U \subseteq A \cup B \ A \cap B \cap U = \{\} \)
moreover assume \( A \cap U \neq \{\} \)
then obtain \( x \) \text{ where} \( \forall x \in U x \in A \) by auto
moreover assume \( B \cap U \neq \{\} \)
then obtain \( y \) \text{ where} \( \forall y \in U y \in B \) by auto
moreover note not-disjoint[of B A y x] not-disjoint[of A B x y]
ultimately show \( \text{False} \) by (cases x y rule: linorder-cases) auto
qed

\text{lemma connected-iff-interval:}
\text{fixes} \ U :: 'a :: linear-continuum-topology set
\text{shows} connected U \( \iff \forall x \in U. \forall y \in U. \forall z. x \leq z \longrightarrow z \leq y \longrightarrow z \in U \)
by (auto intro: connectedI-interval dest: connectedD-interval)

\text{lemma connected-UNIV[simp]:} connected (UNIV::'a::linear-continuum-topology set)
\text{unfolding connected-iff-interval by auto}

\text{lemma connected-Ioi[simp]:} connected \{a::'a::linear-continuum-topology <..\}
\text{unfolding connected-iff-interval by auto}

\text{lemma connected-Ici[simp]:} connected \{a::'a::linear-continuum-topology ..\}
\text{unfolding connected-iff-interval by auto}

\text{lemma connected-Iio[simp]:} connected \{..< a::'a::linear-continuum-topology\}
\text{unfolding connected-iff-interval by auto}
lemma connected-Iic \[\text{simp} \]: connected \(\{a <.. b\}::\text{linear-continuum-topology}\)
  unfolding connected-iff-interval by auto

lemma connected-Ioo \[\text{simp} \]: connected \(\{a <..< b\}::\text{linear-continuum-topology}\)
  unfolding connected-iff-interval by auto

lemma connected-Ioc \[\text{simp} \]: connected \(\{a <.. b\}::\text{linear-continuum-topology}\)
  unfolding connected-iff-interval by auto

lemma connected-Ico \[\text{simp} \]: connected \(\{a ..< b\}::\text{linear-continuum-topology}\)
  unfolding connected-iff-interval by auto

lemma connected-Icc \[\text{simp} \]: connected \(\{a .. b\}::\text{linear-continuum-topology}\)
  unfolding connected-iff-interval by auto

lemma connected-contains-Ioo:
  fixes \(A::\text{linorder-topology \{set\}}\)
  assumes \(A::\text{connected \{A a \in A \ b \in A \ shows \{a <..< b\} \subseteq A}\)
  using connectedD-interval[OF A] by (simp add: subset_eq Ball_def less_imp_le)

99.1 Intermediate Value Theorem

lemma IVT':
  fixes \(f :: a::\text{linear-continuum-topology \Rightarrow} b::\text{linorder-topology}\)
  assumes \(y::f a \leq y \ y \leq f b \ a \leq b\)
  assumes \(*::\text{continuous-on \{a .. b\} f}\)
  shows \(\exists x. \ a \leq x \land x \leq b \land f x = y\)
  proof
    have \(\text{connected \{a..b\}}\)
      unfolding connected-iff-interval by auto
    from connected-continuous-image[OF *, this, THEN connectedD-interval, of f a f b y] y
    show \(?thesis\)
      by (auto simp add: atLeastAtMost_def atLeast_def atMost_def)
  qed

lemma IVT2':
  fixes \(f :: a::\text{linear-continuum-topology \Rightarrow} b::\text{linorder-topology}\)
  assumes \(y::f b \leq y \ y \leq f a \ a \leq b\)
  assumes \(*::\text{continuous-on \{a .. b\} f}\)
  shows \(\exists x. \ a \leq x \land x \leq b \land f x = y\)
  proof
    have \(\text{connected \{a..b\}}\)
      unfolding connected-iff-interval by auto
    from connected-continuous-image[OF *, this, THEN connectedD-interval, of f b f a y] y
    show \(?thesis\)
      by (auto simp add: atLeastAtMost_def atLeast_def atMost_def)
qed

lemma IVT:
  fixes \( f :: 'a :: \text{linear-continuum-topology} \Rightarrow 'b :: \text{linorder-topology} \)
  shows \( f a \leq y \Rightarrow y \leq f b \Rightarrow a \leq b \Rightarrow (\forall x. \ a \leq x \land x \leq b \rightarrow \text{isCont} \ f \ x) \)
  \( \Rightarrow \exists x. \ a \leq x \land x \leq b \land f x = y \)
  by (rule IVT') (auto intro: continuous-at-imp-continuous-on)

lemma IVT2:
  fixes \( f :: 'a :: \text{linear-continuum-topology} \Rightarrow 'b :: \text{linorder-topology} \)
  shows \( f b \leq y \Rightarrow y \leq f a \Rightarrow a \leq b \Rightarrow (\forall x. \ a \leq x \land x \leq b \rightarrow \text{isCont} \ f \ x) \)
  \( \Rightarrow \exists x. \ a \leq x \land x \leq b \land f x = y \)
  by (rule IVT2') (auto intro: continuous-at-imp-continuous-on)

lemma continuous-inj-imp-mono:
  fixes \( f :: 'a :: \text{linear-continuum-topology} \Rightarrow 'b :: \text{linorder-topology} \)
  assumes \( x : a < x < x \)
  assumes \( \text{cont} : \text{continuous-on} \{a..b\} \ f \)
  assumes \( \text{inj} : \text{inj-on} \ f \{a..b\} \)
  shows \( (f a < f x \land f x < f b) \lor (f b < f x \land f x < f a) \)
  proof
    note I = inj-on-iff[OF inj]
    { assume \( f x < f a \ f x < f b \)
      then obtain \( s t \) where \( x \leq s s \leq b a \leq t t \leq x f s = f t f x < f s \)
        using IVT'[of f x min (f a) (f b) b] IVT2'[of f x min (f a) (f b) a] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
      with \( x \) I have False by auto }
    moreover
    { assume \( f a < f x \ f b < f x \)
      then obtain \( s t \) where \( x \leq s s \leq b a \leq t t \leq x f s = f t f s < f x \)
        using IVT'[of f a max (f a) (f b) x] IVT2'[of f b max (f a) (f b) x] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
      with \( x \) I have False by auto }
    ultimately show \( \text{thesis} \)
      using I[of a x] I[of x b] x less-trans[OF \( \text{x} \) by (auto simp add: le-less less-imp-neq neq-iff)]
    qed

99.2 Setup 'a filter for lifting and transfer

context begin interpretation lifting-syntax.

definition rel-filter :: ('a ⇒ 'b ⇒ bool) ⇒ 'a filter ⇒ 'b filter ⇒ bool
  where rel-filter R F G = ((R === op =) === op =) (Rep-filter F) (Rep-filter G)

lemma rel-filter-eventually:
  rel-filter R F G \( \Leftarrow\)
  \( ((R === op =) === op =) \ (\lambda P. \text{eventually} \ P \ F) \ (\lambda P. \text{eventually} \ P \ G) \)
by (simp add: rel-filter-def eventually-def)

lemma filtermap-id [simp, id-simps]: filtermap id = id
by (simp add: fun-eq-iff id-def filtermap-ident)

lemma filtermap-id' [simp]: filtermap (λx. x) = (λF. F)
using filtermap-id unfolding id-def .

lemma Quotient-filter [quot-map]:
assumes Q: Quotient R Abs Rep T
shows Quotient (rel-filter R) (filtermap Abs) (filtermap Rep) (rel-filter T)
unfolding Quotient-alt-def
proof (intro conjI strip)
from Q have ∗: ∀x y. T x y ⇒ Abs x = y
  unfolding Quotient-alt-def by blast

fix F G
assume rel-filter T F G
thus filtermap Abs F = G unfolding filter-eq-iff
  by (auto simp add: eventually-filtermap rel-filter-eventually * rel-fun1 del: iffI elim!: rel-funD)
next
from Q have ∗: ∀x. T (Rep x) x unfolding Quotient-alt-def by blast

fix F
show rel-filter T (filtermap Rep F) F
  by (auto elim: rel-funD intro: * intro!: ext arg-cong | where f = λP. eventually P F | rel-fun1
del: iffI simp add: eventually-filtermap rel-filter-eventually)
qed (auto simp add: map-fun-def o-def eventually-filtermap filter-eq-iff fun-eq-iff rel-filter-eventually
  fun-quotient[OF fun-quotient[OF Q identity-quotient | identity-quotient, unfolded Quotient-alt-def])

lemma eventually-parametric [transfer-rule]:
  ((A ===> op =) ===> rel-filter A ===> op =) eventually eventually
by (simp add: rel-fun-def rel-filter-eventually)

lemma rel-filter-eq [relator-eq]: rel-filter op = = op =
by (auto simp add: rel-filter-eventually rel-fun-eq fun-eq-iff filter-eq-iff)

lemma rel-filter-mono [relator-mono]:
  A ≤ B ⇒ rel-filter A ≤ rel-filter B
unfolding rel-filter-eventually[abs-def]
by (rule le-fun1)+(intro fun-mono fun-mono THEN le-funD, THEN le-funD | order.refl)

lemma rel-filter-conversep [simp]: rel-filter A⁻¹⁻¹ = (rel-filter A)⁻¹⁻¹
by (auto simp add: rel-filter-eventually fun-eq-iff rel-fun-def)
lemma is-filter-parametric-aux:
  assumes is-filter F
  assumes [transfer-rule]: bi-total A bi-unique A
  and [transfer-rule]: \((A \Longrightarrow op =) \Longrightarrow \Longrightarrow op =) \) F G
  shows is-filter G
proof –
  interpret is-filter F by fact
  show ?thesis
  proof
    have F (\(\lambda x. \) True) = G (\(\lambda x. \) True) by transfer-prover
    thus G (\(\lambda x. \) True) by (simp add: True)
  next
    fix P' Q'
    assume G P' G Q'
    moreover
    from bi-total-fun[OF bi-unique A bi-total-eq, unfolded bi-total-def]
    obtain P Q where [transfer-rule]: \((A \Longrightarrow op =) \) P P' (A \Longrightarrow op =) Q
    by blast
    have F P = G P' F Q = G Q' by transfer-prover+
    ultimately have F (\(\lambda x. P \ x \land Q \ x) \) by (simp add: conj)
    moreover have F (\(\lambda x. P \ x \land Q \ x) = G (\(\lambda x. P' \ x \land Q' \ x) \) by transfer-prover
    ultimately show G (\(\lambda x. P' \ x \land Q' \ x) \) by simp
  next
    fix P' Q'
    assume \(\forall x. \) P' \(\longrightarrow\) Q' \(\longrightarrow\) G P'
    moreover
    from bi-total-fun[OF bi-unique A bi-total-eq, unfolded bi-total-def]
    obtain P Q where [transfer-rule]: \((A \Longrightarrow op =) \) P P' (A \Longrightarrow op =) Q
    by blast
    have F P = G P' by transfer-prover
    moreover have \(\forall x. \) P' \(\longrightarrow\) Q \(\longrightarrow\) \(\forall x. \) P' \(\longrightarrow\) Q' \(\longrightarrow\) by transfer-prover
    ultimately have F Q by (simp add: mono)
    moreover have F Q = G Q' by transfer-prover
    ultimately show G Q' by simp
  qed
qed

lemma is-filter-parametric [transfer-rule]:
  \[ \text{[bi-total A; bi-unique A]} \]
  \(\Longrightarrow\) \(\Longrightarrow\) \(\Longrightarrow\) \(\Longrightarrow\) is-filter is-filter
apply (rule rel-funI)
apply (rule iffI)
apply (erule is-filter-parametric-aux)
apply (erule is-filter-parametric-aux [where A=conversep A])
apply (auto simp add: rel-fun-def)
done

lemma left-total-rel-filter [transfer-rule]:
assumes [transfer-rule]: bi-total A bi-unique A
shows left-total (rel-filter A)
proof (rule left-totalI)
  fix F :: 'a filter
from bi-total-fun[OF bi-unique-fun[OF bi-total A bi-unique-eq] bi-total-eq]
obtain G where [transfer-rule]: ((A ===> op =) ===> op =) (λP. eventually P F) G
  unfolding bi-total-def by blast
moreover have is-filter (λP. eventually P F) <-> is-filter G by transfer-prover
hence is-filter G by(simp add: eventually-def is-filter-Rep-filter)
ultimately have rel-filter A F (Abs-filter G)
  by(simp add: rel-filter-eventually eventually-Abs-filter)
thus ∃ G. rel-filter A F G ..
qed

lemma right-total-rel-filter [transfer-rule]:
  [ bi-total A; bi-unique A ] ==> right-total (rel-filter A)
using left-total-rel-filter[of A^(-1)] by simp

lemma bi-total-rel-filter [transfer-rule]:
  assumes bi-total A bi-unique A
  shows bi-total (rel-filter A)
unfolding bi-total-alt-def using assms
by(simp add: left-total-rel-filter right-total-rel-filter)

lemma left-unique-rel-filter [transfer-rule]:
  assumes left-unique A
  shows left-unique (rel-filter A)
proof (rule left-uniqueI)
  fix F F' G
  assume [transfer-rule]: rel-filter A F G rel-filter A F' G
  show F = F'
    unfolding filter-eq-iff
proof
  fix P :: 'a ⇒ bool
  obtain P' where [transfer-rule]: (A ===> op =) P P'
    using left-total-fun[OF assms left-total-eq] unfolding left-total-def by blast
  have eventually P F = eventually P' G
    and eventually P F' = eventually P' G by transfer-prover+
  thus eventually P F = eventually P F' by simp
qed

lemma right-unique-rel-filter [transfer-rule]:
  right-unique A ==> right-unique (rel-filter A)
using left-unique-rel-filter[of A^(-1)] by simp

lemma bi-unique-rel-filter [transfer-rule]:
  bi-unique A ==> bi-unique (rel-filter A)
by(simp add: bi-unique-alt-def left-unique-rel-filter right-unique-rel-filter)

lemma top-filter-parametric [transfer-rule]:
  bi-total A \implies (rel-filter A) top top
by(simp add: rel-filter-eventually All-transfer)

lemma bot-filter-parametric [transfer-rule]: (rel-filter A) bot bot
by(simp add: rel-filter-eventually rel-fun-def)

lemma sup-filter-parametric [transfer-rule]:
  (rel-filter A) === sup sup
by(fastforce simp add: rel-filter-eventually rel-funD)

lemma Sup-filter-parametric [transfer-rule]:
  (rel-set (rel-filter A) === Sup Sup
proof(rule rel-funI)
  fix S T
  assume [transfer-rule]: rel-set (rel-filter A) S T
  show rel-filter A (Sup S) (Sup T)
    by(simp add: rel-filter-eventually eventually-Sup) transfer-prover
qed

lemma principal-parametric [transfer-rule]:
  (rel-set A === principal principal
proof(rule rel-funI)
  fix S S'
  assume [transfer-rule]: rel-set A S S'
  show rel-filter A (principal S) (principal S')
    by(simp add: rel-filter-eventually eventually-principal) transfer-prover
qed

context
  fixes A :: 'a => 'b => bool
  assumes [transfer-rule]: bi-unique A
begin

lemma le-filter-parametric [transfer-rule]:
  (rel-filter A === op =) op \leq op \leq
unfolding le-filter-def[abs-def] by transfer-prover

lemma less-filter-parametric [transfer-rule]:
  (rel-filter A === op =) op < op <
unfolding less-filter-def[abs-def] by transfer-prover

context
  assumes [transfer-rule]: bi-total A
begin

lemma Inf-filter-parametric [transfer-rule]:
(rel-set (rel-filter A) ===> rel-filter A) Inf Inf

unfolding Inf-filter-def[abs-def] by transfer-prover

lemma inf-filter-parametric (transfer-rule):
(rel-filter A ===> rel-filter A ===> rel-filter A) inf inf

proof (intro rel-funI)+
fix F F' G G'
assume [transfer-rule]: rel-filter A F F' rel-filter A G G'
have rel-filter A (Inf {F, G}) (Inf {F', G'}) by transfer-prover
thus rel-filter A (inf F G) (inf F' G') by simp

qed

end

end

end

end

100 Real-Vector-Spaces: Vector Spaces and Algebras over the Reals

theory Real-Vector-Spaces

imports Real Topological-Spaces

begin

100.1 Locale for additive functions

locale additive =
fixes f :: 'a::ab-group-add ⇒ 'b::ab-group-add
assumes add: f (x + y) = fx + fy

begin

lemma zero: f 0 = 0

proof –
have f 0 = f (0 + 0) by simp
also have ... = f 0 + f 0 by (rule add)
finally show f 0 = 0 by simp

qed

lemma minus: f (− x) = − f x

proof –
have f (− x) + f x = f (− x + x) by (rule add [symmetric])
also have ... = − f x + f x by (simp add: zero)
finally show f (− x) = − f x by (rule add-right-imp-eq)

qed
lemma diff: f (x - y) = f x - f y
  using add [of x - y] by (simp add: minus)

lemma setsum: f (setsum g A) = (∑ x∈A. f (g x))
apply (cases finite A)
apply (induct set: finite)
apply (simp add: zero)
apply (simp add: add)
apply (simp add: zero)
done

end

100.2 Vector spaces
locale vector-space =
  fixes scale :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b
  assumes scale-right-distrib [algebra-simps]:
    scale a (x + y) = scale a x + scale a y
and scale-left-distrib [algebra-simps]:
    scale (a + b) x = scale a x + scale b x
and scale-scale [simp]: scale a (scale b x) = scale (a * b) x
and scale-one [simp]: scale 1 x = x
begin

lemma scale-left-commute:
  scale a (scale b x) = scale b (scale a x)
by (simp add: mult.commute)

lemma scale-zero-left [simp]: scale 0 x = 0
and scale-minus-left [simp]: scale (− a) x = − (scale a x)
and scale-left-diff-distrib [algebra-simps]:
  scale (a − b) x = scale a x − scale b x
and scale-setsum-left: scale (setsum f A) x = (∑ a∈A. scale a (f a) x)
proof −
interpret s: additive λa. scale a x
proof qed (rule scale-left-distrib)
show scale 0 x = 0 by (rule s.zero)
show scale (− a) x = − (scale a x) by (rule s.minus)
show scale (a − b) x = scale a x − scale b x by (rule s.diff)
show scale (setsum f A) x = (∑ a∈A. scale (f a) x) by (rule s.setsum)
qed

lemma scale-zero-right [simp]: scale a 0 = 0
and scale-minus-right [simp]: scale a (− x) = − (scale a x)
and scale-right-diff-distrib [algebra-simps]:
  scale a (x − y) = scale a x − scale a y
and scale-setsum-right: scale a (setsum f A) = (∑ x∈A. scale a (f x))
proof −
THEORY “Real-Vector-Spaces”

interpret s: additive λx. scale a x
proof qed (rule scale-right-distrib)
show scale a 0 = 0 by (rule s.zero)
show scale a (− x) = − (scale a x) by (rule s.minus)
show scale a (x − y) = scale a x − scale a y by (rule s.diff)
show scale a (setsum f A) = (∑ x∈A. scale a (f x)) by (rule s.setsum)
qed

lemma scale-eq-0-iff [simp]:
scale a x = 0 ←→ a = 0 ∨ x = 0
proof cases
assume a = 0 thus ?thesis by simp
next
assume anz [simp]: a ≠ 0
{ assume scale a x = 0
  hence scale (inverse a) (scale a x) = 0 by simp
  hence x = 0 by simp }
thus ?thesis by force
qed

lemma scale-left-imp-eq:
[a ≠ 0; scale a x = scale a y] ⇒ x = y
proof —
assume nonzero: a ≠ 0
assume scale a x = scale a y
hence scale a (x − y) = 0
  by (simp add: scale-right-diff-distrib)
  hence x − y = 0 by (simp add: nonzero)
  thus x = y by (simp only: right-minus-eq)
qed

lemma scale-right-imp-eq:
[x ≠ 0; scale a x = scale b x] ⇒ a = b
proof —
assume nonzero: x ≠ 0
assume scale a x = scale b x
hence scale (a − b) x = 0
  by (simp add: scale-left-diff-distrib)
  hence a − b = 0 by (simp add: nonzero)
  thus a = b by (simp only: right-minus-eq)
qed

lemma scale-cancel-left [simp]:
scale a x = scale a y ←→ x = y ∨ a = 0
by (auto intro: scale-left-imp-eq)

lemma scale-cancel-right [simp]:
scale a x = scale b x ←→ a = b ∨ x = 0
by (auto intro: scale-right-imp-eq)
100.3 Real vector spaces

class scaleR =
  fixes scaleR :: real ⇒ 'a ⇒ 'a (infixr * 75)
begin

abbreviation divideR :: 'a ⇒ real ⇒ 'a (infixl '/ 70)
where
  x / R r == scaleR (inverse r) x

end
class real-vector = scaleR + ab-group-add +
  assumes scaleR-add-right: scaleR a (x + y) = scaleR a x + scaleR a y
  and scaleR-add-left: scaleR (a + b) x = scaleR a x + scaleR b x
  and scaleR-scaleR: scaleR a (scaleR b x) = scaleR (a * b) x
  and scaleR-one: scaleR 1 x = x

interpretation real-vector:
  vector-space scaleR :: real ⇒ 'a ⇒ 'a::real-vector
apply unfold-locale
apply (rule scaleR-add-right)
apply (rule scaleR-add-left)
apply (rule scaleR-scaleR)
apply (rule scaleR-one)
done

Recover original theorem names

lemmas scaleR-left-commute = real-vector.scale-left-commute
lemmas scaleR-zero-left = real-vector.scale-zero-left
lemmas scaleR-minus-left = real-vector.scale-minus-left
lemmas scaleR-diff-left = real-vector.scale-left-diff-distrib
lemmas scaleR-setsum-left = real-vector.scale-setsum-left
lemmas scaleR-zero-right = real-vector.scale-zero-right
lemmas scaleR-minus-right = real-vector.scale-minus-right
lemmas scaleR-diff-right = real-vector.scale-right-diff-distrib
lemmas scaleR-setsum-right = real-vector.scale-setsum-right
lemmas scaleR-eq-0-iff = real-vector.scale-eq-0-iff
lemmas scaleR-left-imp-eq = real-vector.scale-left-imp-eq
lemmas scaleR-right-imp-eq = real-vector.scale-right-imp-eq
lemmas scaleR-cancel-left = real-vector.scale-cancel-left
lemmas scaleR-cancel-right = real-vector.scale-cancel-right

Legacy names

lemmas scaleR-left-distrib = scaleR-add-left
lemmas scaleR-right-distrib = scaleR-add-right
lemmas scaleR-left-diff-distrib = scaleR-diff-left
lemmas scaleR-right-diff-distrib = scaleR-diff-right

lemma scaleR-minus1-left [simp]:
  fixes x :: 'a::real-vector
  shows scaleR (-1) x = - x
  using scaleR-minus-left [of 1 x] by simp

class real-algebra = real-vector + ring +
  assumes mult-scaleR-left [simp]: scaleR a x * y = scaleR a (x * y)
  and mult-scaleR-right [simp]: x * scaleR a y = scaleR a (x * y)

class real-algebra-1 = real-algebra + ring-1

class real-div-algebra = real-algebra-1 + division-ring

class real-field = real-div-algebra + field

instantiation real :: real-field
begin

definition real-scaleR-def [simp]: scaleR a x = a * x

instance proof
  qed (simp-all add: algebra-simps)
end

interpretation scaleR-left: additive (λa. scaleR a x::'a::real-vector)
proof qed (rule scaleR-left-distrib)

interpretation scaleR-right: additive (λx. scaleR a x::'a::real-vector)
proof qed (rule scaleR-right-distrib)

lemma nonzero-inverse-scaleR-distrib:
  fixes x :: 'a::real-div-algebra shows
  [a ≠ 0; x ≠ 0] ⇒ inverse (scaleR a x) = scaleR (inverse a) (inverse x)
  by (rule inverse-unique, simp)

lemma inverse-scaleR-distrib:
  fixes x :: 'a::(real-div-algebra, division-ring-inverse-zero)
  shows inverse (scaleR a x) = scaleR (inverse a) (inverse x)
  apply (case_tac a = 0, simp)
  apply (case_tac x = 0, simp)
  apply (erule (1) nonzero-inverse-scaleR-distrib)
done
100.4 Embedding of the Reals into any \textit{real-algebra-1}: of-real

definition
\begin{center}
of-real :: real \Rightarrow 'a::real-algebra-1 where
\end{center}
of-real \, r = \text{scaleR} \, r \, 1

lemma scaleR-conv-of-real: \text{scaleR} \, r \, x = \text{of-real} \, r \ast x
\text{by (simp add: of-real-def)}

lemma of-real-0 [simp]: \text{of-real} \, 0 = 0
\text{by (simp add: of-real-def)}

lemma of-real-1 [simp]: \text{of-real} \, 1 = 1
\text{by (simp add: of-real-def)}

lemma of-real-add [simp]: \text{of-real} \, (x + y) = \text{of-real} \, x + \text{of-real} \, y
\text{by (simp add: of-real-def scaleR-left-distrib)}

lemma of-real-minus [simp]: \text{of-real} \, (- x) = - \text{of-real} \, x
\text{by (simp add: of-real-def)}

lemma of-real-diff [simp]: \text{of-real} \, (x - y) = \text{of-real} \, x - \text{of-real} \, y
\text{by (simp add: of-real-def scaleR-left-diff-distrib)}

lemma of-real-mult [simp]: \text{of-real} \, (x \ast y) = \text{of-real} \, x \ast \text{of-real} \, y
\text{by (simp add: of-real-def mult.commute)}

lemma of-real-setsum [simp]: \text{of-real} \, (\text{setsum} \, f \, s) = (\sum x \in s. \text{of-real} \, (f \, x))
\text{by (induct s rule: infinite-finite-induct) auto}

lemma of-real-setprod [simp]: \text{of-real} \, (\text{setprod} \, f \, s) = (\prod x \in s. \text{of-real} \, (f \, x))
\text{by (induct s rule: infinite-finite-induct) auto}

lemma nonzero-of-real-inverse:
\begin{center}
\begin{align*}
x \neq 0 \Rightarrow \text{of-real} \, (\text{inverse} \, x) & = \\
\text{inverse} \, (\text{of-real} \, x) & = 'a::real-div-algebra
\end{align*}
\end{center}
\text{by (simp add: of-real-def nonzero-inverse-scaleR-distrib)}

lemma of-real-inverse [simp]:
\begin{center}
\begin{align*}
\text{of-real} \, (\text{inverse} \, x) & = \\
\text{inverse} \, (\text{of-real} \, x) & = 'a::{real-div-algebra, division-ring-inverse-zero}
\end{align*}
\end{center}
\text{by (simp add: of-real-def inverse-scaleR-distrib)}

lemma nonzero-of-real-divide:
\begin{center}
\begin{align*}
y \neq 0 \Rightarrow \text{of-real} \, (x / y) & = \\
\text{of-real} \, x / \text{of-real} \, y & = 'a::real-field
\end{align*}
\end{center}
\text{by (simp add: divide-inverse nonzero-of-real-inverse)}

lemma of-real-divide [simp]:
\begin{center}
\begin{align*}
\text{of-real} \, (x / y) & = \\
\end{align*}
\end{center}
(of-real x / of-real y :: 'a::{real-field, field-inverse-zero}) by (simp add: divide-inverse)

lemma of-real-power [simp]:
of_real (x ^ n) = (of-real x :: 'a::{real-algebra-1}) ^ n
by (induct n) simp-all

lemma of-real-eq-iff [simp]: (of-real x = of-real y) = (x = y)
by (simp add: of-real-def)

lemma inj-of-real:
  inj of-real
by (auto intro: injI)

lemmas of-real-eq-0-iff [simp] = of-real-eq-iff [of 0, simplified]

lemma of-real-eq-id [simp]: of-real = (id :: real ⇒ real)
proof
  fix r
  show of-real r = id r
    by (simp add: of-real-def)
qed

Collapse nested embeddings

lemma of-real-of-nat-eq [simp]: of-real (of-nat n) = of-nat n
by (induct n) auto

lemma of-real-of-int-eq [simp]: of-real (of-int z) = of-int z
by (cases z rule: int-diff-cases, simp)

lemma of-real-real-of-nat-eq [simp]: of-real (real n) = of-nat n
by (simp add: real-of-nat-def)

lemma of-real-real-of-int-eq [simp]: of-real (real z) = of-int z
by (simp add: real-of-int-def)

lemma of-real-numeral: of-real (numeral w) = numeral w
using of-real-of-int-eq [of numeral w] by simp

lemma of-real-neg-numeral: of-real (¬ numeral w) = ¬ numeral w
using of-real-of-int-eq [of ¬ numeral w] by simp

Every real algebra has characteristic zero

instance real-algebra-1 < ring-char-0
proof
  from inj-of-real inj-of-nat have inj (of-real ∘ of-nat) by (rule inj-comp)
  then show inj (of-nat :: nat ⇒ 'a) by (simp add: comp-def)
qed
instance real-field < field-char-0 ..

100.5 The Set of Real Numbers

definition Reals :: 'a::real-algebra-1 set where
  Reals = range of-real

notation (xsymbols)
  Reals (\mathbb{R})

lemma Reals-of-real [simp]: of-real r ∈ Reals
by (simp add: Reals-def)

lemma Reals-of-int [simp]: of-int z ∈ Reals
by (subst of-real-of-int-eq [symmetric], rule Reals-of-real)

lemma Reals-of-nat [simp]: of-nat n ∈ Reals
by (subst of-real-of-nat-eq [symmetric], rule Reals-of-real)

lemma Reals-numeral [simp]: numeral w ∈ Reals
by (subst of-real-numeral [symmetric], rule Reals-of-real)

lemma Reals-0 [simp]: 0 ∈ Reals
apply (unfold Reals-def)
apply (rule range-eqI)
apply (rule of-real-0 [symmetric])
done

lemma Reals-1 [simp]: 1 ∈ Reals
apply (unfold Reals-def)
apply (rule range-eqI)
apply (rule of-real-1 [symmetric])
done

lemma Reals-add [simp]: [a ∈ Reals; b ∈ Reals] ⇒ a + b ∈ Reals
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-add [symmetric])
done

lemma Reals-minus [simp]: a ∈ Reals ⇒ − a ∈ Reals
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-minus [symmetric])
done

lemma Reals-diff [simp]: [a ∈ Reals; b ∈ Reals] ⇒ a − b ∈ Reals
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-diff [symmetric])
done

lemma Reals-mult [simp]: \([a \in \text{Reals};\ b \in \text{Reals}] \implies a \ast b \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-mult [symmetric])
done

lemma nonzero-Reals-inverse:
fixes \(a::\text{real-div-algebra}\)
shows \([a \in \text{Reals};\ a \neq 0] \implies \text{inverse } a \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (erule nonzero-of-real-inverse [symmetric])
done

lemma Reals-inverse:
fixes \(a::\text{real-div-algebra, division-ring-inverse-zero}\)
shows \(a \in \text{Reals} \implies \text{inverse } a \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-inverse [symmetric])
done

lemma Reals-inverse iff [simp]:
fixes \(x::\text{real-div-algebra, division-ring-inverse-zero}\)
shows \(\text{inverse } x \in \mathbb{R} \iff x \in \mathbb{R}\)
by (metis Reals-inverse inverse-inverse-eq)

lemma nonzero-Reals-divide:
fixes \(a\ b::\text{real-field}\)
shows \([a \in \text{Reals};\ b \in \text{Reals};\ b \neq 0] \implies a / b \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (erule nonzero-of-real-divide [symmetric])
done

lemma Reals-divide [simp]:
fixes \(a\ b::\text{real-field, field-inverse-zero}\)
shows \([a \in \text{Reals};\ b \in \text{Reals}] \implies a / b \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-divide [symmetric])
done

lemma Reals-power [simp]:
fixes \(a::\text{real-algebra-1}\)
shows \(a \in \text{Reals} \implies a ^ {n} \in \text{Reals}\)
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-power [symmetric])
done

lemma Reals-cases [cases set: Reals]:
  assumes q ∈ ℝ
  obtains (of-real) r where q = of-real r
  unfolding Reals-def
proof
  from q ∈ ℝ: have q ∈ range of-real unfolding Reals-def.
  then obtain r where q = of-real r ..
  then show thesis ..
qed

lemma setsum-in-Reals: assumes ∃ i. i ∈ s =⇒ f i ∈ ℝ shows setsum f s ∈ ℝ
proof (cases finite s)
  case True then show ?thesis using assms
    by (induct s rule: finite-induct) auto
next
  case False then show ?thesis using assms
    by (metis Reals-0 setsum.infinite)
qed

lemma setprod-in-Reals: assumes ∃ i. i ∈ s =⇒ f i ∈ ℝ shows setprod f s ∈ ℝ
proof (cases finite s)
  case True then show ?thesis using assms
    by (induct s rule: finite-induct) auto
next
  case False then show ?thesis using assms
    by (metis Reals-1 setprod.infinite)
qed

lemma Reals-induct [case-names of-real, induct set: Reals]:
  q ∈ ℝ =⇒ (∀ r. P (of-real r)) =⇒ P q
by (rule Reals-cases) auto

100.6 Ordered real vector spaces

class ordered-real-vector = real-vector + ordered-ab-group-add +
  assumes scaleR-left-mono: x ≤ y =⇒ 0 ≤ a =⇒ a *⇘R⇙ x ≤ a *⇘R⇙ y
  assumes scaleR-right-mono: a ≤ b =⇒ 0 ≤ x =⇒ a *⇘R⇙ x ≤ b *⇘R⇙ x
begin

lemma scaleR-mono:
  a ≤ b =⇒ x ≤ y =⇒ 0 ≤ a =⇒ 0 ≤ x =⇒ a *⇘R⇙ x ≤ b *⇘R⇙ y
apply (erule scaleR-right-mono [THEN order-trans], assumption)
apply (erule scaleR-left-mono, assumption)
done
lemma `scaleR-mono'`: 
\[ a \leq b \rightarrow c \leq d \rightarrow 0 \leq a \rightarrow 0 \leq c \rightarrow a * R c \leq b * R d \]
by (rule `scaleR-mono`) (auto intro: order.trans)

lemma `pos-le-divideRI`:
assumes `0 < c`
assumes `c * R a \leq b`
shows `a \leq b / R c`
proof
  from `scaleR-left-mono`[OF assms(2)] assms(1)
  have `c * R a / R c \leq b / R c`
  by simp
  with assms show `?thesis`
  by (simp add: `scaleR-one` `scaleR-scaleR inverse-eq-divide`)
qed

lemma `pos-le-divideR-eq`:
assumes `0 < c`
shows `a \leq b / R c \iff c * R a \leq b`
proof
  assume `a \leq b / R c`
  from `scaleR-left-mono`[OF this] assms
  have `c * R a \leq c * R (b / R c)`
  by simp
  with assms show `c * R a \leq b`
  by (simp add: `scaleR-one` `scaleR-scaleR inverse-eq-divide`)
qed (rule `pos-le-divideRI[OF assms]`)

lemma `scaleR-image-atLeastAtMost`:
\[ c \geq 0 \rightarrow \text{scaleR } c : \{ x..y \} = \{ c * R x..c * R y \} \]
apply (auto intro!: scaleR-left-mono)
apply (rule-tac x = `inverse c * R xa` in image-eql)
apply (simp-all add: `pos-le-divideR-eq`[symmetric] scaleR-scaleR scaleR-one)
done

end

lemma `scaleR-nonneg-nonneg`: \[ 0 \leq a \rightarrow 0 \leq (x::'a::ordered-real-vector) \rightarrow 0 \leq a * R x \]
using `scaleR-left-mono` [of `0 x a`]
by simp

lemma `scaleR-nonneg-nonpos`: \[ 0 \leq a \rightarrow (x::'a::ordered-real-vector) \leq 0 \rightarrow a * R x \leq 0 \]
using `scaleR-left-mono` [of `x 0 a`] by simp

lemma `scaleR-nonpos-nonneg`: \[ a \leq 0 \rightarrow 0 \leq (x::'a::ordered-real-vector) \rightarrow a * R x \leq 0 \]
using \texttt{scaleR-right-mono} \ [of \ a \ 0 \ x] \ by \ \texttt{simp}

\textbf{lemma} \ \texttt{split-scaleR-neg-le}: \ (0 \leq a \land x \leq 0) \ | \ (a \leq 0 \land 0 \leq x) \implies a \ast_R (x::'a::ordered-real-vector) \leq 0 \\
by \ (auto \ simp \ add: \ scaleR-nonneg-nonpos \ scaleR-nonpos-nonpos)

\textbf{lemma} \ \texttt{le-add-iff1}:
\fixes \ c \ d \ e ::'a::ordered-real-vector
\shows \ a \ast_R e + c \leq b \ast_R e + d \iff (a - b) \ast_R e + d \\
by \ (simp \ add: \ algebra-simps)

\textbf{lemma} \ \texttt{le-add-iff2}:
\fixes \ c \ d \ e ::'a::ordered-real-vector \\
\shows \ a \ast_R e + c \leq b \ast_R e + d \iff c \leq (b - a) \ast_R e + d \\
by \ (simp \ add: \ algebra-simps)

\textbf{lemma} \ \texttt{scaleR-left-mono-neg}:
\fixes \ a \ b ::'a::ordered-real-vector \\
\shows \ b \leq a \implies c \leq 0 \implies c \ast_R a \leq c \ast_R b \\
apply \ (drule \ scaleR-left-mono \ [of \ - \ - \ c]) \\
apply \ \texttt{simp-all} \\
done

\textbf{lemma} \ \texttt{scaleR-right-mono-neg}:
\fixes \ c ::'a::ordered-real-vector \\
\shows \ b \leq a \implies c \leq 0 \implies a \ast_R c \leq b \ast_R c \\
apply \ (drule \ scaleR-right-mono \ [of \ - \ - \ c]) \\
apply \ \texttt{simp-all} \\
done

\textbf{lemma} \ \texttt{scaleR-nonpos-nonpos}: \ a \leq 0 \implies (b::'a::ordered-real-vector) \leq 0 \implies 0 \leq a \ast_R b \\
using \ \texttt{scaleR-right-mono-neg} \ [of \ a \ b] \ by \ \texttt{simp}

\textbf{lemma} \ \texttt{split-scaleR-pos-le}:
\fixes \ b ::'a::ordered-real-vector \\
\shows \ (0 \leq a \land 0 \leq b) \lor (a \leq 0 \land b \leq 0) \implies 0 \leq a \ast_R b \\
by \ (auto \ simp \ add: \ scaleR-nonneg-nonneg \ scaleR-nonpos-nonpos)

\textbf{lemma} \ \texttt{zero-le-scaleR-iff}:
\fixes \ b ::'a::ordered-real-vector \\
\shows \ 0 \leq a \ast_R b \iff 0 < a \land 0 \leq b \lor a < 0 \land b \leq 0 \lor a = 0 \ (\text{is} \ ?\text{lhs} = \ ?\text{rhs}) \\
\textbf{proof} \ \texttt{cases} \\
\assume \ a \neq 0 \\
\show \ ?\text{thesis} \\
\textbf{proof} \\
\assume \ \texttt{lhs}: \ ?\text{lhs} \\
\{ 

assume $0 < a$
with lhs have $\text{inverse } a \ast_R 0 \leq \text{inverse } a \ast_R (a \ast_R b)$
by (intro scaleR-mono) auto
hence ?rhs using $(0 < a)$
by simp
}
moreover {
assume $0 > a$
with lhs have $-\text{inverse } a \ast_R 0 \leq -\text{inverse } a \ast_R (a \ast_R b)$
by (intro scaleR-mono) auto
hence ?rhs using $(0 > a)$
by simp
}
ultimately show ?rhs using $(a \neq 0)$ by arith
qed (auto simp: not-le ⟨a = 0⟩ intro!: split-scaleR-pos-le)
qed simp

lemma scaleR-le-0-iff:
fixes b::'a::ordered-real-vector
shows $a \ast_R b \leq 0 \iff 0 < a \land b \leq 0 \lor a < 0 \lor 0 \leq b \lor a = 0$
by (insert zero-le-scaleR-iff [of $-a b$]) force

lemma scaleR-le-cancel-left:
fixes b::'a::ordered-real-vector
shows $c \ast_R a \leq c \ast_R b \iff (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a)$
by (auto simp add: neg-iff scaleR-left-mono scaleR-left-mono-neg
    dest: scaleR-left-mono[where $a=$inverse $c$] scaleR-left-mono-neg[where $c=$inverse $c$])

lemma scaleR-le-cancel-left-pos:
fixes b::'a::ordered-real-vector
shows $0 < c \Rightarrow c \ast_R a \leq c \ast_R b \iff a \leq b$
by (auto simp: scaleR-le-cancel-left)

lemma scaleR-le-cancel-left-neg:
fixes b::'a::ordered-real-vector
shows $c < 0 \Rightarrow c \ast_R a \leq c \ast_R b \iff b \leq a$
by (auto simp: scaleR-le-cancel-left)

lemma scaleR-left-le-one-le:
fixes x::'a::ordered-real-vector and a::real
shows $0 \leq x \Rightarrow a \leq 1 \Rightarrow a \ast_R x \leq x$
using scaleR-right-mono[of $a$ $1$ $x$] by simp

100.7 Real normed vector spaces

class dist =
fixes dist :: 'a ⇒ 'a ⇒ real

class norm =
fixes norm :: 'a ⇒ real
class sgn-div-norm = scaleR + norm + sgn +
assumes sgn-div-norm: sgn $x = x / R \cdot \text{norm} \cdot x$

class dist-norm = dist + norm + minus +
assumes dist-norm: dist $x y = \text{norm}(x - y)$

class open-dist = open + dist +
assumes open-dist: open $S \iff (\forall x \in S. \exists e > 0. \forall y. \text{dist}(y x < e \rightarrow y \in S)$

class real-normed-vector = real-vector + sgn-div-norm + dist-norm + open-dist +
assumes norm-eq-zero [simp]: norm $x = 0 \iff x = 0$
and norm-triangle-ineq: norm $(x + y) \leq \text{norm}(x) + \text{norm}(y)$
and norm-scaleR [simp]: norm $(\text{scaleR}(a) x) = |a| \cdot \text{norm}(x)$

begin

lemma norm-ge-zero [simp]: $0 \leq \text{norm}(x)$
proof –
have $0 = \text{norm}(x + (-1) \cdot R \cdot x)$
using scaleR-add-left [of 1 $-1$ $x]\text{norm-scaleR}[of 0 x]$ by (simp add: scaleR-one)
also have $\ldots \leq \text{norm}(x) + \text{norm}(-1 \cdot R \cdot x)$ by (rule norm-triangle-ineq)
finally show $?\text{thesis}$ by simp
qed

debut

end

class real-normed-algebra = real-algebra + real-normed-vector +
assumes norm-mult-ineq: norm $(x * y) \leq \text{norm}(x) * \text{norm}(y)$

class real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +
assumes norm-one [simp]: norm $1 = 1$

class real-normed-div-algebra = real-div-algebra + real-normed-vector +
assumes norm-mult: norm $(x * y) = \text{norm}(x) * \text{norm}(y)$

class real-normed-field = real-field + real-normed-div-algebra

instance real-normed-div-algebra < real-normed-algebra-1
proof
fix $x y :: 'a$
show $\text{norm}(x * y) \leq \text{norm}(x) * \text{norm}(y)$
by (simp add: norm-mult)
next
have $\text{norm}(1 * 1 :: 'a) = \text{norm}(1 :: 'a) * \text{norm}(1 :: 'a)$
by (rule norm-mult)
thus $\text{norm}(1 :: 'a) = 1$ by simp
qed
lemma norm-zero [simp]: norm (0::'a::real-normed-vector) = 0 by simp

lemma zero-less-norm-iff [simp]:
  fixes x :: 'a::real-normed-vector
  shows (0 < norm x) = (x ≠ 0)
  by (simp add: order-less-le)

lemma norm-not-less-zero [simp]:
  fixes x :: 'a::real-normed-vector
  shows ¬norm x < 0
  by (simp add: linorder-not-less)

lemma norm-le-zero-iff [simp]:
  fixes x :: 'a::real-normed-vector
  shows (norm x ≤ 0) = (x = 0)
  by (simp add: order-le-less)

lemma norm-minus-cancel [simp]:
  fixes x :: 'a::real-normed-vector
  shows norm (−x) = norm x
proof −
  have norm (−x) = norm (scaleR (−1) x)
    by (simp only: scaleR-minus-left scaleR-one)
  also have . . . = |−1| ∗ norm x
    by (rule norm-scaleR)
  finally show ?thesis by simp
qed

lemma norm-minus-commute:
  fixes a b :: 'a::real-normed-vector
  shows norm (a − b) = norm (b − a)
proof −
  have norm (−(b − a)) = norm (b − a)
    by (rule norm-minus-cancel)
  thus ?thesis by simp
qed

lemma norm-triangle-ineq2:
  fixes a b :: 'a::real-normed-vector
  shows norm a − norm b ≤ norm (a − b)
proof −
  have norm (a − b + b) ≤ norm (a − b) + norm b
    by (rule norm-triangle-ineq)
  thus ?thesis by simp
qed

lemma norm-triangle-ineq3:
  fixes a b :: 'a::real-normed-vector
shows $|\|a\| - \|b\|| \leq \|a - b\|$
apply (subst abs-le-iff)
apply auto
apply (rule norm-triangle-ineq2)
apply (subst norm-minus-commute)
apply (rule norm-triangle-ineq2)
done

lemma norm-triangle-ineq4:
fixes $a,b :: 'a::real-normed-vector$
shows $\|a - b\| \leq \|a\| + \|b\|$
proof -
  have $\|a - b\| \leq \|a\| + \|b\|$
    by (rule norm-triangle-ineq)
  then show ?thesis by simp
qed

lemma norm-diff-ineq:
fixes $a,b :: 'a::real-normed-vector$
shows $\|a\| - \|b\| \leq \|a + b\|$
proof -
  have $\|a\| - \|b\| \leq \|a\| + \|b\|$
    by (rule norm-triangle-ineq2)
  thus ?thesis by simp
qed

lemma norm-diff-triangle-ineq:
fixes $a,b,c,d :: 'a::real-normed-vector$
shows $\|a + b - (c + d)\| \leq \|a - c\| + \|b - d\|$
proof -
  have $\|a + b - (c + d)\| = \|a - c\| + \|b - d\|$
    by (simp add: algebra-simps)
  also have ... $\leq \|a - c\| + \|b - d\|$
    by (rule norm-triangle-ineq)
  finally show ?thesis .
qed

lemma norm-triangle-mono:
fixes $a,b :: 'a::real-normed-vector$
shows $\|a\| \leq r; \|b\| \leq s \implies \|a + b\| \leq r + s$
by (metis add-mono-thms-linordered-semiring(1) norm-triangle-ineq order.trans)

lemma norm-setsum:
fixes $f :: 'a => 'b::real-normed-vector$
shows $\|\sum_{i\in A} f i\| \leq \sum_{i\in A} \|f i\|$
by (induct A rule: infinite-finite-induct) (auto intro: norm-triangle-mono)

lemma setsum-norm-le:
fixes $f :: 'a => 'b::real-normed-vector$
assumes $f g$: \( \forall x \in S. \text{norm}(f \cdot x) \leq g \cdot x \)

shows $\text{norm}(\text{setsum} f S) \leq \text{setsum} g S$

by (rule order-trans [OF norm-setsum setsum-mono]) (simp add: $f g$)

lemma abs-norm-cancel [simp]:
fixes $a :: \text{real-normed-vector}$
shows $|\text{norm} a| = \text{norm} a$

by (rule abs-of-nonneg [OF norm-ge-zero])

lemma norm-add-less:
fixes $x y :: \text{real-normed-vector}$
shows $[\text{norm} x < r; \text{norm} y < s] \implies \text{norm}(x + y) < r + s$

by (rule order-le-less-trans [OF norm-triangle-ineq add-strict-mono])

lemma norm-mult-less:
fixes $x y :: \text{real-normed-algebra}$
shows $[\text{norm} x < r; \text{norm} y < s] \implies \text{norm}(x \cdot y) < r \cdot s$

apply (rule order-le-less-trans [OF norm-mult-ineq])
apply (simp add: mult-strict-mono)
done

lemma norm-of-real [simp]:
$\text{norm}(\text{of-real} r :: \text{real-normed-algebra-1}) = |r|$

unfolding of-real-def by simp

lemma norm-numeral [simp]:
$\text{norm}(\text{numeral} w :: \text{real-normed-algebra-1}) = \text{numeral} w$

by (subst of-real-numeral [symmetric], subst norm-of-real, simp)

lemma nonzero-norm-inverse:
fixes $a :: \text{real-normed-div-algebra}$
shows $a \neq 0 \implies \text{norm}(\text{inverse} a) = \text{inverse}(\text{norm} a)$

apply (rule inverse-unique [symmetric])
apply (simp add: norm-mult [symmetric])
done
lemma norm-inverse:
  fixes a :: 'a::{real-normed-div-algebra, division-ring-inverse-zero}
  shows norm (inverse a) = inverse (norm a)
  apply (case_tac a = 0, simp)
  apply (erule nonzero-norm-inverse)
  done

lemma nonzero-norm-divide:
  fixes a b :: 'a::{real-normed-field}
  shows b ≠ 0 ⇒ norm (a / b) = norm a / norm b
  by (simp add: divide-inverse norm-mult nonzero-norm-inverse)

lemma norm-divide:
  fixes a b :: 'a::{real-normed-field, field-inverse-zero}
  shows norm (a / b) = norm a / norm b
  by (simp add: divide-inverse norm-mult norm-inverse)

lemma norm-power-ineq:
  fixes x :: 'a::{real-normed-algebra-1}
  shows norm (x ^ n) ≤ norm x ^ n
  proof (induct n)
    case 0 show norm (x ^ 0) ≤ norm x ^ 0 by simp
  next
    case (Suc n)
    have norm (x * x ^ n) ≤ norm x * norm (x ^ n)
      by (rule norm-mult-ineq)
    also from Suc have "... ≤ norm x * norm x ^ n"
      using norm-ge-zero by (rule mult-left-mono)
    finally show norm (x ^ Suc n) ≤ norm x ^ Suc n
      by simp
  qed

lemma norm-power:
  fixes x :: 'a::{real-normed-div-algebra}
  shows norm (x ^ n) = norm x ^ n
  by (induct n) (simp-all add: norm-mult)

lemma setprod-norm:
  fixes f :: 'a::{comm-semiring-1,real-normed-div-algebra}
  shows setprod (%x. norm (f x)) A = norm (setprod f A)
  by (induct A rule: infinite-finite-induct) (auto simp: norm-mult)

lemma norm-setprod-le:
  norm (setprod f A) ≤ (∏ a∈A. norm (f a :: 'a::{real-normed-algebra-1, comm-monoid-mult})))
  proof (induction A rule: infinite-finite-induct)
    case (insert a A)
    then have norm (setprod f (insert a A)) ≤ norm (f a) * norm (setprod f A)
      by (simp add: norm-mult-ineq)
also have \( \text{norm} \ (\text{setprod } f \ A) \leq (\prod a \in A. \text{norm } (f \ a)) \)
  by (rule insert)
finally show ?case
  by (simp add: insert mult-left-mono)
qed simp-all

lemma \( \text{norm-setprod-diff} \):
  fixes \( z \ w :: 'a::{\text{real-normed-algebra-1}, \text{comm-monoid-mult}} \)
  shows \( (\prod i \in I. \text{norm } (z \ i) \leq 1) \implies (\prod i \in I. \text{norm } (w \ i) \leq 1) \implies \text{norm } (((\prod i \in I. \ z \ i) - (\prod i \in I. \ w \ i)) \leq (\sum i \in I. \text{norm } (z \ i - w \ i)) \)
proof (induction \( I \) rule: \( \text{infinite-finite-induct} \))
  case (insert \( i \) \( I \))
  note \( \text{insert.hyps[simp]} \)

  have \( \text{norm } (((\prod i \in \text{insert } i \ I. \ z \ i) - (\prod i \in \text{insert } i \ I. \ w \ i)) = \text{norm } (((\prod i \in I. \ z \ i) * (z \ i - w \ i) + ((\prod i \in I. \ z \ i) - (\prod i \in I. \ w \ i)) * w \ i) \)
  (\( i s = \text{norm } (?t1 + ?t2) \))
  by (auto simp add: \text{field-simps})

  also have ... \leq \text{norm } ?t1 + \text{norm } ?t2
  by (rule \( \text{norm-triangle-ineq} \))

  also have \( \text{norm } ?t1 \leq \text{norm } ((\prod i \in I. \ z \ i) * \text{norm } (z \ i - w \ i) \)
  by (rule \( \text{norm-mult-ineq} \))

  also have \( \ldots \leq (\prod i \in I. \text{norm } (z \ i)) * \text{norm}(z \ i - w \ i) \)
  by (rule \( \text{mult-right-mono} \) (auto intro: \( \text{norm-setprod-le} \))

  also have \( (\prod i \in I. \text{norm } (z \ i)) \leq (\prod i \in I. \text{norm } (z \ i) \)
  by (intro \( \text{setprod-constant} \) (auto intro: \( \text{insert} \))

  also have \( \text{norm } ?t2 \leq \text{norm } (((\prod i \in I. \ z \ i) - (\prod i \in I. \ w \ i)) * \text{norm } (w \ i) \)
  by (rule \( \text{norm-mult-ineq} \))

  also have \( \text{norm } (w \ i) \leq 1 \)
  by (auto intro: \( \text{insert} \))

  also have \( \text{norm } ((\prod i \in I. \ z \ i) - (\prod i \in I. \ w \ i)) \leq (\sum i \in I. \text{norm } (z \ i - w \ i)) \)
  using \( \text{insert} \) by auto

  finally show ?case
  by (auto simp add: ac-simps \( \text{mult-right-mono mult-left-mono} \))
qed simp-all

lemma \( \text{norm-power-diff} \):
  fixes \( z \ w :: 'a::{\text{real-normed-algebra-1}, \text{comm-monoid-mult}} \)
  assumes \( \text{norm } z \leq 1 \text{ norm } w \leq 1 \)
  shows \( \text{norm } (z^m - w^m) \leq m * \text{norm } (z - w) \)
proof
  have \( \text{norm } (z^m - w^m) = \text{norm } ((\prod i < m. \ z) - (\prod i < m. \ w)) \)
  by (simp add: \text{setprod-constant})

  also have \( \ldots \leq (\sum i < m. \text{norm } (z - w)) \)
  by (intro \( \text{norm-setprod-diff} \) (auto simp add: assms))

  also have \( \ldots = m * \text{norm } (z - w) \)
  by (simp add: \text{real-of-nat-def})

  finally show ?thesis 
qed
100.8 Metric spaces

class metric-space = open-dist +
  assumes dist-eq-0-iff [simp]: dist x y = 0 ↔ x = y
  assumes dist-triangle2: dist x y ≤ dist x z + dist y z

begin

lemma dist-self [simp]: dist x x = 0
  by simp

lemma zero-le-dist [simp]: 0 ≤ dist x y
  using dist-triangle2 [of x x y] by simp

lemma zero-less-dist-iff: 0 < dist x y ↔ x ≠ y
  by (simp add: less-le)

lemma dist-not-less-zero [simp]: ¬ dist x y < 0
  by (simp add: not-less)

lemma dist-le-zero-iff [simp]: dist x y ≤ 0 ↔ x = y
  by (simp add: le-less)

lemma dist-commute: dist x y = dist y x
  proof (rule order-antisym)
    show dist x y ≤ dist y x
      using dist-triangle2 [of x y x] by simp
    show dist y x ≤ dist x y
      using dist-triangle2 [of y x y] by simp
  qed

lemma dist-triangle: dist x z ≤ dist x y + dist y z
  using dist-triangle2 [of x z y] by (simp add: dist-commute)

lemma dist-triangle3: dist x y ≤ dist a x + dist a y
  using dist-triangle2 [of x y a] by (simp add: dist-commute)

lemma dist-triangle-alt: shows dist y z ≤ dist x y + dist x z
  by (rule dist-triangle3)

lemma dist-pos-lt:
  shows x ≠ y ==> 0 < dist x y
  by (simp add: zero-less-dist-iff)

lemma dist-nz:
  shows x ≠ y ↔ 0 < dist x y
  by (simp add: zero-less-dist-iff)

lemma dist-triangle-le:
  shows dist x z + dist y z ≤ e ==> dist x y ≤ e
by (rule order-trans [OF dist-triangle2])

lemma dist-triangle-lt:
  shows dist x z + dist y z < e =⇒ dist x y < e
by (rule le-less-trans [OF dist-triangle2])

lemma dist-triangle-half-l:
  shows dist x1 y < e / 2 =⇒ dist x2 y < e / 2 =⇒ dist x1 x2 < e
by (rule dist-triangle-lt [where z=y], simp)

lemma dist-triangle-half-r:
  shows dist y x1 < e / 2 =⇒ dist y x2 < e / 2 =⇒ dist x1 x2 < e
by (rule dist-triangle-half-l, simp-all add: dist-commute)

subclass topological-space
proof
  have ∃ e::real. 0 < e
    by (fast intro: zero-less-one)
  then show open UNIV
    unfolding open-dist
    by simp
next
  fix S T assume open S open T
  then show open (S ∩ T)
    unfolding open-dist
    apply clarify
    apply (drule (1) bspec)+
    apply (clarify, rename-tac r s)
    apply (rule-tac x=min r s in exI, simp)
    done
next
  fix K assume ∀ S ∈ K. open S thus open (⋃ K)
    unfolding open-dist
    by fast
qed

lemma open-ball: open {y. dist x y < d}
proof (unfold open-dist, intro ballI)
  fix y assume *: y ∈ {y. dist x y < d}
  then show ∃ e>0. ∀ z. dist z y < e =⇒ z ∈ {y. dist x y < d}
    by (auto intro!: exI[of - d - dist x y] simp: field-simps dist-triangle-lt)
qed

subclass first-countable-topology
proof
  fix x
  show ∃ A::nat ⇒ 'a set. (∀ i. x ∈ A i ∧ open (A i)) ∧ (∀ S. open S ∧ x ∈ S =⇒ (∃ i. A i ⊆ S))
    proof (safe intro!: exI[of - λn. {y. dist x y < inverse (Suc n)}])
      fix S assume open S x ∈ S
      then obtain e where e: 0 < e and {y. dist x y < e} ⊆ S
by (auto simp: open-dist subset-eq dist-commute)
moreover 
from e obtain i where inverse (Suc i) < e 
  by (auto dest!: reals-Archimedean)
then have \{y. dist x y < inverse (Suc i)\} ⊆ \{y. dist x y < e\} 
  by auto
ultimately show \exists i. \{y. dist x y < inverse (Suc i)\} ⊆ S 
  by blast
qed (auto intro: open-ball)
qed
end

instance metric-space ⊆ t2-space
proof
fix x y :: 'a::metric-space
assume xy: x ≠ y
let ?U = \{y'. dist x y' < dist x y / 2\}
let ?V = \{x'. dist y x' < dist x y / 2\}
have th0: ∀d x z. (d x z :: real) ≤ d x y + d y z ⇒ d y z = d z y 
  ⇒¬(d x y * 2 < d x z ∧ d z y * 2 < d x z) by arith
  using dist-pos-lt[OF xy] th0[of dist, OF dist-triangle dist-commute]
  using open-ball[of - dist x y / 2] by auto
then show \exists U V. open U ∧ open V ∧ x ∈ U ∧ y ∈ V ∧ U ∩ V = {}
  by blast
qed

Every normed vector space is a metric space.

instance real-normed-vector < metric-space
proof
fix x y z :: 'a show dist x y = 0 ↔ x = y
  unfolding dist-norm by simp
next 
fix x y z :: 'a show dist x y ≤ dist x z + dist y z
  unfolding dist-norm
  using norm-triangle-ineq4 [of x - z y - z] by simp
qed

100.9 Class instances for real numbers

instantiation real :: real-normed-field
begin

definition dist-real-def:
  dist x y = |x - y|

definition open-real-def [code del]:
  open (S :: real set) ↔ (∀x∈S. ∃e>0. ∀y. dist y x < e → y ∈ S)
**Theorem** "Real-Vector-Spaces"

**Definition**  
real-norm-def  
\[ \text{norm } r = |r| \]

**Instance**  
apply (intro-classes, unfold real-norm-def real-scaleR-def)  
apply (rule dist-real-def)  
apply (rule open-real-def)  
apply (simp add: sgn-real-def)  
apply (rule abs-eq-0)  
apply (rule abs-triangle-ineq)  
apply (rule abs-mult)  
done

end

declare [[code abort: open :: real set ⇒ bool]]

**Instance** real :: linorder-topology  
**Proof**  
show (open :: real set ⇒ bool) = generate-topology (range lessThan ∪ range greaterThan)  
proof (rule ext, safe)  
fix S :: real set  
assume open S  
then obtain f where \( \forall x \in S. \quad 0 < f x \land (\forall y. \quad x < f x \rightarrow y \in S) \)  
unfolding open-real-def bchoice_iff ..  
then have \( S = (\bigcup x \in S. \{ x - f x \} \cap \{..< x + f x\} \)  
by (fastforce simp: dist-real-def)  
show generate-topology (range lessThan ∪ range greaterThan) S  
apply (subst *)  
apply (intro generate-topology-Union generate-topology.Int)  
apply (auto intro: generate-topology.Basis)  
done

next
fix S :: real set  
assume generate-topology (range lessThan ∪ range greaterThan) S  
moreover have \( \forall a::real. \quad \text{open } \{..<a\} \)  
unfolding open-real-def dist-real-def  
proof clarify  
fix x a :: real  
assume x < a  
hence \( 0 < a - x \land (\forall y. \quad |y - x| < a - x \rightarrow y \in \{..<a\}) \) by auto  
thus \( \exists e>0. \forall y. \quad |y - x| < e \rightarrow y \in \{..<a\} \) ..  
qed

moreover have \( \forall a::real. \quad \text{open } \{a<..\} \)  
unfolding open-real-def dist-real-def  
proof clarify  
fix x a :: real  
assume a < x  
hence \( 0 < x - a \land (\forall y. \quad |y - x| < x - a \rightarrow y \in \{a<..\}) \) by auto
thus $\exists e > 0. \forall y. |y - x| < e \rightarrow y \in \{a<..\}$ ..

qed

ultimately show open $S$

by induct auto

qed

qed

instance real :: linear-continuum-topology ..

lemmas open-real-greaterThan = open-greaterThan[where 'a=real]

lemmas open-real-lessThan = open-lessThan[where 'a=real]

lemmas open-real-greaterThanLessThan = open-greaterThanLessThan[where 'a=real]

lemmas closed-real-atMost = closed-atMost[where 'a=real]

lemmas closed-real-atLeast = closed-atLeast[where 'a=real]

lemmas closed-real-atLeastAtMost = closed-atLeastAtMost[where 'a=real]

100.10 Extra type constraints

Only allow open in class topological-space.

setup ⟨⟨ Sign.add-const-constraint
 (@{const-name open}, SOME @{typ 'a::topological-space set ⇒ bool}) ⟩⟩

Only allow dist in class metric-space.

setup ⟨⟨ Sign.add-const-constraint
 (@{const-name dist}, SOME @{typ 'a::metric-space ⇒ 'a ⇒ real}) ⟩⟩

Only allow norm in class real-normed-vector.

setup ⟨⟨ Sign.add-const-constraint
 (@{const-name norm}, SOME @{typ 'a::real-normed-vector ⇒ real}) ⟩⟩

100.11 Sign function

lemma norm-sgn:

$\text{norm (sgn (x::'a::real-normed-vector)) = (if x = 0 then 0 else 1)}$

by (simp add: sgn-div-norm)

lemma sgn-zero [simp]: $\text{sgn(0::'a::real-normed-vector) = 0}$

by (simp add: sgn-div-norm)

lemma sgn-zero-iff: $(\text{sgn(x::'a::real-normed-vector) = 0}) = (x = 0)$

by (simp add: sgn-div-norm)

lemma sgn-minus: $\text{sgn} (- x) = - \text{sgn(x::'a::real-normed-vector)}$

by (simp add: sgn-div-norm)

lemma sgn-scaleR:

$\text{sgn (scaleR r x)} = \text{scaleR (sgn r) (sgn(x::'a::real-normed-vector))}$

by (simp add: sgn-div-norm ac-simps)
lemma sgn-one [simp]: sgn (1::'a::real-normed-algebra-1) = 1
by (simp add: sgn-div-norm)

lemma sgn-of-real:
  sgn (of-real r::'a::real-normed-algebra-1) = of-real (sgn r)
unfolding of-real-def by (simp only: sgn-scaleR sgn-one)

lemma sgn-mult:
  fixes x y :: 'a::real-normed-div-algebra
  shows sgn (x * y) = sgn x * sgn y
by (simp add: sgn-div-norm norm-mult mult.commute)

lemma real-sgn-eq:
  sgn (x :: real) = x / |x|
by (simp add: sgn-div-norm divide_inverse)

lemma real-sgn-pos:
  0 < (x :: real) =⇒ sgn x = 1
unfolding real-sgn-eq by simp

lemma real-sgn-neg:
  (x :: real) < 0 =⇒ sgn x = -1
unfolding real-sgn-eq by simp

lemma zero-le-sgn-iff [simp]: 0 ≤ sgn x ←→ 0 ≤ (x::real)
by (cases 0 :: real x rule: linorder_cases) simp-all

lemma zero-less-sgn-iff [simp]: 0 < sgn x ←→ 0 < (x::real)
by (cases 0 :: real x rule: linorder_cases) simp-all

lemma sgn-le-0-iff [simp]: sgn x ≤ 0 ←→ (x::real) ≤ 0
by (cases 0 :: real x rule: linorder_cases) simp-all

lemma sgn-less-0-iff [simp]: sgn x < 0 ←→ (x::real) < 0
by (cases 0 :: real x rule: linorder_cases) simp-all

lemma norm-conv-dist: norm x = dist x 0
unfolding dist-norm by simp

100.12 Bounded Linear and Bilinear Operators
locale linear = additive f for f :: 'a::real-vector ⇒ 'b::real-vector +
  assumes scaleR: f (scaleR r x) = scaleR r (f x)

lemma linearI:
  assumes f x y. f (x + y) = f x + f y
  assumes f x. f (c *R x) = c *R f x
  shows linear f
  by default (rule assms)+

locale bounded-linear = linear f for f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector


+ assumes bounded: $\exists K. \forall x.\ norm (f x) \leq \norm x \ast K$

begin

lemma pos-bounded:
$\exists K > 0. \forall x.\ norm (f x) \leq \norm x \ast K$

proof –
obtain K where \[x.\ norm (f x) \leq \norm x \ast K\]
  using bounded by fast
show thesis
proof (intro exI impI conjI allI)

show 0 < max 1 K
  by (rule order-less-le-trans [OF zero_less_one max_cobounded1])
next
fix x
have norm (f x) \leq \norm x \ast K using K.
also have \ldots \leq \norm x \ast max 1 K
  by (rule mult-left-mono [OF max_cobounded2 norm_ge_zero])
finally show norm (f x) \leq \norm x \ast max 1 K.
qed

qed

lemma nonneg-bounded:
$\exists K \geq 0. \forall x.\ norm (f x) \leq \norm x \ast K$

proof –
from pos-bounded
show thesis by (auto intro: order_less_imp_le)
qed

lemma linear: linear f ..

end

lemma bounded-linear-intro:
assumes $\forall x y. f (x + y) = f x + f y$
assumes $\forall x. f (\text{scaleR} r x) = \text{scaleR} r (f x)$
assumes $\forall x.\ norm (f x) \leq \norm x \ast K$
shows bounded-linear f
by default (fast intro: assms)+

locale bounded-bilinear =
  fixes prod :: ('a::real-normed-vector, 'b::real-normed-vector)
  => 'c::real-normed-vector
  (infix1 ** 70)
assumes add-left: prod (a + a') b = prod a b + prod a' b
assumes add-right: prod a (b + b') = prod a b + prod a b'
assumes scaleR-left: prod (scaleR r a) b = scaleR r (prod a b)
assumes scaleR-right: prod a (scaleR r b) = scaleR r (prod a b)
assumes bounded: $\exists K. \forall a b.\ norm (\text{prod} a b) \leq \norm a \ast \norm b \ast K$
begin

lemma pos-bounded:
  \( \exists K > 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K \)
apply (cut-tac bounded, erule \( \exists \)E)
apply (rule-tac \( x=\max 1 \) \( K \) in extl, safe)
apply (rule order-less-le-trans [OF zero-less-one max.cobounded1])
apply (drule spec, drule spec, erule order-trans)
apply (rule mult-left-mono [OF max.cobounded2])
apply (intro mult-nonneg-nonneg norm-ge-zero)
done

lemma nonneg-bounded:
  \( \exists K \geq 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K \)
proof -
  from pos-bounded
  show ?thesis by (auto intro: order-less-imp-le)
qed

lemma additive-right: additive (\( \lambda b. \text{prod } a b \))
by (rule additive.intro, rule add-right)

lemma additive-left: additive (\( \lambda a. \text{prod } a b \))
by (rule additive.intro, rule add-left)

lemma zero-left: \( \text{prod } 0 \) \( b = 0 \)
by (rule additive.zero [OF additive-left])

lemma zero-right: \( \text{prod } a 0 = 0 \)
by (rule additive.zero [OF additive-right])

lemma minus-left: \( \text{prod } (-a) \) \( b = - \text{prod } a b \)
by (rule additive.minus [OF additive-left])

lemma minus-right: \( \text{prod } a (-b) = - \text{prod } a b \)
by (rule additive.minus [OF additive-right])

lemma diff-left:
  \( \text{prod } (a - a') \) \( b = \text{prod } a b - \text{prod } a' b \)
by (rule additive.diff [OF additive-left])

lemma diff-right:
  \( \text{prod } a (b - b') = \text{prod } a b - \text{prod } a b' \)
by (rule additive.diff [OF additive-right])

lemma bounded-linear-left:
  bounded-linear (\( \lambda a. \text{a ** b} \))
apply (cut-tac bounded, safe)
apply (rule-tac \( K=\text{norm } b * K \) in bounded-linear-intro)
apply (rule add-left)
apply (rule scaleR-left)
apply (simp add: ac-simps)
done

lemma bounded-linear-right:
  bounded-linear (λb. a ** b)
apply (cut-tac bounded, safe)
apply (rule-tac K=norm a * K in bounded-linear-intro)
apply (rule add-right)
apply (rule scaleR-right)
apply (simp add: ac-simps)
done

lemma prod-diff-prod:
  (x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b)
by (simp add: diff-left diff-right)

end

lemma bounded-linear-ident[simp]: bounded-linear (λx. x)
  by default (auto intro!: exI[of - 1])

lemma bounded-linear-zero[simp]: bounded-linear (λx. 0)
  by default (auto intro!: exI[of - 1])

lemma bounded-linear-add:
  assumes bounded-linear f
  assumes bounded-linear g
  shows bounded-linear (λx. f x + g x)
proof –
  interpret f: bounded-linear f by fact
  interpret g: bounded-linear g by fact
  show ?thesis
  proof
    from f.bounded obtain Kf where Kf: ∀x. norm (f x) ≤ norm x * Kf by blast
    from g.bounded obtain Kg where Kg: ∀x. norm (g x) ≤ norm x * Kg by blast
    show ∃K. ∀x. norm (f x + g x) ≤ norm x * K
      using add-mono[OF Kf Kg]
      by (intro exI[of - Kf + Kg]) (auto simp: field-simps intro: norm-triangle-ineq order-trans)
  qed (simp-all add: f.add g.add f.scaleR g.scaleR scaleR-right-distrib)
  qed

lemma bounded-linear-minus:
  assumes bounded-linear f
  shows bounded-linear (λx. - f x)
proof
interpret f: bounded-linear f by fact
show ?thesis apply (unfold-locales)
  apply (simp add: f.add)
  apply (simp add: f.scaleR)
  apply (simp add: f.bounded)
done
qed

lemma bounded-linear-compose:
assumes bounded-linear f
assumes bounded-linear g
shows bounded-linear (\lambda x. f (g x))
proof
interpret f: bounded-linear f by fact
interpret g: bounded-linear g by fact
show ?thesis proof
  fix x y show f (g (x + y)) = f (g x) + f (g y)
    by (simp only: f.add g.add)
next
  fix r x show f (g (scaleR r x)) = scaleR r (f (g x))
    by (simp only: f.scaleR g.scaleR)
next
  from f.pos-bounded
  obtain Kf where f: \lambda x. norm (f x) \leq norm x * Kf and Kf: 0 < Kf by fast
  from g.pos-bounded
  obtain Kg where g: \lambda x. norm (g x) \leq norm x * Kg by fast
  show \exists K. \forall x. norm (f (g x)) \leq norm x * K
    proof (intro exI allI)
      fix x
      have norm (f (g x)) \leq norm (g x) * Kf
        using f.
      also have ... \leq (norm x * Kg) * Kf
        using g Kf [THEN order-less-imp-le] by (rule mult-right-mono)
      also have (norm x * Kg) * Kf = norm x * (Kg * Kf)
        by (rule mult.assoc)
      finally show norm (f (g x)) \leq norm x * (Kg * Kf).
    qed
  qed
qed

lemma bounded-bilinear-mult:
bounded-bilinear (op * :: 'a => 'a => 'a::real-normed-algebra)
apply (rule bounded-bilinear.intro)
apply (rule distrib-right)
apply (rule distrib-left)
apply (rule mult-scaleR-left)
apply (rule mult-scaleR-right)
apply (rule-tac x=1 in exI)
apply \((\text{simp add: norm-mult-ineq})\)
done

**lemma** bounded-linear-mult-left:
  \(\text{bounded-linear} (\lambda x::'a::real-normed-algebra. x * y)\)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-left)

**lemma** bounded-linear-mult-right:
  \(\text{bounded-linear} (\lambda y::'a::real-normed-algebra. x * y)\)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-right)

**lemmas** bounded-linear-mult-const =
  bounded-linear-mult-left [THEN bounded-linear-compose]

**lemmas** bounded-linear-const-mult =
  bounded-linear-mult-right [THEN bounded-linear-compose]

**lemma** bounded-linear-divide:
  \(\text{bounded-linear} (\lambda x::'a::real-normed-field. x / y)\)
  unfolding divide-inverse by (rule bounded-linear-mult-left)

**lemma** bounded-bilinear-scaleR: bounded-bilinear scaleR
  apply (rule bounded-bilinear.intro)
  apply (rule scaleR-left-distrib)
  apply (rule scaleR-right-distrib)
  apply simp
  apply (rule scaleR-left-commute)
  apply (rule-tac \(x=1\) in exI, simp)
done

**lemma** bounded-linear-scaleR-left: bounded-linear \((\lambda r. \text{scaleR } r x)\)
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-left)

**lemma** bounded-linear-scaleR-right: bounded-linear \((\lambda x. \text{scaleR } x r)\)
  using bounded-bilinear-scaleR
  by (rule bounded-bilinear.bounded-linear-right)

**lemma** bounded-linear-of-real: bounded-linear \((\lambda r. \text{of-real } r)\)
  unfolding of-real-def by (rule bounded-linear-scaleR-left)

**lemma** real-bounded-linear:
  \(\text{fixes } f :: \text{real } \Rightarrow \text{real}\)
  \(\text{shows } \text{bounded-linear } f \iff (\exists c::\text{real. } f = (\lambda x. x * c))\)
  \(\text{proof} -\)
  \{ \text{fix } x \text{ assume } \text{bounded-linear } f\)
  then interpret bounded-linear \(f\).
from scaleR[of x 1] have f x = x * f 1
  by simp 
then show ?thesis
  by (auto intro: exI[of - f 1] bounded-linear-mult-left)
qed

instance real-normed-algebra-1 ⊆ perfect-space
proof
  fix x :: 'a
  show ¬ open {x}
    unfolding open-dist dist-norm
    by (clarsimp, rule_tac x = x + of-real (e/2) in exI, simp)
qed

100.13 Filters and Limits on Metric Space

lemma (in metric-space) nhds-metric: nhds x = (INF e:{0 <..}. principal {y. dist y x < e})
unfolding nhds-def
proof (safe intro!: INF-eq)
  fix S assume open S x ∈ S
  then obtain e where {y. dist y x < e} ⊆ S 0 < e
    by (auto simp: open-dist subset-eq)
  then show ∃ e∈{0<..}. principal {y. dist y x < e} ⊆ principal S
    by auto
qed (auto intro: exI[of - {y. dist x y < e} for e] open-ball simp: dist-commute)

lemma (in metric-space) tendsto-iff:
  (f ----> l) F ←→ (∀ e>0. eventually (λx. dist (f x) l < e) F)
unfolding nhds-metric filterlim-INF filterlim-principal by auto

lemma (in metric-space) tendstoI:
  (λe. 0 < e → eventually (λx. dist (f x) l < e) F) ⇒ (f ----> l) F
by (auto simp: tendsto-iff)

lemma (in metric-space) tendstoD:
  (f ----> l) F ⇒ 0 < e ⇒ eventually (λx. dist (f x) l < e) F
by (auto simp: tendsto-iff)

lemma (in metric-space) eventually-nhds-metric:
  eventually P (nhds a) ←→ (∃ d>0. ∀ x. dist x a < d → P x)
unfolding nhds-metric
by (subst eventually-INF-base)
  (auto simp: eventually-principal Bex-def subset-eq intro: exI[of - min a b for a b])

lemma eventually-at:
  fixes a :: 'a :: metric-space
  shows eventually P (at a within S) ←→ (∃ d>0. ∀ x∈S. x ≠ a ∧ dist x a < d}
THEORY "Real-Vector-Spaces"

\[ \rightarrow P x \]
unfolding eventually-at-filter eventually-nhds-metric by (auto simp: dist-nz)

lemma eventually-at-le:
  fixes a :: 'a::metric-space
  shows eventually P (at a within S) \iff \exists d>0. \forall x \in S. x \neq a \land dist x a \leq d \rightarrow P x
  unfolding eventually-at-filter eventually-nhds-metric
  apply auto
  apply (rule_tac x=d / 2 in exI)
  apply auto
  done

lemma metric-tendsto-imp-tendsto:
  fixes a :: 'a::metric-space and b :: 'b::metric-space
  assumes f: (f ----> a) F
  assumes le: eventually (\lambda x. dist (g x) b \leq dist (f x) a) F
  shows (g ----> b) F
proof (rule tendstoI)
  fix e :: real assume 0 < e
  with f have eventually (\lambda x. dist (f x) a < e) F by (rule tendstoD)
  with le show eventually (\lambda x. dist (g x) b < e) F
    using le-less-trans by (rule eventually-elim2)
qed

lemma filterlim-real-sequentially: LIM x sequentially. real x :> at-top
  unfolding filterlim-at-top
  apply (intro allI)
  apply (rule-tac c=natceiling (Z + 1) in eventually-sequentiallyI)
  apply (auto simp: natceiling-le-eq)
  done

100.13.1 Limits of Sequences

lemma LIMSEQ-def: X -----> (L::'a::metric-space) \iff (\forall r>0. \exists n. \forall n\geq n). dist (X n) L < r
  unfolding tendsto-iff eventually-sequentially ..

lemma LIMSEQ-iff-nz: X -----> (L::'a::metric-space) = (\forall r>0. \exists n>0. \forall n\geq n). dist (X n) L < r
  unfolding LIMSEQ-def by (metis Suc-leD zero-less-Suc)

lemma metric-LIMSEQ-I:
  (\forall r. 0 < r \implies \exists n. \forall n\geq n). dist (X n) L < r \implies X -----> (L::'a::metric-space)
by (simp add: LIMSEQ-def)

lemma metric-LIMSEQ-D:
  [X -----> (L::'a::metric-space); 0 < r] \implies \exists n. \forall n\geq n. dist (X n) L < r
by (simp add: LIMSEQ-def)
100.13.2 Limits of Functions

lemma LIM-def: \( f 
\rightarrow \) (\( a :: 'a :: \text{metric-space} \)) 
\rightarrow \) (\( L :: 'b :: \text{metric-space} \)) = 
\( (\forall x > 0 \, \exists s > 0 \, \forall x \cdot x \neq a \land \text{dist } x \cdot a < s \) 
\rightarrow \) \( \text{dist } (f \cdot x) \cdot L < r \)

unfolding tendsto-iff eventually-at by simp

lemma metric-LIM-I:
\( (\forall r > 0 \, \exists s > 0 \, \forall x \cdot x \neq a \land \text{dist } x \cdot a < s \) 
\rightarrow \) \( \text{dist } (f \cdot x) \cdot L < r \)

by (simp add: LIM-def)

lemma metric-LIM-D:
\[ f \rightarrow (a :: 'a :: \text{metric-space}) 
\rightarrow \) (\( L :: 'b :: \text{metric-space} \); \( 0 < r \)) 
\rightarrow \exists \cdot s > 0 \cdot \forall x \cdot x \neq a \land \text{dist } x \cdot a < s \) 
\rightarrow \) \( \text{dist } (f \cdot x) \cdot L < r \)

by (simp add: LIM-def)

lemma metric-LIM-imp-LIM:
assumes f: \( f 
\rightarrow \) a \rightarrow (l :: 'b :: \text{metric-space})
assumes le: \( \forall x \cdot x \neq a 
\rightarrow \) \( \text{dist } (g \cdot x) \cdot m \leq \text{dist } (f \cdot x) \cdot l \)
shows g: \( g 
\rightarrow \) a \rightarrow (m :: 'b :: \text{metric-space})

by (rule metric-tendsto-imp-tendsto [OF f]) (auto simp add: eventually-at-topological le)

lemma metric-LIM-equal2:
assumes 1: \( 0 < R \)
assumes 2: \( \forall x \cdot [x \neq a \land \text{dist } x \cdot a < R] \) 
\rightarrow \( f \cdot x = g \cdot x \)
shows g: \( g 
\rightarrow \) a \rightarrow l \rightarrow \( f 
\rightarrow \) (a :: 'a :: \text{metric-space}) \rightarrow \) l

apply (rule topological-tendstoI)
apply (drule (2) topological-tendstoD)
apply (simp add: eventually-at, safe)
apply (rule-tac x=min d R in exI, safe)
apply (simp add: 1)
apply (simp add: 2)
done

lemma metric-LIM-compose2:
assumes f: \( f 
\rightarrow \) (a :: 'a :: \text{metric-space}) \rightarrow \) b
assumes g: \( g 
\rightarrow \) b \rightarrow \) c
assumes inj: \( \exists \cdot d > 0 \cdot \forall x \cdot x \neq a \land \text{dist } x \cdot a < d \) 
\rightarrow \( f \cdot x \neq b \)
shows (\( \lambda x \cdot g \cdot (f \cdot x) \)) \rightarrow \) a \rightarrow \) c
using inj

by (intro tendsto-compose-eventually [OF g f]) (auto simp: eventually-at)

lemma metric-isCont-LIM-compose2:
fixes f :: 'a :: metric-space \Rightarrow -
assumes f: unfolded isCont-def: isCont f a
assumes g: \( g 
\rightarrow \) f a \rightarrow \) l
assumes inj: \( \exists \cdot d > 0 \cdot \forall x \cdot x \neq a \land \text{dist } x \cdot a < d \) 
\rightarrow \( f \cdot x \neq f a \)
shows (\( \lambda x \cdot g \cdot (f \cdot x) \)) \rightarrow \) a \rightarrow \) l
by (rule metric-LIM-compose \[2 \ [OF \ f \ inj])

100.14 Complete metric spaces

100.15 Cauchy sequences

definition (in metric-space) Cauchy :: (\nat \Rightarrow 'a) \Rightarrow bool where
Cauchy X = (\\forall e>0. \\exists M. \\forall m \geq M. \\forall n \geq M. \ dist (X m) (X n) < e)

100.16 Cauchy Sequences

lemma metric-CauchyI:
(\\\forall e > 0. \exists M. \\forall m \exists M. \\forall n \exists M. \ dist (X m) (X n) < e) \implies \text{Cauchy X}
by (simp add: Cauchy-def)

lemma metric-CauchyD:
\text{Cauchy X} \implies \exists M. \\forall m \exists M. \\forall n \exists M. \ dist (X m) (X n) < e
by (simp add: Cauchy-def)

lemma metric-Cauchy-iff2:
\text{Cauchy X} = (\\forall j. (\exists M. \\forall m \exists M. \\forall n \exists M. \ dist (X m) (X n) < inverse(real (Suc j))))
apply (simp add: Cauchy-def auto)
apply (drule reals-Archimedean, safe)
apply (drule-tac x = n in spec, auto)
apply (rule-tac x = M in exI, auto)
apply (drule-tac x = m in spec, simp)
apply (drule-tac x = na in spec, auto)
done

lemma Cauchy-iff2:
\text{Cauchy X} = (\\forall j. (\exists M. \\forall m \exists M. \\forall n \exists M. |X m - X n| < inverse(real (Suc j))))
unfolding metric-Cauchy-iff2 dist-real-def ..

lemma Cauchy-subseq-Cauchy:
[ Cauchy X; subseq f ] \implies \text{Cauchy (X o f)}
apply (auto simp add: Cauchy-def)
apply (drule-tac x=e in spec, clarify)
apply (rule-tac x=M in exI, clarify)
apply (blast intro: le-trans [OF - seq-suble] dest!: spec)
done

theorem LIMSEQ-imp-Cauchy:
assumes X: X -----> a shows \text{Cauchy X}
proof (rule metric-Cauchyl)
fix e::real assume \(0 < e\)
hence \(0 < e/2\) by simp
with X have \(\exists N. \forall n \geq N. \ dist (X n) a < e/2\) by (rule metric-LIMSEQ-D)
then obtain N where N: \(\forall n \geq N. \ dist (X n) a < e/2\) ..
show \( \exists N. \forall m \geq N. \forall n \geq N. \text{dist}(X m, X n) < e \)

proof (intro exI allI impI)
  fix \( m \) assume \( N \leq m \)
  hence \( m: \text{dist}(X m, a) < e/2 \) using \( N \) by fast
  fix \( n \) assume \( N \leq n \)
  hence \( n: \text{dist}(X n, a) < e/2 \) using \( N \) by fast
  have \( \text{dist}(X m, X n) \leq \text{dist}(X m, a) + \text{dist}(X n, a) \)
    by (rule dist-triangle2)
  also from \( m, n \) have \( \ldots < e \) by simp
  finally show \( \text{dist}(X m, X n) < e \).
qed

lemma convergent-Cauchy: convergent \( X \implies \text{Cauchy} X \)
unfolding convergent-def
by (erule exE, erule LIMSEQ-imp-Cauchy)

100.16.1 Cauchy Sequences are Convergent

class complete-space = metric-space +
assumes Cauchy-convergent: Cauchy \( X \implies \text{convergent} X \)

lemma Cauchy-convergent-iff:
  fixes \( X :: \text{nat} \Rightarrow 'a::complete-space \)
  shows Cauchy \( X = \text{convergent} X \)
by (fast intro: Cauchy-convergent convergent-Cauchy)

100.17 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html

If sequence \( X \) is Cauchy, then its limit is the lub of \( \{ r. \exists N. \forall n \geq N. r < X n \} \)

lemma increasing-LIMSEQ:
  fixes \( f :: \text{nat} \Rightarrow \text{real} \)
  assumes inc: \( \forall n. f n \leq f \text{ (Suc } n) \)
  and bdd: \( \forall n. f n \leq l \)
  and en: \( \forall e. 0 < e \implies \exists n. l \leq f n + e \)
  shows \( f \longrightarrow l \)
proof (rule increasing-tendsto)
  fix \( x \) assume \( x < l \)
  with dense[of 0 l - x] obtain \( e \) where \( 0 < e \leq l - x \)
    by auto
  from en[OF \( 0 < e \)] obtain \( n \) where \( l - e \leq f n \)
    by (auto simp: field-simps)
  with \( e < l - x \) \( 0 < e \) have \( x < f n \) by simp
  with incseq-SucI[of \( f \), OF inc] show eventually \( (\lambda n. x < f n) \) sequentially
    by (auto simp: eventually-sequentially incseq-def intro: less-le-trans)
**THEORY “Real-Vector-Spaces”**

**lemma** real-Cauchy-convergent:

- **fixes** $X : \text{nat} \Rightarrow \text{real}$
- **assumes** $X : \text{Cauchy} \ X$
- **shows** convergent $X$

**proof**

- **def** $S \equiv \{x : \text{real}. \ \exists N. \ \forall n \geq N. \ x < X \ n\}$
- **then have** mem-$S$: $\forall N x. \ \forall n \geq N. \ x < X \ n \Rightarrow x \in S$ **by** auto

{ **fix** $N \ x$ **assume** $N: \forall n \geq N. \ X \ n < x$

**fix** $y : \text{real}$ **assume** $y \in S$

**hence** $\exists M. \ \forall n \geq M. \ y < X \ n$ **by** (simp add: $S$-def)

**then obtain** $M$ **where** $\forall n \geq M. \ y < X \ n$ **by** simp

**also have** $\ldots < x$ **using** $N$ **by** simp

**finally have** $y \leq x$ **by** (rule order-less-imp-le)

**note** bound-isUb$= \text{this}$

**obtain** $N$ **where** $\forall m \geq N. \ \forall n \geq N. \ dist (X \ m) (X \ n) < 1$
- **using** $X$[THEN metric-CauchyD, OF zero-less-one] **by** auto

**hence** $N: \forall n \geq N. \ dist (X \ n) (X \ N) < 1$ **by** simp

**have** [simp]: $S \neq \{\}$

**proof** (intro exI ex-in-conv[THEN iffD1])

**from** $N$ **have** $\forall n \geq N. \ X \ n - 1 < X \ n$

**by** (simp add: abs-diff-less-iff dist-real-def)

**thus** $X \ n - 1 \in S$ **by** (rule mem-$S$)

**qed**

**have** [simp]: bdd-above $S$

**proof**
- **from** $N$ **have** $\forall n \geq N. \ X \ n < X \ N + 1$
  **by** (simp add: abs-diff-less-iff dist-real-def)

**thus** $\forall s. \ s \in S \Rightarrow s \leq X \ N + 1$
  **by** (rule bound-isUb)$\Box$

**qed**

**have** $X \longrightarrow Sup \ S$

**proof** (rule metric-LIMSEQ-I)

**fix** $r : \text{real}$ **assume** $0 < r$

**hence** $r: 0 < r/2$ **by** simp

**obtain** $N$ **where** $\forall n \geq N. \ \forall m \geq N. \ dist (X \ n) (X \ m) < r/2$
- **using** metric-CauchyD [OF $X \ r$] **by** auto

**hence** $\forall n \geq N. \ dist (X \ n) (X \ N) < r/2$ **by** simp

**hence** $N: \forall n \geq N. \ X \ n - r/2 < X \ n \W X \ n < X \ N + r/2$
  **by** (simp only: dist-real-def abs-diff-less-iff)

**from** $N$ **have** $\forall n \geq N. \ X \ n - r/2 < X \ n$ **by** fast

**hence** $X \ n - r/2 \in S$ **by** (rule mem-$S$)

**qed**
hence 1: \( X N - r/2 \leq \text{Sup } S \) by (simp add: cSup-upper)

from \( N \) have \( \forall n \geq N. X n < X N + r/2 \) by fast
from bound-isUb[of this]
have 2: \( \text{Sup } S \leq X N + r/2 \)
  by (intro cSup-least) simp-all

show \( \exists N. \forall n \geq N. \text{dist} (X n) (\text{Sup } S) < r \)
proof (intro exI allI impI)
  fix \( n \)
  assume \( n: N \leq n \)
  from \( N n \) have \( X n < X N + r/2 \) and \( X N - r/2 < X n \) by simp+
  thus \( \text{dist} (X n) (\text{Sup } S) < r \) using 1 2
  by (simp add: abs-diff-less-iff dist-real-def)
  qed
  qed
then show \( ?\text{thesis} \) unfolding convergent-def by auto
  qed

instance real :: complete-space
  by intro-classes (rule real-Cauchy-convergent)

class banach = real-normed-vector + complete-space

instance real :: banach by default

lemma tendsto-at-topI-sequentially:
  fixes \( f :: \text{real} \Rightarrow b :: \text{first-countable-topology} \)
  assumes \( \forall X. \text{filterlim } X \text{ at-top sequentially } \Longrightarrow (\lambda n. f (X n)) \longrightarrow y \)
  shows \( (f \longrightarrow y) \text{ at-top} \)
proof –
  from nhds-countable[of y] guess \( A \). note \( A = \text{this} \)
  have \( \forall m. \exists k. \forall x \geq k. f x \in A m \)
    proof (rule ccontr)
      assume \( \neg (\forall m. \exists k. \forall x \geq k. f x \in A m) \)
      then obtain \( m \) where \( \forall k. \exists x \geq k. f x \notin A m \)
        by auto
      then have \( \exists X. \forall n. (f (X n) \notin A m) \land \forall n. (X n) + 1 \leq X (\text{Suc } n) \)
        by (intro dependent-nat-choice) (auto simp del: max.bounded iff)
      then obtain \( X \) where \( X: \forall n. f (X n) \notin A m \land \forall n. (X n) + 1 \leq X (\text{Suc } n) \)
        by auto
      { fix \( n \) have \( 1 \leq n \longrightarrow \text{real } n \leq X n \)
        using \( X[\text{of } n - 1] \) by auto }
      then have \( \text{filterlim } X \text{ at-top sequentially} \)
        by (force intro!: filterlim-at-top-mono[of filterlim-real-sequentially] simp: eventually-sequentially)
    from topological-tendstoD[of *[OF this] A(2, 3), of m] X(1) show False
      by auto
qed
then obtain \( k \) where \( \forall m \geq k \, m \leq x \implies f \in A \)
by metis
then show ?thesis
unfolding at-top-def A
by (intro filterlim-base [where \( i = k \)]) auto
qed

lemma tendsto-at-topI-sequentially-real:
fixes \( f \) :: real \Rightarrow real
assumes mono: mono \( f \)
assumes limseq: \( (\lambda n. f(\text{real} n)) \dashrightarrow y \)
shows \( (f \dashrightarrow y) \) at-top
proof (rule tendstoI)
fix \( e \) :: real
assume \( 0 < e \)
with limseq obtain \( N :: \text{nat} \) where \( \forall n \leq N \implies |f(\text{real} n) - y| < e \)
by (auto simp: LIMSEQ-def dist-real-def)
{ fix \( x \) :: real
obtain \( n \) where \( x \leq \text{real-of-nat} n \)
using ex-le-of-nat [of \( x \)] ..
note monoD[OF mono this]
also have \( f(\text{real-of-nat} n) \leq y \)
by (rule LIMSEQ-Le-Const[OF limseq])
(auto intro: exI[of - n] monoD[OF mono] simp: real-eq-of-nat[symmetric])
finally have \( f x \leq y \).
}
note \( le = \) this
have eventually \( (\lambda x. \text{real} N \leq x) \) at-top
by (rule eventually-ge-at-top)
then show eventually \( (\lambda x. \text{dist}(f x, y) < e) \) at-top
proof eventually-elim
fix \( x \) assume \( N' :: \text{real} N \leq x \)
with \( N'[\text{of} N] \) le have \( y - f(\text{real} N) < e \) by auto
moreover note monoD[OF mono N']
ultimately show \( \text{dist}(f x, y) < e \)
using le[\text{of} x] by (auto simp: dist-real-def field-simps)
qed
qed

end

101 Limits: Limits on Real Vector Spaces

theory Limits
imports Real-Vector-Spaces
begin

101.1 Filter going to infinity norm

definition at-infinity :: 'a::real-normed-vector filter where
at-infinity = (INF r. principal {x. r ≤ norm x})

lemma eventually-at-infinity: eventually P at-infinity ⟷ (∃ b. ∀ x. b ≤ norm x −→ P x)
  unfolding at-infinity-def
  by (subst eventually-INF-base)
    (auto simp: subset-eq eventually-principal intro: exI[of - max a b for a b])

lemma at-infinity-eq-at-top-bot:
  (at-infinity :: real filter) = sup at-top at-bot
  apply (simp add: filter-eq-iff eventually-sup eventually-at-infinity
    eventually-at-top-linorder eventually-at-bot-linorder)
  apply safe
  apply (rule-tac x = b in exI, simp)
  apply (rule-tac x = - b in exI, simp)
  apply (rule-tac x = max (- Na) N in exI, auto simp: abs-real-def)
  done

lemma at-top-le-at-infinity: at-top ≤ (at-infinity :: real filter)
  unfolding at-infinity-eq-at-top-bot by simp

lemma at-bot-le-at-infinity: at-bot ≤ (at-infinity :: real filter)
  unfolding at-infinity-eq-at-top-bot by simp

lemma filterlim-at-top-imp-at-infinity:
  fixes f :: 'a ⇒ real
  shows filterlim f at-top F ⟷ filterlim f at-infinity F
  by (rule filterlim-mono[OF - at-top-le-at-infinity order-refl])

101.1.1 Boundedness

definition Bfun :: ('a ⇒ 'b::metric-space) ⇒ 'a filter ⇒ bool where
  Bfun-metric-def: Bfun f F = (∃ y. ∃ K>0. eventually (λx. dist (f x) y ≤ K) F)

abbreviation Bseq :: (nat ⇒ 'a::metric-space) ⇒ bool where
  Bseq X ≡ Bfun X sequentially

lemma Bseq-conv-Bfun: Bseq X ⟷ Bfun X sequentially ..

lemma Bseq-ignore-initial-segment: Bseq X ⟷ Bseq (λn. X (n + k))
  unfolding Bfun-metric-def by (subst eventually-sequentially-seg)

lemma Bseq-offset: Bseq (λn. X (n + k)) ⟷ Bseq X
  unfolding Bfun-metric-def by (subst (asm) eventually-sequentially-seg)

lemma Bfun-def:
  Bfun f F ⟷ (∃ K>0. eventually (λx. norm (f x) ≤ K) F)
  unfolding Bfun-metric-def norm-cone-dist
  proof safe
fix $y K$ assume $0 < K$ and $\star$: eventually $(\lambda x. \text{dist}(f x) y \leq K) F$
moreover have eventually $(\lambda x. \text{dist}(f x) 0 \leq \text{dist}(f x) y + \text{dist} 0 y) F$
  by (intro always-eventually) (metis dist-commute dist-triangle)
with $\star$ have eventually $(\lambda x. \text{dist}(f x) 0 \leq K + \text{dist} 0 y) F$
  by eventually-elim auto
with $\langle 0 < K \rangle$ show $\exists K > 0. \text{eventually} (\lambda x. \text{dist}(f x) 0 \leq K + \text{dist} 0 y) F$
  by (intro exI[of - K + \text{dist} 0 y] add-pos-nonneg conjI zero-le-dist) auto
qed auto

lemma BfunI:
  assumes $K$: eventually $(\lambda x. \text{norm}(f x) \leq K) F$ shows $\text{Bfun} f F$
unfolding Bfun-def
proof (intro exI conjI allI)
  show $0 < \max K 1$ by simp
next
  show eventually $(\lambda x. \text{norm}(f x) \leq \max K 1) F$
    using $K$ by (rule eventually-elim1, simp)
qed

lemma BfunE:
  assumes $\text{Bfun} f F$
  obtains $B$ where $0 < B$ and eventually $(\lambda x. \text{norm}(f x) \leq B) F$
using assms unfolding Bfun-def by fast

lemma Cauchy-Bseq: $\text{Cauchy} X \Longrightarrow \text{Bseq} X$
unfolding Cauchy-def Bfun-metric-def eventually-sequentially
apply (erule tac $x=1$ in allE)
apply simp
apply safe
apply (rule-tac $x=X M$ in exI)
apply (rule-tac $x=1$ in exI)
apply (erule-tac $x=M$ in allE)
apply simp
apply (rule-tac $x=M$ in exI)
apply (auto simp: dist-commute)
done

101.1.2 Bounded Sequences

lemma BseqI': $(\forall n. \text{norm}(X n) \leq K) \Longrightarrow \text{Bseq} X$
by (intro BfunI) (auto simp: eventually-sequentially)

lemma BseqI2': $\forall n \geq N. \text{norm}(X n) \leq K \Longrightarrow \text{Bseq} X$
by (intro BfunI) (auto simp: eventually-sequentially)

lemma Bseq-def: $\text{Bseq} X \iff (\exists K > 0. \forall n. \text{norm}(X n) \leq K)$
unfolding Bfun-def eventually-sequentially
proof safe
  fix $N K$ assume $0 < K \forall n \geq N. \text{norm}(X n) \leq K$
then show \( \exists K > 0. \forall n. \text{norm} \ (X \ n) \leq K \)
by (intro exI[of - max (Max (norm ' X ' {..N})) K] max.strict-coboundedI2)
(auto intro!: imageI not-less[where 'a=nat, THEN iffD1] Max-ge simp: le-max iff-diag)
qed auto

lemma BseqE: \( \exists K. \forall n. \text{norm} \ (X \ n) \leq K \) \implies Q 
unfolding Bseq-def by auto

lemma BseqD: Bseq X \implies \exists K. \forall n. \text{norm} \ (X \ n) \leq K
by (simp add: Bseq-def)

lemma BseqI: \( \mid 0 < K; \forall n. \text{norm} \ (X \ n) \leq K \mid \) \implies Bseq X
by (auto simp add: Bseq-def)

lemma Bseq-bdd-above: Bseq (X::nat \Rightarrow real) \implies \bdd-above (range X)
proof (elim BseqE, intro bdd-aboveI2)
fix K n assume \( 0 < K \) \& \( \forall n. \text{norm} \ (X \ n) \leq K \) then show \( X \ n \leq K \)
by (auto elim!: allE[of - n])
qed

lemma Bseq-bdd-below: Bseq (X::nat \Rightarrow real) \implies \bdd-below (range X)
proof (elim BseqE, intro bdd-belowI2)
fix K n assume \( 0 < K \) \& \( \forall n. \text{norm} \ (X \ n) \leq K \) then show \( -K \leq X \ n \)
by (auto elim!: allE[of - n])
qed

lemma lemma-NBseq-def:
\( (\exists K > 0. \forall n. \text{norm} \ (X \ n) \leq K) = (\exists N. \forall n. \text{norm} \ (X \ n) \leq \text{real}(\text{Suc} \ N)) \)
proof safe
fix K :: real
from reals-Archimedean2 obtain n :: nat where K < real n ..
then have K \leq real (Suc n) by auto
moreover assume \( \forall m. \text{norm} \ (X \ m) \leq K \)
ultimately have \( \forall m. \text{norm} \ (X \ m) \leq \text{real} (\text{Suc} \ n) \)
by (blast intro: order-trans)
then show \( \exists N. \forall n. \text{norm} \ (X \ n) \leq \text{real} (\text{Suc} \ N) \) ..
qed (force simp add: real-of-nat-Suc)

alternative definition for Bseq

lemma Bseq-iff: Bseq X = \( (\exists N. \forall n. \text{norm} \ (X \ n) \leq \text{real} (\text{Suc} \ N)) \)
apply (simp add: Bseq-def)
apply (simp (no_asm) add: lemma-NBseq-def)
done

lemma lemma-NBseq-def2:
\( (\exists K > 0. \forall n. \text{norm} \ (X \ n) \leq K) = (\exists N. \forall n. \text{norm} \ (X \ n) < \text{real} (\text{Suc} \ N)) \)
apply (subst lemma-NBseq-def, auto)
apply (rule_tac x = Suc N in exI)
apply (rule-tac [2] x = N in exI)
apply (auto simp add: real-of-nat-Suc)
prefer 2 apply (blast intro: order-less-imp-le)
apply (drule-tac x = n in spec, simp)
done

lemma Bseq-iff1a: Bseq X = (∃ N. ∀ n. norm (X n) < real(Suc N))
by (simp add: Bseq-def lemma-NBseq-def2)

101.1.3 A Few More Equivalence Theorems for Boundedness

alternative formulation for boundedness

lemma Bseq-iff2: Bseq X = (∃ k > 0. ∃ x. ∀ n. norm (X n) + -x) ≤ k)
apply (unfold Bseq-def, safe)
apply (rule-tac [2] x = k + norm x in exI)
apply (rule-tac x = K in exI, simp)
apply (rule exI [where x = 0], auto)
apply (erule order-less-le-trans, simp)
apply (drule-tac x=n in spec)
apply (drule order-trans [OF norm-triangle-ineq4])
apply simp
done

alternative formulation for boundedness

lemma Bseq-iff3:
Bseq X ←→ (∃ k>0. ∃ N. ∀ n. norm (X n + - X N) ≤ k) (is ?P ←→ ?Q)
proof
assume ?P then obtain K
  where *: 0 < K and **: A n. norm (X n) ≤ K by (auto simp add: Bseq-def)
from * have 0 < K + norm (X 0) by (rule order-less-le-trans) simp
from ** have ∀ n. norm (X n - X 0) ≤ K + norm (X 0)
  by (auto intro: order-trans norm-triangle-ineq4)
then have ∀ n. norm (X n + - X 0) ≤ K + norm (X 0)
  by simp
with ⟨0 < K + norm (X 0)› show ?Q by blast
next
assume ?Q then show ?P by (auto simp add: Bseq-iff2)
qed

lemma BseqI2: (∀ n. k ≤ f n & f n ≤ (K::real)) ==> Bseq f
apply (simp add: Bseq-def)
apply (rule-tac x = (|k| + |K|) + 1 in exI, auto)
apply (drule-tac x = n in spec, arith)
done
101.1.4 Upper Bounds and Lubs of Bounded Sequences

**lemma** Bseq-minus-iff: \(\text{Bseq } (\% n. -(X n) :: 'a :: \text{real-normed-vector}) = \text{Bseq } X\)

by (simp add: Bseq-def)

**lemma** Bseq-eq-bounded: \(\text{range } f \subseteq \{a .. b::real\} \Longrightarrow \text{Bseq } f\)

apply (simp add: subset-eq)
apply (rule BseqI [where \(K=\max (\text{norm } a) (\text{norm } b)\)])
apply (erule_tac x=n in allE)
apply auto
done

**lemma** incseq-bounded: \(\text{incseq } X \Longrightarrow \forall i. X i \leq (B::real) \Longrightarrow \text{Bseq } X\)

by (intro Bseq-eq-bounded [of X X 0 B]) (auto simp: incseq-def)

**lemma** decseq-bounded: \(\text{decseq } X \Longrightarrow \forall i. (B::real) \leq X i \Longrightarrow \text{Bseq } X\)

by (intro Bseq-eq-bounded [of X B X 0]) (auto simp: decseq-def)

101.2 Bounded Monotonic Sequences

101.2.1 A Bounded and Monotonic Sequence Converges

**lemma** Bmonoseq-LIMSEQ: \(\forall n. m \leq n \Longrightarrow X n = X m \Longrightarrow \exists L. (X \longrightarrow L)\)

apply (rule_tac x=X m in ext)
apply (rule filterlim-cong [THEN iffD2, OF refl refl - tendsto-const])
unfolding eventually-sequentially
apply blast
done

101.3 Convergence to Zero

definition Zfun :: ('a => 'b::real-normed-vector) => 'a filter => bool
where Zfun f F = (\forall r>0. eventually (\lambda x. \text{norm } (f x) < r) F)

**lemma** ZfunI:
\(\forall r. 0 < r \Longrightarrow \text{eventually } (\lambda x. \text{norm } (f x) < r) F\) \(\Longrightarrow\) Zfun f F

unfolding Zfun-def by simp

**lemma** ZfunD:
\([Zfun f F; 0 < r]\) \(\Longrightarrow\) \(\forall x. \text{norm } (f x) < r\) F \(\Longrightarrow\) Zfun f F

unfolding Zfun-def by simp

**lemma** Zfun-subst:
\(\text{eventually } (\lambda x. f x = g x) F \Longrightarrow Zfun g F \Longrightarrow Zfun f F\)

unfolding Zfun-def by (auto elim!: eventually-rev-mp)

**lemma** Zfun-zero: \(Zfun (\lambda x. 0) F\)

unfolding Zfun-def by simp
lemma Zfun-norm-iff: Zfun (λx. norm (f x)) F = Zfun (λx. f x) F
unfolding Zfun-def by simp

lemma Zfun-imp-Zfun:
assumes f: Zfun f F
assumes g: eventually (λx. norm (g x) ≤ norm (f x) * K) F
shows Zfun (λx. g x) F
proof (cases)
assume K: 0 < K
show ?thesis
proof (rule ZfunI)
fix r::real
assume 0 < r
hence 0 < r / K using K by simp
then have eventually (λx. norm (f x) < r / K) F
using ZfunD [OF f] by fast
with g show eventually (λx. norm (g x) < r) F
proof eventually-elim
  case (elim x)
  hence norm (f x) * K < r
  by (simp add: pos-less-divide-eq K)
  thus ?case
  by (simp add: order-le-less-trans [OF elim (1)])
qed
qed
next
assume ¬ 0 < K
hence K: K ≤ 0 by (simp only: not-less)
show ?thesis
proof (rule ZfunI)
fix r :: real
assume 0 < r
from g show eventually (λx. norm (g x) < r) F
proof eventually-elim
  case (elim x)
  also have norm (f x) * K ≤ norm (f x) * 0
  using K norm-ge-zero by (rule mult-left-mono)
  finally show ?case
  using ⟨0 < r⟩ by simp
qed
qed
qed

lemma Zfun-le: [Zfun g F; ∀ x. norm (f x) ≤ norm (g x)] =⇒ Zfun f F
by (erule-tac K=1 in Zfun-imp-Zfun, simp)

lemma Zfun-add:
assumes f: Zfun f F and g: Zfun g F
shows Zfun (λx. f x + g x) F
proof (rule ZfunI)
fix $r :: \text{real}$ assume $0 < r$

hence $r : 0 < r / 2$ by simp

have eventually $(\lambda x. \text{norm } (f x) < r / 2) \ F$
  using $f r$ by (rule ZfunD)

moreover

have eventually $(\lambda x. \text{norm } (g x) < r / 2) \ F$
  using $g r$ by (rule ZfunD)

ultimately

show eventually $(\lambda x. \text{norm } (f x + g x) < r) \ F$

proof eventually-elim

case (elim $x$)

have $\text{norm } (f x + g x) \leq \text{norm } (f x) + \text{norm } (g x)$
  by (rule norm-triangle-ineq)

also have $\ldots < r / 2 + r / 2$
  using $\text{elim} by$ (rule add-strict-mono)

finally show ?case
  by simp

qed

lemma Zfun-minus: $\text{Zfun } f \ F \Longrightarrow \text{Zfun } (\lambda x. - f x) \ F$

unfolding Zfun-def by simp

lemma Zfun-diff: \[
\text{[Zfun } f \ F; \text{Zfun } g \ F \text{]} \Longrightarrow \text{Zfun } (\lambda x. f x - g x) \ F
\]

using Zfun-add [of $f \ F \lambda x. - g x$] by (simp add: Zfun-minus)

lemma (in bounded-linear) Zfun:
  assumes $g : \text{Zfun } g \ F$
  shows $\text{Zfun } (\lambda x. f (g x)) \ F$

proof

  obtain $K$ where $\forall x. \text{norm } (f x) \leq \text{norm } x \ast K$
  using bounded by fast

  then have eventually $(\lambda x. \text{norm } (f (g x)) \leq \text{norm } (g x) \ast K) \ F$
    by simp

  with $g$ show ?thesis
    by (rule Zfun-imp-Zfun)

qed

lemma (in bounded-bilinear) Zfun:
  assumes $f : \text{Zfun } f \ F$
  assumes $g : \text{Zfun } g \ F$
  shows $\text{Zfun } (\lambda x. f x ** g x) \ F$

proof (rule ZfunI)

  fix $r :: \text{real}$ assume $r : 0 < r$

  obtain $K$ where $K : 0 < K$
    and norm-le: $\forall x y. \text{norm } (x ** y) \leq \text{norm } x \ast \text{norm } y \ast K$
    using pos-bounded by fast

  from $K$ have $K' : 0 < \text{inverse } K$
    by (rule positive-imp-inverse-positive)
have eventually \((\lambda x. \text{norm}\ (f\ x) < r)\) \(F\)
  using \(f\ r\) by (rule ZfunD)
moreover
have eventually \((\lambda x. \text{norm}\ (g\ x) < \text{inverse}\ K)\) \(F\)
  using \(g\ K\) by (rule ZfunD)
ultimately
show eventually \((\lambda x. \text{norm}\ (f\ **\ g\ x) < r)\) \(F\)
proof eventually-elim
  case (elim \(x\))
  have \(\text{norm}\ (f\ **\ g\ x) \leq \text{norm}\ (f\ x) * \text{norm}\ (g\ x) * K\)
    by (rule norm-le)
  also have \(\text{norm}\ (f\ x) * \text{norm}\ (g\ x) * K < r * \text{inverse}\ K * K\)
    by (intro mult-strict-right-mono mult-strict-mono' norm-ge-zero elim K)
  also from \(K\) have \(r * \text{inverse}\ K * K = r\)
    by simp
  finally show \(?case \).
qed

lemma (in bounded-bilinear) Zfun-left:
  \(\text{Zfun}\ f\ F \implies \text{Zfun}\ (\lambda x. f\ x ** a)\ F\)
  by (rule bounded-linear-left [THEN bounded-linear.Zfun])

lemma (in bounded-bilinear) Zfun-right:
  \(\text{Zfun}\ f\ F \implies \text{Zfun}\ (\lambda x. a \ **\ f\ x)\ F\)
  by (rule bounded-linear-right [THEN bounded-linear.Zfun])

lemmas Zfun-mult-right = bounded-bilinear.Zfun-right [OF bounded-bilinear-mult]
lemmas Zfun-mult-left = bounded-bilinear.Zfun-left [OF bounded-bilinear-mult]

lemma tendsto-Zfun-iff: \((f \longrightarrow a)\ F \iff \text{Zfun}\ (\lambda x. f\ x - a)\ F\)
  by (simp only: tendsto-iff Zfun-def dist-norm)

lemma tendsto-0-le: \([f \longrightarrow 0]\ F; \text{eventually}\ (\lambda x. \text{norm}\ (g\ x) \leq \text{norm}\ (f\ x) * K)\ F\]
  \(\implies (g \longrightarrow 0)\ F\)
  by (simp add: Zfun-imp-Zfun tendsto-Zfun-iff)

101.3.1 Distance and norms

lemma tendsto-dist [tendsto-intros]:
  fixes \(l\ m\ ::\ a\ ::\ \text{metric-space}\)
  assumes \(f\ :\ (f \longrightarrow l)\ F\ \text{and}\ g\ :\ (g \longrightarrow m)\ F\)
  shows \(((\lambda x. \text{dist}\ (f\ x)\ (g\ x)) \longrightarrow \text{dist}\ l\ m)\ F\)
proof (rule tendstoD)
  fix \(e::\ \text{real}\) assume \(0 < e\)
  hence \(e2::\ 0 < e/2\) by simp
  from tendstoD \([OF\ f\ e2]\) tendstoD \([OF\ g\ e2]\)
show eventually \((\lambda x. \text{dist} (\text{dist} f x) (g x)) (\text{dist} l m) < e)\) \(F\)

proof (eventually-elim)

case \((\text{elim} \, x)\)
then show \(\text{dist} (\text{dist} f x) (g x) (\text{dist} l m) < e\)
  unfolding dist-real-def
  using dist-triangle2 \([\text{of} \ f \ x \ g \ x \ l]\)
  using dist-triangle2 \([\text{of} \ g \ x \ l \ m]\)
  using dist-triangle3 \([\text{of} \ l \ m \ f \ x]\)
  using dist-triangle \([\text{of} \ f \ x \ m \ g \ x]\)
  by arith
qed

lemma \text{continuous-dist[continuous-intros]}:
  fixes \(f \, g :: \text{-} \Rightarrow \, 'a :: \text{metric-space}\)
  shows \(\text{continuous} \, F \, f \Rightarrow \text{continuous} \, F \, g \Rightarrow \text{continuous} \, F \, (\lambda x. \text{dist} \, (f \, x) \, (g \, x))\)
  unfolding continuous-def by (rule tendsto-dist)

lemma \text{continuous-on-dist[continuous-intros]}:
  fixes \(f \, g :: \text{-} \Rightarrow \, 'a :: \text{metric-space}\)
  shows \(\text{continuous-on} \, s \, f \Rightarrow \text{continuous-on} \, s \, g \Rightarrow \text{continuous-on} \, s \, (\lambda x. \text{dist} \, (f \, x) \, (g \, x))\)
  unfolding continuous-on-def by (auto intro: tendsto-dist)

lemma \text{tendsto-norm [tendsto-intros]}:
  \((f \, \longrightarrow \, a) \, F \Rightarrow ((\lambda x. \text{norm} \, (f \, x)) \, \longrightarrow \, \text{norm} \, a) \, F\)
  unfolding norm-conv-dist by (intro tendsto-intros)

lemma \text{continuous-norm [continuous-intros]}:
  \(\text{continuous} \, F \, f \Rightarrow \text{continuous} \, F \, (\lambda x. \text{norm} \, (f \, x))\)
  unfolding continuous-def by (rule tendsto-norm)

lemma \text{continuous-on-norm [continuous-intros]}:
  \(\text{continuous-on} \, s \, f \Rightarrow \text{continuous-on} \, s \, (\lambda x. \text{norm} \, (f \, x))\)
  unfolding continuous-on-def by (auto intro: tendsto-norm)

lemma \text{tendsto-norm-zero}:
  \((f \, \longrightarrow \, 0) \, F \Rightarrow ((\lambda x. \text{norm} \, (f \, x)) \, \longrightarrow \, 0) \, F\)
  by (drule tendsto-norm, simp)

lemma \text{tendsto-norm-zero-cancel}:
  \(((\lambda x. \text{norm} \, (f \, x)) \, \longrightarrow \, 0) \, F \Rightarrow (f \, \longrightarrow \, 0) \, F\)
  unfolding tendsto-iff dist-norm by simp

lemma \text{tendsto-norm-zero-iff}:
  \(((\lambda x. \text{norm} \, (f \, x)) \, \longrightarrow \, 0) \, F \iff (f \, \longrightarrow \, 0) \, F\)
  unfolding tendsto-iff dist-norm by simp
lemma tendsto-rabs [tendsto-intros]:
(f −→ (l::real)) F ⟹ ((λx. |f x|) −→ |l|) F
by (fold real-norm-def, rule tendsto-norm)

lemma continuous-rabs [continuous-intros]:
continuous F f ⟹ continuous F (λx. |f x :: real|)
unfolding real-norm-def[symmetric] by (rule continuous-norm)

lemma continuous-on-rabs [continuous-intros]:
continuous-on s f ⟹ continuous-on s (λx. |f x :: real|)
unfolding real-norm-def[symmetric] by (rule continuous-on-norm)

lemma tendsto-rabs-zero:
(f −→ (0::real)) F ⟹ ((λx. |f x|) −→ 0) F
by (fold real-norm-def, rule tendsto-norm-zero)

lemma tendsto-rabs-zero-cancel:
((λx. |f x|) −→ (0::real)) F ⟹ (f −→ 0) F
by (fold real-norm-def, rule tendsto-norm-zero-cancel)

lemma tendsto-rabs-zero-iff:
((λx. |f x|) −→ (0::real)) F ⟷ (f −→ 0) F
by (fold real-norm-def, rule tendsto-norm-zero-iff)

101.3.2 Addition and subtraction

lemma tendsto-add [tendsto-intros]:
fixes a b :: 'a::real-normed-vector
shows [(f −→ a) F; (g −→ b) F] ⟹ ((λx. f x + g x) −→ a + b) F
by (simp only: tendsto-Zfun-iff add-diff-add Zfun-add)

lemma continuous-add [continuous-intros]:
fixes f g :: 'a::t2-space ⇒ 'b::real-normed-vector
shows continuous F f ⟹ continuous F g ⟹ continuous F (λx. f x + g x)
unfolding continuous-def by (rule tendsto-add)

lemma continuous-on-add [continuous-intros]:
fixes f g :: - ⇒ 'b::real-normed-vector
shows continuous-on s f ⟹ continuous-on s g ⟹ continuous-on s (λx. f x + g x)
unfolding continuous-on-def by (auto intro: tendsto-add)

lemma tendsto-add-zero:
fixes f g :: - ⇒ 'b::real-normed-vector
shows [(f −→ 0) F; (g −→ 0) F] ⟹ ((λx. f x + g x) −→ 0) F
by (drule (1) tendsto-add, simp)

lemma tendsto-minus [tendsto-intros]:
fixes a :: 'a::real-normed-vector
- THEORY "Limits"
  
  shows \( (f \longrightarrow a) \quad F \Rightarrow \quad ((\lambda x. - f x) \longrightarrow - a) \quad F \)
  
  by (simp only: tendsto-Zfun-iff minus-diff-minus Zfun-minus)

  lemma continuous-minus [continuous-intros]:
  fixes \( f :: \quad 'a::t2-space \Rightarrow \quad 'b::real-normed-vector \)
  shows continuous \( F \quad f = \quad \Rightarrow \quad continuous \quad F \quad (\lambda x. - f x) \)
  unfolding continuous-def by (rule tendsto-minus)

  lemma continuous-on-minus [continuous-intros]:
  fixes \( f :: \quad '\cdot \Rightarrow \quad 'b::real-normed-vector \)
  shows continuous-on \( s \quad f = \quad \Rightarrow \quad continuous-on \quad s \quad (\lambda x. - f x) \)
  unfolding continuous-on-def by (auto intro: tendsto-minus)

  lemma tendsto-minus-cancel:
  fixes \( a :: \quad 'a::real-normed-vector \)
  shows \((\lambda x. - f x) \longrightarrow - a) \quad F = \quad \Rightarrow \quad (f \longrightarrow a) \quad F \)
  by (drule tendsto-minus, simp)

  lemma tendsto-minus-cancel-left:
  \((f \longrightarrow - (y::real-normed-vector)) \quad F \quad \leftarrow\rightarrow \quad ((\lambda x. - f x) \longrightarrow y) \quad F \)
  using tendsto-minus-cancel[of \( f - y \) \( F \)] tendsto-minus[of \( f - y \) \( F \)]
  by auto

  lemma tendsto-diff [tendsto-intros]:
  fixes \( a \quad b :: \quad 'a::real-normed-vector \)
  shows \([\quad f \quad \longrightarrow a \quad F; \quad (g \quad \longrightarrow b) \quad F \] \quad \Rightarrow \quad ((\lambda x. f x - g x) \quad \longrightarrow a - b) \quad F \)
  using tendsto-add[of \( f \quad a \quad F \quad \lambda x. - g x - b \)] by (simp add: tendsto-minus)

  lemma continuous-diff [continuous-intros]:
  fixes \( f \quad g :: \quad 'a::t2-space \Rightarrow \quad 'b::real-normed-vector \)
  shows continuous \( F \quad f = \quad \Rightarrow \quad continuous \quad F \quad g = \quad \Rightarrow \quad continuous \quad F \quad (\lambda x. f x - g x) \)
  unfolding continuous-def by (rule tendsto-diff)

  lemma continuous-on-diff [continuous-intros]:
  fixes \( f \quad g :: \quad 'a::t2-space \Rightarrow \quad 'b::real-normed-vector \)
  shows continuous-on \( s \quad f = \quad \Rightarrow \quad continuous-on \quad s \quad g = \quad \Rightarrow \quad continuous-on \quad s \quad (\lambda x. f x - g x) \)
  unfolding continuous-on-def by (auto intro: tendsto-diff)

  lemma tendsto-setsum [tendsto-intros]:
  fixes \( f :: \quad 'a \Rightarrow \quad 'c::real-normed-vector \)
  assumes \( \forall i. \quad i \in S \quad \Rightarrow \quad (f \quad i \quad \longrightarrow a \quad i) \quad F \)
  shows \((\lambda x. \sum i \in S. f \quad i \quad x) \quad \longrightarrow \quad (\sum i \in S. a \quad i) \quad F \)
  proof (cases finite \( S \))
    assume finite \( S \) thus \( \text{thesis using \( assms \)} \)
    by (induct, simp add: tendsto-const, simp add: tendsto-add)
  next
    assume \( \neg \quad \text{finite \( S \)} \) thus \( \text{thesis} \)
    by (simp add: tendsto-const)
qed

lemma continuous-setsum [continuous-intros]:
  fixes f :: 'a ⇒ 'b::t2-space ⇒ 'c::real-normed-vector
  shows (⋀i. i ∈ S ⇒ continuous F (f i)) ⇒ continuous F (λx. ∑i∈S. f i x)
  unfolding continuous-def by (rule tendsto-setsum)

lemma continuous-on-setsum [continuous-intros]:
  fixes f :: 'a ⇒ - ⇒ 'c::real-normed-vector
  shows (⋀i. i ∈ S ⇒ continuous-on s (f i)) ⇒ continuous-on s (λx. ∑i∈S. f i x)
  unfolding continuous-on-def by (auto intro: tendsto-setsum)

lemmas real-tendsto-sandwich = tendsto-sandwich [where 'b=real]

101.3.3 Linear operators and multiplication

lemma (in bounded-linear) tendsto:
  (g −−→ a) F ⇒ ((λx. f (g x)) −−→ f a) F
  by (simp only: tendsto-Zfun-iff diff [symmetric] Zfun)

lemma (in bounded-linear) continuous:
  continuous F g ⇒ continuous F (λx. f (g x))
  using tendsto[of g - F] by (auto simp: continuous-def)

lemma (in bounded-linear) continuous-on:
  continuous-on s g ⇒ continuous-on s (λx. f (g x))
  using tendsto[of g] by (auto simp: continuous-on-def)

lemma (in bounded-linear) tendsto-zero:
  (g −−→ 0) F ⇒ ((λx. f (g x)) −−→ 0) F
  by (drule tendsto, simp only: zero)

lemma (in bounded-bilinear) tendsto:
  [f −−→ a] F; (g −−→ 0) F] ⇒ ((λx. f x ** g x) −−→ a ** b) F
  by (simp only: tendsto-Zfun-iff prod-diff-prod
    Zfun-add Zfun Zfun-left Zfun-right)

lemma (in bounded-bilinear) continuous:
  continuous F f ⇒ continuous F g ⇒ continuous F (λx. f x ** g x)
  using tendsto[of f - F g] by (auto simp: continuous-def)

lemma (in bounded-bilinear) continuous-on:
  continuous-on s f ⇒ continuous-on s g ⇒ continuous-on s (λx. f x ** g x)
  using tendsto[of f - g] by (auto simp: continuous-on-def)

lemma (in bounded-bilinear) tendsto-zero:
  assumes f: (f −−→ 0) F
  assumes g: (g −−→ 0) F
THEORY “Limits”

shows \((\lambda x. f \circ g x) \rightarrow 0\) \(F\)
using \tendsto [OF \(g\)] by (simp add: zero-left)

lemma (in bounded-bilinear) tendsto-left-zero:
\((f \rightarrow 0) \, F \Longrightarrow ((\lambda x. f \circ \circ c) \rightarrow 0) \, F\)
by (rule bounded-linear.tendsto-zero [OF bounded-linear-left])

lemma (in bounded-bilinear) tendsto-right-zero:
\((f \rightarrow 0) \, F \Longrightarrow ((\lambda x. c \circ f x) \rightarrow 0) \, F\)
by (rule bounded-linear.tendsto-zero [OF bounded-linear-right])

lemmas tendsto-of-real [tendsto-intros] =
bounded-linear.tendsto [OF bounded-linear-of-real]

lemmas tendsto-scaleR [tendsto-intros] =
bounded-bilinear.tendsto [OF bounded-bilinear-scaleR]

lemmas tendsto-mul [tendsto-intros] =
bounded-bilinear.tendsto [OF bounded-bilinear-mul]

lemmas continuous-of-real [continuous-intros] =
bounded-linear.continuous [OF bounded-linear-of-real]

lemmas continuous-scaleR [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-scaleR]

lemmas continuous-mul [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-mul]

lemmas continuous-on-of-real [continuous-intros] =
bounded-linear.continuous-on [OF bounded-linear-of-real]

lemmas continuous-on-scaleR [continuous-intros] =
bounded-linear.continuous-on [OF bounded-bilinear-scaleR]

lemmas continuous-on-mul [continuous-intros] =
bounded-linear.continuous-on [OF bounded-bilinear-mul]

lemmas tendsto-mul-zero =
bounded-bilinear.tendsto-zero [OF bounded-bilinear-mul]

lemmas tendsto-mul-left-zero =
bounded-bilinear.tendsto-left-zero [OF bounded-bilinear-mul]

lemmas tendsto-mul-right-zero =
bounded-bilinear.tendsto-right-zero [OF bounded-bilinear-mul]

lemma tendsto-power [tendsto-intros]:
fixes \(f :: a \Rightarrow b :: \{\text{power, real-normed-algebra}\}\)
shows \((f \to a) F \to ((\lambda x. f x \to n) \to a \to n) F\)
by (induct n) (simp-all add: tendsto-const tendsto-mult)

lemma continuous-power [continuous-intros]:
fixes \(f::'a::t2-space \Rightarrow 'b::{power,real-normed-algebra}\)
shows continuous \(F f \Rightarrow\) continuous \((\lambda x. (f x)^n)\)
unfolding continuous-def by (rule tendsto-power)

lemma continuous-on-power [continuous-intros]:
fixes \(f::'a::t2-space \Rightarrow 'b::{power,real-normed-algebra}\)
shows continuous-on s \(f \Rightarrow\) continuous-on s \((\lambda x. (f x)^n)\)
unfolding continuous-on-def by (auto intro: tendsto-power)

lemma tendsto-setprod [tendsto-intros]:
fixes \(f::'a \Rightarrow 'b::\{real-normed-algebra,comm-ring-1\}\)
assumes \(\forall i. i \in S \Rightarrow (f i \to L i) F\)
shows \((\lambda x. \prod i \in S. f i x) \to (\prod i \in S. L i)) F\)
proof (cases finite S)
  assume finite S thus ?thesis using assms by (induct, simp add: tendsto-const, simp add: tendsto-mult)
next
  assume \(\neg\) finite S thus ?thesis by (simp add: tendsto-const)
qed

lemma continuous-setprod [continuous-intros]:
fixes \(f::'a \Rightarrow 'b::\{real-normed-algebra,comm-ring-1\}\)
shows \((\lambda i. i \in S \Rightarrow \text{continuous } F (f i)) \Rightarrow \text{continuous } F (\lambda x. \prod i \in S. f i x)\)
unfolding continuous-def by (rule tendsto-setprod)

lemma continuous-on-setprod [continuous-intros]:
fixes \(f::'a \Rightarrow 'c::\{real-normed-algebra,comm-ring-1\}\)
shows \((\lambda i. i \in S \Rightarrow \text{continuous-on } s (f i)) \Rightarrow \text{continuous-on } s (\lambda x. \prod i \in S. f i x)\)
unfolding continuous-on-def by (auto intro: tendsto-setprod)

101.3.4 Inverse and division

lemma (in bounded-bilinear) Zfun-prod-Bfun:
assumes \(f::Z\fun f F\)
assumes \(g::\text{Bfun } g F\)
shows \(\text{Zfun } (\lambda x. f x \ast g x) F\)
proof —
  obtain K where K: \(0 \leq K\) and norm-le: \(\forall x y. \text{norm } (x \ast y) \leq \text{norm } x \ast \text{norm } y \ast K\)
  using nonneg-bounded by fast
  obtain B where B: \(0 < B\) and norm-g: eventually \((\lambda x. \text{norm } (g x) \leq B) F\)
  using g by (rule BfunE)
have eventually \((\lambda x. \text{norm} (f x ** g x) \leq \text{norm} (f x) \ast (B \ast K))\) \(F\)

using \(\text{norm-g}\) proof eventually-elim

\[
\begin{align*}
\text{case (elim } x) \\
\quad \text{have } \text{norm} (f x ** g x) \leq \text{norm} (f x) \ast \text{norm} (g x) \ast K \\
& \text{by (rule norm-le)} \\
\quad \text{also have } \ldots \leq \text{norm} (f x) \ast B \ast K \\
& \text{by (intro mult-mono' order-refl norm-g norm-ge-zero} \\
& \text{mult-nonneg-nonneg } K \text{ elim)} \\
\quad \text{also have } \ldots = \text{norm} (f x) \ast (B \ast K) \\
& \text{by (rule mult.assoc)} \\
\text{finally show } \text{norm} (f x ** g x) \leq \text{norm} (f x) \ast (B \ast K) \\
\end{align*}
\]

qed

with \(f\) show \(?\text{thesis}\)

by (rule \(Zfun-imp-Zfun\))

qed

lemma \(\text{in bounded-bilinear}\) \(\text{flip}\):

\[\text{bounded-bilinear } (\lambda x y. y ** x)\]

apply default

apply (rule add-right)

apply (rule add-left)

apply (rule scaleR-right)

apply (rule scaleR-left)

apply (subst mult. commute)

using bounded by fast

lemma \(\text{in bounded-bilinear}\) \(\text{Bfun-prod-Zfun}\):

assumes \(f\): \(\text{Bfun } f\) \(F\)

assumes \(g\): \(\text{Zfun } g\) \(F\)

shows \(\text{Zfun } (\lambda x. f x ** g x)\) \(F\)

using \(\text{flip } g f\) by (rule bounded-bilinear.\(\text{Zfun-prod-Bfun}\))

lemma \(\text{Bfun-inverse-lemma}\):

fixes \(x\) :: \(\text{a::real-normed-div-algebra}\)

shows \([\ r \leq \text{norm } x; \ 0 < r \ ] \implies \text{norm} (\text{inverse } x) \leq \text{inverse } r\)

apply (subst nonzero-norm-inverse, clarsimp)

apply (erule (1) le-imp-inverse-le)

done

lemma \(\text{Bfun-inverse}\):

fixes \(a\) :: \(\text{a::real-normed-div-algebra}\)

assumes \(f\): \((f \rightarrow a)\) \(F\)

assumes \(a\): \(a \neq 0\)

shows \(\text{Bfun } (\lambda x. \text{inverse } (f x))\) \(F\)

proof

from \(a\) have \(0 < \text{norm } a\) by simp

hence \(\exists r>0. \ r < \text{norm } a\) by (rule dense)

then obtain \(r\) where \(r1\): \(0 < r\) and \(r2\): \(r < \text{norm } a\) by fast

have eventually \((\lambda x. \text{dist } (f x) a < r)\) \(F\)
using tendstoD [OF f r1] by fast
hence eventually (λx. norm (inverse (f x)) ≤ inverse (norm a − r)) F
proof eventually-elim
  case (elim x)
  hence 1: norm (f x − a) < r
    by (simp add: dist-norm)
hence 2: f x ≠ 0 using r2 by auto
hence norm (inverse (f x)) = inverse (norm (f x))
  by (rule nonzero-norm-inverse)
also have ... ≤ inverse (norm a − r)
proof (rule le-imp-inverse-le)
  show 0 < norm a − r using r2 by simp
next
  have norm a − norm (f x) ≤ norm (a − f x)
    by (rule norm-triangle-ineq2)
  also have ... = norm (f x − a)
    by (rule norm-minus-commute)
  also have ... < r using 1 .
  finally show norm a − r ≤ norm (f x) by simp
qed
finally show norm (inverse (f x)) ≤ inverse (norm a − r).
qed
thus ?thesis by (rule BfunI)
qed

lemma tendsto-inverse [tendsto-intros]:
  fixes a :: 'a::real-normed-div-algebra
  assumes f: (f −−−> a) F
    assumes a: a ≠ 0
  shows ((λx. inverse (f x)) −−−> inverse a) F
proof —
  from a have 0 < norm a by simp
  with f have eventually (λx. dist (f x) a < norm a) F
    by (rule tendstoD)
  then have eventually (λx. f x ≠ 0) F
    unfolding dist-norm by (auto elim!: eventually-elim)
  unfolding dist-norm by (auto elim!: eventually-elim)
  with a have eventually (λx. inverse (f x) − inverse a =
    − (inverse (f x) * (f x − a) * inverse a)) F
    by (auto elim!: eventually-elim simp: inverse-diff-inverse)
  moreover have Zfun (λx. − (inverse (f x) * (f x − a) * inverse a)) F
    by (intro Zfun-minus Zfun-mult-left
      Bfun-inverse [OF a] f [unfolded tendsto-Zfun-iff])
  ultimately show ?thesis
    unfolding tendsto-Zfun-iff by (rule Zfun-ssubst)
qed

lemma continuous-inverse:
  fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
assumes continuous F f and f (Lim F (λx. x)) ≠ 0
shows continuous F (λx. inverse (f x))
using assms unfolding continuous-def by (rule tendsto-inverse)

lemma continuous-at-within-inverse[continuous-intros]:
fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
assumes continuous (at a within s) f and f a ≠ 0
shows continuous (at a within s) (λx. inverse (f x))
using assms unfolding continuous-within by (rule tendsto-inverse)

lemma isCont-inverse[continuous-intros, simp]:
fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
assumes isCont f a and f a ≠ 0
shows isCont (λx. inverse (f x)) a
using assms unfolding continuous-at by (rule tendsto-inverse)

lemma continuous-on-inverse[continuous-intros]:
fixes f :: 'a::topological-space ⇒ 'b::real-normed-div-algebra
assumes continuous-on s f and ∀x∈s. f x ≠ 0
shows continuous-on s (λx. inverse (f x))
using assms unfolding continuous-on-def by (fast intro: tendsto-inverse)

lemma tendsto-divide [tendsto-intros]:
fixes a b :: 'a::real-normed-field
shows [(f −−→ a) F; (g −−→ b) F; b ≠ 0] 
⟹ ((λx. f x / g x) −−→ a / b) F
by (simp add: tendsto-mul tendsto-inverse divide-inverse)

lemma continuous-divide:
fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
assumes continuous F f and continuous F g and (Lim F (λx. x)) ≠ 0
shows continuous F (λx. f x / (g x))
using assms unfolding continuous-def by (rule tendsto-divide)

lemma continuous-at-within-divide[continuous-intros]:
fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
assumes continuous (at a within s) f continuous (at a within s) g and g a ≠ 0
shows continuous (at a within s) (λx. f x / (g x))
using assms unfolding continuous-within by (rule tendsto-divide)

lemma isCont-divide[continuous-intros, simp]:
fixes f g :: 'a::t2-space ⇒ 'b::real-normed-field
assumes isCont f a isCont g a g a ≠ 0
shows isCont (λx. (f x) / (g x)) a
using assms unfolding continuous-at by (rule tendsto-divide)

lemma continuous-on-divide[continuous-intros]:
fixes f :: 'a::topological-space ⇒ 'b::real-normed-field
assumes continuous-on s f continuous-on s g and ∀x∈s. g x ≠ 0
shows continuous-on s (\(\lambda x. (f x) / (g x)\))
using assms unfolding continuous-on-def by (fast intro: tendsto-divide)

lemma tendsto-sgn [tendsto-intros]:
fixes l :: 'a::real-normed-vector
shows \([f \rightarrow l] F; l \neq 0 \Rightarrow (\lambda x. \text{sgn} (f x)) \rightarrow \text{sgn} l) F\)
unfolding sgn-div-norm by (simp add: tendsto-intros)

lemma continuous-sgn:
fixes f :: 'a::t2-space \Rightarrow 'b::real-normed-vector
assumes continuous F f and f (Lim F (\(\lambda x. x\))) \neq 0
shows continuous F (\(\lambda x. \text{sgn} (f x)\))
using assms unfolding continuous-def by (rule tendsto-sgn)

lemma continuous-at-within-sgn[continuous-intros]:
fixes f :: 'a::t2-space \Rightarrow 'b::real-normed-vector
assumes continuous (at a within s) f and f a \neq 0
shows continuous (at a within s) (\(\lambda x. \text{sgn} (f x)\))
using assms unfolding continuous-within by (rule tendsto-sgn)

lemma isCont-sgn[continuous-intros]:
fixes f :: 'a::t2-space \Rightarrow 'b::real-normed-vector
assumes isCont f a and f a \neq 0
shows isCont (\(\lambda x. \text{sgn} (f x)\)) a
using assms unfolding continuous-at by (rule tendsto-sgn)

lemma continuous-on-sgn[continuous-intros]:
fixes f :: 'a::topological-space \Rightarrow 'b::real-normed-vector
assumes continuous-on s f and \(\forall x \in s. f x \neq 0\)
shows continuous-on s (\(\lambda x. \text{sgn} (f x)\))
using assms unfolding continuous-on-def by (fast intro: tendsto-sgn)

lemma filterlim-at-infinity:
fixes f :: 'a\:\Rightarrow 'b::real-normed-vector
assumes 0 \leq c
shows \((\text{LIM } x \to \text{at-infinity}) F \iff (\forall r > c. \text{eventually} (\forall x \leq \text{norm} (f x)) F)\)
unfolding filterlim-iff eventually-at-infinity
proof safe
fix P :: 'a \Rightarrow bool and b
assume *: \(\forall r > c. \text{eventually} (\lambda x. r \leq \text{norm} (f x)) F\)
and P: \(\forall x. b \leq \text{norm} x \to P x\)
have max b (c + 1) > c by auto
with * have \((\forall x. \text{max} b (c + 1) \leq \text{norm} (f x)) F\)
  by auto
then show \((\lambda x. P (f x)) F\)
proof eventually-elim
fix x assume max b (c + 1) \leq \text{norm} (f x)
with P show \(P (f x)\) by auto
101.4 Relate $at$, $at-left$ and $at-right$

This lemmas are useful for conversion between $at x$ to $at-left x$ and $at-right x$ and also $at-right (0::'a)$.

**lemmas** $filterlim-split-at-real = filterlim-split-at [where 'a=real]$

**lemma** $filtermap-homeomorph$:
- **assumes** $f$: continuous $(\operatorname{at} a) f$
- **assumes** $g$: continuous $(\operatorname{at} (f a)) g$
- **assumes** $bij1$: $\forall x. f (g x) = x$ and $bij2$: $\forall x. g (f x) = x$
- **shows** $filtermap f \ (\operatorname{nhds} a) = \operatorname{nhds} (f a)$

**unfolding** $\text{filter-eq-iff eventually-filtermap eventually-nhds}$

**proof** $safe$
- **fix** $P S$ assume $S$: open $S f a \in S$ and $P$: $\forall x \in S. P x$
- **from** $\text{continuous-within-topological}[\text{THEN iffD1}, \text{rule-format}, OF f S ] P$
- **show** $\exists! S. \text{open S} \land a \in S \land (\forall x \in S. P (f x))$ by $\text{auto}$

**next**
- **fix** $P S$ assume $S$: open $S a \in S$ and $P$: $\forall x \in S. P (f x)$
- **with** $\text{continuous-within-topological}[\text{THEN iffD1}, \text{rule-format}, OF g, of S ] bij2$
- **obtain** $A$ where $open A f a \in A \ (\forall g \in A. g y \in S)$
- **by** $(\text{metis UNIV-I})$
- **with** $P bij1$ show $\exists S. \text{open S} \land f a \in S \land (\forall x \in S. P x)$
- **by** $(\text{force intro!: extI[of - A]})$

**qed**

**lemma** $filtermap-nhds-shift$: $filtermap (\lambda x. x - d) \ (\operatorname{nhds} a) = \operatorname{nhds} (a - d::'a::\text{real-normed-vector})$

**by** $(\text{rule filtermap-homeomorph[where g=\lambda x. x + d']})$ (auto intro: continuous-intros)

**lemma** $filtermap-nhds-minus$: $filtermap (\lambda x. - x) \ (\operatorname{nhds} a) = \operatorname{nhds} (- a::'a::\text{real-normed-vector})$

**by** $(\text{rule filtermap-homeomorph[where g=uminus']})$ (auto intro: continuous-minus)

**lemma** $filtermap-at-shift$: $filtermap (\lambda x. x - d) \ (at a) = (at (a - d::'a::\text{real-normed-vector})$

**by** $(\text{simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift[symmetric]})$

**lemma** $filtermap-at-right-shift$: $filtermap (\lambda x. x - d) \ (at-right a) = (at-right (a - d::real))$

**by** $(\text{simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift[symmetric]})$

**lemma** $at-right-to-0$: $at-right (a::real) = filtermap (\lambda x. x + a) \ (at-right 0)$

**using** $\text{filtermap-at-right-shift[of - a 0] by simp}$

**lemma** $filterlim-at-right-to-0$:
- $filterlim f F \ (at-right (a::real)) \longleftrightarrow filterlim (\lambda x. f \ (x + a)) F \ (at-right 0)$
- **unfolding** $\text{filterlim-def filtermap-filtermap at-right-to-0[of a]} ..$

**lemma** $\text{eventually-at-right-to-0}$:
eventually $P$ (at-right $(a::\text{real})$) $\longleftrightarrow$ eventually $(\lambda x. P (x + a))$ (at-right 0)

unfolding at-right-to-0[of a] by (simp add: eventually-filtermap)

lemma filtermap-at-minus: $\text{filtermap} (\lambda x. - x)$ (at a) = at $(- a::\text{real-normed-vector})$

by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-left-minus: $\text{at-left} (a::\text{real}) = \text{filtermap} (\lambda x. - x)$ (at-right $(- a)$)

by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-right-minus: $\text{at-right} (a::\text{real}) = \text{filtermap} (\lambda x. - x)$ (at-left $(- a)$)

by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma filterlim-at-left-to-right:

$\text{filterlim} f F$ (at-left $(a::\text{real})$) $\longleftrightarrow$ filterlim $(\lambda x. f (- x))$ F (at-right $(- a)$)

unfolding filterlim-def filtermap-filtermap at-left-minus

lemma eventually-at-left-to-right:

eventually $P$ (at-left $(a::\text{real})$) $\longleftrightarrow$ eventually $(\lambda x. P (- x))$ (at-right $(- a)$)

unfolding at-left-minus[of a] by (simp add: eventually-filtermap)

lemma at-top-mirror: $\text{at-top} = \text{filtermap} \text{uminus} (\text{at-bot :: real filter})$

unfolding filter-eq-iff eventually-filtermap eventually-at-top-linorder eventually-at-bot-linorder

by (metis leI minus-less-iff order-less-asym)

lemma at-bot-mirror: $\text{at-bot} = \text{filtermap} \text{uminus} (\text{at-top :: real filter})$

unfolding at-top-mirror filtermap-filtermap

by (simp add: filtermap-ident)

lemma filterlim-at-top-mirror: $(\text{LIM} x \text{ at-top}. f x :> F) \longleftrightarrow (\text{LIM} x \text{ at-bot}. f (-x::\text{real}) :> F)$

unfolding filterlim-def at-top-mirror filtermap-filtermap

lemma filterlim-at-bot-mirror: $(\text{LIM} x \text{ at-bot}. f x :> F) \longleftrightarrow (\text{LIM} x \text{ at-top}. f (-x::\text{real}) :> F)$

unfolding filterlim-def at-bot-mirror filtermap-filtermap

lemma filterlim-uminus-at-top-at-bot: $\text{LIM} x \text{ at-bot}. - x :: real :> at-top$

by (metis leI minus-less-iff order-less-asym)

lemma filterlim-uminus-at-bot-at-top: $\text{LIM} x \text{ at-top}. - x :: real :> at-bot$

by (metis leI less-minus-iff order-less-asym)

lemma filterlim-uminus-at-top: $(\text{LIM} x \text{ F}. f x :> at-top) \longleftrightarrow (\text{LIM} x \text{ F}. - (f x) :: real :> at-bot)$

using filterlim-compose[OF filterlim-uminus-at-bot-at-top, of f F]

using filterlim-compose[OF filterlim-uminus-at-top-at-bot, of $\lambda x. - f x \text{ F}$]

by auto
lemma filterlim-uminus-at-bot: (\(\text{LIM } F. f x \to \text{at-bot}\)) \(\iff\) (\(\text{LIM } F. -(f x)\))
   unfolding filterlim-uminus-at-top by simp

lemma filterlim-inverse-at-top-right: \(\text{LIM } x \text{ at-right } (0::\text{real}). inverse x \to \text{at-top}\)
   unfolding filterlim-at-top-ge[where c=0] eventually-at-filter
   proof safe
   fix Z :: real assume [arith]: 0 < Z
   then have eventually (\(\lambda x. x < \text{inverse } Z\)) (nhds 0)
      by (auto simp add: eventually-nhds-metric dist-real-def intro: exI[of - |inverse Z])
   then show eventually (\(\lambda x. x \neq 0 \to x \in \{0<..\} \to Z \leq \text{inverse } x\)) (nhds 0)
      by (auto elim!: eventually-elim1 simp: inverse-eq-divide field-simps)
   qed

lemma filterlim-inverse-at-top:
   (\(f \longrightarrow (0 :: \text{real})\)) \(\Rightarrow\) eventually (\(\lambda x. 0 < f x\)) \(\Rightarrow\) \(\text{LIM } x \to F. \text{inverse } f x\)
   unfolding filterlim-uminus-at-bot by (rule filterlim-inverse-at-top)

lemma filterlim-inverse-at-bot:
   (\(f \longrightarrow (0 :: \text{real})\)) \(\Rightarrow\) eventually (\(\lambda x. f x < 0\)) \(\Rightarrow\) \(\text{LIM } x \to F. \text{inverse } f x\)
   by (rule filterlim-inverse-at-top)

lemma tendsto-inverse-0:
   fixes x :: 'a::real-normed-div-algebra
   shows (inverse \longrightarrow (0::'a)) at-infinity
   unfolding tendsto-Zfun-iff diff-0-right Zfun-def eventually-at-infinity
   proof safe
   fix r :: real assume 0 < r
   show \(\exists b. \forall x. b \leq \text{norm } x \to \text{norm } (\text{inverse } x :: 'a) < r\)
   proof (intro exI[of - inverse (r / 2)] allI impl)
     fix x :: 'a
     from 0 < r have 0 < inverse (r / 2) by simp
     also assume *: inverse (r / 2) \leq norm x
     finally show norm (inverse x) < r
     using * 0 < r by (subst nonzero-norm-inverse) (simp-all add: inverse-eq-divide field-simps)
   qed
   qed
lemma at-right-to-top: (at-right (0::real)) = filtermap inverse at-top
proof (rule antisym)
  have (inverse --> (0::real)) at-top
    by (metis tendsto-inverse-0 filterlim-mono at-top-le-at-infinity order-refl)
then show filtermap inverse at-top ≤ at-right (0::real)
  by (simp add: le-principal eventually-filtermap eventually-gt-at-top filterlim-def at-within-def)
next
  have filtermap inverse (filtermap inverse (at-right (0::real))) ≤ filtermap inverse at-top
  using filterlim-inverse-at-top-right unfolding filterlim-def by (rule filtermap-mono)
then show at-right (0::real) ≤ filtermap inverse at-top
  by (simp add: filtermap-ident filtermap-filtermap)
qed

lemma eventually-at-right-to-top:
eventually P (at-right (0::real)) ⟷ eventually (λx. P (inverse x)) at-top
unfolding at-right-to-top eventually-filtermap ..

lemma filterlim-at-right-to-top:
filterlim f F (at-right (0::real)) ⟷ (LIM x at-top. f (inverse x) :> F)
unfolding filterlim-def at-right-to-top filtermap-filtermap ..

lemma at-top-to-right: at-top = filtermap inverse (at-right (0::real))
unfolding at-right-to-top filtermap-filtermap inverse-inverse-eq filtermap-ident ..

lemma eventually-at-top-to-right:
eventually P at-top ⟷ eventually (λx. P (inverse x)) (at-right (0::real))
unfolding at-top-to-right eventually-filtermap ..

lemma filterlim-at-top-to-right:
filterlim f F at-top ⟷ (LIM x (at-right (0::real)). f (inverse x) :> F)
unfolding filterlim-def at-top-to-right filtermap-filtermap ..

lemma filterlim-inverse-at-infinity:
fixes x :: 'a::{real-normed-div-algebra, division-ring-inverse-zero}
shows filterlim inverse at-infinity (at (0::'a))
unfolding filterlim-at-infinity[OF order-refl]
proof safe
  fix r :: real assume 0 < r
then show eventually (λx::'a. r ≤ norm (inverse x)) (at 0)
    unfolding eventually-at norm-inverse
    by (intro exI[of - inverse r])
    (auto simp: norm-conv-dist[symmetric] field-simps inverse-eq-divide)
qed

lemma filterlim-inverse-at-iff:
fixes g :: 'a ⇒ 'b::{real-normed-div-algebra, division-ring-inverse-zero}
shows (LIM x F. inverse (g x) :> at 0) ⟷ (LIM x F. g x :> at-infinity)
unfolding filterlim-def filtermap-filtermap[symmetric]

proof
assume filtermap g F ≤ at-infinity
then have filtermap inverse (filtermap g F) ≤ filtermap inverse at-infinity
  by (rule filtermap-mono)
also have ... ≤ at 0
  using tendsto-inverse-0[where 'a='b]
  by (auto intro: filtermap-inverse-at-infinity)
finally show filtermap inverse (filtermap g F) ≤ at 0.

next
assume filtermap inverse (filtermap g F) ≤ at 0
then have filtermap inverse (filtermap inverse (filtermap g F)) ≤ filtermap inverse at 0
  by (rule filtermap-mono)
with filterlim-inverse-at-infinity show filtermap g F ≤ at-infinity
  by (auto intro: order-trans simp: filterlim-def filtermap-filtermap)

qed

lemma tendsto-inverse-0-at-top: LIM x F. f x := at-top ==> ((λx. inverse (f x) :: real) --→ 0) F
by (metis filterlim-at filterlim-mono[OF - at-top-le-at-infinity order-refl] filterlim-inverse-at-iff)

We only show rules for multiplication and addition when the functions are
either against a real value or against infinity. Further rules are easy to derive
by using filterlim if at-top ?F = (LIM x ?F. --→ at-bot).

lemma filterlim-tendsto-pos-mult-at-top:
  assumes f: (f --→ c) F and c: 0 < c
  assumes g: LIM x F. g x := at-top
  shows LIM x F. (f x * g x :: real) := at-top
unfolding filterlim-at-top-gt[where c=0]
proof safe
  fix Z :: real assume 0 < Z
  from f:0 < c have eventually (λx. c / 2 < f x) F
    by (auto dest!: tendstoD[where e=c / 2] elim!: eventually-elim1
        simp: dist-real-def abs-real-def split: split-if-asm)
  moreover from g have eventually (λx. (Z / c) * 2) ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x * g x) F
proof eventually-elim
  fix x assume c / 2 < f x Z / c * 2 ≤ g x
  with (0 < Z) (0 < c) have c / 2 * (Z / c) ≤ f x * g x
    by (intro mult-mono) (auto simp: zero-le-divide-iff)
  with (0 < c) show Z ≤ f x * g x
    by simp
  qed

qed

lemma filterlim-at-top-mult-at-top:
assumes \( f : \mathrm{LIM} x F. f x \Rightarrow \text{at-top} \)
assumes \( g : \mathrm{LIM} x F. g x \Rightarrow \text{at-top} \)
shows \( \mathrm{LIM} x F. (f x \cdot g x :: \text{real}) \Rightarrow \text{at-top} \)
unfolding filterlim-at-top-gl[\text{where } c=0]
proof safe
  fix \( Z :: \text{real} \) assume \( 0 < Z \)
  from \( f \) have \( \text{eventually} (\lambda x. 1 \leq f x) F \)
    unfolding filterlim-at-top by auto
  moreover from \( g \) have \( \text{eventually} (\lambda x. Z \leq g x) F \)
    unfolding filterlim-at-top by auto
  ultimately show \( \text{eventually} (\lambda x. Z \leq f x \cdot g x) F \)
    proof eventually-elim
      fix \( x \) assume \( 1 \leq f x \cdot Z \leq g x \)
      with \( (0 < Z) \) have \( 1 \cdot Z \leq f x \cdot g x \)
        by (intro mult-mono) (auto simp: zero-le-divide-iff)
      then show \( Z \leq f x \cdot g x \)
        by simp
    qed
  qed

lemma filterlim-tendsto-pos-mult-at-bot:
  assumes \( (f \dashv \dashv >) c) F \) \( 0 < (c :: \text{real}) \) filterlim \( g \) at-bot \( F \)
  shows \( \mathrm{LIM} x F. f x \cdot g x \Rightarrow \text{at-bot} \)
  using filterlim-tendsto-pos-mult-at-top[OF assms(1,2), of \( \lambda x. -g x \)] assms(3)
  unfolding filterlim-uminus-at-bot by simp

lemma filterlim-pow-at-top:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  assumes \( 0 < n \) and \( f : \mathrm{LIM} x F. f x \Rightarrow \text{at-top} \)
  shows \( \mathrm{LIM} x F. (f x)^n :: \text{real} \Rightarrow \text{at-top} \)
  using \( (0 < n) \) proof (induct \( n \))
    case (Suc \( n \)) with \( f \) show ?case
      by (cases \( n = 0 \)) (auto intro!: filterlim-at-top-mult-at-top)
  qed simp

lemma filterlim-pow-at-bot-even:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  shows \( 0 < n \Rightarrow \mathrm{LIM} x F. f x \Rightarrow \text{at-bot} \Rightarrow \text{even} n \Rightarrow \mathrm{LIM} x F. (f x)^n :: \text{real} \Rightarrow \text{at-top} \)
  using filterlim-pow-at-top[of \( n \) \( \lambda x. -f x \)] \( F \) by (simp add: filterlim-uminus-at-top)

lemma filterlim-pow-at-bot-odd:
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  shows \( 0 < n \Rightarrow \mathrm{LIM} x F. f x \Rightarrow \text{at-bot} \Rightarrow \text{odd} n \Rightarrow \mathrm{LIM} x F. (f x)^n :: \text{real} \Rightarrow \text{at-bot} \)
  using filterlim-pow-at-top[of \( n \) \( \lambda x. -f x \)] \( F \) by (simp add: filterlim-uminus-at-top)

lemma filterlim-tendsto-add-at-top:
  assumes \( f : (f \dashv \dashv >) c) F \)
assumes \( g : \text{LIM} x F. \ g x :\ at\text{-top} \)
shows \( \text{LIM} x F. \ (f x + g x :: \text{real}) :\ at\text{-top} \)
unfolding \( \text{filterlim-at-top}\text{-gt}[\text{where} \ c=0] \)
proof safe
fix \( Z :: \text{real} \) assume \( 0 < Z \)
from \( f \) have eventually \((\lambda x. \ c - 1 < f x) \) \( F \)
by (auto dest! : \text{tendstoD}[\text{where} \ e=1] \ elim!: \text{eventually-elim1 simp: dist-real-def})
moreover from \( g \) have eventually \((\lambda x. \ Z - (c - 1) < g x) \) \( F \)
unfolding \( \text{filterlim-at-top} \) by auto
ultimately show eventually \((\lambda x. \ Z \leq f x + g x) \) \( F \)
by eventually-elim simp
qed

lemma \( \text{LIM\text{-at-top}-divide}\):  
fixes \( f g :: \ 'a \Rightarrow \) \( \text{real} \)
assumes \( f : (f \ ----> a) \) \( F \ 0 < a \)
assumes \( g : (g \ ----> 0) \) \( F \) eventually \((\lambda x. \ 0 < g x) \) \( F \)
shows \( \text{LIM} x F. \ f x / g x :\ at\text{-top} \)
unfolding \( \text{divide-inverse} \)
by (rule \( \text{filterlim-tendsto-pos-mul-at-top}[OF f] \)) (rule \( \text{filterlim-inverse-at-top}[OF g] \))

lemma \( \text{filterlim-at-top-add-at-top}\):  
assumes \( f : \text{LIM} x F. \ f x :\ at\text{-top} \)
assumes \( g : \text{LIM} x F. \ g x :\ at\text{-top} \)
shows \( \text{LIM} x F. \ (f x + g x :: \text{real}) :\ at\text{-top} \)
unfolding \( \text{filterlim-at-top}\text{-gt}[\text{where} \ c=0] \)
proof safe
fix \( Z :: \text{real} \) assume \( 0 < Z \)
from \( f \) have eventually \((\lambda x. \ 0 \leq f x) \) \( F \)
unfolding \( \text{filterlim-at-top} \) by auto
moreover from \( g \) have eventually \((\lambda x. \ Z \leq g x) \) \( F \)
unfolding \( \text{filterlim-at-top} \) by auto
ultimately show eventually \((\lambda x. \ Z \leq f x + g x) \) \( F \)
by eventually-elim simp
qed

lemma \( \text{tendsto\text{-divide}\text{-0}}\):  
fixes \( f :: - \Rightarrow 'a::\{\text{real-normed-div-algebra}, \text{division-ring-inverse-zero}\} \)
assumes \( f : (f \ ----> c) \) \( F \)
assumes \( g : \text{LIM} x F. \ g x :\ at\text{-infinity} \)
shows \((\lambda x. f x + g x) \ ----> 0) \) \( F \)
using \( \text{tendsto-mult}[OF f \text{filterlim-compose}[OF tendsto-inverse-0 g]] \) by (simp add: divide-inverse)

lemma \( \text{linear\text{-plus\text{-1-le-power}}}\):  
fixes \( x :: \text{real} \)
assumes \( x : 0 \leq x \)
shows \( \text{real} n * x + 1 \leq (x + 1) ^ n \)
proof (induct n)
case (Suc n)
  have real (Suc n) * x + 1 ≤ (x + 1) * (real n * x + 1)
    by (simp add: field-simps real-of-nat-Suc x)
  also have ... ≤ (x + 1) ^ Suc n
    using Suc x by (simp add: mult-left-mono)
finally show ?case .
qed simp
lemma filterlim-realpow-sequentially-gt1:
  fixes x :: 'a :: real-normed-div-algebra
  assumes x [arith]: 1 < norm x
  shows LIM n sequentially. x ^ n > at-infinity
proof (intro filterlim-at-infinity [THEN iffD2] allI impI)
  fix y :: real
  assume 0 < y
  have 0 < norm x − 1 by simp
  then obtain N :: nat where y < real N * (norm x − 1) by (blast dest: reals-Archimedean3)
  also have ...
    ≤ real N * (norm x − 1) + 1 by simp
  also have ... ≤ (norm x − 1 + 1) ^ N by (rule linear-plus-1-le-power) simp
  also have ... = norm x ^ N by simp
  finally have ∀ n ≥ N. y ≤ norm x ^ n
    by (metis order-less-le-trans power-increasing order-less-imp-le x)
  then show eventually (λn. y ≤ norm (x ^ n)) sequentially
    unfolding eventually-sequentially
    by (auto simp: norm-power)
qed simp

101.5 Limits of Sequences

lemma [trans]: X = Y ==> Y ===> z ==> X ===> z
  by simp

lemma LIMSEQ-iff:
  fixes L :: 'a::real-normed-vector
  shows (X ===> L) = (∀ r > 0. ∃ no. ∀ n ≥ no. norm (X n − L) < r)
    unfolding LIMSEQ-def dist-norm ..

lemma LIMSEQ-I:
  fixes L :: 'a::real-normed-vector
  shows (∀ r. 0 < r ==> ∃ no. ∀ n≥no. norm (X n − L) < r) ==> X ===> L
  by (simp add: LIMSEQ-iff)

lemma LIMSEQ-D:
  fixes L :: 'a::real-normed-vector
  shows [X ===> L; 0 < r] ==> ∃ no. ∀ n≥no. norm (X n − L) < r
  by (simp add: LIMSEQ-iff)

lemma LIMSEQ-linear: [ X ===> x ; l > 0 ] ==> (λ n. X (n * l)) ===> x
unfolding tendsto-def eventually-sequentially
by (metis div-le-dividend div-mult-self1-is-m le-trans mult.commute)

lemma Bseq-inverse-lemma:
fixes $x :: 'a::real-normed-div-algebra$
shows $[r \leq \text{norm } x; 0 < r] \Longrightarrow \text{norm } (\text{inverse } x) \leq \text{inverse } r$
apply (subst nonzero-norm-inverse, clarsimp)
apply (erule (1) le-imp-inverse-le)
done

lemma Bseq-inverse:
fixes $x :: 'a::real-normed-div-algebra$
shows $[X ----> a; a \neq 0] \Longrightarrow \text{Bseq } (\lambda n. \text{inverse } (X n))$
by (rule Bfun-inverse)

lemma LIMSEQ-diff-approach-zero:
fixes $L :: 'a::real-normed-vector$
shows $f ----> L ----> (\%x. f x - g x) ----> 0 ----> f ----> L$
by (drule (1) tendsto-add, simp)

lemma LIMSEQ-diff-approach-zero2:
fixes $L :: 'a::real-normed-vector$
shows $f ----> L ----> (\%x. f x - g x) ----> 0 ----> g ----> L$
by (drule (1) tendsto-diff, simp)

An unbounded sequence’s inverse tends to 0

lemma LIMSEQ-inverse-zero:
\forall r::real. \exists N. \forall n\geq N. r < X n \Longrightarrow (\lambda n. \text{inverse } (X n)) ----> 0
apply (rule filterlim-compose[OF tendsto-inverse-0])
apply (simp add: filterlim-at-infinity[OF order-refl] eventually-sequentially)
apply (metis abs-le-D1 linorder-le-cases linorder-not-le)
done

The sequence $(1::'a) / n$ tends to 0 as $n$ tends to infinity

lemma LIMSEQ-inverse-real-of-nat: $(\%n. \text{inverse } (real (Suc n))) ----> 0$
by (metis filterlim-compose tendsto-inverse-0 filterlim-mono order-refl filterlim-Suc filterlim-compose[OF filterlim-real-sequentially] at-top-le-at-infinity)

The sequence $r + (1::'a) / n$ tends to $r$ as $n$ tends to infinity is now easily proved

lemma LIMSEQ-inverse-real-of-nat-add:
$(\%n. r + \text{inverse } (real (Suc n))) ----> r$
using tendsto-add [OF tendsto-const LIMSEQ-inverse-real-of-nat] by auto

lemma LIMSEQ-inverse-real-of-nat-add-minus:
$(\%n. r + \text{inverse } (real (Suc n))) ----> r$
using tendsto-add [OF tendsto-const tendsto-minus [OF LIMSEQ-inverse-real-of-nat]] by auto
lemma LIMSEQ-inverse-real-of-nat-add-minus-mult:
(\%n. r * (1 + -inverse(real(Suc n)))) ----> r
using tendsto-mult [OF tendsto-const LIMSEQ-inverse-real-of-nat-add-minus [of 1]]
by auto

101.6 Convergence on sequences

lemma convergent-add:
fixes X Y :: nat ⇒ 'a::real-normed-vector
assumes convergent (λn. X n)
assumes convergent (λn. Y n)
shows convergent (λn. X n + Y n)
using assms unfolding convergent-def by (fast intro: tendsto-add)

lemma convergent-setsum:
fixes X :: 'a ⇒ nat ⇒ 'b::real-normed-vector
assumes A: \(\forall i. i \in A \implies \text{convergent} (\lambda n. X i n)\)
shows convergent (λn. \(\sum_{i\in A} X i n\) )
proof (cases finite A)
case True from this and assms show ?thesis
  by (induct A set: finite) (simp-all add: convergent-const convergent-add)
qed (simp add: convergent-const)

lemma (in bounded-linear) convergent:
assumes convergent (λn. X n)
shows convergent (λn. f (X n))
using assms unfolding convergent-def by (fast intro: tendsto)

lemma (in bounded-bilinear) convergent:
assumes convergent (λn. X n) and convergent (λn. Y n)
shows convergent (λn. X n ** Y n)
using assms unfolding convergent-def by (fast intro: tendsto)

lemma convergent-minus-iff:
fixes X :: nat ⇒ 'a::real-normed-vector
shows convergent X ─→ convergent (λn. - X n)
apply (simp add: convergent-def)
apply (auto dest: tendsto-minus)
apply (drule tendsto-minus, auto)
done

A monotone sequence converges to its least upper bound.

lemma LIMSEQ-incseq-SUP:
fixes X :: nat ⇒ 'a::{conditionally-complete-linorder, linorder-topology}
assumes u: bdd-above (range X)
assumes X: incseq X
shows X −−−→ (SUP i. X i)
by (rule order-tendstoI)
lemma LIMSEQ-decseq-INF:
  fixes X :: nat ⇒ 'a::(conditionally-complete-linorder, linorder-topology)
  assumes u: bdd-below (range X)
  assumes X: decseq X
  shows X −−−−> (INF i. X i)
  by (rule order-tendstoI)
    (auto simp: eventually-sequentially u cINF-less-iff intro: X[THEN decseqD]
    le-less-trans less-cINF-D[OF u])

Main monotonicity theorem

lemma Bseq-monoseq-convergent: Bseq X ⇒ monoseq X ⇒ convergent (X::nat⇒real)
  by (auto simp: monoseq-iff convergent-def intro: LIMSEQ-decseq-INF LIMSEQ-incseq-SUP
    dest: Bseq-bdd-above Bseq-bdd-below)

lemma Bseq-mono-convergent: Bseq X ⇒ (∀m n. m ≤ n −→ X m ≤ X n) ⇒ convergent (X::nat⇒real)
  by (auto intro!: Bseq-monoseq-convergent incseq-imp-monoseq simp: incseq-def)

lemma Cauchy-iff:
  fixes X :: nat ⇒ 'a::real-normed-vector
  shows Cauchy X ←→ (∀e>0. ∃M. ∀m≥M. ∀n≥M. norm (X m − X n) < e)
    unfolding Cauchy-def dist-norm ..

lemma CauchyI:
  fixes X :: nat ⇒ 'a::real-normed-vector
  shows (∀e. 0 < e ⇒ ∃M. ∀m≥M. ∀n≥M. norm (X m − X n) < e) ⇒ Cauchy X
  by (simp add: Cauchy-iff)

lemma CauchyD:
  fixes X :: nat ⇒ 'a::real-normed-vector
  shows [Cauchy X; 0 < e] ⇒ ∃M. ∀m≥M. ∀n≥M. norm (X m − X n) < e
  by (simp add: Cauchy-iff)

lemma incseq-convergent:
  fixes X :: nat ⇒ real
  assumes incseq X and ∀i. X i ≤ B
  obtains L where X −−−−> L ∀i. X i ≤ L
  proof atomize-elim
    from incseq-bounded[OF assms] ⟨incseq X⟩ Bseq-monoseq-convergent[of X]
    obtain L where X −−−−> L
      by (auto simp: convergent-def monoseq-def incseq-def)
    with ⟨incseq X⟩ show ∃L. X −−−−> L ∧ (∀i. X i ≤ L)
      by (auto intro!: exI[of - L] incseq-le)
  qed
lemma \textit{decseq-convergent}:

fixes \(X : \text{nat} \Rightarrow \text{real}\)

assumes \(\text{decseq } X\) and \(\forall i. \, B \leq X_i\)

obtains \(L\) where \(X \longrightarrow L\) \(\forall i. \, L \leq X_i\)

proof atomize-elim

from \(\text{decseq-bounded}[\text{OF asms}]\) \([\text{decseq } X] \rightarrow \text{Bseq-monoseq-convergent}[\text{of } X]\)

obtain \(L\) where \(X \longrightarrow L\) \(\forall i. \, L \leq X_i\)

by (auto simp: \text{convergent-def monoseq-def decseq-def})

with \([\text{decseq } X]\)

show \(\exists L. \, X \longrightarrow L \land (\forall i. \, L \leq X_i)\)

by (auto intro!: \text{exI} \[\text{of - } L\] \text{decseq-le})

qed

101.6.1 Cauchy Sequences are Bounded

A Cauchy sequence is bounded – this is the standard proof mechanization rather than the nonstandard proof

lemma \textit{lemmaCauchy}:

\(\forall n \geq M. \, \text{norm}\ (X M - X n) < (1 :: real)\)

\[\rightarrow \forall n \geq M. \, \text{norm}\ (X n :: 'a :: real-normed-vector) < 1 + \text{norm}\ (X M)\]

apply (clarify, drule spec, drule (1) mp)

apply (simp only: \text{norm-minus-commute})

apply (drule \text{order-le-less-trans} [OF \text{norm-triangle-ineq2}])

apply simp

done

101.7 Power Sequences

The sequence \(x^n\) tends to 0 if \((0 :: 'a) \leq x\) and \(x < (1 :: 'a)\). Proof will use (NS) Cauchy equivalence for convergence and also fact that bounded and monotonic sequence converges.

lemma \textit{Bseq-realpow}:

\(\| 0 \leq (x :: real); x \leq 1 \| \rightarrow \text{Bseq}\ (%n. \, x \ ^ \ n)\)

apply (simp add: \text{Bseq-def})

apply (rule-tac x = 1 in \text{exI})

apply (simp add: \text{power-abs})

apply (auto dest: \text{power-mono})

done

lemma \textit{monoseq-realpow}:

\(\| 0 \leq x; x \leq 1 \| \rightarrow \text{monoseq}\ (%n. \, x \ ^ \ n)\)

apply (clarify intro!: \text{mono-SucI2})

apply (cut-tac n = n and N = Suc n and a = x in \text{power-decreasing, auto})

done

lemma \textit{convergent-realpow}:

\(\| 0 \leq (x :: real); x \leq 1 \| \rightarrow \text{convergent}\ (%n. \, x \ ^ \ n)\)

by (blast intro!: \text{Bseq-monoseq-convergent Bseq-realpow monoseq-realpow})

lemma \textit{LIMSEQ-inverse-realpow-zero}:

\(1 < (x :: real) \rightarrow (\lambda n. \, \text{inverse}\ (x \ ^ \ n))\)

\(\rightarrow \theta\)
by (rule filterlim-compose[OF tendsto-inverse-0 filterlim-realpow-sequentially-gt1])
simp

lemma LIMSEQ-realpow-zero:
\[ 0 \leq (x::real); x < 1 \implies (\lambda n. x^n) \rightarrow 0 \]

proof cases
  assume 0 \leq x and x \neq 0
  hence x > 0 by simp

assume x1 :: x < 1

from x0 x1 have 1 < inverse x by (rule one_less_inverse)

hence (\lambda n. inverse (inverse x^n)) \rightarrow 0

by (rule LIMSEQ-inverse-realpow-zero)

thus ?thesis by (simp add: power_inverse)

qed (rule LIMSEQ_imp_Suc, simp add: tendsto_const)

lemma LIMSEQ-power-zero:
# fixes x :: 'a::real
# shows norm x < 1 \rightarrow (\lambda n. x^n) \rightarrow 0

apply (erule LIMSEQ_realpow_zero)

apply (simp only: tendsto_Zfun_iff, erule Zfun_le)

apply (simp add: power_abs norm_power_ineq)

done

lemma LIMSEQ-divide-realpow-zero: \(1 < x \implies (\lambda n. a / (x^n)) \rightarrow 0 \)

by (rule tendsto_divide_0)


101.8 Limits of Functions

lemma LIM-eq:
  fixes a :: 'a::real
  and L :: 'b::real
  shows f \rightarrow a \rightarrow L =
  \((\forall r. \exists s. \forall x. x \neq a \& \norm (x-a) < s \rightarrow \norm (f x - L) < r)\)

by (simp add: LIM_def dist_norm)

lemma LIM-I:
  fixes a :: 'a::real
  and L :: 'b::real
  shows (!r. \(0 < r \implies \exists s > 0. \forall x. x \neq a \& \norm (x-a) < s \rightarrow \norm (f x - L) < r\) \rightarrow f \rightarrow a \rightarrow L)
by (simp add: LIM-eq)

lemma LIM-D:
    fixes a :: 'a::real-normed-vector and L :: 'b::real-normed-vector
    shows \(|| f --\to a --\to L; 0 < r ||\) ==\to \exists s > 0. \forall x. x \neq a \& \text{norm}(x - a) < s \text{ --\to norm}(f x - L) < r
    by (simp add: LIM-eq)

lemma LIM-offset:
    fixes a :: 'a::real-normed-vector
    shows f --\to a --\to L =\to (\lambda x. f (x + k)) --\to a --\to L
    unfolding filtermap-at-shift[symmetric, of a k] filterlim-def filtermap-filtermap
    by simp

lemma LIM-offset-zero:
    fixes a :: 'a::real-normed-vector
    shows f --\to a --\to L =\to (\lambda h. f (a + h)) --\to 0 --\to L
    by (drule-tac k = a in LIM-offset, simp add: add.commute)

lemma LIM-offset-zero-cancel:
    fixes a :: 'a::real-normed-vector
    shows (\lambda h. f (a + h)) --\to 0 --\to L =\to f --\to a --\to L
    by (drule-tac k = - a in LIM-offset, simp)

lemma LIM-offset-zero-iff:
    fixes f :: 'a::metric-space => 'b::real-normed-vector
    shows (\lambda h. f (a + h)) --\to 0 --\to L =\iff (\lambda x. f x - l) --\to 0 --\to l F
    unfolding tendsto_iff dist-norm by simp

lemma LIM-zero:
    fixes f :: 'a::topological-space => 'b::real-normed-vector
    shows (f --\to l) F =\iff ((\lambda x. f x - l) --\to 0) F
    unfolding tendsto_iff dist-norm by simp

lemma LIM-zero-cancel:
    fixes f :: 'a::topological-space => 'b::real-normed-vector
    shows ((\lambda x. f x - l) --\to 0) F =\iff (f --\to l) F
    unfolding tendsto_iff dist-norm by simp

lemma LIM-zero-iff:
    fixes f :: 'a::metric-space => 'b::real-normed-vector
    shows ((\lambda x. f x - l) --\to 0) F =\iff (f --\to l) F
    unfolding tendsto_iff dist-norm by simp

lemma LIM-imp-LIM:
    fixes f :: 'a::topological-space => 'b::real-normed-vector
    fixes g :: 'a::topological-space => 'c::real-normed-vector
    assumes f: f --\to l
    assumes le: \(\forall x. x \neq a \Rightarrow \text{norm}(g x - m) \leq \text{norm}(f x - l)\)
shows \( g \rightarrow a \rightarrow m \)
by (rule metric-LIM-imp-LIM [OF f], 
simp add: dist-norm le)

**lemma LIM-equal2:**
fixes \( f, g :: 'a::real-normed-vector \Rightarrow 'b::topological-space \)
assumes 1: \( 0 < R \)
assumes 2: \( \forall x. [x \neq a; \text{norm } (x - a) < R] \implies f x = g x \)
shows \( g \rightarrow a \rightarrow l \implies f \rightarrow a \rightarrow l \)
by (rule metric-LIM-equal2 [OF 1 2], simp-all add: dist-norm)

**lemma LIM-compose2:**
fixes \( a :: 'a::real-normed-vector \)
assumes \( f :: f \rightarrow a \rightarrow b \)
assumes \( g :: g \rightarrow b \rightarrow c \)
assumes inj: \( \exists d > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \implies f x \neq b \)
shows \( (\lambda x. g (f x)) \rightarrow a \rightarrow c \)
by (rule metric-LIM-compose2 [OF f g inj [folded dist-norm]])

**lemma real-LIM-sandwich-zero:**
fixes \( f, g :: 'a::topological-space \Rightarrow \text{real} \)
assumes \( f :: f \rightarrow a \rightarrow 0 \)
assumes 1: \( \forall x. x \neq a \implies 0 \leq g x \)
assumes 2: \( \forall x. x \neq a \implies g x \leq f x \)
shows \( g \rightarrow a \rightarrow 0 \)
proof (rule LIM-imp-LIM [OF f])
fix x assume \( x \neq a \)
have \( \text{norm } (g x - 0) = g x \) by (simp add: 1 x)
also have \( g x \leq f x \) by (rule 2 [OF x])
also have \( f x \leq |f x| \) by (rule abs-ge-self)
also have \( |f x| = \text{norm } (f x - 0) \) by simp
finally show \( \text{norm } (g x - 0) \leq \text{norm } (f x - 0) \).
qed

### 101.9 Continuity

**lemma LIM-isCont-iff:**
fixes \( f :: 'a::real-normed-vector \Rightarrow 'b::topological-space \)
shows \( (f \rightarrow a \rightarrow f a) = (\lambda h. f (a + h)) \rightarrow 0 \rightarrow f a) \)
by (rule iffI [OF LIM-offset-zero LIM-offset-zero-cancel])

**lemma isCont-iff:**
fixes \( f :: 'a::real-normed-vector \Rightarrow 'b::topological-space \)
shows isCont \( f x = (\lambda h. f (x + h)) \rightarrow 0 \rightarrow f x \)
by (simp add: isCont-def LIM-isCont-iff)

**lemma isCont-LIM-compose2:**
fixes \( a :: 'a::real-normed-vector \)
assumes \( f :: [unfolded isCont-def]: \text{isCont } f a \)
assumes $g: g \rightarrow f a \rightarrow l$
assumes inj: $\exists d > 0, \forall x. x \neq a \land \text{norm } (x - a) < d \rightarrow f x \neq f a$
shows $(\lambda x. (g (f x))) \rightarrow a \rightarrow l$
by (rule $LIM-compose2$ [OF $f \ g$ inj])

lemma isCont-norm [simp]:
fixes $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
shows $\text{isCont } f a \Rightarrow \text{isCont } (\lambda x. \text{norm } (f x)) a$
by (fact continuous-norm)

lemma isCont-rabs [simp]:
fixes $f :: 'a::t2-space \Rightarrow \text{real}$
shows $\text{isCont } f a \Rightarrow \text{isCont } (\lambda x. |f x|) a$
by (fact continuous-rabs)

lemma isCont-add [simp]:
fixes $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
shows $[\text{isCont } f a; \text{isCont } g a] \Rightarrow \text{isCont } (\lambda x. f x + g x) a$
by (fact continuous-add)

lemma isCont-minus [simp]:
fixes $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
shows $\text{isCont } f a \Rightarrow \text{isCont } (\lambda x. - f x) a$
by (fact continuous-minus)

lemma isCont-diff [simp]:
fixes $f :: 'a::t2-space \Rightarrow 'b::real-normed-vector$
shows $[\text{isCont } f a; \text{isCont } g a] \Rightarrow \text{isCont } (\lambda x. f x - g x) a$
by (fact continuous-diff)

lemma isCont-mult [simp]:
fixes $f g :: 'a::t2-space \Rightarrow 'b::real-normed-algebra$
shows $[\text{isCont } f a; \text{isCont } g a] \Rightarrow \text{isCont } (\lambda x. f x * g x) a$
by (fact continuous-mult)

lemma (in bounded-linear) isCont:
$\text{isCont } g a \Rightarrow \text{isCont } (\lambda x. f (g x)) a$
by (fact continuous)

lemma (in bounded-bilinear) isCont:
$[\text{isCont } f a; \text{isCont } g a] \Rightarrow \text{isCont } (\lambda x. f x ** g x) a$
by (fact continuous)

lemmas isCont-scaleR [simp] =
bounded-bilinear.isCont [OF bounded-bilinear-scaleR]

lemmas isCont-of-real [simp] =
bounded-linear.isCont [OF bounded-linear-of-real]
lemma isCont-power [simp]:
  fixes f :: `'a::t2-space ⇒ `'b::{power,real-normed-algebra}
  shows isCont f a =⇒ isCont (λx. f x ^ n) a
by (fact continuous-power)

lemma isCont-setsum [simp]:
  fixes f :: `'a ⇒ `'b::t2-space ⇒ `'c::real-normed-vector
  shows ∀i∈A. isCont (f i) a =⇒ isCont (λx. ∑i∈A. f i x) a
by (auto intro: continuous-setsum)

101.10 Uniform Continuity

definition isUCont :: [′a::metric-space ⇒ ′b::metric-space] ⇒ bool where
  isUCont f = (∀r>0. ∃s>0. ∀x y. dist x y < s −→ dist (f x) (f y) < r)

lemma isUCont-isCont: isUCont f =⇒ isCont f x
by (simp add: isUCont-def isCont-def LIM-def, force)

lemma isUCont-Cauchy: [isUCont f; Cauchy X] =⇒ Cauchy (λn. f (X n))
unfolding isUCont-def
apply (rule metric-CauchyI)
apply (drule_tac x=e in spec, safe)
apply (drule_tac e=s in metric-CauchyD, safe)
apply (rule-tac x=M in exI, simp)
done

lemma (in bounded-linear) isUCont: isUCont f
unfolding isUCont-def dist-norm
proof (intro allI impI)
  fix r::real assume r: 0 < r
  obtain K where K: 0 < K and norm-le: ∀x. norm (f x) ≤ norm x * K
    using pos-bounded by fast
  show ∃s>0. ∀x y. norm (x − y) < s −→ norm (f x − f y) < r
proof (rule exI, safe)
    from r K show 0 < r / K by simp
next
  fix x y :: ′a
  assume xy: norm (x − y) < r / K
  have norm (f x − f y) = norm (f (x − y)) by (simp only: diff)
  also have ... ≤ norm (x − y) * K by (rule norm-le)
  also from K xy have ... < r by (simp only: pos-less-divide-eq)
  finally show norm (f x − f y) < r.
  qed
  qed

lemma (in bounded-linear) Cauchy: Cauchy X =⇒ Cauchy (λn. f (X n))
by (rule isUCont [THEN isUCont-Cauchy])

lemma LIM-less-bound:
  fixes f :: real ⇒ real
  assumes ev: b < x ∨ x' ∈ { b <..< x}. 0 ≤ f x' and isCont f x
  shows 0 ≤ f x
proof (rule tendsto-le-const)
  show (f ----> f x) (at-left x)
    using (isCont f x) by (simp add: filterlim-at-split isCont-def)
  show eventually (λx. 0 ≤ f x) (at-left x)
    using ev by (auto simp: eventually-at dist-real-def intro: exI[of f - x - b])
qed simp

101.11 Nested Intervals and Bisection – Needed for Compactness

lemma nested-sequence-unique:
  assumes ∀ n. f n ≤ f (Suc n) ∀ n. g (Suc n) ≤ g n ∀ n. f n ≤ g n (λn. f n - g n) ----> 0
  shows ∃ l::real. ((∀ n. f n ≤ l) ∧ f ----> l) ∧ ((∀ n. l ≤ g n) ∧ g ----> l)
proof –
  have incseq f unfolding incseq-Suc-iff by fact
  have decseq g unfolding decseq-Suc-iff by fact

  { fix n
    from (decseq g) have g n ≤ g 0 by (rule decseqD simp)
    with (∀ n. f n ≤ g n) [THEN spec, of n] have f n ≤ g 0 by auto
    then obtain u where f ----> u ∀ i. f i ≤ u
      using incseq-convergent[OF (incseq f)] by auto
    moreover
      { fix n
        from (incseq f) have f 0 ≤ f n by (rule incseqD simp)
        with (∀ n. f n ≤ g n) [THEN spec, of n] have f 0 ≤ g n by simp
        then obtain l where g ----> l ∀ i. l ≤ g i
          using decseq-convergent[OF (decseq g)] by auto
        moreover note LIMSEQ-unique[OF assms(4) tendsto-diff[OF (f ----> w)]
          ⟨g ----> l]]
      ultimately show thesis by auto
    qed

lemma Bolzano[consumes 1, case-names trans local]:
  fixes P :: real ⇒ real ⇒ bool
  assumes [arith]: a ≤ b
  assumes trans: ∀ a b c. [P a b; P b c; a ≤ b; b ≤ c] P a c
  assumes local: ∀ x. a ≤ x ⇒ x ≤ b ⇒ ∃ d>0. ∀ a b. a ≤ x ∧ x ≤ b ∧ b - a < d ----> P a b
  shows P a b
proof –
  def bisect ≡ rec-nat (a, b) (λn (x, y). if P x ((x+y) / 2) then ((x+y)/2, y) else
(x, (x+y)/2)
def l ≡ λn. fst (bisect n) and u ≡ λn. snd (bisect n)
have l[simp]: l 0 = a ∧ n. l (Suc n) = (if P (l n) ((l n + u n) / 2) then (l n + u n) / 2 else l n)
and u[simp]: u 0 = b ∧ n. u (Suc n) = (if P (l n) ((l n + u n) / 2) then u n else (l n + u n) / 2)
by (simp-all add: l-def u-def bisect-def split; prod.split)

{ fix n have l n ≤ u n by (induct n) auto } note this[simp]

have ∃x. (∀n. l n ≤ x) ∧ l ----> x) ∧ (∀n. x ≤ u n) ∧ u ----> x
proof (safe intro!: nested-sequence-unique)
fix n show l n ≤ l (Suc n) u (Suc n) ≤ u n by (induct n) auto
next
{ fix n have l n − u n = (a − b) / 2ˆn by (induct n) (auto simp: field-simps)
}
then show (λn. l n − u n) ----> 0 by (simp add: LIMSEQ-divide-realpow-zero)
qed fact

then obtain x where x: ∃n. l n ≤ x ∧ n. x ≤ u n and l ----> x u ----> x by auto

obtain d where 0 < d and d: ∀a b. a ≤ x ⇒ x ≤ b ⇒ b − a < d ⇒ P a b
using (l 0 ≤ x) 0 ≤ u 0) local[of x] by auto

show P a b
proof (rule ccontr)
assume ¬P a b
{ fix n have ¬P (l n) (u n)
proof (induct n)
case (Suc n) with trans[of l n (l n + u n) / 2 u n] show ?case by auto
qed (simp add: ¬P a b)
}

moreover
{ have eventually (λn. x − d / 2 ≤ l n) sequentially
using (l 0 < d) l ----> x) by (intro order-tendstoD[of - x]) auto
moreover have eventually (λn. u n < x + d / 2) sequentially
using (l 0 < d) u ----> x) by (intro order-tendstoD[of - x]) auto
ultimately have eventually (λn. P (l n) (u n)) sequentially
proof eventually-elim
fix n assume x − d / 2 ≤ l n u n < x + d / 2
from add-strict-mono[OF this] have u n − l n < d by simp
with x show P (l n) (u n) by (rule d)
qed }
ultimately show False by simp
qed
def T == {a .. b}

from C(1,3) show \exists C' \subseteq C. finite C' \land \{ a..b \} \subseteq \bigcup C'
proof (induct rule: Bolzano)
  case (trans a b c)
  then have *: \{ a .. c \} = \{ a .. b \} \cup \{ b .. c \} by auto
  from trans obtain C1 C2 where C1 \subseteq C \land finite C1 \land \{ a..b \} \subseteq \bigcup C1 C2
  \land finite C2 \land \{ b..c \} \subseteq \bigcup C2
  by (auto simp: *)
  with trans show ?case
  unfolding * by (intro exI[of - C1 \cup C2]) auto
next
  case (local x)
  then have x \in \bigcup C using C by auto
  with C(2) obtain c where x \in c open c c \in C by auto
  then obtain e where 0 < e \{ x - e <..< x + e \} \subseteq c
  by (auto simp: open-real-def dist-real-def subset-eq Ball-def abs-less-iff)
  with \langle c \in C \rangle show ?case
  by (safe intro!: exI[of - e/2] exI[of - {c}] auto)
qed simp

lemma continuous-image-closed-interval:
  fixes a b and f :: real \Rightarrow real
defines S \equiv \{ a..b \}
assumes a \leq b and f: continuous-on S f
shows \exists c d. f'S = \{ c..d \} \land c \leq d
proof –
  have S: compact S S \neq {}
    using \langle a \leq b \rangle by (auto simp: S-def)
  obtain c where c \in S \forall d \in S. f d \leq f c
    using continuous-attains-sup[OF S f] by auto
  moreover obtain d where d \in S \forall c \in S. f d \leq f c
    using continuous-attains-inf[OF S f] by auto
  moreover have connected (f'S)
    using connected-continuous-image[OF f] connected-icc by (auto simp: S-def)
  ultimately have f : S = \{ f d .. f c \} \land f d \leq f c
    by (auto simp: connected-iff-interval)
  then show ?thesis
    by auto
qed

101.12 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma isCont-eq-Ub:
  fixes f :: real \Rightarrow 'a::linorder_topology
shows a \leq b \Rightarrow \forall x::real. a \leq x \land x \leq b \rightarrow isCont f x \Rightarrow
  \exists M. (\forall x. a \leq x \land x \leq b \rightarrow f x \leq M) \land (\exists x. a \leq x \land x \leq b \land f x = M)
using continuous-attains-sup[of { a .. b } f]
by (auto simp add: continuous-at-imp-continuous-on Ball-def Bex-def)

lemma isCont-eq-Lb:
  fixes f :: real ⇒ 'a::linorder-topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x
  using continuous-attains-inf[of { a .. b } f]
  by (auto simp add: continuous-at-imp-continuous-on Ball-def Bex-def)

lemma isCont-bounded:
  fixes f :: real ⇒ 'a::linorder-topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x
  using continuous-attains-sup[of { a .. b } f] by auto

lemma isCont-has-Ub:
  fixes f :: real ⇒ 'a::linorder-topology
  shows a ≤ b ⇒ ∀ x. a ≤ x ∧ x ≤ b → isCont f x
  using continuous-attains-sup[of { a .. b } f] by auto

lemma IVT-objl: (f(a::real) ≤ (y::real) & y ≤ f(b) & a ≤ b &
  (∀ x. a ≤ x & x ≤ b ---> isCont f x))
  --->(∃ x. a ≤ x & x ≤ b & f(x) = y)
  by (blast intro: IVT)

lemma IVT2-objl: (f(b::real) ≤ (y::real) & y ≤ f(a) & a ≤ b &
  (∀ x. a ≤ x & x ≤ b ---> isCont f x))
  --->(∃ x. a ≤ x & x ≤ b & f(x) = y)
  by (blast intro: IVT2)

lemma isCont-Lb-Ub:
  fixes f :: real ⇒ real
  assumes a ≤ b ∀ x. a ≤ x ∧ x ≤ b ---> isCont f x
  shows ∃ L M. (∀ x. a ≤ x ∧ x ≤ b ---> L ≤ f x ∧ f x ≤ M) ∧
  (∀ y. L ≤ y ∧ y ≤ M ---> (∃ x. a ≤ x ∧ x ≤ b ∧ (f x = y)))

proof –
obtain M where M: a ≤ M ≤ b ∀ x. a ≤ x ∧ x ≤ b ---> f x ≤ f M
  using isCont-eq-Ub[OF assms] by auto
obtain L where L: a ≤ L ≤ b ∀ x. a ≤ x ∧ x ≤ b ---> f L ≤ f x
  using isCont-eq-Lb[OF assms] by auto
show ?thesis
  apply (rule-tac x=f L in exI)
  apply (rule-tac x=f M in exI)
  apply (cases L ≤ M)
apply (simp, metis order-trans)
apply (simp, metis order-trans)
done
qed

Continuity of inverse function

lemma isCont-inverse-function:
fixes f g :: real ⇒ real
assumes d: 0 < d
  and inj: ∀ z. |z−x| ≤ d −→ g (f z) = z
  and cont: ∀ z. |z−x| ≤ d −→ isCont f z
shows isCont g (f x)
proof ¬
let ?A = f (x − d) and ?B = f (x + d) and ?D = {x − d..x + d}

have f: continuous-on ?D f
  using cont by (intro continuous-at-imp-continuous-on ballI) auto
then have g: continuous-on (f'?D) g
  using inj by (intro continuous-on-inv) auto

from d f have {min ?A ?B <..< max ?A ?B} ⊆ f ' ?D
  by (intro connected-contains-loo connected-continuous-image) (auto split: split-min split-max)
with g have continuous-on {min ?A ?B <..< max ?A ?B} g
  by (rule continuous-on-subset)
moreover
have ( ?A < f x ∧ f x < ?B ) ∨ ( ?B < f x ∧ f x < ?A )
  using d inj by (intro continuous-inj-imp-mono[OF - - f inj-onI2[of g, OF inj-on1]]) auto
then have f x ∈ {min ?A ?B <..< max ?A ?B}
  by auto
ultimately
show ?thesis
  by (simp add: continuous-on-eq-continuous-at)
qed

lemma isCont-inverse-function2:
fixes f g :: real ⇒ real shows
[a < x; x < b;
 ∀ z. a ≤ z ∧ z ≤ b −→ g (f z) = z;
 ∀ z. a ≤ z ∧ z ≤ b −→ isCont f z]
⇒ isCont g (f x)
apply (rule isCont-inverse-function
  [where f=f and d=min (x − a) (b − x)])
apply (simp-all add: abs-le-iff)
done
lemma isCont-inv-fun:
fixes f g :: real ⇒ real
shows \[ |0 < d; \forall z. |z - x| \leq d \longrightarrow g(f(z)) = z; \forall z. |z - x| \leq d \longrightarrow isCont f x \]
==⇒ isCont g (f x)
by (rule isCont-inverse-function)

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110

lemma LIM-fun-gt-zero:
fixes f :: real ⇒ real
shows f c l =⇒ 0 < l =⇒ \exists r. 0 < r ∧ (∀x. x ≠ c ∧ |c - x| < r → 0 < f x)
apply (drule (1) LIM-D, clarify)
apply (rule-tac x = s in exI)
apply (simp add: abs-less-iff)
done

lemma LIM-fun-less-zero:
fixes f :: real ⇒ real
shows f c l =⇒ l < 0 =⇒ \exists r. 0 < r ∧ (∀x. x ≠ c ∧ |c - x| < r → f x < 0)
apply (drule LIM-D [where r=−l], simp, clarify)
apply (rule-tac x = s in exI)
apply (simp add: abs-less-iff)
done

lemma LIM-fun-not-zero:
fixes f :: real ⇒ real
shows f c l =⇒ l ≠ 0 =⇒ \exists r. 0 < r ∧ (∀x. x ≠ c ∧ |c - x| < r → f x ≠ 0)
end

102 Series: Infinite Series

theory Series
imports Limits
begin

102.1 Definition of infinite summability

definition sums :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a ⇒ bool
(infixr sums 80)
where
f sums s ←→ (λn. \(\sum_{i<n} f \ i\)) ----> s
definition summable :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ bool
where
  summable f ←→ (∃ s. f sums s)

definition suminf :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a
  (binder ∑ 10)
where
  suminf f = (THE s. f sums s)

102.2 Infinite summability on topological monoids

lemma sums-subst[trans]: f = g ⇒ g sums z ⇒ f sums z
  by simp

lemma sums-summable: f sums l ⇒ summable f
  by (simp add: sums-def summable-def, blast)

lemma summable-iff-convergent: summable f ←→ convergent (λn. ∑ i<n. f i)
  by (simp add: summable-def sums-def convergent-def)

lemma suminf-eq-lim: suminf f = lim (λn. ∑ i<n. f i)
  by (simp add: suminf-def sums-def lim-def)

lemma sums-zero[simp, intro]: (λn. 0) sums 0
  unfolding sums-def by (simp add: tendsto-const)

lemma summable-zero[simp, intro]: summable (λn. 0)
  by (rule sums-zero [THEN sums-summable])

lemma sums-group: f sums s ⇒ 0 < k ⇒ (λn. setsum f {n * k ..< n * k + k}) sums s
  apply (simp only: sums-def setsum-nat-group tendsto-def eventually-sequentially)
  apply safe
  apply (erule-tac x=S in allE)
  apply safe
  apply (rule-tac x=N in exI, safe)
  apply (drule-tac x=n*k in spec)
  apply (erule mp)
  apply (erule order-trans)
  apply simp
  done

lemma sums-finite:
  assumes [simp]: finite N and f: ∀ n. n ∉ N ⇒ f n = 0
  shows f sums (∑ n∈N. f n)
proof –
  { fix n
    have setsum f {..<n + Suc (Max N)} = setsum f N

THEORY “Series”
proof cases
  assume \( N = \{\} \)
  with \( f \) have \( f = (\lambda x. \, 0) \) by auto
  then show ?thesis by simp
next
  assume [simp]: \( N \neq \{\} \)
  show ?thesis
  proof (safe intro!: setsum.mono-neutral-right \( f \))
  fix \( i \)
  assume \( i \in N \)
  then have \( i \leq \text{Max} \, N \) by simp
  then show \( i < n + \text{Suc} \, (\text{Max} \, N) \) by simp
  qed
qed

note eq = this
show ?thesis unfolding sums-def
  (rule LIMSEQ-offset ![of - Suc (\text{Max} \, N)])
  (simp add: eq atLeast0LessThan tendsto-const del: add-Suc-right)
qed

lemma summable-finite: \( \text{finite} \, N \implies (\forall n. \, n \notin N \implies f \, n = 0 ) \implies \text{summable} \, f \)
  by (rule sums-summable) (rule sums-finite)

lemma sums-If-finite-set: \( \text{finite} \, A \implies (\lambda r. \, \text{if} \, r \in A \text{ then} \, f \, r \text{ else} \, 0 ) \text{ sums} \, (\sum r \in A. \, f \, r) \)
  using sums-finite ![of A (\lambda r. \, \text{if} \, r \in A \text{ then} \, f \, r \text{ else} \, 0 )] by simp

lemma summable-If-finite-set[simp, intro]: \( \text{finite} \, A \implies \text{summable} \, (\lambda r. \, \text{if} \, r \in A \text{ then} \, f \, r \text{ else} \, 0 ) \)
  by (rule sums-summable) (rule sums-If-finite-set)

lemma sums-If-finite: \( \text{finite} \, \{r. \, \text{P} \, r\} \implies (\lambda r. \, \text{if} \, P \, r \text{ then} \, f \, r \text{ else} \, 0 ) \text{ sums} \, (\sum r \mid P \, r. \, f \, r) \)
  using sums-If-finite ![of \{r. \, \text{P} \, r\}] by simp

lemma summable-If-finite[simp, intro]: \( \text{finite} \, \{r. \, \text{P} \, r\} \implies \text{summable} \, (\lambda r. \, \text{if} \, P \, r \text{ then} \, f \, r \text{ else} \, 0 ) \)
  by (rule sums-summable) (rule sums-If-finite)

lemma sums-single: (\lambda r. \, \text{if} \, r = i \text{ then} \, f \, r \text{ else} \, 0 ) \text{ sums} \, f \, i
  using sums-If-finite ![of \lambda r. \, r = i ] by simp

lemma summable-single[simp, intro]: \( \text{summable} \, (\lambda r. \, \text{if} \, r = i \text{ then} \, f \, r \text{ else} \, 0 ) \)
  by (rule sums-summable) (rule sums-single)

context
  fixes \( f :: \text{nat} \Rightarrow \text{a}:: \{t2-space, \text{comm-monoid-add}\} \)
begin

lemma summable-sums[intro]: \( \text{summable} \, f \implies f \text{ sums} \, (\text{suminf} \, f) \)
by (simp add: summable-def sums-def suminf-def)
  (metis convergent-LIMSEQ-iff convergent-def lim-def)

lemma summable-LIMSEQ: summable f \implies (\lambda n. \sum i<n. f i) ----> suminf f
by (rule summable-sums [unfolded sums-def])

lemma sums-unique: f sums s \implies s = suminf f
by (metis limI suminf-eq-lim sums-def)

lemma sums-iff: f sums x \iff (summable f \land (suminf f = x))
by (metis summable-sums sums-summable sums-unique)

lemma suminf-finite:
  assumes N: finite N and f: \\( \forall n. n \notin N \implies f n = 0 \)
  shows suminf f = (\sum n\in N. f n)
  using sums-finite[OF assms, THEN sums-unique] by simp

end

lemma suminf-zero[simp]: suminf (\lambda n. 0 :: 'a::{t2-space, comm-monoid-add}) = 0
by (rule sums-zero [THEN sums-unique, symmetric])

102.3 Infinite summability on ordered, topological monoids

lemma sums-le:
  fixes f g :: nat \Rightarrow 'a::{ordered-comm-monoid-add, linorder-topology}
  shows \( \forall n. f n \leq g n \implies f sums s \implies g sums t \implies s \leq t \)
  by (rule LIMSEQ-le) (auto intro: setsum-mono simp: sums-def)

context
  fixes f :: nat \Rightarrow 'a::{ordered-comm-monoid-add, linorder-topology}
begin

lemma suminf-le:
  \[ \forall n. f n \leq g n; \text{summable } f; \text{summable } g \] \implies suminf f \leq suminf g
  by (auto dest: sums-summable intro: sums-le)

lemma setsum-le-suminf: summable f \implies \\( \forall m \geq n. 0 \leq f m \implies \text{setsum } f \{..<n\} \leq \text{suminf } f \)
  by (rule sums-le[OF - sums-If-finite-set summable-sums]) auto

lemma suminf-nonneg: summable f \implies \\( \forall n. 0 \leq f n \implies 0 \leq \text{suminf } f \)
  using setsum-le-suminf[of 0] by simp

lemma setsum-less-suminf2: summable f \implies \\( \forall m \geq n. 0 \leq f m \implies n \leq i \implies 0 < f i \implies \text{setsum } f \{..<n\} < \text{suminf } f \)
  using
  setsum-le-suminf[of Suc i]
  add-strict-increasing[of f i setsum f \{..<n\} setsum f \{..<i\}]
setsum-mono2[of `{..<i} `{..<n} f]
by (auto simp: less-imp-le ac-simps)

lemma setsum-less-suminf: summable f \(\Rightarrow\) \(\forall\ m \geq n.\ 0 < f m \Rightarrow\) setsum f `{..<n} < suminf f
using setsum-less-suminf2[of n n] by (simp add: less-imp-le)

lemma suminf-pos2: summable f \(\Rightarrow\) \(\forall\ n.\ 0 < f n \Rightarrow\) 0 < setsum f
using setsum-less-suminf2[of 0 i] by simp

lemma suminf-pos: summable f \(\Rightarrow\) \(\forall\ n.\ 0 < f n \Rightarrow\) 0 < suminf f
using suminf-pos2[of 0] by simp

lemma suminf-le-const: summable f \(\Rightarrow\) \(\forall\ n.\ \sum_{i < n} f i \leq x\) \(\Rightarrow\) suminf f \(\leq\) x
by (metis LIMSEQ-le-const2 summable-LIMSEQ)

lemma suminf-eq-zero-iff: summable f \(\Rightarrow\) \(\forall\ n.\ 0 < f n \Rightarrow\) suminf f = 0 \(\iff\) \(\forall\ n.\ f n = 0\)
proof
assume summable f suminf f = 0 and pos: \(\forall\ n.\ 0 < f n\)
then have f: \((\lambda n. \sum i<n. f i) \longrightarrow 0\)
using summable-LIMSEQ[of f] by simp
then have \(\forall i.\ (\sum n \in i.\ f n) \leq 0\)
proof (rule LIMSEQ-le-const)
fix i show \(\exists N.\ \forall n \geq N.\ (\sum n \in i.\ f n) \leq setsum f `{..<n}\)
using pos by (intro exI[of - Suc i] allI impI setsum-mono2) auto
qed
with pos show \(\forall n.\ f n = 0\)
by (auto intro!: antisym)
qed (metis suminf-zero fun-eq-iff)

lemma suminf-pos-if: summable f \(\Rightarrow\) \(\forall\ n.\ 0 \leq f n \Rightarrow\) 0 < suminf f \(\iff\) \(\exists i.\ 0 < f i\)
using setsum-less-suminf2[of 0] suminf-eq-zero-iff by (simp add: less-le)

end

lemma summable1-nonneg-bounded:
fixes f:: nat \Rightarrow\ 'a::{ordered-comm-monoid-add, linorder-topology, conditionally-complete-linorder}
assumes pos[simp]: \(\forall n.\ 0 \leq f n\) and le: \(\forall n.\ (\sum i<n. f i) \leq x\)
shows summable f
unfolding summable-def sums-def[abs_def]
proof (intro exI order-tendsto)
have [simp, intro]: bdd-above (range \((\lambda n. \sum i<n. f i)\))
using le by (auto simp: bdd-above-def)
{ fix a assume a < (SUP n. \(\sum i<n. f i)\)
then obtain n where a < (\(\sum i<n. f i)\)
by (auto simp add: less-cSUP-iff)
then have \( \forall m. n \leq m \Rightarrow a < \left( \sum_{i<n} f_i \right) \)
by (rule less-le-trans) (auto intro!: setsum-mono2)
then show eventually (\( \lambda n. a < \left( \sum_{i<n} f_i \right) \)) sequentially
by (auto simp: eventually_sequentially)
{ fix a assume (SUP n. \( \sum_{i<n} f_i \)) < a
moreover have \( \forall n. \left( \sum_{i<n} f_i \right) \leq (SUP n. \sum_{i<n} f_i) \)
by (auto intro: cSUP-upper)
ultimately show eventually (\( \lambda n. \left( \sum_{i<n} f_i \right) < a \)) sequentially
by (auto intro: le_less-trans simp: eventually_sequentially) }
qed

102.4 Infinite summability on real normed vector spaces

lemma sums-Suc-iff:
fixes f :: nat ⇒ 'a::real_normed_vector
shows (\( \lambda n. f (Suc n) \)) sums s ←→ f sums (s + f 0)
proof –
have f sums (s + f 0) ←→ (\( \lambda i. \sum_{j<i} f (Suc j) \)) s + f 0
by (subst LIMSEQ-Suc-iff) (simp add: sums_def)
also have ... ←→ (\( \lambda i. \sum_{j<i} f (Suc j) \)) + f 0) s + f 0
by (simp add: ac_simps setsum.reindex image_iff lessThan_Suc_eq_insert_0)
also have ... ←→ (\( \lambda n. f (Suc n) \)) sums s
proof
assume (\( \lambda i. \sum_{j<i} f (Suc j) \)) s + f 0
with tendsto_add[OF this tendsto_const, of - f 0]
show (\( \lambda i. f (Suc i) \)) sums s
by (simp add: sums_def)
qed (auto intro: tendsto_add tendsto_const simp: sums_def)
finally show thesis ..
qed

context
fixes f :: nat ⇒ 'a::real_normed_vector
begin

lemma sums-add: f sums a =⇒ g sums b =⇒ (\( \lambda n. f n + g n \)) sums (a + b)
unfolding sums_def by (simp add: setsum.distrib tendsto_add)

lemma summable-add: summable f =⇒ summable g =⇒ summable (\( \lambda n. f n + g n \))
unfolding summable_def by (auto intro: sums-add)

lemma suminf-add: summable f =⇒ summable g =⇒ suminf f + suminf g = 
(\( \sum n. f n + g n \))
by (intro sums-unique sums-add summable_sums)

lemma sums-diff: f sums a =⇒ g sums b =⇒ (\( \lambda n. f n - g n \)) sums (a - b)
unfolding sums_def by (simp add: setsum-subtractf tendsto_diff)
lemma summable-diff: summable f \rightarrow summable g \implies summable (\lambda n. f n - g n)
unfolding summable-def by (auto intro: sums-diff)

lemma suminf-diff: summable f \rightarrow summable g \implies suminf f - suminf g = 
(\sum n. f n - g n)
by (intro sums-unique sums-diff summable-sums)

lemma sums-minus: f sums a \implies (\lambda n. - f n) sums (- a)
unfolding sums-def by (simp add: setsum-negf tendsto-minus)

lemma summable-minus: summable f \implies summable (\lambda n. - f n)
unfolding summable-def by (auto intro: sums-minus)

lemma suminf-minus: summable f \implies suminf f - (\sum n. f n) = - (\sum n. f n)
by (intro sums-unique [symmetric] sums-minus summable-sums)

lemma sums-Suc: (\lambda i. f (Suc i)) sums l \implies f sums (l + f 0)
by (simp add: sums-Suc-iff)

lemma sums-iff-shift: (\lambda i. f (i + n)) sums s \iff f sums (s + (\sum i<n. f i))
proof (induct n arbitrary: s)
  case (Suc n)
  moreover have (\lambda i. f (Suc i + n)) sums s \iff (\lambda i. f (i + n)) sums (s + f n)
  by (subst sums-Suc-iff) simp
ultimately show ?case
  by (simp add: ac-simps)
qed simp

lemma summable-iff-shift: summable (\lambda n. f (n + k)) \iff summable f
by (metis diff-add-cancel summable-def sums-iff-shift[abs-def])

lemma sums-split-initial-segment: f sums s \implies (\lambda i. f (i + n)) sums (s - (\sum i<n. f i))
by (simp add: sums-iff-shift)

lemma summable-ignore-initial-segment: summable f \implies summable (\lambda n. f(n + k))
by (simp add: summable-iff-shift)

lemma suminf-minus-initial-segment: summable f \implies (\sum n. f (n + k)) = (\sum n. f n) - (\sum i<k. f i)
by (rule sums-unique[symmetric]) (auto simp: sums-iff-shift)

lemma suminf-split-initial-segment: summable f \implies suminf f = (\sum n. f(n + k)) + (\sum i<k. f i)
by (auto simp add: suminf-minus-initial-segment)

lemma suminf-exist-split:
fixes \( r :: \text{real} \) assumes \( 0 < r \) and \( \text{summable } f \)

shows \( \exists N. \forall n \geq N. \text{norm} \left( \sum i. f(i + n) \right) < r \)

proof –

from \( \text{LIMSEQ-D[OF summable-LIMSEQ[OF \langle \text{summable } f \rangle \cdot 0 < r \rangle]} \)

obtain \( N :: \text{nat} \) where \( \forall n \geq N. \text{norm} \left( \sum_{i} f(i + n) \right) < r \)

by auto

thus \( \text{thesis} \)

by \( (\text{auto simp: norm-minus-commute suminf-minus-initial-segment[OF \langle \text{summable } f \rangle \cdot 0 < r \rangle]} \)

qed

lemma \( \text{summable-LIMSEQ-zero: } \text{summable } f \Longrightarrow f \cdot\cdot\cdot > 0 \)

apply \( (\text{drule summable-iff-convergent [THEN iffD1]} \))

apply \( (\text{drule convergent-Cauchy}) \)

apply \( (\text{simp only: Cauchy-iff LIMSEQ-iff, safe}) \)

apply \( (\text{drule-tac } x = r \text{ in spec, safe}) \)

apply \( (\text{rule-tac } x = M \text{ in exI, safe}) \)

apply \( (\text{drule-tac } x = \text{Suc } n \text{ in spec, simp}) \)

apply \( (\text{drule-tac } x = n \text{ in spec, simp}) \)

done

end

context

fixes \( f :: 'i \\Rightarrow \text{nat} \Rightarrow 'a::\text{real-normed-vector} \) and \( I :: 'i \text{ set} \)

begin

lemma \( \text{sums-setsum: } (\forall i. i \in I \Longrightarrow (f(i) \text{ sums } (x \cdot i))) \Longrightarrow (\lambda n. \sum_{i \in I.} f(i \cdot n) \text{ sums } (\sum_{i \in I.} x \cdot i) \) \)

by \( (\text{induct } I \text{ rule: infinite-finite-induct}) \) \( (\text{auto intro: sums-add}) \)

lemma \( \text{suminf-setsum: } (\forall i. i \in I \Longrightarrow \text{summable } (f(i))) \Longrightarrow (\sum n. \sum_{i \in I.} f(i \cdot n) = (\sum_{i \in I.} \sum n. f(i \cdot n) \) \)

using \( \text{sums-unique[OF sums-setsum, OF summable-sums]} \) \( \text{by simp} \)

lemma \( \text{summable-setsum: } (\forall i. i \in I \Longrightarrow \text{summable } (f(i))) \Longrightarrow \text{summable } (\lambda n. \sum_{i \in I.} f(i \cdot n) \) \)

using \( \text{sums-summable[OF sums-setsum[OF \langle \text{summable } \sum f \rangle \cdot 0 < r \rangle]} \) .

end

lemma \( \text{(in bounded-linear) sums: } (\lambda n. X \cdot n) \text{ sums } a \Longrightarrow (\lambda n. f \cdot (X \cdot n)) \text{ sums } (f \cdot a) \)

unfolding \( \text{sums-def} \) \( \text{by} \) \( (\text{drule tendsto, simp only: setsum}) \)

lemma \( \text{(in bounded-linear) summable: } \text{summable } (\lambda n. X \cdot n) \Longrightarrow \text{summable } (\lambda n. f \cdot (X \cdot n)) \)

unfolding \( \text{summable-def} \) \( \text{by} \) \( (\text{auto intro: sums}) \)

lemma \( \text{(in bounded-linear) suminf: } \text{summable } (\lambda n. X \cdot n) \Longrightarrow f \cdot (\sum n. X \cdot n) = \)
\[ \sum n. f(X_n) \]

by (intro sums-unique sums summable-sums)

\textbf{lemmas} sums-of-real = bounded-linear.sums [OF bounded-linear-of-real]
\textbf{lemmas} summable-of-real = bounded-linear.summable [OF bounded-linear-of-real]
\textbf{lemmas} suminf-of-real = bounded-linear.suminf [OF bounded-linear-of-real]

\textbf{lemmas} sums-scaleR-left = bounded-linear.sums [OF bounded-linear-scaleR-left]
\textbf{lemmas} summable-scaleR-left = bounded-linear.summable [OF bounded-linear-scaleR-left]
\textbf{lemmas} suminf-scaleR-left = bounded-linear.suminf [OF bounded-linear-scaleR-left]

\textbf{lemmas} sums-scaleR-right = bounded-linear.sums [OF bounded-linear-scaleR-right]
\textbf{lemmas} summable-scaleR-right = bounded-linear.summable [OF bounded-linear-scaleR-right]
\textbf{lemmas} suminf-scaleR-right = bounded-linear.suminf [OF bounded-linear-scaleR-right]

102.5 Infinite summability on real normed algebras

context
fixes \( f :: \text{nat} \Rightarrow 'a::real-normed-algebra \)

begin

\textbf{lemma} sums-mult: \( f \) sums \( a \) \( \Longrightarrow \) \( (\lambda n. c * f n) \) sums \( (c * a) \)
by (rule bounded-linear.sums [OF bounded-linear-mult-right])

\textbf{lemma} summable-mult: summable \( f \) \( \Longrightarrow \) summable \( (\lambda n. c * f n) \)
by (rule bounded-linear.summable [OF bounded-linear-mult-right])

\textbf{lemma} suminf-mult: summable \( f \) \( \Longrightarrow \) suminf \( (\lambda n. c * f n) = c * \text{suminf} f \)
by (rule bounded-linear.suminf [OF bounded-linear-mult-right, symmetric])

\textbf{lemma} sums-mult2: \( f \) sums \( a \) \( \Longrightarrow \) \( (\lambda n. f n * c) \) sums \( (a * c) \)
by (rule bounded-linear.sums [OF bounded-linear-mult-left])

\textbf{lemma} summable-mult2: summable \( f \) \( \Longrightarrow \) summable \( (\lambda n. f n * c) \)
by (rule bounded-linear.summable [OF bounded-linear-mult-left])

\textbf{lemma} suminf-mult2: summable \( f \) \( \Longrightarrow \) suminf \( f * c = (\sum n. f n * c) \)
by (rule bounded-linear.suminf [OF bounded-linear-mult-left])

end

102.6 Infinite summability on real normed fields

context
fixes \( c :: 'a::real-normed-field \)

begin

\textbf{lemma} sums-divide: \( f \) sums \( a \) \( \Longrightarrow \) \( (\lambda n. f n / c) \) sums \( (a / c) \)
by (rule bounded-linear.sums [OF bounded-linear-divide])
lemma summable-divide: summable $f \implies$ summable $(\lambda n. f n / c)$
  by (rule bounded-linearsummable [OF bounded-linear-divide])

lemma suminf-divide: summable $f \implies$ suminf $(\lambda n. f n / c) = \text{suminf} f / c$
  by (rule bounded-linear,suminf [OF bounded-linear-divide, symmetric])

Sum of a geometric progression.

lemma geometric-sums: norm $c < 1 \implies (\lambda n. c^n)$ sums $(1 / (1 - c))$
proof
  assume less-1: norm $c < 1$
  hence neq-1: $c \neq 1$ by auto
  hence neq-0: $c - 1 \neq 0$ by simp
  from less-1 have lim-0: $(\lambda n. c^n) \lim > 0$
    by (rule LIMSEQ-power-zero)
  hence $(\lambda n. c^n / (c - 1 - 1 / (c - 1))) \lim > 0 / (c - 1 - 1 / (c - 1))$
    using neq-0 by (intro tendsto-intros)
  hence $(\lambda n. (c^n - 1) / (c - 1)) \lim > 1 / (1 - c)$
    by (simp add: nonzero-minus-divide-right [OF neq-0] diff-divide-distrib)
  thus $(\lambda n. c^n)$ sums $(1 / (1 - c))$
    by (simp add: sums-def geometric-sum neq-1)
qed

lemma summable-geometric: norm $c < 1 \implies$ summable $(\lambda n. c^n)$
  by (rule geometric-sums [THEN sums-summable])

lemma suminf-geometric: norm $c < 1 \implies$ suminf $(\lambda n. c^n) = 1 / (1 - c)$
  by (rule sums-unique[symmetric]) (rule geometric-sums)

end

lemma power-half-series: $(\lambda n. (1/2::real)^\text{Suc n})$ sums 1
proof
  have 2: $(\lambda n. (1/2::real)^n)$ sums 2 using geometric-sums [of 1/2::real]
    by auto
  have $(\lambda n. (1/2::real)^\text{Suc n}) = (\lambda n. (1/2)^n / 2)$
    by simp
  thus thesis using sums-divide [OF 2, of 2]
    by simp
qed

102.7 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison)

lemma summable-Cauchy:
  fixes $f$ :: nat $\Rightarrow$ 'a::banach
  shows summable $f \longleftrightarrow (\forall e>0. \exists N. \forall m \geq N. \forall n. \text{norm} (\text{setsum} f \{m..<n\}) < e)$
  apply (simp only: summable-iff-convergent Cauchy-convergent-iff [symmetric]
  Cauchy-iff, safe)
apply (drule spec, drule (1) mp)
apply (erule exE, rule-tac x=M in exI, clarify)
apply (rule-tac x=m and y=n in linorder-le-cases)
apply (frule (1) order-trans)
apply (drule-tac x=n in spec, drule (1) mp)
apply (drule-tac x=m in spec, drule (1) mp)
apply (simp-all add: setsum-diff [symmetric])
apply (drule spec, drule (1) mp)
apply (erule exE, rule-tac x=N in exI, clarify)
apply (rule-tac x=m and y=n in linorder-le-cases)
apply (subst norm-minus-commute)
apply (simp-all add: setsum-diff [symmetric])
done

context
fixes f :: nat ⇒ 'a::banach
begin

Absolute convergence imples normal convergence

lemma summable-norm-cancel: summable (λn. norm (f n)) ⇒ summable f
apply (simp only: summable-Cauchy, safe)
apply (drule-tac x=e in spec, safe)
apply (rule-tac x=N in exI, safe)
apply (drule-tac x=m in spec, safe)
apply (rule order-le-less-trans [OF norm-setsum])
apply (rule order-le-less-trans [OF abs-ge-self])
apply simp
done

lemma summable-norm: summable (λn. norm (f n)) ⇒ norm (suminf f) ≤ (∑ n. norm (f n))
by (auto intro: LIMSEQ-le tendsto-norm summable-norm-cancel summable-LIMSEQ norm-setsum)

Comparison tests

lemma summable-comparison-test: ∃ N. ∀ n≥N. norm (f n) ≤ g n ⇒ summable g ⇒ summable f
apply (simp add: summable-Cauchy, safe)
apply (drule-tac x=e in spec, safe)
apply (rule-tac x = N + Na in exI, safe)
apply (rotate-tac 2)
apply (drule-tac x = m in spec)
apply (auto, rotate-tac 2, drule-tac x = n in spec)
apply (rule-tac y = ∑ k=m..<n. norm (f k) in order-le-less-trans)
apply (rule norm-setsum)
apply (rule-tac y = setsum g {m..<n} in order-le-less-trans)
apply (auto intro: setsum-mono simp add: abs-less-iff)
done
Lemma `summable-comparison-test`: `summable g ==> (∀n. n ≥ N ==> norm(f n) ≤ g n) ==> summable f`

by (rule summable-comparison-test) auto

102.8 The Ratio Test

Lemma `summable-ratio-test`:

assumes `c < 1 && n. n ≥ N ==> norm (f (Suc n)) ≤ c * norm (f n)`

shows `summable f`

proof

cases

assume `0 < c`

show `summable f`

proof

(rule summable-comparison-test)

show `∃N'. ∀n≥N'. norm (f n) ≤ (norm (f N) / (c ^ N)) * c ^ n`

proof (intro exI allI impI)

fix `n` assume `N ≤ n` then show `norm (f n) ≤ (norm (f N) / (c ^ N)) * c ^ n`

proof (induct rule: inc-induct)

case (step m)

moreover have `norm (f (Suc m)) / c ^ Suc m * c ^ n ≤ norm (f m) / c ^ m`

using `0 < c : (c < 1)`

assms (2)[OF `N ≤ m`]

by (simp add: field-simps)

ultimately show `?case`

by simp

qed

next

assume `c: ~ 0 < c`

{ fix `n` assume `n ≥ N`

then have `norm (f (Suc n)) ≤ c * norm (f n)`

by fact

also have `... ≤ 0`

using `c` by (simp add: not-less mult-nonpos-nonneg)

finally have `f (Suc n) = 0`

by auto }

then show `summable f`

by (intro sums-summable[OF sums-finite, of `{.. Suc N}`]) (auto simp: not-le Suc-less-eq2)

qed

end

Relations among convergence and absolute convergence for power series.

Lemma `abel-lemma`:

fixes `a :: nat ⇒ 'a::real-normed-vector`

assumes `r: 0 ≤ r and r0: r < r0 and M: ∀n. norm (a n) * r0 ^ n ≤ M`

...
THEORY “Series”

shows summable \((\lambda n. \text{norm} (a n) * r^n)\)
proof (rule summable-comparison-test)
show summable \((\lambda n. M * (r / r0)^n)\)
using assms
by (auto simp add; summable-mult summable-geometric)
next
fix \(n\)
show \(\text{norm} \ (\text{norm} (a n) * r^n) \leq M \times (r / r0)^n\)
using \(r \ r0 \ M\) [of \(n\)]
apply (auto simp add: abs-mult field-simps power-divide)
apply (cases \(r = 0\), simp)
apply (cases \(n\), auto)
done
qed

Summability of geometric series for real algebras

lemma complete-algebra-summable-geometric:
fixes \(x :: 'a:: real-normed-algebra-1, \text{banach}\)
shows \(\text{norm} x < 1 \implies \text{summable} \ ((\lambda n. \text{norm} (x^n))\)
proof (rule summable-comparison-test)
show \(\exists N. \forall n \geq N. \text{norm} (x^n) \leq \text{norm} x \times n\)
by (simp add: norm-power-ineq)
show \(\text{norm} x < 1 \implies \text{summable} \ ((\lambda n. \text{norm} (x^n))\)
by (simp add: summable-geometric)
qed

102.9 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm

lemma setsum-triangle-reindex:
fixes \(n :: \text{nat}\)
shows \((\Sigma (i,j) \in \{(i,j). \ i + j < n\ \}. \ f \ i \ j) = (\Sigma k < n. \ \Sigma i \leq k. \ f \ i \ (k-i))\)
apply (simp add: setsum.Sigma)
apply (rule setsum.reindex-bij-witness[where \(j=\lambda (i,j). \ (i+j, i) \ \text{and} \ i=\lambda (k, i). \ (i, k - i)]\])
apply auto
done

lemma Cauchy-product-sums:
fixes \(a \ b :: \text{nat} \Rightarrow 'a:: real-normed-algebra, \text{banach}\)
assumes \(a:: \text{summable} \ ((\lambda k. \text{norm} (a k))\)
assumes \(b:: \text{summable} \ ((\lambda k. \text{norm} (b k))\)
shows \((\lambda k. \Sigma i \leq k. \ a \ i \times b \ (k-i)) \ \text{sums} ((\Sigma k. \ a \ k) * (\Sigma k. \ b \ k))\)
proof –
let \(?S1 = \lambda n::\text{nat}. \ \{.. < n\} \times \{.. < n\}\)
let \(?S2 = \lambda n::\text{nat}. \ \{(i,j). \ i + j < n\}\)
have \(\text{S1-mono: } \forall m. \ m \leq n \implies ?S1 \ m \subseteq ?S1 \ n\ \text{by auto}\)
have $S2$-le-$S1$: $\forall n. \exists S2 n \subseteq S1 n$ by auto
have $S1$-le-$S2$: $\forall n. \exists S1 (n \div 2) \subseteq S2 n$ by auto
have finite-$S1$: $\forall n. \text{finite}(\exists S1 n)$ by simp
with $S2$-le-$S1$ have finite-$S2$: $\forall n. \text{finite}(\exists S2 n)$ by (rule finite-subset)

let $\alpha = \lambda(i,j). a i \cdot b j$
let $\beta = \lambda(i,j). \text{norm}(a i) \cdot \text{norm}(b j)$
have f-nonneg: $\lambda x. \theta \leq \beta x$ by (auto)
hence unfolding real-norm-def
by (simp only: abs-of-nonneg setsum-nonneg [rule-format])

have $(\lambda n. (\sum k<n. a k) \cdot (\sum k<n. b k)) \longrightarrow (\sum k a k) \cdot (\sum k b k)$
by (intro tendsto-mult summable-LIMSEQ summable-norm-cancel [OF a] summable-norm-cancel [OF b])
hence $f$: $(\lambda n. \text{setsum} \ (\exists S1 n)) \longrightarrow (\sum k a k) \cdot (\sum k b k)$
by (simp only: setsum-product setsum Sigma [rule-format] finite-lessThan)

have $(\lambda n. (\sum k<n. \text{norm}(a k)) \cdot (\sum k<n. \text{norm}(b k))) \longrightarrow (\sum k \text{norm}(a k)) \cdot (\sum k \text{norm}(b k))$
using $a b$ by (intro tendsto-mult summable-LIMSEQ)
hence $(\lambda n. \text{setsum} \ (\exists S1 n)) \longrightarrow (\sum k \text{norm}(a k)) \cdot (\sum k \text{norm}(b k))$
by (simp only: setsum-product setsum Sigma [rule-format] finite-lessThan)

have convergent $(\lambda n. \text{setsum} \ (\exists S1 n))$
by (rule convergent$I$)
hence Cauchy: Cauchy $(\lambda n. \text{setsum} \ (\exists S1 n))$
by (rule convergent-Cauchy)

have $Z\text{func} (\lambda n. \text{setsum} \ (\exists S1 n - \exists S2 n))$ sequentially
proof (rule $Z\text{func} I$, simp only: eventually-sequentially norm-setsum-f)
fix $r :: \text{real}$
assume $r: 0 < r$
from CauchyD $(\text{OF Cauchy } r)$ obtain $N$
where $\forall m\geq N. \forall n\geq N. \text{norm}(\text{setsum} \ (\exists S1 m - \exists S2 n)) < r$

... hence $\forall n. \exists N. \forall n\geq N. \text{setsum} \ (\exists S1 n - \exists S2 n) < r$
proof (intro exI allI impl)
fix $n$ assume $2 * N \leq n$
hence $n: N \leq n \div 2$ by simp
have setsum $ (\exists S1 n - \exists S2 n) \leq \text{setsum} \ (\exists S1 n - \exists S1 (n \div 2))$
by (intro setsum-mono2 finite-Diff finite-S1 f-nonneg
  Diff-mono subset-refl S1-le-S2)
also have ... $\leq r$
using $n \div 2$-dividend by (rule $N$)
finally show setsum $ (\exists S1 n - \exists S2 n) < r$.
qed
qed

hence \( \text{Zfun} (\lambda n. \text{setsum} ?g (\text{?S1 n} - \text{?S2 n})) \) sequentially

apply (rule \text{Zfun-le} [rule-format])

apply (simp only: \text{norm-setsum-f})

apply (rule \text{order-trans} [OF \text{norm-setsum} \text{setsum-mono}])

apply (auto simp add: \text{norm-mult-ineq})

done

hence \( 2: (\lambda n. \text{setsum} ?g (\text{?S1 n}) - \text{setsum} ?g (\text{?S2 n})) \rightarrow 0 \)

unfolding \text{tendsto-Zfun-iff} \text{diff-0-right}

by (simp only: \text{setsum-diff} \text{finite-S1 S2-le-S1})

\[
\text{with} \: \text{1} \: \text{have} (\lambda n. \text{setsum} ?g (\text{?S2 n})) \rightarrow (\sum k. a k) \cdot (\sum k. b k)
\]

by (rule \text{LIMSEQ-diff-approach-zero2})

thus ?thesis by (simp only: \text{sums-def} \text{setsum-triangle-reindex})

qed

lemma \text{Cauchy-product}:

fixes \( a, b :: \text{nat} \Rightarrow 'a::{\text{real-normed-algebra,banach}} \)

assumes \( a :: \text{summable} (\lambda k. \text{norm} (a k)) \)

assumes \( b :: \text{summable} (\lambda k. \text{norm} (b k)) \)

shows \((\sum k. a k) \cdot (\sum k. b k) = (\sum k. \sum i \leq k. a i \cdot b (k - i))\)

using \( a, b \)

by (rule \text{Cauchy-product-sums} [THEN \text{sums-unique}])

102.10 Series on reals

lemma \text{summable-norm-comparison-test}:

\( \exists N. \forall n \geq N. \text{norm} (f n) \leq g n \Rightarrow \text{summable} g \Rightarrow \text{summable} (\lambda n. \text{norm} (f n)) \)

by (rule \text{summable-comparison-test}) auto

lemma \text{summable-rabs-comparison-test}:

\( [\exists N. \forall n \geq N. |f n| \leq g n; \text{summable} g] \Rightarrow \text{summable} (\lambda n. |f n :: \text{real}|) \)

by (rule \text{summable-comparison-test}) auto

lemma \text{summable-rabs-cancel}:

\( \text{summable} (\lambda n. |f n :: \text{real}|) \Rightarrow \text{summable} f \)

by (rule \text{summable-norm-cancel}) simp

lemma \text{summable-rabs}:

\( \text{summable} (\lambda n. |f n :: \text{real}|) \Rightarrow |\text{suminf} f| \leq (\sum n. |f n|) \)

by (fold \text{real-norm-def}) (rule \text{summable-norm})

end

103 Deriv: Differentiation

theory \text{Deriv}
imports \text{Limits}
begin
103.1 Frechet derivative

definition
\textit{has-derivative} :: (\textit{a}::real-normed-vector \Rightarrow \textit{b}::real-normed-vector) \Rightarrow (\textit{a} \Rightarrow \textit{b}) \\
\Rightarrow \textit{a} \text{ filter} \Rightarrow \text{bool} \\
\textbf{(infix)} (\textit{has}'-derivative) 50)

where
\begin{align*}
(f \text{ has-derivative } f') F \leftrightarrow \\
\text{bounded-linear } f' \land \\
\text{(} (\lambda y. ((f y - f (\text{Lim } F (\lambda x. x))) - f' (y - \text{Lim } F (\lambda x. x))) / R \text{ norm } (y - \text{Lim } F (\lambda x. x))) \rightarrow 0 \text{)} F
\end{align*}

Usually the filter \(F\) is \textit{at} \(x\) \textit{within} \(s\). \((f \text{ has-derivative } D) \text{ (at } x \text{ within } s)\) means: \(D\) is the derivative of function \(f\) \textit{at} point \(x\) \textit{within} the set \(s\). Where \(s\) is used to express left or right sided derivatives. In most cases \(s\) is either a variable or \texttt{UNIV}.

\textbf{lemma} \textit{has-derivative-eq-rhs}: \((f \text{ has-derivative } f') F \implies f' = g' \implies (f \text{ has-derivative } g') F \\
\textbf{by simp}

definition
\textit{has-field-derivative} :: (\textit{a}::real-normed-field \Rightarrow \textit{a}) \Rightarrow \textit{a} \Rightarrow \textit{a} \text{ filter} \Rightarrow \text{bool} \\
\textbf{(infix)} (\textit{has}'-field'-'derivative) 50)

where
\begin{align*}
(f \text{ has-field-derivative } D) F \leftrightarrow (f \text{ has-derivative } (\lambda x. x) \ast D) F
\end{align*}

\textbf{lemma} \textit{DERIV-cong}: \((f \text{ has-field-derivative } X) F \implies X = Y \implies (f \text{ has-field-derivative } Y') F \\
\textbf{by simp}

definition
\textit{has-vector-derivative} :: (real \Rightarrow \textit{b}::real-normed-vector) \Rightarrow \textit{b} \Rightarrow \textit{a} \text{ filter} \Rightarrow \text{bool} \\
\textbf{(infix)} (\textit{has}'-vector'-'derivative) 50)

where
\begin{align*}
(f \text{ has-vector-derivative } f') \text{ net} \leftrightarrow (f \text{ has-derivative } (\lambda x. x) \ast R f') \text{ net}
\end{align*}

\textbf{lemma} \textit{has-vector-derivative-eq-rhs}: \((f \text{ has-vector-derivative } X) F \implies X = Y \implies (f \text{ has-vector-derivative } Y') F \\
\textbf{by simp}

\textbf{ML} \langle
\textbf{structure} Derivative-Intros = Named-Thms \\
\textbf{(}
\textbf{val} name = @\{binding derivative-intros\} \\
\textbf{val} description = structural introduction rules for derivatives \\
\textbf{)}
\rangle
setup "
let
    eval eq-thms = [@{thm has-derivative-eq-rhs}, @{thm DERIV-cong}, @{thm has-vector-derivative-eq-rhs}]
    fun eq-rule thm = get-first (try (fn eq-thm => eq-thm OF [thm])) eq-thms
in
    Derivative-Intros.setup #>
    Global-Theory.add-thms-dynamic (@
      @{binding derivative-eq-intros}, map-filter eq-rule o Derivative-Intros.get o Context.proof-of)
end;
"

The following syntax is only used as a legacy syntax.

abbreviation (input)
  FDERIV :: (′a::real-normed-vector ⇒ ′b::real-normed-vector) ⇒ ′a ⇒ (′a ⇒ ′b)
⇒ bool
  ((FDERIV (-)/ (-)/ :-) [1000, 1000, 60] 60)
where
  FDERIV f x := f' ≡ (f has-derivative f') (at x)

lemma has-derivative-bounded-linear: (f has-derivative f') F ⇒ bounded-linear f'
  by (simp add: has-derivative-def)
lemma has-derivative-linear: (f has-derivative f') F ⇒ linear f'
  using bounded-linear.linear[OF has-derivative-bounded-linear].
lemma has-derivative-ident[derivative-intros, simp]: ((λx. x) has-derivative (λx. x)) F
  by (simp add: has-derivative-def tendsto-const)
lemma has-derivative-const[derivative-intros, simp]: ((λx. c) has-derivative (λx. 0)) F
  by (simp add: has-derivative-def tendsto-const)
lemma (in bounded-linear) bounded-linear: bounded-linear f ..
lemma (in bounded-linear) has-derivative:
  (g has-derivative g') F ⇒ ((λx. f (g x)) has-derivative (λx. f (g' x))) F
  using assms unfolding has-derivative-def
  apply safe
  apply (erule bounded-linear-compose [OF bounded-linear])
  apply (drule tendsto)
  apply (simp add: scaleR diff add zero)
  done
lemmas has-derivative-scaleR-right [derivative-intros] =
bounded-linear.has-derivative [OF bounded-linear-scaleR-right]

lemmas has-derivative-scaleR-left [derivative-intros] =
bounded-linear.has-derivative [OF bounded-linear-scaleR-left]

lemmas has-derivative-mult-right [derivative-intros] =
bounded-linear.has-derivative [OF bounded-linear-mult-right]

lemmas has-derivative-mult-left [derivative-intros] =
bounded-linear.has-derivative [OF bounded-linear-mult-left]

lemma has-derivative-add[simp, derivative-intros]:
assumes f: (f has-derivative f') F and g: (g has-derivative g') F
shows ((λx. f x + g x) has-derivative (λx. f' x + g' x)) F
unfolding has-derivative-def
proof safe
let ?x = Lim F (λx. x)
let ?D = λ f y. ((f y - f ?x) - f' (y - ?x)) / R norm (y - ?x)
have ((λx. ?D f f' x + ?D g g' x) ----> (0 + 0)) F
  using f by (intro tendsto-add) (auto simp: has-derivative-def)
then show (?D (λx. f x + g x) (λx. f' x + g' x) ----> 0) F
  by (simp add: field-simps scaleR-add-right scaleR-diff-right)
qed (blast intro: bounded-linear-add f g has-derivative-bounded-linear)

lemma has-derivative-setsum[simp, derivative-intros]:
assumes f: ∃i. i ∈ I ⇒ (f i has-derivative f' i) F
shows ((λx. ∑ i∈I. f i x) has-derivative (λx. ∑ i∈I. f' i x)) F
proof cases
  assume finite I from this f show ?thesis
    by (simp add: f)
qed simp

lemma has-derivative-minus[simp, derivative-intros]: (f has-derivative f') F ⇒
((λx. f x) has-derivative (λx. f' x)) F
using has-derivative-scaleR-right[of f f' F -1] by simp

lemma has-derivative-diff[simp, derivative-intros]:
(f has-derivative f') F ⇒ (g has-derivative g') F ⇒ (λx. f x - g x) has-derivative
(λx. f' x - g' x) F
  by (simp only: diff-conv-add-uminus has-derivative-add has-derivative-minus)

lemma has-derivative-at-within:
  (f has-derivative f') (at x within s) ⟷
  (bounded-linear f' ∧ ((λy. (f y - f x) - f' (y - x)) / R norm (y - x) ----> 0) (at x within s))
  by (cases at x within s = bot) (simp add: has-derivative-def Lim-ident-at)

lemma has-derivative-iff-norm:
  (f has-derivative f') (at x within s) ⟷

(bounded-linear f' ∧ ((λy. norm ((f y − f x) − f' (y − x)) / norm (y − x))) −−−> 0) (at x within s)

using tendsto-norm-zero-iff[of - at x within s, where 'b='b, symmetric]
by (simp add: has-derivative-at-within divide-inverse ac-simps)

lemma has-derivative-at:
(f has-derivative D) (at x) −→ (bounded-linear D ∧ (λh. norm (f (x + h) − f x − D h) / norm h) −− 0 −−−> 0)
unfolding has-derivative-iff-norm LIM-offset-zero-iff[of - x] by simp

lemma field-has-derivative-at:
fixes x :: 'a::real-normed-field
shows (f has-derivative op * D) (at x) −→ (λh. (f (x + h) − f x) / h) −− 0 −−−> D
apply (unfold has-derivative-at)
apply (simp add: bounded-linear-mult-right)
apply (simp cong: LIM-cong add: nonzero-norm-divide [symmetric])
apply (subst diff-divide-distrib)
apply (subst times-divide-eq-left [symmetric])
apply (simp cong: LIM-cong)
apply (simp add: tendsto-norm-zero-iff LIM-zero-iff)
done

lemma has-derivativeI:
bounded-linear f' −→ ((λy. ((f y − f x) − f' (y − x)) / R norm (y − x)) −−−> 0) (at x within s) −→
(f has-derivative f') (at x within s)
by (simp add: has-derivative-at-within)

lemma has-derivativeI-sandwich:
assumes e: 0 < e and bounded: bounded-linear f'
and sandwich: (∀y. y ∈ s ⇒ y ≠ x ⇒ dist y x < e ⇒ norm ((f y − f x) − f' (y − x)) / norm (y − x) ≤ H y)
and (H −−−> 0) (at x within s)
shows (f has-derivative f') (at x within s)
unfolding has-derivative-iff-norm
proof safe
show ((λy. norm ((f y − f x) − f' (y − x)) / norm (y − x)) −−−> 0) (at x within s)
proof (rule tendsto-sandwich[where f=λx. 0])
show (H −−−> 0) (at x within s) by fact
show eventually (λn. norm ((f n − f x) − f' (n − x)) / norm (n − x) ≤ H n)
(at x within s)
unfolding eventually-at using e sandwich by auto
qed (auto simp: le-divide-eq tendsto-const)
qed fact

lemma has-derivative-subset: (f has-derivative f') (at x within s) −→ t ⊆ s −→ (f has-derivative f') (at x within t)
proof (auto simp add: has-derivative-iff-norm intro: tendsto-within-subset)

lemmas has-derivative-within-subset = has-derivative-subset

103.2 Continuity

lemma has-derivative-continuous:
assumes f: \((f \text{ has-derivative } f')\) (at \(x\) within \(s\))
shows continuous (at \(x\) within \(s\)) \(f\)

proof –
from \(f\) interpret \(F\): bounded-linear \(f'\) by (rule has-derivative-bounded-linear)
note \(F\).tendsto[\(F\).tendsto-intros]
let \(?L = \lambda f. (f \ ----\> 0)\) (at \(x\) within \(s\))
have \(?L (\lambda y. \text{ norm } ((f y - f x) - f' (y - x)) / \text{ norm } (y - x))\)
using \(f\) unfolding has-derivative-iff-norm by blast
then have \(?L (\lambda y. \text{ norm } ((f y - f x) - f' (y - x)) / \text{ norm } (y - x) * \text{ norm } (y - x))\) (is \(?m\))
  by (rule tendsto-mult-zero) (auto intro!: tendsto-eq-intros)
also have \(?m \ ----\> \?L (\lambda y. \text{ norm } ((f y - f x) - f' (y - x)))\)
  by (intro filterlim-cong) (simp-all add: eventually-at-filter)
finally have \(?L (\lambda y. (f y - f x) - f' (y - x))\)
  by (rule tendsto-norm-zero-cancel)
then have \(?L (\lambda y. (f y - f x) - f' (y - x)) + f' (y - x))\)
  by (rule tendsto-eq-intros) (auto intro!: tendsto-eq-intros simp: \(F\).zero)
then have \(?L (\lambda y. f y - f x)\)
  by simp
from tendsto-add[OF this tendsto-const, of \(f x\)] show \(?thesis\)
  by (simp add: continuous-within)

qed

103.3 Composition

lemma tendsto-at-iff-tendsto-nhds-within: \(f x = y \Longrightarrow (f \ ----\> y)\) (at \(x\) within \(s\))
\((f \ ----\> y) (\text{nhds } x) (\text{principal } s)\)
 unfolding tendsto-def eventually-inf-principal eventually-at-filter
  by (intro ext all-cong imp-cong) (auto elim!: eventually-elim1)

lemma has-derivative-in-compose:
assumes f: \((f \text{ has-derivative } f')\) (at \(x\) within \(s\))
assumes g: \((g \text{ has-derivative } g') (at (f x) within (f's))\)
shows \((\lambda x. g (f x)) \text{ has-derivative } (\lambda x. g' (f' x))\) (at \(x\) within \(s\))

proof –
from \(f\) interpret \(F\): bounded-linear \(f'\) by (rule has-derivative-bounded-linear)
from \(g\) interpret \(G\): bounded-linear \(g'\) by (rule has-derivative-bounded-linear)
from \(F\).bounded obtain \(kF\) where \(kF: \\langle x. \text{ norm } (f' x) \rangle \leq \text{ norm } x * kF\) by fast
from \(G\).bounded obtain \(kG\) where \(kG: \\langle x. \text{ norm } (g' x) \rangle \leq \text{ norm } x * kG\) by fast
 note \(G\).tendsto[\(G\).tendsto-intros]

let \(?L = \lambda f. (f \ ----\> 0)\) (at \(x\) within \(s\))
let \( ?D = \lambda f. f' \cdot x \cdot y \cdot (f y - f x) - f'(y - x) \)
let \( ?N = \lambda f. f' \cdot x \cdot y \cdot \text{norm} \left( ?D f f' f x y \right) / \text{norm} (y - x) \)
let \( ?g f = \lambda x. g (f x) \) and \( ?g f' = \lambda x. g' (f' x) \)
def \( \text{Ng} \equiv \lambda y. ?N \cdot g' (f x) (f y) \)
def \( \text{Ng} \equiv \lambda y. ?N \cdot g' (f x) (f y) \)

show \( ?\text{thesis} \)

proof (rule has-derivativeI-sandwich[of \( 1 \)])
  show bounded-linear \((\lambda x. g' (f' x))\)
  using \( g \) by (blast intro: bounded-linear-compose has-derivative-bounded-linear)

next

fix \( y : 'a \) assume neq: \( y \neq x \)

have \( ?N \cdot g f \cdot g f' (x y) = \text{norm} \left( g' \left( ?D f f' x y \right) + ?D g g' (f x) (f y) \right) / \text{norm} (y - x) \)
  by (simp add: G.diff G.add field-simps)

also have \( \ldots \leq \text{norm} \left( g' \left( ?D f f' x y \right) \right) / \text{norm} (y - x) + \text{Ng} y * \left( \text{norm} (f y - f x) / \text{norm} (y - x) \right) \)
  by (simp add: add-divide-distrib[symmetric] divide-right-mono norm-triangle-ineq 
  G.zero \( \text{Ng-def} \) 

also have \( \ldots \leq \text{Ng} y * \text{kG} + \text{Ng} y * \left( \text{Ng} y + \text{kF} \right) \)

proof (intro add-mono mult-left-mono)
  have \( \text{norm} (f y - f x) = \text{norm} \left( ?D f f' x y + f'(y - x) \right) \)
  by simp

  also have \( \ldots \leq \text{norm} \left( ?D f f' x y \right) + \text{norm} (f'(y - x)) \)
  by (rule norm-triangle-ineq)

  also have \( \ldots \leq \text{norm} \left( ?D f f' x y \right) + \text{norm} (y - x) * \text{kF} \)

  using \( \text{kF} \) by (intro add-mono) simp

  finally show \( \text{norm} (f y - f x) / \text{norm} (y - x) \leq \text{Ng} y + \text{kF} \)

  by (simp add: neq \( \text{Ng-def field-simps} \)

qed (insert \( \text{kG}, \text{simp-all add: \( \text{Ng-def} \text{Ng-def neq zero-le-divide-iff field-simps} \) 

finally show \( ?N \cdot g f \cdot g f' x y \leq \text{Ng} y * \text{kG} + \text{Ng} y * \left( \text{Ng} y + \text{kF} \right) \).

next

have \( \text{[tendsto-intros]}: \text{\( ?L \text{Ng} \) 

  using \( f \) unfolding has-derivative-iff-norm \( \text{Ng-def} \) ..

from \( \text{have} \) \( (f \longrightarrow f x) \) (at \( x \) within \( s \))
by (blast intro: has-derivative-continuous continuous-within [THEN iffD1])

then have \( f' : LIM x \) at \( x \) within \( s \) \. \( f x :> \) inf (nhds \( f x \)) \( \text{principal \( f's \)\) 

unfolding filterlim-def
by (simp add: eventually-filtermap eventually-at-filter le-principal)

have \( (\text{\( ?N \cdot g \cdot g' (f x) \longrightarrow 0 \) (at \( f x \) within \( f's \)})

using \( g \) unfolding has-derivative-iff-norm ..

then have \( g' : (\text{\( ?N \cdot g \cdot g' (f x) \longrightarrow 0 \) (inf \( \text{nhds \( f x \}) \) \( \text{principal \( f's \)\}) \)

by (rule tendsto-at-iff-tendsto-nhds-within[THEN iffD1, rotated]) simp

have \( \text{[tendsto-intros]}: \text{\( ?L \text{Ng} \) 

unfolding \( \text{Ng-def} \) by (rule filterlim-compose[OF \( g f f' \)]

show \( (\text{\( \lambda y. \text{Ng} y * \text{kG} + \text{Ng} y * \left( \text{Ng} y + \text{kF} \right) \longrightarrow 0 \) (at \( x \) within \( s \))

by (intro tendsto-eq-intros) auto
lemma has-derivative-compose:
\( (f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \Rightarrow (g \text{ has-derivative } g') \text{ (at } x \text{ within } s) \)
\( ((\lambda x. f (f x)) \text{ has-derivative } (\lambda x. g' ((f x))) \text{ (at } x \text{ within } s) \)
by (blast intro: has-derivative-in-compose has-derivative-subset)

lemma (in bounded-bilinear) FDERIV:
\( \text{assumes } f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \text{ and } g: (g \text{ has-derivative } g') \text{ (at } x \text{ within } s) \)
\( \text{shows } ((\lambda x. f x ** g x) \text{ has-derivative } (\lambda h. f x ** g' h + f' h ** g x)) \text{ (at } x \text{ within } s) \)
proof –
from bounded-linear bounded [OF has-derivative-bounded-linear [OF f]] obtain KF where norm-F: \( \forall x. \text{norm } (f' x) \leq \text{norm } x * KF \) by fast

from pos-bounded obtain K where K: \( 0 < K \) and norm-prod:
\( \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K \) by fast
let \( ?D = \lambda y. f y - f x - f'(y - x) \)
let \( ?N = \lambda y. \text{norm } (f f'(y)) / \text{norm } (y - x) \)
def Ng == ?N g' and Nf == ?N f'
let \( ?\text{fun1} = \lambda y. \text{norm } (f y ** g y - f x ** g x - (f x ** g' (y - x) + f'(y - x) ** g x)) / \text{norm } (y - x) \)
let \( ?\text{fun2} = \lambda y. \text{norm } (f x) * Ng g * Nf g * \text{norm } (g y) * K + KF * \text{norm } (g y - g x) * K \)
let \( ?F = \text{at } x \text{ within } s \)

show ?thesis
proof (rule has-derivativeI-sandwich[of 1])
show bounded-linear (\( \lambda h. f x ** g' h + f' h ** g x \))
by (intro bounded-linear-add
bounded-linear-compose [OF bounded-linear-right] bounded-linear-compose
[OF bounded-linear-left]
has-derivative-bounded-linear [OF g] has-derivative-bounded-linear [OF f])
next
from g have \( (g ----> g x) \) \( ?F \)
by (intro continuous-within[THEN iffD1] has-derivative-continuous)
moreover from f g have \( (Nf ----> 0) \) \( ?F \) (\( Ng ----> 0 \)) \( ?F \)
by (simp-all add: has-derivative-iff-norm Ng-def Nf-def)
ultimately have \( (?\text{fun2} ----> 0) \) \( ?F \)
by (intro tendsto-intros) (simp-all add: LIM-zero-iff)
then show \( (?\text{fun2} ----> 0) \) \( ?F \)
by simp
next
fix y::'d assume y ≠ x
have \( ?\text{fun1} y = \text{norm } (f x ** ?D g g' y + ?D f f' y ** g y + f'(y - x) ** (g}
\[
y - g x) / \text{norm} (y - x)
\]
\begin{itemize}
\item by simp add: \text{diff-left} \text{diff-right} \text{add-left} \text{add-right} \text{field-simps}
\item also have \ldots \leq (\text{norm} (f x) * \text{norm} (?D g g' y) * K + \text{norm} (?D f f' y) * \\
\text{norm} (g y) * K + \\
\text{norm} (y - x) * K F * \text{norm} (g y - g x) * K) / \text{norm} (y - x)
\end{itemize}
\begin{itemize}
\item by (intro divide-right-mono \text{mult-mono}')
\item order-trans \{OF \text{norm-triangle-ineq add-mono}\}
\item order-trans \{OF \text{norm-prod \text{mult-right-mono}}\}
\item \text{mult-nonneg-nonneg order-refl \text{norm-ge-zero norm-F} K [THEN order-less-imp-le]}\}
\end{itemize}
\begin{itemize}
\item also have \ldots = \text{?fun2} y
\item by simp add: \text{add-divide-distrib Ng-def Nf-def}
\end{itemize}
finally show \text{?fun1} y \leq \text{?fun2} y.
\begin{itemize}
\item qed simp
\end{itemize}
\begin{itemize}
\item lemmas \text{has-derivative-mult\{simp, derivative-intros\} = bounded-bilinear.FDERIV\{OF bounded-bilinear-mult\}}
\item lemmas \text{has-derivative-scaleR\{simp, derivative-intros\} = bounded-bilinear.FDERIV\{OF bounded-bilinear-scaleR\}}
\end{itemize}
\begin{itemize}
\item lemma \text{has-derivative-setprod\{simp, derivative-intros\}}:
\item fixes \( f :: i \Rightarrow 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{real-normed-field} \)
\item assumes \text{f:} \( \bigwedge i. i \in I \implies (f i \text{ has-derivative } f' i) \) (at \text{x} within \text{s})
\item shows \((\lambda x. \prod i\in I. f i x) \text{ has-derivative } (\lambda y. \sum i\in I. f' i y * (\prod j\in I - \{i\}. f j x))) \) (at \text{x} within \text{s})
\item proof cases
\item assume finite \( I \) from this \text{f} show \text{?thesis}
\item proof induct
\item case (insert \text{i} \text{I})
\item let \( ?P = \lambda y. f i x * (\sum i\in I. f' i y * (\prod j\in I - \{i\}. f j x)) + (f' i y) * (\prod i\in I. f i x)\)
\item have \((\lambda x. f i x * (\prod i\in I. f i x)) \text{ has-derivative } ?P) \) (at \text{x} within \text{s})
\item using by (intro has-derivative-mult) auto
\item also have \( ?P = (\lambda y. \sum i'\in \text{insert } i \text{ } I. f' i' y * (\prod j\in \text{insert } i \text{ } I - \{i'\}. f j x))\)
\item using insert(1,2) by (auto simp add: setsum-right-distrib insert-Diff-if intro: \text{ext setsum.cong})
\item finally show \text{?case}
\item using insert by simp
\item qed simp
\end{itemize}
\begin{itemize}
\item lemma \text{has-derivative-power\{simp, derivative-intros\}}:
\item fixes \( f :: 'a :: \text{real-normed-vector} \Rightarrow 'b :: \text{real-normed-field} \)
\item assumes \text{f:} \( f \text{ has-derivative } f' ) \) (at \text{x} within \text{s})
\item shows \((\lambda x. f x ^ n) \text{ has-derivative } (\lambda y. \text{of-nat } n * f' y * f x ^ (n - 1))) \) (at \text{x} within \text{s})
\item using \text{has-derivative-setprod\{OF f, of \{< n\}\} by (simp add: setprod-constant ac-simps)}
\end{itemize}
lemma has-derivative-inverse':
  fixes x :: 'a::real-normed-div-algebra
  assumes x: x ≠ 0
  shows (inverse has-derivative (λh. − (inverse x * h * inverse x))) (at x within s)
  (is (?inv has-derivative ?f) -)
proof (rule has-derivativeI-sandwich)
show bounded-linear (λh. − (?inv x * h * ?inv x))
  apply (rule bounded-linear-minus)
  apply (rule bounded-linear-mult-const)
  apply (rule bounded-linear-const-mult)
  apply (rule bounded-linear-ident)
done
next
show 0 < norm x using x by simp
next
show ((λy. norm (?inv y − ?inv x) * norm (?inv x)) −−−> 0) (at x within s)
  apply (rule tendsto-mult-left-zero)
  apply (rule tendsto-norm-zero)
  apply (rule LIM-zero)
  apply (rule tendsto-inverse)
  apply (rule tendsto-ident-at)
  apply (rule x)
done
next
fix y:'a assume h: y ≠ x dist y x < norm x
then have y ≠ 0
  by (auto simp: norm-cone-dist dist-commute)
have norm (?inv y − ?inv x − ?f (y − x)) / norm (y − x) = norm ((?inv y − ?inv x) * (y − x) * ?inv x) / norm (y − x)
  apply (subst inverse-diff-inverse [OF ⟨y ≠ 0 ⟩ x])
  apply (subst minus-diff-minus)
  apply (subst norm-minus-cancel)
  apply (simp add: left-diff-distrib)
done
also have ... ≤ norm (?inv y − ?inv x) * norm (y − x) * norm (?inv x) / norm (y − x)
  apply (rule divide-right-mono [OF norm-ge-zero])
  apply (rule order-trans [OF norm-mult-ineq])
  apply (rule mult-right-mono [OF norm-ge-zero])
  apply (rule norm-mult-ineq)
done
also have ... = norm (?inv y − ?inv x) * norm (?inv x)
  by simp
finally show norm (?inv y − ?inv x − ?f (y − x)) / norm (y − x) ≤
  norm (?inv y − ?inv x) * norm (?inv x) .
qed
**THEORY** “Deriv” 1605

* **lemma** *has-derivative-inverse*[simp, derivative-intros]:
  * fixes *f :: - ⇒ 'a::real-normed-div-algebra*
  * assumes *x: f x ≠ 0 and f: (f has-derivative f') (at x within s)*
  * shows *((λx. inverse (f x)) has-derivative (λh. − (inverse (f x) * f' h * inverse (f x)))) (at x within s)*
  * using *has-derivative-compose[OF f has-derivative-inverse', OF x]*

* **lemma** *has-derivative-divide*[simp, derivative-intros]:
  * fixes *f :: - ⇒ 'a::real-normed-div-algebra*
  * assumes *f: (f has-derivative f') (at x within s) and g: (g has-derivative g') (at x within s)*
  * assumes *x: g x ≠ 0*
  * shows *((λx. f x / g x) has-derivative (λh. − f x * (inverse (g x) * g' h * inverse (g x)) + f' h / g x)) (at x within s)*
  * using *has-derivative-mult[OF f has-derivative-inverse][OF x g]]*
  * by (simp add: field-simps)

Conventional form requires mult-AC laws. Types real and complex only.

* **lemma** *has-derivative-divide*[derivative-intros]:
  * fixes *f :: - ⇒ 'a::real-normed-field*
  * assumes *f: (f has-derivative f') (at x within s) and g: (g has-derivative g') (at x within s) and x: g x ≠ 0*
  * shows *((λx. f x / g x) has-derivative (λh. (f' h * g x − f x * g' h) / (g x * g x))) (at x within s)*
  * proof –
    { fix *h*
      have *f' h / g x − f x * (inverse (g x) * g' h * inverse (g x)) =*
        *((f' h * g x − f x * g' h) / (g x * g x)) =*
        *by (simp add: field-simps x) *
    }
  then show ?thesis
  * using *has-derivative-divide [OF f g] x*
  * by simp
  qed

**103.4 Uniqueness**

This can not generally shown for *op has-derivative*, as we need to approach the point from all directions. There is a proof in *Multivariate-Analysis* for euclidean-space.

* **lemma** *has-derivative-zero-unique:*
  * assumes *((λx. 0) has-derivative F) (at x) shows F = (λh. 0)*
  * proof –
    { interpret F: bounded-linear F
      using assms by (rule has-derivative-bounded-linear)
    let *r = λh. norm (F h) / norm h*
    have *: ?r = 0 ➔ 0*
    qed
using assms unfolding has-derivative-at by simp

show \( F = (\lambda h. 0) \)

proof
  fix \( h \) show \( F \ h = 0 \)
  proof (rule ccontr)
    assume \( \ast \ast \): \( F \ h \neq 0 \)
    hence \( h \): \( h \neq 0 \)
      by (clarsimp simp add: F.zero)
    with \( \ast \ast \) have \( 0 < ?r \ h \)
      by simp
    from LIM-D[OF \( \ast \) this] obtain \( s \) where
      \( s \): \( 0 < s \)
      and \( \tau \): \( \forall x. x \neq 0 \Rightarrow \) norm \( x < s \) \( \Rightarrow \) ?r \( x < ?r \ h \)
      by auto
    with \( \tau \) have \( 0 < ?r \)
      using \( \ast \ast \) by (rule r)
    thus \( False \)
      using \( \ast \ast \) by (simp add: F.scaleR)
  qed
qed

lemma has-derivative-unique:
  assumes \( (f \ has-derivative F) \ (at x) \) and \( (f \ has-derivative F') \ (at x) \)
  shows \( F = F' \)

proof
  have \( ((\lambda x. 0) \ has-derivative (\lambda h. F \ h - F' \ h)) \ (at x) \)
    using has-derivative-diff[OF assms] by simp
  hence \( (\lambda h. F \ h - F' \ h) = (\lambda h. 0) \)
    by (rule has-derivative-zero-unique)
  thus \( F = F' \)
    unfolding fun-eq_iff right-minus-eq.
  qed

103.5 Differentiability predicate

definition
differentiable :: ('a::real-normed-vector ⇒ 'b::real-normed-vector) ⇒ 'a filter ⇒ bool
  (infix differentiable 50)
where
  \( f \ differentiable F \) \( \Longleftrightarrow (\exists D. (f \ has-derivative D) \ F) \)

lemma differentiable-subset: \( f \ differentiable \ (at x \ within s) \) \( \imp \ t \subseteq s \Rightarrow \) \( f \ differentiable \ (at x \ within t) \)
  unfolding differentiable-def by (blast intro: has-derivative-subset)

lemmas differentiable-within-subset = differentiable-subset

lemma differentiable-ident [simp, derivative-intros]: \( (\lambda x. x) \ differentiable F \)
  unfolding differentiable-def by (blast intro: has-derivative-ident)
lemma differentiable-const [simp, derivative-intros]: $(\lambda z. a)$ differentiable $F$
unfolding differentiable-def by (blast intro: has-derivative-const)

lemma differentiable-compose:
$f$ differentiable $(at (g x) within (g's)) \implies g$ differentiable $(at x within s) \implies (\lambda x. f (g x))$ differentiable $(at x within s)$
unfolding differentiable-def by (blast intro: has-derivative-in-compose)

lemma differentiable-sum [simp, derivative-intros]:
$f$ differentiable $F \implies g$ differentiable $F \implies (\lambda x. f x + g x)$ differentiable $F$
unfolding differentiable-def by (blast intro: has-derivative-add)

lemma differentiable-minus [simp, derivative-intros]:
$f$ differentiable $F \implies (\lambda x. f x - g x)$ differentiable $F$
unfolding differentiable-def by (blast intro: has-derivative-minus)

lemma differentiable-diff [simp, derivative-intros]:
$f$ differentiable $F \implies g$ differentiable $F \implies (\lambda x. f x - g x)$ differentiable $F$
unfolding differentiable-def by (blast intro: has-derivative-diff)

lemma differentiable-mult [simp, derivative-intros]:
fixes $f g :: 'a :: real-normed-vector \Rightarrow 'b :: real-normed-algebra$
shows $f$ differentiable $(at x within s) \implies g$ differentiable $(at x within s) \implies (\lambda x. f x * g x)$ differentiable $(at x within s)$
unfolding differentiable-def by (blast intro: has-derivative-mult)

lemma differentiable-inverse [simp, derivative-intros]:
fixes $f :: 'a :: real-normed-vector \Rightarrow 'b :: real-normed-field$
shows $f$ differentiable $(at x within s) \implies f x \neq 0 \implies (\lambda x. inverse (f x))$ differentiable $(at x within s)$
unfolding differentiable-def by (blast intro: has-derivative-inverse)

lemma differentiable-divide [simp, derivative-intros]:
fixes $f g :: 'a :: real-normed-vector \Rightarrow 'b :: real-normed-field$
shows $f$ differentiable $(at x within s) \implies g$ differentiable $(at x within s) \implies (\lambda x. f x / g x)$ differentiable $(at x within s)$
unfolding divide-inverse using assms by simp

lemma differentiable-power [simp, derivative-intros]:
fixes $f g :: 'a :: real-normed-vector \Rightarrow 'b :: real-normed-field$
shows $f$ differentiable $(at x within s) \implies (\lambda x. f x ^ n)$ differentiable $(at x within s)$
unfolding differentiable-def by (blast intro: has-derivative-power)
lemma differentiable-scaleR [simp, derivative-intros]:
  \( f \) differentiable (at \( x \) within \( s \)) \( \Longrightarrow \) \( g \) differentiable (at \( x \) within \( s \)) \( \Longrightarrow \) \( (\lambda x. f x \cdot_R g x) \) differentiable (at \( x \) within \( s \))
  unfolding differentiable-def by (blast intro: has-derivative-scaleR)

lemma has-derivative-imp-has-field-derivative:
  \( (f \text{ has-derivative } D) F \implies (\forall x. x \cdot_D' = D x) \implies (f \text{ has-field-derivative } D') F \)
  unfolding has-field-derivative-def
  by (rule has-derivative-eq-rhs[of \( f \) \( D \)]) (simp-all add: fun-eq-iff mult.commute)

lemma has-field-derivative-imp-has-derivative:
  \( (f \text{ has-field-derivative } D') F \implies (f \text{ has-derivative } \cdot_D D') F \)
  by (simp add: has-field-derivative-def)

lemma DERIV-subset:
  \( (f \text{ has-field-derivative } f') (at \( x \) within \( s \)) \implies \subseteq \)
  \( (f \text{ has-field-derivative } f') (at \( x \) within \( t \)) \)
  by (simp add: has-field-derivative-def has-derivative-within-subset)

abbreviation (input)
  \( \text{DERIV} : (\text{a} : \text{real-normed-field} \Rightarrow (\text{b} : (\text{a} \Rightarrow \text{bool}) \Rightarrow (\text{a} \Rightarrow \text{bool})) [1000, 1000, 60] 60) \)

where
  \( \text{DERIV } f x : D \equiv (f \text{ has-field-derivative } D') (at \( x \)) \)

abbreviation
  \( \text{has-real-derivative} : (\text{real} \Rightarrow \text{real} \Rightarrow \text{real filter} \Rightarrow \text{bool}) \)

where
  \( (f \text{ has-real-derivative } D) F \equiv (f \text{ has-field-derivative } D') F \)

lemma real-differentiable-def:
  \( f \) differentiable at \( x \) within \( s \) \( \iff \) \( (\exists D. (f \text{ has-real-derivative } D) (at \( x \) within \( s \))) \)

proof safe
  assume \( f \) differentiable at \( x \) within \( s \)

  then obtain \( f' \) where \( *: (f \text{ has-derivative } f') (at \( x \) within \( s \)) \)
    unfolding differentiable-def by auto

  then obtain \( c \) where \( f' = (\text{op} \cdot c) \)
    by (metis real-bounded-linear has-derivative-bounded-linear mult.commute fun-eq-iff)

with \( * \) show \( \exists D. (f \text{ has-real-derivative } D) (at \( x \) within \( s \)) \)
    unfolding has-field-derivative-def by auto

qed (auto simp: differentiable-def has-field-derivative-def)

lemma real-differentiableE [elim?]:
  assumes \( f : f \text{ differentiable} (at \( x \) within \( s \)) \) obtains \( df \) where \( (f \text{ has-real-derivative } df) (at \( x \) within \( s \)) \)
  using assms by (auto simp: real-differentiable-def)

lemma differentiableD: \( f \) differentiable (at \( x \) within \( s \)) \( \Longrightarrow \) \( \exists D. (f \text{ has-real-derivative } \)
THEORY “Deriv”

lemma differentiableI: (f has-real-derivative D) (at x within s) =⇒ f differentiable (at x within s)
  by (force simp add: real-differentiable-def)

lemma DERIV-def: DERIV f x := D ⇐⇒ (∀h. (f (x + h) − f x) / h −−> 0) −−> D
  apply (simp add: has-field-derivative-def has-derivative-at bounded-linear-mult-right LIM-zero-iff[symmetric, of - D])
  apply (rule filterlim-cong)
  apply (simp-all add: eventually-at-filter field-simps nonzero-norm-divide)
  done

lemma mult-commute-abs: (λx. x * c) = op * (c::'a::ab-semigroup-mult)
  by (simp add: fun-eq-iff mult.commutative)

103.6 Derivatives

lemma DERIV-D: DERIV f x := D =⇒ (∀h. (f (x + h) − f x) / h −−> 0) −−> D
  by (simp add: DERIV-def)

lemma DERIV-const: ((λx. k) has-field-derivative 0) F =⇒ (∀x. (λx. k) has-field-derivative 0) F
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-const])

lemma DERIV-ident: ((λx. x) has-field-derivative 1) F =⇒ (∀x. (λx. x) has-field-derivative 1) F
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-ident])

lemma field-differentiable-add: ((f has-field-derivative f') F =⇒ (g has-field-derivative g') F =⇒
  ((λz. f z + g z) has-field-derivative f' + g') F)
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-add])
  (auto simp: has-field-derivative-def field-simps mult-commute-abs)

corollary DERIV-add: (f has-field-derivative D) (at x within s) =⇒ (g has-field-derivative E) (at x within s)
  =⇒ ((λx. f x + g x) has-field-derivative D + E) (at x within s)
  by (rule field-differentiable-add)

lemma field-differentiable-minus: ((f has-field-derivative f') F =⇒
  ((λz. − (f z)) has-field-derivative −f') F)
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-minus])
  (auto simp: has-field-derivative-def field-simps mult-commute-abs)

corollary DERIV-minus: (f has-field-derivative D) (at x within s) =⇒ ((λx. − f
THEORY "Deriv"

lemma field-differentiable-diff[derivative-intros]:
  (f has-field-derivative f') F =⇒ (g has-field-derivative g') F =⇒ ((λz. f z - g z) has-field-derivative f' - g') F
  by (simp only: assms diff-conv-add-uminus field-differentiable-add field-differentiable-minus)

corollary DERIV-diff:
  (f has-field-derivative D) (at x within s) =⇒ (g has-field-derivative E) (at x within s) =⇒ ((λx. f x - g x) has-field-derivative D - E) (at x within s)
  by (rule field-differentiable-diff)

lemma DERIV-continuous: (f has-field-derivative D) (at x within s) =⇒ continuous (at x within s) f
  by (drule has-derivative-continuous[OF has-field-derivative-imp-has-derivative]) simp

corollary DERIV-isCont: DERIV f x => D =⇒ isCont f x
  by (rule DERIV-continuous)

lemma DERIV-continuous-on:
  (∀x. x ∈ s =⇒ (f has-field-derivative D) (at x)) =⇒ continuous-on s f
  by (metis DERIV-continuous continuous-at-imp-continuous-on)

lemma DERIV-mult':
  (f has-field-derivative D) (at x within s) =⇒ (g has-field-derivative E) (at x within s) =⇒
  (λx. f x * g x) has-field-derivative f x * E + D * g x) (at x within s)
  by (rule has-derivative-imp-has-field-derivative[OF has-field-derivative-mult])
  (auto simp: field-simps mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma DERIV-mult[derivative-intros]:
  (f has-field-derivative Da) (at x within s) =⇒ (g has-field-derivative Db) (at x within s) =⇒
  ((λx. f x * g x) has-field-derivative Da * g x + Db * f x) (at x within s)
  by (rule has-derivative-imp-has-field-derivative[OF has-field-derivative-mult])
  (auto simp: field-simps dest: has-field-derivative-imp-has-derivative)

Derivative of linear multiplication

lemma DERIV-cmult:
  (f has-field-derivative D) (at x within s) ==> ((λx. c * f x) has-field-derivative c * D) (at x within s)
  by (drule DERIV-mult'[OF DERIV-const], simp)

lemma DERIV-cmult-right:
  (f has-field-derivative D) (at x within s) ==> ((λx. f x * c) has-field-derivative D * c) (at x within s)
using DERIV-cmult by (force simp add: ac-simps)

lemma DERIV-cmult-Id [simp]: (op * c has-field-derivative c) (at x within s)
by (cut_tac c = c and x = x in DERIV-ident [THEN DERIV-cmult], simp)

lemma DERIV-cdivide:
(f has-field-derivative D) (at x within s) ⇒ ((λx. f x / c) has-field-derivative D / c) (at x within s)
using DERIV-cmult-right[of f D x s 1 / c] by simp

lemma DERIV-unique:
DERIV f x :> D ⇒ DERIV f x :> E ⇒ D = E
unfolding DERIV-def by (rule LIM-unique)

lemma DERIV-setsum[derivative-intros]:
(∀ n. n ∈ S ⇒ ((λx. f x n) has-field-derivative (f' x n)) F)⇒
((λx. setsum (f x) S) has-field-derivative setsum (f' x) S) F
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-setsum])
(auto simp: setsum-right-distrib mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma DERIV-inverse'[derivative-intros]:
(f has-field-derivative D) (at x within s)⇒ f x ≠ 0 ⇒
((λx. inverse (f x)) has-field-derivative − (inverse (f x) * D * inverse (f x))) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-inverse])
(auto dest: has-field-derivative-imp-has-derivative)

Power of −1

lemma DERIV-inverse:
x ≠ 0 ⇒ ((λx. inverse(x)) has-field-derivative − (inverse (x ^ Suc (Suc 0)))) (at x within s)
by (drule DERIV-inverse' [OF DERIV-ident]) simp

Derivative of inverse

lemma DERIV-inverse-fun:
(f has-field-derivative d) (at x within s)⇒ f x ≠ 0 ⇒
((λx. inverse (f x)) has-field-derivative − (d * inverse(f x ^ Suc (Suc 0)))) (at x within s)
by (drule (1) DERIV-inverse') (simp add: ac-simps nonzero-inverse-mult-distrib)

Derivative of quotient

lemma DERIV-divide[derivative-intros]:
(f has-field-derivative D) (at x within s)⇒
(g has-field-derivative E) (at x within s)⇒ g x ≠ 0 ⇒
((λx. f x / g x) has-field-derivative (D * g x - f x * E) / (g x * g x)) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-divide])
(auto dest: has-field-derivative-imp-has-derivative simp: field-simps)
lemma DERIV-quotient:
(f has-field-derivative d) (at x within s) \implies
(g has-field-derivative e) (at x within s) \implies g x \neq 0 \implies
((\lambda y. f y / g y) has-field-derivative (d * g x - (e * f x)) / (g x - Suc (Suc 0)))
(at x within s)
by (drule (2) DERIV-divide) (simp add: mult.commute)

lemma DERIV-power-Suc:
(f has-field-derivative D) (at x within s) \implies
((\lambda x. f x ^ Suc n) has-field-derivative (1 + of-nat n) * (D * f x ^ n)) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative_def)

lemma DERIV-power[derivative-intros]:
(f has-field-derivative D) (at x within s) \implies
((\lambda x. f x ^ n) has-field-derivative of-nat n * (D * f x ^ (n - Suc 0))) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative_def)

lemma DERIV-pow: ((\lambda x. x ^ n) has-field-derivative real n * (x ^ (n - Suc 0)))
(at x within s)
apply (cut-tac DERIV-power[OF DERIV-ident])
apply (simp add: real_of_nat_def)
done

lemma DERIV-chain': (f has-field-derivative D) (at x within s) \implies DERIV g (f x) :> E \implies
((\lambda x. g (f x)) has-field-derivative E * D) (at x within s)
using has-derivative-compose[of f op * D x s g op * E]
unfolding has-field-derivative_def mult-commute-abs ac-simps.

corollary DERIV-chain2: DERIV f (g x) :> Da \implies (g has-field-derivative Db)
(at x within s) \implies
((\lambda x. f (g x)) has-field-derivative Da * Db) (at x within s)
by (rule DERIV-chain')

Standard version

lemma DERIV-chain:
DERIV f (g x) :> Da \implies (g has-field-derivative Db)
(at x within s) \implies
(f o g has-field-derivative Da * Db) (at x within s)
by (drule (1) DERIV-chain', simp add: o-def mult.commute)

lemma DERIV-image-chain:
(f has-field-derivative Da) (at (g x) within (g ' s)) \implies (g has-field-derivative Db)
(at x within s) \implies
(f o g has-field-derivative Da * Db) (at x within s)
using has-derivative-in-compose [of g op * Db x s f op * Da ]
lemma DERIV-chain-s:
assumes \((\forall x. x \in s \implies \text{DERIV } g x : g'(x))\)
  and \(\text{DERIV } f x : f'\)
  and \(x \in s\)
shows \(\text{DERIV } (\lambda x. g(f x)) x : f' \ast g'(f x)\)
by (metis (full-types) DERIV-chain \mult \text{commute assms})

lemma DERIV-chain3:
assumes \((\forall x. \text{DERIV } g x : g'(x))\)
  and \(\text{DERIV } f x : f'\)
shows \(\text{DERIV } (\lambda x. g(f x)) x : f' \ast g'(f x)\)
by (metis UNIV-I DERIV-chain-s \[\text{of UNIV}\] assms)

declare DERIV-power \[where 'a=real, unfolded real-of-nat-def \text{[symmetric]}, \text{derivative-intras}]\]

Alternative definition for differentiability

lemma DERIV-LIM-iff:
fixes \(f : 'a::{real-normed-vector,inverse} \Rightarrow 'a\) shows
  \((\%h. (f(a + h) - f(a)) / h) \lim 0 \limto D) =
  \((\%x. (f(x) - f(a)) / (x-a)) \lim 0 \limto D)\)
apply (rule iffI)
apply (drule-tac k=- a in LIM-offset)
apply simp
apply (drule-tac k=a in LIM-offset)
apply (simp add: add.commute)
done

lemma DERIV-iff2: \((\text{DERIV } f x : D) \iff \text{DERIV } f (z - f x) / (z - x) \limto x \limto D)\)
  by (simp add: DERIV-def DERIV-LIM-iff)

lemma DERIV-cong-ev: \(x = y \implies \text{eventually } (\lambda x. f x = g x) (\text{nhds } x) \implies u = v \implies \text{DERIV } f x : u \iff \text{DERIV } g y : v\)
unfolding DERIV-iff2
proof (rule filterlim-cong)
  assume *: \text{eventually } (\lambda x. f x = g x) (\text{nhds } x)
  moreover from * have \(f x = g x\) by (auto simp: eventually-nhds)
  moreover assume \(x = y u = v\)
  ultimately show \text{eventually } (\lambda x. (f x - f x) / (x - y) = (g x - g y) / (x a - y)) (at x)
    by (auto simp: eventually-at-filter elim: eventually-elim1)
qed simp-all

lemma DERIV-shift:
(\text{DERIV} f \ (x + z) :> y) \leftrightarrow (\text{DERIV} \ (\lambda x. f \ (x + z)) \ x :> y)
\text{by (simp add: DERIV-def field-simps)}

\text{lemma \ DERIV-mirror:}
(\text{DERIV} f \ (- x) :> y) \leftrightarrow (\text{DERIV} \ (\lambda x. f \ (- x::real) :: real) \ x :> - y)
\text{by (simp add: DERIV-def filterlim-at-split filterlim-at-left-to-right tendsto-minus-cancel-left field-simps conj-commute)}

\text{Caratheodory formulation of derivative at a point}
\text{lemma \ CARAT-DERIV:}
(\text{DERIV} f \ x :> l) \leftrightarrow (\exists g. \ (\forall z. f \ z - f \ x = g \ z * (z - x)) \land \text{isCont} \ g \ x \land g \ x = l)
\text{by (is \ ?lhs = \ ?rhs)}

\text{proof}
\text{assume \ der: \ DERIV} f \ x :> l
\text{show \ (\exists g. \ (\forall z. f \ z - f \ x = g \ z * (z - x)) \land \text{isCont} \ g \ x \land g \ x = l)
\text{by \ simp \ intro \ exI \ conjI}
\text{let \ ?g = (%z. \ if z = x then l else \ (f \ z - f \ x) / (z - x))
\text{show \ (\forall z. f \ z - f \ x = ?g \ z * (z - x)) \ by \ simp \ isCont \ ?g \ x \ using \ der
\text{by (simp add: isCont-iff DERIV-def cong: LIM-equal [rule-format])
\text{show \ ?g \ x = l \ by \ simp \ qed
\text{qed
\text{next
\text{assume \ ?rhs
\text{then obtain \ g \ where
\ (\forall z. f \ z - f \ x = g \ z * (z - x)) \ and \ isCont \ g \ x \ and \ g \ x = l \ by \ blast
\text{thus \ (DERIV} f \ x :> l)
\text{by \ (auto \ simp \ add: isCont-iff DERIV-def cong: LIM-cong)
\text{qed
Let’s do the standard proof, though theorem \text{LIM-mult2} follows from a NS proof

103.7 Local extrema
If (\theta::'a) < f' \ x then \ x is Locally Strictly Increasing At The Right
\text{lemma \ DERIV-pos-inc-right:}
\text{fixes} \ f :: real => real
\text{assumes \ der: \ DERIV} f \ x :> l
\text{and \ l: \ 0 < l
\text{shows \ (\exists d > 0. \ \forall h > 0. \ h < d \ --- \ f(x) < f(x + h)
\text{proof --}
\text{from \ l \ \der \ [THEN DERIV-D, \ THEN LIM-D [where \ r = l]]
\text{have \ (\exists s > 0. \ (\forall z. \ z \neq 0 \land \ |z| < s \ --- \ |(f(x+z) - f \ x) / z - l| < l)
\text{by simp
\text{then obtain \ s
\text{where \ s: \ 0 < s

THEORY "Deriv"

1614
and all: ![z. z \neq 0 \land |z| < s \rightarrow |(f(x+z) - f x) / z - l| < l
by auto
thus \texttt{?thesis}
proof (intro exI conjI strip)
  show 0<s using s .
fix h::real
assume 0 < h h < s
with all [of h] show f x < f (x+h)
proof (simp add: abs-if pos-less-divide-eq split add: split-if-asm)
  assume \((f (x+h) - f x) / h < l\)
  with l
  have 0 < ((f (x+h) - f x) / h) by arith
  thus \(f x < f (x+h)\)
by (simp add: pos-less-divide-eq h)
qed
qed

lemma DERIV-neg-dec-left:
fixes f :: real
assumes der: DERIV f x :> l
and l: \(l < 0\)
shows \(\exists d > 0. \forall h > 0. h < d \rightarrow f(x) < f(x-h)\)
proof
  from l der \[THEN DERIV-D, THEN LIM-D \[where r = -l]]
  have \(\exists s > 0. (\forall z. z \neq 0 \land |z| < s \rightarrow |(f(x+z) - f x) / z - l| < -l)\)
  by simp
then obtain s
where s: \(0 < s\)
  and all: ![z. z \neq 0 \land |z| < s \rightarrow |(f(x+z) - f x) / z - l| < -l
by auto
thus \texttt{?thesis}
proof (intro exI conjI strip)
  show 0<s using s .
fix h::real
assume 0 < h h < s
with all [of \(-h\)] show f x < f (x-h)
proof (simp add: abs-if pos-less-divide-eq split add: split-if-asm)
  assume \(-((f (x-h) - f x) / h) < l\)
  with l
  have 0 < ((f (x-h) - f x) / h) by arith
  thus \(f x < f (x-h)\)
by (simp add: pos-less-divide-eq h)
qed
qed

lemma DERIV-pos-inc-left:
fixes f :: real => real

shows $\text{DERIV} f \ x : l \implies 0 < l \implies \exists d > 0. \forall h > 0. h < d \implies f(x - h) < f(x)$
apply (rule $\text{DERIV-neg-dec-left}$ [of $\% x. - f x - l x$, simplified])
apply (auto simp add: $\text{DERIV-minus}$)
done

lemma $\text{DERIV-neg-dec-right}$:
fixes $f :: \text{real} \implies \text{real}$
shows $\text{DERIV} f \ x : l \implies l < 0 \implies \exists d > 0. \forall h > 0. h < d \implies f(x) > f(x + h)$
apply (rule $\text{DERIV-pos-inc-right}$ [of $\% x. - f x - l x$, simplified])
apply (auto simp add: $\text{DERIV-minus}$)
done

lemma $\text{DERIV-local-max}$:
fixes $f :: \text{real} \implies \text{real}$
assumes $\text{der}$: $\text{DERIV} f \ x : l$
and $d$: $0 < d$
and $\text{le}$: $\forall y. |x - y| < d \implies f(y) \leq f(x)$
shows $l = 0$
proof (cases rule: linorder-cases [of $l$ $0$])
case equal
thus ?thesis .
next
case less
from $\text{DERIV-neg-dec-left}$ [OF $\text{der} \ \text{less}$]
obtain $d'$ where $d' : 0 < d'$
and $lt$: $\forall h > 0. h < d' \implies f(x) < f(x - h)$ by blast
from real-lbound-gt-zero [OF $d \ d'$]
obtain $e$ where $0 < e \land e < d \land e < d'$ ..
with $lt \ \text{le}$ [THEN spec [where $x=x-e$]]
show ?thesis by (auto simp add: abs-if)
next
case greater
from $\text{DERIV-pos-inc-right}$ [OF $\text{der} \ \text{greater}$]
obtain $d'$ where $d' : 0 < d'$
and $lt$: $\forall h > 0. h < d' \implies f(x) < f(x + h)$ by blast
from real-lbound-gt-zero [OF $d \ d'$]
obtain $e$ where $0 < e \land e < d \land e < d'$ ..
with $lt \ \text{le}$ [THEN spec [where $x=x+e$]]
show ?thesis by (auto simp add: abs-if)
qed

Similar theorem for a local minimum
lemma $\text{DERIV-local-min}$:
fixes $f :: \text{real} \implies \text{real}$
shows $\land \text{DERIV} f \ x : l; 0 < d; \forall y. |x-y| < d \implies f(x) \leq f(y) \implies l = 0$
by (drule $\text{DERIV-minus}$ [THEN $\text{DERIV-local-max}$], auto)

In particular, if a function is locally flat
lemma DERIV-local-const:
  fixes f :: real => real
  shows [[ DERIV f x :: l; 0 < d; \forall y. \|x-y\| < d --> f(x) = f(y) ]] ==> l = 0
  by (auto dest!: DERIV-local-max)

103.8 Rolle's Theorem

Lemma about introducing open ball in open interval

lemma lemma-interval-lt:
  \[ a < x; x < b \] ==> \exists d::real. 0 < d & (\forall y. \|x-y\| < d --> a < y & y < b)

apply (simp add: abs-less-iff)
apply (insert linorder-linear[of x-a b-x-x], safe)
apply (rule-tac x=x-a in exI)
apply (rule-tac [2] x=b-x in exI, auto)
done

lemma lemma-interval:
  \[ a < x; x < b \] ==> \exists d::real. 0 < d & (\forall y. \|x-y\| < d --> a < y & y < b)

apply (drule lemma-interval-lt, auto)
apply force
done

Rolle's Theorem. If f is defined and continuous on the closed interval \([a,b]\)
and differentiable on the open interval \((a,b)\), and \(f(a) = f(b)\),
then there exists \(x_0 \in (a,b)\) such that \(f'(x_0) = 0\).

theorem Rolle:
assumes lt: a < b
  and eq: f(a) = f(b)
  and con: \(\forall x. a < x \& x < b --> isCont f x\)
  and dif [rule-format]: \(\forall x. a < x \& x < b --> f \text{ differentiable at } x\)
shows \(\exists z::real. a < z \& z < b \& DERIV f z :> 0\)
proof
  have le: a \leq b using lt by simp
  from isCont-eq-Ub [OF le con]
  obtain x where x-max: \(\forall z. a \leq z \& z \leq b --> f z \leq f x\)
                and alex: a \leq x and xleb: x \leq b
    by blast
  from isCont-eq-Lb [OF le con]
  obtain x' where x'-min: \(\forall z. a < z \& z < b --> f x' \leq f z\)
                and alex': a \leq x' and x'leb: x' \leq b
    by blast
  show ?thesis
proof cases
  assume axb: a < x \& x < b
    -- f attains its maximum within the interval
hence \( ax: a < x \) and \( xb: x < b \) by arith +
from lemma-interval \([\text{OF } ax \text{ } xb]\)
obtain \( d \) where \( d: \theta < d \) and bound: \( \forall y. \vert x-y \vert < d \longrightarrow a \leq y \land y \leq b \)
by blast
hence bound': \( \forall y. \vert x-y \vert < d \longrightarrow f y \leq f x \) using x-max
by blast
from differentiableD \([\text{OF } \text{dif } [\text{OF } axb]]\)
obtain \( l \) where der: DERIV \( f x :> l \) ..
have \( l=0 \) by (rule DERIV-local-max \([\text{OF } \text{der } d \text{ bound'}]\))
— the derivative at a local maximum is zero
thus \( \text{thesis using } ax \text{ } xb \text{ der by auto} \)
next
assume notaxb: \( \neg (a < x \land x < b) \)
hence \( xeqab: x=a \mid x=b \) using alex zleb by arith
hence \( \text{fb-eq-fx': } f b = f x \) by (auto simp add: eq)
show \( \text{thesis} \)
proof cases
assume \( ax'xb: a < x' \land x' < b \)
— \( f \) attains its minimum within the interval
hence \( ax': a<x' \land x'xb: x'<b \) by arith +
from lemma-interval \([\text{OF } ax' \text{ } x'b]\)
obtain \( d \) where \( d: \theta < d \) and bound: \( \forall y. \vert x'-y \vert < d \longrightarrow a \leq y \land y \leq b \)
by blast
hence bound': \( \forall y. \vert x'-y \vert < d \longrightarrow f x' \leq f y \) using x'-min
by blast
from differentiableD \([\text{OF } \text{dif } [\text{OF } ax' \text{ } x'b]]\)
obtain \( l \) where der: DERIV \( f x' :> l \) ..
have \( l=0 \) by (rule DERIV-local-min \([\text{OF } \text{der } d \text{ bound'}]\))
— the derivative at a local minimum is zero
thus \( \text{thesis using } ax' x'b \text{ der by auto} \)
next
assume notax'xb: \( \neg (a < x' \land x' < b) \)
— \( f \) is constant throughout the interval
hence \( x'eqab: x'=a \mid x'=b \) using alex' zleb by arith
hence \( \text{fb-eq-fx': } f b = f x' \) by (auto simp add: eq)
from dense \([\text{OF } \text{lt}]\)
obtain \( r \) where ar: \( a < r \) and \( rb: r < b \) by blast
from lemma-interval \([\text{OF } \text{ar } \text{rb}]\)
obtain \( d \) where \( d: \theta < d \) and bound: \( \forall y. \vert r-y \vert < d \longrightarrow a \leq y \land y \leq b \)
by blast
have eq-fb: \( \forall z. \ a \leq z \longrightarrow z \leq b \longrightarrow f z = f b \)
proof (clarify)
fix \( z::\text{real} \)
assume \( az: a \leq z \) and \( zb: z \leq b \)
show \( f z = f b \)
proof (rule order-antisym)
  show \( f z \leq f b \) by (simp add: fb-eq-fx x-max az zb)
  show \( f b \leq f z \) by (simp add: fb-eq-fx' x'-min az zb)
qed
**THEORY “Deriv”**

```plaintext
qed
have bound': \forall y. |r-y| < d \rightarrow f r = f y
proof (intro strip)
  fix y::real
  assume lt:: |r-y| < d
  hence f y = f b by (simp add: eq-fb bound)
  thus f r = f y by (simp add: eq-fb ar rb order-less-imp-le)
qed

from differentiableD [OF dif [OF conjI [OF ar rb]]]
obtain l where der: DERIV f r := l ..
have l=0 by (rule DERIV-local-const [OF der d bound'])
  -- the derivative of a constant function is zero
thus ?thesis using ar rb der by auto
qed

103.9 Mean Value Theorem

lemma lemma-MVT:
  \( f a - (f b - f a)/(b-a) * a = f b - (f b - f a)/(b-a) * (b::real) \)
by (cases a = b) (simp-all add: field-simps)

theorem MVT:
  assumes lt: a < b
  and con: \( \forall x. a \leq x \& x \leq b \rightarrow \text{isCont} f x \)
  and dif [rule-format]: \( \forall x. a < x \& x < b \rightarrow f \text{ differentiable (at x)} \)
shows \( \exists l z::real. a < z \& z < b \& \text{DERIV} f z := l \& \)
  \( (f(b) - f(a) = (b-a) * l) \)
proof -
  let \( \tilde{F} = \%x. f x - (f b - f a)/(b-a) * x \)
  have contF: \( \forall x. a \leq x \& x \leq b \rightarrow \text{isCont} \tilde{F} x \)
    using con by (fast intro: continuous-intros)
  have difF: \( \forall x. a < x \& x < b \rightarrow \tilde{F} \text{ differentiable (at x)} \)
    by (clarify)
  fix x::real
  assume ax: a < x and xb: x < b
  from differentiableD [OF dif [OF conjI [OF ax xb]]]
  obtain l where der: DERIV f x := l ..
  show \( \tilde{F} \text{ differentiable (at x)} \)
    by (rule differentiableI [where D = l - (f b - f a)/(b-a)],
    blast intro: DERIV-diff DERIV-cmult-Id der)
  qed

from Rolle [where f = \tilde{F}, OF lt lemma-MVT contF difF]
obtain z where az: a < z and zb: z < b and der: DERIV \tilde{F} z := 0
  by blast
  have DERIV (%x. (f b - f a)/(b-a)) * x z := (f b - f a)/(b-a)
    by (rule DERIV-cmult-Id)
hence derF: DERIV (%x. \tilde{F} x + (f b - f a)/(b-a) * x) z
```


THEORY "Deriv"

1620

by (rule DERIV-add [OF der])

show ?thesis

proof (intro exI conjI)

show a < z using az .

show z < b using zb .

show \( f b - f a = (b - a) \times (f b - f a)/(b-a) \) by (simp)

show DERIV f z => ((f b - f a)/(b-a)) using derF by simp

qed

lemma MVT2:

|| a < b; \forall x. a \leq x & x \leq b \longrightarrow DERIV f x => f'(x) ||

===> \exists z :: real. a < z & z < b & (f b - f a = (b - a) * f'(z))

apply (drule intro: DERIV-MVT)

apply (blast intro: DERIV-isCont)

apply (force dest: order-less-imp-eq simp add: real-differentiable-def)

apply (blast dest: DERIV-unique order-less-imp-eq)

done

A function is constant if its derivative is 0 over an interval.

lemma DERIV-isconst-end:

fixes f :: real => real

shows || a < b;

\forall x. a \leq x & x \leq b \longrightarrow isCont f x;

\forall x. a < x \& x < b \longrightarrow DERIV f x => 0 ||

===> \forall x. a \leq x & x \leq b \longrightarrow f x = f a

apply (drule MVT, assumption)

apply (blast intro: differentiableI)

apply (auto dest!: DERIV-unique simp add: diff-eq-eq)

done

lemma DERIV-isconst1:

fixes f :: real => real

shows || a < b;

\forall x. a \leq x & x \leq b \longrightarrow isCont f x;

\forall x. a < x \& x < b \longrightarrow DERIV f x => 0 ||

===> \forall x. a \leq x & x \leq b \longrightarrow f x = f a

apply safe

apply (drule-tac x = a in order-le-imp-less-or-eq, safe)

apply (drule-tac b = x in DERIV-isconst-end, auto)

done

lemma DERIV-isconst2:

fixes f :: real => real

shows || a < b;

\forall x. a \leq x & x \leq b \longrightarrow isCont f x;

\forall x. a < x \& x < b \longrightarrow DERIV f x => 0;

a \leq x; x \leq b ||
<table>
<thead>
<tr>
<th>Theory</th>
<th>“Deriv”</th>
<th>Page 1621</th>
</tr>
</thead>
</table>

```plaintext
==⇒ f x = f a
apply (blast dest: DERIV-isconst1)
done

lemma DERIV-isconst3: fixes a b x y :: real
  assumes a < b and x ∈ {a <..< b} and y ∈ {a <..< b}
  assumes derivable: ∀x. x ∈ {a <..< b} ⇒ DERIV f x :> 0
  shows f x = f y
proof (cases x = y)
case False
  let ?a = min x y
  let ?b = max x y

  have ∀z. ?a ≤ z ∧ z ≤ ?b −→ DERIV f z :> 0
  proof (rule allI, rule impl)
    fix z :: real assume ?a ≤ z ∧ z ≤ ?b
    hence a < z and z < b using ⟨x ∈ {a <..< b}⟩ and ⟨y ∈ {a <..< b}⟩ by auto
    hence z ∈ {a <..< b} by auto
    thus DERIV f z :> 0 by (rule derivable)
  qed

  hence isCont: ∀z. ?a ≤ z ∧ z ≤ ?b −→ isCont f z
  and DERIV: ∀z. ?a < z ∧ z < ?b −→ DERIV f z :> 0 using DERIV-isCont
  by auto

  have ?a < ?b using ⟨x ≠ y⟩ by auto
  from DERIV-isconst2[OF this isCont DERIV, of x] and DERIV-isconst2[OF this isCont DERIV, of y]
  show ?thesis by auto
  qed auto

lemma DERIV-isconst-all:
  fixes f :: real ⇒ real
  shows ∀x. DERIV f x :> 0 ==> f(x) = f(y)
apply (rule linorder-cases [of x y])
apply (blast intro: sym DERIV-isCont DERIV-isconst-end)+
done

lemma DERIV-const-ratio-const:
  fixes f :: real ⇒ real
  shows [|a ≠ b; ∀x. DERIV f x :> k |] ==> (f(b) - f(a)) = (b - a) * k
apply (rule linorder-cases [of a b], auto)
apply (drule-tac [|] f = f in MVT)
apply (auto dest: DERIV-isCont DERIV-unique simp add: real-differentiable-def)
apply (auto dest: DERIV-unique simp add: ring-distrib)
done

lemma DERIV-const-ratio-const2:
  fixes f :: real ⇒ real
```
apply (apply (apply (next lemma proof
THEORY "Deriv"
1622
next lemma real-average-minus-second [simp]: ((b + a) / 2 − a) = (b − a)/(2::real)
by (simp)
lemma DERIV-pos-imp-increasing-open:
fixes a :: real and b :: real and f :: real => real
assumes a < b and ∀ x. a < x => x < b => (EX y. DERIV f x => y & y > 0)
and con: ∀ x. a ≤ x => x ≤ b => isCont f x
shows f a < f b
proof (rule ccontr)
assume f: ~ f a < f b
have EX l z. a < z & z < b & DERIV f z => l 
& f b − f a = (b − a) * l
  apply (rule MVT)

Gallileo’s “trick”: average velocity = av. of end velocities

lemma DERIV-const-average:
fixes v :: real => real
assumes neq: a ≠ (b::real)
and der: ∀ x. DERIV v x => k
shows v ((a + b)/2) = (v a + v b)/2
proof (cases rule: linorder-cases [of a b])
  case equal with neq show ?thesis by simp
next
case less
  have (v b − v a) / (b − a) = k
    by (rule DERIV-const-ratio-const2 [OF neq der])
  hence (b−a) * ((v b − v a) / (b−a)) = (b−a) * k by simp
  moreover have (v ((a + b) / 2) − v a) / ((a + b) / 2 − a) = k
    by (rule DERIV-const-ratio-const2 [OF - der], simp add: neq)
  ultimately show ?thesis using neq by force
next
case greater
  have (v b − v a) / (b − a) = k
    by (rule DERIV-const-ratio-const2 [OF neq der])
  hence (b−a) * ((v b − v a) / (b−a)) = (b−a) * k by simp
  moreover have (v ((b + a) / 2) − v a) / ((b + a) / 2 − a) = k
    by (rule DERIV-const-ratio-const2 [OF - der], simp add: neq)
  ultimately show ?thesis using neq by (force simp add: add.commute)
qed

shows ||a ≠ b; ∀ x. DERIV f x := k || => (f(b) − f(a))/(b−a) = k
apply (rule-tac c1 = b−a in mult-right-cancel [THEN iffD1])
apply (auto dest!: DERIV-const-ratio-const simp add: mult.assoc)
done
using assms Deriv.differentiableI
apply force+
done
then obtain l z where z: a < z < b DERIV f z := l
and f b − f a = (b − a) * l
by auto
with assms f have ~ (l > 0)
by (metis linorder-not-le mult-le-0-iff diff-le-0-iff-le)
with assms z show False
by (metis DERIV-unique)
qed

lemma DERIV-pos-imp-increasing:
  fixes a::real and b::real and f::real
  assumes a < b and ∀ x. a ≤ x & x ≤ b --> (∃ y. DERIV f x := y & y > 0)
  shows f a < f b
by (metis DERIV-pos-imp-increasing-open [of a b f] assms DERIV-continuous less-imp-le)

lemma DERIV-nonneg-imp-nondecreasing:
  fixes a::real and b::real and f::real
  assumes a ≤ b and ∀ x. a ≤ x & x ≤ b --> (∃ y. DERIV f x := y & y ≥ 0)
  shows f a ≤ f b
proof (rule ccontr, cases a = b)
  assume A: ~ f a ≤ f b
  assume B: a ~ = b
  with assms have EX l z. a < z & z < b & DERIV f z := l
  & f b − f a = (b − a) * l
  apply −
  apply (rule MVT)
  apply auto
  apply (metis DERIV-isCont)
  apply (metis differentiableI less-le)
  done
then obtain l z where z: a < z < b DERIV f z := l
  and C: f b − f a = (b − a) * l
  by auto
with A have a < b f b < f a by auto
with C have l ≥ 0 by (auto simp add: not-le algebra-simps)
  (metis A add-le-cancel-right assms(1) less-eq-real-def mult-right-mono add-left-mono
  linear order-refl)
with assms z show False
by (metis DERIV-unique order-less-imp-le)
qed
lemma DERIV-neg-imp-decreasing-open:
  fixes a::real and b::real and f::real => real
  assumes a < b and \( \forall x. \ a < x \implies x < b \implies (\exists y. \ \text{DERIV} \ f \ x :> y \ \& \ y < 0) \)
  and con: \( \forall x. \ a \leq x \implies x \leq b \implies \text{isCont} \ f \)
  shows f a > f b
proof –
  have \((\forall x. \ - f \ x) \ a < (\forall x. \ - f \ x) \ b\)
  apply (rule DERIV-pos-imp-increasing-open [of a b \%x. \ - f x])
  using assms
  apply auto
  apply (metis field-differentiable-minus neg-0-less-iff-less)
  done
thus ?thesis
  by simp
qed

lemma DERIV-neg-imp-decreasing:
  fixes a::real and b::real and f::real => real
  assumes a < b and \( \forall x. \ a \leq x \ \& \ x \leq b \implies (\exists y. \ \text{DERIV} \ f \ x :> y \ \& \ y < 0) \)
  shows f a > f b
by (metis DERIV-neg-imp-decreasing-open [of a b f] assms DERIV-continuous less-imp-le)

lemma DERIV-nonpos-imp-nonincreasing:
  fixes a::real and b::real and f::real => real
  assumes a \leq b and \( \forall x. \ a \leq x \ \& \ x \leq b \implies (\exists y. \ \text{DERIV} \ f \ x :> y \ \& \ y \leq 0) \)
  shows f a \geq f b
proof –
  have \((\forall x. \ - f \ x) \ a \leq (\forall x. \ - f \ x) \ b\)
  apply (rule DERIV-nonneg-imp-nondecreasing [of a b \%x. \ - f x])
  using assms
  apply auto
  apply (metis DERIV-minus neg-0-le-iff-le)
  done
thus ?thesis
  by simp
qed

lemma DERIV-pos-imp-increasing-at-bot:
  fixes f :: real => real
  assumes \( \forall x. \ x \leq \ b \implies (\exists y. \ \text{DERIV} \ f \ x :> y \ \& \ y > 0) \)
  and lim: \( f \ \lim \rightarrow \ f \ \text{lim} \) at-bot
  shows f \lim < f b
proof –
  have \( f \lim \leq f \ (b - 1) \)
apply (rule tendsto-ge-const [OF - lim])
apply (auto simp: trivial-limit-at-bot-linorder eventually-at-bot-linorder)
apply (rule_tac x=b - 2 in exI)
apply (force intro: order.strict-implies-order DERIV-pos-imp-increasing [where f=f] assms)
done
also have \ldots < f b
  by (force intro: DERIV-pos-imp-increasing [where f=f] assms)
finally show \?thesis.
qed

lemma DERIV-neg-imp-decreasing-at-top:
  fixes f :: real
  assumes der: \( \forall x. x \geq b \Longrightarrow (EX y. DERIV f x :> y \& y < 0) \)
    and lim: \( f \leftarrow\rightarrow flim \) at-top
  shows flim < f b
  apply (rule DERIV-pos-imp-increasing-at-bot \[ where f = \lambda i. f (-i) \& b = -b, simplified])
  apply (metis DERIV-mirror der le-minus-iff neg-0-less-iff-less)
  apply (metis filterlim-at-top-mirror lim)
done

Derivative of inverse function

lemma DERIV-inverse-function:
  fixes f g :: real \Rightarrow real
  assumes der: \( \text{DERIV} f (g x) :> D \)
  assumes neq: \( D \neq 0 \)
  assumes a: \( a < x \) and \( b \): \( x < b \)
  assumes inj: \( \forall y. a < y \land y < b \longrightarrow f (g y) = y \)
  assumes cont: isCont g x
  shows \( \text{DERIV} g x :> \text{inverse} D \)
unfolding DERIV-iff2
proof (rule LIM-equal2)
  show \( 0 < \min (x - a) (b - x) \)
    using a b by arith
next
  fix y
  assume norm (y - x) < \( \min (x - a) (b - x) \)
  hence a < y and y < b
    by (simp-all add: abs-less-iff)
  thus \((g y - g x) / (y - x)) = \text{inverse} ((f (g y) - x) / (g y - g x))\)
    by (simp add: inj)
next
  have \( \lambda z. (f z - f (g x)) / (z - g x) \) -- \( g x \) --\( D \)
    by (rule der [unfolded DERIV-iff2])
  hence \( \lambda z. (f z - x) / (z - g x) \) -- \( g x \) --\( D \)
    using inj a b by simp
  have \( \exists d>0. \forall y. y \neq x \& \text{norm} (y - x) < d \longrightarrow g y \neq g x \)
proof (rule exI, safe)
  show \( \theta < \min (x - a) (b - x) \)
  using a b by simp

next
  fix y
  assume norm \((y - x) < \min (x - a) (b - x)\)
  hence y: \( a < y < b \)
  by (simp-all add: abs-less-iff)
  assume g y = g x
  hence \( f (g y) = f (g x) \) by simp
  hence y = x using inj y a b by simp
  also assume y \#= x
  finally show False by simp
qed

have \((\lambda y. (f (g y) - x) / (g y - g x)) = D\)
  using cont 1 2 by (rule isCont-LIM-compose2)
thus \((\lambda y. inverse ((f (g y) - x) / (g y - g x))) = inverse D\)
  using neq by (rule tendsto-inverse)
qed

103.10 Generalized Mean Value Theorem

theorem GMVT:
fixes a b :: real
assumes alb: \( a < b \)
and fc: \( \forall x. a \leq x \land x \leq b \rightarrow isCont f x \)
and fd: \( \forall x. a < x \land x < b \rightarrow f \) differentiable \( (at x) \)
and gc: \( \forall x. a \leq x \land x \leq b \rightarrow isCont g x \)
and gd: \( \forall x. a < x \land x < b \rightarrow g \) differentiable \( (at x) \)
shows \( \exists c \in (a, b). c \neq a \land c \neq b \land f' c = (g b - g a) / (b - a) \)
proof
  let \( h = \lambda x. (f b - f a) * (g x) - (g b - g a) * (f x) \)
  from assms have a < b by simp
  moreover have \( \forall x. a \leq x \land x \leq b \rightarrow isCont \ h x \)
  using fc gc by simp
  moreover have \( \forall x. a < x \land x < b \rightarrow h \) differentiable \( (at x) \)
  using fd gd by simp
  ultimately have \( \exists l z. a < z \land z < b \land \) DERIV \( h z : l \land h b - h a = (b - a) * l \) by (rule MVT)
  then obtain l where ldef: \( \exists z. a < z \land z < b \land \) DERIV \( h z : l \land h b - h a = (b - a) * l \) ..
  then obtain c where cdef: \( c < a \land c < b \land \) DERIV \( h c : l \land h b - h a = (b - a) * l \) ..
  from cdef have cint: \( a < c \land c < b \) by auto
  with gd have g differentiable \( (at c) \) by simp
hence \( \exists D. \text{DERIV} \ g \ c :> D \) by (rule differentiableD)
then obtain \( g'c \) where \( g'c\text{def}: \text{DERIV} \ g \ c :> g'c \) .

from \( \text{cdef} \) have \( a < c \land c < b \) by auto
with \( f \text{ld} \) have \( f \) differentiable \((\text{at} \ c)\) by simp
hence \( \exists D. \text{DERIV} \ f \ c :> D \) by (rule differentiableD)
then obtain \( f'c \) where \( f'c\text{def}: \text{DERIV} \ f \ c :> f'c \) .

from \( \text{cdef} \) have \( \text{DERIV} \ ?h \ c :> l \) by auto
moreover have \( \text{DERIV} \ ?h \ c :> g'c * (f b - f a) - f'c * (g b - g a) \)
using \( g'c\text{def} f'c\text{def} \) by (auto intro!: derivative-eq-intros)
ultimately have \( \text{leq: } l = g'c * (f b - f a) - f'c * (g b - g a) \) by (rule DERIV-unique)

\{
from \( \text{cdef} \) have \( ?h b - ?h a = (b - a) * l \) by auto
also from \( \text{leq} \) have \( ... = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) \) by simp
finally have \( ?h b - ?h a = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) \)
by simp
\}
moreover
\{
have \( ?h b - ?h a = ((f b)*g b - (f a)*(g b) - (g b)*(f b) + (g a)*(f b)) -
((f b)*g a - (f a)*(g a) - (g b)*(f a) + (g a)*(f a)) \)
by (simp add: algebra-simps)
hence \( ?h b - ?h a = 0 \) by auto
\}
ultimately have \((b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) = 0 \) by auto
with \( \text{ab} \) have \( g'c * (f b - f a) - f'c * (g b - g a) = 0 \) by simp
hence \( g'c * (f b - f a) = f'c * (g b - g a) \) by simp
hence \( (f b - f a) * g'c = (g b - g a) * f'c \) by (simp add: ac-simps)
with \( g'c\text{def} f'c\text{def} \) cint show \( \text{thesis} \) by auto
qed

lemma \( \text{GMVT}'\):
\( \text{fixes } f \ g :: \text{real } \Rightarrow \text{real} \)
assumes \( a < b \)
assumes \( \text{isCont} f: \bigwedge \ z. \ a \leq z \Longrightarrow z \leq b \Longrightarrow \text{isCont} f \ z \)
assumes \( \text{isCont} g: \bigwedge \ z. \ a \leq z \Longrightarrow z \leq b \Longrightarrow \text{isCont} g \ z \)
assumes \( \text{DERIV} g: \bigwedge \ z. \ a < z \Longrightarrow z < b \Longrightarrow \text{DERIV} g \ z :> (g' \ z) \)
assumes \( \text{DERIV} f: \bigwedge \ z. \ a < z \Longrightarrow z < b \Longrightarrow \text{DERIV} f \ z :> (f' \ z) \)
shows \( \exists c. \ a < c \land c < b \land (f b - f a) * g'c = (g b - g a) * f'c \)
proof
have \( \exists g'c f'c \ c. \text{DERIV} \ g \ c :> g'c \land \text{DERIV} \ f \ c :> f'c \land \)
\( a < c \land c < b \land (f b - f a) * g'c = (g b - g a) * f'c \)
using \( \text{assms} \) by (intro \( \text{GMVT} \)) (force simp: real-differentiable-def)+
then obtain $c$ where $a < c < b$ $(f b - f a) \ast g' c = (g b - g a) \ast f' c$
using DERIV-f DERIV-g by (force dest: DERIV-unique)
then show \textit{thesis}
by auto
qed

103.11 L’Hopital’s rule

\begin{verbatim}
lemma isCont-If-ge:
  fixes $a :: ''a :: linorder-topology$
  shows continuous (at-left $a$) $g$ \implies $(f -\neg-\neg> g a)$ (at-right $a$) \implies isCont $(\lambda x.
if $x \leq a$ then $g x$ else $f x$) $a$
unfolding isCont-def continuous-within
apply (intro filterlim-split-at)
apply (subst filterlim-cong [OF refl refl], where $g = g$]
apply (simp-all add: eventually-at-filter less-le)
apply (subst filterlim-cong [OF refl refl], where $g = f$]
apply (simp-all add: eventually-at-filter less-le)
done

lemma lhopital-right-0:
  fixes $f0$ $g0 :: ''a :: real \Rightarrow real$
  assumes $f$-$0$: $(f0 -\neg-\neg> 0)$ (at-right $0$)
  assumes $g$-$0$: $(g0 -\neg-\neg> 0)$ (at-right $0$)
  assumes $ev$:
  \hspace{1em} eventually $(\lambda x. g0 x \neq 0)$ (at-right $0$)
  \hspace{1em} eventually $(\lambda x. g' x \neq 0)$ (at-right $0$)
  \hspace{1em} eventually $(\lambda x. DERIV f0 x :> f' x)$ (at-right $0$)
  \hspace{1em} eventually $(\lambda x. DERIV g0 x :> g' x)$ (at-right $0$)
  assumes $lim$:
  \hspace{1em} $(\lambda x. (f' x / g' x)) -\neg-\neg> x$ (at-right $0$)
  shows $(\lambda x. f0 x / g0 x) -\neg-\neg> x$ (at-right $0$)
proof --
def $f \equiv \lambda x. \text{if } x \leq 0 \text{ then } 0 \text{ else } f0 x$
then have $f 0 = 0$ by simp

def $g \equiv \lambda x. \text{if } x \leq 0 \text{ then } 0 \text{ else } g0 x$
then have $g 0 = 0$ by simp

have eventually $(\lambda x. g0 x \neq 0 \land g' x \neq 0 \land
  DERIV f0 x :> (f' x) \land DERIV g0 x :> (g' x))$ (at-right $0$)
using $ev$ by eventually-elim auto
then obtain $a$ where [arith]: $0 < a$
  and $g0-neq-0$: $\forall x. 0 < x \implies x < a \implies g0 x \neq 0$
  and $g'-neq-0$: $\forall x. 0 < x \implies x < a \implies g' x \neq 0$
  and $f0$: $\forall x. 0 < x \implies x < a \implies DERIV f0 x :> (f' x)$
  and $g0$: $\forall x. 0 < x \implies x < a \implies DERIV g0 x :> (g' x)$
unfolding eventually-at by (auto simp: dist-real-def)

have g-neq-0: $\forall x. 0 < x \implies x < a \implies g x \neq 0$
\end{verbatim}
THEORY "Deriv"

using $g0$-neq-$0$ by (simp add: $g$-def)

{ fix $x$ assume $0 < x < a$ then have DERIV $f$ $x$ $(f' \ x)$
  by (intro DERIV-cong-ev [THEN iffD1, OF $\cdot\cdot\cdot f0$[OF $x$]])
  (auto simp: $f$-def eventually-nhds-metric dist-real-def intro!: exI[of - $x$]) }

note $f = \this$

{ fix $x$ assume $0 < x < a$ then have DERIV $g$ $x$ $(g' \ x)$
  by (intro DERIV-cong-ev [THEN iffD1, OF $\cdot\cdot\cdot g0$[OF $x$]])
  (auto simp: $g$-def eventually-nhds-metric dist-real-def intro!: exI[of - $x$]) }

note $g = \this$

have isCont $f$ $0$
  unfolding $f$-def by (intro isCont-If-ge $f$-$0$ continuous-const)

have isCont $g$ $0$
  unfolding $g$-def by (intro isCont-If-ge $g$-$0$ continuous-const)

have $\exists \zeta. \forall x \in \{0 <..< a\}. \ 0 < \zeta \ x \wedge \zeta \ x < x \wedge f \ x \ / \ g \ x = f' (\zeta \ x) / g' (\zeta \ x)$
proof (rule bchoice, rule)
  fix $x$ assume $x \in \{0 <..< a\}$
  then have $x$[arith]: $0 < x < a$ by auto
  with $g'$-neq-$0$ $g$-neq-$0$ $\langle g \ 0 = 0 \rangle$ have $g'$: $\forall x. \ 0 < x \Longrightarrow x < a \Longrightarrow 0 \neq g' \ x$
  by auto
  have $\forall x. \ 0 \leq x \Longrightarrow x < a \Longrightarrow \isCont f \ x$
  using isCont $f$ $0$. $f$ by (auto intro: DERIV-isCont simp: le-less)
  moreover have $\forall x. \ 0 \leq x \Longrightarrow x < a \Longrightarrow \isCont g \ x$
  using isCont $g$ $0$. $g$ by (auto intro: DERIV-isCont simp: le-less)
  ultimately have $\exists c. \ 0 < c \wedge c < x \wedge (f \ x - f \ 0) \ast g' \ c = (g \ x - g \ 0) \ast f' \ c$
  using $f \ g \ (x < a)$ by (intro GMVT') auto
  then obtain $c$ where $\ast: \ 0 < c \ c < x \ (f \ x - f \ 0) \ast g' \ c = (g \ x - g \ 0) \ast f' \ c$
  by blast
  moreover
  from $\ast \ g'(1)[of \ c]$ $g'(2)$ have $(f \ x - f \ 0) \ / \ (g \ x - g \ 0) = f' \ c \ / g' \ c$
  by (simp add: field-simps)
  ultimately show $\exists y. \ 0 < y \wedge y < x \wedge f \ x \ / \ g \ x = f' \ y \ / g' \ y$
  using $(f \ 0 = 0). \ (g \ 0 = 0)$ by (auto intro!: exI[of - $c$])
  qed

then obtain $\zeta$ where $\forall x \in \{0 <..< a\}. \ 0 < \zeta \ x \wedge \zeta \ x < x \wedge f \ x \ / \ g \ x = f' (\zeta \ x) / g' (\zeta \ x)$
  then have $\zeta$: eventually $(\lambda x. \ 0 < \zeta \ x \wedge \zeta \ x < x \wedge f \ x \ / \ g \ x = f' (\zeta \ x) / g' (\zeta \ x))$ (at-right $0$)
  unfolding eventually-at by (intro exI[of - $a$]) (auto simp: dist-real-def)
  moreover
  from $\zeta$ have eventually $(\lambda x. \ \norm(\zeta \ x) \leq x)$ (at-right $0$)
  by eventually-elim auto
  then have $(\lambda x. \ \norm(\zeta \ x)) \longrightarrow 0$ (at-right $0$)
  by (rule-tac real-tendsto-sandwich[where $f=\lambda x. \ 0$ and $h=\lambda x. \ x$])
THEORY "Deriv"

(auto intro: tendsto-const tendsto-ident-at)
then have \( \zeta \rightarrow 0 \) (at-right 0)
  by (rule tendsto-norm-zero-cancel)
with \( \zeta \) have filterlim \( \zeta \) (at-right 0) (at-right 0)
  by (auto elim!: eventually-elim1 simp: filterlim-at)
from this lim have \( ((\lambda t. f' (\zeta t) / g' (\zeta t)) \rightarrow x) \) (at-right 0)
  by (rule-tac filterlim-compose[of _ _ _])
ultimately have \( ((\lambda t. f t / g t) \rightarrow x) \) (at-right 0) (is \?P)
  by (rule-tac filterlim-cong[THEN iffD1, OF refl refl])
(auto elim: eventually-elim1)
also have \?P \iff \?thesis
  by (rule filterlim-cong) (auto simp: f-def g-def eventually-at-filter)
finally show \?thesis
  qed

lemma lhoptial-right:
\[
(f :: real \Rightarrow real) \rightarrow 0 \Rightarrow (g \rightarrow 0) \Rightarrow (at-right x) \\
\text{eventually (} \lambda x. \ g \ x \neq 0 \text{) (at-right x) } \\
\text{eventually (} \lambda x. \ g' \ x \neq 0 \text{) (at-right x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } f x \Rightarrow f' x \text{) (at-right x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } g x \Rightarrow g' x \text{) (at-right x) } \\
((\lambda x. \ (f' x / g' x)) \rightarrow y) \Rightarrow (at-right x) \\
((\lambda x. \ f x / g x) \rightarrow y) \Rightarrow (at-left x)
\]
unfolding eventually-at-right-to-0[of - x] filterlim-at-right-to-0[of - x] DERIV-shift
by (rule lhoptial-right-0)

lemma lhoptial-left:
\[
(f :: real \Rightarrow real) \rightarrow 0 \Rightarrow (g \rightarrow 0) \Rightarrow (at-left x) \\
\text{eventually (} \lambda x. \ g \ x \neq 0 \text{) (at-left x) } \\
\text{eventually (} \lambda x. \ g' \ x \neq 0 \text{) (at-left x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } f x \Rightarrow f' x \text{) (at-left x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } g x \Rightarrow g' x \text{) (at-left x) } \\
((\lambda x. \ (f' x / g' x)) \rightarrow y) \Rightarrow (at-left x) \\
((\lambda x. \ f x / g x) \rightarrow y) \Rightarrow (at-left x)
\]
unfolding eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror
by (rule lhoptial-right[where \( f' = \lambda x. \ f' (- x) \)]) (auto simp: DERIV-mirror)

lemma lhoptial:
\[
(f :: real \Rightarrow real) \rightarrow 0 \Rightarrow (g \rightarrow 0) \Rightarrow (at x) \\
\text{eventually (} \lambda x. \ g \ x \neq 0 \text{) (at x) } \\
\text{eventually (} \lambda x. \ g' \ x \neq 0 \text{) (at x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } f x \Rightarrow f' x \text{) (at x) } \\
\text{eventually (} \lambda x. \ \text{DERIV } g x \Rightarrow g' x \text{) (at x) } \\
((\lambda x. \ (f' x / g' x)) \rightarrow y) \Rightarrow (at x) \\
((\lambda x. \ f x / g x) \rightarrow y) \Rightarrow (at x)
\]
unfolding eventually-at-split filterlim-at-split
by (auto intro!: lhoptial-right[of \( f \ x \ g \ g' f' \)] lhoptial-left[of \( f \ x \ g \ g' f' \)]

lemma lhoptial-right-0-at-top:
fixes \( f \) :: real \( \Rightarrow \) real
assumes \( g \cdot 0 \): \text{LIM} \( x \) \text{ at-right} \( 0 \). \( g \ x \) \text{ at-top}
assumes \( \text{ev} \):
  eventually \((\lambda x. g' x \neq 0)\) (at-right \( 0 \))
  eventually \((\lambda x. \text{DERIV} f x \Rightarrow f' x)\) (at-right \( 0 \))
  eventually \((\lambda x. \text{DERIV} g x \Rightarrow g' x)\) (at-right \( 0 \))
assumes \( \text{lim} \):
\((\lambda x. (f' x / g' x)) \longrightarrow x)\) (at-right \( 0 \))
shows \((\lambda x. f x / g x) \longrightarrow x)\) (at-right \( 0 \))
unfolding tendsto-iff
proof safe

fix \( e :: \text{real} \) assume \( 0 < e \)

with \( \text{lim}[\text{unfolded tendsto-iff, rule-format, of} e / 4] \)
have eventually \((\lambda t. \text{dist} (f' t / g' t) x < e / 4)\) (at-right \( 0 \)) by simp
from eventually-conj \( OF \) eventually-conj \( OF \text{ev}(1) \) ev(2) eventually-conj \( OF \text{ev}(3) \) this]
obtain \( a \) where \([\text{arith}]: 0 < a \)
  and \( g' \cdot \text{neq}-0: \forall x. 0 < x \Rightarrow x < a \Rightarrow g' x \neq 0 \)
  and \( f0: \forall x. 0 < x \Rightarrow x < a \Rightarrow \text{DERIV} f x \Rightarrow (f' x) \)
  and \( g0: \forall x. 0 < x \Rightarrow x < a \Rightarrow \text{DERIV} g x \Rightarrow (g' x) \)
  and \( Df: \forall t. 0 < t \Rightarrow t < a \Rightarrow \text{dist} (f' t / g' t) x < e / 4 \)
unfolding eventually-at-le by (auto simp: dist-real-def)

from \( Df \) have
  eventually \((\lambda t. t < a)\) (at-right \( 0 \)) eventually \((\lambda t::\text{real}. 0 < t)\) (at-right \( 0 \))
unfolding eventually-at by (auto intro!: \text{exI}[of - a] simp: dist-real-def)

moreover
have eventually \((\lambda t. 0 < g t)\) (at-right \( 0 \)) eventually \((\lambda t. g a < g t)\) (at-right \( 0 \))
  using \( g \cdot 0 \) by (auto elim: eventually-elms simp: \text{filterlim-at-top-dense})

moreover
have inv-g: \( ((\lambda x. \text{inverse} (g x)) \longrightarrow 0)\) (at-right \( 0 \))
using tendsto-inverse-0 \text{filterlim-mono}[OF \( g \cdot 0 \) at-top-le-at-infinity order-refl]
by (rule \text{filterlim-compose})
then have \((\lambda x. \text{norm} (1 - g a * \text{inverse} (g x))) \longrightarrow \text{norm} (1 - g a * 0))\)
(at-right \( 0 \))
  by (intro tendsto-intros)
then have \((\lambda x. \text{norm} (1 - g a / g x)) \longrightarrow 1)\) (at-right \( 0 \))
  by (simp add: inverse-eq-divide)
from this[unfolded tendsto-iff, rule-format, of 1]
have eventually \((\lambda x. \text{norm} (1 - g a / g x) < 2)\) (at-right \( 0 \))
  by (auto elim!: eventually-elms simp: dist-real-def)

moreover
from inv-g have \((\lambda t. \text{norm} ((f a - x * g a) * \text{inverse} (g t))) \longrightarrow \text{norm} ((f a - x * g a) * 0))\)
  (at-right \( 0 \))
  by (intro tendsto-intros)
then have \((\lambda t. \text{norm}(f a - x * g a) / \text{norm}(g t)) \rightarrow\to 0)\) (at-right 0) by (simp add: inverse-eq-divide).

from this unfolded tendsto-iff, rule-format, of e / 2 \(0 < e\)

have eventually \((\lambda t. \text{norm}(f a - x * g a) / \text{norm}(g t) < e / 2)\) (at-right 0)

by (auto simp: dist-real-def)

ultimately show eventually \((\lambda t. \text{dist}(f t / g t) x < e)\) (at-right 0)

proof eventually-elim

fix \(t\) assume \(t\) [arith]: \(0 < t t < a g a < g t 0 < g t\)

assume ineq: \(\text{norm}(1 - g a / g t) < 2 \text{ norm}(f a - x * g a) / \text{norm}(g t) < e / 2\)

have \(\exists y. t < y \land y < a \land (g a - g t) * f' y = (f a - f t) * g' y\)

using \(f 0 g 0 t(1,2)\) by (intro GMVT') (force intro!: DERIV-isCont)+

then obtain \(y\) where [arith]: \(t < y y < a\)

and D-eq0: \((g a - g t) * f' y = (f a - f t) * g' y\)

by blast

from D-eq0 have D-eq: \((f t - f a) / (g t - g a) = f' y / g' y\)

using \((g a < g t) \lor g'\)-neq-0[of \(y\)] by (auto simp add: field-simps)

have \(\ast t / g t - x = ((f t - f a) / (g t - g a) - x) * (1 - g a / g t) + (f a - x * g a) / g t\)

by (simp add: field-simps)

have \(\text{norm}(f t / g t - x) \leq\)

\(\text{norm}(((f t - f a) / (g t - g a) - x) * (1 - g a / g t)) + \text{norm}(f a - x * g a) / \text{norm}(g t)\)

unfolding \(\ast\) by (rule norm-triangle-ineq)

also have \(\ldots = \text{dist}(f' y / g' y) x * \text{norm}(1 - g a / g t) + \text{norm}(f a - x * g a) / \text{norm}(g t)\)

by (simp add: abs-mult D-eq dist-real-def)

also have \(\ldots < (e / 4) * 2 + e / 2\)

using ineq \(DF[\text{of } y]:0 < e\) by (intro add-le-less-mono mult-mono) auto

finally show \(\text{dist}(f t / g t) x < e\)

by (simp add: dist-real-def)

qed

qed

lemma lhoptial-right-at-top:

\(\text{LIM } x \text{ at-right } x. (g::real \Rightarrow \text{real}) x := \text{at-top} \Rightarrow\)

eventually \((\lambda x. g' x \neq 0)\) (at-right \(x\)) \(\Rightarrow\)

eventually \((\lambda x. \text{DERIV} f x \Rightarrow f' x)\) (at-right \(x\)) \(\Rightarrow\)

eventually \((\lambda x. \text{DERIV} g x \Rightarrow g' x)\) (at-right \(x\)) \(\Rightarrow\)

\((\lambda x. (f' x / g' x)) \rightarrow\to y)\) (at-right \(x\)) \(\Rightarrow\)

\((\lambda x. (f' x / g x)) \rightarrow\to y)\) (at-right \(x\))

unfolding eventually-at-right-to-0[of \(-\ x\)] filterlim-at-right-to-0[of \(-\ x\)] DERIV-shift

by (rule lhoptial-right-0-at-top)

lemma lhoptial-left-at-top:

\(\text{LIM } x \text{ at-left } x. (g::real \Rightarrow \text{real}) x := \text{at-top} \Rightarrow\)
eventually \((\lambda x. \, g' \ x \neq 0)\) \((\text{at-left } x) \implies\)
eventually \((\lambda x. \, \text{DERIV } f \ x \rightarrow f' \ x)\) \((\text{at-left } x) \implies\)
eventually \((\lambda x. \, \text{DERIV } g \ x \rightarrow g' \ x)\) \((\text{at-left } x) \implies\)
\(((\lambda x. \, (f' \ x / g' \ x)) \rightarrow y)\) \((\text{at-left } x) \implies\)
\(((\lambda x. \, f \ x / g \ x) \rightarrow y)\) \((\text{at-left } x) \implies\)

unfolding eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror
by (rule lhopital-right-at-top[where \(f'=\lambda x. \, -f'(\, -x)\)]) (auto simp: DERIV-mirror)

lemma lhopital-at-top:
\(\text{LIM } x \; \text{at } x. \; (g :: \text{real } \Rightarrow \text{real}) \; \rightarrow \; \text{at-top} \implies\)
\(\text{eventually } \left(\lambda x. \, \, g' \ x \neq 0\right) \; \text{at-top} \implies\)
\(\text{eventually } \left(\lambda x. \, \text{DERIV } f \ x \rightarrow f' \ x\right) \; \text{at-top} \implies\)
\(\text{eventually } \left(\lambda x. \, \text{DERIV } g \ x \rightarrow g' \ x\right) \; \text{at-top} \implies\)
\(\left(\left(\lambda x. \, (f' \ x / g' \ x)\right) \rightarrow y\right) \; \text{at-top} \implies\)
\(\left(\left(\lambda x. \, f \ x / g \ x\right) \rightarrow y\right) \; \text{at-top} \implies\)

unfolding eventually-at-split filterlim-at-split
by (auto intro: lhopital-right-at-top[of \(g \ x \, g' \ f \ f'\)] lhopital-left-at-top[of \(g \ x \, g' \ f \ f'\)])

lemma lhopital-at-top-at-top:
fixes \(g \cdot f :: \text{real } \Rightarrow \text{real}\)
assumes \(g'\cdot0::\text{LIM } x \; \text{at-top}. \; g \ x \rightarrow \text{at-top}\)
assumes \(g'::\text{eventually } \left(\lambda x. \, \, g' \ x \neq 0\right) \; \text{at-top} \implies\)
assumes \(Df::\text{eventually } \left(\lambda x. \, \text{DERIV } f \ x \rightarrow f' \ x\right) \; \text{at-top} \implies\)
assumes \(Dg::\text{eventually } \left(\lambda x. \, \text{DERIV } g \ x \rightarrow g' \ x\right) \; \text{at-top} \implies\)
assumes \(lim::\left(\left(\lambda x. \, (f' \ x / g' \ x)\right) \rightarrow x\right) \; \text{at-top} \implies\)
shows \(\left(\left(\lambda x. \, f \ x / g \ x\right) \rightarrow x\right) \; \text{at-top} \implies\)

unfolding filterlim-at-top-to-right
proof (rule lhopital-right-0-at-top)
let \(\, ?F = \lambda x. \, f\) \((\text{inverse } x)\)
let \(\, ?G = \lambda x. \, g\) \((\text{inverse } x)\)
let \(\, ?R = \text{at-right } (0 :: \text{real})\)
let \(\, ?D = \lambda x. \, f'\) \((\text{inverse } x)\) \(\ast - \text{ (inverse } x \ast \text{ Suc } (\text{Suc } 0))\)

show \(\text{LIM } x \; ?R. \; ?G \ x \rightarrow \text{at-top}\)
using \(\, g \cdot0\) unfolding filterlim-at-top-to-right .

show eventually \((\lambda x. \, \text{DERIV } ?G \ x \rightarrow ?D \; g' \ x)\) \(?R\)
unfolding eventually-at-right-to-top
using \(Dg\) eventually-ge-at-top[where \(c=1::\text{real}\)]
apply eventually-elem
apply (rule DERIV-cong)
apply (rule DERIV-chain[where \(f=\text{inverse}\)])
apply (auto intro!: DERIV-inverse)
done

show eventually \((\lambda x. \, \text{DERIV } ?F \ x \rightarrow ?D \; f' \ x)\) \(?R\)
unfolding eventually-at-right-to-top
using \(Df\) eventually-ge-at-top[where \(c=1::\text{real}\)]
apply eventually-elim
apply (rule DERIV-cong)
apply (rule DERIV-chain[where f=inverse])
apply (auto intro: DERIV-inverse)
done

show eventually (λx. ?D g' x ≠ 0) ?R
  unfolding eventually-at-right-to-top
  using g' eventually-ge-at-top[where c=1::real]
  by eventually-elim auto

qed

end

104 NthRoot: Nth Roots of Real Numbers

theory NthRoot
imports Parity Deriv
begin

lemma abs-sgn-eq: abs (sgn x :: real) = (if x = 0 then 0 else 1)
  by (simp add: sgn-real-def)

lemma inverse-sgn: sgn (inverse a) = inverse (sgn a :: real)
  by (simp add: sgn-real-def)

lemma power-eq-iff-eq-base:
  fixes a b :: real
  shows 0 < n ⇒ 0 ≤ a ⇒ 0 ≤ b ⇒ a ^ n = b ^ n ←→ a = b
  using power-eq-imp-eq-base[of a n b] by auto

104.1 Existence of Nth Root
Existence follows from the Intermediate Value Theorem

lemma realpow-pos-nth:
  assumes n: 0 < n
  assumes a: 0 < a
  shows ∃r>0. r ^ n = (a::real)
proof
  have ∃r>0. r ≤ (max 1 a) ∧ r ^ n = a
    proof (rule IVT)
      show 0 ^ n ≤ a using n a by (simp add: power-0-left)
show \( \theta \leq \max 1 \ a \) by simp

from \( n \) have \( n1 : 1 \leq n \) by simp

have \( a \leq \max 1 \ a^1 \) by simp

also have \( \max 1 \ a^1 \leq \max 1 \ a^n \) using \( n1 \) by (rule power-increasing, simp)

finally show \( a \leq \max 1 \ a^n \).

show \( \forall r. \ 0 \leq r \land r \leq \max 1 \ a \pto \isCont (\lambda x. x^n) \ r \)

by simp

qed

then obtain \( r \) where \( r : 0 \leq r \land r^n = a \) by fast

with \( n \ a \) have \( r \neq 0 \) by (auto simp add: power-0-left)

thus \( ?thesis \)

qed

lemma realpow-pos-nth2: \( (\theta::real) < a \pto \exists r>0. \ r^n = a \)

by (blast intro: realpow-pos-nth)

Uniqueness of nth positive root

lemma realpow-pos-nth-unique: \[ \[ 0 < n; 0 < a \] \pto \exists !r. 0 < r \land r^n = (a::real) \]

by (auto intro!: realpow-pos-nth simp: power-eq-iff-eq-base)

104.2 Nth Root

We define roots of negative reals such that \( \text{root} \ n \ (-x) = -\text{root} \ n \ x \). This allows us to omit side conditions from many theorems.

lemma inj-sgn-power: assumes \( 0 < n \) shows inj \( (\lambda y. \ sgn \ y \ast |y|^n :: \text{real}) \) (is inj \( ?f \))

proof (rule injI)

have \( x : \forall a b :: \text{real}. \ (0 < a \land b < 0) \lor (a < 0 \land 0 < b) \pto a \neq b \) by auto

fix \( x y \) assume \( ?f x = ?f y \) with power-eq-iff-eq-base[of \( n \ |x| |y| \) \( < n \) show \( x = y \)

by (cases rule: linorder-cases[of \( 0 \ x \), case-product linorder-cases[of \( 0 \ y \)])

(simp-all add: \( x \))

qed

lemma sgn-power-injE: \( \text{sgn} \ a \ast |a|^n = x \pto x = \text{sgn} \ b \ast |b|^n \pto 0 < n \)

\( \pto a = (b::real) \)

using inj-sgn-power[THEN injD, of \( n \ a \ b \)] by simp

definition root :: \( \text{nat} \pto \text{real} \pto \text{real} \) where

\( \text{root} \ n \ x = (\text{if} \ n = 0 \ \text{then} \ 0 \ \text{else} \ \text{the-inv} (\lambda y. \ sgn \ y \ast |y|^n) \ x) \)

lemma root-0 [simp]: \( \text{root} \ 0 \ x = 0 \)

by (simp add: root-def)

lemma root-sgn-power: \( 0 < n \pto \text{root} \ n \ (\text{sgn} \ y \ast |y|^n) = y \)
using the-inv-f[OF inj-sgn-power] by (simp add: root-def)

lemma sgn-power-root:
assumes \( 0 < n \)
shows \( sgn (root n x) \cdot (root n x)^n = x \) (is \( ?f \) (root n x) = x)
proof cases
assume \( x \neq 0 \)
with realpow-pos-nth[OF \( 0 < n \), of \( x \)] obtain \( r \) where \( 0 < r \cdot r^n = |x| \) by auto
with \( x \neq 0 \) have \( S : x \in \text{range } ?f \) by (intro image-eqI[of - - sgn x * r]) (auto simp: abs-mult sgn-mult power-mult-distrib abs-sgn-eq mult-sgn-abs)
from \( 0 < n \) f-the-inv-into-f[OF inj-sgn-power[OF \( 0 < n \)]] this show \( ?\text{thesis} \) by (simp add: root-def)
qed (insert \( 0 < n \) root-sgn-power[of n 0], simp)

lemma split-root: \( P (\text{root } n x) \iff (n = 0 \implies P 0) \land (0 < n \implies (\forall y. sgn y \cdot |y|^n = x \implies P y)) \)
apply (cases \( n = 0 \))
apply simp-all
apply (metis root-sgn-power sgn-power-root)
done

lemma real-root-zero [simp]: \( \text{root } n 0 = 0 \)
by (simp split: split-root add: sgn-zero-iff)

lemma real-root-minus: \( \text{root } n (-x) = - \text{root } n x \)
by (clarsimp split: split-root elim!: sgn-power-injE simp: sgn-minus)

lemma real-root-less-mono: \( \[0 < n; x < y\] \implies \text{root } n x < \text{root } n y \)
proof (clarsimp split: split-root)
have \( x : \bigwedge a b : \text{real}. (0 < b \land a < 0) \implies \neg a > b \) by auto
fix \( a b : \text{real} \) assume \( \[0 < n \land sgn a \cdot |a|^n < sgn b \cdot |b|^n \] \) then show \( a < b \)
using power-less-imp-less-base[of a n b] power-less-imp-less-base[of \( -b n-a \)]
by (simp add: sgn-real-def power-less-zero-eq x[of a \( ^n = ((- b) \cdot n) \) split: split-if-asm])
qed

lemma real-root-gt-zero: \( \[0 < n; 0 < x\] \implies 0 < \text{root } n x \)
using real-root-less-mono[of n 0 x] by simp

lemma real-root-ge-zero: \( 0 \leq x \implies 0 \leq \text{root } n x \)
using real-root-gt-zero[of n x] by (cases \( n = 0 \)) (auto simp add: le-less)

lemma real-root-pow-pos:
\( \[0 < n; 0 < x\] \implies \text{root } n x \cdot n = x \)
using sgn-power-root[of n x] real-root-gt-zero[of n x] by simp

lemma real-root-pow-pos2 [simp]:
THEORY "NthRoot"

\[ \text{if } 0 < n; 0 \leq x \text{ then } \text{root } n \ x \ ^{\ n} = x \]
by (auto simp add: order-le-less real-root-pow-pos)

lemma sgn-root: \(0 < n \Rightarrow \text{sgn} (\text{root } n \ x) = \text{sgn} x\)
by (auto split: split-root simp: sgn-real-def power-less-zero-eq)

lemma odd-real-root-pow: odd \(n \Rightarrow \text{root } n \ x \ ^{\ n} = x\)
using sgn-power-root[of \(n \ x\)] by (simp add: odd-pos sgn-real-def split: split-if-asm)

lemma real-root-power-cancel: \(0 < n; 0 \leq x \text{ then } \text{root } n \ (x \ ^{\ n}) = x\)
using root-sgn-power[of \(n \ y\)] by (auto simp add: le-less power-0-left)

lemma odd-real-root-power-cancel: odd \(n \Rightarrow \text{root } n \ (x \ ^{\ n}) = x\)
using root-sgn-power[of \(n \ x\)] by (simp add: odd-pos sgn-real-def power-0-left split: split-if-asm)

lemma real-root-pos-unique: \(0 < n; 0 \leq y; y \ ^{\ n} = x \Rightarrow \text{root } n \ x = y\)
using root-sgn-power[of \(n \ y\)] by (auto simp add: le-less power-0-left)

lemma odd-real-root-unique:
\([\text{odd } n; y \ ^{\ n} = x] \Rightarrow \text{root } n \ x = y\)
by (erule subst, rule odd-real-root-power-cancel)

lemma real-root-one [simp]: \(0 < n \Rightarrow \text{root } n \ 1 = 1\)
by (simp add: real-root-pos-unique)

Root function is strictly monotonic, hence injective

lemma real-root-le-mono: \(0 < n; x \leq y \Rightarrow \text{root } n \ x \leq \text{root } n \ y\)
by (auto simp add: order-le-less real-root-less-mono)

lemma real-root-less-iff [simp]:
\(0 < n \Rightarrow (\text{root } n \ x < \text{root } n \ y) = (x < y)\)
apply (cases \(x < y\))
apply (simp add: real-root-less-mono)
apply (simp add: linorder-not-less real-root-le-mono)
done

lemma real-root-le-iff [simp]:
\(0 < n \Rightarrow (\text{root } n \ x \leq \text{root } n \ y) = (x \leq y)\)
apply (cases \(x \leq y\))
apply (simp add: real-root-le-mono)
apply (simp add: linorder-not-le real-root-less-mono)
done

lemma real-root-eq-iff [simp]:
\(0 < n \Rightarrow (\text{root } n \ x = \text{root } n \ y) = (x = y)\)
by (simp add: order-eq-iff)

lemmas real-root-gt-0-iff [simp] = real-root-less-iff [where \(x = 0\), simplified]
lemmas real-root-lt-0-iff [simp] = real-root-less-iff [where y=0, simplified]
lemmas real-root-le-0-iff [simp] = real-root-le-iff [where x=0, simplified]
lemmas real-root-eq-0-iff [simp] = real-root-eq-iff [where y=0, simplified]

lemma real-root-gt-1-iff [simp]: 0 < n =⇒ (1 < root n y) = (1 < y)
by (insert real-root-less-iff [where x=1], simp)

lemma real-root-lt-1-iff [simp]: 0 < n =⇒ (root n x < 1) = (x < 1)
by (insert real-root-less-iff [where y=1], simp)

lemma real-root-le-1-iff [simp]: 0 < n =⇒ (1 ≤ root n y) = (1 ≤ y)
by (insert real-root-le-iff [where x=1], simp)

lemma real-root-eq-1-iff [simp]: 0 < n =⇒ (root n x = 1) = (x = 1)
by (insert real-root-eq-iff [where y=1], simp)

Roots of multiplication and division

lemma real-root-mult: root n (x * y) = root n x * root n y
by (auto split: split-root elim!: sgn-power-injE simp: sgn-mult abs-mult power-mult-distrib)

lemma real-root-inverse: root n (inverse x) = inverse (root n x)
by (auto split: split-root elim!: sgn-power-injE simp: inverse-sgn power-inverse)

lemma real-root-divide: root n (x / y) = root n x / root n y
by (simp add: divide-inverse real-root-mult real-root-inverse)

lemma real-root-abs: 0 < n =⇒ root n |x| = |root n x|
by (simp add: abs-if real-root-minus)

lemma real-root-power: 0 < n =⇒ root n (x ^ k) = root n x ^ k
by (induct k) (simp-all add: real-root-mult)

Roots of roots

lemma real-root-Suc-0 [simp]: root (Suc 0) x = x
by (simp add: odd-real-root-unique)

lemma real-root-mult-exp: root (m * n) x = root m (root n x)
by (auto split: split-root elim!: sgn-power-injE
    simp: sgn-zero-iff sgn-mult power-mult[symmetric] abs-mult power-mult-distrib
    abs-sgn-eq)

lemma real-root-commute: root m (root n x) = root n (root m x)
by (simp add: real-root-mult-exp [symmetric] mult.commute)

Monotonicity in first argument
lemma real-root-strict-decreasing:
\[ 0 < n; n < N; 1 < x \Rightarrow \sqrt[N]{x} < \sqrt[n]{x} \]
apply (subgoal-tac root n (root N x) ^ n < root n (root N x) ^ N, simp)
done

lemma real-root-strict-increasing:
\[ 0 < n; n < N; 0 < x; x < 1 \Rightarrow \sqrt[n]{x} < \sqrt[N]{x} \]
apply (simp add: real-root-commute power-strict-decreasing del: real-root-pow-pos2)
done

lemma real-root-decreasing:
\[ 0 < n; n < N; 1 \leq x \Rightarrow \sqrt[N]{x} \leq \sqrt[n]{x} \]
by (auto simp add: order-le-less real-root-strict-decreasing)

lemma real-root-increasing:
\[ 0 < n; n < N; 0 \leq x; x \leq 1 \Rightarrow \sqrt[n]{x} \leq \sqrt[N]{x} \]
by (auto simp add: order-le-less real-root-strict-increasing)

Continuity and derivatives

lemma isCont-real-root: isCont (root n) x
proof cases
  assume n: 0 < n
  let \( \bar{f} = \lambda y::real. sgn y \cdot |y|^n \)
  have continuous-on \{0..\} \cup \{.. 0\} \( \lambda x. if 0 < x then x ^ n else -(((-x) ^ n)) :: real \)
  using n by (intro continuous-on-If continuous-intros) auto
  then have continuous-on UNIV \( \bar{f} \)
  by (rule continuous-on-cong[THEN iffD1, rotated 2]) (auto simp: not-less real-sgn-neg le-less n)
  then have \( simp; \forall x. isCont \bar{f} x \)
  by (simp add: continuous-on-eq-continuous-at)
  have isCont (root n) \( \bar{f} (root n x) \)
  by (rule isCont-inverse-function [where \( f=\bar{f} \) and \( d=1 \)]) (auto simp: root-sgn-power n)
  then show \( ?thesis \)
  by (simp add: sgn-power-root n)
qed (simp add: root-def[abs-def])

lemma tendsto-real-root[tendsto-intros]:
\( f \quad \text{tendsto} \quad x \Rightarrow (\lambda x. \text{root n (f x)}) \quad \text{tendsto} \quad \text{root n x} \)
using isCont-tendsto-compose[of isCont-real-root, of f x F].

lemma continuous-real-root[continuous-intros]:
continuous \( F \) \( f \Rightarrow \) continuous \( F \) \( (\lambda x. \text{root n (f x)}) \)
unfolding continuous-def by (rule tendsto-real-root)

lemma continuous-on-real-root[continuous-intros]:
  continuous-on s f \implies continuous-on s (λx. root n (f x))
unfolding continuous-on-def by (auto intro: tendsto-real-root)

lemma DERIV-real-root:
  assumes n: 0 < n
  assumes x: 0 < x
  shows DERIV (root n) x :> inverse (real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show 0 < x using x .
  show \forall y. 0 < y \land y < x + 1 \implies root n y ^ n = y
    using n by simp
  show DERIV (λx. x ^ n) (root n x) :> real n * root n x ^ (n - Suc 0)
    by (rule DERIV-pow)
  show real n * root n x ^ (n - Suc 0) ≠ 0
    using n x by simp
qed (rule isCont-real-root)

lemma DERIV-odd-real-root:
  assumes n: odd n
  assumes x: x ≠ 0
  shows DERIV (root n) x :> inverse (real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show x - 1 < x by simp
  show x < x + 1 by simp
  show \forall y. x - 1 < y \land y < x + 1 \implies root n y ^ n = y
    using n by (simp add: odd-real-root-pow)
  show DERIV (λx. x ^ n) (root n x) :> real n * root n x ^ (n - Suc 0)
    by (rule DERIV-pow)
  show real n * root n x ^ (n - Suc 0) ≠ 0
    using odd-pos [OF n] x by simp
qed (rule isCont-real-root)

lemma DERIV-even-real-root:
  assumes n: 0 < n and even n
  assumes x: x < 0
  shows DERIV (root n) x :> inverse (- real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show x - 1 < x by simp
  show x < 0 using x .
next
  show \forall y. x - 1 < y \land y < 0 \implies (- root n y ^ n) = y
proof (rule allI, rule implI, erule conjE)
  fix y assume x - 1 < y and y < 0
  hence root n (-y) ^ n = -y using (0 < n) by simp
  with real-root-minus and (even n)
show \( - (\text{root } n \ y \ ^n) = y \) by simp

qed

next

show DERIV (\( \lambda x. - (x ^n) \)) (\( \text{root } n \ x \)) :\( > - \text{real } n * \text{root } n \ x ^{(n - \text{Suc } 0)} \)
  by (auto intro!: derivative-eq-intros simp: real-of-nat-def)

show \(- \text{real } n * \text{root } n \ x ^{(n - \text{Suc } 0)} \neq 0 \)
  using \( n \ x \) by simp

qed (rule isCont-real-root)

lemma DERIV-real-root-generic:
  assumes \( 0 < n \) and \( x \neq 0 \)
  and \( \text{even } n ; 0 < x \) \( \Rightarrow \) \( D = \text{inverse} (\text{real } n * \text{root } n \ x ^{(n - \text{Suc } 0)}) \)
  and \( \text{even } n ; x < 0 \) \( \Rightarrow \) \( D = - \text{inverse} (\text{real } n * \text{root } n \ x ^{(n - \text{Suc } 0)}) \)
  and \( \text{odd } n \Rightarrow D = \text{inverse} (\text{real } n * \text{root } n \ x ^{(n - \text{Suc } 0)}) \)
  shows DERIV (\( \text{root } n \)) \( x \) :\( > D \)
  using assms by (cases even n, cases \( 0 < x \),
  auto intro: DERIV-real-root[THEN DERIV-cong]
  DERIV-odd-real-root[THEN DERIV-cong]
  DERIV-even-real-root[THEN DERIV-cong])

104.3 Square Root

definition sqrt :: real \( \Rightarrow \) real where
  \( \text{sqrt} = \text{root } 2 \)

lemma pos2: \( 0 < (2::nat) \) by simp

lemma real-sqrt-unique: \( [y^2 = x; 0 \leq y] \Rightarrow \text{sqrt } x = y \)
  unfolding sqrt-def by (rule real-root-pos-unique [OF pos2])

lemma real-sqrt-abs [simp]: \( \text{sqrt } (x^2) = |x| \)
  apply (rule real-sqrt-unique)
  apply (rule power2-abs)
  apply (rule abs-ge-zero)
  done

lemma real-sqrt-pow2 [simp]: \( 0 \leq x \Rightarrow (\text{sqrt } x)^2 = x \)
  unfolding sqrt-def by (rule real-root-pow-pos2 [OF pos2])

lemma real-sqrt-pow2-iff [simp]: \( (\text{sqrt } x)^2 = x \) = \( (0 \leq x) \)
  apply (rule iffI)
  apply (erule subst)
  apply (rule zero-le-power2)
  apply (erule real-sqrt-pow2)
  done

lemma real-sqrt-zero [simp]: \( \text{sqrt } 0 = 0 \)
  unfolding sqrt-def by (rule real-root-zero)
**THEORY “NthRoot”**

```isar
lemma real-sqrt-one [simp]: sqrt 1 = 1
  unfolding sqrt-def by (rule real-root-one [OF pos2])

lemma real-sqrt-four [simp]: sqrt 4 = 2
  using real-sqrt-abs[of 2] by simp

lemma real-sqrt-minus: sqrt (- x) = - sqrt x
  unfolding sqrt-def by (rule real-root-minus)

lemma real-sqrt-mult: sqrt (x * y) = sqrt x * sqrt y
  unfolding sqrt-def by (rule real-root-mult)

lemma real-sqrt-mult-self [simp]: sqrt a * sqrt a = |a|

lemma real-sqrt-inverse: sqrt (inverse x) = inverse (sqrt x)
  unfolding sqrt-def by (rule real-root-inverse [OF pos2])

lemma real-sqrt-divide: sqrt (x / y) = sqrt x / sqrt y
  unfolding sqrt-def by (rule real-root-divide)

lemma real-sqrt-power: sqrt (x ^ k) = sqrt x ^ k
  unfolding sqrt-def by (rule real-root-power [OF pos2])

lemma real-sqrt-gt-zero: 0 < x ⇒ 0 < sqrt x
  unfolding sqrt-def by (rule real-root-gt-zero [OF pos2])

lemma real-sqrt-ge-zero: 0 ≤ x ⇒ 0 ≤ sqrt x
  unfolding sqrt-def by (rule real-root-ge-zero)

lemma real-sqrt-less-mono: x < y ⇒ sqrt x < sqrt y
  unfolding sqrt-def by (rule real-root-less-mono [OF pos2])

lemma real-sqrt-le-mono: x ≤ y ⇒ sqrt x ≤ sqrt y
  unfolding sqrt-def by (rule real-root-le-mono [OF pos2])

lemma real-sqrt-less-iff [simp]: (sqrt x < sqrt y) = (x < y)
  unfolding sqrt-def by (rule real-root-less-iff [OF pos2])

lemma real-sqrt-le-iff [simp]: (sqrt x ≤ sqrt y) = (x ≤ y)
  unfolding sqrt-def by (rule real-root-le-iff [OF pos2])

lemma real-sqrt-eq-iff [simp]: (sqrt x = sqrt y) = (x = y)
  unfolding sqrt-def by (rule real-root-eq-iff [OF pos2])

lemma real-le-lsqrt: 0 ≤ x ⇒ 0 ≤ sqrt y
  using real-sqrt-le-iff[of x y] by simp

lemma real-le-rsqrt: x^2 ≤ y ⇒ x ≤ sqrt y
```

using real-sqrt-le-mono[of x^2 y] by simp

lemma real-less-rsqrt: \( x^2 < y \implies x < \sqrt{y} \)
using real-sqrt-less-mono[of x^2 y] by simp

lemma sqrt-even-pow2:
assumes n: even n
shows \( \sqrt{(2 ^ n)} = 2 ^ (n \div 2) \)
proof
- from n obtain m where m: n = 2 * m
  unfolding even-mult-two-ex ..
  from m have \( \sqrt{(2 ^ m)} = \sqrt{(2 ^ m \cdot 2)} \)
  by (simp only: power-mult[symmetric] mult.commute)
then show \?thesis
  using m by simp
qed

lemmas real-sqrt-gt-0-iff [simp] = real-sqrt-less-iff[where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-lt-0-iff [simp] = real-sqrt-less-iff[where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-ge-0-iff [simp] = real-sqrt-le-iff[where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-le-0-iff [simp] = real-sqrt-le-iff[where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-eq-0-iff [simp] = real-sqrt-eq-iff[where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-gt-1-iff [simp] = real-sqrt-less-iff[where x=1, unfolded real-sqrt-one]
lemmas real-sqrt-lt-1-iff [simp] = real-sqrt-less-iff[where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-ge-1-iff [simp] = real-sqrt-le-iff[where x=1, unfolded real-sqrt-one]
lemmas real-sqrt-le-1-iff [simp] = real-sqrt-le-iff[where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-eq-1-iff [simp] = real-sqrt-eq-iff[where y=1, unfolded real-sqrt-one]

lemma isCont-real-sqrt: isCont sqrt x
unfolding sqrt-def by (rule isCont-real-root)

lemma tendsto-real-sqrt[tendsto-intros]:
  (\( \lambda x. \sqrt{x} \)) \( \to \) \( x \)
unfolding sqrt-def by (rule tendsto-real-root)

lemma continuous-real-sqrt[continuous-intros]:
  continuous \( \sqrt{f} \) \( \to \) continuous \( \sqrt{f} \)
unfolding sqrt-def by (rule continuous-real-root)

lemma continuous-on-real-sqrt[continuous-intros]:
  continuous-on \( s \) \( \sqrt{f} \) \( \to \) continuous-on \( s \) \( \sqrt{f} \)
unfolding sqrt-def by (rule continuous-on-real-root)

lemma DERIV-real-sqrt-generic:
assumes x \neq 0
assumes x > 0 \implies D = inverse (sqrt x) / 2
assumes x < 0 \implies D = - inverse (sqrt x) / 2
shows DERIV sqrt x \( \to \) D
using assms unfolding sqrt-def
by (auto intro: DERIV-real-root-generic)

lemma DERIV-real-sqrt:
\[ 0 < x \implies \text{DERIV} \sqrt{x} :> \frac{1}{2} \]
using DERIV-real-sqrt-generic by simp

declare DERIV-real-sqrt-generic[THEN DERIV-chain2, derivative-intros]
derivative-intros

lemma not-real-square-gt-zero [simp]: \(\sim (0 :: \text{real}) < x \cdot x) = (x = 0)\)
apply auto
apply (cut-tac x = x and y = 0 in linorder-less-linear)
apply (simp add: zero-less-mult-iff)
done

lemma real-sqrt-abs2 [simp]: \(\sqrt{x \cdot x}) = |x|\)
apply (subst power2-eq-square [symmetric])
apply (rule real-sqrt-abs)
done

lemma real-inv-sqrt-pow2: \(0 < x \implies (\frac{1}{\sqrt{x}})^2 = \frac{1}{x}\)
by (simp add: power-inverse [symmetric])

lemma real-sqrt-eq-zero-cancel: \([|0 \leq x; \sqrt{x} = 0]|) \Longrightarrow x = 0\)
by simp

lemma real-sqrt-ge-one: \(1 \leq x \Longrightarrow 1 \leq \sqrt{x}\)
by simp

lemma sqrt-divide-self-eq:
assumes nneg: \(0 \leq x\)
shows \(\sqrt{x} / x = \frac{1}{\sqrt{x}}\)
proof cases
  assume x=0 thus \(?thesis by simp
next
  assume nz: \(x \neq 0\)
  hence pos: \(0 < x\) using nneg by arith
  show \(?thesis
proof (rule right-inverse-eq [THEN iffD1, THEN sym])
  show \(\sqrt{x} / x \neq 0\) by (simp add: divide-inverse nneg nz)
  show \(\frac{1}{\sqrt{x}} / (\sqrt{x} / x) = 1\)
    by (simp add: divide-inverse mult_assoc [symmetric]
        power2-eq-square [symmetric] real-inv-sqrt-pow2 pos nz)
  qed
  qed
lemma real-div-sqrt: \(0 \leq x \implies x / \sqrt{x} = \sqrt{x}\)
apply (cases x = 0)
apply simp-all
using sqrt-divide-self-eq[of x]
apply (simp add: inverse-eq-divide field-simps)
done

lemma real-divide-square-eq [simp]: \((r::real) * a) / (r * r) = a / r
apply (simp add: divide-inverse)
apply (case-tac r=0)
apply (auto simp add: ac-simps)
done

lemma lemma-real-divide-sqrt-less: 0 < u ==> u / sqrt 2 < u
by (simp add: divide-less-eq)

lemma four-x-squared:
fixes x::real
shows \(4 * x^2 = (2 * x)^2\)
by (simp add: power2-eq-square)

lemma sqrt-at-top:
LIM x at-top. sqrt x :: real
by (rule filterlim-at-top-at-top [where Q=\(\lambda x. True\) and P=\(\lambda x. 0 < x\) and g=power2])
  (auto intro: eventually-gt-at-top)

104.4 Square Root of Sum of Squares

lemma sum-squares-bound:
fixes x::'a::linordered-field
shows \(2 * x * y \leq x^2 + y^2\)
proof
  have \((x-y)^2 = x^2 - 2 * x * y + y^2\)
    by algebra
  then have \(0 \leq x^2 - 2 * x * y + y^2\)
    by (metis sum-power2-ge-zero zero-le-double-add-iff-zero-le-single-add power2-eq-square)
  then show \(?thesis\)
    by arith
qed

lemma arith-geo-mean:
fixes u::'a::linordered-field assumes \(u^2 = x * y\) \(x \geq 0\) \(y \geq 0\) shows \(u \leq (x + y)/2\)
apply (rule power2-le-imp-le)
using sum-squares-bound assms
apply (auto simp: zero-le-mult-iff)
by (auto simp: algebra-simps power2-eq-square)

lemma arith-geo-mean-sqrt:
fixes x::real assumes \(x \geq 0\) \(y \geq 0\) shows \(sqrt(x * y) \leq (x + y)/2\)
apply (rule arith-geo-mean)
THEORY “NthRoot”

using assms
apply (auto simp: zero-le-mult-iff)
done

lemma real-sqrt-sum-squares-mult-ge-zero [simp]:
  \( \sqrt{(x^2 + y^2)(xa^2 + ya^2)} \)
by (metis real-sqrt-ge-0-iff split-mult-pos-le sum-power2-ge-zero)

lemma real-sqrt-sum-squares-mult-squared-eq [simp]:
  \( \sqrt{(x^2 + y^2)(xa^2 + ya^2)} \)^2 = (x^2 + y^2) \cdot (xa^2 + ya^2)
by (simp add: zero-le-mult-iff)

lemma real-sqrt-sum-squares-eq-cancel [simp]:
  \( \sqrt{x^2 + y^2} = x \Rightarrow y = 0 \)
by (drule-tac f = %x. x^2 in arg-cong, simp)

lemma real-sqrt-sum-squares-eq-cancel2 [simp]:
  \( \sqrt{x^2 + y^2} = y \Rightarrow x = 0 \)
by (drule-tac f = %x. x^2 in arg-cong, simp)

lemma real-sqrt-sum-squares-ge1 [simp]:
  \( x \leq \sqrt{x^2 + y^2} \)
by (rule power2-le-imp-le, simp-all)

lemma real-sqrt-sum-squares-ge2 [simp]:
  \( y \leq \sqrt{x^2 + y^2} \)
by (rule power2-le-imp-le, simp-all)

lemma real-sqrt-ge-abs1 [simp]:
  \( |x| \leq \sqrt{x^2 + y^2} \)
by (rule power2-le-imp-le, simp-all)

lemma real-sqrt-ge-abs2 [simp]:
  \( |y| \leq \sqrt{x^2 + y^2} \)
by (rule power2-le-imp-le, simp-all)

lemma le-real-sqrt-sumsq [simp]:
  \( x \leq \sqrt{x \cdot y} \)
by (simp add: power2-eq-square [symmetric])

lemma real-sqrt-sum-squares-triangle-ineq:
  \( \sqrt{(a + c)^2 + (b + d)^2} \leq \sqrt{a^2 + b^2} + \sqrt{c^2 + d^2} \)
apply (rule power2-le-imp-le, simp)
apply (simp add: power2-sum)
apply (simp only: mult.assoc distrib-left [symmetric])
apply (rule mult-left-mono)
apply (rule power2-le-imp-le)
apply (simp add: power2-sum power-mult-distrib)
apply (simp add: ring-distribs)
apply (subgoal-tac 0 \leq b^2 \cdot c^2 + a^2 \cdot d^2 - 2 \cdot (a \cdot c) \cdot (b \cdot d), simp)
apply (rule-tac b=(a \cdot d - b \cdot c)^2 in ord-le-eq-trans)
apply (rule zero-le-power2)
apply (simp add: power2-diff power-mult-distrib)
apply (simp)
apply simp
apply (simp add: add-increasing)
done

lemma real-sqrt-sum-squares-less:
  \[ |x| < u / \sqrt{2}; |y| < u / \sqrt{2} \implies \sqrt{x^2 + y^2} < u \]
apply (rule power2-less-imp-less, simp)
apply (erule power-strict-mono [OF - abs-ge-zero pos2])
apply (simp add: power-divide)
apply (erule order-le-less-trans [OF abs-ge-zero])
apply (simp add: zero-less-divide-iff)
done

Needed for the infinitely close relation over the nonstandard complex numbers

lemma lemma-sqrt-hcomplex-capprox:
  \[ |0 < u; x < u/2; y < u/2; 0 \leq x; 0 \leq y | \implies \sqrt{x^2 + y^2} < u \]
apply (rule_tac y = u/sqrt 2 in order-le-less-trans)
apply (erule_tac [2] lemma-real-divide-sqrt-less)
apply (rule power2-le-imp-le)
apply (auto simp add: zero-le-divide-iff power-divide)
apply (rule_tac t = u in real-sum-of-halves [THEN subst])
apply (rule add-mono)
apply (auto simp add: four-x-squared intro: power-mono)
done

Legacy theorem names:

lemmas real-root-pos2 = real-root-power-cancel
lemmas real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]
lemmas real-root-pos-pos-le = real-root-ge-zero
lemmas real-sqrt-mult-distrib = real-sqrt-mult
lemmas real-sqrt-mult-distrib2 = real-sqrt-mult
lemmas real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff

lemma real-root-pos: 0 < x \implies \text{root}(Suc n) (x' ^ (Suc n)) = x
by (rule real-root-power-cancel [OF zero-less-Suc order-less-imp-le])

end

105 Transcendental: Power Series, Transcendental Functions etc.

theory Transcendental
imports Fact Series Deriv NthRoot
begin

lemma root-test-convergence:
  fixes f :: nat \Rightarrow 'a::banach
assumes \( f: (\lambda n. \text{root } n (\text{norm } f n)) \rightarrow x \) — could be weakened to \( \text{lim sup} \)

assumes \( x < 1 \)

shows summable \( f \)

proof

- have \( 0 \leq x \)
  - by (rule LIMSEQ-le[OF tendsto-const f]) (auto intro!: exI[of - 1])

from \( x < 1 \) obtain \( z \) where \( z: x < z z < 1 \)
  - by (metis dense)

from \( f (x < z) \) have eventually \( (\lambda n. \text{root } n (\text{norm } f n)) < z \) sequentially
  - by (rule order-tendstoD)

then have eventually \( (\lambda n. \text{norm } f n) \leq z^n \) sequentially
  - using eventually-ge-at-top

proof eventually-elim

fix \( n \) assume less: \( \text{root } n (\text{norm } f n) < z \) and \( n: 1 \leq n \)

from power-strict-mono[OF less, of \( n \)] \( n \) show \( \text{norm } f n \leq z^n \)
  - by simp

qed

then show summable \( f \)
  - unfolding eventually-sequentially
    using \( z (0 \leq x) \) by (auto intro!: summable-comparison-test[OF - summable-geometric])

qed

105.1 Properties of Power Series

lemma lemma-realpow-diff:
  fixes \( y :: 'a::monoid-mult \)
  shows \( p \leq n \Rightarrow y ^ (\text{Suc } n - p) = (y ^ (n - p)) * y \)

proof

- assume \( p \leq n \)
  hence \( \text{Suc } n - p = \text{Suc } (n - p) \) by (rule Suc-diff-le)

thus \( \text{thesis} \) by (simp add: power-commutes)

qed

lemma lemma-realpow-diff-sumr2:
  fixes \( y :: 'a::\{comm-ring,monoid-mult\} \)
  shows \( x ^ (\text{Suc } n) - y ^ (\text{Suc } n) = (x - y) * (\sum p<\text{Suc } n. (x ^ p) * y ^ (n - p)) \)

proof (induct \( n \))

- case \( \text{Suc } n \)
  have \( x ^ (\text{Suc } n) - y ^ (\text{Suc } n) = x * (x * x ^ n) - y * (y * y ^ n) \)
    - by simp
  also have \( ... = y * (x ^ (\text{Suc } n) - y ^ (\text{Suc } n)) + (x - y) * (x * x ^ n) \)
    - by (simp add: algebra-simps)
  also have \( ... = y * ((x - y) * (\sum p<\text{Suc } n. (x ^ p) * y ^ (n - p))) + (x - y) * (x * x ^ n) \)
    - by simp

qed
by (simp only: Suc)
also have ... = (x - y) * (Suc n). ((x ^ p) * (y ^ (n - Suc n))) + (x - y)
* (x * x ^ n)
by (simp only: mult.left-commute)
also have ... = (x - y) * ((Suc p) * (Suc (n - p)))
by (simp add: field-simps Suc-diff-le setsum-left-distrib setsum-right-distrib)
finally show ?case.
qed simp

corollary power-diff-sumr2: — COMPLEX-POLYFUN in HOL Light
fixes x :: 'a::{comm-ring,monoid-mult}
shows x ^ n - y ^ n = (x - y) * (n * (y ^ (n - Suc n)))
using lemma-realpow-diff-sumr2[of x n - 1]
by (cases n = 0) (simp-all add: field-simps)

lemma lemma-realpow-rev-sumr:
\( \sum_{p < Suc n} (x ^ p) * (y ^ (n - Suc p)) = \)
\( \sum_{p < Suc n} (x ^ (n - Suc p)) * (y ^ p) \)
by (subst nat-diff-setsum-reindex[symmetric]) simp

lemma power-diff-1-eq:
fixes x :: 'a::{comm-ring,monoid-mult}
shows \( n \neq 0 \Rightarrow x ^ n - 1 = (x - 1) * (\sum_{i < n} (x ^ i)) \)
using lemma-realpow-diff-sumr2[of 1 - x]
by (cases n) auto

lemma one-diff-power-eq:
fixes x :: 'a::{comm-ring,monoid-mult}
shows \( n \neq 0 \Rightarrow 1 - x ^ n = (1 - x) * (\sum_{i < n} (x ^ i)) \)
using lemma-realpow-diff-sumr2[of 1 - x]
by (cases n) auto

lemma one-diff-power-eq:
fixes x :: 'a::{comm-ring,monoid-mult}
shows \( n \neq 0 \Rightarrow 1 - x ^ n = (1 - x) * (\sum_{i < n} (x ^ i)) \)
by (metis one-diff-power-eq[of n x] nat-diff-setsum-reindex)

Power series has a 'circle' of convergence, i.e. if it sums for \( x \), then it sums absolutely for \( z \) with \( |z| < |x| \).

lemma power-insidea:
fixes x z :: 'a::real-normed-div-algebra
assumes 1: summable (\( \lambda n. f n * x ^ n \))
and 2: norm z < norm x
shows summable (\( \lambda n. norm (f n * z ^ n) \))
proof –
from 2 have x-neq-0: \( x \neq 0 \) by clarsimp
from 1 have (\( \lambda n. f n * x ^ n \)) ----> 0
by (rule summable-LIMSEQ-zero)

hence convergent (\( \lambda n. f n * x ^ n \))
by (rule convergentI)

**hence** Cauchy \((\lambda n. \ f \ n \ * \ x \ ^\ n)\)

by (rule convergent-Cauchy)

**hence** Bseq \((\lambda n. \ f \ n \ * \ x \ ^\ n)\)

by (rule Cauchy-Bseq)

then obtain \(K\) where \(3: \ 0 < K\) and \(4: \ \forall n. \ \text{norm} (f \ n \ * \ x \ ^\ n) \leq K\)

by (simp add: Bseq-def, safe)

have \(\exists N. \ \forall n \geq N. \ \text{norm} (f \ n \ * \ z \ ^\ n) \leq K \ * \ \text{norm} (z \ ^\ n)\)

proof (intro exI allI impI)

fix \(n::\text{nat}\)

assume \(0 \leq n\)

have \(\text{norm} (\text{norm} (f \ n \ * \ z \ ^\ n)) \ * \ \text{norm} (x \ ^\ n) = \text{norm} (f \ n \ * \ x \ ^\ n) \ * \ \text{norm} (z \ ^\ n)\)

by (simp add: norm-mult abs-mult)

also have \(\ldots \leq K \ * \ \text{norm} (z \ ^\ n)\)

by (simp only: mult-right-mono 4 norm-ge-zero)

also have \(\ldots = K \ * \ \text{norm} (z \ ^\ n) \ * \ \text{inverse} (\text{norm} (x \ ^\ n))\)

by (simp add: x-neq-0)

finally show \(\text{norm} (\text{norm} (f \ n \ * \ z \ ^\ n)) \leq K \ * \ \text{norm} (z \ ^\ n)\)

by (simp add: mult-le-cancel-right x-neq-0)

qed

moreover have summable \((\lambda n. \ K \ * \ \text{norm} (z \ ^\ n) \ * \ \text{inverse} (\text{norm} (x \ ^\ n)))\)

proof (from 2 have \(\text{norm} (\text{norm} (z \ * \ \text{inverse} x)) < 1\)

using x-neq-0)

by (simp add: norm-mult nonzero-norm-inverse divide-inverse [where 'a=real, symmetric])

**hence** summable \((\lambda n. \ \text{norm} (z \ * \ \text{inverse} x) \ ^\ n)\)

by (rule summable-geometric)

**hence** summable \((\lambda n. \ K \ * \ \text{norm} (z \ * \ \text{inverse} x) \ ^\ n)\)

by (rule summable-mult)

thus summable \((\lambda n. \ K \ * \ \text{norm} (z \ ^\ n) \ * \ \text{inverse} (\text{norm} (x \ ^\ n)))\)

using x-neq-0

by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib

power-inverse norm-power mult.assoc)

qed

ultimately show summable \((\lambda n. \ \text{norm} (f \ n \ * \ z \ ^\ n))\)

by (rule summable-comparison-test)

qed

**lemma** powser-inside:

**fixes** \(f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-div-algebra},\text{banach}\}\)

**shows**

summable \((\lambda n. \ f \ n \ * (x \ ^\ n))\) \(\Rightarrow \ \text{norm} z < \text{norm} x \Rightarrow \)

summable \((\lambda n. \ f \ n \ * (z \ ^\ n))\)
lemmas sum-split-even-odd:
  fixes f :: nat ⇒ real
  shows 
    (∑ i<2 * n. if even i then f i else g i) =
    (∑ i<n. f (2 * i)) + (∑ i<n. g (2 * i + 1))
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have 
    (∑ i<2 * Suc n. if even i then f i else g i) =
    (∑ i<n. f (2 * i)) + (∑ i<n. g (2 * i + 1)) + (f (2 * n) + g (2 * n + 1))
  using Suc.hyps unfolding One-nat-def by auto
  also have ... = (∑ i<Suc n. f (2 * i)) + (∑ i<Suc n. g (2 * i + 1))
  by auto
  finally show ?case .
qed

lemma sums-if':
  fixes g :: nat ⇒ real
  assumes g sums x
  shows (λ n. if even n then 0 else g ((n - 1) div 2)) sums x
unfolding sums-def
proof (rule LIMSEQ-I)
  fix r :: real
  assume 0 < r
  from ⟨g sums x⟩[unfolded sums-def, THEN LIMSEQ-D, OF this]
  obtain no where no-eq: ∀ n. n ≥ no =⇒ (norm (setsum g {..<n} - x) < r)
  by blast
  let ?SUM = λ m. ∑ i<m. if even i then 0 else g ((i - 1) div 2)
  { fix m
    assume m ≥ 2 * no
    hence m div 2 ≥ no by auto
    have sum-eq: ?SUM (2 * (m div 2)) = setsum g {..< m div 2}
      using sum-split-even-odd by auto
    hence (norm (?SUM (2 * (m div 2)) - x) < r)
      using no-eq unfolding sum-eq using ⟨m div 2 ≥ no⟩ by auto
    moreover
    have ?SUM (2 * (m div 2)) = ?SUM m
    proof (cases even m)
      case True
      show ?thesis
      unfolding even-nat-div-two-times-two[OF True, unfolded numeral-2-eq-2[symmetric]]
      ..
    next
THEORY "Transcendental" 1652

case False
  hence even (Suc m) by auto
from even-nat-div-two-times-two[OF this, unfolded numeral-2-eq-2[symmetric]]
  odd-nat-plus-one-div-two[OF False, unfolded numeral-2-eq-2[symmetric]]
have eq: Suc (2 * (m div 2)) = m by auto
  hence even (2 * (m div 2)) using odd m by auto
have ?SUM m = ?SUM (Suc (2 * (m div 2))) unfolding eq ..
also have .. = ?SUM (2 * (m div 2)) using (even (2 * (m div 2))) by auto
finally show ?thesis by auto
qed
ultimately have (norm (?SUM m − x) < r) by auto
thus ∃ no. ∀ m ≥ no. norm (?SUM m − x) < r by blast
qed

lemma sums-if:
  fixes g :: nat ⇒ real
  assumes g sums x and f sums y
  shows (λ n. if even n then f (n div 2) else g ((n − 1) div 2)) sums (x + y)
proof −
  let ?s = λ n. if even n then 0 else f ((n − 1) div 2)
  { fix B T E
    have (if B then (0 :: real) else E) + (if B then T else 0) = (if B then T else E)
      by (cases B) auto
  } note if-sum = this
  have g-sums: (λ n. if even n then f (n div 2) else g ((n − 1) div 2)) sums x
    using sums-if'[OF ?g-sums x] .
  { have if-eq: ∃ B T E. (if ¬ B then T else E) = (if B then E else T) by auto
    have ?s sums y using sums-if'[OF f sums y] .
    from this[unfolded sums-def, THEN LIMSEQ-Suc]
    have (λ n. if even n then f (n div 2) else 0) sums y
      by (simp add: lessThan-Suc-eq-insert-0 image-iff setsum.reindex if-eq sums-def cong del: if-cng)
  } from sums-add[OF g-sums this] show ?thesis unfolding if-sum .
qed

105.2 Alternating series test / Leibniz formula

lemma sums-alternating-upper-lower:
  fixes a :: nat ⇒ real
  assumes mono: ∃ n. a (Suc n) ≤ a n and a-pos: ∃ n. 0 ≤ a n and a −−−−−−→ 0
  shows ∃ l. (∀ n. (∑ i<2*n. −1^i*a i) ≤ l) ∧ (∀ n. ∑ i<2*n. −1^i*a i) −−−−−−→ l) ∧
proof

(i: \exists l. ((\forall n. l \leq i < 2n \Rightarrow \sum i < 2n + 1. (-1)^{i*a} i)) \land (\forall n. \sum i < 2n + 1. -1^{i*a} i) \Rightarrow l)

proof (rule nested-sequence-an) have fg-diff: \exists l. ?f n - ?g n = - a (2 * n)

let ?f n = \exists l. - a (2 * n)

assumes a-zero: a -\in\mathbb{R}

fixes a

theorem a-pos: \forall n. 0 \leq a n

and a-monotone: \forall n. a (Suc n) \leq a n

shows summable: summable (\lambda n. (-1)^{n*a} n)

and \exists l. \forall n. \lambda n. (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

and \exists l. \forall n. \lambda n. (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

and \exists l. \forall n. \lambda n. (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

proof

let ?S = \lambda n. (-1)^{n*a} n

let ?P = \lambda n. \sum i < n. \sum i

let \forall n. ?P (2 * n)

lemma summable-Leibniz':

fixes a :: nat \Rightarrow \mathbb{R}

assumes a-zero: a -\in\mathbb{R}

and a-pos: \forall n. 0 \leq a n

and a-monotone: \forall n. a (Suc n) \leq a n

shows summable: summable (\lambda n. (-1)^{n*a} n)

and \exists l. \forall n. \lambda n (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

and \exists l. \forall n. \lambda n. (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

and \exists l. \forall n. \lambda n. (-1)^{i*a} i \leq \sum i \leq \sum i (\lambda i. (-1)^{i*a} i)

proof

let \exists l. \forall n. \lambda n. (-1)^{n*a} n

let \forall n. \sum i < n. \sum i

let \forall n. ?P (2 * n)


THEORY "Transcendental"

let \( ?g = \lambda n. ?P (2 * n + 1) \)

obtain \( l :: \text{real} \)

where below-l: \( \forall n. ?f n \leq l \)
and \( \forall n \). \( n \leq ?g n \)

and above-l: \( \forall n. l \leq ?g n \)

using sums-alternating-upper-lower[OF a-monotone a-pos a-zero] by blast

let \( ?Sa = \lambda m. \sum _{n < m} ?S n \)

have \( ?Sa \leq l \)

proof (rule LIMSEQ_I)

fix \( r :: \text{real} \)
assume \( 0 < r \)

with \( \forall f \). \( \exists l. \exists (\forall n \geq f - l \text{ norm } (f n - l) < r) \)

obtain \( f-no \) where \( f; \forall n. n \geq f-no \implies \text{norm } (f n - l) < r \)

by auto

from \( \forall r \) \( \forall \exists l. \exists (\forall n \geq f - l \text{ norm } (f n - l) < r) \)

obtain \( g-no \) where \( g; \forall n. n \geq g-no \implies \text{norm } (g n - l) < r \)

by auto

{

fix \( n :: \text{nat} \)
assume \( n \geq (\max (2 * f-no) (2 * g-no)) \)

hence \( n \geq 2 * f-no \) and \( n \geq 2 * g-no \)

by auto

have \( \text{norm } (\forall Sa n - l) < r \)

by auto

proof (cases even n)

case True

have \( \forall Sa n - l \)

by auto

from even-nat-div-two-times-two[OF this]

hence \( n 

\text{div } 2 = n \)

unfolding numeral-2-eq-2[ symmetric] by auto

with \( n \geq 2 * f-no \)

have \( n \text{ div } 2 \geq f-no \)

by auto

from \( f[OF this] \)

show \( ?thesis \)

unfolding n-eq atLeastLessThanSuc-atLeastAtMost .

next

case False

hence \( \text{even } (n - 1) \)

by simp

from even-nat-div-two-times-two[OF this]

have \( n-eq: 2 * ((n - 1) \text{ div } 2) = n - 1 \)

unfolding numeral-2-eq-2[ symmetric] by auto

hence \( n-eq: n - 1 + 1 = n \)

using odd-pos[OF False] by auto

from \( n-eq \)

have \( (n - 1) \text{ div } 2 \geq g-no \)

by auto

from \( g[OF this] \)

show \( ?thesis \)

unfolding n-eq range-eq .

qed

}

thus \( \exists \text{no. } \forall n \geq \text{no. } \text{norm } (\forall Sa n - l) < r \)

by blast
qed

hence sums-l: (\lambda i. \((-1)^i \cdot a \cdot i) \cdot \text{sums } l

unfolding \text{sums-def}.

thus \text{summable } S \text{ using summable-def by auto}

have \( l = \text{suminf } S \text{ using sums-unique[\text{OF sums-l}]}. \)

fix \( n \)

show \( \text{suminf } S \leq g \)

unfolding sums-unique[\text{OF sums-l}, \text{symmetric}] using above-l by auto

show \( \exists n \leq \text{suminf } S \)

unfolding sums-unique[\text{OF sums-l}, \text{symmetric}] using below-l by auto

show \( \exists g \longrightarrow \text{suminf } S \)

using \( \exists g \longrightarrow l \text{ = suminf } S \) by auto

show \( \exists f \longrightarrow \text{suminf } S \)

using \( \exists f \longrightarrow l \text{ = suminf } S \) by auto

qed

theorem summable-Leibniz:

fixes \( a :: \text{nat} \Rightarrow \text{real} \)

assumes \( a\text{-zero: } a \longrightarrow 0 \text{ and monoseq } a \)

shows \( \text{summable } (\lambda n. (-1)^n \cdot a \cdot n) \text{ (is ?summable)} \)

and \( 0 < a \cdot 0 \longrightarrow \)

\( (\forall n. \sum i. -1^i \cdot a \cdot i) \in \{ \sum i<2^* n. -1^i \cdot a \cdot i. \sum i<2^* n+1. -1^i \cdot a \cdot i \} \)

(is ?pos)

and \( a \cdot 0 < 0 \longrightarrow \)

\( (\forall n. \sum i. -1^i \cdot a \cdot i) \in \{ \sum i<2^* n+1. -1^i \cdot a \cdot i. \sum i<2^* n+1. -1^i \cdot a \cdot i \} \)

(is ?neg)

and \( (\lambda n. \sum i<2^* n. -1^i \cdot a \cdot i) \longrightarrow \) \( (\sum i. -1^i \cdot a \cdot i) \text{ (is ?f)} \)

and \( (\lambda n. \sum i<2^* n+1. -1^i \cdot a \cdot i) \longrightarrow \) \( (\sum i. -1^i \cdot a \cdot i) \text{ (is ?g)} \)

proof –

have \( \exists \text{summable } \land \exists \text{pos } \land \exists \text{neg } \land \exists f \land \exists g \)

proof (cases \( \forall n. 0 \leq a \cdot n \) \land \( \forall m. \forall n \geq m. a \cdot n \leq a \cdot m \))

case True

hence ord: \( \land n. m. m \leq n \Longrightarrow a \cdot n \leq a \cdot m \) and ge0: \( \land n. 0 \leq a \cdot n \)

by auto

\{ fix \( n \)

have \( a \cdot (\text{Suc } n) \leq a \cdot n \)

using ord[where \( n=\text{Suc } n \text{ and } m=n] \text{ by auto }

\} note mono = this

note leibniz = summable-Leibniz'[\text{OF } (a \longrightarrow 0), \text{ge0}]

from leibniz[OF mono]

show \( \exists \text{thesis } \text{using } (0 \leq a \cdot 0) \text{ by auto}

next

let \( a = \lambda n. -a \cdot n \)

case False

with monoseq-le[\text{OF } :\text{monoseq } a] (a \longrightarrow 0] \]

have \( \forall n. a \cdot n \leq 0 \) \land \( \forall m. \forall n \geq m. a \cdot m \leq a \cdot n \) by auto
hence ord: \( \forall n \in \mathbb{N}. \forall m \leq n \rightarrow \exists a \in \mathbb{R} \text{ and } \forall m, n \in \mathbb{N}. 0 \leq a \in \mathbb{R} \)

by auto

\{ fix n have \(?a\ (\text{Suc } n) \leq \?a\ n\ \text{using ord[where } n=\text{Suc } n\ \text{and } m=n]\)

by auto \}

note monotone = this

note leibniz =
  summable-Leibniz[OF - ge0, of \( \lambda x. x \),
  OF tendsto minus[OF (a ----> 0), unfolded minus-zero] monotone]

have summable (\( \lambda n. (\text{-}1)^n * \?a\ n \))
  using leibniz(1) by auto

then obtain l where (\( \lambda n. (\text{-}1)^n * \?a\ n \)) sums l

unfolding summable-def by auto

from this[THEN sums-minus] have (\( \lambda n. (\text{-}1)^n * \?a\ n \)) sums \(-l\)

by auto

hence \(?\text{summable}\) unfolding summable-def by auto

moreover have \(\forall a b :: \mathbb{R}. |a - b| = |a - b|\)

unfolding minus diff minus by auto

from suminf-minus[OF leibniz(1), unfolded mult minus minus]

have move minus: \((\sum n. (\text{-}1)^n * \?a\ n)) = - (\sum n. (\text{-}1)^n * \?a\ n)\)

by auto

have \(?\text{pos}\) using \(\{0 \leq \?a\ 0\}\) by auto

moreover have \(?\text{neg}\)
  using leibniz(2,4)

unfolding mult minus right setsum negf move minus neg le iff le

by auto

moreover have \(?f\ \text{and } \?g\)
  using leibniz(3,5)[unfolded mult minus right setsum negf move minus, THEN
  tendsto minus cancel]

by auto

ultimately show \(?\text{thesis}\) by auto

qed

then show \(?\text{summable and } \?\text{pos and } \?\text{neg and } \?f\ \text{and } \?g\)

by safe

qed

105.3 Term-by-Term Differentiability of Power Series

definition diffs :: \( \text{nat} \Rightarrow \text{'}a::\text{ring-1} \Rightarrow \text{nat} \Rightarrow \text{'}a \)

where diffs \( c = (\lambda n. \text{of-nat} (\text{Suc } n) * c (\text{Suc } n)) \)

Lemma about distributing negation over it

lemma diffs minus: \( \text{diffs} (\lambda n. - c n) = (\lambda n. - \text{diffs} c n)\)

by (simp add: diffs-def)
lemma sums-Suc-imp:
\[(f :: \text{nat} \Rightarrow \text{a} :: \text{real-normed-vector}) \; 0 = 0 \implies (\lambda n. f (\text{Suc} \; n)) \; \text{sums} \; s \implies (\lambda n. f \; n) \; \text{sums} \; s\]
using sums-Suc-iff[of \; f] by simp

lemma diffs-equiv:
fixes \; x :: \text{a} :: \{\text{real-normed-vector}, \text{ring-1}\}
shows (\lambda n. \text{diffs} \; c \; n \; \ast \; x \; ^{\text{n}}) \; \text{sums} \; s =\; (\lambda n. \text{of-nat} \; n \; \ast \; c \; n \; \ast \; x \; ^{\text{n}}) \; \text{sums} \; s
by (simp add: summable-sums sums-Suc-imp)

lemma lemma-termdiff1:
fixes \; z :: \text{a} :: \{\text{monoid-mult, comm-ring}\}
shows (\sum p \prec n. (((z + h) \; ^{\cdot} (m - p)) \; \ast \; (z \; ^{\cdot} m)) - (z \; ^{\cdot} m)) = (\sum p \prec m. (z \; ^{\cdot} p) \; \ast \; (((z + h) \; ^{\cdot} (m - p)) - (z \; ^{\cdot} (m - p))))
by (auto simp add: algebra-simps power-add [symmetric])

lemma lemma-termdiff2:
fixes h :: \text{a} :: \{\text{field}\}
assumes h: \; h \neq 0
shows (\frac{(z + h) \; ^{\cdot} n - z \; ^{\cdot} n }{ h - \text{of-nat} \; n \; \ast \; z \; ^{\cdot} (n - \text{Suc} \; 0)} = h \; \ast \; (\sum p < n - \text{Suc} \; 0. \; \sum q < n - \text{Suc} \; 0 - p. \; (z + h) \; ^{\cdot} q \; \ast \; z \; ^{\cdot} (n - 2 - q))\; (\text{is} \; \text{lhs} = \; ?\text{rhs})
apply (subgoal-tac h \; \ast \; ?\text{lhs} \; = \; h \; \ast \; ?\text{rhs}, \; simp\; add: \; h)
apply (simp\; add: right-diff-distrib diff-divide-distrib h)
apply (simp\; add: mult.assoc [symmetric])
apply (cases n, simp)
apply (simp\; add: lemma-realpow-diff-sumr2 h
right-diff-distrib [symmetric] mult.assoc
del: power-Suc setsun-lessThan-Suc of-nat-Suc)
apply (subst lemma-realpow-rev-sumr)
apply (subst sumr-diff-mult-const2)
apply simp
apply (simp only: lemma-termdiff1 setsum-right-distrib)
apply (rule setsum.cong [OF refl])
apply (simp\; add: less-iff-Suc-add)
apply (clarify)
apply (simp\; add: setsum-right-distrib lemma-realpow-diff-sumr2 ac-simps
del: setsum-lessThan-Suc power-Suc)
apply (subst mult.assoc [symmetric], subst\; power-add [symmetric])
apply (simp\; add: ac-simps)
done
lemma real-setsum-nat-ivl-bounded2:
fixes K :: 'a::linordered-semidom
assumes f: \( \forall p::\text{nat}. \ p < n \Rightarrow f p \leq K \)
and K: \( 0 \leq K \)
shows setsum f {..<n-k} \leq of-nat n * K
apply (rule order-trans [OF setsum_mono])
apply (rule f, simp)
apply (simp add: mult-right-mono K)
done

lemma lemma-termdiff3:
fixes h z :: 'a::real-normed-field
assumes 1: h \neq 0
and 2: \text{norm } z \leq K
and 3: \text{norm } (z + h) \leq K
shows \text{norm } (((z + h)^n - z^n) / h - of-nat n * z ^ (n - Suc 0)) \\
\leq of-nat n * of-nat (n - Suc 0) * K ^ (n - 2) * \text{norm } h
proof
have \text{norm } (((z + h)^n - z^n) / h - of-nat n * z ^ (n - Suc 0)) = \\
\text{norm } (\sum p<n - Suc 0. \sum q<n - Suc 0 - p. \\
(z + h) ^ q * z ^ (n - 2 - q)) * \text{norm } h
by (metis (lifting, no-types) lemma-termdiff2 [OF 1] mult.commute norm-mult)
also have \ldots \leq of-nat n * (of-nat (n - Suc 0) * K ^ (n - 2)) * \text{norm } h
proof (rule mult-right_mono [OF - norm-ge-zero])
from norm-ge-zero 2 have K: \( 0 \leq K \)
by (rule order-trans)
have le-Kn: \( \forall i j n. \ i + j = n \Rightarrow \text{norm } ((z + h) ^ i * z ^ j) \leq K ^ n \)
apply (erule subst)
apply (simp only: norm-mult norm-power power-add)
apply (intro mult-mono power-mono 2 3 norm-ge-zero zero-le-power K)
done
show \text{norm } (\sum p<n - Suc 0. \sum q<n - Suc 0 - p. \\
(z + h) ^ q * z ^ (n - 2 - q)) \leq of-nat n * (of-nat (n - Suc 0) * K ^ (n - 2))
apply (intro \\
order-trans [OF norm-setsum]
real-setsum-nat-ivl-bounded2
mult-nonneg-nonneg
of-nat-0-le-iff
zero-le-power K)
apply (rule le-Kn, simp)
done
qed
also have \ldots = of-nat n * of-nat (n - Suc 0) * K ^ (n - 2) * \text{norm } h
by (simp only: mult.assoc)
finally show \text{thesis}.
qed

lemma lemma-termdiff4:
fixes \( f :: 'a::real-normed-vector \Rightarrow 'b::real-normed-vector \)

assumes \( k : 0 < (k::real) \)

and \( le : \forall h. [h \neq \theta; \text{norm } h < k] \implies \text{norm } (f h) \leq K * \text{norm } h \)

shows \( f \limpoint 0 \limpoint 0 \)

proof (rule tendsto-norm-zero-cancel)
show \( \lambda h \text{. norm } (f h) \limpoint 0 \)

proof (rule real-tendsto-sandwich)
show eventually \( (\lambda h \text{. norm } (f h)) \limpoint 0 \)

proof (rule simp)

show \( (\lambda h \text{. norm } (f h)) \limpoint 0 \)

qed

lemma lemma-termdiff5:

fixes \( g :: 'a::real-normed-vector \Rightarrow 'b::banach \)

assumes \( k : 0 < (k::real) \)

assumes \( f : \text{summable } f \)

assumes \( le : \forall h n. [h \neq \theta; \text{norm } h < k] \implies \text{norm } (g h n) \leq f n * \text{norm } h \)

shows \( (\lambda h \text{. \text{suminf } (g h)}) \limpoint 0 \)

proof (rule lemma-termdiff4 [OF k])
fix \( h :: 'a \)

assume \( h \neq \theta \text{ and } \text{norm } h < k \)
hence \( A : \forall n \text{. norm } (g h n) \leq f n * \text{norm } h \)
by (simp add: le)

hence \( \exists N. \forall n \geq N. \text{norm } (g h n) \leq f n * \text{norm } h \)
by simp

moreover from \( f \) have \( B : \text{summable } (\lambda n \text{. } f n * \text{norm } h) \)
by (rule summable-mult2)

ultimately have \( C : \text{summable } (\lambda n \text{. } \text{norm } (g h n)) \)
by (rule summable-comparison-test)
hence \( \text{norm } (\text{suminf } (g h)) \leq (\sum n \text{. norm } (g h n)) \)
by (rule summable-norm)
also from \( A C B \) have \( (\sum n \text{. norm } (g h n)) \leq (\sum n \text{. } f n * \text{norm } h) \)
by (rule suminf-le)
also from \( f \) have \( (\sum n \text{. } f n * \text{norm } h) = \text{suminf } f * \text{norm } h \)
by (rule suminf-mult2 [symmetric])
finally show \( \text{norm } (\text{suminf } (g h)) \leq \text{suminf } f * \text{norm } h \).

qed

FIXME: Long proofs

lemma termdiffs-aux:

fixes \( x :: \{\text{real-normed-field}, \text{banach}\} \)
THEORY "Transcendental"

assumes 1: summable (λn. diffs (diffs c) n * K " n)
and 2: norm x < norm K
shows (λh. ∑ n. c n * (((x + h) " n - x " n) / h
and of-nat n * x " (n - Suc 0))) -- 0 --> 0
proof –
from dense [OF 2]
obtain r where r1: norm x < r and r2: r < norm K by fast
from norm-ge-zero r1 have r: 0 < r
by (rule order-le-less-trans)
hence r-neq-0: r ≠ 0 by simp
show ?thesis
proof (rule lemma-termdiff5)
show 0 < r − norm x using r1 by simp
from r r2 have norm (of-real r::'a) < norm K
by simp
with I have summable (λn. norm (diffs (diffs c) n * (of-real r " n)))
by (rule powser-insidea)
hence summable (λn. diffs (diffs (λn. norm (c n))) n * r " n)
using r
by (simp add: diffs-def norm-mult norm-power del: of-nat-Suc)
hence summable (λn. of-nat n * diffs (λn. norm (c n)) n * r " (n − Suc 0))
by (rule diffs-equiv [THEN sums-summable])
also have (λn. of-nat n * diffs (λn. norm (c n)) n * r " (n − Suc 0)) =
(λn. diffs (λn. of-nat (m − Suc 0) * norm (c m) * inverse r) n * (r " n))
apply (rule ext)
apply (simp add: diffs-def)
apply (case_tac n, simp-all add: r-neq-0)
done
finally have summable
(λn. of-nat n * (of-nat (n − Suc 0) * norm (c n) * inverse r) * r " (n − Suc 0))
by (rule diffs-equiv [THEN sums-summable])
also have
(λn. of-nat n * (of-nat (n − Suc 0) * norm (c n) * inverse r) * r " (n − Suc 0)) =
(λn. norm (c n) * of-nat n * of-nat (n − Suc 0) * r " (n − 2))
apply (rule ext)
apply (case_tac n, simp)
apply (rename_tac nat)
apply (case_tac nat, simp)
apply (simp add: r-neq-0)
done
finally
show summable (λn. norm (c n) * of-nat n * of-nat (n − Suc 0) * r " (n − 2))
next
fix h::'a and n::nat
assume h: h ≠ 0
assume norm h < r − norm x
hence norm x + norm h < r by simp
with norm-triangle-ineq have xh: norm (x + h) < r
by (rule order-le-less-trans)
show norm (c n * ((x + h) * n - x * n) / h - of-nat n * x * (n - Suc 0)))
  ≤ norm (c n * of-nat n * of-nat (n - Suc 0) * r * (n - 2) * norm h
apply (simp only: norm-mul mult_assoc)
apply (rule mult-left-mono[OF norm-ge-zero])
apply (simp add: mult_assoc[symmetric])
apply (metis h lemma-termdiff3 less-eq-real-def r1 xh)
done
qed

lemma termdiffs:
fixes K x :: 'a::{real-normed_field,banach}
assumes 1: summable (λn. c n * K ^ n)
  and 2: summable (λn. (diffs c) n * K ^ n)
  and 3: summable (λn. (diffs (diffs c)) n * K ^ n)
  and 4: norm x < norm K
shows DERIV (λx. ∑n. c n * x ^ n) x :> (∑n. (diffs c) n * x ^ n)
unfolding DERIV-def
proof (rule LIM-zero-cancel)
  show (λh. (suminf (λn. c n * (x + h) ^ n) - suminf (λn. c n * x ^ n)) / h
         - suminf (λn. diffs c n * x ^ n)) --> 0 --> 0
    by (simp add: less-diff-eq)
next
  fix h :: 'a
  assume norm (h - 0) < norm K - norm x
  hence norm x + norm h < norm K by simp
  hence 5: norm (x + h) < norm K
       by (rule norm-triangle-ineq [THEN order-le-less-trans])
  have summable (λn. c n * x ^ n)
       and summable (λn. c n * (x + h) ^ n)
       and summable (λn. diffs c n * x ^ n)
       using 1 2 4 5 by (auto elim: powser-inside)
  then have ((∑n. c n * (x + h) ^ n) - (∑n. c n * x ^ n)) / h - (∑n. diffs c n * x ^ n)
    = (∑n. (c n * (x + h) ^ n - c n * x ^ n) / h - of-nat n * c n * x ^ (n - Suc 0))
    by (intro sums-unique sums-diff sums-divide diffs-equiv summable-sums)
  then show ((∑n. c n * (x + h) ^ n) - (∑n. c n * x ^ n)) / h - (∑n. diffs c n * x ^ n)
    = (∑n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0)))
    by (simp add: algebra_simps)
next
  show (λh. ∑n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0))) --> 0 --> 0
    by (rule termdiffs-aux[OF 3 4])
105.4 Derivability of power series

lemma DERIV-series':
  fixes f :: real ⇒ nat ⇒ real
  assumes DERIV-f: \( \lambda\). DERIV (\( \lambda\). f (x n)) x0 ⇒ (f' x0 n)
  and allf-summable: \( \lambda\). x. x ∈ \{ a <..< b \} ⇒ summable (f x) and x0-in-l: x0 ∈ \{ a <..< b \}
  and summable (f' x0)
  and summable L
  and L-def: \( \lambda\). x. y. \[ x ∈ \{ a <..< b \} \land \[ y ∈ \{ a <..< b \} \] ⇒ |f x n - f y n|
  ≤ L n * |x - y|
  shows DERIV (\( \lambda\). suminf (f x)) x0 ⇒ (suminf (f' x0))
  unfolding DERIV-def
proof (rule LIM-I)
  fix r :: real
  assume 0 < r hence 0 < r/3 by auto

  obtain N-L where N-L: \( \lambda\). n. N-L ≤ n ⇒ \( \{ \sum i. L (i + n) \} < r/3 \)
    using suminf-exist-split[OF \( \{ 0 < r/3 \} \land \{ \text{summable L} \} \) by auto

  obtain N-f' where N-f': \( \lambda\). n. N-f' ≤ n ⇒ \( \{ \sum i. f' x0 (i + n) \} < r/3 \)
    using suminf-exist-split[OF \( \{ 0 < r/3 \} \land \{ \text{summable (f' x0)} \} \) by auto

  let ?N = Suc (max N-L N-f')
  have \( \{ \sum i. f' x0 (i + ?N) \} < r/3 \) (is \( \text{?f'-part < r/3} \) and
    L-estimate: \( \{ \sum i. L (i + ?N) \} < r/3 \) using N-L[of \( ?N \) and N-f'[of \( ?N \)] by auto

  let ?diff = \( \lambda\). x. (f (x0 + x) - f x0 i) / x

  let ?r = r / (3 * real ?N)
  from \( \{ 0 < r \} \) have \( \{ 0 < ?r \} \) by simp

  let ?s = \( \lambda\). n. SOME s. 0 < s ∧ (\( \forall\). x. x \neq 0 ∧ \[ x < s \] ⇒ \[ ?diff n x - f' x0 \]
    \( n < ?r \)
  def S' ≡ Min \( \{ ?s \cdot \{ ..< \{ ?N \} \} \)

  have \( \{ 0 < S' \) unfolding S'-def
proof (rule iffD2[OF Min-gr-iff])
  show \( \forall\). x ∈ \( \{ ?s \cdot \{ ..< \{ ?N \} \} \), \( 0 < x \)
proof
  fix x
  assume x ∈ \( ?s \cdot \{ ..< \{ ?N \} \)
  then obtain n where x = ?s n and n ∈ \( \{ ..< \{ ?N \} \)
    using image-iff[OF \( \text{THEN iffD1} \) by blast
  from DERIV-D[OF DERIV-f[where \( n=n \), THEN LIM-D, \( \{ 0 < \{ ?r \} \),
THEORY "Transcendental"

unfolded real-norm-def

obtain \( s \) where \( s \)-bound: \( 0 < s \wedge (\forall x. x \neq 0 \wedge |x| < s \rightarrow |\text{diff} n x - f'| x0 n | < ?r) \)

by auto

have \( 0 < ?s n \) by (rule someI2[where \( a=s \)]) (auto simp add: \( s \)-bound)

thus \( 0 < x \) unfolding \( x = ?s n \).

qed

qed auto

def \( S \equiv \min (\min (x0 - a) (b - x0)) S' \)

hence \( 0 < S \) and \( S-a: S \leq x0 - a \) and \( S-b: S \leq b - x0 \)

and \( S \leq S' \) using \( x0-in-I \) and \( 0 < S' \)

by auto

{ 
fix \( x \)
assume \( x \neq 0 \) and \( |x| < S \)
hence \( x-in-I: x0 + x \in \{ a..<b \} \)
using \( S-a \) \( S-b \) by auto

note \( \text{diff-smbl} = \text{summable-diff}[OF \text{allf-summable}[OF x-in-I]] \text{allf-summable}[OF x0-in-I]] \)

note \( \text{div-smbl} = \text{summable-divide}[OF \text{diff-smbl}] \)

note \( \text{all-smbl} = \text{summable-diff}[OF \text{div-smbl} \langle\text{summable} (f' x0)\rangle] \)

note \( \text{ign} = \text{summable-ignore-initial-segment}[where \( k=?N \)] \)

note \( \text{diff-shift-smbl} = \text{summable-diff}[OF \text{ign}[OF \text{allf-summable}[OF x-in-I]]] \)

note \( \text{div-shift-smbl} = \text{summable-divide}[OF \text{diff-shift-smbl}] \)

note \( \text{all-shift-smbl} = \text{summable-diff}[OF \text{div-smbl} \text{ign}[OF \langle\text{summable} (f' x0)\rangle]] \)

{ fix \( n \)
have \( |\text{diff} (n + ?N) x | \leq L (n + ?N) * |(x0 + x) - x0 | / | x | \)
using \( \text{divide-right-mono}[OF \text{L-def}[OF x-in-I x0-in-I]] \text{abs-ge-zero] \)
unfolding \( \text{abs-divide} \).
hence \( |(\text{diff} (n + ?N) x | | \leq L (n + ?N) \)
using \( (x \neq 0) \) by auto }

note \( 1 = \text{this} \) and \( 2 = \text{summable-rabs-comparison-test}[OF \text{- ign}[OF \langle\text{summable}\ L\rangle]] \)
then have \( |\sum i. \text{diff} (i + ?N) x | \leq (\sum i. L (i + ?N)) \)

by (metis \( \text{lifting} \) \text{abs-idempotent} \text{order-trans}[OF \text{summable-rabs}[OF 2] \)
\text{suminf-le}[OF \text{- 2 ign}[OF \langle\text{summable} L\rangle]])
then have \( |\sum i. \text{diff} (i + ?N) x | \leq r \div 3 \) (is \( \text{?L-part} \leq r/3) \)
using \( \text{L-estimate} \) by auto

have \( \sum n<?N. \text{diff} n x - f' x0 n | \leq (\sum n<?N. |\text{diff} n x - f' x0 n |) .. \)
also have \( \ldots < (\sum n<?N. ?r) \)
proof (rule \text{setsum-strict-mono})
fix \( n \)
assume \( n \in \{..<?N\} \)
have $|x| < S$ using $\langle |x| < S \rangle$.
also have $S \leq S' < S$ using $\langle S \leq S' \rangle$.
also have $S' \leq ?s n$ unfolding $S'$-def
proof (rule Min-le-iff[THEN iffD2])
  have $\exists s n \in \{ ?s (\cdots) \}_{\cdots} \wedge ?s n \leq ?s n$
    using $\langle n \in \{ \cdots \}_{\cdots} \rangle$ by auto
  thus $\exists a \in ( ?s (\cdots))$. $a \leq ?s n$ by blast
qed auto
finally have $|x| < ?s n$

from DERIV-D[OF DERIV-f[where $n=n$], THEN LIM-D, OF $\langle 0 < ?r \rangle$, unfolded real-norm-def diff-0-right, unfolded some-eq-ex[symmetric], THEN conjunct2]
  have $\forall x. x \neq 0 \wedge |x| < ?s n \rightarrow |?diff n x - f' x0 n| < ?r$.
  with $(x \neq 0)$ and $\langle |x| < ?s n \rangle$ show $|?diff n x - f' x0 n| < ?r$
    by blast
qed auto
also have $\ldots = of-nat (\text{card} \{ \cdots \}_{\cdots}) * ?r$
  by (rule setsam-constant)
also have $\ldots = real \ ?N * ?r$
  unfolding real-eq-of-nat by auto
also have $\ldots = r/3$ by auto
finally have $\langle \sum n < ?N. ?diff n x - f' x0 n | < r / 3 \ (\text{is} \ ?diff-part < r / 3) \rangle$.

from suminf-diff[OF allf-summable[OF x-in-I] allf-summable[OF x0-in-I]]
  have $\langle (\sum n. ?diff n x - f' x0 n) = \langle \sum n. ?diff n x - f' x0 n \rangle \rangle$
  unfolding suminf-diff[OF div-smbl summable (f' x0)], symmetric
  using suminf-divide[OF div-smbl, symmetric] by auto
also have $\ldots \leq ?diff-part + \langle (\sum n. ?diff (n + ?N) x) - (\sum n. f' x0 (n + ?N)) \rangle$
  unfolding suminf-split-initial-segment[OF all-smbl, where $k=?N$
    unfolding suminf-diff[OF div-shft-smbl ign[OF summable (f' x0)]],
    apply (subst (5) add-commute)
    by (rule abs-triangle-ineq)
  also have $\ldots \leq ?diff-part + ?L-part + ?f'-part$
    using abs-triangle-ineq4 by auto
  also have $\ldots < r / 3 + r/3 + r/3$
    using $\langle ?diff-part < r / 3 \rangle$ and $\langle ?f'-part < r / 3 \rangle$
    by (rule add-strict-mono [OF add-less-le-mono])
finally have $\langle (\sum n. f' x0 x) - \suminf f x0 \rangle / x - \suminf (f' x0) | < r$
  by auto

} \text{thus} $\exists s > 0, \forall x. x \neq 0 \wedge \text{norm} (x - 0) < s \rightarrow$
  \text{norm} $\langle (\sum n. f (x0 + x) n) - (\sum n. f x0 n) \rangle / x - (\sum n. f' x0 n) \rangle < r$
  using $\langle 0 < S \rangle$ unfolding real-norm-def diff-0-right by blast
qed
lemma DERIV-power-series':
fixes f :: nat ⇒ real
assumes converges: \( \forall x. x \in \{-R <..< R\} \implies \text{summable (} \lambda n. f \cdot n \cdot \text{real (Suc } n) \cdot x^n\) \)
and x0-in-I: \( x0 \in \{-R <..< R\} \) and \( 0 < R \)
shows DERIV \( \lambda x. (\sum_{n=0}^{\infty} f \cdot n \cdot x^n) \) \( x0 \) :: \( (\sum_{n=0}^{\infty} f \cdot n \cdot \text{real (Suc } n) \cdot x^n) \)
(is DERIV \( \lambda x. (\text{suminf (} \lambda f \cdot x)) \) \( x0 \) :: \( (\text{suminf (} \lambda f' \cdot x0))) \)
proof −

{ fix R'
  assume 0 < R' and R' < R and \(-R' < x0 \) and \( x0 < R' \)
  hence x0 \in \{-R' <..< R'\} \) and \( R' \in \{-R <..< R\} \) and \( x0 \in \{-R <..< R\} \)
  by auto
  have DERIV \( \lambda x. (\text{suminf (} \lambda f \cdot x)) \) \( x0 \) :: \( (\text{suminf (} \lambda f' \cdot x0)) \)
proof (rule DERIV-series')
  show \text{summable (} \lambda n. f \cdot n \cdot \text{real (Suc } n) \cdot R'^n\) \)
  proof −
    have \((R' + R) / 2 < R \) and \( 0 < (R' + R) / 2 \)
    using \( 0 < R' \) and \( R' < R \) by auto
    hence in-Rball: \((R' + R) / 2 \in \{-R <..< R\} \)
    using \( R' < R \) by auto
    have norm R' < norm \((R' + R) / 2\)
    using \( 0 < R' \) and \( R' < R \) by auto
    from powser-insidea[OF converges[OF in-Rball this]] show \?thesis
    by auto
  qed
qed

\}
{n x y
assume x \in \{-R' <..< R'\} \) and \( y \in \{-R' <..< R'\} \)
show \( \text{\lfloor f' x n - f' y n \rfloor} \leq \lfloor f \cdot n \cdot \text{real (Suc } n) \cdot R'^n\) \cdot \text{\lfloor x-y\rfloor} \)
proof −

{ have \( \lfloor f \cdot n \cdot x^n \cdot (\text{Suc } n) - f \cdot n \cdot y^n \cdot (\text{Suc } n) \rfloor \)
  = \( \lfloor f \cdot n \cdot (x^n - y^n) \rfloor \cdot \text{\lfloor \sum_{p=0}^{\infty} n \cdot p \cdot y^n \cdot (n - p) \rfloor} \)
  unfolding right-diff-distrib[ symmetric] lemma-realpow-diff-sumr2 abs-mult
  by auto
  also have \( \ldots \leq \lfloor f \cdot n \cdot (x^n - y^n) \rfloor \cdot \lfloor \text{real (Suc } n) \cdot R' \cdot n \rfloor \)
  proof (rule mult-left-mono)
    have \( \lfloor \sum_{p=0}^{\infty} n \cdot p \cdot y^n \cdot (n - p) \rfloor \leq (\sum_{p=0}^{\infty} n \cdot p \cdot y^n \cdot (n - p)) \)
    by (rule setsum-abs)
    also have \( \ldots \leq (\sum_{p=0}^{\infty} n \cdot R' \cdot n) \)
    proof (rule setsum-mono)
      fix p
      assume p \in \{0,..< Suc n\} \)
      hence p \leq n by auto
      \{
        fix n
        fix x :: real
      \}
  qed
proof (rule mult-left-mono)
  fix p
  assume p \in \{0,..< Suc n\} \)
  hence p \leq n by auto
  \{
    fix n
    fix x :: real
  \}
THEORY "Transcendental"

assume \( x \in \{-R'<..<R'\} \)
hence \(|x| \leq R'\) by auto
hence \(|x^n| \leq R'^n\)

unfolding power-abs by (rule power-mono, auto)

from mult-mono[OF this[OF \( x \in \{-R'<..<R'\} \), of p] this[OF \( y \in \{-R'<..<R'\} \), of n-p]] \( 0 < R' \)

have \(|x^p \ast y^{(n-p)}| \leq R'^p \ast R'^{(n-p)}\)

unfolding abs-mult by auto

thus \(|x^p \ast y^{(n-p)}| \leq R'^n\)

unfolding power-add[symmetric] using \( p \leq n \) by auto

qed

also have \( \ldots = \text{real} (\text{Suc } n) \ast R' \ast n \)

unfolding abs-real-of-nat-cancel abs-of-nonneg[of zero-le-power[of \( 0 < R' \)]]

show \(|f n| \leq |x - y|\)

unfolding abs-mult[symmetric] by auto

qed

also have \( \ldots = |f n \ast \text{real} (\text{Suc } n) \ast R' \ast n| \ast |x - y|\)

unfolding abs-mult mult[assoc][symmetric] by algebra

finally show \( \text{thesis} \).

qed

{ fix \( n \)

show \( \text{DERIV } (\lambda x. f n x) x0 :> (\text{if } f' x0 n) \)

by (auto intro!: derivative_eq_intros simp del: power-Suc simp: real-of-nat-def)

}

{ fix \( x \)

assume \( x \in \{-R'..<..<R'\} \)

hence \( R' \in \{-R..<..<R\} \) and \( \text{norm } x < \text{norm } R' \)

using assms \( R' < R \) by auto

have \( \text{summable } (\lambda n. f n \ast x^n) \)

proof (rule summable-comparison-test, intro exI allI impl)

fix \( n \)

have le: \(|f n| \ast 1 \leq |f n| \ast \text{real} (\text{Suc } n) \)

by (rule mult-left-mono) auto

show \( \text{norm } (f n \ast x^n) \leq \text{norm } (f n \ast \text{real} (\text{Suc } n) \ast x^n) \)

unfolding real-norm-def abs-mult

by (rule mult-right-mono) (auto simp add: le[unfolded mult-1-right])

qed (rule powser-insidea[of converges[of \( R' \in \{-R..<..<R\} \)] \( \text{norm } x < \text{norm } R' \)]

from this[THEN summable-mult2[where c=x], unfolded mult.assoc, unfolded mult.commute]
show summable (?f x) by auto 
\}
show summable (?f' x0)
  using converges[OF ?x0 ∈ \{-R <..< R\}] .
show x0 ∈ \{-R' <..< R'\}
  using \{x0 ∈ \{-R <..< R\}\} .
qed 
} note for-subinterval = this
let ?R = (R + |x0|) / 2
have |x0| < ?R using assms by auto
hence − ?R < x0 proof (cases x0 < 0)
case True
  hence − x0 < ?R using (|x0| < ?R) by auto
  thus ?thesis unfolding neg-less-iff-less[ symmetric, of − x0] by auto
next
case False
  have − ?R < 0 using assms by auto
  also have ... ≤ x0 using False by auto
  finally show ?thesis .
qed
hence 0 < ?R ?R < R − ?R < x0 and x0 < ?R using assms by auto
from for-subinterval[OF this]
show ?thesis .
qed

105.5 Exponential Function
definition exp :: 'a ⇒ 'a::{real-normed-field,banach}
where exp = (λx. ∑ n. x ^ n / R real (fact n))
lemma summable-exp-generic:
  fixes x :: 'a::{real-normed-algebra-1,banach}
  defines S-def: S ≡ λn. x ^ n / R real (fact n)
  shows summable S
proof –
  have S-Suc: ∀n. S (Suc n) = (x * S n) / R real (Suc n)
    unfolding S-def by (simp del: mult-Suc)
  obtain r :: real where r0: 0 < r and r1: r < 1
    using dense [OF zero-less-one] by fast
  obtain N :: nat where N: norm x < real N * r
    using reals-Archimedean3 [OF r0] by fast
  from r1 show ?thesis
proof (rule summable-ratio-test [rule-format])
  fix n :: nat
  assume n: N ≤ n
  have norm x ≤ real N * r
    using N by (rule order_less_imp_le)
also have real $N \cdot r \leq \text{real}(\operatorname{Suc} n) \cdot r$
using $r \cdot n$ by (simp add: mult-right-mono)
finally have $\|x\| \cdot \|S_n\| \leq \text{real}(\operatorname{Suc} n) \cdot r \cdot \|S_n\|
using \|S_n\| \leq \text{real}(\operatorname{Suc} n) \cdot r \cdot \|S_n\|
by (rule mult-right-mono)
hence $\|x\| \cdot \|S_n\| / \text{real}(\operatorname{Suc} n) \leq r \cdot \|S_n\|
by (simp add: pos-divide-le-eq ac-simps)
thus $\|S_{\operatorname{Suc} n}\| \leq \text{real}(\operatorname{Suc} n) \cdot r \cdot \|S_n\|
by (simp add: S-Suc inverse-eq-divide)
qed

lemma summable-norm-exp:
fixes $x$ :: 'a::{real-normed-algebra-1,banach}
shows summable ($\lambda n. \|x^n\| / \text{real}(\text{fact} n))$
proof (rule summable-norm-comparison-test [OF exI])
show summable ($\lambda n. \|x^n\| / \text{real}(\text{fact} n))
by (rule summable-exp-generic)
fix $n$
show $\|x^n\| / \text{real}(\text{fact} n)) \leq \|x^n\| / \text{real}(\text{fact} n))$
by (simp add: norm-power-ineq)
qed

lemma summable-exp: summable ($\lambda n. 1 / \text{real}(\text{fact} n)) \cdot x^n$
using summable-exp-generic [where $x=x$] by simp

lemma exp-converges: ($\lambda n. \|x^n\| / \text{real}(\text{fact} n)) \sums x$
unfolding exp-def by (rule summable-exp-generic [THEN summable-sums])

lemma exp-fdiffs:
diffs ($\lambda n. \|x^n\| / \text{real}(\text{fact} n)) = ($\lambda n. \|x^n\| (\text{fact} n))$
by (simp add: diffs-def mult.assoc [symmetric] real-of-nat-def of-nat-mult
del: mult-Suc of-nat-Suc)

lemma diffs-of-real: diffs ($\lambda n. \text{real}(f n)) = ($\lambda n. \text{real}(\text{diffs} f n))
by (simp add: diffs-def)

lemma DERIV-exp [simp]: DERIV exp $x := \exp(x)$
unfolding exp-def scaleR-conv-of-real
apply (rule DERIV-cong)
apply (rule termdiffs [where $K=\text{of-real}(1 + \|x\|)]
apply (simp-all only: diffs-of-real scaleR-conv-of-real exp-fdiffs)
apply (rule exp-converges [THEN sums-summable, unfolded scaleR-conv-of-real])
apply (simp del: of-real-add)
done

declare DERIV-exp[THEN DERIV-chain2, derivative-intros]
lemma isCont-exp: isCont exp x
  by (rule DERIV-exp [THEN DERIV-isCont])

lemma isCont-exp'[simp]: isCont f a \implies isCont \(\lambda x. \text{exp} (f x)\) a
  by (rule isCont-o2 [OF isCont-exp])

lemma tendsto-exp [tendsto-intros]:
  \((f \xrightarrow{} a) F \implies ((\lambda x. \text{exp} (f x)) \xrightarrow{} a) F\)
  by (rule isCont-tendsto-compose [OF isCont-exp])

lemma continuous-exp [continuous-intros]:
  continuous F f \implies continuous F (\lambda x. \text{exp} (f x))
  unfolding continuous-def by (rule tendsto-exp)

lemma continuous-on-exp [continuous-intros]:
  continuous-on s f \implies continuous-on s (\lambda x. \text{exp} (f x))
  unfolding continuous-on-def by (auto intro: tendsto-exp)

105.5.1 Properties of the Exponential Function

lemma powser-zero:
  fixes f :: nat \Rightarrow 'a::{real-normed-algebra-1}
  shows \(\sum n. f n \cdot 0^n = f 0\)
  proof -
    have \((\sum n<1. f n \cdot 0^n) = (\sum n. f n \cdot 0^n)\)
      by (subst suminf-finite[where N=N(0)]) (auto simp: power-0-left)
    thus ?thesis unfolding One-nat-def by simp
  qed

lemma exp-zero [simp]: exp 0 = 1
  unfolding exp-def by (simp add: scaleR-conv-of-real powser-zero)

lemma exp-series-add:
  fixes x y :: 'a::{real-field}
  defines S-def: \(S \equiv \lambda x n. x \cdot n / n!\) real (fact n)
  shows \(\sum i\leq n. S x i \cdot S y (n-i)\)
  proof (induct n)
    case 0
    show ?case
      unfolding S-def by simp
  next
    case (Suc n)
    have S-Suc: \(\forall x n. S x (Suc n) = (x \cdot S x n) / n!\) real (Suc n)
      unfolding S-def by (simp del: mult-Suc)
    hence times-S: \(\forall x n. x \cdot S x n = real (Suc n) \cdot S x (Suc n)\)
      by simp
    have real (Suc n) \cdot S x (Suc n) = (x + y) \cdot S x (Suc n)
by (simp only: times-S)
also have \(\ldots = (x + y) \ast (\sum_{i \leq n} S \times i \ast S \times y \times (n-i))\)
by (simp only: Suc)
also have \(\ldots = x \ast (\sum_{i \leq n} S \times i \ast S \times y \times (n-i))\)
\(+ y \ast (\sum_{i \leq n} S \times i \ast S \times y \times (n-i))\)
by (rule distrib-right)
also have \(\ldots = (\sum_{i \leq n} (x \ast S \times i) \ast S \times y \times (n-i))\)
\(+ (\sum_{i \leq n} S \times i \ast (y \ast S \times y \times (n-i)))\)
by (simp only: setsum-right-distrib ac-simps)
also have \(\ldots = (\sum_{i \leq n} \text{real} \ast (\text{Suc } i) \ast R \ast (S \times x \ast i \ast S \times y \ast (n-i)))\)
\(+ (\sum_{i \leq n} \text{real} \ast (\text{Suc } n-i) \ast R \ast (S \times i \ast S \times y \ast (Suc \times n-i)))\)
by (simp add: times-S Suc-diff-le)
also have \(\sum_{i \leq n} \text{real} \ast (\text{Suc } i) \ast R \ast (S \times x \ast i \ast S \times y \ast (n-i))\) =
(\(\sum_{i \leq Suc \times n} \text{real} \ast i \ast R \ast (S \times x \ast i \ast S \times y \ast (Suc \times n-i))\))
by (subst setsum-atMost-Suc-shift simp)
also have \(\sum_{i \leq Suc \times n} \text{real} \ast (Suc \times n-i) \ast R \ast (S \times i \ast S \times y \ast (Suc \times n-i))\) =
(\(\sum_{i \leq Suc \times n} \text{real} \ast (Suc \times n-i) \ast R \ast (S \times i \ast S \times y \ast (Suc \times n-i))\))
by (simp only: setsum.distrib [symmetric] scaleR-left-distrib [symmetric]
real-of-nat-add [symmetric] simp)
also have \(\ldots = \text{real} \ast (Suc \times n) \ast R \ast (\sum_{i \leq Suc \times n} S \times i \ast S \times y \ast (Suc \times n-i))\)
by (simp only: scaleR-right.setsum)
finally show
\(S \times (x + y) \ast (Suc \times n) = (\sum_{i \leq Suc \times n} S \times i \ast S \times y \ast (Suc \times n-i))\)
by (simp del: setsum-cl-int-Suc)

qed

lemma exp-add: \(\exp \ast (x + y) = \exp \ast x \ast \exp y\)
unfolding exp-def
by (simp only: Cauchy-product summable-norm-exp exp-series-add)

lemma mult-exp-exp: \(\exp \ast x \ast \exp y = \exp \ast (x + y)\)
by (rule exp-add [symmetric])

lemma exp-of-real: \(\exp \ast (\text{of-real } x) = \text{of-real } (\exp \ast x)\)
unfolding exp-def
apply (subst suminf-of-real)
apply (rule summable-exp-generic)
apply (simp add: scaleR-conv-of-real)
done

lemma exp-not-eq-zero [simp]: \(\exp \neq 0\)
proof
have \(\exp \ast x \ast \exp \ast (\neg x) = 1\)
by (simp add: mult-exp-exp)
also assume \(\exp \ast x = 0\)
finally show False by simp
qed

lemma exp-minus: \( \exp (-x) = \inverse {\exp x} \)
by (rule inverse-unique [symmetric], simp add: mult-exp-exp)

lemma exp-diff: \( \exp (x - y) = \exp x / \exp y \)
using exp-add [of \( x - y \)] by (simp add: exp-minus divide-inverse)

105.5.2 Properties of the Exponential Function on Reals

Comparisons of \( \exp x \) with zero.

Proof: because every exponential can be seen as a square.

lemma exp-ge-zero [simp]: \( 0 \leq \exp (x :: \text{real}) \)
proof -
  have \( 0 \leq \exp (x/2) * \exp (x/2) \) by simp
  thus ?thesis by (simp add: exp-add [symmetric])
qed

lemma exp-gt-zero [simp]: \( 0 < \exp (x :: \text{real}) \)
by (simp add: order-less-le)

lemma not-exp-less-zero [simp]: \( \neg \exp (x :: \text{real}) < 0 \)
by (simp add: not-less)

lemma not-exp-le-zero [simp]: \( \neg \exp (x :: \text{real}) \leq 0 \)
by (simp add: not-le)

lemma abs-exp-cancel [simp]: \( |\exp x :: \text{real}| = \exp x \)
by simp

lemma exp-real-of-nat-mult: \( \exp (\text{real} n * x) = \exp x ^ n \)
by (induct n) (auto simp add: real-of-nat-Suc distrib-left exp-add mult.commute)

Strict monotonicity of exponential.

lemma exp-ge-add-one-self-aux:
  assumes \( 0 \leq (x :: \text{real}) \) shows \( 1 + x \leq \exp (x) \)
using order-le-imp-less-or-eq [OF assms]
proof
  assume \( 0 < x \)
  have \( 1 + x \leq (\sum n < 2. \ inverse (\text{real} (fact n)) * x ^ n) \)
    by (auto simp add: numeral-2-eq-2)
  also have \( \ldots \leq (\sum n. \ inverse (\text{real} (fact n)) * x ^ n) \)
    apply (rule setsum-le-suminf [OF summable-exp])
    using \( 0 < x \)
    apply (auto simp add: zero-le-mult-iff)
  done
  finally show \( 1 + x \leq \exp x \)
  by (simp add: exp-def)
next
  assume \( 0 = x \)
  then show \( 1 + x \leq \exp x \)
    by auto
qed

lemma \textit{exp-gt-one}: \( 0 < (x::\text{real}) \implies 1 < \exp x \)
proof –
  assume \( x: 0 < x \)
  hence \( 1 < 1 + x \) by simp
  also from \( x \) have \( 1 + x \leq \exp x \)
    by (simp add: exp-ge-add-one-self-aux)
  finally show \( \text{thesis} \).
qed

lemma \textit{exp-less-mono}:
  fixes \( x, y :: \text{real} \)
  assumes \( x < y \)
  shows \( \exp x < \exp y \)
proof –
  from \( (x < y) \) have \( 0 < y - x \) by simp
  hence \( 1 < \exp (y - x) \) by (rule exp-gt-one)
  hence \( 1 < \exp y / \exp x \) by (simp only: exp-diff)
  thus \( \exp x < \exp y \) by simp
qed

lemma \textit{exp-less-cancel}:
  \( \exp (x :: \text{real}) < \exp y \implies x < y \)
  unfolding linorder-not-le [symmetric]
  by (auto simp add: order-le-less \( \exp \)-less-mono)

lemma \textit{exp-less-cancel-iff} \[iff\]: \( \exp (x :: \text{real}) < \exp y \iff x < y \)
  by (auto intro: \( \exp \)-less-mono \( \exp \)-less-cancel)

lemma \textit{exp-le-cancel-iff} \[iff\]: \( \exp (x :: \text{real}) \leq \exp y \iff x \leq y \)
  by (auto simp add: linorder-not-less [symmetric])

lemma \textit{exp-inj-iff} \[iff\]: \( \exp (x :: \text{real}) = \exp y \iff x = y \)
  by (simp add: order-eq-iff)

Comparisons of \( \exp x \) with one.

lemma \textit{one-less-exp-iff} \[simp\]: \( 1 < \exp (x :: \text{real}) \iff 0 < x \)
  using \( \exp \)-less-cancel-iff \[where \( x=0 \) and \( y=x \)] by simp

lemma \textit{exp-less-one-iff} \[simp\]: \( \exp (x :: \text{real}) < 1 \iff x < 0 \)
  using \( \exp \)-less-cancel-iff \[where \( x=x \) and \( y=0 \)] by simp

lemma \textit{one-le-exp-iff} \[simp\]: \( 1 \leq \exp (x :: \text{real}) \iff 0 \leq x \)
  using \( \exp \)-le-cancel-iff \[where \( x=0 \) and \( y=x \)] by simp
THEORY “Transcendental”

lemma exp-le-one-iff [simp]: exp (x::real) ≤ 1 ←→ x ≤ 0
using exp-le-cancel-iff [where x=x and y=0] by simp

lemma exp-eq-one-iff [simp]: exp (x::real) = 1 ←→ x = 0
using exp-inj-iff [where x=x and y=0] by simp

lemma lemma-exp-total: 1 ≤ y =⇒ ∃x. 0 ≤ x & x ≤ y - 1 & exp(x::real) = y
proof (rule IVT)
assume 1 ≤ y
hence 0 ≤ y - 1 by simp
hence 1 + (y - 1) ≤ exp (y - 1) by (rule exp-ge-add-one-self-aux)
thus y ≤ exp (y - 1) by simp
qed (simp-all add: le-diff-eq)

lemma exp-total: 0 < (y::real) =⇒ ∃x. exp x = y
proof (rule linorder-le-cases [of 1 y])
assume 1 ≤ y
thus ∃x. exp x = y by (fast dest: lemma-exp-total)
next
assume 0 < y and y ≤ 1
hence 1 ≤ inverse y by (simp add: one-le-inverse-iff)
then obtain x where exp x = inverse y by (fast dest: lemma-exp-total)
hence exp (- x) = y by (simp add: exp-minus)
thus ∃x. exp x = y ..
qed

105.6 Natural Logarithm

definition ln :: real ⇒ real
where ln x = (THE u. exp u = x)

lemma ln-exp [simp]: ln (exp x) = x
by (simp add: ln-def)

lemma exp-ln [simp]: 0 < x =⇒ exp (ln x) = x
by (auto dest: exp-total)

lemma exp-ln-iff [simp]: exp (ln x) = x =⇒ 0 < x
by (metis exp-gt-zero exp-ln)

lemma ln-unique: exp y = x =⇒ ln x = y
by (erule subst, rule ln-exp)

lemma ln-one [simp]: ln 1 = 0
by (rule ln-unique) simp

lemma ln-mult: 0 < x =⇒ 0 < y =⇒ ln (x * y) = ln x + ln y
by (rule ln-unique) (simp add: exp-add)
lemma \textit{ln-inverse:} \(0 < x \Rightarrow \ln \text{ (inverse } x) = -\ln x\)
by (rule \textit{ln-unique}) (simp add: \textit{exp-minus})

lemma \textit{ln-div:} \(0 < x \Rightarrow 0 < y \Rightarrow \ln \text{ (} x / y) = \ln x - \ln y\)
by (rule \textit{ln-unique}) (simp add: \textit{exp-diff})

lemma \textit{ln-realpow:} \(0 < x \Rightarrow \ln \text{ (} x ^ n) = real \ n \ast \ln x\)
by (rule \textit{ln-unique}) (simp add: \textit{exp-real-of-nat-mult})

lemma \textit{ln-less-cancel-iff} \[\text{simp}]: \(0 < x \Rightarrow 0 < y \Rightarrow \ln x < \ln y \iff x < y\)
by (subst \textit{exp-less-cancel-iff} [\textit{symmetric}]) simp

lemma \textit{ln-le-cancel-iff} \[\text{simp}]: \(0 < x \Rightarrow 0 < y \Rightarrow \ln x \leq \ln y \iff x \leq y\)
by (simp add: \textit{linorder-not-less} [\textit{symmetric}])

lemma \textit{ln-inj-iff} \[\text{simp}]: \(0 < x \Rightarrow 0 < y \Rightarrow \ln x = \ln y \iff x = y\)
by (simp add: \textit{order-eq-iff})

lemma \textit{ln-add-one-self-le-self} \[\text{simp}]: \(0 \leq x \Rightarrow \ln \text{ (} 1 + x) \leq x\)
apply (rule \textit{exp-le-cancel-iff} \[THEN \textit{iffD1}\])
apply (simp add: \textit{exp-ge-add-one-self-aux})
done

lemma \textit{ln-less-self} \[\text{simp}]: \(0 < x \Rightarrow \ln x < x\)
by (rule \textit{order-less-le-trans} \[\textit{where } y=\ln \text{ (} 1 + x)\]) simp-all

lemma \textit{ln-ge-zero} \[\text{simp}]: \(1 \leq x \Rightarrow 0 \leq \ln x\)
using \textit{ln-le-cancel-iff} \[\textit{of 1 x}\] by simp

lemma \textit{ln-ge-zero-imp-ge-one} \(0 \leq \ln x \Rightarrow 0 < x \Rightarrow 1 \leq x\)
using \textit{ln-le-cancel-iff} \[\textit{of 1 x}\] by simp

lemma \textit{ln-ge-zero-iff} \[\text{simp}]: \(0 < x \Rightarrow 0 \leq \ln x \iff 1 \leq x\)
using \textit{ln-le-cancel-iff} \[\textit{of 1 x}\] by simp

lemma \textit{ln-less-zero-iff} \[\text{simp}]: \(0 < x \Rightarrow \ln x < 0 \iff x < 1\)
using \textit{ln-less-cancel-iff} \[\textit{of x 1}\] by simp

lemma \textit{ln-gt-zero} \[\text{simp}]: \(1 < x \Rightarrow 0 < \ln x\)
using \textit{ln-less-cancel-iff} \[\textit{of 1 x}\] by simp

lemma \textit{ln-gt-zero-imp-gt-one} \(0 < \ln x \Rightarrow 0 < x \Rightarrow 1 < x\)
using \textit{ln-less-cancel-iff} \[\textit{of 1 x}\] by simp

lemma \textit{ln-gt-zero-iff} \[\text{simp}]: \(0 < x \Rightarrow \ln x < 0 \iff x > 1\)
using \textit{ln-inj-iff} \[\textit{of x 1}\] by simp
THEORY "Transcendental"

lemma ln-less-zero: \(0 < x \implies x < 1 \implies \ln x < 0\)
  by simp

lemma ln-neg-is-const: \(x \leq 0 \implies \ln x = (\text{THE } x. \text{False})\)
  by (auto simp add: ln-def intro!: arg-cong[where \(f=\text{The}\)])

lemma isCont-ln: assumes \(x \neq 0\) shows isCont \(\ln x\)
proof cases
  assume \(0 < x\)
  moreover then have isCont \(\ln (\exp (\ln x))\)
    by (intro isCont-inv-fun[where \(d=|x|\) and \(f=\exp\)]) auto
  ultimately show \(\text{thesis}\)
    by simp
next
  assume \(\neg 0 < x\) with \(\langle x \neq 0 \rangle\) show isCont \(\ln x\)
    unfolding isCont-def
    by (rule tendsto-ln)

done

lemma DERIV-ln: \(0 < x \implies \text{DERIV } \ln x :> \text{inverse } x\)
  apply (rule DERIV-inverse-function [where \(f=\exp\) and \(a=0\) and \(b=x+1\)])
  apply (auto intro: DERIV-cong [OF DERIV-exp exp-ln] isCont-ln)
  done

lemma DERIV-ln-divide: \(0 < x \implies \text{DERIV } \ln x :> 1 / x\)
by (rule DERIV-ln[THEN DERIV-cong], simp, simp add: divide-inverse)

declare DERIV-ln-divide[THEN DERIV-chain2, derivative-intros]

lemma ln-series:
  assumes 0 < x and x < 2
  shows ln x = (SUM n. (-1) ^ n * (1 / real (n + 1)) * (x - 1) ^ (Suc n))
  (is ln x = suminf (?f (x - 1)))
proof -
  let ?f' = λx n. (-1) ^ n * (x - 1) ^ n

  have ln x - suminf (?f (x - 1)) = ln 1 - suminf (?f (1 - 1))
  proof (rule DERIV-isconst3[where x=1])
    fix x :: real
    assume x ∈ {0 <..< 2}
    hence 0 < x and x < 2 by auto
    have norm (1 - x) < 1
      using (0 < x) and (x < 2) by auto
    have 1 / x = 1 / (1 - (1 - x)) by auto
    also have ... = (SUM n. (1 - x) ^ n)
      using unfolding geometric-sums[OF norm (1 - x) < 1] by (rule sums-unique)
    also have ... = suminf (?f' x)
      unfolding power-mult-distrib[symmetric]
      by (rule arg-cong[where f=suminf], rule arg-cong[where f=op ^], auto)
  finally have DERIV ln x :> suminf (?f' x)
  unfolding divide-inverse by auto
moreover have repos: ∃ h x :: real. h - 1 + x = h + x - 1 by auto
  have DERIV (λx. suminf (?f x)) (x - 1) :>
    (SUM n. (-1) ^ n * (1 / real (n + 1)) * real (Suc n) * (x - 1) ^ n)
  proof (rule DERIV-power-series')
    show x - 1 ∈ {- 1<..<1} and (0 :: real) < 1
      using (0 < x) (x < 2) by auto
    fix x :: real
    assume x ∈ {- 1<..<1}
    hence norm (-x) < 1 by auto
    show summable (λn. -1 ^ n * (1 / real (n + 1)) * real (Suc n) * x ^ n)
      unfolding One_nat_def
      by (auto simp add: power-mult-distrib[symmetric] summable-geometric[OF norm (-x) < 1])
  qed
  hence DERIV (λx. suminf (?f x)) (x - 1) :> suminf (?f' x)
    unfolding One_nat_def by auto
  hence DERIV (λx. suminf (?f (x - 1))) x :> suminf (?f' x)
    unfolding DERIV-def repos .
  ultimately have DERIV (λx. ln x - suminf (?f (x - 1))) x :> (suminf (?f' x) - suminf (?f' x))
    by (rule DERIV-diff)
  thus DERIV (λx. ln x - suminf (?f (x - 1))) x :> 0 by auto
THEORY "Transcendental"

qed (auto simp add: assms)
thus ?thesis by auto
qed

lemma exp-first-two-terms: exp x = 1 + x + (∑ n. inverse(fact (n+2)) * (x ^ (n+2)))
proof
have exp x = suminf (λn. inverse(fact n) * (x ^ n))
  by (simp add: exp-def)
also from summable-exp have ... = (∑ n. inverse(fact(n+2)) * (x ^ (n+2)))
  + (∑ n::nat<2. inverse(fact n) * (x ^ n)) (is - - + ?a)
  by (rule suminf-split-initial-segment)
also have ?a = 1 + x
  by (simp add: numeral-2-eq-2)
finally show ?thesis
  by simp
qed

lemma exp-bound: 0 <= (x::real) ==> x <= 1 ==> exp x <= 1 + x + x^2
proof
assume a: 0 <= x
assume b: x <= 1
{ fix n :: nat
  have 2 * 2 ^ n <= fact (n + 2)
    by (induct n) simp-all
  hence real ((2::nat) * 2 ^ n) <= real (fact (n + 2))
    by (simp only: real-of-nat-le-iff)
  hence 2 * 2 ^ n <= real (fact (n + 2))
    by simp
  hence inverse (fact (n + 2)) <= inverse (2 * 2 ^ n)
    by (rule le-imp-inverse-le) simp
  hence inverse (fact (n + 2)) <= 1/2 * (1/2) ^ n
    by (simp add: power-inverse)
  hence inverse (fact (n + 2)) * (x^n * x^2) <= 1/2 * (1/2) ^ n * (1 * x^2)
    by (rule mult-mono)
    (rule mult mono, simp-all add: power-le-one a b)
  hence inverse (fact (n + 2)) * x ^ (n + 2) <= (x^2/2) * ((1/2) ^ n)
    unfolding power-add by (simp add: ac-simps del: fact-Suc)
}

note aux1 = this
have (λn. x^2 / 2 * (1 / 2) ^ n) sums (x^2 / 2 * (1 / (1 - 1 / 2)))
  by (intro sums-mult geometric-sums, simp)

hence aux2: (λn. x^2 / 2 * (1 / 2) ^ n) sums x^2
  by simp

have suminf (λn. inverse(fact (n+2)) * (x ^ (n+2))) <= x^2
proof
  have suminf (λn. inverse(fact (n+2)) * (x ^ (n+2))) <=
    suminf (λn. (x^2/2) * ((1/2) ^ n))
apply (rule suminf-le)
apply (rule allI, rule aux1)
apply (rule summable-exp [THEN summable-ignore-initial-segment])
  by (rule sums-summable, rule aux2)
also have ... = x^2
  by (rule sums-unique [THEN sym], rule aux2)
finally show ?thesis .
qed
thus ?thesis unfolding exp-first-two-terms by auto
qed

lemma ln-one-minus-pos-upper-bound: \( 0 < x \Rightarrow x < 1 \Rightarrow \ln (1 - x) < -x \)
proof
  assume a: \( 0 \leq (x :: real) \) and b: \( x < 1 \)
  have \( (1 - x) \times (1 + x + x^2) = (1 - x^3) \)
    by (simp add: algebra-simps power2-eq-square power3-eq-cube)
  also have ... \leq 1
    by (auto simp add: a)
  finally have \( (1 - x) \times (1 + x + x^2) \leq 1 \).
  moreover have c: \( 0 < 1 + x + x^2 \)
    by (simp add: add-pos-nonneg a)
  ultimately have \( 1 - x \leq 1 / (1 + x + x^2) \)
    by (elim mult-imp-le-div-pos)
  also have ... \leq 1 / \exp x
    by (metis a abs-one b exp-bound exp-gt-zero frac-le less-eq-real-def real-sqrt-abs
      real-sqrt-pow2-iff real-sqrt-power)
  also have ... = \exp (-x)
    by (auto simp add: exp-minus divide-inverse)
  finally have \( 1 - x \leq \exp (-x) \).
  also have \( 1 - x = \exp (\ln (1 - x)) \)
    by (metis b diff-0 exp-ln-iff less-iff-diff-less-0 minus-diff-eq)
  finally have \( \exp (\ln (1 - x)) \leq \exp (-x) \).
  thus ?thesis by (auto simp only: exp-le-cancel-iff)
qed

lemma exp-ge-add-one-self [simp]: \( 1 + (x :: real) \leq \exp x \)
apply (case-tac 0 \leq x)
apply (erule exp-ge-add-one-self-aux)
apply (case-tac x \leq -1)
apply (subgoal-tac 1 + x \leq 0)
apply (erule order-trans)
apply simp
apply simp
apply (subgoal-tac 1 + x = \exp(\ln (1 + x)))
apply (erule ss subst)
apply (subgoal-tac ln (1 - (- x)) \leq (- x))
apply simp

apply (rule ln-one-minus-pos-upper-bound)
apply auto
done

lemma ln-one-plus-pos-lower-bound: $0 < x \Rightarrow x < 1 \Rightarrow x - x^2 < \ln (1 + x)$
proof -
  assume a: $0 < x$ and b: $x < 1$
  have $\exp (x - x^2) = \exp x / \exp (x^2)$
    by (rule exp-diff)
  also have ... $< (1 + x + x^2) / \exp (x^2)$
    by (metis a b divide-right-mono exp-bound exp-ge-zero)
  also have ... $< (1 + x + x^2) / (1 + x^2)$
    by (simp add: a divide-left-mono add-pos-nonneg)
  also have ... $< 1 + x$
    by (simp add: field-simps add-strict-increasing zero-le-mult-iff)
  finally have $\exp (x - x^2) < \exp (\ln (1 + x))$.
  also have ...
    by (auto simp only: exp-ln-iff [THEN sym])
qed

finally have $\exp (x - x^2) < \exp (\ln (1 + x))$.
  thus ?thesis
    by (metis exp-le-cancel-iff)
qed

lemma ln-one-minus-pos-lower-bound:
$0 < x \Rightarrow x < (1 / 2) \Rightarrow - x - 2 * x^2 < \ln (1 - x)$
proof -
  assume a: $0 < x$ and b: $x < (1 / 2)$
  from b have c: $x < 1$ by auto
  then have $\ln (1 - x) = - \ln (1 + x / (1 - x))$
    by (auto simp only: exp-ln-iff [THEN sym])
  finally have ...$= - \exp (\ln (1 - x))$.
  also have $- (x / (1 - x)) < ...$
  proof -
    have $\ln (1 + x / (1 - x)) < x / (1 - x)$
      by (intro ln-add-one-self-le-self)
    thus ?thesis
      by auto
  qed
  also have $- (x / (1 - x)) = -x / (1 - x)$
    by auto
  finally have $d: - x / (1 - x) < \ln (1 - x)$.

have \(0 < 1 - x\) using \(a\) \(b\) by simp

hence \(c < -2 * x^2 < -x / (1 - x)\)

using mult-right-le-one-le[of \(x\) \(2\) \(x\)] \(a\) \(b\)

by (simp add: field-simps power2-eq-square)

from \(e\) \(d\) show \(-x - 2 * x^2 <\) \(\leq\) \(\ln\) \((1 - x)\)

by (rule order-trans)

qed

lemma \(\text{ln-add-one-self-le-self2}\): \(-1 < x \Longrightarrow \ln(1 + x) \leq x\)

apply (subgoal-tac \(\ln\) \((1 + x)\) \(\leq\) \((\exp\) \(x)\), simp)

apply (subst \(\ln\)-le-cancel-iff)

apply auto

done

lemma \(\text{abs-ln-one-plus-x-minus-x-bound-nonneg}\):
\(\leq\) \(x\) \(\Rightarrow\) \(x\) \(\Longrightarrow\) \(\ln\) \((1 + x) - x\) \(\leq\) \(x^2\)

proof –

assume \(x\): \(0 \leq x\)

assume \(xl\): \(x \leq 1\)

from \(x\) have \(\ln\) \((1 + x)\) \(\leq\) \(x\)

by (rule \(\ln\)-add-one-self-le-self)

then have \(\ln\) \((1 + x) - x \leq 0\)

by simp

then have \(\text{abs}((ln(1 + x) - x)) = - (\ln(1 + x) - x)\)

by (rule \(\text{abs-of-nonpos}\))

also have \(\ldots = x - \ln\) \((1 + x)\)

by simp

also have \(\ldots \leq x^2\)

proof –

from \(x\) \(xl\) have \(x - x^2 \leq \ln\) \((1 + x)\)

by (intro \(\ln\)-one-plus-pos-lower-bound)

thus \(?\)thesis

by simp

qed

finally show \(?\)thesis .

qed

lemma \(\text{abs-ln-one-plus-x-minus-x-bound-nonpos}\):
\((-1 / 2) \leq x \Longrightarrow x \leq 0 \Longrightarrow \text{abs}((ln(1 + x) - x) \leq 2 * x^2\)

proof –

assume \(a\): \((-1 / 2) \leq x\)

assume \(b\): \(x \leq 0\)

have \(\text{abs}((ln(1 + x) - x)) = x - \ln(1 - (-x))\)

apply (subst \(\text{abs-of-nonpos}\))

apply simp

apply (rule \(\ln\)-add-one-self-le-self2)

using \(a\) apply \(auto\)

done

also have \(\ldots \leq 2 * x^2\)
apply (subgoal-tac \(-x\) \(-2 \cdot (-x)^2 \leq \ln (1 - (-x)))
apply (simp add: algebra-simps)
apply (rule ln-one-minus-pos-lower-bound)
using a b apply auto
done
finally show \(\text{thesis} \).
qed

lemma abs-ln-one-plus-x-minus-x-bound:
\(\text{abs } x \leq 1 / 2 \Rightarrow \text{abs } (\ln (1 + x) - x) \leq 2 \cdot x^2\)
apply (case-tac 0 \leq x)
apply (rule order-trans)
apply (rule abs-ln-one-plus-x-minus-x-bound-nonneg)
apply auto
apply (rule abs-ln-one-plus-x-minus-x-bound-nonpos)
apply auto
done

lemma ln-x-over-x-mono: \(\exp 1 \leq x \Rightarrow x \leq y \Rightarrow (\ln y / y) \leq (\ln x / x)\)
proof –
assume \(x\): \(\exp 1 \leq x \leq y\)
moreover have \(0 < \exp (1::real)\) by simp
ultimately have \(a: \ 0 < x\) and \(b: \ 0 < y\)
  by (fast intro; less-le-trans order-trans)+
have \(x \cdot \ln y - x \cdot \ln x = x \cdot (\ln y - \ln x)\)
  by (simp add: algebra-simps)
also have \(... = x \cdot \ln(y / x)\)
  by (simp only: ln-div a b)
also have \(y / x = (x + (y - x)) / x\)
  by simp
also have \(... = 1 + (y - x) / x\)
  using \(x\) \(a\) by (simp add: field-simps)
also have \(x \cdot \ln(1 + (y - x) / x) \leq x \cdot ((y - x) / x)\)
  using \(x\) \(a\)
  by (intro mult-left-mono \(\ln\)-add-one-self-le-self) simp-all
also have \(... = y - x\) using \(a\) by simp
also have \(... = (y - x) \cdot \ln (\exp 1)\) by simp
also have \(... \leq (y - x) \cdot \ln x\)
  apply (rule mult-left-mono)
  apply (subst ln-le-cancel-iff)
  apply fact
  apply (rule a)
  apply (rule x)
  using \(x\) apply simp
  done
also have \(... = y \cdot \ln x - x \cdot \ln x\)
  by (rule left-diff-distrib)
finally have \(x \cdot \ln y \leq y \cdot \ln x\)
  by arith
then have \( \ln y \leq (y \ast \ln x) / x \) using \( \text{by (simp add: field-simps)} \)
also have \( \ldots = y \ast (\ln x / x) \) by simp
finally show \( ?\text{thesis} \) using \( b \) by (simp add: field-simps)
qed

lemma \( \ln\text{-le-minus-one} \): \( 0 < x \Longrightarrow \ln x \leq x - 1 \)
using \( \text{exp-ge-add-one-self[of } \ln x \] by simp \)

lemma \( \ln\text{-eq-minus-one} \):
assumes \( 0 < x \ln x = x - 1 \)
shows \( x = 1 \)
proof (cases rule: linorder-cases)
assume \( x < 1 \)
from dense[OF \( \{ x < 1 \} \) obtain \( a \) where \( x < a a < 1 \) by blast
from \( x < a \) have \( ?l x < ?l a \)
proof (rule DERIV-pos-imp-increasing, safe)
fix \( y \)
assume \( x \leq y y \leq a \)
with \( \{ 0 < x \} \{ a < 1 \} \) have \( 0 < 1 / y - 1 0 < y \)
by (auto simp: field-simps)
with \( D \) show \( \exists z. \DERIV ?l y :> z \wedge 0 < z \)
by auto
qed
also have \( \ldots \leq 0 \)
using \( \ln\text{-le-minus-one} \) \( \{ 0 < x \} \{ x < a \} \) by (auto simp: field-simps)
finally show \( x = 1 \) using \( \text{assms by auto} \)
next
assume \( 1 < x \)
from dense[OF this] obtain \( a \) where \( 1 < a a < x \) by blast
from \( \{ a < x \} \) have \( ?l x < ?l a \)
proof (rule DERIV-neg-imp-decreasing, safe)
fix \( y \)
assume \( a \leq y y \leq x \)
with \( \{ 1 < a \} \) have \( 1 / y - 1 0 < y \)
by (auto simp: field-simps)
with \( D \) show \( \exists z. \DERIV ?l y :> z \wedge z < 0 \)
by blast
qed
also have \( \ldots \leq 0 \)
using \( \ln\text{-le-minus-one} \) \( \{ 1 < a \} \) by (auto simp: field-simps)
finally show \( x = 1 \) using \( \text{assms by auto} \)
next
assume \( x = 1 \)
then show ?thesis by simp
qed

lemma exp-at-bot: (exp ---> (0::real)) at-bot
  unfolding tendsto-Zfun-iff
proof (rule ZfunI, simp add: eventually-at-bot-dense)
  fix r :: real assume 0 < r
  { fix x
    assume x < ln r
    then have exp x < exp (ln r)
      by simp
    with ⟨0 < r⟩ have exp x < r
      by simp
  }
  then show ∃ k. ∀ n<k. exp n < r by auto
qed

lemma exp-at-top: LIM x at-top. exp x :: real ---> at-top
  by (rule filterlim-at-top-at-top[where Q=λx. True and P=λx. 0 < x and g=ln])
    (auto intro: eventually-gt-at-top)

lemma ln-at-0: LIM x at-right 0. ln x :: real ---> at-bot
  by (rule filterlim-at-bot-at-right[where Q=λx. True and P=λx. 0 < x and g=exp])
    (auto simp: eventually-gt-at-top)

lemma ln-at-top: LIM x at-top. ln x :: real ---> at-top
  by (rule filterlim-at-top-at-top[where Q=λx. True and P=λx. 0 < x and g=exp])
    (auto intro: eventually-gt-at-top)

lemma tendsto-power-div-exp-0: ((λx. x ^ k / exp x) ---> (0::real)) at-top
proof (induct k)
  case 0
  show ((λx. x ^ 0 / exp x) ---> (0::real)) at-top
    by (simp add: inverse-eq-divide[symmetric])
      (metis filterlim-compose[OF tendsto-inverse-0] exp-at-top filterlim-monotone at-top-le-at-infinity order_refl)
next
case (Suc k)
  show ?case
  proof (rule lhospital-at-top-at-top)
    show eventually (λx. DERIV (λx. x ^ Suc k) x :> (real (Suc k) * x ^ k)) at-top
      by eventually-elim (intro derivative-eq-intros, auto)
    show eventually (λx. DERIV exp x :> exp x) at-top
      by eventually-elim auto
    show eventually (λx. exp x ≠ 0) at-top
by auto
from tendsto-mult[of tendsto-const Suc, of real (Suc k)]
show \((\lambda x. \text{real} (\text{Suc} k) \cdot x - k / \exp x) \to 0\) at-top
by simp
qed (rule exp-at-top)
qed

definition pour :: [real,real] => real (infixr pour 80)
— exponentiation with real exponent
where \(x \text{ pour} \ a = \exp(a * \ln x)\)

definition log :: [real,real] => real
— logarithm of \(x\) to base \(a\)
where \(\log a x = \ln x / \ln a\)

lemma tendsto-log [tendsto-intros]:
\[ (f \to a) F; (g \to b) F; 0 < a; a \neq 1; 0 < b \implies (\lambda x. \log f x \cdot g x) \to \log a b) F \]
unfolding log-def by (intro tendsto-intros) auto

lemma continuous-log:
assumes continuous F f
and continuous F g
and \(0 < f (\text{Lim } F (\lambda x. x))\)
and \(f (\text{Lim } F (\lambda x. x)) \neq 1\)
and \(0 < g (\text{Lim } F (\lambda x. x))\)
shows continuous F (λx. log (f x) (g x))
using assms unfolding continuous-def by (rule tendsto-log)

lemma continuous-at-within-log[continuous-intros]:
assumes continuous (at a within s) f
and continuous (at a within s) g
and \(0 < f a\)
and \(f a \neq 1\)
and \(0 < g a\)
shows continuous (at a within s) (λx. log (f x) (g x))
using assms unfolding continuous-within by (rule tendsto-log)

lemma isCont-log[continuous-intros, simp]:
assumes isCont f a isCont g a \(0 < f a\) \(a \neq 1\) \(0 < g a\)
shows isCont (λx. log (f x) (g x)) a
using assms unfolding continuous-at by (rule tendsto-log)

lemma continuous-on-log[continuous-intros]:
assumes continuous-on s f continuous-on s g
and \(\forall x \in s. 0 < f x\) \(\forall x \in s. f x \neq 1\) \(\forall x \in s. 0 < g x\)
shows continuous-on s (λx. log (f x) (g x))
using assms unfolding continuous-on-def by (fast intro: tendsto-log)

lemma powr-one-eq-one [simp]: 1 powr a = 1
  by (simp add: powr-def)

lemma powr-zero-eq-one [simp]: x powr 0 = 1
  by (simp add: powr-def)

lemma powr-one-gt-zero-iff [simp]: (x powr 1 = x) = (0 < x)
  by (simp add: powr-def)

declare powr-one-gt-zero-iff [THEN iffD2, simp]

lemma powr-mult: 0 < x =⇒ 0 < y =⇒ (x * y) powr a = (x powr a) * (y powr a)
  by (simp add: powr-def exp-add [symmetric] ln-mult distrib-left)

lemma powr-gt-zero [simp]: 0 < x powr a
  by (simp add: powr-def)

lemma powr-ge-pzero [simp]: 0 <= x powr y
  by (rule order-less-imp-le, rule powr-gt-zero)

lemma powr-not-zero [simp]: x powr a ≠ 0
  by (simp add: powr-def)

lemma powr-divide: 0 < x =⇒ 0 < y =⇒ (x / y) powr a = (x powr a) / (y powr a)
  apply (simp add: divide-inverse positive-imp-inverse-positive powr-mult)
  done

lemma powr-divide2: x powr a / x powr b = x powr (a - b)
  apply (simp add: powr-def)
  apply (subst exp-diff [THEN sym])
  apply (simp add: left-diff-distrib)
  done

lemma powr-add: x powr (a + b) = (x powr a) * (x powr b)
  by (simp add: powr-def exp-add [symmetric] distrib-right)

lemma powr-mult-base: 0 < x =⇒ x * x powr y = x powr (1 + y)
  using assms by (auto simp: powr-add)

lemma powr-powr: (x powr a) powr b = x powr (a * b)
  by (simp add: powr-def)

lemma powr-powr-swap: (x powr a) powr b = (x powr b) powr a
  by (simp add: powr-powr mult.commute)
lemma pour-minus: \( x \, \text{powr} \, (-a) = \text{inverse} \, (x \, \text{powr} \, a) \)
by (simp add: powr-def exp-minus [symmetric])

lemma pour-minus-divide: \( x \, \text{powr} \, (-a) = 1/(x \, \text{powr} \, a) \)
by (simp add: divide-inverse powr-minus)

lemma pour-less-mono: \( a < b \implies 1 < x \implies x \, \text{powr} \, a < x \, \text{powr} \, b \)
by (simp add: powr-def)

lemma pour-less-cancel: \( x \, \text{powr} \, a < x \, \text{powr} \, b \implies 1 < x \implies a < b \)
by (simp add: powr-def)

lemma pour-less-cancel-iff [simp]: \( 1 < x \implies (x \, \text{powr} \, a < x \, \text{powr} \, b) = (a < b) \)
by (blast intro: powr-less-cancel powr-less-mono)

lemma log-ln: \( \ln x = \log (\exp (1)) \, x \)
by (simp add: log-def)

lemma DERIV-log:
assumes \( x > 0 \)
shows \( \text{DERIV} (\lambda y. \log b \, y) \, x :> 1/ (\ln b \, \ast \, x) \)
proof –
def \( \text{lb} \equiv 1/\ln b \)
moreover have \( \text{DERIV} (\lambda y. \text{lb} \, \ast \, \ln y) \, x :> \text{lb} / x \)
using \( (x > 0) \) by (auto intro!: derivative-eq-intros)
ultimately show \( \text{?thesis} \)
by (simp add: log-def)
qed

lemmas DERIV-log[THEN DERIV-chain2, derivative-intros]

lemma pour-log-cancel [simp]: \( 0 < a \implies a \neq 1 \implies 0 < x \implies a \, \text{powr} \, (\log a \, x) = x \)
by (simp add: powr-def log-def)

lemma log-powr-cancel [simp]: \( 0 < a \implies a \neq 1 \implies \log a \, (a \, \text{powr} \, y) = y \)
by (simp add: log-def powr-def)

lemma log-mult:
\( 0 < a \implies a \neq 1 \implies 0 < x \implies 0 < y \implies \log a \, (x \, \ast \, y) = \log a \, x + \log a \, y \)
by (simp add: log-def ln-mult divide-inverse distrib-right)

lemma log-eq-div-ln-mult-log:
\( 0 < a \implies a \neq 1 \implies 0 < b \implies b \neq 1 \implies 0 < x \implies \log a \, x = (\ln b/\ln a) \, \ast \, \log b \, x \)
by (simp add: log-def divide-inverse)

Base 10 logarithms

lemma log-base-10-eq1: \(0 < x \Rightarrow \log_{10} x = (\ln (\exp 1) / \ln 10) \cdot \ln x\)
by (simp add: log-def)

lemma log-base-10-eq2: \(0 < x \Rightarrow \log_{10} x = (\log_{10} (\exp 1)) \cdot \ln x\)
by (simp add: log-def)

lemma log-one [simp]: \(\log_a 1 = 0\)
by (simp add: log-def)

lemma log-eq-one [simp]: \([\begin{array}{ll}0 < a; \ a \neq 1 \end{array}] \Rightarrow \log_a a = 1\)
by (simp add: log-def)

lemma log-inverse: \(0 < a \Rightarrow a \neq 1 \Rightarrow 0 < x \Rightarrow \log_a (\text{inverse} x) = -\log a x\)
apply (rule-tac a1 = log a x in add-left-cancel [THEN iffD1])
apply (simp add: log-mult [symmetric])
done

lemma log-divide: \(0 < a \Rightarrow a \neq 1 \Rightarrow 0 < x \Rightarrow 0 < y \Rightarrow \log_a (x/y) = \log a x - \log a y\)
by (simp add: log-mult divide-inverse log-inverse)

lemma log-less-cancel-iff [simp]: \(1 < a \Rightarrow \begin{array}{ll}0 < x; \ a \neq 1 \end{array} \Rightarrow \log a x < \log a y \iff x < y\)
apply safe
apply (rule-tac [2] powr-less-cancel)
apply (drule-tac a = log a x in powr-less-mono, auto)
done

lemma log-inj:
assumes \(1 < b\)
shows inj-on (log b) \(\{0 <..\}\)
proof (rule inj-onI, simp)
fix \(x\ y\)
assume pos: \(0 < x 0 < y\ and \ \ast: \log b x = \log b y\)
show \(x = y\)
proof (cases rule: linorder-cases)
assume \(x = y\)
then show \(\ast\)thesis by simp
next
assume \(x < y\ hence \log b x < \log b y\)
using log-less-cancel-iff[OF \(1 < b\)] pos by simp
then show \(\ast\)thesis using \(\ast\) by simp
next
assume \(y < x\ hence \log b y < \log b x\)
using log-less-cancel-iff[OF \(1 < b\)] pos by simp
then show ?thesis using * by simp
qed
qed

lemma log-le-cancel-iff [simp]:
  \(1 < a \Rightarrow 0 < x \Rightarrow 0 < y \Rightarrow (\log_a x \leq \log_a y) = (x \leq y)\)
by (simp add: linorder_not_less [symmetric])

lemma zero-less-log-cancel-iff [simp]:
  \(1 < a \Rightarrow 0 < x \Rightarrow 0 < \log_a x \iff x < 1\)
using log-less-cancel-iff [of a 1 x] by simp

lemma log-le-zero-cancel-iff [simp]:
  \(1 < a \Rightarrow 0 < x \Rightarrow \log_a x \leq 0 \iff x \leq 1\)
using log-le-cancel-iff [of a x 1] by simp

lemma log-less-zero-cancel-iff [simp]:
  \(1 < a \Rightarrow 0 < x \Rightarrow \log_a x < 0 \iff x < 1\)
using log-less-cancel-iff [of a x a] by simp

lemma log-le-one-cancel-iff [simp]:
  \(1 < a \Rightarrow 0 < x \Rightarrow \log_a x \leq 1 \iff x \leq a\)
using log-le-cancel-iff [of a x a] by simp

lemma powr-realpow: \(0 < x \Rightarrow x \powr (\real n) = x \powr\)
apply (induct n)
apply simp
apply (subgoal_tac real(Suc n) = real n + 1)
apply (erule ss subst)
apply (subst powr-add, simp, simp)
done

lemma powr-realpow-numeral: \(0 < x \Rightarrow x \powr (\numeral n :: \real) = x \powr\)
  (numeral n)
unfolding real-of-nat-numeral [symmetric] by (rule powr-realpow)

lemma powr2-sqrt [simp]: \(0 < x \Rightarrow \sqrt{x} \powr 2 = x\)
THEORY “Transcendental”

by(simp add: powr-realpow-numeral)

lemma powr-realpow2: \(0 \leq x \Rightarrow 0 < n \Rightarrow x^n = (\text{if } x = 0 \text{ then } 0 \text{ else } x \text{ powr } (\text{real } n))\)
apply (case-tac x = 0, simp, simp)
apply (rule powr-realpow [THEN sym], simp)
done

lemma powr-int:
assumes \(x > 0\)
shows \(x \text{ powr } i = (\text{if } i \geq 0 \text{ then } x \text{ } ^{\text{nat } i} \text{ else } 1 / x \text{ } ^{\text{nat } (-i)})\)
proof (cases i < 0)
case True
have r: \(x \text{ powr } i = 1 / x \text{ powr } (-i)\) by (simp add: powr-minus field-simps)
show \(?thesis\) using ⟨i < 0⟩⟨x > 0⟩ by (simp add: r field-simps powr-realpow [symmetric])
next
case False
then show \(?thesis\) by (simp add: assms powr-realpow [symmetric])
qed

lemma powr-one: \(0 < x \Rightarrow x \text{ powr } 1 = x\)
using powr-realpow [of x 1] by simp

lemma powr-numeral: \(0 < x \Rightarrow x \text{ powr numeral } n = x^{\text{numeral } n}\)
by (fact powr-realpow-numeral)

lemma powr-neg-one: \(0 < x \Rightarrow x \text{ powr } (-1) = 1 / x\)
using powr-int [of x - 1] by simp

lemma powr-neg-numeral: \(0 < x \Rightarrow x \text{ powr } (-\text{numeral } n) = 1 / x^{\text{numeral } n}\)
using powr-int [of x - numeral n] by simp

lemma root-powr-inverse: \(0 < n \Rightarrow 0 < x \Rightarrow \text{root } n x = x \text{ powr } (1/n)\)
by (rule real-root-pos-unique) (auto simp: powr-realpow[symmetric] powr-powr)

lemma ln-powr: \(\ln (x \text{ powr } y) = y \ast \ln x\)
by (simp add: powr-def)

lemma ln-root: \([n > 0; b > 0] \Rightarrow \ln (\text{root } n b) = \ln b / n\)
by(simp add: root-powr-inverse ln-powr)

lemma ln-sqrt: \(0 < x \Rightarrow \ln (\sqrt{x}) = \ln x / 2\)
by (simp add: ln-powr powr-numeral ln-powr[symmetric] mult.commute)

lemma log-root: \([n > 0; a > 0] \Rightarrow \log_b (\text{root } n a) = \log b a / n\)
by(simp add: log-def ln-root)

lemma log-powr: \(\log_b (x \text{ powr } y) = y \ast \log b x\)
by (simp add: log-def ln-powr)
lemma log-nat-power: \(0 < x \implies \log b (x ^ n) = \text{real } n \cdot \log b x\)
by (simp add: log-powr powr-realpow [symmetric])

lemma log-base-change: \(0 < a \implies a \neq 1 \implies \log b x = \frac{\log a x}{\log a b}\)
by (simp add: log-def)

lemma log-base-pow: \(0 < a \implies \log (a ^ n) x = \frac{\log a x}{n}\)
by (simp add: log-def ln-realpow)

lemma log-base-powr: \(\log (a ^ b) x = \frac{\log a x}{b}\)
by (simp add: log-def ln-powr)

lemma log-base-root: \(\lceil n > 0; b > 0 \rceil \implies \log (\text{root } n b) x = n \cdot \frac{\log b x}{x}\)
by (simp add: log-def ln-root)

lemma ln-bound: \(1 \leq x \implies \ln x \leq x\)
apply (subgoal-tac ln (1 + (x - 1)) \(\leq x - 1\))
apply simp
apply (rule ln-add-one-self-le-self, simp)
done

lemma powr-mono: \(a \leq b \implies 1 \leq x \implies x \cdot a \leq x \cdot b\)
apply (cases x = 1, simp)
apply (cases a = b, simp)
apply (rule order-less-imp-le)
apply (rule powr-less-mono, auto)
done

lemma ge-one-powr-ge-zero: \(1 \leq x \implies 0 \leq a \implies 1 \leq x \cdot a\)
apply (subst powr-zero-eq-one [THEN sym])
apply (rule powr-mono, assumption+)
done

lemma powr-less-mono2: \(0 < a \implies 0 < x \implies x < y \implies x \cdot a < y \cdot a\)
apply (unfold powr-def)
apply (rule exp-less-mono)
apply (rule mult-strict-left-mono)
apply (subst ln-less-cancel-iff, assumption)
apply (rule order-less-trans)
pref 2
apply assumption+
done

lemma powr-less-mono2-neg: \(a < 0 \implies 0 < x \implies x < y \implies y \cdot a < x \cdot a\)
apply (unfold powr-def)
apply (rule exp-less-mono)
apply \( \text{rule mult-strict-left-mono-neg} \)
apply \( \text{subst ln-less-cancel-iff} \)
apply assumption
apply \( \text{rule order-less-trans} \)
prefer 2
apply assumption

\[ \text{done} \]

lemma \( \text{powr-mono2} \):
\[ 0 < a =\rightarrow 0 < x =\rightarrow x < y =\rightarrow x \text{ powr } a < y \text{ powr } a \]
apply \( \text{case-tac } a = 0, \text{ simp} \)
apply \( \text{case-tac } x = y, \text{ simp} \)
apply \( \text{metis less-eq-real-def powr-less-mono2} \)

\[ \text{done} \]

lemma \( \text{powr-inj} \):
\[ 0 < a \Longrightarrow a \neq 1 \Longrightarrow a \text{ powr } x = a \text{ powr } y \longleftrightarrow x = y \]
unfolding \( \text{powr-def exp-inj-iff} \) by \( \text{simp} \)

lemma \( \text{ln-powr-bound} \):
\[ 1 < x =\rightarrow 0 < a =\rightarrow 0 < x =\rightarrow \ln x < (x \text{ powr } a) / a \]
by \( \text{metis less-eq-real-def ln-less-self mult-imp-le-div-pos ln-powr mult.commute order.strict-trans2 powr-gt-zero zero-less-one} \)

lemma \( \text{ln-powr-bound2} \):
assumes \( 1 < x \text{ and } 0 < a \)
shows \( (\ln x) \text{ powr } a < (a \text{ powr } a) \ast x \)
proof
from assms have \( \ln x < (x \text{ powr } (1 / a)) / (1 / a) \)
by \( \text{metis less-eq-real-def ln-powr-bound zero-less-divide-1-iff} \)
also have \( ... = a \ast (x \text{ powr } (1 / a)) \)
by \( \text{simp} \)
finally have \( (\ln x) \text{ powr } a < (a \ast (x \text{ powr } (1 / a))) \text{ powr } a \)
by \( \text{metis assms less-imp-le ln-gt-zero powr-mono2} \)
also have \( ... = (a \text{ powr } a) \ast ((x \text{ powr } (1 / a)) \text{ powr } a) \)
by \( \text{metis assms(2) powr-mult powr-gt-zero} \)
also have \( (x \text{ powr } (1 / a)) \text{ powr } a = x \text{ powr } ((1 / a) \ast a) \)
by \( \text{rule powr-powr} \)
also have \( ... = x \text{ using assms} \)
by \( \text{auto} \)
finally show \( ?\text{thesis} . \)
qd

lemma \( \text{tendsto-powr} \) [tendsto-intros]:
\[ \[ f \longrightarrow a \] F ; \[ g \longrightarrow b \] F; \( a \neq 0 \] \( \Longrightarrow \) \( (\lambda x. f x \text{ powr } g x) \longrightarrow a \text{ powr } b) \]
unfolding \( \text{powr-def} \) by \( \text{intro tendsto-intros} \)

lemma \( \text{continuous-powr} \):
assumes \( \text{continuous } F f \)
and \( \text{continuous } F g \)
and \( f (\text{Lim } F (\lambda x. x)) \neq 0 \)
shows continuous \( F (\lambda x. (f x) \text{ powr } (g x)) \)
using assms unfolding continuous-def by (rule tendsto-powr)

lemma \text{continuous-at-within-powr}[\text{continuous-intros}]:
assumes continuous (at a within s) \( f \)
\quad and continuous (at a within s) \( g \)
\quad and \( f a \neq 0 \)
shows continuous (at a within s) \( (\lambda x. (f x) \text{ powr } (g x)) \)
using assms unfolding continuous-within by (rule tendsto-powr)

lemma \text{isCont-powr}[\text{continuous-intros, simp}]:
assumes isCont \( f a \) isCont \( g a \) \( f a \neq 0 \)
shows isCont \( (\lambda x. (f x) \text{ powr } (g x)) a \)
using assms unfolding continuous-at by (rule tendsto-powr)

lemma \text{continuous-on-powr}[\text{continuous-intros}]:
assumes continuous-on s \( f \) continuous-on s \( g \) \text{ and } \forall x \in s. \ f x \neq 0
shows continuous-on s \( (\lambda x. (f x) \text{ powr } (g x)) \)
using assms unfolding continuous-on-def by (fast intro: tendsto-powr)

lemma \text{tendsto-zero-powrI}:
assumes eventually \( (\lambda x. \ 0 < f x) \ F \) and \( (\ --\ --\ 0) \ F \)
\quad and \( 0 < d \)
shows \((\lambda x. f x \text{ powr } d) \ --\ --\ 0) \ F \)
proof (rule tendstoI)
fix e :: real assume \( 0 < e \)
def Z \equiv e \text{ powr } (1 / d)
with \( \langle 0 < e \rangle \) have \( 0 < Z \) by simp
with assms have eventually \( (\lambda x. \ 0 < f x \land \dist f x 0 < Z) \ F \)
\quad by (intro eventually-conj tendstoD)
moreover
from assms have \( \langle x. \ 0 < x \land \dist x 0 < Z \implies x \text{ powr } d < Z \text{ powr } d \) \( \langle 0 < e \rangle \)
\quad by (intro powr-less-mono2) (auto simp: dist-real-def)
with assms \( \langle 0 < e \rangle \) have \( \langle x. \ 0 < x \land \dist x 0 < Z \implies \dist x \text{ powr } d 0 < e \)
\quad unfolding dist-real-def Z-def by (auto simp: powr-powr)
ultimately
show eventually \( (\lambda x. \dist (f x \text{ powr } d) 0 < e) \ F \) by (rule eventually-elim1)
qed

lemma \text{tendsto-neg-powr}:
assumes \( s < 0 \)
\quad and \( \lim x F. \ f x :> \text{at-top} \)
shows \((\langle \lambda x. f x \text{ powr } s \rangle \ --\ --\ 0) \ F \)
proof (rule tendstoI)
fix e :: real assume \( 0 < e \)
def Z \equiv e \text{ powr } (1 / s)
from assms have eventually \( (\lambda x. Z < f x) \ F \)
by (simp add: filterlim-at-top-dense)
moreover
from assms have \( \forall x. Z < x \Rightarrow x \pow s < Z \pow s \)
  by (auto simp: Z-def intro!: powr-less-mono2-neg)
with assms \( 0 < e \) have \( \forall x. Z < x \Rightarrow \dist (x \pow s) 0 < e \)
  by (simp add: powr-powr Z-def dist-real-def)
ultimately
show eventually \( (x \pow s) 0 < e \) \( F \) by (rule eventually-elim1)
qed

lemma tendsto-exp-limit-at-right:
fixes \( x :: \text{real} \)
shows \( ((\lambda y. (1 + x \ast y) \pow (1 / y)) \longrightarrow \exp x) \) (at-right 0)
proof cases
  assume \( x \neq 0 \)
  have \( ((\lambda y. \ln (1 + x \ast y)) :: \text{real}) \has-derivative I \ast x \) (at 0)
    by (auto intro!: derivative-eq-intros)
  then have \( ((\lambda y. \ln (1 + x \ast y) / y) \longrightarrow x) \) (at 0)
    by (auto simp: has-field-derivative-def field-has-derivative-at)
  then have \( *: ((\lambda y. \exp (\ln (1 + x \ast y) / y)) \longrightarrow \exp x) \) (at 0)
    by (rule tendsto-intros)
  then show \thesis
  proof (rule filterlim-mono-eventually)
    show eventually \( (\lambda xa. \exp (\ln (1 + x \ast xa) / xa) = (1 + x \ast xa) \pow (1 / xa)) \) (at-right 0)
      unfolding eventually-at-right[OF zero-less-one]
      using \( x \neq 0 \) by (intro exI[OF - 1 / |x|]) (auto simp: field-simps powr-def)
    qed (simp add: tendsto-const)
  qed (simp add: at-eq-sup-left-right)

lemma tendsto-exp-limit-at-top:
fixes \( x :: \text{real} \)
shows \( ((\lambda y. (1 + x / y) \pow y) \longrightarrow \exp x) \) at-top
apply (subst filterlim-at-top-to-right)
apply (simp add: inverse-eq-divide)
apply (rule tendsto-exp-limit-at-right)
done

lemma tendsto-exp-limit-sequentially:
fixes \( x :: \text{real} \)
shows \( (\lambda n. (1 + x / n) ^ n) \longrightarrow \exp x \)
proof (rule filterlim-mono-eventually)
  from reals-Archimedean2 [of abs x] obtain \( n :: \text{nat} \) where \*: \( \text{real } n > abs x \) ..
hence eventually \( (\lambda n :: \text{nat}. 0 < 1 + x / real n) \) at-top
  apply (intro eventually-sequentially1 [of n])
apply (case_tac \( x \geq 0 \))
apply (rule add-pos-nonneg, auto intro: divide-nonneg-nonneg)
apply (subgoal-tac x / real xa > -1)
apply (auto simp add: field-simps)
done
then show eventually (λn. (1 + x / n) powr n = (1 + x / n) ^ n) at-top
by (rule eventually-elim) (erule powr-realpow)
show (λn. (1 + x / real n) powr real n) ----> exp x
by (rule filterlim-compose [OF tendsto-exp-limit-at-top filterlim-real-sequentially])
qed auto

105.7 Sine and Cosine

definition sin-coeff :: nat ⇒ real where
  sin-coeff = (λn. if even n then 0 else -1 ^ ((n - Suc 0) div 2) / real (fact n))

definition cos-coeff :: nat ⇒ real where
  cos-coeff = (λn. if even n then (-1 ^ (n div 2)) / real (fact n) else 0)

definition sin :: real ⇒ real where
  sin = (λx. ∑n. sin-coeff n * x ^ n)

definition cos :: real ⇒ real where
  cos = (λx. ∑n. cos-coeff n * x ^ n)

lemma sin-coeff-0 [simp]: sin-coeff 0 = 0
unfolding sin-coeff-def by simp

lemma cos-coeff-0 [simp]: cos-coeff 0 = 1
unfolding cos-coeff-def by simp

lemma sin-coeff-Suc: sin-coeff (Suc n) = cos-coeff n / real (Suc n)
unfolding cos-coeff-def sin-coeff-def
by (simp del: mult-Suc)

lemma cos-coeff-Suc: cos-coeff (Suc n) = - sin-coeff n / real (Suc n)
unfolding cos-coeff-def sin-coeff-def
by (simp del: mult-Suc, auto simp add: odd-Suc-mult-two-ex)

lemma summable-sin: summable (λn. sin-coeff n * x ^ n)
unfolding sin-coeff-def
apply (rule summable-comparison-test [OF - summable-exp [where x=|x|] ])
apply (auto simp add: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
done

lemma summable-cos: summable (λn. cos-coeff n * x ^ n)
unfolding cos-coeff-def
apply (rule summable-comparison-test [OF - summable-exp [where x=|x|] ])
apply (auto simp add: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
done
THEORY "Transcendental"

lemma sin-converges: (\lambda n. sin-coeff n * x ^ n) sums sin(x)
unfolding sin-def by (rule summable-sin [THEN summable-sums])

lemma cos-converges: (\lambda n. cos-coeff n * x ^ n) sums cos(x)
unfolding cos-def by (rule summable-cos [THEN summable-sums])

lemma diffs-sin-coeff: diffs sin-coeff = cos-coeff
by (simp add: diffs-def sin-coeff-Suc real-of-nat-def del: of-nat-Suc)

lemma diffs-cos-coeff: diffs cos-coeff = (\lambda n. - sin-coeff n)
by (simp add: diffs-def cos-coeff-Suc real-of-nat-def del: of-nat-Suc)

Now at last we can get the derivatives of exp, sin and cos

lemma DERIV-sin [simp]: DERIV sin x :-> cos(x)
unfolding sin-def cos-def
apply (rule DERIV-cong, rule termdiffs [where K=1 + |x|])
apply (simp-all add: diffs-sin-coeff diffs-cos-coeff
summable-minus summable-sin summable-cos)
done

declare DERIV-sin[THEN DERIV-chain2, derivative-intros]

lemma DERIV-cos [simp]: DERIV cos x :-> -sin(x)
unfolding cos-def sin-def
apply (rule DERIV-cong, rule termdiffs [where K=1 + |x|])
apply (simp-all add: diffs-sin-coeff diffs-cos-coeff diffs-minus
summable-minus summable-sin summable-cos suminf-minus)
done

declare DERIV-cos[THEN DERIV-chain2, derivative-intros]

lemma isCont-sin: isCont sin x
by (rule DERIV-sin [THEN DERIV-isCont])

lemma isCont-cos: isCont cos x
by (rule DERIV-cos [THEN DERIV-isCont])

lemma isCont-sin' [simp]: isCont f a :-> isCont (\lambda x. sin (f x)) a
by (rule isCont-o2 [OF isCont-sin])

lemma isCont-cos' [simp]: isCont f a :-> isCont (\lambda x. cos (f x)) a
by (rule isCont-o2 [OF isCont-cos])

lemma tendsto-sin [tendsto-intros]:
(f ---a> a) F --- (\lambda x. sin (f x)) ---a> sin a) F
by (rule isCont-tendsto-compose [OF isCont-sin])

lemma tendsto-cos [tendsto-intros]:
(f ---a> a) F --- (\lambda x. cos (f x)) ---a> cos a) F
by (rule isCont-tendsto-compose [OF isCont-cos])

lemma continuous-sin [continuous-intros]:
  continuous F f \implies continuous F (\lambda x. sin (f x))
  unfolding continuous-def by (rule tendsto-sin)

lemma continuous-on-sin [continuous-intros]:
  continuous-on s f \implies continuous-on s (\lambda x. sin (f x))
  unfolding continuous-on-def by (auto intro: tendsto-sin)

lemma continuous-cos [continuous-intros]:
  continuous F f \implies continuous F (\lambda x. cos (f x))
  unfolding continuous-def by (rule tendsto-cos)

lemma continuous-on-cos [continuous-intros]:
  continuous-on s f \implies continuous-on s (\lambda x. cos (f x))
  unfolding continuous-on-def by (auto intro: tendsto-cos)

105.8 Properties of Sine and Cosine

lemma sin-zero [simp]: sin 0 = 0
  unfolding sin-def sin-coeff-def by (simp add: powser-zero)

lemma cos-zero [simp]: cos 0 = 1
  unfolding cos-def cos-coeff-def by (simp add: powser-zero)

lemma sin-cos-squared-add [simp]: (sin x)^2 + (cos x)^2 = 1
  proof
  have \forall x. DERIV (\lambda x. (sin x)^2 + (cos x)^2) x :> 0
    by (auto intro!: derivative-eq-intros)
  hence (sin x)^2 + (cos x)^2 = (sin 0)^2 + (cos 0)^2
    by (rule DERIV-isconst-all)
  thus (sin x)^2 + (cos x)^2 = 1 by simp
  qed

lemma sin-cos-squared-add2 [simp]: (cos x)^2 + (sin x)^2 = 1
  by (subst add.commute, rule sin-cos-squared-add)

lemma sin-cos-squared-add3 [simp]: cos x * cos x + sin x * sin x = 1
  using sin-cos-squared-add2 [unfolded power2-eq-square].

lemma sin-squared-eq: (sin x)^2 = 1 - (cos x)^2
  unfolding eq-diff-eq by (rule sin-cos-squared-add)

lemma cos-squared-eq: (cos x)^2 = 1 - (sin x)^2
  unfolding eq-diff-eq by (rule sin-cos-squared-add2)

lemma abs-sin-le-one [simp]: |sin x| \leq 1
  by (rule power2-le-imp-le, simp-all add: sin-squared-eq)
lemma \( \sin \geq -1 \) [simp]: \(-1 \leq \sin x\)
using abs-sin-le-one [of \( x \)] unfolding abs-le-iff by simp

lemma \( \sin \leq 1 \) [simp]: \( \sin x \leq 1 \)
using abs-sin-le-one [of \( x \)] unfolding abs-le-iff by simp

lemma \( \abs{\cos} \leq 1 \) [simp]: \( |\cos x| \leq 1 \)
by (rule power2-le-imp-le, simp-all add: cos-squared-eq)

lemma \( \cos \geq -1 \) [simp]: \(-1 \leq \cos x\)
using abs-cos-le-one [of \( x \)] unfolding abs-le-iff by simp

lemma \( \cos \leq 1 \) [simp]: \( \cos x \leq 1 \)
using abs-cos-le-one [of \( x \)] unfolding abs-le-iff by simp

lemma DERIV-fun-pow:

\[
\text{DERIV } (\lambda x. \ (g \ x) ^ n) \ x \ : \ real \ n * (g \ x) ^ {(n - 1)} * m
\]
by (auto intro!: derivative-eq-intros simp add: real-of-nat-def)

lemma DERIV-fun-exp:

\[
\text{DERIV } (\exp(g \ x)) \ x \ : \ exp(g \ x) * m
\]
by (auto intro!: derivative-intros)

lemma DERIV-fun-sin:

\[
\text{DERIV } (\sin(g \ x)) \ x \ : \ \cos(g \ x) * m
\]
by (auto intro!: derivative-intros)

lemma DERIV-fun-cos:

\[
\text{DERIV } (\cos(g \ x)) \ x \ : \ - \sin(g \ x) * m
\]
by (auto intro!: derivative-eq-intros simp: real-of-nat-def)

lemma sin-cos-add-lemma:
\[
(sin \ (x + y) - (\sin x * \cos y + \cos x * \sin y))^2 + (\cos \ (x + y) - (\cos x * \cos y - \sin x * \sin y))^2 = 0
\]
is \(?f x = 0\)

proof –
have \( \forall x. \text{DERIV } (\lambda x. \ ?f x) \ x \ : \ 0 \)
by (auto intro!: derivative-eq-intros simp add: algebra-simps)
hence \( ?f x = ?f 0 \)
by (rule DERIV-isconst-all)
thus \(?thesis\) by simp
qed

lemma \( \sin \text{-add}\):
\[
\sin \ (x + y) = \sin x * \cos y + \cos x * \sin y
\]
using sin-cos-add-lemma unfolding realpow-two-sum-zero-iff by simp

lemma \( \cos \text{-add}\):
\[
\cos \ (x + y) = \cos x * \cos y - \sin x * \sin y
\]
using sin-cos-add-lemma unfolding realpow-two-sum-zero-iff by simp
lemma sin-cos-minus-lemma:
\[(\sin(-x) + \sin(x))^2 + (\cos(-x) - \cos(x))^2 = 0 \text{ (is } f x = 0 \text{)}\]
proof -
  have \(\forall x. \text{ DERIV } (\lambda x. f x) x :> 0\)
    by (auto intro!: derivative-eq-intros simp add: algebra-simps)
  hence \(f x = f 0\)
    by (rule DERIV-isconst-all)
  thus \(?thesis\) by simp
qed

lemma sin-minus [simp]: \(\sin(-x) = -\sin(x)\)
using sin-cos-minus-lemma [where \(x = x\)] by simp

lemma cos-minus [simp]: \(\cos(-x) = \cos(x)\)
using sin-cos-minus-lemma [where \(x = x\)] by simp

lemma sin-diff: \(\sin(x - y) = \sin x \cdot \cos y - \cos x \cdot \sin y\)
using sin-add [of \(x - y\)] by simp

lemma sin-diff2: \(\sin(x - y) = \cos y \cdot \sin x - \sin y \cdot \cos x\)
by (simp add: sin-diff mult.commute)

lemma cos-diff: \(\cos(x - y) = \cos x \cdot \cos y + \sin x \cdot \sin y\)
using cos-add [of \(x - y\)] by simp

lemma cos-diff2: \(\cos(x - y) = \cos y \cdot \cos x + \sin y \cdot \sin x\)
by (simp add: cos-diff mult.commute)

lemma sin-double [simp]: \(\sin(2 \cdot x) = 2 \cdot \sin x \cdot \cos x\)
using sin-add [where \(x=x\) and \(y=x\)] by simp

lemma cos-double: \(\cos(2\cdot x) = ((\cos x)^2) - ((\sin x)^2)\)
using cos-add [where \(x=x\) and \(y=x\)]
by (simp add: power2-eq-square)

lemma sin-x-le-x: assumes \(x: x \geq 0\) shows \(\sin x \leq x\)
proof -
  let \(\bar{f} = \lambda x. x - \sin x\)
  from \(x\) have \(\bar{f} x \geq 0\)
    apply (rule DERIV-nonneg-imp-nondecreasing)
    apply (intro allI impI exI[of - 1 - cos x for x])
    apply (auto intro!: derivative-eq-intros simp: field-simps)
  done
  thus \(\sin x \leq x\) by simp
qed

lemma sin-x-ge-neg-x: assumes \(x: x \geq 0\) shows \(\sin x \geq - x\)
proof -
THEORY "Transcendental" 1699

let \$f = \lambda x . x + \sin x$

from \$x\$ have \$f x \geq f 0\$
  apply (rule DERIV-nonneg-imp-nondecreasing)
  apply (intro allI impI exI \[ of - 1 + \cos x \for x \])
  apply (auto intro!: derivative-eq-intros simp: field-simps real-0-le-add-iff)
  done
thus \$\sin x \geq -x\$ by simp

qed

lemma abs-sin-x-le-abs-x: |\$\sin x\$| \leq |\$x\$|
  using sin-x-ge-neg-x \[ of x \]
  sin-x-le-x \[ of x \]
  sin-x-ge-neg-x \[ of -x \]
  sin-x-le-x \[ of -x \]
  by (auto simp: abs-real-def)

105.9 The Constant Pi

definition pi :: real
  where pi = 2 \ast (THE x. 0 \leq (x::real) \& x \leq 2 \& \cos x = 0)

Show that there’s a least positive \$x\$ with \$\cos x = 0\$; hence define pi.

lemma sin-paired:
  (\$\lambda n. -1 \^ n \div (real (fact (2 \ast n + 1))) \ast x \^ (2 \ast n + 1)\$) sums \$\sin x\$

proof
  have (\$\lambda n. \sum k = n \ast 2..<n \ast 2 + 2. \sin-coeff k \ast x \^ k\$) sums \$\sin x\$
    by (rule sin-converges \[ THEN sums-group \], simp)
  thus \$?thesis unfolding One-nat-def sin-coeff-def \$ by (simp add: ac-simps)

qed

lemma sin-gt-zero:
  assumes 0 < \$x\$ and \$x\$ < 2
  shows 0 < \$\sin x\$

proof
  let \$\forall f = \lambda n. \sum k = n\ast2..<n\ast2+2. -1 \^ k / (real (fact (2\ast k + 1))) \ast x \^ (2 \ast k + 1)\$
  have pos: \$\forall n. 0 < \?f n\$
    proof
      fix \$n\$ :: nat
      let \$\forall k2 = real (Suc (Suc (4 \ast n)))\$
      let \$\forall k3 = real (Suc (Suc (4 \ast n)))\$
      have \$x \ast x < \?k2 \ast \?k3\$
        using assms by (intro mult-strict-mono', simp-all)
      hence \$x \ast x \ast x \ast x \^ (n \ast 4) < \?k2 \ast \?k3 \ast x \ast x \^ (n \ast 4)\$
        by (intro mult-strict-right-mono zero-less-power \$0 < x\$)
      thus \$0 < \?f n\$
        by (simp del: mult-Suc,
          simp add: less-divide-eq field-simps del: mult-Suc)
    qed

have sums: \$?f\$ sums \$\sin x\$
  by (rule sin-paired \[ THEN sums-group \], simp)

show 0 < \$\sin x\$

qed
unfolding sums-unique \([OF \text{ sums}]\)
using sums-summable \([OF \text{ sums}]\) pos
by (rule suminf-pos)
qed

**lemma** cos-double-less-one: \(0 < x \implies x < 2 \implies \cos(2 \cdot x) < 1\)
using sin-gt-zero \([\text{where } x = x]\) by (auto simp add: cos-squared-eq cos-double)

**lemma** cos-paired: \((\lambda n. -1 ^ n / (\text{real} (2 \cdot n)) \cdot x ^ (2 \cdot n)) \text{ sums } \cos x\)
proof
  have \((\lambda n. \sum k = n \cdot 2..<n \cdot 2 + 2 \cdot \cos-coeff k \cdot x ^ k) \text{ sums } \cos x\)
    by (rule cos-converges \([\text{THEN sums-group}, \text{simp}]\))
  thus \(?thesis\) unfolding cos-coeff-def by (simp add: ac-simps)
qed

**lemmas** realpow-num-eq-if = power-eq-if

**lemma** sumr-pos-lt-pair:
fixes \(f\) :: \(\text{nat} \Rightarrow \text{real}\)
shows \([\text{summable } f; \ \forall d. 0 < f (k + (\text{Suc} (\text{Suc} 0) \cdot d)) + f (k + ((\text{Suc} (\text{Suc} 0) \cdot d) + 1))]; \ \text{implies } \text{setsum } f \{..<k\} < \text{suminf } f\)
unfolding One-nat-def
apply (subst suminf-split-initial-segment \([\text{where } k=k]\))
apply assumption
apply simp
apply (drule-tac k=k in summable-ignore-initial-segment)
apply (drule-tac k=Suc (Suc 0) in sums-group \([\text{OF summable-sums}, \text{simp}]\))
apply simp
apply (frule sums-unique)
apply (drule sums-summable)
apply simp
apply (erule suminf-pos)
apply (simp add: ac-simps)
done

**lemma** cos-two-less-zero \([\text{simpl}]\):
\(\cos 2 < 0\)
proof
  note fact-Suc \([\text{simpl del}]\)
  from cos-paired
  have \((\lambda n. -(-1 ^ n / (\text{real} (2 \cdot n)) \cdot 2 ^ (2 \cdot n))) \text{ sums } - \cos 2\)
    by (rule sums-minus)
  then have \*: \((\lambda n. -(-1 ^ n \cdot 2 ^ (2 \cdot n) / (\text{real} (2 \cdot n)))) \text{ sums } - \cos 2\)
    by simp
  then have \**: \text{summable} \((\lambda n. -(-1 ^ n \cdot 2 ^ (2 \cdot n) / (\text{real} (2 \cdot n))))\)
    by (rule sums-summable)
  have \(0 < (\sum n<Suc (Suc (Suc 0))). -(-1 ^ n \cdot 2 ^ (2 \cdot n) / (\text{real} (2 \cdot n))))\)
THEORY “Transcendental”

by (simp add: fact-num-eq-if-nat realpow-num-eq-if)
moreover have \( \sum n<\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (4 \times d)))) \)
< \( \sum n. - (\neg 1 \times n \times 2 \times (2 \times n) / \text{real} (\text{fact} (2 \times n))) \)
proof -
\{ fix d 
\thinspace have \( 4 \times \text{real} (\text{fact} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (4 \times d)))))) \)
< \( \text{real} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (4 \times d)))))))))) \times d \)
by (simp only: real-of-nat-mult) (auto intro!: mult-strict-mono fact-less-mono-nat)
then have \( 4 \times \text{real} (\text{fact} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (4 \times d)))))))))))) \)
< \( \text{inverse} (\text{real} (\text{fact} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (\text{Suc} (4 \times d)))))))))))))) \)
by (simp add: inverse-eq-divide less-divide-eq)
\}

note *** = this
have [simp]: \( \{ x \times y :: \text{real}. \quad 0 < x \times y \leftrightarrow y < x \} \) by arith
from ** show ?thesis by (rule sumr-pos-lt-pair)
(simp add: divide-inverse mult.assoc [symmetric] ***)
qed ultimately have \( 0 < \sum n. - (\neg 1 \times n \times 2 \times (2 \times n) / \text{real} (\text{fact} (2 \times n))) \)
by (rule order-less-trans)
moreover from * have \( - \cos 2 = \sum n. - (\neg 1 \times n \times 2 \times (2 \times n) / \text{real} (\text{fact} (2 \times n))) \)
by (rule sums-unique)
ultimately have \( 0 < - \cos 2 \) by simp
then show ?thesis by simp
qed

lemmas cos-two-neq-zero [simp] = cos-two-less-zero [THEN less-imp-neq]
lemmas cos-two-le-zero [simp] = cos-two-less-zero [THEN order-less-imp-le]

lemma cos-is-zero: \( \exists ! x. \quad 0 \leq x \land x \leq 2 \land \cos x = 0 \)
proof (rule ex-ex1I)
show \( \exists x. \quad 0 \leq x \land x \leq 2 \land \cos x = 0 \)
by (rule IVT2, simp-all)
next
fix \( x \times y \)
assume \( x. \quad 0 \leq x \land x \leq 2 \land \cos x = 0 \)
assume \( y. \quad 0 \leq y \land y \leq 2 \land \cos y = 0 \)
have [simp]: \( \forall x. \quad \cos \text{ differentiable} \ (\text{at} \ x) \)
unfolding real-differentiable-def by (auto intro: DERIV-cos)
from \( x \times y \) show \( x = y \)
apply (cut-tac less-linear [of \( x \times y \)], auto)
apply (drule-tac \( f = \cos \) in Rolle)
apply (drule-tac \( 5 \) \( f = \cos \) in Rolle)
apply (auto dest!: DERIV-cos [THEN DERIV-unique])
apply (metis order-less-le-trans less-le sin-gt-zero)
apply (metis order-less-le-trans less-le sin-gt-zero)
done

qed

lemma pi-half: pi/2 = (THE x. 0 ≤ x & x ≤ 2 & cos x = 0)
  by (simp add: pi-def)

lemma cos-pi-half [simp]: cos (pi / 2) = 0
  by (simp add: pi-half cos-is-zero)

lemma pi-half-gt-zero [simp]: 0 < pi / 2
  apply (rule order-le-neq-trans)
  apply (simp add: pi-half cos-is-zero)
  apply (metis cos-pi-half cos-zero zero-neq-one)
done

lemmas pi-half-neq-zero [simp] = pi-half-gt-zero [THEN less-imp-neq, symmetric]
lemmas pi-half-ge-zero [simp] = pi-half-gt-zero [THEN order-less-imp-le]

lemma pi-half-less-two [simp]: pi / 2 < 2
  apply (rule order-le-neq-trans)
  apply (simp add: pi-half cos-is-zero)
  apply (metis cos-pi-half cos-two-neq-zero)
done

lemmas pi-half-neq-two [simp] = pi-half-less-two [THEN less-imp-neq]
lemmas pi-half-le-two [simp] = pi-half-less-two [THEN order-less-imp-le]

lemma pi-gt-zero [simp]: 0 < pi
  using pi-half-gt-zero by simp

lemma pi-ge-zero [simp]: 0 ≤ pi
  by (rule pi-gt-zero [THEN order-less-imp-le])

lemma pi-neq-zero [simp]: pi ≠ 0
  by (rule pi-gt-zero [THEN less-imp-neq, symmetric])

lemma pi-not-less-zero [simp]: ¬ pi < 0
  by (simp add: linorder-not-less)

lemma minus-pi-half-less-zero: -(pi/2) < 0
  by simp

lemma m2pi-less-pi: ¬ (2 * pi) < pi
  by simp

lemma sin-pi-half [simp]: sin(pi/2) = 1
using sin-cos-squared-add2 [where $x = \pi/2$]
using sin-gt-zero [of $\pi$-half-gt-zero $\pi$-half-less-two]
by (simp add: power2-eq-1-iff)

lemma cos-pi [simp]: $\cos \pi = -1$
using cos-add [where $x = \pi/2$ and $y = \pi/2$] by simp

lemma sin-pi [simp]: $\sin \pi = 0$
using sin-add [where $x = \pi/2$ and $y = \pi/2$] by simp

lemma sin-cos-eq: $\sin x = \cos (\pi/2 - x)$
by (simp add: cos-diff)

lemma minus-sin-cos-eq: $-\sin x = \cos (x + \pi/2)$
by (simp add: cos-add)

lemma cos-sin-eq: $\cos x = \sin (\pi/2 - x)$
by (simp add: sin-diff)

lemma sin-periodic-pi [simp]: $\sin (x + \pi) = -\sin x$
by (simp add: sin-add)

lemma sin-periodic-pi2 [simp]: $\sin (\pi + x) = -\sin x$
by (simp add: sin-add)

lemma cos-periodic-pi [simp]: $\cos (x + \pi) = -\cos x$
by (simp add: cos-add)

lemma sin-periodic [simp]: $\sin (x + 2*\pi) = \sin x$
by (simp add: sin-add cos-double)

lemma cos-periodic [simp]: $\cos (x + 2*\pi) = \cos x$
by (simp add: cos-add cos-double)

lemma cos-npi [simp]: $\cos (\text{real } n * \pi) = -1^n$
by (induct n) (auto simp add: real-of-nat-Suc distrib-right)

lemma cos-npi2 [simp]: $\cos (\pi * \text{real } n) = -1^n$
by (metis cos-npi mult.commute)

lemma sin-npi [simp]: $\sin (\text{real } (n::nat) * \pi) = 0$
by (induct n) (auto simp add: real-of-nat-Suc distrib-right)

lemma sin-npi2 [simp]: $\sin (\pi * \text{real } (n::nat)) = 0$
by (simp add: mult.commute [of pi])

lemma cos-two-pi [simp]: $\cos (2 * \pi) = 1$
by (simp add: cos-double)
lemma sin-two-pi [simp]: \( \sin(2 \pi) = 0 \)
  by simp

lemma sin-gt-zero2: \[
\begin{align*}
| 0 < x; x < \pi/2 | & \implies 0 < \sin x \\
\end{align*}
\]
  by (metis sin-gt-zero order-less-trans pi-half-less-two)

lemma sin-less-zero:
  assumes \(-\pi/2 < x; x < 0\)
  shows \(\sin x < 0\)
proof
  have \(0 < \sin(-x)\)
    using assms
    by (simp only: sin-gt-zero2)
  thus ?thesis
    by simp
qed

lemma pi-less-4: \(\pi < 4\)
  using pi-half-less-two
  by auto

lemma cos-gt-zero:
  assumes \(0 < x; x < \pi/2\)
  shows \(\cos x > 0\)
proof
  apply (cut-tac pi-less-4)
  apply (cut-tac f = cos and a = 0 and b = x and y = 0 in IVT2-objl, safe, simp-all)
  apply (cut-tac cos-is-zero, safe)
  apply (rename_tac y z)
  apply (drule_tac x = y in spec)
  apply (drule_tac x = pi/2 in spec, simp)
  done

lemma cos-gt-zero-pi:
  assumes \(-\pi/2 < x; x < \pi/2\)
  shows \(\cos x > 0\)
proof
  apply (rule_tac x = x and y = 0 in linorder-cases)
  apply (metis cos-gt-zero cos-minus minus-less-iff neg-0-less-iff-less)
  apply (auto intro: cos-gt-zero)
  done

lemma cos-ge-zero:
  assumes \(-\pi/2 \leq x; x \leq \pi/2\)
  shows \(\cos x \geq 0\)
proof
  have \(\exists y \geq \pi. y < 2 \land y < 2 \pi\)
  proof (cases 2 < 2 * pi)
    case True
    with dense[OF \(\pi < 2\)]
    show ?thesis
      by auto
  next
    case False
    have \(\pi < 2 \pi\)
      by auto
from dense[OF this] and False show thesis by auto
qed
then obtain y where pi < y and y < 2 and y < 2 * pi by blast
hence 0 < sin y using sin-gt-zero by auto
moreover
  have sin y < 0 using sin-gt-zero-pi[of y - pi][pi < y] and (y < 2 * pi)
  sin-periodic-pi[of y - pi] by auto
  ultimately show False by auto
qed

lemma sin-ge-zero: [[ 0 ≤ x; x ≤ pi ]] ==> 0 ≤ sin x
  by (auto simp add: order-le-less sin-gt-zero-pi)

FIXME: This proof is almost identical to lemma cos-is-zero. It should be possible to factor out some of the common parts.

lemma cos-total: [[ -1 ≤ y; y ≤ 1 ]] ==> EX! x. 0 ≤ x & x ≤ pi & (cos x = y)
proof (rule ex-ex1I)
  assume y: -1 ≤ y y ≤ 1
  show ∃ x. 0 ≤ x & x ≤ pi & cos x = y
    by (rule IVT2, simp-all add: y)
next
fix a b
  assume a: 0 ≤ a & a ≤ pi & cos a = y
  assume b: 0 ≤ b & b ≤ pi & cos b = y
  have [simp]: ∀ x. cos differentiable (at x)
    unfolding real-differentiable-def by (auto intro: DERIV-cos)
  from a b show a = b
    apply (cut-tac less-linear [of a b], auto)
    apply (drule-tac f = cos in Rolle)
    apply (drule-tac [5] f = cos in Rolle)
    apply (auto dest!: DERIV-cos [THEN DERIV-unique])
    apply (metis order-less-le-trans less-le sin-gt-zero-pi)
    apply (metis order-less-le-trans less-le sin-gt-zero-pi)
  done
qed

lemma sin-total:
  [[ -1 ≤ y; y ≤ 1 ]] ==> EX! x. -(pi/2) ≤ x & x ≤ pi/2 & (sin x = y)
  apply (rule ccontr)
  apply (subgoal-tac ∀ x. (-(pi/2) ≤ x & x ≤ pi/2 & (sin x = y)) = (0 ≤ (x + pi/2) & (x + pi/2) ≤ pi & (cos (x + pi/2) = -y)))
  apply (erule contrapos-np)
  apply simp
  apply (cut-tac y=-y in cos-total, simp) apply simp
  apply (erule ex1E)
  apply (rule-tac a = x - (pi/2) in ex1I)
  apply (simp (no-asm) add: add.assoc)
  apply (rotate-tac 3)
apply (drule-tac \( x = xa + \pi/2 \) in spec, safe, simp-all add: cos-add)
done

lemma reals-Archimedean4:
| \( 0 < y; 0 \leq x \) | == > \( \exists n. \) real \( n \cdot y \leq x \) & \( x < \) real (Suc \( n \)) * \( y \)
apply (auto dest!: reals-Archimedean3)
apply (drule pi-gt-zero [THEN reals-Archimedean4], safe)
apply (subgoal-tac \( x < \) real (LEAST \( m::nat. x < \) real \( m \cdot y \)) * \( y \))
prefer 2 apply (erule LeastI)
apply (case-tac LEAST \( m::nat. x < \) real \( m \cdot y \), simp)
apply (rename-tac \( m \))
apply (subgoal-tac \( \sim x < \) real \( m \cdot y \))
prefer 2 apply (rule not-less-Least, simp, force)
done

lemma cos-zero-lemma:
| \( 0 \leq x; \) cos \( x = 0 \) | == > \( \exists n::nat. \) ~even \( n \) & \( x = \) real \( n \cdot (\pi/2) \)
apply (drule pi-gt-zero [THEN reals-Archimedean4], safe)
apply (subgoal-tac \( x = x - \) real \( n \cdot \pi \) & \( x - \) real \( n \cdot \pi \) \leq \( \pi \) & \( \cos (x - \) real \( n \cdot \pi) = 0 \) )
apply (auto simp add: algebra-simps real-of-nat-Suc)
prefer 2 apply (simp add: cos-diff)
apply (simp add: cos-diff)
apply (subgoal-tac EX! \( x. \) \( 0 \leq x \) & \( x \leq \pi \) & \( \cos x = 0 \))
apply (rule-tac [2] cos-total, safe)
apply (drule-tac \( x = x - \) real \( n \cdot \pi \) in spec)
apply (drule-tac \( x = \pi/2 \) in spec)
apply (simp add: cos-diff)
apply (rule-tac \( x = \) Suc \( (2 \cdot n) \) in exI)
apply (simp add: real-of-nat-Suc algebra-simps, auto)
done

lemma sin-zero-lemma:
| \( 0 \leq x; \) sin \( x = 0 \) | == > \( \exists n::nat. \) even \( n \) & \( x = \) real \( n \cdot (\pi/2) \)
apply (subgoal-tac \( \exists n::nat. \) ~even \( n \) & \( x + \pi/2 = \) real \( n \cdot (\pi/2) \) )
apply (clarify, rule-tac \( x = n - 1 \) in exI)
apply (force simp add: odd-Suc-mult-two-ex real-of-nat-Suc distrib-right)
apply (rule cos-zero-lemma)
apply (simp-all add: cos-add)
done

lemma cos-zero-iff:
(\( \cos x = 0 \)) =
((\( \exists n::nat. \) ~even \( n \) & \( x = \) real \( n \cdot (\pi/2) \)) | 
(\( \exists n::nat. \) ~even \( n \) & \( x = -(\) real \( n \cdot (\pi/2) \)))))
apply (rule iffI)
apply (cut-tac linorder-linear [of 0 x], safe)
apply (drule cos-zero-lemma, assumption+)
apply (cut-tac x=-x in cos-zero-lemma, simp, simp)
apply (force simp add: minus-equation-iff [of x])
apply (auto simp only: odd-Suc-mult-two-ex real-of-nat-Suc distrib-right)
apply (auto simp add: cos-diff cos-add)
done

lemma sin-zero-iff:
  (sin x = 0) =
    ((\exists n::nat. even n & (x = real n * (pi/2))) |
    (\exists n::nat. even n & (x = -(real n * (pi/2)))))
apply (rule iffI)
apply (cut-tac linorder-linear [of 0 x], safe)
apply (drule sin-zero-lemma, assumption+)
apply (force simp add: minus-equation-iff [of x])
apply (auto simp add: even-mult-two-ex)
done

lemma cos-monotone-0-pi:
  assumes 0 \leq y and y < x and x \leq pi
  shows cos x < cos y
proof -
  have - (x - y) < 0 using assms by auto
  from MVT2[OF (y < x) DERIV-cos[THEN impI, THEN allI]]
  obtain z where y < z and z < x and cos-diff: cos x - cos y = (x - y) * - sin z
    by auto
  hence 0 < z and z < pi using assms by auto
  hence 0 < sin z using sin-gt-zero-pi by auto
  hence cos x - cos y < 0
    unfolding cos-diff minus-mult-commute[symmetric]
    using (- (x - y) < 0) by (rule mult-pos-neg2)
  thus ?thesis by auto
qed

lemma cos-monotone-0-pi':
  assumes 0 \leq y and y \leq x and x \leq pi
  shows cos x \leq cos y
proof (cases y < x)
case True
  show ?thesis using cos-monotone-0-pi[OF 0 \leq y] True \( x \leq pi] by auto
next
case False
hence \( y = x \) using \((y \leq x)\) by auto
thus \(?thesis\) by auto
qed

lemma cos-monotone-minus-pi-0:
assumes \(-\pi \leq y\) and \(y < x\) and \(x \leq 0\)
shows \(\cos y < \cos x\)
proof
  have \(0 \leq -x\) and \(-x < -y\) and \(-y \leq \pi\)
  using assms by auto
  from cos-monotone-0-pi[OF this] show \(?thesis\)
  unfolding cos-minus .
qed

lemma cos-monotone-minus-pi-0':
assumes \(-\pi \leq y\) and \(y \leq x\) and \(x \leq 0\)
shows \(\cos y \leq \cos x\)
proof (cases \(y < x\))
  case True
  show \(?thesis\) using cos-monotone-minus-pi-0[OF \((-\pi \leq y)\ True (x \leq 0)\)]
  by auto
next
  case False
  hence \(y = x\) using \((y \leq x)\) by auto
  thus \(?thesis\) by auto
qed

lemma sin-monotone-2pi':
assumes \(-(\pi / 2) \leq y\) and \(y \leq x\) and \(x \leq \pi / 2\)
shows \(\sin y \leq \sin x\)
proof
  have \(0 \leq y + \pi / 2\) and \(y + \pi / 2 \leq x + \pi / 2\) and \(x + \pi / 2 \leq \pi\)
  using pi-ge-two and assms by auto
  from cos-monotone-0-pi'[OF this] show \(?thesis\)
  unfolding minus-sin-cos-eq[symmetric] by auto
qed

105.10  Tangent

definition tan :: real \Rightarrow real
  where tan = (\x. \sin x / \cos x)

lemma tan-zero [simp]: tan 0 = 0
  by (simp add: tan-def)

lemma tan-pi [simp]: tan pi = 0
  by (simp add: tan-def)

lemma tan-npi [simp]: tan (real (n::nat) * pi) = 0
by (simp add: tan-def)

lemma tan-minus [simp]: tan (−x) = − tan x
  by (simp add: tan-def)

lemma tan-periodic [simp]: tan (x + 2*π) = tan x
  by (simp add: tan-def)

lemma lemma-tan-add1:
  [| cos x ≠ 0; cos y ≠ 0; cos (x + y) ≠ 0 ||] ==> 1 − tan x * tan y = cos (x + y)/(cos x * cos y)
  by (simp add: tan-def cos-add field-simps)

lemma add-tan-eq:
  [| cos x ≠ 0; cos y ≠ 0 ||] ==> tan x + tan y = sin(x + y)/(cos x * cos y)
  by (simp add: tan-def sin-add field-simps)

lemma tan-add:
  [| cos x ≠ 0; cos y ≠ 0; cos (x + y) ≠ 0 ||]
  ==> tan (x + y) = (tan(x) + tan(y))/(1 − tan(x) * tan(y))
  by (simp add: add-tan-eq lemma-tan-add1, simp add: tan-def)

lemma tan-double:
  [| cos x ≠ 0; cos (2 * x) ≠ 0 ||]
  ==> tan (2 * x) = (2 * tan x) / (1 − (tan x)^2)
  using tan-add [of x x] by (simp add: power2-eq-square)

lemma tan-gt-zero: [| 0 < x; x < pi/2 ||] ==> 0 < tan x
  by (simp add: tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma tan-less-zero:
  assumes lb: − pi/2 < x and x < 0
  shows tan x < 0
  proof −
    have 0 < tan (−x) using assms by (simp only: tan-gt-zero)
    thus ?thesis by simp
  qed

lemma tan-half: tan x = sin (2 * x) / (cos (2 * x) + 1)
  unfolding tan-def sin-double cos-double sin-squared-eq
  by (simp add: power2-eq-square)

lemma DERIV-tan [simp]: cos x ≠ 0 ==> DERIV tan x := inverse ((cos x)^2)
  unfolding tan-def
  by (auto intro!: derivative-eq-intros, simp add: divide-inverse power2-eq-square)

lemma isCont-tan: cos x ≠ 0 ==> isCont tan x
  by (rule DERIV-tan [THEN DERIV-isCont])

lemma isCont-tan' [simp]:
THEORY "Transcendental"

\[ \text{isCont } f \; a; \; \cos (f \; a) \neq 0 \implies \text{isCont } (\lambda x. \tan \; (f \; x)) \; a \]
\text{by (rule isCont-o2 \; [OF - isCont-tan])}

\textbf{lemma tendsto-tan} : \text{tendsto-at} \implies \text{tendsto-compose} \; \[ \text{by (rule isCont-tendsto-compose \; [OF isCont-tan])} \]

\textbf{lemma continuous-tan}:
\[ \text{continuous } F \; x \implies \cos \; (f \; (\lim \; F \; (\lambda x. \tan \; (f \; x)))) \neq 0 \implies \text{continuous } F \; (\lambda x. \tan \; (f \; x)) \]
\text{unfolding continuous-def \; by (rule tendsto-tan)}

\textbf{lemma isCont-tan"} : \text{continuous-intros}:
\[ \text{continuous } (at \; x) \; f \implies \cos \; (f \; x) \neq 0 \implies \text{continuous } (at \; x) \; (\lambda x. \tan \; (f \; x)) \]
\text{unfolding continuous-at \; by (rule tendsto-tan)}

\textbf{lemma continuous-within-tan} : \text{continuous-intros}:
\[ \text{continuous } (at \; x \; within \; s) \; f \implies \cos \; (f \; x) \neq 0 \implies \text{continuous } (at \; x \; within) \; (\lambda x. \tan \; (f \; x)) \]
\text{unfolding continuous-within \; by (rule tendsto-tan)}

\textbf{lemma continuous-on-tan} : \text{continuous-intros}:
\[ \text{continuous-on } s \; f \implies \forall x \in s. \cos \; (f \; x) \neq 0 \implies \text{continuous-on } s \; (\lambda x. \tan \; (f \; x)) \]
\text{unfolding continuous-on-def \; by (auto intro: tendsto-tan)}

\textbf{lemma LIM-cos-div-sin} : \text{(\lambda x. \cos(x)/\sin(x)) \rightarrow pi/2 \rightarrow 0}
\text{by (rule LIM-cong-limit, (rule tendsto-intros)+, simp-all)}

\textbf{lemma lemma-tan-total} : \text{0 < y \rightarrow \exists x. 0 < x & x < pi/2 & y < tan x}
\text{apply (cut-tac LIM-cos-div-sin)}
\text{apply (simp only: LIM-eq)}
\text{apply (drule-tac x = inverse y in spec, safe, force)}
\text{apply (drule-tac ?d1.0 = s in pi-half-gt-zero [THEN [2] real-lbound-gt-zero], safe)}
\text{apply (rule-tac x = (pi/2) - e in exI)}
\text{apply (simp (no-asm-simp))}
\text{apply (drule-tac x = (pi/2) - e in spec)}
\text{apply (auto simp add: tan-def sin-diff cos-diff)}
\text{apply (rule inverse-less-iff-less [THEN iffD1])}
\text{apply (auto simp add: divide-inverse)}
\text{apply (rule mult-pos-pos)}
\text{apply (subgoal-tac [3] 0 < sin e & 0 < cos e)}
\text{apply (auto intro: cos-gt-zero sin-gt-zero2 simp add: mult.commute)}
\text{done}

\textbf{lemma tan-total-pos} : \text{0 < y \rightarrow \exists x. 0 < x & x < pi/2 & tan x = y}
\text{apply (frule order-le-imp-less-or-eq, safe)}
\text{prefer 2 apply force}
\text{apply (drule lemma-tan-total, safe)}
THEORY “Transcendental”

apply (cut-tac f = tan and a = 0 and b = x and y = y in IVT-objl)
apply (auto intro!: DERIV-tan [THEN DERIV-isCont])
apply (drule-tac y = xa in order-le-imp-less-or-eq)
apply (auto dest: cos-gt-zero)
done

lemma lemma-tan-total1: EX x. -(pi/2) < x & x < (pi/2) & tan x = y
apply (cut-tac linorder-linear [of y], safe)
apply (drule tan-total-pos)
apply (cut-tac [2] y=-y in tan-total-pos, safe)
apply (rule-tac [3] x = -x in exI)
apply (auto del: exI intro!: exI)
done

lemma tan-total: EX! x. -(pi/2) < x & x < (pi/2) & tan x = y
apply (cut-tac y = y in lemma-tan-total1, auto)
apply hypsubst-thin
apply (cut-tac x = xa and y = y in linorder-less-linear, auto)
apply (subgoal-tac [2] EX z. y < z & z < xa & DERIV tan z :> 0)
apply (subgoal-tac [2] xa < z & z < y & DERIV tan z :> 0)
apply (rule-tac [4] Rolle)
apply (rule-tac [2] Rolle)
apply (auto del: exI intro!: DERIV-tan DERIV-isCont exI
simp add: real-differentiable-def)

Now, simulate TRYALL

apply (rule-tac [!] DERIV-tan asm-rl)
apply (auto del!: DERIV-unique [OF - DERIV-tan]
simp add: cos-gt-zero-pi [THEN less-imp-neq, THEN not-sym])
done

lemma tan-monotone:
assumes -(pi / 2) < y and y < x and x < pi / 2
shows tan y < tan x
proof
  have ∀ x'. y ≤ x' ∧ x' ≤ x → DERIV tan x' :> inverse ((cos x')^2)
proof (rule allI, rule impI)
  fix x' :: real
  assume y ≤ x' ∧ x' ≤ x
  hence -(pi/2) < x' and x' < pi/2 using assms by auto
  from cos-gt-zero-pi[OF this]
  have cos x' ≠ 0 by auto
  thus DERIV tan x' :> inverse ((cos x')^2) by (rule DERIV-tan)
qed
from MVT2[OF (y < x) this]
obtain z where y < z and z < x
  and tan-diff: tan x - tan y = (x - y) * inverse ((cos z)^2) by auto
hence -(pi / 2) < z and z < pi / 2 using assms by auto
hence 0 < cos z using cos-gt-zero-pi by auto
hence inv-pos: \(0 < \text{inverse} \ (\cos z)^2\) by auto
have \(0 < x - y\) using \(y < x\) by auto
with inv-pos have \(0 < \tan x - \tan y\) unfolding tan-diff by auto
thus ?thesis by auto
qed

lemma tan-monotone':
assumes \(- (pi / 2) < y\)
and \(y < pi / 2\)
and \(-(pi / 2) < x\)
and \(x < pi / 2\)
shows \((y < x) = (\tan y < \tan x)\)
proof
assume \(y < x\)
thus \(\tan y < \tan x\)
using tan-monotone and \(- (pi / 2) < y; \text{and} \ x < pi / 2;\) by auto
next
assume \(\tan y < \tan x\)
show \(y < x\)
proof (rule ccontr)
assume \(\neg y < x\) hence \(x \leq y\) by auto
hence \(\tan x \leq \tan y\)
proof (cases \(x = y\))
case True thus ?thesis by auto
next
case False hence \(x < y\) using \(x \leq y\) by auto
from tan-monotone[OF \(- (pi/2) < x; \text{this} \ (y < pi / 2);]\] show ?thesis by auto
qed
thus False using \(\tan y < \tan x\) by auto
qed
qed

lemma tan-inverse: \(1 / (\tan y) = \tan (pi / 2 - y)\)
unfolding tan-def sin-cos-eq[of y] cos-sin-eq[of y] by auto

lemma tan-periodic-pi[simp]: \(\tan (x + pi) = \tan x\)
by (simp add: tan-def)

lemma tan-periodic-nat[simp]:
fixes \(n:: \text{nat}\)
shows \(\tan (x + \text{real} \ n * pi) = \tan x\)
proof (induct \(n\) arbitrary: \(x\))
case \(0\)
then show ?case by simp
next
case \(\text{Suc} n\)
have split-pi-off: \(x + \text{real} \ (\text{Suc} n) * pi = (x + \text{real} \ n * pi) + pi\)
    unfolding Suc-eq-plus1 real-of-nat-add real-of-one distrib-right by auto
show ?case unfolding split-pi-off using Suc by auto

qed

lemma tan-periodic-int\[simp\]: fixes i :: int shows \( \tan (x + \text{real } i \cdot \pi) = \tan x \)
proof (cases \( 0 \leq i \))
case True
  hence i-nat: real i = real (nat i) by auto
  show ?thesis unfolding i-nat by auto
next
case False
  hence i-nat: real i = - real (nat (-i)) by auto
  have \( \tan x = \tan (x + \text{real } i \cdot \pi - \text{real } i \cdot \pi) \) by auto
  also have \dots = \( \tan (x + \text{real } i \cdot \pi) \)
    unfolding i-nat mult-minus-left diff-minus-eq-add by (rule tan-periodic-nat)
  finally show ?thesis by auto
qed

lemma tan-periodic-n\[simp\]: \( \tan (x + \text{numeral } n \cdot \pi) = \tan x \)
  using tan-periodic-int\[of - numeral n\] unfolding real-numeral.

105.11 Inverse Trigonometric Functions

definition arcsin :: real => real
  where arcsin y = (THE x. -(pi/2) \leq x \& x \leq pi/2 \& \sin x = y)

definition arccos :: real => real
  where arccos y = (THE x. 0 \leq x \& x \leq pi \& \cos x = y)

definition arctan :: real => real
  where arctan y = (THE x. -(pi/2) < x \& x < pi/2 \& \tan x = y)

lemma arcsin:
  \(-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin y} \& \text{arcsin y} \leq pi/2 \& \sin(\text{arcsin y}) = y \)
  unfolding arcsin-def by (rule theI' [OF sin-total])

lemma arcsin-pi:
  \(-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin y} \& \text{arcsin y} \leq pi \& \sin(\text{arcsin y}) = y \)
  apply (drule (1) arcsin)
  apply (force intro: order-trans)
  done

lemma sin-arcsin [simp]: \(-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow \sin(\text{arcsin y}) = y \)
  by (blast dest: arcsin)

lemma arcsin-bounded: \(-1 \leq y \Longrightarrow y \leq 1 \Longrightarrow -(pi/2) \leq \text{arcsin y} \& \text{arcsin y} \leq pi/2 \)
THEORY "Transcendental"

by (blast dest: arcsin)

lemma arcsin-lbound: \(-1 \leq y \implies y \leq 1 \implies -(\pi/2) \leq \arcsin y\)
by (blast dest: arcsin)

lemma arcsin-ubound: \(-1 \leq y \implies y \leq 1 \implies \arcsin y \leq \pi/2\)
by (blast dest: arcsin)

lemma arcsin-lt-bounded:
[\[ | -1 < y; y < 1 | \] ==> -(\pi/2) < \arcsin y \& \arcsin y < \pi/2]
apply (frule order-less-imp-le)
apply (frule-tac y = y in order-less-imp-le)
apply (frule arcsin-bounded)
apply (safe, simp)
apply (drule-tac y = arcsin y in order-le-imp-less-or-eq)
apply (drule-tac [!] y = \pi/2 in order-le-imp-less-or-eq, safe)
apply (drule-tac [!] f = \sin in arg-cong, auto)
done

lemma arcsin-sin: [\[ | -(\pi/2) \leq x; x \leq \pi/2 | \] ==> \arcsin(\sin x) = x]
apply (unfold arcsin-def)
apply (rule the1-equality)
apply (rule sin-total, auto)
done

lemma arccos:
[\[ | -1 \leq y; y \leq 1 | \] ==> 0 \leq \arccos y \& \arccos y \leq \pi \& \cos(\arccos y) = y]
unfolding arccos-def by (rule theI' [OF cos-total])

lemma cos-arccos [simp]: [\[ -1 \leq y; y \leq 1 | \] ==> \cos(\arccos y) = y]
by (blast dest: arccos)

lemma arccos-bounded: [\[ -1 \leq y; y \leq 1 | \] ==> 0 \leq \arccos y \& \arccos y \leq \pi]
by (blast dest: arccos)

lemma arccos-bound: [\[ -1 \leq y; y \leq 1 | \] ==> 0 \leq \arccos y]
by (blast dest: arccos)

lemma arccos-ubound: [\[ -1 \leq y; y \leq 1 | \] ==> \arccos y \leq \pi]
by (blast dest: arccos)

lemma arccos-lt-bounded:
[\[ | -1 < y; y < 1 | \] ==> 0 < \arccos y \& \arccos y < \pi]
apply (frule order-less-imp-le)
apply (frule-tac y = y in order-less-imp-le)
apply (frule arccos-bounded, auto)
apply (drule-tac y = \arccos y in order-le-imp-less-or-eq)
apply (drule-tac [2] \( y = \pi \) in order-le-imp-less-or-eq, auto)
done

lemma arccos-cos: \( \|0 \leq x; x \leq \pi\| \implies \arccos(\cos x) = x \)
apply (simp add: arccos-def)
apply (auto intro!: the1-equality cos-total)
done

lemma arccos-cos2: \( \|x \leq 0; -\pi \leq x\| \implies \arccos(\cos x) = -x \)
apply (simp add: arccos-def)
apply (auto intro!: the1-equality cos-total)
done

lemma cos-arcsin: \( [-1 \leq x; x \leq 1] \implies \cos(\arcsin x) = \sqrt{1 - x^2} \)
apply (subgoal-tac \( x^2 \leq 1 \))
apply (rule power2-eq-imp-eq)
apply (simp add: cos-squared-eq)
apply (rule cos-ge-zero)
apply (erule (1) arcsin-lbound)
apply (erule (1) arcsin-abound)
apply simp
apply (subgoal-tac \( |x|^2 \leq 1^2, simp \))
apply (rule power-mono, simp, simp)
done

lemma sin-arcsin: \( [-1 \leq x; x \leq 1] \implies \sin(\arccos x) = \sqrt{1 - x^2} \)
apply (subgoal-tac \( x^2 \leq 1 \))
apply (rule power2-eq-imp-eq)
apply (simp add: sin-squared-eq)
apply (rule sin-ge-zero)
apply (erule (1) arccos-lbound)
apply (erule (1) arccos-abound)
apply simp
apply (subgoal-tac \( |x|^2 \leq 1^2, simp \))
apply (rule power-mono, simp, simp)
done

lemma arctan [simp]: \( -(\pi/2) < \arctan y \& \arctan y < \pi/2 \& \tan(\arctan y) = y \)
unfolding arctan-def by (rule the1' [OF tan-total])

lemma tan-arctan: \( \tan(\arctan y) = y \)
by auto

lemma arctan-bounded: \( -(\pi/2) < \arctan y \& \arctan y < \pi/2 \)
by (auto simp only: arctan)

lemma arctan-lbound: \( -(\pi/2) < \arctan y \)
THEORY "Transcendental"

by auto

lemma arctan-ubound: arctan y < pi/2
by (auto simp only: arctan)

lemma arctan-unique:
assumes -(pi/2) < x
and x < pi/2
and tan x = y
shows arctan y = x
using assms arctan[of y] tan-total[of y] by (fast elim: ex1E)

lemma arctan-tan: -(pi/2) < x =⇒ x < pi/2 =⇒ arctan (tan x) = x
by (rule arctan-unique) simp-all

lemma arctan-zero-zero [simp]: arctan 0 = 0
by (rule arctan-unique) simp-all

lemma arctan-minus: arctan (- x) = - arctan x
apply (rule arctan-unique)
apply (simp only: neg-less-iff-less arctan-ubound)
apply (metis minus-less-iff arctan-lbound)
apply simp
done

lemma cos-arctan-not-zero [simp]: cos (arctan x) ≠ 0
by (intro less-imp-neq [symmetric] cos-gt-zero-pi arctan-lbound arctan-ubound)

lemma cos-arctan: cos (arctan x) = 1 / sqrt (1 + x^2)
proof (rule power2-eq-imp-eq)
have 0 < 1 + x^2 by (simp add: add-pos-nonneg)
show 0 ≤ 1 / sqrt (1 + x^2) by simp
show 0 ≤ cos (arctan x)
by (intro less-imp-le cos-gt-zero-pi arctan-lbound arctan-ubound)
have (cos (arctan x))^2 * (1 + (tan (arctan x))^2) = 1
unfolding tan-def by (simp add: distrib-left power-divide)
thus (cos (arctan x))^2 = (1 / sqrt (1 + x^2))^2
using 0 < 1 + x^2 by (simp add: power-divide eq-divide-eq)
qed

lemma sin-arctan: sin (arctan x) = x / sqrt (1 + x^2)
using add-pos-nonneg [OF zero-less-one zero-le-power2 [of x]]
using tan-arctan [of x] unfolding tan-def cos-arctan
by (simp add: eq-divide-eq)

lemma tan-sec: cos x ≠ 0 =⇒ 1 + (tan x)^2 = (inverse (cos x))^2
apply (rule power-inverse [THEN subst])
apply (rule_tac c1 = (cos x)^2 in mult-right-cancel [THEN iffD1])
apply (auto dest: field-power-not-zero
  simp add: power-mult-distrib distrib-right power-divide tan-def
  mult.assoc power-inverse [symmetric])
done

lemma arctan-less-iff: arctan x < arctan y ⟷ x < y
  by (metis tan-monotone' arctan-lbound arctan-ubound tan-arctan)

lemma arctan-le-iff: arctan x ≤ arctan y ⟷ x ≤ y
  by (simp only: not-less [symmetric] arctan-less-iff)

lemma arctan-eq-iff: arctan x = arctan y ⟷ x = y
  by (simp only: eq-iff [where 'a=real] arctan-le-iff)

lemma zero-less-arctan-iff [simp]: 0 < arctan x ⟷ 0 < x
  using arctan-less-iff[of 0 x] by simp

lemma arctan-less-zero-iff [simp]: arctan x < 0 ⟷ x < 0
  using arctan-less-iff[of x 0] by simp

lemma zero-le-arctan-iff [simp]: 0 ≤ arctan x ⟷ 0 ≤ x
  using arctan-le-iff[of 0 x] by simp

lemma arctan-le-zero-iff [simp]: arctan x ≤ 0 ⟷ x ≤ 0
  using arctan-le-iff[of x 0] by simp

lemma arctan-eq-zero-iff [simp]: arctan x = 0 ⟷ x = 0
  using arctan-eq-iff[of x 0] by simp

lemma continuous-on-arcsin': continuous-on {−1 .. 1} arcsin
proof
  have continuous-on (sin {− pi / 2 .. pi / 2}) arcsin
    by (rule continuous-on-inv) (auto intro: continuous-intros simp: arcsin-sin)
  also have sin {− pi / 2 .. pi / 2} = {−1 .. 1}
  proof
    fix x :: real
    assume x ∈ {−1..1}
    then show x ∈ sin {− pi / 2 .. pi / 2}
      using arcsin-lbound arcsin-ubound
      by (intro image-eqI[where x=arcsin x]) auto
  qed simp
  finally show ?thesis .
qed

lemma continuous-on-arcsin [continuous-intros]:
  continuous-on s f ⟹ (∀x∈s. −1 ≤ f x ∧ f x ≤ 1) ⟹ continuous-on s (λx. arcsin (f x))
  using continuous-on-compose[of s f, OF - continuous-on-subset[OF continuous-on-arcsin']]
  by (auto simp: comp-def subset-eq)
lemma isCont-arcsin: \(-1 < x \Rightarrow x < 1 \Rightarrow \text{isCont arcsin } x\)
using continuous-on-arcsin'[THEN continuous-on-subset, of \{ -1 <..< 1 \}']
by (auto simp: continuous-on-eq-continuous-at subset-eq)

lemma continuous-on-arccosiatrics: \(\text{continuous-on } \{ -1 .. 1 \}\) arccos
proof
  have \(\text{continuous-on (cos ' \{ 0 .. pi \}) arccos}\)
    by (rule continuous-on-inv) (auto intro: continuous-intros simp: arccos-cos)
  also have \(\text{cos ' \{ 0 .. pi \}} = \{ -1 .. 1 \}\)
    proof safe
      fix \(x\) :: real
      assume \(x\) \(\in \{ -1 .. 1 \}\)
      then show \(x\) \(\in \text{cos ' \{ 0 .. pi \}}\)
        using arccos-lbound arccos-ubound
        by (intro image-eqI \where \(x = \text{arccos } x\)) auto
  qed
finally show \(?\text{thesis} .\).
qed

lemma continuous-on-arccos [continuous-intros]:
  \(\text{continuous-on } s f \Rightarrow (\forall x \in s. -1 \leq f x \land f x \leq 1) \Rightarrow \text{continuous-on } s (\lambda x. \text{arccos } (f x))\)
using continuous-on-compose[of \(s f\), OF - continuous-on-subset[of \(s\) \text{continuous-on-arccos}']]
by (auto simp: comp-def subset-eq)

lemma isCont-arccos: \(-1 < x \Rightarrow x < 1 \Rightarrow \text{isCont arccos } x\)
using continuous-on-arccos'[THEN continuous-on-subset, of \{ -1 <..< 1 \}']
by (auto simp: continuous-on-eq-continuous-at subset-eq)

lemma isCont-arctan: \(\text{isCont arctan } x\)
apply (rule arctan-lbound [of \(x\), THEN dense, THEN exE], clarify)
apply (rule arctan-ubound [of \(x\), THEN dense, THEN exE], clarify)
apply (subgoal-tac isCont-arctan (\(\text{tan } (\text{arctan } x)\)), simp)
apply (erule (1) isCont-inverse-function2 \where \(f = \text{tan}\))
apply (metis arctan-tan order-le-less-trans order-less-le-trans)
apply (metis cos-gt-zero-pi isCont-tan order-less-le-trans less-le)
done

lemma tendsto-arctan [tendsto-intros]: \((f \longrightarrow x) \ F \Rightarrow ((\lambda x. \text{arctan } (f x)) \longrightarrow \text{arctan } x) \ F\)
by (rule isCont-tendsto-compose [OF isCont-arctan])

lemma continuous-arctan [continuous-intros]: \(\text{continuous } F f \Rightarrow \text{continuous } F (\lambda x. \text{arctan } (f x))\)
unfolding continuous-def by (rule tendsto-arctan)

lemma continuous-on-arctan [continuous-intros]: \(\text{continuous-on } s f \Rightarrow \text{continuous-on } s (\lambda x. \text{arctan } (f x))\)
THEORY "Transcendental"

unfolding continuous-on-def by (auto intro: tendsto-arctan)

lemma DERIV-arcsin: 
\[-1 < x; x < 1 \implies \text{DERIV} \arcsin x :> \text{inverse} (\sqrt{1 - x^2})\]
apply (rule DERIV-inverse-function [where f=sin and a=-1 and b=1])
apply (simp add: cos-arcsin)
apply (subgoal-tac \(|x|^2 < 1^2, simp\))
apply (rule power-strict-mono, simp, simp, simp)
apply assumption
apply assumption
apply simp
apply (erule (1) isCont-arcsin)
done

lemma DERIV-arccos: 
\[-1 < x; x < 1 \implies \text{DERIV} \arccos x :> \text{inverse} (-\sqrt{1 - x^2})\]
apply (rule DERIV-inverse-function [where f=cos and a=-1 and b=1])
apply (simp add: sin-arccos)
apply (subgoal-tac \(|x|^2 < 1^2, simp\))
apply (rule power-strict-mono, simp, simp, simp)
apply assumption
apply assumption
apply simp
apply (erule (1) isCont-arccos)
done

lemma DERIV-arctan: \text{DERIV} \arctan x :> \text{inverse} (1 + x^2)
apply (rule DERIV-inverse-function [where f=tan and a=x - 1 and b=x + 1])
apply (rule DERIV-cong [OF DERIV-tan])
apply (rule cos-arctan-not-zero)
apply (simp add: power-inverse tan-sec [symmetric])
apply (subgoal-tac 0 < 1 + x^2, simp)
apply (simp add: add-pos-nonneg)
apply (simp, simp, simp, rule isCont-arctan)
done

declare
DERIV-arcsin[THEN DERIV-chain2, derivative-intros]
DERIV-arccos[THEN DERIV-chain2, derivative-intros]
DERIV-arctan[THEN DERIV-chain2, derivative-intros]

lemma filterlim-tan-at-right: filterlim tan at-bot (at-right (- pi/2))
by (rule filterlim-at-bot-at-right[where Q=\(\lambda x. -\pi/2 < x \land x < \pi/2\) and P=\(\lambda x. True\) and g=arctan])
(auto simp: le-less eventually-at dist-real-def simp del: less-divide-eq-numeral1 intro!: tan-monotone exI[of - pi/2])
lemma filterlim-tan-at-left: \( \limsup_{x \to \pi/2^-} \tan(x) = \lim_{x \to \pi/2^-} \tan(x) \)
by (rule filterlim-at-top-at-left \[ where \ Q = \lambda x. -\pi/2 < x \land x < \pi/2 \ and \ P = \lambda x. True \ and \ g = \arctan \]
(auto simp: le-less eventually-at-real-def simp: less-less-eq-numeral1 intro: tan-monotone exI \[ of \ -\pi/2 \])

lemma tendsto-arctan-at-top: \( \lim_{x \to \pi/2^+} \arctan(x) \)
proof (rule tendstoI)
fix \( e :: \text{real} \)
assume \( 0 < e \)
def \( y \equiv \pi/2 - \min(\pi/2, e) \)
then have \( y: 0 \leq y < \pi/2 \)
using \( \langle 0 < e \rangle \)
(auto)
show eventually \( (\lambda x). \text{dist}(\arctan(x), \pi/2) < e \)
at-top
proof (intro eventually-at-top-dense \[ THEN iffD2 \] exI allI impI)
fix \( x \)
assume \( \tan y < x \)
then have \( \arctan(\tan y) < \arctan(x) \)
by (simp add: arctan-less-iff)
with \( y \)
have \( y < \arctan(x) \)
by (subst (asm) arctan-tan simp-all)
with \( \arctan\)\[ of \ x, \text{arith} \] \( y: 0 < e \)
show \( \text{dist}(\arctan(x), \pi/2) < e \)
by (simp add: dist-real-def)
qed
qed

lemma tendsto-arctan-at-bot: \( \lim_{x \to -\pi/2^-} \arctan(x) = -\pi/2 \)
unfolding filterlim-at-bot-mirror arctan-minus
by (intro tendsto-minus tendsto-arctan-at-top)

105.12 More Theorems about Sin and Cos
lemma cos-45: \( \cos(\pi/4) = \sqrt{2}/2 \)
proof
let \( ?c = \cos(\pi/4) \and \ ?s = \sin(\pi/4) \)
have nonneg: \( 0 \leq ?c \)
by (simp add: cos-ge-zero)
have \( 0 = \cos(\pi/4 + \pi/4) \)
by simp
also have \( \cos(\pi/4 + \pi/4) = ?c^2 - ?s^2 \)
by (simp only: cos-add power2-eq-square)
also have \( \ldots = 2 * ?c^2 - 1 \)
by (simp add: sin-squared-eq)
finally have \( ?c^2 = (\sqrt{2}/2)^2 \)
by (simp add: power-divide)
thus \( ?\text{thesis} \)
using nonneg by (rule power2-eq-imp-eq) simp qed

lemma cos-30: cos (pi / 6) = sqrt 3 / 2
proof –
  let ?c = cos (pi / 6) and ?s = sin (pi / 6)
  have pos-c: 0 < ?c
    by (rule cos-gt-zero, simp, simp)
  have 0 = cos (pi / 6 + pi / 6 + pi / 6)
    by simp
  also have ... = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s
    by (simp only: cos-add sin-add)
  also have ... = ?c * (?c^2 - 3 * ?s^2)
    by (simp add: algebra-simps power2-eq-square)
  finally have ?c^2 = (sqrt 3 / 2)^2
    using pos-c by (simp add: sin-squared-eq power-divide)
  thus ?thesis
    using pos-c [THEN order-less-imp-le]
    by (rule power2-eq-imp-eq) simp qed

lemma sin-45: sin (pi / 4) = sqrt 2 / 2
by (simp add: sin-cos-eq cos-45)

lemma sin-60: sin (pi / 3) = sqrt 3 / 2
by (simp add: sin-cos-eq cos-30)

lemma cos-60: cos (pi / 3) = 1 / 2
apply (rule power2-eq-imp-eq)
apply (simp add: algebra-simps power2-eq-square)
apply (rule cos-ge-zero, rule order-trans [where y=0], simp-all)
done

lemma sin-30: sin (pi / 6) = 1 / 2
by (simp add: sin-cos-eq cos-60)

lemma tan-30: tan (pi / 6) = 1 / sqrt 3
unfolding tan-def by (simp add: sin-30 cos-30)

lemma tan-45: tan (pi / 4) = 1
unfolding tan-def by (simp add: sin-45 cos-45)

lemma tan-60: tan (pi / 3) = sqrt 3
unfolding tan-def by (simp add: sin-60 cos-60)

lemma sin-cos-npi [simp]: sin (real (Suc (2 * n)) * pi / 2) = (-1)^n
proof –
  have sin ((real n + 1/2) * pi) = cos (real n * pi)
    by (auto simp add: algebra-simps sin-add)
thus \textit{thesis} by (simp add: real-of-nat-Suc distrib-right add-divide-distrib mult.commute [of pi])
qeda

\textbf{lemma} \textit{cos-2npi} [simp]: \textit{cos} \((2 * \text{real} \ (n::\text{nat}) * \pi) = 1\)
by (simp add: cos-durable mult.assoc power-add [symmetric] numeral-2-eq-2)

\textbf{lemma} \textit{cos-3over2-pi} [simp]: \textit{cos} \((3 / 2 * \pi) = 0\)
apply (subgoal-tac cos \((pi + pi/2) = 0\,\,\text{simp}\))
apply (subst cos-add, simp)
done

\textbf{lemma} \textit{sin-2npi} [simp]: \textit{sin} \((2 * \text{real} \ (n::\text{nat}) * \pi) = 0\)
by (auto simp add: mult.assoc)

\textbf{lemma} \textit{sin-3over2-pi} [simp]: \textit{sin} \((3/2 * \pi) = -1\)
apply (subgoal-tac sin \((pi + pi/2) = -1\,\,\text{simp}\))
apply (subst sin-add, simp)
done

\textbf{lemma} \textit{cos-pi-eq-zero} [simp]: \textit{cos} \((\pi * \text{real} \ (\text{Suc} (2 * m)) / 2) = 0\)
apply (simp only: cos-add sin-add real-of-nat-Suc distrib-right distrib-left add-divide-distrib)
apply auto
done

\textbf{lemma} \textit{DERIV-cos-add} [simp]: \textit{DERIV} \((\lambda x. \text{cos} \ (x + k))\) \(\lim xa := - \text{sin} \ (xa + k)\)
by (auto intro!: derivative-eq-intros)

\textbf{lemma} \textit{sin-zero-abs-cos-one}: \textit{sin} \(x = 0 \Longrightarrow |\text{cos} \ x| = 1\)
by (auto simp add: sin-zero-iff even-mult-two-ex)

\textbf{lemma} \textit{cos-one-sin-zero}: \textit{cos} \(x = 1 \Longrightarrow \text{sin} \ x = 0\)
using sin-cos-squared-add3 [where \(x = x\)] by auto

\textbf{105.13} \textbf{Machins formula}

\textbf{lemma} \textit{arctan-one}: \textit{arctan} \(1 = \pi / 4\)
by (rule arctan-unique, simp-all add: tan-45 m2pi-less-pi)

\textbf{lemma} \textit{tan-total-pi4}:
assumes \(|x| < 1\)
sows \(\exists z. - (\pi / 4) < z \land z < \pi / 4 \land \tan z = x\)
proof
show \(- (\pi / 4) < \text{arctan} \ x \land \text{arctan} \ x < \pi / 4 \land \tan (\text{arctan} \ x) = x\)
unfolding arctan-one [symmetric] arctan-minus [symmetric]
unfolding arctan-less-iff using assms by auto
qeda
lemma arctan-add:
assumes \(|x| \leq 1 \text{ and } |y| < 1\)
shows \(\arctan x + \arctan y = \arctan \left(\frac{x + y}{1 - x \cdot y}\right)\)

proof (rule arctan-unique [symmetric])
have \((- \pi / 4) \leq \arctan x \text{ and } (- \pi / 4) < \arctan y\)
  unfolding arctan-one [symmetric] arctan-minus [symmetric]
  unfolding arctan-le-iff arctan-less-iff using assms by auto
from add-le-less-mono [OF this]
show \(1: - (\pi / 2) < \arctan x + \arctan y\) by simp
have \(\arctan x \leq \pi / 4 \text{ and } \arctan y < \pi / 4\)
  unfolding arctan-one [symmetric]
  unfolding arctan-le-iff arctan-less-iff using assms by auto
from add-le-less-mono [OF this]
show 2: \(\arctan x + \arctan y < \pi / 2\) by simp
show \(\tan (\arctan x + \arctan y) = (x + y) / (1 - x \cdot y)\)
  using cos-gt-zero-pi [OF 1 2] by (simp add: tan-add)
qed

theorem machin: \(\pi / 4 = 4 \cdot \arctan (1/5) - \arctan (1/239)\)
proof –
  have \(|1/5| < (1 :: real)\) by auto
  from arctan-add[OF less-imp-le[OF this] this]
  have 2 * \(\arctan (1/5) = \arctan (5/12)\) by auto
  moreover
  have \(|5/12| < (1 :: real)\) by auto
  from arctan-add[OF less-imp-le[OF this] this]
  have 2 * \(\arctan (5/12) = \arctan (120/119)\) by auto
  moreover
  have \(|1| \leq (1 :: real) \text{ and } |1/239| < (1 :: real)\) by auto
  from arctan-add[OF this]
  have \(\arctan 1 + \arctan (1/239) = \arctan (120/119)\) by auto
  ultimately have \(\arctan 1 + \arctan (1/239) = 4 \cdot \arctan (1/5)\) by auto
  thus \(?thesis\) unfolding arctan-one by algebra
qed

105.14 Introducing the arcus tangens power series

lemma monoseq-arctan-series:
fixes \(x :: real\)
assumes \(|x| \leq 1\)
shows \(\text{monoseq} \ (\lambda n. 1 / \text{ real} (n*2+1) \cdot x^{(n*2+1)}) \ (\text{is monoseq } ?a)\)
proof (cases \(x = 0\))
case \text{True}
  thus \(?thesis\) unfolding monoseq-def One-nat-def by auto
next
case \text{False}
  have \(\text{norm } x \leq 1 \text{ and } x \leq 1 \text{ and } -1 \leq x\) using assms by auto
  show \(\text{monoseq } ?a\)
proof –
{ 
fix n
fix x :: real
assume 0 ≤ x and x ≤ 1
have \( \frac{1}{\operatorname{real}(\text{Suc}(\text{Suc}(n*2)))} * x \leq \text{Suc}(\text{Suc}(n*2)) \leq \frac{1}{\operatorname{real}(\text{Suc}(n*2))} * x \) 
proof (rule mult-mono)
show \( \frac{1}{\operatorname{real}(\text{Suc}(\text{Suc}(n*2)))} \leq \frac{1}{\operatorname{real}(\text{Suc}(n*2))} \) 
by (rule frac-le simp-all)
show 0 ≤ \( \frac{1}{\operatorname{real}(\text{Suc}(n*2))} \) 
by auto
show \( x \leq x \) 
by (rule power-decreasing simp add: ⟨0 ≤ x⟩ ⟨x ≤ 1⟩)
show 0 ≤ \( x \) 
by (rule zero-le-power simp add: ⟨0 ≤ x⟩)
qed
}

note mono = this

show ?thesis
proof (cases 0 ≤ x)
case True
from mono [OF this ⟨x ≤ 1⟩, THEN allI]
show ?thesis unfolding Suc-eq-plus1 [symmetric]
by (rule mono-SucI2)
next
case False
hence 0 ≤ −x and −x ≤ 1 using (−1 ≤ x) by auto
from mono[OF this]
have \( \lambda n. \frac{1}{\operatorname{real}(\text{Suc}(\text{Suc}(n*2)))} * x \leq \text{Suc}(\text{Suc}(n*2)) \geq \frac{1}{\operatorname{real}(\text{Suc}(n*2))} * x \) by auto
thus ?thesis unfolding Suc-eq-plus1 [symmetric] by (rule mono-SucI1[OF allI])
qed
qed

lemma zzeroseq-arctan-series:
fixes x :: real
assumes \( |x| \leq 1 \)
shows \( (\lambda n. \frac{1}{\operatorname{real}(n*2+1)} * x^{(n*2+1)}) \) ----> 0 (is ?a ----> 0)
proof (cases x = 0)
case True
thus ?thesis
unfolding One-nat-def by (auto simp add: tendsto-const)
next
case False
have norm x ≤ 1 and x ≤ 1 and −1 ≤ x using assms by auto
show ?a ----> 0
proof (cases \( |x| < 1 \))
case True
hence \( \text{norm } x < 1 \) by auto

from tendsto-mult[\{OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero|OF (norm x < 1), THEN LIMSEQ-Suc\}]
  have \((\lambda n. 1 / real (n + 1) * x ^ (n + 1)) \longrightarrow 0\)
    unfolding inverse-eq-divide Suc-eq-plus1 by simp
  then show \(?thesis using pos2 by (rule LIMSEQ-linear)\)
next
  case False
  hence \( x = -1 \lor x = 1 \) using \(|x| \leq 1\) by auto
  unfolding One-nat-def by auto

from tendsto-mult[\{OF LIMSEQ-inverse-real-of-nat|THEN LIMSEQ-linear, OF pos2, unfolded inverse-eq-divide\}]
tendsto-const[\{of x\}]
  show \(?thesis unfolding n-eq Suc-eq-plus1 by auto\)
qed

lemma \text{summable-arctan-series}:
  fixes \( x :: \text{real} \) and \( n :: \text{nat} \)
  assumes \(|x| \leq 1\)
  shows \( \text{summable } (\lambda k. (-1)^k * (1 / real (k*2+1) * x ^ (k*2+1)))\)
    (is \( \text{summable } (\?c x)\))
    by (rule summable-Leibniz(1), rule zeroseq-arctan-series[OF assms], rule monoseq-arctan-series[OF assms])

lemma \text{less-one-imp-sqr-less-one}:
  fixes \( x :: \text{real} \)
  assumes \(|x| < 1\)
  shows \( x^2 < 1\)
  proof
    have \(|x|^2 < 1\)
      by (metis abs-power2 assms pos2 power2-abs power-0 power-strict-decreasing zero-eq-power2 zero-less-iff)
    thus \(?thesis using zero-le-power2 by auto\)
  qed

lemma \text{DERIV-arctan-series}:
  assumes \(|x| < 1\)
  shows \( \text{DERIV } (\lambda x'. \sum k. (-1)^k * (1 / real (k*2+1) * x ^ (k*2+1))) \ x :> \sum k. (-1)^k * x ^ (k*2)\)
    (is \( \text{DERIV } (?\text{arctan} - :) \ ?\text{Int}\))
  proof
    let \( ?f = \lambda n. \text{if even } n \text{ then } (-1)^n \text{ div } 2 * 1 / \text{real } (\text{Suc} \ n) \text{ else } 0\)
    have \( n\text{-even}: \land n :: \text{nat}. \text{even } n \Longrightarrow 2 * (n \text{ div } 2) = n\)
      by presburger
    then have \( \land n \ x'. \ ?f n * \text{real } (\text{Suc} \ n) * x' ^ n = \)
      \( (\text{if even } n \text{ then } (-1)^{(n \text{ div } 2)} * x' ^ {(2 * (n \text{ div } 2))} \text{ else } 0)\)
      by auto
\[
\{ \\
\text{fix } x :: \text{real} \\
\text{assume } |x| < 1 \\
\text{hence } x^2 < 1 \text{ by (rule less-one-imp-sqr-less-one)} \\
\text{have summable } (\lambda n. -1 \ ^n * (x^2) \ ^n) \\
\text{by (rule summable-Leibniz(1), auto intro: LIMSEQ-realpow-zero monoseq-realpow} \\
(x^2 < 1) \ \text{order-less-imp-}[OF (x^2 < 1)] \\
\text{hence summable } (\lambda n. -1 \ ^n * x \ ^{(2*n)}) \text{ unfolding power-mult} . \\
\} \text{ note summable-Integral = this} \\
\}
\]

\[
\{ \\
\text{fix } f :: \text{nat} \Rightarrow \text{real} \\
\text{have } \exists x. f \text{ sums x } = (\lambda n. \text{if even } n \text{ then } f \text{ (n div } 2) \text{ else 0}) \text{ sums x} \\
\text{proof} \\
\text{fix } x :: \text{real} \\
\text{assume } f \text{ sums x} \\
\text{from sums-if[OF sums-zero this]} \\
\text{show } (\lambda n. \text{if even } n \text{ then } f \text{ (n div } 2) \text{ else 0}) \text{ sums x} \\
\text{by auto} \\
\text{next} \\
\text{fix } x :: \text{real} \\
\text{assume } (\lambda n. \text{if even } n \text{ then } f \text{ (n div } 2) \text{ else 0}) \text{ sums x} \\
\text{from LIMSEQ-linear[OF this[unfolded sums-def] pos2, unfolded sum-split-even-odd[unfolded mult.commute]} \\
\text{show } f \text{ sums x unfolding sums-def by auto} \\
\text{qed} \\
\text{note sums-even = this} \\
\}
\]

\[
\text{have } \text{Int-eq: } \sum n. \text{if } n \text{ * real } (Suc n) \text{ * } x \ ^n = \ ?Int \\
\text{unfolding if-eq mult.commute[of - 2] suminf-def sums-even[of } \lambda n. -1 \ ^n \ * x \ ^{(2 * n)}, \text{ symmetric]} \\
\text{by auto} \\
\}
\]

\[
\{ \\
\text{fix } x :: \text{real} \\
\text{have if-eq': } \forall n. \text{if even } n \text{ then } -1 \ ^n \ * (n \text{ div } 2) \ * 1 / \text{ real } (Suc n) \text{ else 0} \ * x \\
\text{Suc } n \text{ =} \\
\text{if even } n \text{ then } -1 \ ^n \ * (n \text{ div } 2) \ * (1 / \text{ real } (Suc (2 * (n \text{ div } 2)))) \ * x \ ^\text{Suc (2 * (n \text{ div } 2)))} \text{ else 0) } \\
\text{using n-even by auto} \\
\text{have idx-eq: } \forall n. \text{ n * 2 + 1 = Suc (2 * n) by auto} \\
\text{have } (\sum n. \text{if } n \text{ * x } ^\text{(Suc n)}) = \ ?arctan x \\
\text{unfolding if-eq' idx-eq suminf-def sums-even[of } \lambda n. -1 \ ^n \ * (1 / \text{ real } (Suc (2 * n))) \ * x \ ^\text{Suc (2 * n)}, \text{ symmetric]} \\
\text{by auto} \\
\} \text{ note arctan-eq = this} \\
\}
have $\text{DERIV} \ (\lambda x. \sum n. \ ?f n * x^-n(Suc n)) \ x :> (\sum n. \ ?f n * \text{real} (\text{Suc n}) * x^n)$

proof (rule $\text{DERIV-power-series}$)
  \begin{enumerate}
  \item show $x \in \{-1 <..< 1\}$ using $\langle |x| < 1 \rangle$ by auto
    \begin{enumerate}
    \item fix $x' :: \text{real}$
    \item assume $x'$-bounds: $x' \in \{-1 <..< 1\}$
    \item hence $|x'| < 1$ by auto
    \end{enumerate}
  \item let $\ ?S = \sum n. \ (-1)^n * x'^(2 * n)$
  \item show summable $(\lambda n. \ ?f n * \text{real} (\text{Suc n}) * x'^n)$ unfolding if-eq
    \begin{enumerate}
    \item by (rule sums-summable[where $l=0 + \ ?S$], rule sums-if, rule sums-zero, rule summable-sums, rule summable-Integral[OF $\langle |x'| < 1 \rangle$])
    \end{enumerate}
  \item qed auto
  \end{enumerate}
thus $\ ?\text{thesis}$ unfolding Int-eq arctan-eq.
\qed

lemma arctan-series:
  \begin{enumerate}
  \item assumes $|x| \leq 1$
  \item shows $\text{arctan} x = (\sum k. \ (-1)^k * (1 / \text{real} (k+2+1)) * x^-((k+2)+1))$
  \item (is - = suminf $(\lambda x. \ ?c x k)$)
  \end{enumerate}
\proof
\begin{enumerate}
\item let $\ ?c' = \lambda x n. \ (-1)^n * x'^(n+2)$
  \begin{enumerate}
  \item fix $r x :: \text{real}$
  \item assume $\theta < r$ and $r < 1$ and $|x| < r$
  \item have $|x| < 1$ using $\langle r < 1 \rangle$ and $\langle |x| < r \rangle$ by auto
  \item from $\text{DERIV-arctan-series[OF this]}$ have $\text{DERIV} \ (\lambda x. \ ?c x k)$
  \item (suminf $(\ ?c' x)$).
  \end{enumerate}
\item note $\text{DERIV-arctan-suminf} = \text{this}$
\end{enumerate}
\proof
\begin{enumerate}
\item fix $x :: \text{real}$
\item assume $|x| \leq 1$
\item note $\text{summable-Leibniz[OF zeroseq-arctan-series[OF this]} \text{monoseq-arctan-series[OF this]}$]
\item note $\text{arctan-series-borders} = \text{this}$
\end{enumerate}
\proof
\begin{enumerate}
\item fix $x :: \text{real}$
\item assume $|x| \leq 1$
\item have $\text{arctan} x = (\sum k. \ ?c x k)$
\item proof --
  \begin{enumerate}
  \item obtain $r$ where $|x| < r$ and $r < 1$
  \item using dense[OF $\langle |x| < 1 \rangle$] by blast
  \item hence $\theta < r$ and $-r < x$ and $x < r$ by auto
  \end{enumerate}
\end{enumerate}
have suminf-eq-arctan-bounded: \( \forall x \ a \ b. \ [ -r < a ; b < r ; a < b ; a \leq x ; x \leq b ] \implies \suminf (\ ?c \ x) - \arctan x = \suminf (\ ?c \ a) - \arctan a \) 

proof - 
fix x a b 
assume \(-r < a \) and \( b < r \) and \( a < b \) and \( a \leq x \) and \( x \leq b \) 
hence \(|x| < r \) by auto 
show \( \suminf (\ ?c \ x) - \arctan x = \suminf (\ ?c \ a) - \arctan a \) 
proof (rule DERIV-isconst2[of a b]) 
show \( a < b \) and \( a \leq x \) and \( x \leq b \) 
using \( (a < b) \ (a \leq x) \ (x \leq b) \) by auto 
have \( \forall \ x. \ -r < x \land x < r \implies \text{DERIV} (\lambda x. \ \suminf (\ ?c \ x) - \arctan x) \) 
\( x > : 0 \) 
proof (rule allI, rule impI) 
fix x 
assume \(-r < x \land x < r \) 
hence \(|x| < r \) by auto 
hence \(|x| < 1 \) using \((r < 1) \) by auto 
have \(| - (x^2) | < 1 \) 
using \( \text{less-one-imp-sqr-less-one[OF \langle|x| < 1\rangle]} \) by auto 
hence \((\lambda n. \ (- (x^2)) \ ^{n}) \sums (1 / (1 - (- (x^2))))\) 
unfolding \( \text{real-norm-def[symmetric]} \) by \( \text{rule geometric-sums} \) 
hence \((\ ?c' \ x) \sums (1 / (1 - (- (x^2))))\) 
unfolding \( \text{power-mult-distrib[symmetric]} \ \text{power-mult mult.commute[of - 2]} \) by auto 
\( \text{hence \( \text{suminf-c'-eq-geom: inverse} \ (1 + x^2) = \suminf (\ ?c' \ x) \) } \) 
using \( \text{sums-unique} \) unfolding \( \text{inverse-eq-divide by auto} \) 
have \( \text{DERIV} (\lambda x. \ \suminf (\ ?c \ x)) \ x > : (\text{inverse} (1 + x^2)) \) 
unfolding \( \text{suminf-c'-eq-geom} \) 
by \( \text{rule DERIV-arctan-suminf[OF \langle 0 < r \rangle \langle r < 1 \rangle \langle |x| < r \rangle] } \) 
from \( \text{DERIV-diff[OF this DERIV-arctan]} \) 
show \( \text{DERIV} (\lambda x. \ \suminf (\ ?c \ x) - \arctan x) \ x > : 0 \) 
by auto 
qed 

hence \( \text{DERIV-in-rball:} \ \forall y. \ a \leq y \land y \leq b \implies \text{DERIV} (\lambda x. \ \suminf (\ ?c \ x) - \arctan x) \ y \ > : 0 \) 
using \( (-r < a \ \langle b < r \rangle) \) by auto 
thus \( \forall y. \ a < y \land y < b \implies \text{DERIV} (\lambda x. \ \suminf (\ ?c \ x) - \arctan x) \ y \ > : 0 \) 
using \( \langle |x| < r \rangle \) by auto 
show \( \forall y. \ a \leq y \land y \leq b \implies \text{isCont} (\lambda x. \ \suminf (\ ?c \ x) - \arctan x) \ y \) 
using \( \text{DERIV-in-rball DERIV-isCont by auto} \) 
qed 

have \( \text{suminf-arctan-zero:} \ \suminf (\ ?c \ 0) - \arctan 0 = 0 \) 
unfolding \( \text{Suc-eq-plus1[symmetric]} \ \text{power-Suc2 mult-zero-right arctan-zero-zero} \ \text{suminf-zero} \) 
by auto
have \( \operatorname{suminf} (\lambda x . ?c x) - \arctan x = 0 \)

proof (cases \( x = 0 \))
  case True
  thus \(?thesis\) using \( \operatorname{suminf-arctan-zero} \) by auto

next
  case False
  hence \( 0 < |x| \) and \(-|x| < |x|\) by auto
  have \( \operatorname{suminf} (\lambda c \cdot (-|x|)) - \arctan (-|x|) = \operatorname{suminf} (?c 0) - \arctan 0 \)
    by (rule \( \operatorname{suminf-eq-arctan-bounded} \) [where \( x=0 \) and \( a=-|x| \) and \( b=|x| \], symmetric))
    (simp-all only: \(|x| < r\) (- |x| < |x|) neg-less-iff-less)
  moreover
  have \( \operatorname{suminf} (\lambda x . ?c x) - \arctan x = \operatorname{suminf} (\lambda c \cdot (-|x|)) - \arctan x \)
    by (rule \( \operatorname{suminf-eq-arctan-bounded} \) [where \( x=x \) and \( a=-|x| \) and \( b=|x| \])
    (simp-all only: \(|x| < r\) (- |x| < |x|) neg-less-iff-less)
  ultimately
  show \(?thesis\) using \( \operatorname{suminf-arctan-zero} \) by auto
  thus \(?thesis\) by auto
  qed

  note when-less-one = this

show \( \arctan x = \operatorname{suminf} (\lambda n . ?c x n) \)
proof (cases \(|x| < 1\))
  case True
  thus \(?thesis\) by (rule when-less-one)

next
  case False
  hence \(|x| = 1\) using \(|x| \leq 1\) by auto
  let \(?a = \lambda x n . |1 / \text{real} (n*2+1) * x ^ (n*2+1)|\)
  let \(?\text{diff} = \lambda x n . | \arctan x - (\sum i < n . ?c x i)|\)
  
  fix \( n :: \text{nat} \)
  have \( 0 < (1 :: \text{real}) \) by auto
  moreover
  
  fix \( x :: \text{real} \)
  assume \( 0 < x \) and \( x < 1 \)
  hence \(|x| \leq 1\) and \(|x| < 1\) by auto
  from \((0 < x)\) have \( 0 < 1 / \text{real} (0 * 2 + (1 :: \text{nat})) \) * \( x ^ (0 * 2 + 1)\)
    by auto
  note bounds = mp[OF \( \arctan-series-borders(2)[OF \ (|x| \leq 1)]\) this, unfolded when-less-one[OF \(|x| < 1\), symmetric], THEN spec]
  have \( 0 < 1 / \text{real} (n*2+1) \) * \( x ^ (n*2+1)\)
    by (rule \( \text{mult-pos-pos} \), auto simp only: zero-less-power[OF \( 0 < x \)], auto)
  hence \( a-pos: \ ?a x n = 1 / \text{real} (n*2+1) \) * \( x ^ (n*2+1)\)
    by (rule abs-of-pos)
  have \(?\text{diff} x n \leq ?a x n\)
proof (cases even $n$)
  case True
  hence $\text{sgn-pos} \colon (-1)^n = (1::\text{real})$ by auto
  from (even $n$) obtain $m$ where $2 \ast m = n$
    unfolding even-mult-two-ex by auto
  from bounds[of $m$, unbounded this atLeastAtMost-if]
  have $|\text{arctan } x - (\sum_{i<n} (?c \times x \times i))| \leq (\sum_{i<n+1} (?c \times x \times i)) - (\sum_{i<n} (?c \times x \times i))$
    by auto
  also have $\ldots = ?c \times x \times n$ unfolding One-nat-def by auto
  also have $\ldots = ?a \times x \times n$ unfolding $\text{sgn-pos a-pos}$ by auto
  finally show $\text{thesis}$.
next
  case False
  hence $\text{sgn-neg} \colon (-1)^n = (-1::\text{real})$ by auto
  from (odd $n$) obtain $m$ where $2 \ast m + 1 = n$
    unfolding odd-Suc-mult-two-ex by auto
  hence $m+\text{plus} \colon 2 \ast (m + 1) = n + 1$ by auto
  from bounds[of $m + 1$, unbounded this atLeastAtMost-if, THEN conjunct1]
  bounds[of $m$, unbounded $m$-def atLeastAtMost-if, THEN conjunct2]
  have $|\text{arctan } x - (\sum_{i<n} (?c \times x \times i))| \leq (\sum_{i<n+1} (?c \times x \times i)) - (\sum_{i<n+1} (?c \times x \times i))$
    by auto
  also have $\ldots = -?c \times x \times n$ unfolding One-nat-def by auto
  also have $\ldots = ?a \times x \times n$ unfolding $\text{sgn-neg a-pos}$ by auto
  finally show $\text{thesis}$.
qed
  hence $0 \leq ?a \times x \times n - ?\text{diff } x \times n$ by auto
}
  hence $\forall x \in \{0 <..< 1 \}. 0 \leq ?a \times x \times n - ?\text{diff } x \times n$ by auto
moreover have $\forall x. \text{isCont } (\lambda x. ?a \times x \times n - ?\text{diff } x \times n) x$
  unfolding diff-conv-add-uminus divide-inverse
by (auto intro!: isCont-add isCont-rabs isCont-const isCont-setsum
  simp del: add-uminus-conv-diff)
ultimately have $0 \leq ?a \times 1 \times n - ?\text{diff } 1 \times n$
by (rule LIM-less-bound)
  hence $?\text{diff } 1 \times n \leq ?a \times 1 \times n$ by auto
}
  have $?a \times 1 \times n$ by auto
  unfolding tendsto-rabs-zero-iff power-one divide-inverse One-nat-def
by (auto intro!: tendsto-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat)
  have $?\text{diff } 1 \times n$ by auto
proof (rule LIMSEQ-I)
  fix $r :: \text{real}$
  assume $0 < r$
  obtain $N :: \text{nat}$ where $N-I : \forall n. N \leq n \implies ?a \times 1 \times n < r$
    using LIMSEQ-D[OF $?a \times 1 \times n$ $0 < r$] by auto

fix $n$
assume $N \leq n$ from (?diff 1 $n \leq ?a 1 n$) N-I[OF this]
have norm (?diff 1 $n - 0$) < $r$ by auto
}
thus $\exists N. \forall n \geq N. \text{norm} (\text{?diff 1 $n - 0$}) < r$ by blast
qed
from this [unfolded tendsto-rabs-zero-iff, THEN tendsto-add [OF - tendsto-const],
of - arctan 1, THEN tendsto-minus]
have (?c 1) sums (arctan 1) unfolding sums-def by auto
hence arctan 1 = ($\sum i. ?c 1 i$) by (rule sums-unique)

show ?thesis
proof (cases $x = 1$)
  case True then show ?thesis by (simp add: arctan 1 = ($\sum i. ?c 1 i$))
next
  case False hence $x = -1$ using $|x| = 1$ by auto

have $-(pi / 2) < 0$ using pi-gt-zero by auto
have $-(2 * pi) < 0$ using pi-gt-zero by auto

have c-minus-minus: $\forall i. ?c (-1) i = - ?c 1 i$
  unfolding One-nat-def by auto

have arctan $(-1) = arctan (\tan (- (pi / 4)))$
  unfolding tan-45 tan-minus ..
  also have ... = $-(pi / 4)$
    by (rule arctan-tan, auto simp add: order-less-trans[OF $-(pi / 2) < 0$
pi-gt-zero])
  also have ... = $-(arctan (\tan (pi / 4)))$
    unfolding neg-equal-iff-equal by (rule arctan-tan[symmetric], auto simp
add: order-less-trans[OF $-(2 * pi) < 0$ pi-gt-zero])
  also have ... = $-(\sum i. ?c 1 i)$
    unfolding tan-45 ..
  also have ... = $-(\sum i. ?c 1 i)$
    using arctan 1 = ($\sum i. ?c 1 i$) by auto
  also have ... = $-(\sum i. ?c 1 i)$
    unfolding suminf-minus[OF sums-summable[OF $((?c 1) \text{ sums (arctan 1)})$]]
    unfolding c-minus-minus by auto
finally show ?thesis using $x = -1$ by auto
qed
qed
qed

lemma arctan-half:
  fixes $x :: \text{real}$
  shows $\arctan x = 2 * \arctan (x / (1 + \sqrt{(1 + x^2)}))$
proof –
obtain \( y \) where low: \(- (\pi / 2) < y \) and high: \( y < \pi / 2 \) and y-eq: \( \tan y = x \)
using \( \text{tan-total} \) by blast
hence low2: \(- (\pi / 2) < y / 2 \) and high2: \( y / 2 < \pi / 2 \)
by auto

have \( 0 < \cos y \) using \( \cos-gt-zero-\pi[\text{OF low high}] \).
hence \( \cos y \neq 0 \) and \( \cos-sqrt: \sqrt{((\cos y)^2)} = \cos y \)
by auto

have \( 1 + (\tan y)^2 = 1 + (\sin y)^2 / (\cos y)^2 \)
unfolding \( \text{tan-def power-divide} \) ..
also have \( . . . = (\cos y)^2 / (\cos y)^2 + (\sin y)^2 / (\cos y)^2 \)
using \( \langle \cos y \neq 0 \rangle \) by auto
also have \( . . . = 1 / (\cos y)^2 \)
unfolding \( \text{add-divide-distrib[symmetric]} \) \( \sin-cos-squared-add2 \) ..
finally have \( 1 + (\tan y)^2 = 1 / (\cos y)^2 \).

have \( \sin y / (\cos y + 1) = \tan y / ((\cos y + 1) / \cos y) \)
unfolding \( \text{tan-def using} \ (\cos y \neq 0) \) by (simp add: field-simps)
also have \( . . . = \tan y / (1 + 1 / \cos y) \)
using \( \langle \cos y \neq 0 \rangle \) unfolding \( \text{add-divide-distrib} \) by auto
also have \( . . . = \tan y / (1 + \sqrt{1 / (\cos y)^2}) \)
unfolding \( \cos-sqrt \) ..
also have \( . . . = \tan y / (1 + \sqrt{1 / (\cos y)^2}) \)
unfolding \( \text{real-sqrt-divide} \) by auto
finally have eq: \( \sin y / (\cos y + 1) = \tan y / (1 + \sqrt{1 + (\tan y)^2}) \)
unfolding \( .1 + (\tan y)^2 = 1 / (\cos y)^2 \).

have \( \arctan x = y \)
using \( \arctan-tan \) low high y-eq by auto
also have \( . . . = 2 * (\arctan (\tan (y / 2))) \)
using \( \arctan-tan[\text{OF low2 high2}] \) by auto
also have \( . . . = 2 * (\arctan (\sin y / (\cos y + 1))) \)
unfolding \( \text{tan-half} \) by auto
finally show \( \text{thesis} \)
unfolding eq (\( \tan y = x \)).

qed

lemma \( \arctan-monotone: x < y \Rightarrow \arctan x < \arctan y \)
by (simp only: \( \arctan-less-iiff \))

lemma \( \arctan-monotone': x \leq y \Rightarrow \arctan x \leq \arctan y \)
by (simp only: \( \arctan-le-iiff \))

lemma \( \arctan-inverse: \)
assumes \( x \neq 0 \)
shows \( \arctan (1 / x) = \text{sgn} x * \pi / 2 - \arctan x \)
proof (rule \( \arctan-unique \))
show \( -(\pi / 2) < \text{sgn} x * \pi / 2 - \arctan x \)

using arctan-bounded [of x] assms

unfolding sgn-real-def

apply (auto simp add: algebra-simps)

apply (drule zero-less-arctan-iff [THEN iffD2])

apply arith

done

show sgn x * pi / 2 – arctan x < pi / 2

using arctan-bounded [of x] assms

unfolding sgn-real-def arctan-minus

by (auto simp add: algebra-simps)

show tan (sgn x * pi / 2 – arctan x) = 1 / x

unfolding tan-inverse [of arctan x]

by (simp add: tan-def cos-arctan sin-arctan sin-diff cos-diff)

qed

theorem pi-series: pi / 4 = (∑ k. (−1)^k * 1 / real (k*2+1)) (is = ?SUM)

proof –

have pi / 4 = arctan 1 using arctan-one by auto

also have . . . = ?SUM using arctan-series[of 1] by auto

finally show ?thesis by auto

qed

105.15 Existence of Polar Coordinates

lemma cos-x-y-le-one: |x / sqrt (x^2 + y^2)| ≤ 1

apply (rule power2-le-imp-le [OF - zero-le-one])

apply (simp add: power-divide divide-le-eq not-sum-power2-lt-zero)

done

lemma cos-arccos-abs: |y| ≤ 1 → cos (arccos y) = y

by (simp add: abs-le-iff)

lemma sin-arccos-abs: |y| ≤ 1 → sin (arccos y) = sqrt (1 - y^2)

by (simp add: sin-arccos abs-le-iff)

lemmas cos-arccos-lemma1 = cos-arccos-abs [OF cos-x-y-le-one]

lemmas sin-arccos-lemma1 = sin-arccos-abs [OF cos-x-y-le-one]

lemma polar-Ex: ∃ r a. x = r * cos a & y = r * sin a

proof –

have polar-ex1: (∃ y. 0 < y) → ∃ r a. x = r * cos a & y = r * sin a

apply (rule_tac x = sqrt (x^2 + y^2) in exI)

apply (rule-tac x = arccos (x / sqrt (x^2 + y^2)) in exI)

apply (simp add: cos-arccos-lemma1 sin-arccos-lemma1 power-divide real-sqrt-mult [symmetric] right-diff-distrib)

done

show ?thesis
proof (cases 0::real y rule: linorder-cases)
  case less
  then show ?thesis by (rule polar-ex1)
next
  case equal
  then show ?thesis by (force simp add: intro!: cos-zero sin-zero)
next
  case greater
  then show ?thesis using polar-ex1 [where y=−y]
  by auto (metis cos-minus minus-minus minus-mult-right sin-minus)
qed
qed

end

106 Complex: Complex Numbers: Rectangular and Polar Representations

theory Complex
imports Transcendental
begin

We use the codatatype-command to define the type of complex numbers. This might look strange at first, but allows us to use primcorec to define complex-functions by defining their real and imaginary result separate.
codatatype complex = Complex (Re: real) (Im: real)

lemma complex-surj: Complex (Re z) (Im z) = z
  by (rule complex.collapse)

lemma complex-eqI [intro?]: [Re x = Re y; Im x = Im y] ==> x = y
  by (rule complex.expand) simp

lemma complex-eq-iff: x = y <-> Re x = Re y ∧ Im x = Im y
  by (auto intro: complex.expand)

106.1 Addition and Subtraction

instantiation complex :: ab-group-add
begin

primcorec zero-complex where
  Re 0 = 0
| Im 0 = 0

primcorec plus-complex where
\[ \text{Re} (x + y) = \text{Re} x + \text{Re} y \]
\[ \text{Im} (x + y) = \text{Im} x + \text{Im} y \]

**primcorec** `uminus-complex` where
\[ \text{Re} (-x) = -\text{Re} x \]
\[ \text{Im} (-x) = -\text{Im} x \]

**primcorec** `minus-complex` where
\[ \text{Re} (x - y) = \text{Re} x - \text{Re} y \]
\[ \text{Im} (x - y) = \text{Im} x - \text{Im} y \]

**instance**
by `intro-classes` (simp-all `add`: `complex-eq-iff`)

`end`

### 106.2 Multiplication and Division

**instantiation** `complex :: field-inverse-zero`

**begin**

**primcorec** `one-complex` where
\[ \text{Re} 1 = 1 \]
\[ \text{Im} 1 = 0 \]

**primcorec** `times-complex` where
\[ \text{Re} (x \ast y) = \text{Re} x \ast \text{Re} y - \text{Im} x \ast \text{Im} y \]
\[ \text{Im} (x \ast y) = \text{Re} x \ast \text{Im} y + \text{Im} x \ast \text{Re} y \]

**primcorec** `inverse-complex` where
\[ \text{Re} (\text{inverse } x) = \text{Re} x / ((\text{Re} x)^2 + (\text{Im} x)^2) \]
\[ \text{Im} (\text{inverse } x) = -\text{Im} x / ((\text{Re} x)^2 + (\text{Im} x)^2) \]

definition `x / (y::complex) = x * inverse y`

**instance**
by `intro-classes`
(simp-all `add`: `complex-eq-iff` `divide-complex-def` `distrib-left` `distrib-right` `right-diff-distrib` `left-diff-distrib` `power2-eq-square` `add-divide-distrib` [symmetric])

**end**

**lemma** `Re-divide`: \[ \text{Re} (x / y) = (\text{Re} x \ast \text{Re} y + \text{Im} x \ast \text{Im} y) / ((\text{Re} y)^2 + (\text{Im} y)^2) \]
unfolding `divide-complex-def` by (simp `add`: `add-divide-distrib`)

**lemma** `Im-divide`: \[ \text{Im} (x / y) = (\text{Im} x \ast \text{Re} y - \text{Re} x \ast \text{Im} y) / ((\text{Re} y)^2 + (\text{Im} y)^2) \]
unfolding divide-complex-def times-complex.sel inverse-complex.sel by (simp-all add: divide-simps)

lemma Re-power2: Re (x ^ 2) = (Re x)^2 - (Im x)^2 by (simp add: power2-eq-square)

lemma Im-power2: Im (x ^ 2) = 2 * Re x * Im x by (simp add: power2-eq-square)

lemma Re-power-real: Re x ^ n = Re x ^ n by (induct n) simp-all

lemma Im-power-real: Im x ^ n = 0 by (induct n) simp-all

106.3 Scalar Multiplication

instantiation complex :: real-field begin

primcorec scaleR-complex where
  Re (scaleR r x) = r * Re x
| Im (scaleR r x) = r * Im x

instance proof
  fix a b :: real and x y :: complex
  show scaleR a (x + y) = scaleR a x + scaleR a y
    by (simp add: complex-eq-iff distrib-left)
  show scaleR (a + b) x = scaleR a x + scaleR b x
    by (simp add: complex-eq-iff distrib-right)
  show scaleR a (scaleR b x) = scaleR (a * b) x
    by (simp add: complex-eq-iff mult.assoc)
  show scaleR 1 x = x
    by (simp add: complex-eq-iff)
  show scaleR a x * y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
  show x * scaleR a y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
qed

end

106.4 Numerals, Arithmetic, and Embedding from Reals

abbreviation complex-of-real :: real ⇒ complex
  where complex-of-real ≡ of-real

declare [[coercion complex-of-real]]
declare [[coercion of-int :: int ⇒ complex]]
declare [[coercion of-nat :: nat ⇒ complex]]

lemma complex-Re-of-nat [simp]: \( \text{Re} (\text{of-nat } n) = \text{of-nat } n \)
  by (induct n) simp-all

lemma complex-Im-of-nat [simp]: \( \text{Im} (\text{of-nat } n) = 0 \)
  by (induct n) simp-all

lemma complex-Re-of-int [simp]: \( \text{Re} (\text{of-int } z) = \text{of-int } z \)
  by (cases z rule: int-diff-cases) simp

lemma complex-Im-of-int [simp]: \( \text{Im} (\text{of-int } z) = 0 \)
  by (cases z rule: int-diff-cases) simp

lemma complex-Re-numeral [simp]: \( \text{Re} (\text{numeral } v) = \text{numeral } v \)
  using complex-Re-of-int [of numeral v] by simp

lemma complex-Im-numeral [simp]: \( \text{Im} (\text{numeral } v) = 0 \)
  using complex-Im-of-int [of numeral v] by simp

lemma Re-complex-of-real [simp]: \( \text{Re} (\text{complex-of-real } z) = z \)
  by (simp add: of-real-def)

lemma Im-complex-of-real [simp]: \( \text{Im} (\text{complex-of-real } z) = 0 \)
  by (simp add: of-real-def)

106.5 The Complex Number \( i \)

primcorec \( ii :: \text{complex} \) (i) where
  \( \text{Re } ii = 0 \)
| \( \text{Im } ii = 1 \)

lemma Complex-eq[simp]: Complex a b = a + i * b
  by (simp add: complex-eq-iff)

lemma complex-eq: a = Re a + i * Im a
  by (simp add: complex-eq-iff)

lemma fun-complex-eq: \( f = (\lambda x. \text{Re} (f x) + i * \text{Im} (f x)) \)
  by (simp add: fun-eq-iff complex-eq)

lemma i-squared [simp]: \( ii * ii = -1 \)
  by (simp add: complex-eq-iff)

lemma power2-i [simp]: \( ii^2 = -1 \)
  by (simp add: power2-eq-square)

lemma inverse-i [simp]: inverse ii = - ii
  by (rule inverse-unique simp)
lemma divide-i [simp]: \( x / i = -i * x \)
by (simp add: divide-complex-def)

lemma complex-i-mult-minus [simp]: \( i * (i * x) = -x \)
by (simp add: mult.assoc [symmetric])

lemma complex-i-not-zero [simp]: \( i \neq 0 \)
by (simp add: complex-eq-iff)

lemma complex-i-not-one [simp]: \( i \neq 1 \)
by (simp add: complex-eq-iff)

lemma complex-i-not-numeral [simp]: \( i \neq \text{numeral } w \)
by (simp add: complex-eq-iff)

lemma complex-i-not-neg-numeral [simp]: \( i \neq -\text{numeral } w \)
by (simp add: complex-eq-iff)

lemma complex-split-polar: \( \exists r a. z = \text{complex-of-real } r * (\cos a + i * \sin a) \)
by (simp add: complex-eq-iff polar-Ex)

106.6 Vector Norm

instantiation complex :: real-normed-field
begin

definition norm z = sqrt ((Re z)^2 + (Im z)^2)

abbreviation cmod :: complex ⇒ real
where cmod ≡ norm

definition complex-sgn-def:
sgn x = x / R cmod x

definition dist-complex-def:
dist x y = cmod (x - y)

definition open-complex-def:
open (S :: complex set) ←→ (∀ x∈S. ∃ e>0. ∀ y. dist y x < e → y ∈ S)

instance proof
fix r :: real and x y :: complex and S :: complex set
show (norm x = 0) = (x = 0)
  by (simp add: norm-complex-def complex-eq-iff)

show norm (x + y) ≤ norm x + norm y
  by (simp add: norm-complex-def complex-eq-iff real-sqrt-sum-squares-triangle-ineq)

show norm (scaleR r x) = |r| * norm x
  by (simp add: norm-complex-def complex-eq-iff power-mult-distrib distrib-left)
THEORY “Complex”

[ symmetric] real-sqrt-mult
  show norm (x * y) = norm x * norm y
  by (simp add: norm-complex-def complex-eq-iff real-sqrt-mult [symmetric] power2-eq-square algebra-simps)
qed (rule complex-sgn-def dist-complex-def open-complex-def)+
end

lemma norm-ii [simp]: norm ii = 1
  by (simp add: norm-complex-def)

lemma cmod-unit-one: cmod (cos a + i * sin a) = 1
  by (simp add: norm-complex-def)

lemma cmod-complex-polar: cmod (r * (cos a + i * sin a)) = |r|
  by (simp add: norm-mult cmod-unit-one)

lemma complex-Re-le-cmod: Re x ≤ cmod x
  unfolding norm-complex-def
  by (rule real-sqrt-sum-squares-ge1)

lemma complex-mod-minus-le-complex-mod: − cmod x ≤ cmod x
  by (rule order-trans [OF - norm-ge-zero] simp)

lemma complex-mod-triangle-ineq2: cmod (b + a) − cmod b ≤ cmod a
  by (rule ord-le-eq-trans [OF norm-triangle-ineq2] simp)

lemma abs-Re-le-cmod: |Re x| ≤ cmod x
  by (simp add: norm-complex-def)

lemma abs-Im-le-cmod: |Im x| ≤ cmod x
  by (simp add: norm-complex-def)

lemma cmod-le: cmod z ≤ |Re z| + |Im z|
  apply (subst complex-eq)
  apply (rule order-trans)
  apply (rule norm-triangle-ineq)
  apply (simp add: norm-mult)
  done

lemma cmod-eq-Re: Im z = 0 ⇒ cmod z = |Re z|
  by (simp add: norm-complex-def)

lemma cmod-eq-Im: Re z = 0 ⇒ cmod z = |Im z|
  by (simp add: norm-complex-def)

lemma cmod-power2: cmod z ^ 2 = (Re z) ^ 2 + (Im z) ^ 2
  by (simp add: norm-complex-def)
lemma cmod-plus-Re-le-0-iff: \( cmod z + \Re z \leq 0 \iff \Re z = - cmod z \) using abs-Re-le-cmod[of \( z \)] by auto

lemma Im-eq-0: \( |\Re z| = cmod z \Rightarrow Im z = 0 \) by (subst (asm) power-eq-iff-eq-base[symmetric, where \( n=2 \)])

( auto simp add: norm-complex-def )

lemma abs-sqrt-wlog: fixes \( x::'a::linordered-idom \) assumes \( \forall x::'a. x \geq 0 =\Rightarrow P x \) ( \( x^2 \) ) shows \( P |x| (x^2) \) by (metis abs-ge-zero assms power2-abs)

lemma complex-abs-le-norm: \( |\Re z| + |\Im z| \leq \sqrt{2} * \norm z \) unfolding norm-complex-def apply (rule abs-sqrt-wlog [where \( x=Re z \)])

apply (rule abs-sqrt-wlog [where \( x=Im z \)])

apply (rule power2-le-imp-le)

apply (simp-all add: power2-sum add.commute sum-squares-bound real-sqrt-mult [symmetric])

done

Properties of complex signum.

lemma sgn-eq: \( \sgn z = z / \text{complex-of-real} (cmod z) \)

by (simp add: sgn-div-norm divide-inverse scaleR-conv-of-real mult.commute)

lemma Re-sgn [simp]: \( \Re(\sgn z) = \Re(z)/cmod z \)

by (simp add: complex-sgn-def divide-inverse)

lemma Im-sgn [simp]: \( \Im(\sgn z) = \Im(z)/cmod z \)

by (simp add: complex-sgn-def divide-inverse)

106.7 Completeness of the Complexes

lemma bounded-linear-Re: bounded-linear Re

by (rule bounded-linear-intro [where \( K=1 \)], simp-all add: norm-complex-def)

lemma bounded-linear-Im: bounded-linear Im

by (rule bounded-linear-intro [where \( K=1 \)], simp-all add: norm-complex-def)

lemmas Cauchy-Re = bounded-linear.Cauchy [OF bounded-linear-Re]

lemmas Cauchy-Im = bounded-linear.Cauchy [OF bounded-linear-Im]

lemmas tendsto-Re [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Re]

lemmas tendsto-Im [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Im]

lemmas isCont-Re [simp] = bounded-linear.isCont [OF bounded-linear-Re]

lemmas isCont-Im [simp] = bounded-linear.isCont [OF bounded-linear-Im]

lemmas continuous-Re [simp] = bounded-linear.continuous [OF bounded-linear-Re]

lemmas continuous-Im [simp] = bounded-linear.continuous [OF bounded-linear-Im]

lemmas continuous-on-Re [continuous-intros] = bounded-linear.continuous-on [OF bounded-linear-Re]
lemmas continuous-on-Im [continuous-intros] = bounded-linear.continuous-on[OF bounded-linear-Im]
lemmas has-derivative-Re [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Re]
lemmas has-derivative-Im [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Im]
lemmas sums-Re = bounded-linear.sums [OF bounded-linear-Re]
lemmas sums-Im = bounded-linear.sums [OF bounded-linear-Im]

lemma tendsto-Complex [tendsto-intros]:
  \((f \to a) F \Longrightarrow (g \to b) F \Longrightarrow ((\lambda x. \text{Complex} (f \ x) (g \ x)) \to Complex a \ b) F\)
  by (auto intro!: tendsto-intros)

lemma tendsto-complex-iff:
  \((f \to x) F \iff ((\lambda x. \text{Re} (f \ x)) \to \text{Re} x) F \land ((\lambda x. \text{Im} (f \ x)) \to \text{Im} x) F\)
proof safe
  assume \((\lambda x. \text{Re} (f \ x)) \to \text{Re} x) F \land ((\lambda x. \text{Im} (f \ x)) \to \text{Im} x) F\)
  from tendsto-Complex[OF this] show \((f \to x) F\)
    unfolding complex.collapse .
qed (auto intro: tendsto-intros)

lemma continuous-complex-iff: continuous \(F f \iff\)
  continuous \(F (\lambda x. \text{Re} (f \ x)) \land\) continuous \(F (\lambda x. \text{Im} (f \ x))\)
unfolding continuous-def tendsto-complex-iff ..

lemma has-vector-derivative-complex-iff: \((f \text{ has-vector-derivative} x) F \iff\)
  \(((\lambda x. \text{Re} (f \ x)) \text{ has-field-derivative} (\text{Re} x)) F \land\)
  \(((\lambda x. \text{Im} (f \ x)) \text{ has-field-derivative} (\text{Im} x)) F\)
unfolding has-vector-derivative-def has-field-derivative-def has-derivative-def tendsto-complex-iff
by (simp add: field-simps bounded-linear-scaleR-left bounded-linear-mult-right)

lemma has-field-derivative-Re[derivative-intros]:
  \((f \text{ has-vector-derivative} D) F \Longrightarrow ((\lambda x. \text{Re} (f \ x)) \text{ has-field-derivative} (\text{Re} D)) F\)
unfolding has-vector-derivative-complex-iff by safe

lemma has-field-derivative-Im[derivative-intros]:
  \((f \text{ has-vector-derivative} D) F \Longrightarrow ((\lambda x. \text{Im} (f \ x)) \text{ has-field-derivative} (\text{Im} D)) F\)
unfolding has-vector-derivative-complex-iff by safe

instance complex :: banach

proof
  fix \(X :: \text{nat} \Rightarrow \text{complex}\)
  assume \(X: \text{Cauchy} X\)
  then have \((\lambda n. \text{Complex} (\text{Re} (X n)) (\text{Im} (X n))) \to \text{Complex} (\lim (\lambda n. \text{Re} (X n))) (\lim (\lambda n. \text{Im} (X n)))\)
    by (intro tendsto-Complex convergent-LIMSEQ-iff THEN iffD1) Cauchy-convergent-iff[THEN iffD1] Cauchy-Re Cauchy-Im)
then show convergent X
unfolding complex.collapse by (rule convergentI)
qed

declare
DERIV-power[where 'a=complex, unfolded of-nat-def[symmetric], derivative-intras]

106.8 Complex Conjugation

primcorec cnj :: complex ⇒ complex where
Re (cnj z) = Re z
| Im (cnj z) = − Im z

lemma complex-cnj-cancel-iff [simp]: (cnj x = cnj y) = (x = y)
by (simp add: complex-eq-iff)

lemma complex-cnj-cnj [simp]: cnj (cnj z) = z
by (simp add: complex-eq-iff)

lemma complex-cnj-zero [simp]: cnj 0 = 0
by (simp add: complex-eq-iff)

lemma complex-cnj-zero-iff [iff]: (cnj z = 0) = (z = 0)
by (simp add: complex-eq-iff)

lemma complex-cnj-add [simp]: cnj (x + y) = cnj x + cnj y
by (simp add: complex-eq-iff)

lemma cnj-setsum [simp]: cnj (setsum f s) = (∑ x∈s. cnj (f x))
by (induct s rule: infinite-finite-induct) auto

lemma complex-cnj-diff [simp]: cnj (x − y) = cnj x − cnj y
by (simp add: complex-eq-iff)

lemma complex-cnj-minus [simp]: cnj (− x) = − cnj x
by (simp add: complex-eq-iff)

lemma complex-cnj-one [simp]: cnj 1 = 1
by (simp add: complex-eq-iff)

lemma complex-cnj-mult [simp]: cnj (x * y) = cnj x * cnj y
by (simp add: complex-eq-iff)

lemma cnj-setprod [simp]: cnj (setprod f s) = (∏ x∈s. cnj (f x))
by (induct s rule: infinite-finite-induct) auto

lemma complex-cnj-inverse [simp]: cnj (inverse x) = inverse (cnj x)
by (simp add: complex-eq-iff)
lemma complex-cnj-divide [simp]: cnj (x / y) = cnj x / cnj y
  by (simp add: divide-complex-def)

lemma complex-cnj-power [simp]: cnj (x ^ n) = cnj x ^ n
  by (induct n) simp-all

lemma complex-cnj-of-nat [simp]: cnj (of-nat n) = of-nat n
  by (simp add: complex-eq-iff)

lemma complex-cnj-of-int [simp]: cnj (of-int z) = of-int z
  by (simp add: complex-eq-iff)

lemma complex-cnj-numeral [simp]: cnj (numeral w) = numeral w
  by (simp add: complex-eq-iff)

lemma complex-cnj-neg-numeral [simp]: cnj (- numeral w) = - numeral w
  by (simp add: complex-eq-iff)

lemma complex-cnj-scaleR [simp]: cnj (scaleR r x) = scaleR r (cnj x)
  by (simp add: complex-eq-iff)

lemma complex-mod-cnj [simp]: cmod (cnj z) = cmod z
  by (simp add: norm-complex-def)

lemma complex-cnj-complex-of-real [simp]: cnj (of-real x) = of-real x
  by (simp add: complex-eq-iff)

lemma complex-cnj-i [simp]: cnj ii = - ii
  by (simp add: complex-eq-iff)

lemma complex-add-cnj: z + cnj z = complex-of-real (2 * Re z)
  by (simp add: complex-eq-iff)

lemma complex-diff-cnj: z - cnj z = complex-of-real (2 * Im z) + ii
  by (simp add: complex-eq-iff)

lemma complex-mult-cnj: z * cnj z = complex-of-real ((Re z)^2 + (Im z)^2)
  by (simp add: complex-eq-iff power2-eq-square)

lemma complex-mod-mult-cnj: cmod (z * cnj z) = (cmod z)^2
  by (simp add: norm-mult power2-eq-square)

lemma complex-mod-sqrt-Real-mult-cnj: cmod z = sqrt (Re (z * cnj z))
  by (simp add: norm-complex-def power2-eq-square)

lemma complex-Im-mult-cnj-zero [simp]: Im (z * cnj z) = 0
  by simp

lemma bounded-linear-cnj: bounded-linear cnj
THEORY "Complex"

using complex-cnj-add complex-cnj-scaleR
by (rule bounded-linear-intro [where K=1], simp)

lemmas tendsto-cnj [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-cnj]
lemmas isCont-cnj [simp] = bounded-linear.isCont [OF bounded-linear-cnj]
lemmas continuous-cnj [simp, continuous-intros] = bounded-linear.continuous [OF bounded-linear-cnj]
lemmas continuous-on-cnj [simp, continuous-intros] = bounded-linear.continuous-on [OF bounded-linear-cnj]
lemmas has-derivative-cnj [simp, derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-cnj]

lemma lim-cnj: \((\lambda x. \text{cnj}(f x)) \to cnj l) \iff (f \to l)\)

lemma sums-cnj: \((\lambda x. \text{cnj}(f x)) \sums cnj l) \iff (f \sums l)\)
by (simp add: sums-def lim-cnj cnj-setsum [symmetric] del: cnj-setsum)

106.9 Basic Lemmas

lemma complex-eq-0: \(z=0 \iff (\text{Re } z)^2 + (\text{Im } z)^2 = 0\)
by (metis zero-complex.sel complex-eqI sum-power2-eq-zero-iff)

lemma complex-neq-0: \(z \neq 0 \iff (\text{Re } z)^2 + (\text{Im } z)^2 > 0\)
by (metis complex-eq-0 less-numeral_extra(3) sum-power2-gt-zero-iff)

lemma complex-norm-square: \(\text{of-real} ((\text{norm } z)^2) = z \star \text{cnj } z\)
by (cases z)
  (auto simp: complex-eq-iff norm-complex-def power2-eq-square [symmetric] of-real-power [symmetric]
  simp del: of-real-power)

lemma re-complex-div-eq-0: \(\text{Re } (a / b) = 0 \iff \text{Re } (a \star \text{cnj } b) = 0\)
by (auto simp add: Re-divide)

lemma im-complex-div-eq-0: \(\text{Im } (a / b) = 0 \iff \text{Im } (a \star \text{cnj } b) = 0\)
by (auto simp add: Im-divide)

lemma complex-div-gt-0:
  \((\text{Re } (a / b) > 0 \iff \text{Re } (a \star \text{cnj } b) > 0) \land (\text{Im } (a / b) > 0 \iff \text{Im } (a \star \text{cnj } b) > 0)\)
proof cases
  assume \(b = 0\) then show \(?\)thesis by auto
next
  assume \(b \neq 0\)
  then have \(0 < (\text{Re } b)^2 + (\text{Im } b)^2\)
  by (simp add: complex-eq-iff sum-power2-gt-zero-iff)
  then show \(?\)thesis by (simp add: Re-divide Im-divide zero-less-divide-iff)
qed
lemma re-complex-div-gt-0: \( Re \left( \frac{a}{b} \right) > 0 \) \( \iff \) \( Re \left( a \cdot \text{cnj} \ b \right) > 0 \)
and
im-complex-div-gt-0: \( Im \left( \frac{a}{b} \right) > 0 \) \( \iff \) \( Im \left( a \cdot \text{cnj} \ b \right) > 0 \)
using complex-div-gt-0 by auto

lemma re-complex-div-ge-0: \( Re \left( \frac{a}{b} \right) \geq 0 \) \( \iff \) \( Re \left( a \cdot \text{cnj} \ b \right) \geq 0 \)
by (metis le-less re-complex-div-eq-0 re-complex-div-gt-0)

lemma im-complex-div-ge-0: \( Im \left( \frac{a}{b} \right) \geq 0 \) \( \iff \) \( Im \left( a \cdot \text{cnj} \ b \right) \geq 0 \)
by (metis im-complex-div-eq-0 im-complex-div-gt-0 le-less)

lemma re-complex-div-lt-0: \( Re \left( \frac{a}{b} \right) < 0 \) \( \iff \) \( Re \left( a \cdot \text{cnj} \ b \right) < 0 \)
by (metis less-asym neq-iff re-complex-div-eq-0 re-complex-div-gt-0)

lemma im-complex-div-lt-0: \( Im \left( \frac{a}{b} \right) < 0 \) \( \iff \) \( Im \left( a \cdot \text{cnj} \ b \right) < 0 \)
by (metis im-complex-div-eq-0 im-complex-div-gt-0 less-asym neq-iff)

lemma re-complex-div-le-0: \( Re \left( \frac{a}{b} \right) \leq 0 \) \( \iff \) \( Re \left( a \cdot \text{cnj} \ b \right) \leq 0 \)
by (metis not-le re-complex-div-gt-0)

lemma im-complex-div-le-0: \( Im \left( \frac{a}{b} \right) \leq 0 \) \( \iff \) \( Im \left( a \cdot \text{cnj} \ b \right) \leq 0 \)
by (metis im-complex-div-gt-0 not-le)

lemma Re-setsum[simp]: \( Re \left( \text{setsum } f \ s \right) = \left( \sum_{x \in s} Re \left( f \ x \right) \right) \)
by (induct s rule: infinite-finite-induct) auto

lemma Im-setsum[simp]: \( Im \left( \text{setsum } f \ s \right) = \left( \sum_{x \in s} Im \left( f \ x \right) \right) \)
by (induct s rule: infinite-finite-induct) auto

lemma sums-complex-iff: \( f \ x \sum x \leftrightarrow \left( \left( \lambda x. Re \left( f \ x \right) \right) \sums Re x \right) \land \left( \left( \lambda x. Im \left( f \ x \right) \right) \sums Im x \right) \)
unfolding sums-def tendsto-complex-iff Im-setsum Re-setsum ..

lemma summable-complex-iff: \( \text{summable } f \leftrightarrow \text{summable } \left( \lambda x. Re \left( f \ x \right) \right) \land \text{summable } \left( \lambda x. Im \left( f \ x \right) \right) \)
unfolding summable-def sums-complex-iff[abs-def] by (metis complex.sel)

lemma summable-complex-of-real [simp]: \( \text{summable } \left( \lambda n. \text{complex-of-real } \left( f \ n \right) \right) \leftrightarrow \text{summable } f \)
unfolding summable-complex-iff by simp

lemma summable-Re: \( \text{summable } f \implies \text{summable } \left( \lambda x. Re \left( f \ x \right) \right) \)
unfolding summable-complex-iff by blast

lemma summable-Im: \( \text{summable } f \implies \text{summable } \left( \lambda x. Im \left( f \ x \right) \right) \)
unfolding summable-complex-iff by blast

lemma complex-is-Real-iff: \( z \in \mathbb{R} \leftrightarrow Im \ z = 0 \)
by (auto simp: Reals-def complex-eq-iff)
lemma Reals-cnj-iff: \( z \in \mathbb{R} \iff \text{cnj} z = z \)
by (auto simp: complex-is-Real-iff complex-eq-iff)

lemma in-Reals-norm: \( z \in \mathbb{R} \Rightarrow \text{norm}(z) = |\text{Re} z| \)
by (simp add: complex-is-Real-iff norm-complex-def)

lemma series-comparison-complex:
fixes \( f : \mathbb{N} \Rightarrow 'a::banach \)
assumes sg: \( \text{summable } g \)
and \( \forall n. g n \in \mathbb{R} \land \text{Re } (g n) \geq 0 \)
and \( \forall n. n \geq N \Rightarrow \text{norm}(f n) \leq \text{norm}(g n) \)
shows \( \text{summable } f \)
proof –
  have \( \forall n. \text{cmod } (g n) = \text{Re } (g n) \) using assms
  by (metis abs-of-nonneg in-Reals-norm)
  show ?thesis
    apply (rule summable-comparison-test' [where \( g = \lambda n. \text{norm } (g n) \) and \( N=N \)])
    using sg
    apply (auto simp: summable-def)
    apply (rule-tac x = \( \text{Re } s \) in exI)
    apply (auto simp: g sums-Re)
    apply (metis fg g)
  done
qed

106.10 Finally! Polar Form for Complex Numbers

106.10.1 \( \cos \theta + i \sin \theta \)

primcorec cis :: real \( \Rightarrow \) complex where
  \( \text{Re } (\text{cis } a) = \cos a \)
| \( \text{Im } (\text{cis } a) = \sin a \)

lemma cis-zero [simp]: \( \text{cis } 0 = 1 \)
by (simp add: complex-eq-iff)

lemma norm-cis [simp]: \( \text{norm } (\text{cis } a) = 1 \)
by (simp add: norm-complex-def)

lemma sgn-cis [simp]: \( \text{sgn } (\text{cis } a) = \text{cis } a \)
by (simp add: sgn-div-norm)

lemma cis-neq-zero [simp]: \( \text{cis } a \neq 0 \)
by (metis norm-cis norm-zero-neq-one)

lemma cis-mult: \( \text{cis } a \ast \text{cis } b = \text{cis } (a + b) \)
by (simp add: complex-eq-iff cos-add sin-add)

lemma DeMoivre: \( (\text{cis } a) \ast ^n = \text{cis } (\text{real } n \ast a) \)
by (induct n, simp-all add: real-of-nat Suc algebra-simps cis-mult)

lemma cis-inverse [simp]: inverse(cis a) = cis (−a)
  by (simp add: complex-eq-iff)

lemma cis-divide: cis a / cis b = cis (a − b)
  by (simp add: divide-complex-def cis-mult)

lemma cos-n-Re-cis-pow-n: cos (real n * a) = Re(cis a ^ n)
  by (auto simp add: DeMoivre)

lemma sin-n-Im-cis-pow-n: sin (real n * a) = Im(cis a ^ n)
  by (auto simp add: DeMoivre)

lemma cis-pi: cis pi = −1
  by (simp add: complex-eq-iff)

106.10.2  r(cos θ + i sin θ)

definition rcis :: real ⇒ real ⇒ complex where
  rcis r a = complex-of-real r * cis a

lemma Re-rcis [simp]: Re(rcis r a) = r * cos a
  by (simp add: rcis-def)

lemma Im-rcis [simp]: Im(rcis r a) = r * sin a
  by (simp add: rcis-def)

lemma rcis-Ex: ∃ r a. z = rcis r a
  by (simp add: complex-eq-iff polar-Ex)

lemma complex-mod-rcis [simp]: cmod(rcis r a) = abs r
  by (simp add: rcis-def norm-mult)

lemma cis-rcis-eq: cis a = rcis 1 a
  by (simp add: rcis-def)

lemma rcis-mult: rcis r1 a * rcis r2 b = rcis (r1 * r2) (a + b)
  by (simp add: rcis-def cis-mult)

lemma rcis-zero-mod [simp]: rcis 0 a = 0
  by (simp add: rcis-def)

lemma rcis-zero-arg [simp]: rcis r 0 = complex-of-real r
  by (simp add: rcis-def)

lemma rcis-eq-zero_iff [simp]: rcis r a = 0 ↔ r = 0
  by (simp add: rcis-def)
lemma DeMoivre2: \((rcis\ r\ a) \ ^ n = rcis\ (r \ ^ n)\ \text{real}\ (n \ast a)\)
by (simp add: rcis-def power-mult-distrib DeMoivre)

lemma rcis-inverse: \(\text{inverse}(rcis\ r\ a) = rcis\ (1/r)\ (-a)\)
by (simp add: divide-inverse rcis-def)

lemma rcis-divide: \(rcis\ r1\ a / rcis\ r2\ b = rcis\ (r1/r2)\ (a - b)\)
by (simp add: rcis-def cis-divide [symmetric])

106.10.3 Complex exponential

abbreviation \(expi :\ complex \Rightarrow complex\)
where \(expi \equiv\ exp\)

lemma cis-conv-exp: \(cis\ b = exp\ (i \ast b)\)
proof –
{ fix n :: nat
have \(i \ ^ n = fact\ n \ast (cos-coeff\ n + i \ast sin-coeff\ n)\)
  by (induct n)
  (simp-all add: sin-coeff-Suc cos-coeff-Suc complex-eq-iff Re-divide Im-divide field-simps
    power2-eq-square real-of-nat-Suc add-nonneg-eq-0-iff
    real-of-nat-def [symmetric])
then have \((i \ast complex-of-real b) \ ^ n / R\ fact\ n = of-real\ (cos-coeff\ n \ast b^n) + i \ast of-real\ (sin-coeff\ n \ast b^n)\)
  by (simp add: field-simps)
then show \(?thesis\)
  by (auto simp add: cis.ctr exp-def simp del: of-real-mult
    intro!: sums-unique sums-add sums-mult sums-of-real sin-converges
    cos-converges)
qed

lemma expi-def: \(expi\ z = exp\ (Re\ z) \ast cis\ (Im\ z)\)
unfolding cis-conv-exp exp-of-real [symmetric] mult-exp-exp by (cases z) simp

lemma Re-exp: \(Re\ (expi\ z) = exp\ (Re\ z) \ast cos\ (Im\ z)\)
unfolding expi-def by simp

lemma Im-exp: \(Im\ (expi\ z) = exp\ (Re\ z) \ast sin\ (Im\ z)\)
unfolding expi-def by simp

lemma complex-expi-Ex: \(\exists\ a\ r.\ z = complex-of-real\ r \ast expi\ a\)
apply (insert rcis-Ex [of z])
apply (auto simp add: expi-def rcis-def mult.assoc [symmetric])
apply (rule-tac x = ii \ast complex-of-real a in exI, auto)
done

lemma expi-two-pi-i [simp]: \(expi((2::complex) \ast complex-of-real\ pi \ast ii) = 1\)
by (simp add: expi-def complex-eq-iff)
106.10.4 Complex argument

definition arg :: complex ⇒ real where

arg z = (if z = 0 then 0 else (SOME a. sgn z = cis a ∧ −pi < a ∧ a ≤ pi))

lemma arg-zero: arg 0 = 0

by (simp add: arg-def)

lemma arg-unique:

assumes sgn z = cis x and −pi < x and x ≤ pi

shows arg z = x

proof –

from assms have z ≠ 0 by auto

have (SOME a. sgn z = cis a ∧ −pi < a ∧ a ≤ pi) = x

proof

fix a def d ≡ a − x

assume a: sgn z = cis a ∧ −pi < a ∧ a ≤ pi

from a assms have − (2*pi) < d ∧ d < 2*pi

unfolding d-def by simp

moreover from a assms have cos a = cos x and sin a = sin x

by (simp-all add: complex-eq-iff)

hence cos: cos d = 1 unfolding d-def cos-diff by simp

moreover from cos have sin d = 0 by (rule cos-one-sin-zero)

ultimately have d = 0

unfolding sin-zero-iff even-mult-two-ex

by (auto simp add: numeral-2-eq-2 less-Suc-eq)

thus a = x unfolding d-def by simp

qed (simp add: assms del: Re-sgn Im-sgn)

with (z ≠ 0) show arg z = x

unfolding arg-def by simp

qed

lemma arg-correct:

assumes z ≠ 0 shows sgn z = cis (arg z) ∧ −pi < arg z ∧ arg z ≤ pi

proof (simp add: arg-def assms, rule someI-ex)

obtain r a where z = rcis r a using rcis-Ex by fast

with assms have r ≠ 0 by auto

def b ≡ if 0 < r then a else a + pi

have b: sgn z = cis b

unfolding z b-def rcis-def using (r ≠ 0)

by (simp add: of-real-def sgn-scaleR sgn-if complex-eq-iff)

have cis-2pi-nat: ∃n. cis (2 * pi * real-of-nat n) = 1

by (induct-tac n) (simp-all add: distrib-left cis-mult [symmetric] complex-eq-iff)

have cis-2pi-int: ∃x. cis (2 * pi * real-of-int x) = 1

by (case-tac x rule: int-diff-cases)

(simp add: right-diff-distrib cis-divide [symmetric] cis-2pi-nat)

def c ≡ b − 2*pi * of-int [(b − pi) / (2*pi)]

have sgn z = cis c

unfolding b c-def

by (simp add: cis-divide [symmetric] cis-2pi-int)
moreover have $-\pi < c \land c \leq \pi$

using ceiling-correct [of $(b - \pi) / (2*\pi)$]

by (simp add: c-def less-divide-eq divide-le-eq algebra-simps)

ultimately show $\exists a. \ sgn z = \text{cis} a \land -\pi < a \land a \leq \pi$ by fast

qed

lemma arg-bounded: $-\pi < \text{arg} z \land \text{arg} z \leq \pi$

by (cases $z = 0$) (simp-all add: arg-zero arg-correct)

lemma cis-arg: $z \neq 0 \implies \text{cis} (\text{arg} z) = \text{sgn} z$

by (simp add: arg-correct)

lemma rcis-cmod-arg: $\text{rcis} (\text{cmod} z) (\text{arg} z) = z$

by (cases $z = 0$) (simp-all add: rcis-def cis-arg sgn-div-norm of-real-def)

lemma cos-arg-i-mult-zero [simp]: $y \neq 0 \implies \Re y = 0 \implies 
\cos (\text{arg} y) = 0$

using cis-arg [of $y$] by (simp add: complex-eq-iff)

106.11 Square root of complex numbers

primcorec csqrt :: complex $\Rightarrow$ complex where

$\Re (\text{csqrt} z) = \sqrt{((\text{cmod} z + \Re z) / 2)}$

$\Im (\text{csqrt} z) = (\text{if } \Im z = 0 \text{ then } 1 \text{ else } \text{sgn} (\Im z)) \ast \sqrt{((\text{cmod} z - \Re z) / 2)}$

lemma csqrt-of-real-nonneg [simp]: $\Im x = 0 \implies \Re x \geq 0 \implies \text{csqrt} x = \sqrt{\Re x}$

by (simp add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re)

lemma csqrt-of-real-nonpos [simp]: $\Im x = 0 \implies \Re x \leq 0 \implies \text{csqrt} x = i \ast \sqrt{\Re x}$

by (simp add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re)

lemma csqrt-0 [simp]: $\text{csqrt} 0 = 0$

by simp

lemma csqrt-1 [simp]: $\text{csqrt} 1 = 1$

by simp

lemma csqrt-ii [simp]: $\text{csqrt} i = (1 + i) / \sqrt{2}$

by (simp add: complex-eq-iff Re-divide Im-divide real-sqrt-divide real-div-sqrt)

lemma power2-csqrt [algebra]: $(\text{csqrt} z)^2 = z$

proof cases

assume $\Im z = 0$ then show $\text{thesis}$

using real-sqrt-pow2 [of $\Re z$] real-sqrt-pow2 [of $-\Re z$]

by (cases 0::real $\Re z$ rule: linorder-cases)

(simp-all add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re)

next

assume $\Im z \neq 0$
moreover
have \( \text{cmod } z \ast \text{cmod } z - \text{Re } z \ast \text{Re } z = \text{Im } z \ast \text{Im } z \)
  by (simp add: norm-complex-def power2-eq-square)
moreover
have \( |\text{Re } z| \leq \text{cmod } z \)
  by (simp add: norm-complex-def)
ultimately show \(?thesis\)
  by (simp add: Re-power2 Im-power2 complex-eq-iff real-sgn-eq
    field-simps real-sqrt-mult[ symmetric] real-sqrt-divide)
qed

lemma \( \text{csqrt-eq-0} [simp] \): \( \text{csqrt } z = 0 \longleftrightarrow z = 0 \)
by auto (metis power2-csqrt power-eq-0-iff)

lemma \( \text{csqrt-eq-1} [simp] \): \( \text{csqrt } z = 1 \longleftrightarrow z = 1 \)
by auto (metis power2-csqrt power2-eq-1-iff)

lemma \( \text{csqrt-principal} \): \( 0 < \text{Re } (\text{csqrt } z) \lor \text{Re } (\text{csqrt } z) = 0 \land 0 \leq \text{Im } (\text{csqrt } z) \)
by (auto simp add: not-less cmod-plus-Re-le-0-iff Im-eq-0)

lemma \( \text{Re-csqrt} \)
  by (metis csqrt-principal le-less)

lemma \( \text{csqrt-square} \):
  assumes \( 0 < \text{Re } b \lor (\text{Re } b = 0 \land 0 \leq \text{Im } b) \)
  shows \( \text{csqrt } (b \ast 2) = b \)
proof
  have \( \text{csqrt } (b \ast 2) = b \lor \text{csqrt } (b \ast 2) = -b \)
    unfolding power2-eq-iff[symmetric] by (simp add: power2-csqrt)
  moreover have \( \text{csqrt } (b \ast 2) \neq -b \lor b = 0 \)
    using \( \text{csqrt-principal[of } b \ast 2) \) assms by (intro disjCI notI) (auto simp: complex-eq-iff)
  ultimately show \(?thesis\)
    by auto
qed

lemma \( \text{csqrt-minus} [simp] \):
  assumes \( \text{Im } x < 0 \lor (\text{Im } x = 0 \land 0 \leq \text{Re } x) \)
  shows \( \text{csqrt } (- x) = i \ast \text{csqrt } x \)
proof
  have \( \text{Im } (\text{csqrt } x) \leq 0 \)
    using assms by (auto simp add: cmod-eq-Re mult-le-0-iff field-simps complex-Re-le-cmod)
  then show \( 0 < \text{Re } (i \ast \text{csqrt } x) \lor \text{Re } (i \ast \text{csqrt } x) = 0 \land 0 \leq \text{Im } (i \ast \text{csqrt } x) \)
    by (auto simp add: Re-csqrt simp del: csqrt.simps)
qed
also have \( (i \ast \text{csqrt } x) \ast 2 = - x \)
  by (simp add: power2-csqrt power-mult-distrib)
finally show thesis.

qed

Legacy theorem names
lemmas expand-complex-eq = complex-eq-iff
lemmas complex-Re-Im-cancel-iff = complex-eq-iff
lemmas complex-equality = complex-eqI
lemmas cnj-def = norm-complex-def
lemmas complex-norm-def = norm-complex-def
lemmas complex-divide-def = divide-complex-def

lemma legacy-Complex-simps:
  shows Complex-eq-0: Complex a b = 0 0 = 0 0 = 0
    and complex-add: Complex a b + Complex c d = Complex (a + c) (b + d)
    and complex-minus: - (Complex a b) = Complex (- a) (- b)
    and complex-diff: Complex a b - Complex c d = Complex (a - c) (b - d)
    and Complex-eq-1: Complex a b = 1 1 = 1 0 = 0
    and Complex-eq-neg-1: Complex a b = - 1 1 = - 1 0 = 0
    and complex-mult: Complex a b * Complex c d = Complex (a * c - b * d) (a * d + b)
      and complex-inverse: inverse (Complex a b) = Complex (a / (a^2 + b^2)) (- b / (a^2 + b^2))
      and Complex-eq-numeral: Complex a b = numeral w 0 = 0
        and Complex-eq-neg-numeral: Complex a b = - numeral w 0 = 0
        and Complex-scaleR: scaleR r (Complex a b) = Complex (r * a) (r * b)
        and Complex-eq-i: (Complex x y = ii) = (x = 0 y = 1)
        and i-mult-Complex: ii * Complex a b = Complex (- b) a
        and Complex-mult-i: Complex a b * ii = Complex (- b) a
        and i-complex-of-real: ii * complex-of-real r = Complex 0 r
        and complex-of-real-i: complex-of-real r * ii = Complex 0 r
        and Complex-add-complex-of-real: Complex x y + complex-of-real r = Complex (x + r) y
          and complex-of-real-add-Complex: complex-of-real r + Complex x y = Complex (r + x) y
          and Complex-mult-complex-of-real: Complex x y * complex-of-real r = Complex (x * r) (y * r)
          and complex-of-real-mult-Complex: complex-of-real r * Complex x y = Complex (r * x) (r * y)
          and complex-eq-cancel-iff2: (Complex x y = complex-of-real xa) = (x = xa y = 0)
          and complex-cnj: cnj (Complex a b) = Complex a (- b)
          and Complex-setsum: setsum (%x. Complex (f x) 0) s = Complex (setsum f s) 0
            and Complex-setsum: Complex (setsum f s) 0 = setsum (%x. Complex (f x) 0) s
            and complex-of-real-def: complex-of-real r = Complex r 0
            and complex-norm: cnj (Complex x y) = sqrt (x^2 + y^2)
by (simp-all add: norm-complex-def field-simps complex-eq-iff Re-divide Im-divide del: Complex-eq)

lemma Complex-in-Reals: Complex x 0 ∈ R
  by (metis Reals-of-real complex-of-real-def)
end

107 MacLaurin: MacLaurin Series

theory MacLaurin
  imports Transcendental
begin

107.1 Maclaurin’s Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it’s been broken down into lemmas.

lemma Maclaurin-lemma:
  0 < h ==>
  ∃ B. f h = (∑ m<n. (j m / real (fact m)) * (hˆm)) +
  (B * ((hˆn) / (fact n)))
  by (rule exI [where x = (f h - (∑ m<n. (j m / real (fact m)) * hˆm)) * real(fact n) / (hˆn)]) simp

lemma eq-diff-eq': (x = y - z) = (y = x + (z::real))
  by arith

lemma fact-diff-Suc [rule-format]:
  n < Suc m ==>
  fact (Suc m - n) = (Suc m - n) * fact (m - n)
  by (subst fact-reduce-nat, auto)

lemma Maclaurin-lemma2:
  fixes B
  assumes DERIV : ∀ m t. m < n ∧ 0 ≤ t ∧ t ≤ h ---> DERIV (diff m) t :> diff (Suc m) t
  and INIT : n = Suc k
  defines difg ≡ (λm t. diff m t - (∑ p<n-m. diff (m + p) 0 / real (fact p)) + t ^ p) +
  B * (t - (n - Suc 0) / real (fact (n - m))))
  (is difg ≡ (λm t. diff m t - (?difg m t)))
  shows ∀ m t. m < n & 0 ≤ t & t ≤ h ---> DERIV (difg m) t :> difg (Suc m) t
  proof (rule allI impI)+
  fix m t assume INIT2: m < n & 0 ≤ t & t ≤ h
  have DERIV (difg m) t :> diff (Suc m) t -
    (∑ x<n - m. real x * t ^ (x - Suc 0) * diff (m + x) 0 / real (fact x)) +
real \((n - m) \cdot t \cdot (n - \text{Suc } m) \cdot B / \text{real } (\text{fact } (n - m))\) unfolding difg-def
by (auto intro!: derivative-eq-intros DERIV[rule-format, OF INIT2]
simp: real-of-nat-def[symmetric])

moreover
from INIT2 have intvl: \([-<n - m] = \text{insert } 0 (\text{Suc } \{-<n - \text{Suc } m\})\) and
0 < n - m
unfolding atLeast0LessThan[symmetric] by auto
have \((\sum x < n - m. \text{real } x \cdot t \cdot (x - \text{Suc } 0) \cdot \text{diff } (m + x) \cdot 0 / \text{real } (\text{fact } x)) =
(\sum x < n - \text{Suc } m. \text{real } (\text{Suc } x) \cdot t \cdot x \cdot \text{diff } (\text{Suc } m + x) \cdot 0 / \text{real } (\text{fact } (\text{Suc } x)))\)
unfolding intvl atLeast0LessThan by (subst setsum.insert) (auto simp: setsum.reindex)
moreover
have fact-neq-0: \(\forall x::\text{nat}. \text{real } (\text{fact } x) + \text{real } x \cdot \text{real } (\text{fact } x) \neq 0\)
by (metis fact-at-zero-nat not-add-less1 real-of-nat-add real-of-nat-mult real-of-nat-zero-iiff)
have \(\forall x. \text{real } (\text{Suc } x) \cdot t \cdot x \cdot \text{diff } (\text{Suc } m + x) \cdot 0 / \text{real } (\text{fact } (\text{Suc } x)) =
\text{diff } (\text{Suc } m + x) \cdot 0 \cdot t \cdot x / \text{real } (\text{fact } x)\)
by (auto simp: field-simps real-of-nat-Suc fact-neq-0 intro: nonzero-divide-eq-eq[THEN iffD2])
moreover
have real \((n - m) \cdot t \cdot (n - \text{Suc } m) \cdot B / \text{real } (\text{fact } (n - m)) =
B \cdot (t \cdot (n - \text{Suc } m) \cdot \text{real } (\text{fact } (n - \text{Suc } m)))\)
using \(\{0 < n - m\}\) by (simp add: fact-reduce-nat)
ultimately show DERIV \((\text{difg } m) t \Rightarrow \text{difg } (\text{Suc } m) t\)
unfolding difg-def by simp

lemma Maclaurin:
assumes h: \(\text{0 < h}\)
assumes n: \(\text{0 < n}\)
assumes diff-0: \(\text{diff } 0 = f\)
assumes diff-Suc: \(\forall m. t. m < n \& 0 \leq t \& t \leq h \Rightarrow \text{DERIV } (\text{difg } m) t \Rightarrow \text{difg } (\text{Suc } m) t\)
shows \(\exists t. \text{0 < t \& t < h \& \ f h =}
\text{setsum } (\%m. (\text{diff } m \cdot 0 / \text{real } (\text{fact } m)) \cdot h \cdot m \cdot \{\text{-<n}\} +
(\text{diff } n \cdot t / \text{real } (\text{fact } n)) \cdot h \cdot n\)

proof
from n obtain m where m: \(n = \text{Suc } m\)
by (cases n) (simp add: n)
obtain B where \(f h =
(\sum m < n. \text{diff } m \cdot 0 / \text{real } (\text{fact } m)) \cdot h \cdot m +
B \cdot (h \cdot n / \text{real } (\text{fact } n))\)
using Maclaurin-lemma [OF h] ..
def g \equiv (\lambda t. f t -
(\text{setsum } (\lambda m. (\text{diff } m \cdot 0 / \text{real } (\text{fact } m)) \cdot t \cdot m) \cdot \{\text{-<n}\})\)
+ (B * (t \cdot n / \text{real(fact } n))))}

have \(g^2\): \(g 0 = 0\) \& \(g h = 0\)
by (simp add: m f-h g-def lessThan-Suc-eq-0 image-iff diff-0 setsum.reindex)

def \(\text{difg} \equiv (\%m. t. \text{diff } m t - (\text{setsum } (\%p. (\text{diff } (m + p) 0 / \text{real } ( \text{fact } p)) * (t ^ p) \{..<n-m\}) \}} + (B * ((t ^ (n - m)) / \text{real } ( \text{fact } (n - m)))))))\)

have \(\text{difg-0}: \text{difg } 0 = g\)
unfolding \(\text{difg-def } g\)-def by (simp add: diff-0)

have \(\text{difg-Suc}: \forall (m::\text{nat } t::\text{real} ) \quad m < n \land (0::\text{real} ) \leq t \land t \leq h \rightarrow \text{DERIV } (\text{difg } m) t :: \text{difg } (\text{Suc } m) t\)
using \(\text{dif-Suc } m\) unfolding \(\text{difg-def } \)by (rule Macaulay-lemma2)

have \(\text{difg-eq-0}: \forall m<n. \text{difg } m 0 = 0\)
by (auto simp: difg-def m Suc-diff-le lessThan-Suc-eq-insert-0 image-iff setsum.reindex)

have \(\text{isCont-difg} : \forall m x. [m < n; 0 \leq x; x \leq h] \Rightarrow \text{isCont } (\text{difg } m) x\)
by (rule DERIV-isCont [OF difg-Suc [rule-format]]) simp

have \(\text{differentiable-difg}: \forall m x. [m < n; 0 \leq x; x \leq h] \Rightarrow \text{difg } m \text{ differentiable } (\text{at } x)\)
by (rule differentiableI [OF difg-Suc [rule-format]]) simp

have \(\text{difg-Suc-eq-0}: \forall m t. [m < n; 0 \leq t; t \leq h; \text{DERIV } (\text{difg } m) t :: 0]\)
\Rightarrow \text{difg } (\text{Suc } m) t = 0
by (rule DERIV-unique [OF difg-Suc [rule-format]]) simp

have \(m < n\) using \(m\) by simp

have \(\exists t. 0 < t \land t < h \land \text{DERIV } (\text{difg } m) t :: 0\)
using \((m < n)\)
proof (induct \(m\))
  case 0
  show ?case
  proof (rule Rolle)
    show \(0 < h\) by fact
    show \(\text{difg } 0 0 = \text{difg } 0 h\) by (simp add: difg-0 g2)
    show \(\forall x. 0 \leq x \land x \leq h \rightarrow \text{isCont } (\text{difg } (0::\text{nat})) x\)
      by (simp add: isCont-difg n)
    show \(\forall x. 0 < x \land x < h \rightarrow \text{difg } (0::\text{nat}) \text{ differentiable } (\text{at } x)\)
      by (simp add: differentiable-difg n)
  qed
next
  case (Suc \(m\))
  hence \(\exists t. 0 < t \land t < h \land \text{DERIV } (\text{difg } m) t :: 0\) by simp
then obtain $t$ where $0 < t < h$ \( \text{DERIV} \ (\text{difg} \ m') \ t :> 0 \) by fast

have $\exists t'. \ 0 < t' \land t' < t \land \text{DERIV} \ (\text{difg} \ (\text{Suc} \ m')) \ t' :> 0$

proof (rule Rolle)
  
  show $0 < t$ by fact
  
  show $\text{difg} \ (\text{Suc} \ m') \ 0 = \text{difg} \ (\text{Suc} \ m') \ t$
    using $t \langle \text{Suc} \ m' < n \rangle$ by (simp add: difg-Suc-eq-0 difg-eq-0)
  
  show $\forall x. \ 0 \leq x \land x \leq t \longrightarrow \text{isCont} \ (\text{difg} \ (\text{Suc} \ m')) \ x$
    using $t < h$ $\langle \text{Suc} \ m' < n \rangle$ by (simp add: isCont-difg)
  
  show $\forall x. \ 0 < x \land x < t \longrightarrow \text{difg} \ (\text{Suc} \ m')$ differentiable (at $x$)
    using $t < h$ $\langle \text{Suc} \ m' < n \rangle$ by (simp add: differentiable-difg)

qed

thus $?case$
  
  using $(t < h)$ by auto

qed

then obtain $t$ where $0 < t < h$ \( \text{DERIV} \ (\text{difg} \ m) \ t :> 0 \) by fast

hence $\text{difg} \ (\text{Suc} \ m) \ t = 0$
  
  using $\langle m < n \rangle$ by (simp add: difg-Suc-eq-0)

proof
  
  show $?thesis$
    using $\langle t < h \rangle$ by auto

qed

lemma Maclaurin-objl:

\[ 0 < h \land n > 0 \land \text{difg} \ 0 = f \land (\forall m. \ m < n \land 0 \leq t \land t \leq h \longrightarrow \text{DERIV} \ (\text{diff} \ m) \ t :> \text{diff} \ (\text{Suc} \ m) \ t) \]

\[ \longrightarrow (\exists t. \ 0 < t \land t < h \land \begin{array}{l}
    \text{f} \ h = \langle \sum \ m < n. \ \text{diff} \ m \ 0 / \ \text{real} \ (\text{fact} \ m) \ast h \ ^{\ ^{*}} \ m \rangle + \\
    \text{diff} \ m \ t / \ \text{real} \ (\text{fact} \ n) \ast h \ ^{\ ^{*}} \ n
  \end{array}
\]

by (blast intro: Maclaurin)

lemma Maclaurin2:

assumes \( \text{INIT1} \): $0 < h$ and \( \text{INIT2} \): $\text{diff} \ 0 = f$

and $\text{DERIV} \ : \ \forall m \ t.$

$m < n \land 0 \leq t \land t \leq h \longrightarrow \text{DERIV} \ (\text{diff} \ m) \ t :> \text{diff} \ (\text{Suc} \ m) \ t$

shows $\exists t. \ 0 < t \land t \leq h \land \text{f} \ h =$

\[ \langle \sum \ m < n. \ \text{diff} \ m \ 0 / \ \text{real} \ (\text{fact} \ m) \ast h \ ^{\ ^{*}} \ m \rangle + \\
    \text{diff} \ m \ t / \ \text{real} \ (\text{fact} \ n) \ast h \ ^{\ ^{*}} \ n
\]

proof (cases $n$)
case 0 with INIT1 INIT2 show thesis by fastforce
next
case Suc
hence n > 0 by simp
from INIT1 this INIT2 DERIV have \exists t > 0. t < h \land
f h =
(\sum m < n. \text{diff} m 0 / \text{real \ (fact m)} \ast \text{h} ^ m) + \text{diff} n t / \text{real \ (fact n)} \ast \text{h} ^ n
by (rule Maclaurin)
thus thesis by fastforce
qed

lemma Maclaurin2-objl:
0 < h \& \text{diff} 0 = f \&
(\forall m \cdot
m < n \& 0 \leq t \& t \leq h \rightarrow \text{DERIV} \ (\text{diff} m) \ t := \text{diff} \ (\text{Suc} m) \ t)
\rightarrow (\exists t. 0 < t \&
\ t \leq h \&
\ f h =
(\sum m < n. \text{diff} m 0 / \text{real \ (fact m)} \ast \text{h} ^ m) +
\text{diff} n t / \text{real \ (fact n)} \ast \text{h} ^ n)
by (blast intro: Maclaurin2)

lemma Maclaurin-minus:
assumes h < 0 0 < n \text{diff} 0 = f
and \text{DERIV}: \forall m. m < n \& h \leq t \& t \leq 0 \rightarrow \text{DERIV} \ (\text{diff} m) \ t := \text{diff} \ (\text{Suc} m) \ t
shows \exists t. h < t \& t < 0 \&
f h = (\sum m < n. \text{diff} m 0 / \text{real \ (fact m)} \ast \text{h} ^ m) +
\text{diff} n t / \text{real \ (fact n)} \ast \text{h} ^ n
proof

Transform ABL' into derivative-intros format.

note \text{DERIV'} = \text{DERIV-chain}[OF - \text{DERIV}[rule-format], THEN DERIV-cong]
from assms
have \exists t > 0, t < - h \land
f (- (- h)) =
(\sum m < n.
(- 1) ^ m \ast \text{diff} m (- 0) / \text{real \ (fact m)} \ast (- h) ^ m) +
(- 1) ^ n \ast \text{diff} n (- t) / \text{real \ (fact n)} \ast (- h) ^ n
by (intro Maclaurin) (auto intro!: derivative-eq-intros DERIV')
then guess t ..
moreover
have -1 \ast n \ast \text{diff} n (- t) \ast (- h) \ast n / \text{real \ (fact n)} = \text{diff} n (- t) \ast h \ast n / \text{real \ (fact n)}
by (auto simp add: power-mult-distrib[symmetric])
moreover
have (\sum m < n. -1 \ast m \ast \text{diff} m 0 \ast (- h) \ast m / \text{real \ (fact m)}) = (\sum m < n. \text{diff} m 0 \ast h \ast m / \text{real \ (fact m)})
by (auto intro: setsum.cong simp add: power-mult-distrib[symmetric])
ultimately have \( h < - t \land \\
- t < 0 \land \\
f h = \\
(\sum_{m<n} \text{diff } m \ 0 / \text{real } (\text{fact } m) * h ^ m) + \text{diff } n \ (- t) / \text{real } (\text{fact } n) * h ^ n \\
\) by auto \\
thus \(?thesis ..
qed

lemma Maclaurin-minus-objl:
\( h < 0 \land n > 0 \land \text{diff } 0 = f \land \\
\forall m \ t. \\
m < n \land h < t \land t \leq 0 \rightarrow \text{DERIV } (\text{diff } m \ t :> \text{diff } (\text{Suc } m \ t)) \\
\rightarrow \ (\exists t. h < t \land \\
t < 0 \land \\
f h = \\
(\sum_{m<n} \text{diff } m \ 0 / \text{real } (\text{fact } m) * h ^ m) + \\
\text{diff } n \ t / \text{real } (\text{fact } n) * h ^ n \\
) by (blast intro: Maclaurin-minus)

107.2 More Convenient ”Bidirectional” Version.

lemma Maclaurin-bi-le-lemma [rule-format]:
\( n > 0 \rightarrow \\
\text{diff } 0 \ 0 = \\
(\sum_{m<n} \text{diff } m \ 0 * 0 ^ m / \text{real } (\text{fact } m)) + \\
\text{diff } n \ 0 * 0 ^ n / \text{real } (\text{fact } n) \\
) by (induct n) auto

lemma Maclaurin-bi-le:
assumes \( \text{diff } 0 = f \land \\
\text{DERIV } : \forall m \ t. \ m < n \land \text{abs } t \leq \text{abs } x \rightarrow \text{DERIV } (\text{diff } m \ t :> \text{diff } (\text{Suc } m \ t) \\
\text{shows } \exists t. \ \text{abs } t \leq \text{abs } x \land \\
f x = \\
(\sum_{m<n} \text{diff } m \ 0 / \text{real } (\text{fact } m) * x ^ m) + \\
\text{diff } n \ t / \text{real } (\text{fact } n) * x ^ n \ (\text{is } \exists t. - \land f x = \text{?f x t) \\
proof cases \\
assume n = 0 with \langle \text{diff } 0 = f \rangle \ show ?thesis by force \\
next \\
assume n \neq 0 \\
show ?thesis \\
proof (cases rule: linorder-cases) \\
assume x = 0 with \langle n \neq 0 \rangle \ (\text{diff } 0 = f ; \text{DERIV} \\
have \langle 0 \rangle \leq \langle x \rangle \land f x = ?f x 0 \ by (auto simp add: Maclaurin-bi-le-lemma) \\
thus ?thesis .. \\
next \\
assume x < 0 \\
with \langle n \neq 0 \rangle \ \text{DERIV} \)
lemma Maclaurin-all-lt:
assumes INIT1: diff 0 = f and INIT2: 0 < n and INIT3: x ≠ 0
and DERIV: ∀ m x. DERIV (diff m) x := diff(Suc m) x
shows ∃ t. 0 < abs t & abs t < abs x & f x =
(∑ m<n. (diff m 0 / real (fact m)) * x ^ m) +
(diff n t / real (fact n)) * x ^ n (is ∃ t. - | - & f x = ?f x t)
proof (cases rule: linorder-cases)
assume x = 0 with INIT3 show ?thesis..
next
assume x < 0
with assms have ∃ t>x. t < 0 ∧ f x = ?f x t by (intro Maclaurin-minus) auto
then guess t ..
with (x < 0) have 0 < |t| & |t| < |x| ∧ f x = ?f x t by simp
thus ?thesis ..
next
assume x > 0
with assms have ∃ t>0. t < x ∧ f x = ?f x t by (intro Maclaurin) auto
then guess t ..
with (x > 0) have 0 < |t| & |t| < |x| ∧ f x = ?f x t by simp
thus ?thesis ..
qed

diff 0 = f /
diff 0 = f /
(∀ m x. DERIV (diff m) x := diff(Suc m) x) /
(x ≠ 0 & n > 0)
--- (exists t. 0 < abs t & abs t < abs x &
f x = (∑ m<n. (diff m 0 / real (fact m)) * x ^ m) +
(diff n t / real (fact n)) * x ^ n)
by (blast intro: Maclaurin-all-lt)
THEORY "MacLaurin"

\[
\sum_{m<n} \left( \frac{\text{diff } m (0::\text{real})}{\text{real}(\text{fact } m)} \right) \cdot x^m = \text{diff } 0 0
\]

by (induct n, auto)

**lemma** Maclaurin-all-le:
assumes INIT: \( \text{diff } 0 = f \)
and DERIV: \( \forall m. \text{DERIV } (\text{diff } m) \cdot x = \text{diff } (\text{Suc } m) \cdot x \)
shows \( \exists t. \text{abs } t \leq \text{abs } x \& f x = \text{diff } 0 0 \)

**proof cases**
assume \( n = 0 \) with INIT show \(?thesis by force
next
assume \( n \neq 0 \)
show \(?thesis

**proof cases**
assume \( x = 0 \)
with \( n \neq 0 \) have \( \sum_{m<n} \left( \frac{\text{diff } m 0}{\text{real}(\text{fact } m)} \right) \cdot x^m = \text{diff } 0 0 \)
by (intro Maclaurin-zero) auto

with INIT \( x = 0 \) \( n \neq 0 \) have \( |0| \leq |x| \& f x = ?f x 0 \) by force
thus \(?thesis ..
next
assume \( x \neq 0 \)
with \( n \neq 0 \) DERIV have \( \exists t. 0 < |t| \& |t| < |x| \& f x = ?f x t \)
by (intro Maclaurin-all-lt) auto
then guess \( t \)

hence \( |t| \leq |x| \& f x = ?f x t \) by simp
thus \(?thesis ..
qed

**lemma** Maclaurin-all-le-objl: \( \text{diff } 0 = f \&
(\forall m. \text{DERIV } (\text{diff } m) \cdot x = \text{diff } (\text{Suc } m) \cdot x)
\implies (\exists t. \text{abs } t \leq \text{abs } x \&
\quad f x = \sum_{m<n} \left( \frac{\text{diff } m 0}{\text{real}(\text{fact } m)} \right) \cdot x^m + \text{diff } n t / \text{real}(\text{fact } n) \cdot x^n )
\)
by (blast intro: Maclaurin-all-le)

107.3 Version for Exponential Function

**lemma** Maclaurin-exp-lt: \[ x \sim 0; \ n > 0 \]
\[\implies (\exists t. 0 < \text{abs } t \& \text{abs } t < \text{abs } x \&
\quad \exp x = \sum_{m<n} \left( \frac{\exp m 0}{\text{real}(\text{fact } m)} \right) \cdot x^m + \exp t / \text{real}(\text{fact } n) \cdot x^n )
\)
by (cut-tac diff = %n. \exp and \( f = \exp \) and \( x = x \) and \( n = n \) in Maclaurin-all-lt-objl, auto)
lemma Maclaurin-exp-le:

$$\exists t. \text{abs } t \leq \text{abs } x \&$$

$$\exp x = (\sum_{m<n} (x \cdot m) / \text{real (fact } m)) +$$

$$\exp t / \text{real (fact } n) \cdot x \cdot n$$

by (cut-tac diff = %n. exp and f = exp and x = x and n = n in Maclaurin-all-le-objl, auto)

107.4 Version for Sine Function

lemma mod-exhaust-less-4:

$$m \mod 4 = 0 \mid m \mod 4 = 1 \mid m \mod 4 = 2 \mid m \mod 4 = (3::nat)$$

by auto

lemma Suc-Suc-mult-two-diff-two [rule-format, simp]:

$$n \neq 0 \rightarrow \text{Suc (Suc (}2 * n - 2)) = 2*n$$

by (induct n, auto)

lemma lemma-Suc-Suc-4n-diff-2 [rule-format, simp]:

$$n \neq 0 \rightarrow \text{Suc (Suc (}4 * n - 2)) = 4*n$$

by (induct n, auto)

lemma Suc-mult-two-diff-one [rule-format, simp]:

$$n \neq 0 \rightarrow \text{Suc (}2 * n - 1) = 2*n$$

by (induct n, auto)

It is unclear why so many variant results are needed.

lemma sin-expansion-lemma:

$$\sin (x + \text{real (Suc } m) * \pi / 2) =$$

$$\cos (x + \text{real (m) * } \pi / 2)$$

by (simp only: cos-add sin-add real-of-nat-Suc add-divide-distrib distrib-right, auto)

lemma Maclaurin-sin-expansion2:

$$\exists t. \text{abs } t \leq \text{abs } x \&$$

$$\sin x =$$

$$\sum_{m<n} \sin-coeff m \cdot x \cdot m$$

$$+ ((\sin(t + 1/2 \cdot \text{real (n) * } \pi) / \text{real (fact } n)) \cdot x \cdot n)$$

apply (cut-tac f = sin and n = n and x = x)

and diff = %n x. sin (x + 1/2*real n * pi) in Maclaurin-all-it-objl)

apply safe

apply (simp (no-asm))

apply (simp (no-asm) add: sin-expansion-lemma)

apply (force intro!: derivative-eq-intros)

apply (subst (asm) setsum.neutral, auto)[1]

apply (rule ccontr, simp)

apply (erule_tac x = x in spec, simp)

apply (erule ssubst)

apply (rule_tac x = t in exI, simp)

apply (rule setsum.cong[OF refl])
apply (auto simp add: sin-coeff-def sin-zero-iff odd-Suc-mult-two-ex)
done

lemma Maclaurin-sin-expansion:
\[ \exists t. \sin x = \sum_{m<n} \sin\text{-coeff } m \cdot x^m + \left(\frac{\sin\left(t + 1/2 \cdot \text{real } (n) \cdot \pi\right)}{\text{real } (n) \cdot \text{fact } n}\right) \cdot x^n \]
apply (insert Maclaurin-sin-expansion2 \[ of x \ n \])
apply (blast intro: elim:)
done

lemma Maclaurin-sin-expansion3:
\[ |n > 0; 0 < x| \implies \exists t. 0 < t \land t < x \land \sin x = \sum_{m<n} \sin\text{-coeff } m \cdot x^m + \left(\frac{\sin\left(t + 1/2 \cdot \text{real } (n) \cdot \pi\right)}{\text{real } (n) \cdot \text{fact } n}\right) \cdot x^n \]
apply (cut-tac f = sin and n = n and h = x and diff = \%n x. sin (x + 1/2*real (n) *pi) in Maclaurin-obj)
apply safe
apply simp
apply (simp (no-asms) add: sin-expansion-lemma)
apply (force intro!: derivative-eq-intros)
apply (erule ssubst)
apply (rule_tac x = t in exI, simp)
apply (rule setsum.cong[OF refl])
apply (auto simp add: sin-coeff-def sin-zero-iff odd-Suc-mult-two-ex)
done

lemma Maclaurin-sin-expansion4:
\[ 0 < x \implies \exists t. 0 < t \land t < x \land \sin x = \sum_{m<n} \sin\text{-coeff } m \cdot x^m + \left(\frac{\sin\left(t + 1/2 \cdot \text{real } (n) \cdot \pi\right)}{\text{real } (n) \cdot \text{fact } n}\right) \cdot x^n \]
apply (cut-tac f = sin and n = n and h = x and diff = \%n x. sin (x + 1/2*real (n) *pi) in Maclaurin2-obj)
apply safe
apply simp
apply (simp (no-asms) add: sin-expansion-lemma)
apply (force intro!: derivative-eq-intros)
apply (erule ssubst)
apply (rule_tac x = t in exI, simp)
apply (rule setsum.cong[OF refl])
apply (auto simp add: sin-coeff-def sin-zero-iff odd-Suc-mult-two-ex)
done
107.5 Maclaurin Expansion for Cosine Function

lemma sumr-cos-zero-one [simp]:
\[ \sum_{m < (Suc \ n)} \cos\text{-coeff} \ m \times 0^m = 1 \]
by (induct n, auto)

lemma cos-expansion-lemma:
\[ \cos (x + \text{real}( Suc \ m) \times \text{pi} / 2) = -\sin (x + \text{real} \ m \times \text{pi} / 2) \]
by (simp only: cos-add sin-add real-af-nat-Suc distrib-right add-divide-distrib, auto)

lemma Maclaurin-cos-expansion:
\[ \exists t. \ \text{abs} \ t \leq \text{abs} \ x \ & \ 
\cos x = \\
\left( \sum_{m < n} \cos\text{-coeff} \ m \times x^m \right) \\
+ \left( (\cos (t + 1/2 \times \text{real} \ n \times \text{pi}) / \text{real} \ (\text{fact} n)) \times x^n \right) \]
apply (cut-tac f = \cos and n = n and x = x and diff = \%n x. \cos (x + 1/2*\text{real} \ n \times \text{pi}) in Maclaurin-all-lt-objl)
apply safe
apply (simp (no-asmp))
apply (simp (no-asmp) add; cos-expansion-lemma)
apply (case-tac n, simp)
apply (simp del: setsum-lessThan-Suc)
apply (erule contr, simp)
apply (erule_tac \_ = x in spec, simp)
apply (erule subst)
apply (rule_tac \_ = x \_ in \_I, simp)
apply (rule setsum.cong[OF refl])
apply (auto simp add: cos-coeff-def cos-zero-iff even-mult-two-ex)
done

lemma Maclaurin-cos-expansion2:
\[ \| 0 < x; n > 0 \| \Longrightarrow \exists t. \ 0 < t \ & \ t < x \ & \\
\cos x = \\
\left( \sum_{m < n} \cos\text{-coeff} \ m \times x^m \right) \\
+ \left( (\cos (t + 1/2 \times \text{real} \ n \times \text{pi}) / \text{real} \ (\text{fact} n)) \times x^n \right) \]
apply (cut-tac f = cos and n = n and h = x and diff = \%n x. \cos (x + 1/2*\text{real} \ n \times \text{pi}) in Maclaurin-objl)
apply safe
apply simp
apply (simp (no-asmp) add; cos-expansion-lemma)
apply (erule subst)
apply (rule_tac \_ = x \_ in \_I, simp)
apply (rule setsum.cong[OF refl])
apply (auto simp add: cos-coeff-def cos-zero-iff even-mult-two-ex)
done

lemma Maclaurin-minus-cos-expansion:
\[ \| x < 0; n > 0 \| \Longrightarrow \exists t. \ x < t \ & \ t < 0 \ &
\[ \cos x = (\sum_{m < n} \cos-coeff\ m \cdot x \cdot m) + ((\cos(t + 1/2 \cdot \text{real}(n) \cdot \pi)) / \text{real}(\text{fact}(n)) \cdot x \cdot n) \]

apply (cut-tac f = cos and n = n and h = x and diff = \%n \cdot \cos(x + 1/2\cdot\text{real}(n) \cdot \pi)) in Maclaurin-minus-objl
apply safe
apply simp
apply (simp (no-asn) add: cos-expansion-lemma)
apply (erule ssubst)
apply (rule_tac x = t in exI, simp)
apply (rule setsum.cong[OF refl])
apply (auto simp add: cos-coeff-def cos-zero-iff even-mult-two-ex)
done

lemma sin-bound-lemma:
\[ |x = y; \text{abs}\ u \leq (v::real) | \implies |(x + u) - y| \leq v \]
by auto

lemma Maclaurin-sin-bound:
\[ \text{abs}(\sin x - (\sum_{m < n} \sin-coeff\ m \cdot x \cdot m)) \leq \text{inverse}(\text{real}(\text{fact}(n)) \cdot |x| \cdot n) \]

proof –
have \([! x (y::real), x \leq 1 \implies 0 \leq y \implies x \cdot y \leq 1 \cdot y\]
by (rule-tac mult-right-mono, simp-all)

note est = this[simplified]

let ?diff = \(\lambda(n::nat)\) \(x\). if \(n \mod 4\) = 0 then \(\sin(x)\) else if \(n \mod 4\) = 1 then \(\cos(x)\) else if \(n \mod 4\) = 2 then \(\sin(x)\) else \(-\cos(x)\)

have diff-0: ?diff 0 = sin by simp

have DERIV-diff: \(\forall m x. \text{DERIV} (\text{?diff m}) x :> \text{?diff (Suc m)}) x\)
apply (clarify)
apply (subst (1 2 3) mod-Suc-ev-Suc-mod)
apply (cut-tac m=m in mod-exhaust-less-4)
apply (safe, auto intro!: derivative-eq-intros)
done

from Maclaurin-all-le [OF diff-0 DERIV-diff] obtain t where \(t1: |t| \leq |x|\) and \(t2: \sin x = (\sum_{m < n} \text{?diff m 0} / \text{real(\text{fact}(m))} \cdot x \cdot m) + \text{?diff n t} / \text{real(\text{fact}(n))} \cdot x \cdot n\) by fast

have diff-m-0:
\(\forall m. \text{?diff m 0} = (\text{if even m then 0 else } -1 \cdot ((m - \text{Suc 0}) \cdot \text{div}\ 2))\)
apply (subst even-even-mod-4-iff)
apply (cut-tac m=m in mod-exhaust-less-4)
apply (elim disjE, simp-all)
apply (safe dest!: mod-eqD, simp-all)
THEORY “Taylor”

done

show ?thesis

unfolding sin-coeff-def

apply (subst t2)

apply (rule sin-bound-lemma)

apply (rule setsum.cong[OF refl])

apply (subst diff-m-0, simp)

apply (auto intro: mult-right-mono [where b=1, simplified] mult-right-mono simp add: est ac-simps divide-inverse power-abs [symmetric] abs-mult)

done

qed

end

108 Taylor: Taylor series

theory Taylor

imports MacLaurin

begin

We use MacLaurin and the translation of the expansion point c to 0 to prove Taylor’s theorem.

lemma taylor-up:
  assumes INIT: n>0 diff 0 = f
  and DERIV: (∀ m t. m < n & a ≤ t & t ≤ b → DERIV (diff m) t :> (diff (Suc m) t))
  and INTERV: a ≤ c c < b
  shows ∃ t. c < t & t < b &
          f b = (∑ m<n. (diff m c / real (fact m)) * (b - c)ˆm) + (diff n t / real (fact n)) * (b - c)ˆn

proof
  from INTERV have 0 < b−c by arith
  moreover
  from INIT have n>0 ((λm x. diff m (x + c)) 0) = (λx. f (x + c)) by auto
  moreover
  have ALL m t. m < n & 0 <= t & t <= b - c --> DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c)
  proof
    fix m t
    assume m < n & 0 <= t & t <= b - c
    with DERIV and INTERV have DERIV (diff m) (t + c) :> diff (Suc m) (t + c) by auto
    moreover
    from DERIV-ident and DERIV-const have DERIV (%x. x + c) t :> 1+0
    by (rule DERIV-add)
    ultimately have DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c) * (1+0)
    by (rule DERIV-chain2)
    thus DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c) by simp
end
qed
ultimately
have EX:EX t>0. t < b − c &
  f (b − c + c) = (∑ m<n. diff m (0 + c) / real (fact m) * (b − c) ^ m) +
  diff n (t + c) / real (fact n) * (b − c) ^ n
by (rule Maclaurin)
show ?thesis
proof −
  from EX obtain x where
  X: 0 < x & x < b − c &
  f (b − c + c) = (∑ m<n. diff m (0 + c) / real (fact m) * (b − c) ^ m) +
  diff n (x + c) / real (fact n) * (b − c) ^ n ..
  let ?H = x + c
  from X have c ?H & ?H < b & f b = (∑ m<n. diff m c / real (fact m) * (b - c) ^ m) +
  diff n ?H / real (fact n) * (b − c) ^ n
by fastforce
thus ?thesis by fastforce
qed
qed

lemma taylor-down:
  assumes INIT: n>0 diff 0 = f
  and DERIV: (∀ m t. m < n & a ≤ t & t ≤ b → DERIV (diff m) t := (diff m x)
  and INTERV: a < c c ≤ b
  shows ∃ t. a < t & t < c &
  f a = (∑ m<n. (diff m c / real (fact m)) * (a − c) ^m) + (diff n t / real (fact n)) * (a − c) ^ n
proof −
  from INTERV have a−c < 0 by arith
  moreover
  from INIT have n>0 ((λm x. diff m (x + c)) 0) = (λx. f (x + c)) by auto
  moreover
  have ALL m t. m < n & a−c <= t & t <= 0 → DERIV (∀x. diff m (x + c)) t := diff (Suc m) (t + c)
  proof (rule allI impl)+
    fix m t
    assume m < n & a−c <= t & t <= 0
    with DERIV and INTERV have DERIV (diff m) (t + c) := diff (Suc m) (t + c) by auto
    moreover
    from DERIV-ident and DERIV-const have DERIV (∀x. x + c) t := 1+0
    by (rule DERIV-add)
    ultimately have DERIV (∀x. diff m (x + c)) t := diff (Suc m) (t + c) * (1+0)
    by (rule DERIV-chain2)
    thus DERIV (∀x. diff m (x + c)) t := diff (Suc m) (t + c) by simp
  qed
  ultimately
have EX: EX t > a - c, t < 0 &
  f (a - c + c) = (SUM m<n. diff m (0 + c) / real (fact m) * (a - c) ^ m) +
  diff n (t + c) / real (fact n) * (a - c) ^ n

by (rule Maclaurin-minus)

show ?thesis

proof -
  from EX obtain x where X: a - c < x & x < 0 &
  f (a - c + c) = (SUM m<n. diff m (0 + c) / real (fact m) * (a - c) ^ m) +
  diff n (x + c) / real (fact n) * (a - c) ^ n ..

  let ?H = x + c
  from X have a < ?H & ?H < c & f a = (SUM m<n. diff m c / real (fact m) * (a - c) ^ m) +
  diff n ?H / real (fact n) * (a - c) ^ n
  by fastforce

thus ?thesis by fastforce

qed

lemma taylor:
  assumes INIT: n > 0 diff 0 = f
  and DERIV: (∀ m t. m < n & a ≤ t & t ≤ b --> DERIV (diff m) t :> (diff (Suc m) t))
  and INTERV: a ≤ c & c ≤ b a ≤ x x ≤ b x ≠ c
  shows ∃ t. (if x < c then (x < t & t < c) else (c < t & t < x)) &
  f x = (SUM m<n. diff m c / real (fact m) * (x - c) ^ m) + (diff n t / real (fact n)) * (x - c) ^ n

proof (cases x < c)
  case True
  note INIT
  moreover from DERIV and INTERV
  have ∀ m t. m < n ∧ x ≤ t ∧ t ≤ b --> DERIV (diff m) t :> (diff (Suc m) t)
  by fastforce
  moreover note True
  moreover from INTERV have c ≤ b by simp
  ultimately have EX: ∃ t > x. t < c ∧ f x =
  (SUM m<n. diff m c / real (fact m) * (x - c) ^ m) + diff n t / real (fact n) * (x - c) ^ n
  by (rule taylor-down)
  with True show ?thesis by simp

next
  case False
  note INIT
  moreover from DERIV and INTERV
  have ∀ m t. m < n ∧ a ≤ t ∧ t ≤ x --> DERIV (diff m) t :> (diff (Suc m) t)
  by fastforce
  moreover from INTERV have a ≤ c by arith
  moreover from False and INTERV have c < x by arith
  ultimately have EX: ∃ t > c. t < x ∧ f x =
\[
\sum_{m < n} \text{diff}_m c / \text{real}(\text{fact}_m) \ast (x - c) \wedge m + \text{diff}_n t / \text{real}(\text{fact}_n) \ast (x - c) \wedge n
\]
by (rule taylor-up)
with False show thesis by simp
qed

109 Complex-Main: Comprehensive Complex Theory

theory Complex-Main
imports
  Main
  Real
  Complex
  Transcendental
  Taylor
  Deriv
begin
end

References