Light-Weight Containers

Andreas Lochbihler

May 27, 2015
Abstract

This development provides a framework for container types like sets and maps such that generated code implements these containers with different (efficient) data structures. Thanks to type classes and refinement during code generation, this light-weight approach can seamlessly replace Isabelle’s default setup for code generation. Heuristics automatically pick one of the available data structures depending on the type of elements to be stored, but users can also choose on their own. The extensible design permits to add more implementations at any time.

To support arbitrary nesting of sets, we define a linear order on sets based on a linear order of the elements and provide efficient implementations. It even allows to compare complements with non-complements.
Contents

1 Introduction 7

2 An executable linear order on sets 9
  2.1 Auxiliary definitions ........................................ 9
  2.2 Definitions to prove equations about the cardinality of data types ........................................ 11
    2.2.1 Specialised range constants .......................... 11
    2.2.2 Cardinality primitives for polymorphic HOL types ... 13
  2.3 Shortcut fusion for lists ................................. 14
    2.3.1 The type of generators for finite lists ............ 14
    2.3.2 Generators for ‘a list .............................. 17
    2.3.3 Destroying lists .................................. 22
  2.4 List fusion for lexicographic order .................. 25
    2.4.1 Setup for list fusion ................................ 25
  2.5 Every partial order can be extended to a total order ... 26
  2.6 An executable linear order on sets .................... 28
    2.6.1 Definition of the linear order ..................... 28
    2.6.2 Implementation based on sorted lists ............... 36
    2.6.3 Implementation of proper intervals for sets ....... 38
    2.6.4 Proper intervals for HOL types .................. 40
    2.6.5 List fusion for the order and proper intervals on ‘a set 43
    2.6.6 Drop notation .................................. 47
    2.6.7 Introduction ................................... 48

3 Light-weight containers 49
  3.1 A linear order for code generation .................... 49
    3.1.1 Optional comparators ................................ 49
    3.1.2 Generator for the ccompare-class .................. 50
    3.1.3 Instantiations for HOL types ....................... 51
    3.1.4 Proper intervals .................................. 51
  3.2 Instantiate proper-interval of for ‘a list ............. 56
  3.3 A type class for optional equality testing ............ 57
    3.3.1 Generator for the ceq-class ....................... 57
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Type class instances for HOL types</td>
<td>58</td>
</tr>
<tr>
<td>3.4</td>
<td>A type class for optional enumerations</td>
<td>60</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Definition</td>
<td>60</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Generator for the \textit{cenum}-class</td>
<td>61</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Instantiations</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>Locales to abstract over HOL equality</td>
<td>64</td>
</tr>
<tr>
<td>3.6</td>
<td>More on red-black trees</td>
<td>65</td>
</tr>
<tr>
<td>3.6.1</td>
<td>More lemmas</td>
<td>65</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Build the cross product of two RBTs</td>
<td>65</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Build an RBT where keys are paired with themselves</td>
<td>67</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Folding and quantifiers over RBTs</td>
<td>68</td>
</tr>
<tr>
<td>3.6.5</td>
<td>List fusion for RBTs</td>
<td>68</td>
</tr>
<tr>
<td>3.7</td>
<td>Mappings implemented by red-black trees</td>
<td>70</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Type definition</td>
<td>71</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Operations</td>
<td>71</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Properties</td>
<td>73</td>
</tr>
<tr>
<td>3.8</td>
<td>Additional operations for associative lists</td>
<td>76</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Operations on the raw type</td>
<td>76</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Operations on the abstract type ((a', b')\textit{alist})</td>
<td>78</td>
</tr>
<tr>
<td>3.9</td>
<td>Sets implemented by distinct lists</td>
<td>80</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Operations on the raw type with parametrised equality</td>
<td>80</td>
</tr>
<tr>
<td>3.9.2</td>
<td>The type of distinct lists</td>
<td>82</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Operations</td>
<td>82</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Properties</td>
<td>83</td>
</tr>
<tr>
<td>3.10</td>
<td>Sets implemented by red-black trees</td>
<td>86</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Type and operations</td>
<td>89</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Primitive operations</td>
<td>89</td>
</tr>
<tr>
<td>3.10.3</td>
<td>Properties</td>
<td>90</td>
</tr>
<tr>
<td>3.11</td>
<td>Sets implemented as Closures</td>
<td>94</td>
</tr>
<tr>
<td>3.12</td>
<td>Different implementations of sets</td>
<td>94</td>
</tr>
<tr>
<td>3.12.1</td>
<td>Auxiliary functions</td>
<td>94</td>
</tr>
<tr>
<td>3.12.2</td>
<td>Delete code equation with set as constructor</td>
<td>98</td>
</tr>
<tr>
<td>3.12.3</td>
<td>Set implementations</td>
<td>100</td>
</tr>
<tr>
<td>3.12.4</td>
<td>Set operations</td>
<td>100</td>
</tr>
<tr>
<td>3.12.5</td>
<td>Type class instantiations</td>
<td>129</td>
</tr>
<tr>
<td>3.12.6</td>
<td>Generator for the \textit{set-impl}-class</td>
<td>130</td>
</tr>
<tr>
<td>3.12.7</td>
<td>Pretty printing for sets</td>
<td>132</td>
</tr>
<tr>
<td>3.13</td>
<td>Different implementations of maps</td>
<td>133</td>
</tr>
<tr>
<td>3.13.1</td>
<td>Map implementations</td>
<td>133</td>
</tr>
<tr>
<td>3.13.2</td>
<td>Map operations</td>
<td>133</td>
</tr>
<tr>
<td>3.13.3</td>
<td>Type classes</td>
<td>136</td>
</tr>
<tr>
<td>3.13.4</td>
<td>Generator for the \textit{mapping-impl}-class</td>
<td>137</td>
</tr>
<tr>
<td>3.14</td>
<td>Infrastructure for operation identification</td>
<td>138</td>
</tr>
</tbody>
</table>
## User guide

### 4.1 Characteristics .............................................. 141
### 4.2 Getting started ........................................... 142
### 4.3 New types as elements .................................... 142
- 4.3.1 Equality testing ........................................ 143
- 4.3.2 Ordering ................................................ 145
- 4.3.3 Heuristics for picking an implementation ............ 146
- 4.3.4 Set comprehensions .................................... 147
- 4.3.5 Nested sets ............................................. 148
### 4.4 New implementations for containers ..................... 150
- 4.4.1 Model and verify the data structure .................. 150
- 4.4.2 Generalise the data structure .......................... 151
- 4.4.3 Hide the invariants of the data structure ............ 152
- 4.4.4 Connecting to the container ............................ 153
### 4.5 Changing the configuration .............................. 156
### 4.6 New containers types ..................................... 156
### 4.7 Troubleshooting ............................................ 157
- 4.7.1 Nesting of mappings ................................... 157
- 4.7.2 Wellsortedness errors .................................. 157
- 4.7.3 Exception raised at run-time ............................ 158
- 4.7.4 LC slows down my code ................................ 159
Chapter 1

Introduction

This development focuses on generating efficient code for container types like sets and maps. It falls into two parts: First, we define linear order on sets (Ch. 2) that is efficiently executable given a linear order on the elements. Second, we define an extensible framework LC (for light-weight containers) that supports multiple (efficient) implementations of container types (Ch. 3) in generated code. Both parts heavily exploit type classes and the refinement features of the code generator [2]. This way, we are able to implement the HOL types for sets and maps directly, as the name light-weight containers (LC) emphasises.

In comparison with the Isabelle Collections Framework (ICF) [4, 3], the style of refinement is the major difference. In the ICF, the container types are replaced with the types of the data structures inside the logic. Typically, the user has to define his operations that involve maps and sets a second time such that they work on the concrete data structures; then, she has to prove that both definitions agree. With LC, the refinement happens inside the code generator. Hence, the formalisation can stick with the types ‘a set and (‘a,’b) mapping and there is no need to duplicate definitions or prove refinement. The drawback is that with LC, we can only implement operations that can be fully specified on the abstract container type. In particular, the internal representation of the implementations may not affect the result of the operations. For example, it is not possible to pick non-deterministically an element from a set or fold a set with a non-commutative operation, i.e., the result depends on the order of visiting the elements.

For more documentation and introductory material refer to the userguide (Chapter 4) and the ITP-2013 paper [5].
CHAPTER 1. INTRODUCTION
Chapter 2

An executable linear order on sets

2.1 Auxiliary definitions

lemma insert-bind-set: insert a A >>= f = f a ∪ (A >>= f)
⟨proof⟩

lemma set-bind-iff:
  set (List.bind xs f) = Set.bind (set xs) (set ○ f)
⟨proof⟩

lemma set-bind-conv-fold: set xs >>= f = fold (op ∪ ○ f) xs {}
⟨proof⟩

lemma card-gt-1D:
  assumes card A > 1
  shows ∃ x y. x ∈ A ∧ y ∈ A ∧ x ≠ y
⟨proof⟩

lemma card-eq-1-iff: card A = 1 ↔ (∃ x. A = {x})
⟨proof⟩

lemma card-eq-Suc-0-ex1: card A = Suc 0 ↔ (∃! x. x ∈ A)
⟨proof⟩

context linorder begin

lemma sorted-last: [ sorted xs; x ∈ set xs ] → x ≤ last xs
⟨proof⟩

end

lemma empty-filter-conv: [] = filter P xs ↔ (∀ x∈ set xs. ¬ P x)
⟨proof⟩


definition ID :: 'a ⇒ 'a where ID = id

lemma ID-code [code, code-unfold]: ID = (λx. x) ⟨proof⟩

lemma ID-Some: ID (Some x) = Some x ⟨proof⟩

lemma ID-None: ID None = None ⟨proof⟩

lexicographic order on pairs

context
fixes leq-a :: 'a ⇒ 'a ⇒ bool (infix ⊑ 50)
and less-a :: 'a ⇒ 'a ⇒ bool (infix ⊏ 50)
and leq-b :: 'b ⇒ 'b ⇒ bool (infix ⊑ 50)
and less-b :: 'b ⇒ 'b ⇒ bool (infix ⊏ 50)
begin

definition less-eq-prod :: ('a × 'b) ⇒ ('a × 'b) ⇒ bool (infix ⊑ 50)
where less-eq-prod = (λ(x1, x2) (y1, y2). x1 ⊑ a y1 ∨ x1 ⊑ a y1 ∧ x2 ⊑ b y2)

definition less-prod :: ('a × 'b) ⇒ ('a × 'b) ⇒ bool (infix ⊏ 50)
where less-prod = (λ(x1, x2) (y1, y2). x1 ⊑ a y1 ∨ x1 ⊑ a y1 ∧ x2 ⊑ b y2)

lemma less-eq-prod-simps [simp]:
(x1, x2) ⊑ (y1, y2) ⟷ x1 ⊑ a y1 ∨ x1 ⊑ a y1 ∧ x2 ⊑ b y2
⟨proof⟩

lemma less-prod-simps [simp]:
(x1, x2) ⊏ (y1, y2) ⟷ x1 ⊑ a y1 ∨ x1 ⊑ a y1 ∧ x2 ⊑ b y2
⟨proof⟩

end

context
fixes leq-a :: 'a ⇒ 'a ⇒ bool (infix ⊑ 50)
and less-a :: 'a ⇒ 'a ⇒ bool (infix ⊏ 50)
and leq-b :: 'b ⇒ 'b ⇒ bool (infix ⊑ 50)
and less-b :: 'b ⇒ 'b ⇒ bool (infix ⊏ 50)
assumes lin-a: class.linorder leq-a less-a
and lin-b: class.linorder leq-b less-b
begin

abbreviation (input) less-eq-prod′ :: ('a × 'b) ⇒ ('a × 'b) ⇒ bool (infix ⊑ 50)
where less-eq-prod′ ≡ less-eq-prod leq-a less-a leq-b
2.2. DEFINITIONS TO PROVE EQUATIONS ABOUT THE CARDINALITY OF DATA TYPES

abbreviation (input) less-prod' :: ('a × 'b) ⇒ ('a × 'b) ⇒ bool (infix ⊏ 50)
where less-prod' ≡ less-prod leq-a less-a less-b

lemma linorder-prod:
  class.linorder op ⊑ op ⊏
⟨proof⟩
end

hide-const less-eq-prod' less-prod'
end

theory Card-Datatype
imports ~/src/HOL/Library/Cardinality
begin

2.2 Definitions to prove equations about the cardinality of data types

2.2.1 Specialised range constants

definition rangeIt :: 'a ⇒ ('a ⇒ 'a) ⇒ 'a set
where rangeIt x f = range (λn. (f ^^ n) x)

definition rangeC :: ('a ⇒ 'b) set ⇒ 'b set
where rangeC F = (∪f ∈ F. range f)

lemma infinite-rangeIt:
  assumes inj: inj f
  and x: ∀y. x ≠ f y
  shows ¬finite (rangeIt x f)
⟨proof⟩

lemma in-rangeC: f ∈ A ⇒ f x ∈ rangeC A
⟨proof⟩

lemma in-rangeCE: assumes y ∈ rangeC A
  obtains f x where f ∈ A y = f x
⟨proof⟩

lemma in-rangeC-singleton: f x ∈ rangeC {f}
⟨proof⟩

lemma in-rangeC-singleton-const: x ∈ rangeC {λ_. x}
⟨proof⟩

lemma rangeC-rangeC: f ∈ rangeC A ⇒ f x ∈ rangeC (rangeC A)
 CHAPTER 2. AN EXECUTABLE LINEAR ORDER ON SETS

lemma rangeC-eq-empty: rangeC A = {} ←→ A = {}
(proof)

lemma Ball-rangeC-iff:
\((\forall x \in \text{rangeC} A. P x) \iff (\forall f \in A. \forall x. P (f x))\)
(proof)

lemma Ball-rangeC-singleton:
\((\forall x \in \text{rangeC} \{f\}. P x) \iff (\forall x. P (f x))\)
(proof)

lemma Ball-rangeC-rangeC:
\((\forall x \in \text{rangeC} (\text{rangeC} A). P x) \iff (\forall f \in \text{rangeC} A. \forall x. P (f x))\)
(proof)

lemma finite-rangeC:
assumes inj: \(\forall f \in A. \text{inj} f\)
and disjoint: \(\forall f \in A. \forall g \in A. f \neq g \implies (\forall x y. f x \neq g y)\)
shows finite \((\text{rangeC} (A :: ('a \Rightarrow 'b) set)) \iff \text{finite} A \land (A \neq \{\} \implies \text{finite} (\text{UNIV} :: 'a set))\)
(is ?lhs ↔ ?rhs)
(proof)

lemma finite-rangeC-singleton-const:
finite \((\text{rangeC} \{\lambda -. x\})\)
(proof)

lemma card-Un:
\([ \text{finite} A; \text{finite} B ] \implies \text{card} (A \cup B) = \text{card} (A) + \text{card} (B) - \text{card}(A \cap B)\)
(proof)

lemma card-rangeC-singleton-const:
\(\text{card} (\text{rangeC} \{\lambda -. f\}) = 1\)
(proof)

lemma card-rangeC:
assumes inj: \(\forall f \in A. \text{inj} f\)
and disjoint: \(\forall f \in A. \forall g \in A. f \neq g \implies (\forall x y. f x \neq g y)\)
shows card \((\text{rangeC} (A :: ('a \Rightarrow 'b) set)) = \text{CARD}('a) * \text{card} A\)
(is ?lhs = ?rhs)
(proof)

lemma rangeC-Int-rangeC:
\([ \forall f \in A. \forall g \in B. \forall x y. f x \neq g y ] \implies \text{rangeC} A \cap \text{rangeC} B = \{\}\)
(proof)

lemmas rangeC-simps =
2.2. DEFINITIONS TO PROVE EQUATIONS ABOUT THE CARDINALITY OF DATA TYPES

\[
\begin{align*}
\text{in-rangeC-singleton} & \\
\text{in-rangeC-singleton-const} & \\
\text{rangeC-rangeC} & \\
\text{rangeC-eq-empty} & \\
\text{Ball-rangeC-singleton} & \\
\text{Ball-rangeC-rangeC} & \\
\text{finite-rangeC} & \\
\text{finite-rangeC-singleton-const} & \\
\text{card-rangeC-singleton-const} & \\
\text{card-rangeC} & \\
\text{rangeC-Int-rangeC} & \\
\end{align*}
\]

\[\text{bundle card-datatype} =\]
\[\text{rangeC-simps \ [simp]} \]
\[\text{card-Un \ [simp]} \]
\[\text{fun-eq-iff \ [simp]} \]
\[\text{Int-Un-distrib \ [simp]} \]
\[\text{Int-Un-distrib2 \ [simp]} \]
\[\text{card-eq-0-iff \ [simp]} \]
\[\text{imageI \ [simp]} \]
\[\text{image-eqI \ [simp \ del]} \]
\[\text{conj-cong \ [cong]} \]
\[\text{infinite-rangeIt \ [simp]} \]

2.2.2 Cardinality primitives for polymorphic HOL types

\[
\begin{align*}
\text{definition card-fun :: \nat \Rightarrow \nat \Rightarrow \nat} & \\
\text{where \ card-fun \ a \ b = (if \ a \neq 0 \land b \neq 0 \lor b = 1 \ then \ b \ \hat{\ } \ a \ else \ 0)} & \\
\text{lemma CARD-fun \ [card-simps]}: & \\
\quad \text{CARD}(a \Rightarrow \ b) = \text{card-fun \ CARD}(a) \ \text{CARD}(b) & \\
\quad \langle \text{proof} \rangle & \\
\end{align*}
\]

\[
\begin{align*}
\text{definition card-sum :: \nat \Rightarrow \nat \Rightarrow \nat} & \\
\text{where \ card-sum \ a \ b = (if \ a = 0 \lor b = 0 \ then \ 0 \ else \ a + b)} & \\
\text{lemma CARD-sum \ [card-simps]}: & \\
\quad \text{CARD}(a + b) = \text{card-sum \ CARD}(a) \ \text{CARD}(b) & \\
\quad \langle \text{proof} \rangle & \\
\end{align*}
\]

\[
\begin{align*}
\text{definition card-option :: \nat \Rightarrow \nat} & \\
\text{where \ card-option \ n = (if \ n = 0 \ then \ 0 \ else \ Suc \ n)} & \\
\text{lemma CARD-option \ [card-simps]}: & \\
\quad \text{CARD}(\text{a option}) = \text{card-option \ CARD}(\text{a}) & \\
\quad \langle \text{proof} \rangle & \\
\end{align*}
\]

\[
\begin{align*}
\text{definition card-prod :: \nat \Rightarrow \nat \Rightarrow \nat} & \\
\end{align*}
\]
where \( \text{card-prod } a \ b = a \times b \)

lemma \( \text{CARD-prod } [\text{card-simps}]: \)
\[
\text{CARD}(\ 'a \ 'b) = \text{card-prod } \text{CARD}(\ 'a) \ \text{CARD}(\ 'b)
\]
\langle proof \rangle

definition \( \text{card-list } :: \text{nat} \Rightarrow \text{nat} \)
where \( \text{card-list } = 0 \)

lemma \( \text{CARD-list } [\text{card-simps}]: \text{CARD}(\ 'a \ \text{list}) = \text{card-list } \text{CARD}(\ 'a) \)
\langle proof \rangle

definition \( \text{terminates } :: (\ 'a, 's) \text{ raw-generator} \Rightarrow 's \ \text{set} \)
where stop: \( \neg \ \text{fst } g \ s \Rightarrow s \in \text{terminates-on } g \)
| unfold: \( \ [\ \text{fst } g \ s; \ \text{snd } (\ \text{snd } g \ s) \in \text{terminates-on } g ] \Rightarrow s \in \text{terminates-on } g \)

definition \( \text{terminates } :: (\ 'a, 's) \text{ raw-generator} \Rightarrow \text{bool} \)
where \( \text{terminates } g \longleftrightarrow (\ \text{terminates-on } g = \text{UNIV}) \)
2.3. SHORTCUT FUSION FOR LISTS

**lemma** terminatesI [intro?]:
\( (\forall s. s \in \text{terminates-on } g) \implies \text{terminates } g \)
⟨proof⟩

**lemma** terminatesD:
\( \text{terminates } g \implies s \in \text{terminates-on } g \)
⟨proof⟩

**lemma** terminates-on-stop:
\( \text{terminates-on } (\lambda -. \text{False}, \text{next}) = \text{UNIV} \)
⟨proof⟩

**lemma** wf-terminates:
\( \text{assumes } \text{wf } R \)
\( \text{and step: } (\forall s. \text{fst } g \ s \implies (\text{snd } (\text{snd } g \ s), s) \in R) \)
\( \text{shows } \text{terminates } g \)
⟨proof⟩

**lemma** terminates-wfD:
\( \text{assumes } \text{terminates } g \)
\( \text{shows } \text{wf } \{ (\text{snd } (\text{snd } g \ s), s) \mid s . \text{fst } g \ s \} \)
⟨proof⟩

**lemma** terminates-wfE:
\( \text{assumes } \text{terminates } g \)
\( \text{obtains } R \text{ where } \text{wf } R \quad (\forall s. \text{fst } g \ s \implies (\text{snd } (\text{snd } g \ s), s) \in R) \)
⟨proof⟩

context fixes \( g :: (\'a, \'s) \text{ raw-generator} \) begin

**partial-function** (option) terminates-within :: \( \:'s \Rightarrow \text{nat option} \)
where
\( \text{terminates-within } s = \)
\( \begin{aligned}
\text{let } (\text{has-next}, \text{next}) = g \\
\text{if } \text{has-next } s \text{ then } \\
\text{map-option } (\lambda n. n + 1) (\text{terminates-within } (\text{snd } (\text{next } s))) \\
\text{else } \text{Some } 0
\end{aligned} \)

**lemma** terminates-on-conv-dom-terminates-within:
\( \text{terminates-on } g = \text{dom } \text{terminates-within} \)
⟨proof⟩

end

**lemma** terminates-within-unfold:
\( \text{has-next } s \implies \\
\text{terminates-within } (\text{has-next}, \text{next}) \ s = \text{map-option } (\lambda n. n + 1) (\text{terminates-within } (\text{has-next}, \text{next}) \ (\text{snd } (\text{next } s))) \)
⟨proof⟩
typedef (′a, ′s) generator = {g :: (′a, ′s) raw-generator. terminates g}

morphisms generator Generator
⟨proof⟩

setup-lifting type-definition-generator

lemma terminates-on-generator-eq-UNIV:
  terminates-on (generator g) = UNIV
⟨proof⟩

lemma terminates-within-stop:
  terminates-within (λ-. False, next) s = Some 0
⟨proof⟩

lemma terminates-within-generator-neq-None:
  terminates-within (generator g) s ≠ None
⟨proof⟩

locale list =
  fixes g :: (′a, ′s) generator begin

definition has-next :: ′s ⇒ bool
  where has-next = fst (generator g)

definition next :: ′s ⇒ ′a × ′s
  where next = snd (generator g)

function unfoldr :: ′s ⇒ ′a list
  where unfoldr s = (if has-next s then let (a, s′) = next s in a # unfoldr s′ else [])
⟨proof⟩
termination
⟨proof⟩
declare unfoldr.simps [simp del]

lemma unfoldr-simps:
  has-next s ⇒ unfoldr s = fst (next s) # unfoldr (snd (next s))
  ¬ has-next s ⇒ unfoldr s = []
⟨proof⟩

end

declare
  list.has-next-def[code]
  list.next-def[code]
  list.unfoldr.simps[code]
2.3. SHORTCUT FUSION FOR LISTS

context
begin
interpretation lifting-syntax (proof)

lemma generator-has-next-transfer [transfer-rule]:
  (per-generator op = op = === op =) fst list.has-next
  (proof)

lemma generator-next-transfer [transfer-rule]:
  (per-generator op = op = === op =) snd list.next
  (proof)

end

lemma unfoldr-eq-Nil-iff [iff]:
  list.unfoldr g s = [] ←→ ¬ list.has-next g s
  (proof)

lemma Nil-eq-unfoldr-iff [simp]:
  [] = list.unfoldr g s ←→ ¬ list.has-next g s
  (proof)

2.3.2 Generators for 'a list

primrec list-has-next :: 'a list ⇒ bool
where
  list-has-next [] ←→ False
  | list-has-next (x # xs) ←→ True

primrec list-next :: 'a list ⇒ 'a × 'a list
where
  list-next (x # xs) = (x, xs)

lemma terminates-list-generator: terminates (list-has-next, list-next)
  (proof)

lift-definition list-generator :: ('a, 'a list) generator
  is (list-has-next, list-next)
  (proof)

lemma has-next-list-generator [simp]:
  list.has-next list-generator = list-has-next
  (proof)

lemma next-list-generator [simp]:
  list.next list-generator = list-next
  (proof)

lemma unfoldr-list-generator:
list.unfoldr list-generator xs = xs
⟨proof⟩

lemma terminates-replicate-generator:
  terminates (λn :: nat. 0 < n, λn. (a, n - 1))
⟨proof⟩

lift-definition replicate-generator :: 'a ⇒ ('a, nat) generator
  is λa. (λn. 0 < n, λn. (a, n - 1))
⟨proof⟩

lemma has-next-replicate-generator [simp]:
  list.has-next (replicate-generator a) n =→ 0 < n
⟨proof⟩

lemma next-replicate-generator [simp]:
  list.next (replicate-generator a) n = (a, n - 1)
⟨proof⟩

lemma unfoldr-replicate-generator:
  list.unfoldr (replicate-generator a) n = replicate n a
⟨proof⟩

context fixes f :: 'a ⇒ 'b begin

lift-definition map-generator :: ('a, 's) generator ⇒ ('b, 's) generator
  is λ(has-next, next). (has-next, λs. let (a, s') = next s in (f a, s'))
⟨proof⟩

lemma has-next-map-generator [simp]:
  list.has-next (map-generator g) = list.has-next g
⟨proof⟩

lemma next-map-generator [simp]:
  list.next (map-generator g) = apfst f ◦ list.next g
⟨proof⟩

lemma unfoldr-map-generator:
  list.unfoldr (map-generator g) = map f ◦ list.unfoldr g
  (is ?lhs = ?rhs)
⟨proof⟩

end

context fixes g1 :: ('a, 's1) raw-generator
  and g2 :: ('a, 's2) raw-generator
begin

fun append-has-next :: 's1 × 's2 ⇒ bool
2.3. SHORTCUT FUSION FOR LISTS

where
append-has-next (Inl (s1, s2)) ←→ fst g1 s1 ∨ fst g2 s2
| append-has-next (Inr s2) ←→ fst g2 s2

fun append-next :: 's1 × 's2 + 's2 ⇒ 'a × ('s1 × 's2 + 's2)
where
append-next (Inl (s1, s2)) = (if fst g1 s1 then
  let (x, s1') = snd g1 s1 in (x, Inl (s1', s2))
  else append-next (Inr s2))
| append-next (Inr s2) = (let (x, s2') = snd g2 s2 in (x, Inr s2'))
end

lift-definition append-generator :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ ('a, 's1 × 's2 + 's2) generator
is λg1 g2. (append-has-next g1 g2, append-next g1 g2)
⟨proof⟩

definition append-init :: 's1 ⇒ 's2 ⇒ 's1 × 's2 + 's2
where append-init s1 s2 = Inl (s1, s2)

lemma has-next-append-generator [simp]:
list.has-next (append-generator g1 g2) (Inl (s1, s2)) ←→
list.has-next g1 s1 ∨ list.has-next g2 s2
list.has-next (append-generator g1 g2) (Inr s2) ←→ list.has-next g2 s2
⟨proof⟩

lemma next-append-generator [simp]:
list.next (append-generator g1 g2) (Inl (s1, s2)) =
(if list.has-next g1 s1 then
  let (x, s1') = list.next g1 s1 in (x, Inl (s1', s2))
  else list.next (append-generator g1 g2) (Inr s2))
list.next (append-generator g1 g2) (Inr s2) = apsnd Inr (list.next g2 s2)
⟨proof⟩

lemma unfoldr-append-generator-Inr:
list.unfoldr (append-generator g1 g2) (Inr s2) = list.unfoldr g2 s2
⟨proof⟩

lemma unfoldr-append-generator-Inl:
list.unfoldr (append-generator g1 g2) (Inl (s1, s2)) =
list.unfoldr g1 s1 @ list.unfoldr g2 s2
⟨proof⟩

lemma unfoldr-append-generator:
list.unfoldr (append-generator g1 g2) (append-init s1 s2) =
list.unfoldr g1 s1 @ list.unfoldr g2 s2
⟨proof⟩
lift-definition zip-generator :: (\texttt{a, 's1}) generator \Rightarrow (\texttt{b, 's2}) generator \Rightarrow (\texttt{a \times b, 's1 \times 's2}) generator

is \lambda \texttt{(has-next1, next1)} (\texttt{has-next2, next2)}.

(\lambda \texttt{(s1, s2). has-next1 s1 \land has-next2 s2},
\lambda \texttt{(s1, s2). let \texttt{(x, s1')} = next1 s1; \texttt{(y, s2')} = next2 s2
in ((x, y), (s1', s2'))})

⟨proof⟩

abbreviation (input) zip-init :: \texttt{'s1} \Rightarrow \texttt{'s2} \Rightarrow \texttt{'s1 \times 's2}
where zip-init \equiv \texttt{Pair}

lemma has-next-zip-generator [simp]:
list.has-next (zip-generator g1 g2) (s1, s2) \iff
list.has-next g1 s1 \land list.has-next g2 s2

⟨proof⟩

lemma next-zip-generator [simp]:
list.next (zip-generator g1 g2) (s1, s2) =
((\texttt{fst (list.next g1 s1)}, \texttt{fst (list.next g2 s2)}),
(\texttt{snd (list.next g1 s1)}, \texttt{snd (list.next g2 s2)}))

⟨proof⟩

lemma unfoldr-zip-generator:
list.unfoldr (zip-generator g1 g2) (zip-init s1 s2) =
zip (list.unfoldr g1 s1) (list.unfoldr g2 s2)

⟨proof⟩

context fixes bound :: \texttt{nat}
begin

lift-definition upt-generator :: (\texttt{nat, nat}) generator
is \lambda n. n < bound, \lambda n. (n, Suc n)

⟨proof⟩

lemma has-next-upt-generator [simp]:
list.has-next upt-generator n \iff n < bound

⟨proof⟩

lemma next-upt-generator [simp]:
list.next upt-generator n = (n, Suc n)

⟨proof⟩

lemma unfoldr-upt-generator:
list.unfoldr upt-generator n = [n..<\texttt{bound}]

⟨proof⟩

end
2.3. SHORTCUT FUSION FOR LISTS

context fixes bound :: int begin

lift-definition upto-generator :: (int, int) generator
  is (\n. n ≤ bound, \n. (n, n + 1))
⟨proof⟩

lemma has-next upto-generator [simp]:
  list.has-next upto-generator n \iff n ≤ bound
⟨proof⟩

lemma next upto-generator [simp]:
  list.next upto-generator n = (n, n + 1)
⟨proof⟩

lemma unfoldr upto-generator:
  list.unfoldr upto-generator n = [n..bound]
⟨proof⟩

end

context
  fixes P :: 'a ⇒ bool
begin

context
  fixes g :: ('a, 's) raw-generator
begin

inductive filter-has-next :: 's ⇒ bool
where
  [ fst g s; P (fst (snd g s)) ] \implies filter-has-next s
| [ fst g s; \neg P (fst (snd g s)); filter-has-next (snd (snd g s)) ] \implies filter-has-next s

partial-function (tailrec) filter-next :: 's ⇒ 'a × 's
where
  filter-next s = (let (x, s') = snd g s in if P x then (x, s') else filter-next s')

end

lift-definition filter-generator :: ('a, 's) generator ⇒ ('a, 's) generator
  is \g. (filter-has-next g, filter-next g)
⟨proof⟩

lemma has-next filter-generator:
  list.has-next (filter-generator g) s \iff
  list.has-next g s ∧ (let (x, s') = list.next g s in if P x then True else list.has-next (filter-generator g) s')
⟨proof⟩
lemma next-filter-generator:
list.next (filter-generator g) s =
(let (x, s') = list.next g s
  in if P x then (x, s') else list.next (filter-generator g) s')
⟨proof⟩

lemma has-next-filter-generator-induct [consumes 1, case-names find step]:
assumes list.has-next (filter-generator g) s
and find: \( s \vdash \) list.has-next g s; P (fst (list.next g s)) \( \Rightarrow Q s \)
and step: \( s \vdash \) list.has-next g s; ¬ P (fst (list.next g s)); Q (snd (list.next g s)) \( \Rightarrow Q s \)
shows Q s
⟨proof⟩

lemma filter-generator-empty-conv:
list.has-next (filter-generator g) s \iff \( \exists x \in \text{set} (\text{list.unfoldr g s}). P x \) (is ?lhs \iff ?rhs)
⟨proof⟩

lemma unfoldr-filter-generator:
list.unfoldr (filter-generator g) s = filter P (list.unfoldr g s)
⟨proof⟩

end

2.3.3 Destroying lists

definition hd-fusion :: ('a, 's) generator \Rightarrow 's \Rightarrow 'a
where hd-fusion g s = hd (list.unfoldr g s)

lemma hd-fusion-code [code]:
hd-fusion g s = (if list.has-next g s then fst (list.next g s) else undefined)
⟨proof⟩

declare hd-fusion-def [symmetric, code-unfold]

definition fold-fusion :: ('a, 's) generator \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 's \Rightarrow 'b \Rightarrow 'b
where fold-fusion g f s = fold f (list.unfoldr g s)

lemma fold-fusion-code [code]:
fold-fusion g f s b =
(if list.has-next g s then
  let (x, s') = list.next g s
  in fold-fusion g f s' (f x b)
  else b)
⟨proof⟩

declare fold-fusion-def [symmetric, code-unfold]
2.3. SHORTCUT FUSION FOR LISTS

**definition** gen-length-fusion :: ('a, 's) generator ⇒ 's ⇒ nat
where gen-length-fusion g n s = n + length (list.unfoldr g s)

**lemma** gen-length-fusion-code [code]:
\[
\text{gen-length-fusion g n s} = \begin{cases} 
\text{Suc n} & \text{if list.has-next g s} \\
\text{n} & \text{else}
\end{cases}
\]

**definition** length-fusion :: ('a, 's) generator ⇒ 's ⇒ nat
where length-fusion g s = length (list.unfoldr g s)

**lemma** length-fusion-code [code]:
\[
\text{length-fusion g} = \text{gen-length-fusion g 0}
\]

**declare** length-fusion-def [symmetric, code-unfold]

**definition** map-fusion :: ('a ⇒ 'b) ⇒ ('a, 's) generator ⇒ 'b list
where map-fusion f g s = map f (list.unfoldr g s)

**lemma** map-fusion-code [code]:
\[
\text{map-fusion f g s} = \begin{cases} 
\text{let (x, s')} = \text{list.next g s} \\
\text{in f x # map-fusion f g s'} & \text{if list.has-next g s} \\
\text{[]} & \text{else}
\end{cases}
\]

**declare** map-fusion-def [symmetric, code-unfold]

**definition** append-fusion :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 's1 ⇒ 's2 ⇒ 'a list
where append-fusion g1 g2 s1 s2 = list.unfoldr g1 s1 @ list.unfoldr g2 s2

**lemma** append-fusion [code]:
\[
\text{append-fusion g1 g2 s1 s2} = \begin{cases} 
\text{let (x, s')} = \text{list.next g1 s1} \\
\text{in x # append-fusion g1 g2 s1' s2} & \text{if list.has-next g1 s1} \\
\text{list.unfoldr g2 s2} & \text{else}
\end{cases}
\]

**declare** append-fusion-def [symmetric, code-unfold]

**definition** zip-fusion :: ('a, 's1) generator ⇒ ('b, 's2) generator ⇒ 's1 ⇒ 's2 ⇒ ('a × b) list
where zip-fusion g1 g2 s1 s2 = zip (list.unfoldr g1 s1) (list.unfoldr g2 s2)

**lemma** zip-fusion-code [code]:
zip-fusion \ g1 \ g2 \ s1 \ s2 =
(if list.has-next g1 s1 \et\ list.has-next g2 s2 then
    let \((x, s1') = list.next g1 s1\);
    \((y, s2') = list.next g2 s2\)
    in \((x, y) \# z\ip-fusion g1 g2 s1' s2'\)
else \[]
⟨proof⟩

declare zip-fusion-def [symmetric, code-unfold]

definition list-all-fusion :: \('a, 's\) generator \Rightarrow \('a \Rightarrow bool\) \Rightarrow 's \Rightarrow bool
where list-all-fusion P g s = List.list-all P (list.unfoldr g s)

lemma list-all-fusion-code [code]:
list-all-fusion g P s \iff
(list.has-next g s \iff
    (let \((x, s') = list.next g s\)
    in P x \et\ list-all-fusion g P s'))
⟨proof⟩

declare list-all-fusion-def [symmetric, code-unfold]

definition list-all2-fusion :: \('a \Rightarrow 'b \Rightarrow bool\) \Rightarrow ('a, 's1) generator \Rightarrow ('b, 's2) generator \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool
where
    list-all2-fusion P g1 g2 s1 s2 =
    list-all2 P (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma list-all2-fusion-code [code]:
list-all2-fusion P g1 g2 s1 s2 =
(if list.has-next g1 s1 then
    list.has-next g2 s2 \et\n    (let \((x, s1') = list.next g1 s1\);
    \((y, s2') = list.next g2 s2\)
    in P x y \et\ list-all2-fusion P g1 g2 s1' s2')
else \neg list.has-next g2 s2)
⟨proof⟩

declare list-all2-fusion-def [symmetric, code-unfold]

definition singleton-list-fusion :: ('a, 'state) generator \Rightarrow 'state \Rightarrow bool
where singleton-list-fusion gen state = (case list.unfoldr gen state of \[] \Rightarrow True
    | _ \Rightarrow False)

lemma singleton-list-fusion-code [code]:
singleton-list-fusion g s \iff
    list.has-next g s \et\ \neg list.has-next g (snd (list.next g s))
⟨proof⟩
2.4. LIST FUSION FOR LEXICOGRAPHIC ORDER

end

theory Lexicographic-Order imports
  List-Fusion
  ~~/src/HOL/Library/Char-ord
begin

hide-const (open) List.lexordp

2.4 List fusion for lexicographic order

context linorder begin

lemma lexordp-take-index-conv:
  lexordp xs ys ↔
  (length xs < length ys ∧ take (length xs) ys = xs) ∨
  (∃ i < min (length xs) (length ys). take i xs = take i ys ∧ xs ! i < ys ! i)
  (is ?lhs = ?rhs)
⟨proof⟩

lemma lexordp-lex: (xs, ys) ∈ lex {(xs, ys). xs < ys} ←→ lexordp xs ys ∧ length xs = length ys
⟨proof⟩

definition lexord-fusion :: (′a, ′s1) generator ⇒ (′a, ′s2) generator ⇒ ′s1 ⇒ ′s2 ⇒ bool
where [code del]: lexord-fusion g1 g2 s1 s2 = lexordp (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma lexord-fusion-code:
  lexord-fusion g1 g2 s1 s2 ←→
  (if list.has-next g1 s1 then
   if list.has-next g2 s2 then
     let (x, s1') = list.next g1 s1;
     (y, s2') = list.next g2 s2
     in x < y ∨ ¬ y < x ∧ lexord-fusion g1 g2 s1' s2'
   else False
   else list.has-next g2 s2)

2.4.1 Setup for list fusion

context ord begin

definition lexord-eq-fusion :: (′a, ′s1) generator ⇒ (′a, ′s2) generator ⇒ ′s1 ⇒ ′s2 ⇒ bool
where [code del]: lexord-eq-fusion g1 g2 s1 s2 = lexordp-eq (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma lexord-eq-fusion-code:
  lexord-eq-fusion g1 g2 s1 s2 ←→
  (if list.has-next g1 s1 then
   if list.has-next g2 s2 then
     let (x, s1') = list.next g1 s1;
     (y, s2') = list.next g2 s2
     in x < y ∨ ¬ y < x ∧ lexord-fusion g1 g2 s1' s2'
   else False
   else list.has-next g2 s2)
CHAPTER 2. AN EXECUTABLE LINEAR ORDER ON SETS

\[\text{proof}\]

\textbf{lemma} lexord-eq-fusion-code:

\begin{align*}
\text{lexord-eq-fusion } & g_1 \ g_2 \ s_1 \ s_2 \iff \\
\text{list.has-next } & g_1 \ s_1 \\
\text{list.has-next } & g_2 \ s_2 \\
\text{(let } (x, s_1') = \text{list.next } g_1 \ s_1; \\
(y, s_2') = \text{list.next } g_2 \ s_2 \\
\text{in } x < y \lor \neg y < x \land \text{lexord-eq-fusion } g_1 \ g_2 \ s_1' \ s_2')
\end{align*}

\[\text{proof}\]

\textbf{end}

\textbf{lemmas} [code] =

\text{lexord-fusion-code ord.lexord-fusion-code}
\text{lexord-eq-fusion-code ord.lexord-eq-fusion-code}

\textbf{lemmas} [symmetric, code-unfold] =

\text{lexord-fusion-def ord.lexord-fusion-def}
\text{lexord-eq-fusion-def ord.lexord-eq-fusion-def}

\textbf{end}

\textbf{theory} Extend-Partial-Order
\textbf{imports} Main
\textbf{begin}

2.5 Every partial order can be extended to a total order

\textbf{lemma} ChainsD: \[
x \in C \land C \in \text{Chains } r \implies (x, y) \in r \lor (y, x) \in r
\]

\[\text{proof}\]

\textbf{lemma} Chains-Field: \[
C \in \text{Chains } r \implies x \in \text{Field } r
\]

\[\text{proof}\]

\textbf{lemma} total-onD:

\[
\text{total-on } A \ r; x \in A \implies (x, y) \in r \lor x = y \lor (y, x) \in r
\]

\[\text{proof}\]

\textbf{lemma} linear-order-imp-linorder: linear-order \{(A, B), \text{leq } A B\} \implies \text{class.linorder}
\text{leq } (\lambda x y. \text{leq } x y \land \neg \text{leq } y x)

\[\text{proof}\]

\textbf{lemma} (in linorder) linear-order: linear-order \{(A, B), A \leq B\}

\[\text{proof}\]
2.5. **EVERY PARTIAL ORDER CAN BE EXTENDED TO A TOTAL ORDER**

**definition** order-consistent :: ('a × 'a) set ⇒ ('a × 'a) set ⇒ bool

**where** order-consistent r s ←→ (∀ a a'. (a, a') ∈ r → (a', a) ∈ s → a = a')

**lemma** order-consistent-sym:
order-consistent r s ⇒ order-consistent s r

**proof**

**lemma** antisym-order-consistent-self:
antisym r ⇒ order-consistent r r

**proof**

**lemma** refl-on-trancl:
assumes refl-on A r
shows refl-on A (r^+)

**proof**

**lemma** total-on-refl-on-consistent-into:
assumes r: total-on A r refl-on A r
and consist: order-consistent r s
and x: x ∈ A and y: y ∈ A and s: (x, y) ∈ s
shows (x, y) ∈ r

**proof**

**lemma** porder-linorder-tranclpE [consumes 5, case-names base step]:
assumes r: partial-order-on A r
and s: linear-order-on B s
and consist: order-consistent r s
and B-subset-A: B ⊆ A
and trancl: (x, y) ∈ (r ∪ s)^+
obtains (x, y) ∈ r
 | u v where (x, u) ∈ r (u, v) ∈ s (v, y) ∈ r

**proof**

**lemma** porder-on-consistent-linorder-on-trancl-antisym:
assumes r: partial-order-on A r
and s: linear-order-on B s
and consist: order-consistent r s
and B-subset-A: B ⊆ A
shows antisym ((r ∪ s)^+)

**proof**

**lemma** porder-on-linorder-on-tranclp-porder-onI:
assumes r: partial-order-on A r
and s: linear-order-on B s
and consist: order-consistent r s
and subset: B ⊆ A
shows partial-order-on A ((r ∪ s)^+)
28  CHAPTER 2.  AN EXECUTABLE LINEAR ORDER ON SETS

\langle\text{proof}\rangle

\text{lemma porder-extend-to-linorder}:  
\text{assumes } r: \text{partial-order-on } A r  
\text{obtains } s \text{ where } \text{linear-order-on } A s \quad \text{order-consistent } r s
\langle\text{proof}\rangle

\text{end}

\text{theory Set-Linorder}
\text{imports}
\text{Containers-Auxiliary}
\text{Lexicographic-Order}
\text{Extend-Partial-Order}
\sim\sim/src/HOL/Library/Cardinality
\begin{eqnarray*}
2.6  \text{ An executable linear order on sets} \end{eqnarray*}

2.6.1  \text{Definition of the linear order}  

\text{Extending finite and cofinite sets}  
Partition sets into finite and cofinite sets and distribute the rest arbitrarily such that complement switches between the two.  
\text{consts infinite-complement-partition :: } 'a \text{ set set}

\text{specification} \left(\text{infinite-complement-partition}\right)  
\text{infinite-complement-partition: finite } (A :: 'a \text{ set}) \implies A \in \text{infinite-complement-partition}  
\text{complement-partition: } \neg \text{finite } (\text{UNIV} :: 'a \text{ set})  
\implies (A :: 'a \text{ set}) \in \text{infinite-complement-partition} \leftrightarrow \neg A \notin \text{infinite-complement-partition}
\langle\text{proof}\rangle

\text{lemma not-in-complement-partition:}  
\neg \text{finite } (\text{UNIV} :: 'a \text{ set})  
\implies (A :: 'a \text{ set}) \notin \text{infinite-complement-partition} \leftrightarrow \neg A \in \text{infinite-complement-partition}
\langle\text{proof}\rangle

\text{lemma not-in-complement-partition-False:}  
\left[ (A :: 'a \text{ set}) \in \text{infinite-complement-partition}; \neg \text{finite } (\text{UNIV} :: 'a \text{ set}) \right]  
\implies \neg A \in \text{infinite-complement-partition} = \text{False}
\langle\text{proof}\rangle

\text{lemma infinite-complement-partition-finite [simp]:}  
\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \text{infinite-complement-partition} = (\text{UNIV} :: 'a \text{ set set})
\langle\text{proof}\rangle

\text{lemma Compl-eq-empty-iff: } \neg A = \{\} \leftrightarrow A = \text{UNIV}
A lexicographic-style order on finite subsets

context ord begin

definition set-less-aux :: 'a set ⇒ 'a set ⇒ bool (infix ⊏"50") where
A ⊏ B ←→ finite A ∧ finite B ∧ (∃y ∈ B − A. ∀z ∈ (A − B) ∪ (B − A). y ≤ z ∧ (z ≤ y −→ y = z))
definition set-less-eq-aux :: 'a set ⇒ 'a set ⇒ bool (infix ⊑"50") where
A ⊑ B ←→ A ∈ infinite-complement-partition ∧ A = B ∨ A ⊏ B

lemma set-less-aux-irrefl [iff]: ¬ A ⊏ A ⟨proof⟩

lemma set-less-eq-aux-refl [iff]: A ⊑ A ←→ A ∈ infinite-complement-partition ⟨proof⟩

lemma set-less-aux-empty [simp]: ¬ A ⊏ {} ⟨proof⟩

lemma set-less-eq-aux-empty [simp]: A ⊑ {} ←→ A = {} ⟨proof⟩

lemma set-less-aux-antisym: [ A ⊏ B; B ⊏ A ] =⇒ False ⟨proof⟩

lemma set-less-aux-conv-set-less-eq-aux: A ⊏ B ←→ A ⊑ B ∧ ¬ B ⊑ A ⟨proof⟩

lemma set-less-eq-aux-antisym: [ A ⊑ B; B ⊑ A ] =⇒ A = B ⟨proof⟩

lemma set-less-aux-finiteD: A ⊏ B =⇒ finite A ∧ B ∈ infinite-complement-partition ⟨proof⟩


lemma Compl-set-less-aux-Compl: finite (UNIV :: 'a set) =⇒ − A ⊏ − B ←→ B ⊑ A ⟨proof⟩

lemma Compl-set-less-eq-aux-Compl: finite (UNIV :: 'a set) =⇒ − A ⊑ − B ←→ B ⊑ A ⟨proof⟩
\textbf{Lemma} set-less-aux-insert-same:
\[ x \in A \iff x \in B \implies \text{insert } x A \sqsubseteq B \iff A \sqsubseteq B \]
\langle proof \rangle

\textbf{Lemma} set-less-eq-aux-insert-same:
\[ [ A \in \text{infinite-complement-partition}; \text{insert } x B \in \text{infinite-complement-partition};\]
\[ x \in A \iff x \in B ] \implies \text{insert } x A \sqsubseteq B \iff A \sqsubseteq B \]
\langle proof \rangle

end

\textbf{Context} order begin

\textbf{Lemma} set-less-aux-singleton-iff:
\[ A \sqsubseteq \{ x \} \iff \text{finite } A \land (\forall a \in A. x < a) \]
\langle proof \rangle

end

\textbf{Context} linorder begin

\textbf{Lemma} wlog-le [case-names sym le]:
\[ \text{assumes } \forall a, b. P a b \implies P b a \]
\[ \text{and } \forall a, b. a \leq b \implies P a b \]
\[ \text{shows } P b a \]
\langle proof \rangle

\textbf{Lemma} empty-set-less-aux [simp]: \[ \emptyset \sqsubseteq A \iff A \neq \emptyset \land \text{finite } A \]
\langle proof \rangle

\textbf{Lemma} empty-set-less-eq-aux [simp]: \[ \emptyset \sqsubseteq A \iff \text{finite } A \]
\langle proof \rangle

\textbf{Lemma} set-less-aux-trans:
\[ \text{assumes } AB: A \sqsubseteq B \text{ and } BC: B \sqsubseteq C \]
\[ \text{shows } A \sqsubseteq C \]
\langle proof \rangle

\textbf{Lemma} set-less-eq-aux-trans [trans]:
\[ [ A \sqsubseteq B; B \sqsubseteq C ] \implies A \sqsubseteq C \]
\langle proof \rangle

\textbf{Lemma} set-less-trans-set-less-eq [trans]:
\[ [ A \sqsubseteq B; B \sqsubseteq C ] \implies A \sqsubseteq C \]
\langle proof \rangle

\textbf{Lemma} set-less-eq-aux-porder: partial-order-on infinite-complement-partition \{(A, B) \}
\[ A \sqsubseteq B \]
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

⟨proof⟩

lemma psubset-finite-imp-set-less-aux:
  assumes AsB: A ⊂ B and B: finite B
  shows A ⊏′ B
⟨proof⟩

lemma subset-finite-imp-set-less-eq-aux:
  [ A ⊆ B; finite B ] ⇒ A ⊑′ B
⟨proof⟩

lemma empty-set-less-aux-finite-iff:
  finite A =⇒ {} ⊏′ A ←→ A ≠ {} 
⟨proof⟩

lemma set-less-aux-finite-total:
  assumes A: finite A and B: finite B
  shows A ⊏′ B ∨ A = B ∨ B ⊏′ A
⟨proof⟩

lemma set-less-eq-aux-finite-total:
  [ finite A; finite B ] ⇒ A ⊑′ B ∨ A = B ∨ B ⊑′ A
⟨proof⟩

lemma set-less-eq-aux-finite-total2:
  [ finite A; finite B ] ⇒ A ⊑′ B ∨ B ⊑′ A
⟨proof⟩

lemma set-less-aux-rec:
  assumes A: finite A and B: finite B
  and A': A ≠ {} and B': B ≠ {}
  shows A ⊏′ B ←→ Min B < Min A ∨ Min A = Min B ∧ A = {Min A} ⊏′ B
  - {Min A}
⟨proof⟩

lemma set-less-eq-aux-rec:
  assumes finite A finite B A ≠ {} B ≠ {}
  shows A ⊑′ B ←→ Min B < Min A ∨ Min A = Min B ∧ A = {Min A} ⊑′ B
  - {Min A}
⟨proof⟩

lemma set-less-aux-Min-antimono:
  [ Min A < Min B; finite A; finite B; A ≠ {} ] ⇒ B ⊏′ A
⟨proof⟩

lemma sorted-Cons-Min; sorted (x # xs) =⇒ Min (insert x (set xs)) = x
⟨proof⟩

lemma set-less-aux-code:
[sorted xs; distinct xs; sorted ys; distinct ys]
\implies set xs \subseteq set ys \iff \text{ord.lexordp op > xs ys}

(\text{proof})

\text{lemma set-less-eq-aux-code:}
\begin{array}{l}
\text{assumes sorted xs \quad distinct xs \quad sorted ys \quad distinct ys} \\
\text{shows set xs \subseteq set ys \iff \text{ord.lexordp-eq op > xs ys}}
\end{array}

(\text{proof})

end

\text{Extending op \subseteq to have \{\} as least element}

context ord begin
\begin{array}{l}
\text{definition set-less-eq-aux': 'a set \Rightarrow 'a set \Rightarrow bool (infix \subseteq'' 50)} \\
\text{where A \subseteq'' B \iff A \subseteq'' B \land A = \{\} \land B \in \text{infinite-complement-partition}}
\end{array}

\text{lemma set-less-eq-aux'-refl:}
\begin{array}{l}
A \subseteq'' A \iff A \in \text{infinite-complement-partition}
\end{array}

(\text{proof})

\text{lemma set-less-eq-aux'-antisym: [ A \subseteq'' B; B \subseteq'' A ] \implies A = B}

(\text{proof})

\text{lemma set-less-eq-aux'-infinite-complement-partitionD:}
\begin{array}{l}
A \subseteq'' B \implies A \in \text{infinite-complement-partition} \land B \in \text{infinite-complement-partition}
\end{array}

(\text{proof})

\text{lemma empty-set-less-eq-def [simp]: \{\} \subseteq'' B \iff B \in \text{infinite-complement-partition}}

(\text{proof})

end

context linorder begin
\text{lemma set-less-eq-aux'-trans: [ A \subseteq'' B; B \subseteq'' C ] \implies A \subseteq'' C}

(\text{proof})

\text{lemma set-less-eq-aux'-porder: partial-order-on infinite-complement-partition \{(A, B), A \subseteq'' B\}}

(\text{proof})

end

Extending op \subseteq to a total order on infinite-complement-partition

context ord begin
\text{definition set-less-eq-aux'': 'a set \Rightarrow 'a set \Rightarrow bool (infix \subseteq'''' 50)}


where set-less-eq-aux'' =
  (SOME sleq.
  (linear-order-on UNIV \{ (a, b). a \leq b \} \rightarrow linear-order-on infinite-complement-partition
  \{ (A, B). sleq A B \}) \land order-consistent \{ (A, B). A \subseteq'' B \} \{ (A, B). sleq A B \})

lemma set-less-eq-aux''-spec:
  shows linear-order \{ (a, b). a \leq b \} \rightarrow linear-order-on infinite-complement-partition
  \{ (A, B). A \subseteq'' B \}
  (is PROP "thesis1")
  and order-consistent \{ (A, B). A \subseteq'' B \} \{ (A, B). A \subseteq'' B \} (is "thesis2")
⟨proof⟩
end

context linorder begin

lemma set-less-eq-aux''-linear-order:
  linear-order-on infinite-complement-partition \{ (A, B). A \subseteq'' B \}
⟨proof⟩

lemma set-less-eq-aux''-refl [iff]: A \subseteq'' A \iff A \in infinite-complement-partition
⟨proof⟩

lemma set-less-eq-aux''-into-set-less-eq-aux'':
  assumes A \subseteq'' B
  shows A \subseteq'''' B
⟨proof⟩

lemma finite-set-less-eq-aux''-finite:
  assumes finite A and finite B
  shows A \subseteq'''' B \iff A \subseteq'''' B
⟨proof⟩

lemma set-less-eq-aux''-finite:
  finite (UNIV :: 'a set) \implies set-less-eq-aux'' = set-less-eq-aux
⟨proof⟩

lemma set-less-eq-aux''-antisym:
  [ A \subseteq'' B; B \subseteq'' A; A \in infinite-complement-partition; B \in infinite-complement-partition ]
  \implies A = B
⟨proof⟩

lemma set-less-eq-aux''-trans: [ A \subseteq'' B; B \subseteq'' C ] \implies A \subseteq'' C
⟨proof⟩

lemma set-less-eq-aux''-total:
  [ A \in infinite-complement-partition; B \in infinite-complement-partition ]
  \implies A \subseteq'' B \lor B \subseteq'' A
Extend \( \sqsubseteq'''' \) to cofinite sets

context ord begin

definition set-less-eq :: 'a set ⇒ 'a set ⇒ bool (infix \( \sqsubseteq \) 50)
where
\[
A \sqsubseteq B \iff (\text{if } A \in \text{infinite-complement-partition then } A \sqsubseteq'''' B \lor B \notin \text{infinite-complement-partition} \quad \text{else } B \notin \text{infinite-complement-partition} \land \neg B \sqsubseteq'''' A)
\]
definition set-less :: 'a set ⇒ 'a set ⇒ bool (infix \( \sqsubseteq \) 50)
where \( A \sqsubseteq B \iff A \subseteq B \land \neg B \subseteq A \)

lemma set-less-eq-def2: \[
A \subseteq B \iff (\text{if finite (UNIV :: 'a set) then } A \subseteq'''' B \quad \text{else if } A \in \text{infinite-complement-partition then } A \subseteq'''' B \lor B \notin \text{infinite-complement-partition} \quad \text{else } B \notin \text{infinite-complement-partition} \land \neg B \subseteq'''' A)
\]

end

context linorder begin

lemma set-less-eq-refl [iff]: \( A \subseteq A \)

lemma set-less-eq-antisym: \[ A \subseteq B; B \subseteq A \implies A = B \]

lemma set-less-eq-trans: \[ A \subseteq B; B \subseteq C \implies A \subseteq C \]

lemma set-less-eq-total: \( A \subseteq B \lor B \subseteq A \)

lemma set-less-eq-linorder: class.linorder op \( \subseteq \) op \( \subseteq \)

lemma set-less-eq-conv-set-less: set-less-eq A B \( \iff \) A = B \( \lor \) set-less A B

lemma Compl-set-less-eq-Compl: \( - A \subseteq - B \iff B \subseteq A \)
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

\textbf{lemma} Compl-set-less-Compl: \(- A \sqsubseteq - B \iff B \sqsubseteq A\)
\langle proof \rangle

\textbf{lemma} set-less-eq-finite-iff: [ finite A; finite B ] \implies A \sqsubseteq B \iff A \sqsubseteq ^' B
\langle proof \rangle

\textbf{lemma} set-less-finite-iff: [ finite A; finite B ] \implies A \sqsubseteq B \iff A \sqsubseteq ^' B
\langle proof \rangle

\textbf{lemma} infinite-set-less-eq-Complement:
[ finite A; finite B; \neg finite (UNIV :: 'a set)] \implies A \sqsubseteq - B
\langle proof \rangle

\textbf{lemma} infinite-set-less-Complement:
[ finite A; finite B; \neg finite (UNIV :: 'a set)] \implies A \sqsubseteq B
\langle proof \rangle

\textbf{lemma} infinite-Complement-set-less-eq:
[ finite A; finite B; \neg finite (UNIV :: 'a set)] \implies \neg A \sqsubseteq B
\langle proof \rangle

\textbf{lemma} infinite-Complement-set-less:
[ finite A; finite B; \neg finite (UNIV :: 'a set)] \implies \neg A \sqsubseteq B
\langle proof \rangle

\textbf{lemma} empty-set-less-eq [iff]: \{\} \sqsubseteq A
\langle proof \rangle

\textbf{lemma} set-less-eq-empty [iff]: A \sqsubseteq \{\} \iff A = \{\}
\langle proof \rangle

\textbf{lemma} empty-set-less-iff [iff]: \{\} \sqsubseteq A \iff A \neq \{\}
\langle proof \rangle

\textbf{lemma} not-set-less-empty [simp]: \neg A \sqsubseteq \{\}
\langle proof \rangle

\textbf{lemma} set-less-eq-UNIV [iff]: A \sqsubseteq UNIV
\langle proof \rangle

\textbf{lemma} UNIV-set-less-eq [iff]: UNIV \sqsubseteq A \iff A = UNIV
\langle proof \rangle

\textbf{lemma} set-less-UNIV-iff [iff]: A \sqsubseteq UNIV \iff A \neq UNIV
\langle proof \rangle

\textbf{lemma} not-UNIV-set-less [simp]: \neg UNIV \sqsubseteq A
\langle proof \rangle
2.6.2 Implementation based on sorted lists

type-synonym 'a proper-interval = 'a option ⇒ 'a option ⇒ bool

class proper-interval = ord +
  fixes proper-interval :: 'a proper-interval

class proper-interval = proper-intrel +
  assumes proper-interval-simps:
    proper-interval None None = True
    proper-interval None (Some y) = (∃ z. z < y)
    proper-interval (Some x) None = (∃ z. x < z)
    proper-interval (Some x) (Some y) = (∃ z. x < z ∧ z < y)

context proper-intrvl

begin

function set-less-eq-aux-Compl :: 'a option ⇒ 'a list ⇒ 'a list ⇒ bool
where
  set-less-eq-aux-Compl ao [] ys ←→ True
  | set-less-eq-aux-Compl ao xs [] ←→ True
  | set-less-eq-aux-Compl ao (x # xs) (y # ys) ←→
    (if x < y then proper-interval ao (Some x) xs (y # ys)
    else if y < x then proper-interval ao (Some y) ∨ set-less-eq-aux-Compl (Some y) (x # xs) ys
    else proper-interval ao (Some y))
⟨proof⟩
termination ⟨proof⟩

fun Compl-set-less-eq-aux :: 'a option ⇒ 'a list ⇒ 'a list ⇒ bool
where
  Compl-set-less-eq-aux ao [] [] ←→ ¬ proper-interval ao None
  | Compl-set-less-eq-aux ao [] (y # ys) ←→ ¬ proper-interval ao (Some y) ∧ Compl-set-less-eq-aux (Some y) [] ys
  | Compl-set-less-eq-aux ao (x # xs) [] ←→ ¬ proper-interval ao (Some x) ∧ Compl-set-less-eq-aux (Some x) xs []
  | Compl-set-less-eq-aux ao (x # xs) (y # ys) ←→
    (if x < y then ¬ proper-interval ao (Some x) ∧ Compl-set-less-eq-aux (Some x) ys (y # ys)
    else if y < x then ¬ proper-interval ao (Some y) ∧ Compl-set-less-eq-aux (Some y) (x # xs) ys
    else ¬ proper-interval ao (Some y))

fun set-less-aux-Compl :: 'a option ⇒ 'a list ⇒ 'a list ⇒ bool where
  set-less-aux-Compl ao [] [] ←→ proper-interval ao None
  | set-less-aux-Compl ao [] (y # ys) ←→ proper-interval ao (Some y) ∨ set-less-aux-Compl (Some y) [] ys
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

| set-less-aux-Compl ao (x # xs) [] ←→ proper-interval ao (Some x) ∨ set-less-aux-Compl (Some x) xs [] |
| set-less-aux-Compl ao (x # xs) (y # ys) ←→ (if x < y then proper-interval ao (Some x) ∨ set-less-aux-Compl (Some x) xs (y # ys) else if y < x then proper-interval ao (Some y) ∨ set-less-aux-Compl (Some y) (x # xs) ys else proper-interval ao (Some y)) |

function Compl-set-less-aux :: 'a option ⇒ 'a list ⇒ 'a list ⇒ bool where
Compl-set-less-aux ao [] ys ←→ False
Compl-set-less-aux ao xs [] ←→ False
Compl-set-less-aux ao (x # xs) (y # ys) ←→ (if x < y then ∼ proper-interval ao (Some x) ∧ Compl-set-less-aux (Some x) xs (y # ys) else if y < x then ∼ proper-interval ao (Some y) ∧ Compl-set-less-aux (Some y) (x # xs) ys else ∼ proper-interval ao (Some y))
⟨proof⟩
termination ⟨proof⟩
end
lemmas [code] =
  proper-intrel.set-less-eq-aux-Compl.simps
  proper-intrel.set-less-aux-Compl.simps
  proper-intrel.Compl-set-less-eq-aux.simps
  proper-intrel.Compl-set-less-aux.simps

class linorder-proper-interval = linorder + proper-interval
begin

theorem assumes fin: finite (UNIV :: 'a set)
  and xs: sorted xs  distinct xs
  and ys: sorted ys  distinct ys
shows set-less-eq-aux-Compl2-cone-set-less-eq-aux-Compl:
set xs ⊆' − set ys ←→ set-less-eq-aux-Compl None xs ys (is ?Compl2)
  and Compl1-set-less-eq-aux-cone-Compl-set-less-eq-aux:
  − set xs ⊆' set ys ←→ Compl-set-less-eq-aux None xs ys (is ?Compl1)
⟨proof⟩

lemma set-less-aux-Compl-iff:
set-less-aux-Compl ao xs ys ←→ set-less-eq-aux-Compl ao xs ys ∧ ∼ Compl-set-less-eq-aux ao ys xs
⟨proof⟩

lemma Compl-set-less-aux-Compl-iff:
Compl-set-less-aux-Compl ao xs ys ←→ Compl-set-less-eq-aux-Compl ao xs ys ∧ ∼ set-less-eq-aux-Compl ao ys xs
theorem assumes \( \text{fin} : \text{finite} \) (\( \text{UNIV} :: 'a \text{ set} \))
and \( \text{xs} : \text{sorted} \ \text{xs} \ \text{distinct} \ \text{xs} \)
and \( \text{ys} : \text{sorted} \ \text{ys} \ \text{distinct} \ \text{ys} \)
shows \( \text{set-less-aux-Compl2-conv-set-less-aux-Compl} : \)
set \( \text{xs} \subseteq ' - \text{set} \ \text{ys} \longleftrightarrow \text{set-less-aux-Compl None} \ \text{xs} \ \text{ys} \) (is \( ?\text{Compl2} \))
and \( \text{Compl1-set-less-aux-cone-Compl-set-less-aux} : \)
\( - \text{set} \ \text{xs} \subseteq ' - \text{set} \ \text{ys} \longleftrightarrow \text{Compl-set-less-aux None} \ \text{xs} \ \text{ys} \) (is \( ?\text{Compl1} \))
\( \langle \text{proof} \rangle \)
end

2.6.3 Implementation of proper intervals for sets

definition length-last :: 'a list \Rightarrow 'a list \Rightarrow nat \times 'a
where length-last \ x \ y = (length \ x, \ \text{last} \ x)

lemma length-last-Nil [code]: length-last \ [] = (0, \ \text{undefined})
\( \langle \text{proof} \rangle \)

lemma length-last-Cons-code [symmetric, code]:
\( \text{fold} (\lambda \ x \ (n, -) \ . \ (n + 1, \ x)) \ \text{xs} \ (1, \ x) = \text{length-last} \ (x \ # \ \text{xs}) \)
\( \langle \text{proof} \rangle \)

context proper-intrvl begin

fun exhaustive-above :: 'a \Rightarrow 'a list \Rightarrow bool
where exhaustible-above \ x \ [] \longleftrightarrow \neg \ \text{proper-interval} \ (\text{Some} \ x) \ \text{None}
| exhaustible-above \ x \ (y \ # \ \text{ys}) \longleftrightarrow \neg \ \text{proper-interval} \ (\text{Some} \ x) \ (\text{Some} \ y) \ \land \ \text{exhaustible-above} \ y \ \text{ys}

fun exhaustive :: 'a list \Rightarrow bool
where exhaustive [] = False
| exhaustive \ (x \ # \ \text{xs}) \longleftrightarrow \neg \ \text{proper-interval} \ \text{None} \ (\text{Some} \ x) \ \land \ \text{exhaustible-above} \ x \ \text{xs}

fun proper-interval-set-aux :: 'a list \Rightarrow 'a list \Rightarrow bool
where
\text{proper-interval-set-aux} \ x \ y \ [\ [] \ \longleftrightarrow \text{False}
| \text{proper-interval-set-aux} \ [\ y \ # \ \text{ys}] \ (y \ # \ \text{ys}) \longleftrightarrow \text{ys} \neq [] \ \lor \ \text{proper-interval} \ (\text{Some} \ y) \ \text{None}
| \text{proper-interval-set-aux} \ (x \ # \ \text{xs}) \ (y \ # \ \text{ys}) \longleftrightarrow
\text{if} \ x < y \ \text{then} \ False
\text{else if} \ y < x \ \text{then} \ \text{proper-interval} \ (\text{Some} \ y) \ (\text{Some} \ x) \ \lor \ \text{ys} \neq [] \ \lor \ \neg \ \text{exhaustible-above} \ x \ \text{xs}
\text{else} \ \text{proper-interval-set-aux} \ x \ \text{ys}

fun proper-interval-set-Compl-aux :: 'a option \Rightarrow nat \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool
where
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

\[ \text{proper-interval-set-Compl-aux ao n } \]  
\[ \text{CARD}'(a) > n + 1 \]  
\[ \text{proper-interval-set-Compl-aux ao n } \]  
\[ \text{CARD}'(a) - n; (\text{len-y, y'}) = \text{length-last} (y \# ys) \]  
\[ \text{in m \neq len-y \land (m = len-y + 1 \rightarrow \neg \text{proper-interval} (\text{Some y'} None))} \]  
\[ \text{proper-interval-set-Compl-aux ao n } \]  
\[ \text{CARD}'(a) - n; (\text{len-x, x'}) = \text{length-last} (x \# xs) \]  
\[ \text{in m \neq len-x \land (m = len-x + 1 \rightarrow \neg \text{proper-interval} (\text{Some x'} None))} \]  
\[ \text{proper-interval-set-Compl-aux ao n } \]  
\[ \text{CARD}'(a) - n; (\text{len-y, y'}) = \text{length-last} (y \# ys) \]  
\[ \text{let m = CARD} (\text{CARD}'(a)) - n; (\text{len-x, x'}) = \text{length-last} (x \# xs) \]  
\[ \text{if } x < y \text{ then } \neg \text{proper-interval ao (Some x)} \lor \text{proper-interval-set-Compl-aux ao (Some x)} (\text{Some x}) (n + 1) \text{ xs (y \# ys)} \]  
\[ \text{else if } y < x \text{ then } \neg \text{proper-interval ao (Some y)} \lor \text{proper-interval-set-Compl-aux ao (Some y)} (\text{Some y}) (n + 1) \text{ (x \# xs) ys} \]  
\[ \text{else } \neg \text{proper-interval ao (Some x)} \land (\text{ys} = [\]  
\[ \text{let m = card (UNIV :: 'a set) - n in m - length ys \neq 2 \lor m - length xs \neq 2}) \]  
\[ \text{fun } \text{proper-interval-Compl-set-aux :: 'a option } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow \text{bool} \]  
\[ \text{where} \]  
\[ \text{proper-interval-Compl-set-aux ao (x \# xs) (y \# ys) } \]  
\[ \text{if } x < y \text{ then } \neg \text{proper-interval ao (Some x)} \right\}  
\[ \text{else if } y < x \text{ then } \neg \text{proper-interval ao (Some y)} \right\}  
\[ \text{else } \neg \text{proper-interval ao (Some x)} \land (\text{ys} = [\]  
\[ \text{| proper-interval-Compl-set-aux ao - - } \right\} \text{False} \]  
\[ \text{end} \]  
\[ \text{lemmas [code] =} \]  
\[ \text{proper-intrel.exhaustive-above.simps} \]  
\[ \text{proper-intrel.exhaustive.simps} \]  
\[ \text{proper-intrel.proper-interval-set-Compl-aux.simps} \]  
\[ \text{proper-intrel.proper-interval-Compl-set-aux.simps} \]  
\[ \text{context linorder-proper-interval begin} \]  
\[ \text{lemma exhaustive-above-iff:} \]  
\[ [\text{sorted xs}; \text{distinct xs}; \forall x' \in \text{set xs. } x < x'] \rightarrow \text{exhaustive-above x xs } \right\} \text{set xs = \{} z. z > x \} \]  
\[ \langle \text{proof} \rangle \]  
\[ \text{lemma exhaustive-correct:} \]  
\[ \text{assumes sorted xs \quad \text{distinct xs}} \]
shows exhaustive $xs \iff \text{set } xs = \text{UNIV}$

\[\text{proof}\]

\textbf{Theorem} proper-interval-set-aux:
\begin{itemize}
  \item assumes fin: finite (UNIV :: 'a set)
  \item and $xs$: sorted $xs$ distinct $xs$
  \item and $ys$: sorted $ys$ distinct $ys$
  \item shows proper-interval-set-aux $xs$ $ys \iff (\exists A. \text{set } xs \sqsubseteq A \land A \sqsubseteq \text{set } ys)$
\end{itemize}
\[\text{proof}\]

\textbf{Lemma} proper-interval-set-Compl-aux:
\begin{itemize}
  \item assumes fin: finite (UNIV :: 'a set)
  \item and $xs$: sorted $xs$ distinct $xs$
  \item and $ys$: sorted $ys$ distinct $ys$
  \item shows proper-interval-set-Compl-aux None 0 $xs$ $ys \iff (\exists A. \text{set } xs \sqsubseteq A \land A \sqsubseteq \text{set } ys)$
\end{itemize}
\[\text{proof}\]

\textbf{Lemma} proper-interval-Compl-set-aux:
\begin{itemize}
  \item assumes fin: finite (UNIV :: 'a set)
  \item and $xs$: sorted $xs$ distinct $xs$
  \item and $ys$: sorted $ys$ distinct $ys$
  \item shows proper-interval-Compl-set-aux None $xs$ $ys \iff (\exists A. \neg\text{set } xs \sqsubseteq A \land A \sqsubseteq \text{set } ys)$
\end{itemize}
\[\text{proof}\]

\textbf{End}

\subsection{Proper intervals for HOL types}

\textbf{Instantiation} \textit{unit} :: proper-interval \begin{itemize}
  \item fun proper-interval-unit :: unit proper-interval where
    \begin{itemize}
      \item proper-interval-unit None None = True
      \item proper-interval-unit - - = False
    \end{itemize}
  \item instance \[proof\]
\end{itemize}

\textbf{Instantiation} \textit{bool} :: proper-interval \begin{itemize}
  \item fun proper-interval-bool :: bool proper-interval where
    \begin{itemize}
      \item proper-interval-bool (Some $x$) (Some $y$) \iff False
      \item proper-interval-bool (Some $x$) None \iff \neg x
      \item proper-interval-bool None (Some $y$) \iff y
      \item proper-interval-bool None None = True
    \end{itemize}
  \item instance \[proof\]
\end{itemize}

\textbf{Instantiation} \textit{nat} :: proper-interval \begin{itemize}
  \item fun proper-interval-nat :: nat proper-interval where
    \begin{itemize}
      \item proper-interval-nat no None = True
    \end{itemize}
\end{itemize}
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

| proper-interval-nat None (Some x) $\iff$ $x > 0$
| proper-interval-nat (Some x) (Some y) $\iff$ $y - x > 1$

instance ⟨proof⟩
end

instantiation int :: proper-interval begin
fun proper-interval-int :: int proper-interval where
  proper-interval-int (Some x) (Some y) $\iff$ $y - x > 1$
instance ⟨proof⟩
end

instantiation integer :: proper-interval begin
context includes integer.lifting begin
lift-definition proper-interval-integer :: integer proper-interval is proper-interval ⟨proof⟩
instance ⟨proof⟩
end
end
lemma proper-interval-integer-simps [code]:
  includes integer.lifting
  fixes x y :: integer and xo yo :: integer option
  shows
  proper-interval (Some x) (Some y) = ($1 < y - x$)
  proper-interval None yo = True
  proper-interval xo None = True
⟨proof⟩

instantiation natural :: proper-interval begin
context includes natural.lifting begin
lift-definition proper-interval-natural :: natural proper-interval is proper-interval ⟨proof⟩
instance ⟨proof⟩
end
end
lemma proper-interval-natural-simps [code]:
  includes natural.lifting
  fixes x y :: natural and xo :: natural option
  shows
  proper-interval xo None = True
  proper-interval None (Some y) $\iff$ $y > 0$
  proper-interval (Some x) (Some y) $\iff$ $y - x > 1$
⟨proof⟩

lemma le-Nibble0-iff: $x \leq$ Nibble0 $\iff$ $x =$ Nibble0 ⟨proof⟩

lemma Nibble0-less-iff: Nibble0 < $x$ $\iff$ $x \neq$ Nibble0 ⟨proof⟩

lemma le-NibbleF: $x \leq$ NibbleF ⟨proof⟩
lemma NibbleF-le-iff: Nibble ≤ x ↔ x = Nibble
⟨proof⟩

lemma nat-of-char-less-256: nat-of-char x < 256
⟨proof⟩

instantiation nibble :: proper-interval begin
fun proper-interval-nibble :: nibble proper-interval where
  proper-interval-nibble None None ↔ True
| proper-interval-nibble None (Some x) ↔ x ≠ Nibble0
| proper-interval-nibble (Some x) None ↔ x ≠ NibbleF
| proper-interval-nibble (Some x) (Some y) ↔ nat-of-nibble y - nat-of-nibble x > 1
instance
⟨proof⟩
end

lemma char-less-iff-nat-of-char: x < y ↔ nat-of-char x < nat-of-char y
(is ?lhs ↔ ?rhs)
⟨proof⟩

lemma nat-of-char-inject [simp]: nat-of-char x = nat-of-char y ↔ x = y
⟨proof⟩

lemma char-le-iff-nat-of-char: x ≤ y ↔ nat-of-char x ≤ nat-of-char y
⟨proof⟩

instantiation char :: proper-interval begin
fun proper-interval-char :: char proper-interval where
  proper-interval-char None None ↔ True
| proper-interval-char None (Some x) ↔ x ≠ Char Nibble0 Nibble0
| proper-interval-char (Some x) None ↔ x ≠ Char NibbleF NibbleF
| proper-interval-char (Some x) (Some y) ↔ nat-of-char y - nat-of-char x > 1
instance
⟨proof⟩
end

instantiation Enum.finite-1 :: proper-interval begin
definition proper-interval-finite-1 :: Enum.finite-1 proper-interval
where proper-interval-finite-1 x y ↔ x = None ∧ y = None
instance ⟨proof⟩
end

instantiation Enum.finite-2 :: proper-interval begin
fun proper-interval-finite-2 :: Enum.finite-2 proper-interval where
  proper-interval-finite-2 None None ↔ True
| proper-interval-finite-2 None (Some x) ↔ x = finite-2.a2
| proper-interval-finite-2 (Some x) None ↔ x = finite-2.a1
| proper-interval-finite-2 (Some x) (Some y) ↔ False
instance ⟨proof⟩
end

instantiation Enum.finite-3 :: proper-interval begin
fun proper-interval-finite-3 :: Enum.finite-3 proper-interval where
  proper-interval-finite-3 None None ↔ True
| proper-interval-finite-3 None (Some x) ↔ x ≠ finite-3.a1
| proper-interval-finite-3 (Some x) None ↔ x ≠ finite-3.a3
| proper-interval-finite-3 (Some x) (Some y) ↔ x = finite-3.a1 ∧ y = finite-3.a3
instance ⟨proof⟩
end

2.6.5 List fusion for the order and proper intervals on 'a set

definition length-last-fusion :: ('a, 's) generator ⇒ 's ⇒ nat × 'a
where length-last-fusion g s = length-last (list.unfoldr g s)

lemma length-last-fusion-code [code]:
  length-last-fusion g s =
  (if list.has-next g s then
    let (x, s') = list.next g s
    in fold-fusion g (λx (n, -). (n + 1, x)) s' (1, x)
  else (0, undefined))
⟨proof⟩

declare length-last-fusion-def [symmetric, code-unfold]

context proper-intrvl begin

definition set-less-eq-aux-Compl-fusion :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 'a option ⇒ 's1 ⇒ 's2 ⇒ bool
where
  set-less-eq-aux-Compl-fusion g1 g2 ao s1 s2 =
  set-less-eq-aux-Compl ao (list.unfoldr g1 s1) (list.unfoldr g2 s2)

definition Compl-set-less-eq-aux-fusion :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 'a option ⇒ 's1 ⇒ 's2 ⇒ bool
where
  Compl-set-less-eq-aux-fusion g1 g2 ao s1 s2 =
  Compl-set-less-eq-aux ao (list.unfoldr g1 s1) (list.unfoldr g2 s2)

definition set-less-aux-Compl-fusion :: ('a, 's1) generator ⇒ ('a, 's2) generator ⇒ 'a option ⇒ 's1 ⇒ 's2 ⇒ bool
where
  set-less-aux-Compl-fusion g1 g2 ao s1 s2 =
  set-less-aux-Compl ao (list.unfoldr g1 s1) (list.unfoldr g2 s2)

definition Compl-set-less-aux-fusion :: ('a, 's1) generator ⇒ ('a, 's2) generator
\(\Rightarrow \ 'a \text{ option} \Rightarrow \ 's1 \Rightarrow \ 's2 \Rightarrow \text{bool} \)

\[ \text{Compl-set-less-aux-fusion } g1 \ g2 \ ao \ s1 \ s2 = \text{Compl-set-less-aux } ao \big( \text{list.unfoldr } g1 \ s1 \big) \ (\text{list.unfoldr } g2 \ s2) \]

\textbf{definition} exhaustive-above-fusion :: \((a, 's) \text{ generator} \Rightarrow \ 'a \Rightarrow \ 's \Rightarrow \text{bool} \)

\[ \text{exhaustive-above-fusion } g \ a \ s = \text{exhaustive-above } a \big( \text{list.unfoldr } g \ s \big) \]

\textbf{definition} exhaustive-fusion :: \((a, 's) \text{ generator} \Rightarrow \ 's \Rightarrow \text{bool} \)

\[ \text{exhaustive-fusion } g \ s = \text{exhaustive } \big( \text{list.unfoldr } g \ s \big) \]

\textbf{definition} proper-interval-set-aux-fusion :: \((a, 's1) \text{ generator} \Rightarrow \ (a, 's2) \text{ generator} \Rightarrow \ 's1 \Rightarrow \ 's2 \Rightarrow \text{bool} \)

\[ \text{proper-interval-set-aux-fusion } g1 \ g2 \ s1 \ s2 = \text{proper-interval-set-aux } \big( \text{list.unfoldr } g1 \ s1 \big) \ (\text{list.unfoldr } g2 \ s2) \]

\textbf{definition} proper-interval-set-Compl-aux-fusion :: \((a, 's1) \text{ generator} \Rightarrow \ (a, 's2) \text{ generator} \Rightarrow \ 'a \text{ option} \Rightarrow \ \text{nat} \Rightarrow \ 's1 \Rightarrow \ 's2 \Rightarrow \text{bool} \)

\[ \text{proper-interval-set-Compl-aux-fusion } g1 \ g2 \ ao \ n \ s1 \ s2 = \text{proper-interval-set-Compl-aux } ao \ n \big( \text{list.unfoldr } g1 \ s1 \big) \ (\text{list.unfoldr } g2 \ s2) \]

\textbf{definition} proper-interval-Compl-set-aux-fusion :: \((a, 's1) \text{ generator} \Rightarrow \ (a, 's2) \text{ generator} \Rightarrow \ 'a \text{ option} \Rightarrow \ 's1 \Rightarrow \ 's2 \Rightarrow \text{bool} \)

\[ \text{proper-interval-Compl-set-aux-fusion } g1 \ g2 \ ao \ s1 \ s2 = \text{proper-interval-Compl-set-aux } ao \big( \text{list.unfoldr } g1 \ s1 \big) \ (\text{list.unfoldr } g2 \ s2) \]

\textbf{lemma} set-less-eq-aux-Compl-fusion-code:

\[ \text{set-less-eq-aux-Compl-fusion } g1 \ g2 \ ao \ s1 \ s2 \leftarrow \big( \text{if list.has-next } g1 \ s1 \ \text{then list.has-next } g2 \ s2 \ \text{if } x < y \ \text{then proper-interval } ao \ (\text{Some } x) \ \vee \ \text{set-less-eq-aux-Compl-fusion } g1 \ g2 (\text{Some } x) \ s1' \ s2' \ \text{else proper-interval } ao \ (\text{Some } y) \ \vee \ \text{set-less-eq-aux-Compl-fusion } g1 \ g2 (\text{Some } y) \ s1 \ s2' \big) \]

\text{proof}

\textbf{lemma} Compl-set-less-eq-aux-fusion-code:

\[ \text{Compl-set-less-eq-aux-fusion } g1 \ g2 \ ao \ s1 \ s2 \leftarrow \big( \text{if list.has-next } g1 \ s1 \ \text{then list.has-next } g2 \ s2 \ \text{if } x < y \ \text{then \ proper-interval } ao \ (\text{Some } x) \ \wedge \ \text{Compl-set-less-eq-aux-fusion} \big) \]
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

\( g_1 \ g_2 \ (\text{Some } x) \ s_1' \ s_2 \)

\( \text{else if } y < x \text{ then } \neg \text{proper-interval } ao (\text{Some } y) \wedge \text{Compl-set-less-eq-aux-fusion} \)

\( g_1 \ g_2 \ (\text{Some } y) \ s_1 \ s_2' \)

\( \text{else } \neg \text{proper-interval } ao (\text{Some } y) \)

\( \neg \text{proper-interval } ao (\text{Some } x) \wedge \text{Compl-set-less-eq-aux-fusion} \ g_1 \ g_2 \ (\text{Some } x) \ s_1' \ s_2 \)

\( \text{else if list.has-next } g_2 \ s_2 \text{ then} \)

\( \text{let } (y, s_2') = \text{list.next } g_2 \ s_2 \)

\( \text{in } \neg \text{proper-interval } ao (\text{Some } y) \wedge \text{Compl-set-less-eq-aux-fusion} \ g_1 \ g_2 \ (\text{Some } y) \ s_1 \ s_2' \)

\( \text{else } \neg \text{proper-interval } ao \text{ None} \)

\langle \text{proof} \rangle

\textbf{lemma} set-less-aux-Compl-fusion-code:

\( \text{set-less-aux-Compl-fusion } g_1 \ g_2 \ ao \ s_1 \ s_2 \leftrightarrow \)

\( (\text{if list.has-next } g_1 \ s_1 \text{ then} \)

\( \text{let } (x, s_1') = \text{list.next } g_1 \ s_1 \)

\( \text{in } \text{if list.has-next } g_2 \ s_2 \text{ then} \)

\( \text{let } (y, s_2') = \text{list.next } g_2 \ s_2 \)

\( \text{in } x < y \text{ then proper-interval } ao (\text{Some } x) \lor \text{set-less-aux-Compl-fusion} \)

\( g_1 \ g_2 \ (\text{Some } x) \ s_1' \ s_2 \)

\( \text{else if } y < x \text{ then proper-interval } ao (\text{Some } y) \lor \text{set-less-aux-Compl-fusion} \)

\( g_1 \ g_2 \ (\text{Some } y) \ s_1 \ s_2' \)

\( \text{else proper-interval } ao (\text{Some } y) \)

\( \text{else proper-interval } ao (\text{Some } x) \lor \text{set-less-aux-Compl-fusion} \ g_1 \ g_2 \ (\text{Some } x) \ s_1' \ s_2 \)

\( \text{else if list.has-next } g_2 \ s_2 \text{ then} \)

\( \text{let } (y, s_2') = \text{list.next } g_2 \ s_2 \)

\( \text{in proper-interval } ao (\text{Some } y) \lor \text{set-less-aux-Compl-fusion} \ g_1 \ g_2 \ (\text{Some } y) \ s_1 \ s_2' \)

\( \text{else proper-interval } ao \text{ None} \)

\langle \text{proof} \rangle

\textbf{lemma} Compl-set-less-aux-fusion-code:

\( \text{Compl-set-less-aux-fusion } g_1 \ g_2 \ ao \ s_1 \ s_2 \leftrightarrow \)

\( \text{list.has-next } g_1 \ s_1 \wedge \text{list.has-next } g_2 \ s_2 \wedge \)

\( (\text{let } (x, s_1') = \text{list.next } g_1 \ s_1; \)

\( (y, s_2') = \text{list.next } g_2 \ s_2 \)

\( \text{in } x < y \text{ then } \neg \text{proper-interval } ao (\text{Some } x) \wedge \text{Compl-set-less-aux-fusion} \ g_1 \ g_2 \ (\text{Some } x) \ s_1' \ s_2 \)

\( \text{else if } y < x \text{ then } \neg \text{proper-interval } ao (\text{Some } y) \wedge \text{Compl-set-less-aux-fusion} \)

\( g_1 \ g_2 \ (\text{Some } y) \ s_1 \ s_2' \)

\( \text{else } \neg \text{proper-interval } ao (\text{Some } y) \)

\langle \text{proof} \rangle

\textbf{lemma} exhaustive-above-fusion-code:

\( \text{exhaustive-above-fusion } g \ y \ s \leftrightarrow \)

\( (\text{if list.has-next } g \ s \text{ then} \)

\( \text{let } (x, s') = \text{list.next } g \ s \)
CHAPTER 2. AN EXECUTABLE LINEAR ORDER ON SETS

\[\begin{align*}
    &\text{in } \neg \text{proper-interval (Some y) (Some x)} \land \text{exhaustive-above-fusion } g \ x \ s' \\
    &\text{else } \neg \text{proper-interval (Some y) None}
\end{align*}\]

(\textit{proof})

\textbf{lemma} \textit{exhaustive-fusion-code}:
\[\text{exhaustive-fusion } g \ s =\]
\[\text{(list.has-next } g \ s \land\]
\[\text{(let } (x, \ s') = \text{list.next } g \ s\]
\[\text{in } \neg \text{proper-interval None (Some x) } \land \text{exhaustive-above-fusion } g \ x \ s')\]

(\textit{proof})

\textbf{lemma} \textit{proper-interval-set-aux-fusion-code}:
\[\text{proper-interval-set-aux-fusion } g1 \ g2 \ s1 \ s2 \leftrightarrow\]
\[\text{list.has-next } g2 \ s2 \land\]
\[\text{(let } (y, \ s2') = \text{list.next } g2 \ s2\]
\[\text{in if list.has-next } g1 \ s1 \text{ then}\]
\[\text{let } (x, \ s1') = \text{list.next } g1 \ s1\]
\[\text{in if } x < y \text{ then False}\]
\[\text{else if } y < x \text{ then proper-interval (Some y) (Some x) } \lor \text{list.has-next } g2\]
\[s2' \lor \neg \text{exhaustive-above-fusion } g1 \ x \ s1'\]
\[\text{else proper-interval-set-aux-fusion } g1 \ g2 \ s1' \ s2'\]
\[\text{else list.has-next } g2 \ s2' \lor \text{proper-interval (Some y) None}\]

(\textit{proof})

\textbf{lemma} \textit{proper-interval-set-Compl-aux-fusion-code}:
\[\text{proper-interval-set-Compl-aux-fusion } g1 \ g2 \ ao \ n \ s1 \ s2 \leftrightarrow\]
\[\text{(if list.has-next } g1 \ s1 \text{ then}\]
\[\text{let } (x, \ s1') = \text{list.next } g1 \ s1\]
\[\text{in if list.has-next } g2 \ s2 \text{ then}\]
\[\text{let } (y, \ s2') = \text{list.next } g2 \ s2\]
\[\text{in if } x < y \text{ then}\]
\[\text{proper-interval ao (Some x) } \lor\]
\[\text{proper-interval-set-Compl-aux-fusion } g1 \ g2 \ (Some x) \ (n + 1) \ s1' \ s2\]
\[\text{else if } y < x \text{ then}\]
\[\text{proper-interval ao (Some y) } \lor\]
\[\text{proper-interval-set-Compl-aux-fusion } g1 \ g2 \ (Some y) \ (n + 1) \ s1 \ s2'\]
\[\text{else}\]
\[\text{proper-interval ao (Some x) } \land\]
\[\text{(let } m = \text{CARD('a) } - \ n\]
\[\text{in } m - \text{length-fusion } g2 \ s2' \neq 2 \lor m - \text{length-fusion } g1 \ s1' \neq 2)\]
\[\text{else}\]
\[\text{let } m = \text{CARD('a) } - \ n; \ (\text{len-x, } x') = \text{length-last-fusion } g1 \ s1\]
\[\text{in } m \neq \text{len-x } \land (m = \text{len-x} + 1 \rightarrow \neg \text{proper-interval (Some x') None})\]

\textit{else if list.has-next } g2 \ s2 \text{ then}\]
\[\text{let } (y, \ s2') = \text{list.next } g2 \ s2;\]
\[m = \text{CARD('a) } - \ n;\]
\[\text{(len-y, } y') = \text{length-last-fusion } g2 \ s2\]
\[\text{in } m \neq \text{len-y } \land (m = \text{len-y} + 1 \rightarrow \neg \text{proper-interval (Some y') None})\]
2.6. AN EXECUTABLE LINEAR ORDER ON SETS

else \( \text{CARD}(a) > n + 1 \)

\(<proof>\)

\textbf{lemma} proper-interval-Compl-set-aux-fusion-code:

\(\text{proper-interval-Compl-set-aux-fusion } g_1 \ g_2 \ \text{ao } s_1 \ s_2 \iff \)

\(\text{list}.\text{has-next } g_1 \ s_1 \land \text{list}.\text{has-next } g_2 \ s_2 \land \)

\(\text{let } (x, s_1') = \text{list}.\text{next } g_1 \ s_1;\)

\(\quad (y, s_2') = \text{list}.\text{next } g_2 \ s_2\)

\(\text{in if } x < y \text{ then}\)

\(\quad \neg \text{proper-interval } \text{ao } (\text{Some } x) \land \text{proper-interval-Compl-set-aux-fusion } g_1 \ g_2\)

\(\quad (\text{Some } x) \; s_1' \; s_2\)

\(\quad \text{else if } y < x \text{ then}\)

\(\quad \neg \text{proper-interval } \text{ao } (\text{Some } y) \land \text{proper-interval-Compl-set-aux-fusion } g_1 \ g_2\)

\(\quad (\text{Some } y) \; s_1 \; s_2'\)

\(\quad \text{else } \neg \text{proper-interval } \text{ao } (\text{Some } x) \land (\text{list}.\text{has-next } g_2 \ s_2' \lor \text{list}.\text{has-next } g_1 \ s_1')\)

\(<proof>\)

\textbf{end}

\textbf{lemmas} [code] =

\(\text{set-less-eq-aux-Compl-fusion-code } \text{proper-intrel.set-less-eq-aux-Compl-fusion-code}\)

\(\text{Compl-set-less-eq-aux-fusion-code } \text{proper-intrel.Compl-set-less-eq-aux-fusion-code}\)

\(\text{set-less-Compl-fusion-code } \text{proper-intrel.set-less-Compl-fusion-code}\)

\(\text{Compl-set-less-aux-fusion-code } \text{proper-intrel.Compl-set-less-aux-fusion-code}\)

\(\text{exhaustive-above-fusion-code } \text{proper-intrel.exhaustive-above-fusion-code}\)

\(\text{exhaustive-fusion-code } \text{proper-intrel.exhaustive-fusion-code}\)

\(\text{proper-interval-set-aux-fusion-code } \text{proper-intrel.proper-interval-set-aux-fusion-code}\)

\(\text{proper-interval-set-Compl-aux-fusion-code } \text{proper-intrel.proper-interval-set-Compl-aux-fusion-code}\)

\(\text{proper-interval-Compl-set-aux-fusion-code } \text{proper-intrel.proper-interval-Compl-set-aux-fusion-code}\)

\textbf{lemmas} [symmetric, code-unfold] =

\(\text{set-less-eq-aux-Compl-fusion-def } \text{proper-intrel.set-less-eq-aux-Compl-fusion-def}\)

\(\text{Compl-set-less-eq-aux-fusion-def } \text{proper-intrel.Compl-set-less-eq-aux-fusion-def}\)

\(\text{set-less-Compl-fusion-def } \text{proper-intrel.set-less-Compl-fusion-def}\)

\(\text{Compl-set-less-aux-fusion-def } \text{proper-intrel.Compl-set-less-aux-fusion-def}\)

\(\text{exhaustive-above-fusion-def } \text{proper-intrel.exhaustive-above-fusion-def}\)

\(\text{exhaustive-fusion-def } \text{proper-intrel.exhaustive-fusion-def}\)

\(\text{proper-interval-set-aux-fusion-def } \text{proper-intrel.proper-interval-set-aux-fusion-def}\)

\(\text{proper-interval-set-Compl-aux-fusion-def } \text{proper-intrel.proper-interval-set-Compl-aux-fusion-def}\)

\(\text{proper-interval-Compl-set-aux-fusion-def } \text{proper-intrel.proper-interval-Compl-set-aux-fusion-def}\)

\textbf{2.6.6 Drop notation}

\textbf{context ord begin}

\textbf{no-notation} set-less-aux (infix \(\sqsubseteq\) 50)

\textbf{and} set-less-eq-aux (infix \(\sqsubseteq\) 50)

\textbf{and} set-less-eq-aux' (infix \(\sqsubseteq\)\* 50)
and set-less-eq-aux (infix $\sqsubseteq$ 50)
and set-less-eq (infix $\subseteq$ 50)
and set-less (infix $\subsetneq$ 50)

2.6.7 Introduction

In the following, we provide generators for the major classes of the container framework: ceq, corder, cenum, set-impl, and mapping-impl. In this file we provide some common infrastructure on the ML-level which will be used by the individual generators.

⟨ML⟩

end

theory Collection-Order
imports
  Set-Linorder
  Containers-Generator
begin
Chapter 3

Light-weight containers

3.1 A linear order for code generation

3.1.1 Optional comparators

```plaintext
class ccompare =  
  fixes ccompare :: 'a comparator option  
  assumes ccompare: \(\wedge\) comp. ccompare = Some comp \(\Rightarrow\) comparator comp  
begin  
abbreviation ccomp :: 'a comparator where ccomp \equiv\ the (ID ccompare)  
abbreviation cless :: 'a \(\Rightarrow\) 'a \(\Rightarrow\) bool where cless \equiv\ lt-of-comp (the (ID ccomp))  
abbreviation cless-eq :: 'a \(\Rightarrow\) 'a \(\Rightarrow\) bool where cless-eq \equiv\ le-of-comp (the (ID ccomp))  
end
```

```plaintext
lemma (in ccompare) ID-ccompare':  
  \(\wedge\) c. ID ccompare = Some c \(\Rightarrow\) comparator c  
⟨proof⟩
```

```plaintext
lemma (in ccompare) ID-ccompare:  
  \(\wedge\) c. ID ccompare = Some c \(\Rightarrow\) class.linorder (le-of-comp c) (lt-of-comp c)  
⟨proof⟩
```

```plaintext
syntax -CCOMPARE :: type \(\Rightarrow\) logic (  
  \(1\)CCOMPARE/(1'(\_'))))  
⟨ML⟩
```

```plaintext
definition is-ccompare :: 'a :: ccompare itself \(\Rightarrow\) bool  
  where is-ccompare \(\iff\) ID CCOMPARE('a) \(\neq\) None
```

```plaintext
context ccompare  
begin  
lemma cless-eq-conv-cless:  
  fixes a b :: 'a
```
```
assumes $ID CCOMPARE('a) \neq None$
shows $cless-eq a b \iff cless a b \lor a = b$
(proof)
end

3.1.2 Generator for the $ccompare$-class

This generator registers itself at the derive-manager for the class $ccompare$. To be more precise, one can choose whether one does not want to support a comparator by passing parameter "no", one wants to register an arbitrary type which is already in class $compare$ using parameter "compare", or one wants to generate a new comparator by passing no parameter. In the last case, one demands that the type is a datatype and that all non-recursive types of that datatype already provide a comparator, which can usually be achieved via "derive comparator type" or "derive compare type".

- instantiation $type :: (type,...,type) (no) \ corder$
- instantiation $datatype :: (type,...,type) \ corder$
- instantiation $datatype :: (compare,...,compare) (compare) \ corder$

If the parameter "no" is not used, then the corresponding $is-ccompare$-theorem is automatically generated and attributed with [simp, code-post].

To create a new comparator, we just invoke the functionality provided by the generator. The only difference is the boilerplate-code, which for the generator has to perform the class instantiation for a comparator, whereas here we have to invoke the methods to satisfy the corresponding locale for comparators.

This generator can be used for arbitrary types, not just datatypes. When passing no parameters, we get same limitation as for the order generator.

lemma $corder-intro: class.linorder le lt \implies a = Some (le, lt) \implies a = Some (le',lt') \implies class.linorder le' lt' \langle \text{proof} \rangle$

lemma $comparator-subst: c1 = c2 \implies comparator c1 \implies comparator c2 \langle \text{proof} \rangle$

lemma $\langle \text{in \ compare} \rangle \ comparator-subst: \forall \ comp. \ compare = comp \implies comparator comp$

\langle \text{proof} \rangle

\langle \text{ML} \rangle
3.1.3 Instantiations for HOL types

derive (linorder) compare-order
Enum.finite-1 Enum.finite-2 Enum.finite-3 natural String.literal
derive (compare) ccompare
unit bool nat int Enum.finite-1 Enum.finite-2 Enum.finite-3 integer natural nibble
char String.literal
derive (no) ccompare Enum.finite-4 Enum.finite-5

derive ccompare sum list option prod
derive (no) ccompare fun

lemma is-ccompare-fun [simp]: ¬ is-ccompare TYPE('a ⇒ 'b)
⟨proof⟩

instantiation set :: (ccompare) ccompare begin
definition CCOMPARE('a set) =
  map-option (λ c. comp-of-ords (ord.set-less-eq (le-of-comp c)) (ord.set-less (le-of-comp c)) (ID CCOMPARE('a))
instance ⟨proof⟩
end

lemma is-ccompare-set [simp, code-post]:
is-ccompare TYPE('a set) ↔ is-ccompare TYPE('a :: ccompare)
⟨proof⟩

definition cless-eq-set :: 'a :: ccompare set ⇒ 'a set ⇒ bool
where [simp, code del]: cless-eq-set = le-of-comp (the (ID CCOMPARE('a set)))
definition cless-set :: 'a :: ccompare set ⇒ 'a set ⇒ bool
where [simp, code del]: cless-set = lt-of-comp (the (ID CCOMPARE('a set)))

lemma ccompare-set-code [code]:
  CCOMPARE('a :: ccompare set) =
  (case ID CCOMPARE('a) of None ⇒ None | Some - ⇒ Some (comp-of-ords cless-eq-set cless-set))
⟨proof⟩
derive (no) ccompare Predicate.pred

3.1.4 Proper intervals

class cproper-interval = ccompare +
fixes cproper-interval :: 'a option ⇒ 'a option ⇒ bool
assumes cproper-interval:
  [ ID CCOMPARE('a) ≠ None; finite (UNIV :: 'a set) ]
  ⇒ class.proper-interval cless cproper-interval
begin
lemma ID-compare-interval:
[ ID CCOMPARE('a) = Some c; finite (UNIV :: 'a set) ]
⇒ class.linorder-proper-interval (le-of-comp c) (lt-of-comp c) cproper-interval
⟨proof⟩
end

instantiation unit :: cproper-interval begin
definition cproper-interval = (proper-interval :: unit proper-interval)
instance ⟨proof⟩
end

instantiation bool :: cproper-interval begin
definition cproper-interval = (proper-interval :: bool proper-interval)
instance ⟨proof⟩
end

instantiation nat :: cproper-interval begin
definition cproper-interval = (proper-interval :: nat proper-interval)
instance ⟨proof⟩
end

instantiation int :: cproper-interval begin
definition cproper-interval = (proper-interval :: int proper-interval)
instance ⟨proof⟩
end

instantiation integer :: cproper-interval begin
definition cproper-interval = (proper-interval :: integer proper-interval)
instance ⟨proof⟩
end

instantiation natural :: cproper-interval begin
definition cproper-interval = (proper-interval :: natural proper-interval)
instance ⟨proof⟩
end

instantiation nibble :: cproper-interval begin
definition cproper-interval = (proper-interval :: nibble proper-interval)
instance ⟨proof⟩
end

instantiation char :: cproper-interval begin
definition cproper-interval = (proper-interval :: char proper-interval)
instance ⟨proof⟩
end

instantiation Enum.finite-1 :: cproper-interval begin
3.1. A LINEAR ORDER FOR CODE GENERATION

definition \texttt{cproper-interval} = (\texttt{proper-interval :: Enum.finite-1 proper-interval})

instance ⟨proof⟩

end

instantiation \texttt{Enum.finite-2 :: cproper-interval} begin
definition \texttt{cproper-interval} = (\texttt{proper-interval :: Enum.finite-2 proper-interval})

instance ⟨proof⟩

end

instantiation \texttt{Enum.finite-3 :: cproper-interval} begin

definition \texttt{cproper-interval} = (\texttt{proper-interval :: Enum.finite-3 proper-interval})

instance ⟨proof⟩

end

instantiation \texttt{Enum.finite-4 :: cproper-interval} begin

definition (\texttt{cproper-interval :: Enum.finite-4 proper-interval}) - - = undefined

instance ⟨proof⟩

end

instantiation \texttt{Enum.finite-5 :: cproper-interval} begin

definition (\texttt{cproper-interval :: Enum.finite-5 proper-interval}) - - = undefined

instance ⟨proof⟩

end

lemma \texttt{lt-of-comp-sum}: \texttt{lt-of-comp (comparator-sum ca cb) sx sy} =

\begin{cases}
\text{case sx of Inl x ⇒ (case sy of Inl y ⇒ lt-of-comp ca x y | Inr y ⇒ True)} \hspace{1cm} | \text{Inr x ⇒ (case sy of Inl y ⇒ False | Inr y ⇒ lt-of-comp cb x y)}
\end{cases}

⟨proof⟩

instantiation \texttt{sum :: (cproper-interval, cproper-interval) cproper-interval} begin

fun \texttt{cproper-interval-sum :: ('a + 'b) proper-interval} where

\texttt{cproper-interval-sum None None} ←→ True

\begin{cases}
\text{cproper-interval-sum None (Some (Inl x))} ←→ cproper-interval None (Some x) \\
\text{cproper-interval-sum None (Some (Inr y))} ←→ True \\
\text{cproper-interval-sum (Some (Inl x)) None} ←→ True \\
\text{cproper-interval-sum (Some (Inl x)) (Some (Inl y))} ←→ cproper-interval (Some x) (Some y) \\
\text{cproper-interval-sum (Some (Inl x)) (Some (Inr y))} ←→ cproper-interval (Some x) (Some y) \\
\text{cproper-interval-sum (Some (Inr y)) None} ←→ cproper-interval (Some y) None \\
\text{cproper-interval-sum (Some (Inr y)) (Some (Inl x))} ←→ False \\
\text{cproper-interval-sum (Some (Inr y)) (Some (Inr y))} ←→ cproper-interval (Some x) (Some y)
\end{cases}

instance ⟨proof⟩

end

lemma \texttt{lt-of-comp-less-prod}: \texttt{lt-of-comp (comparator-prod c-a c-b)} =
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

less-prod (le-of-comp c-a) (lt-of-comp c-a) (lt-of-comp c-b) 
⟨proof⟩

lemma lt-of-comp-prod: lt-of-comp (comparator-prod c-a c-b) (x1, x2) (y1, y2) = 
(lt-of-comp c-a x1 y1 ∨ le-of-comp c-a x1 y1 ∧ lt-of-comp c-b x2 y2) 
⟨proof⟩

instantiation prod :: (cproper-interval, cproper-interval) cproper-interval begin
fun cproper-interval-prod :: ('a × 'b) proper-interval where
  cproper-interval-prod None None <-> True
| cproper-interval-prod None (Some (y1, y2)) <-> cproper-interval None (Some y1) ∨ cproper-interval None (Some y2)
| cproper-interval-prod (Some (x1, x2)) None <-> cproper-interval (Some x1)
  None ∨ cproper-interval (Some x2) None
| cproper-interval-prod (Some (x1, x2)) (Some (y1, y2)) <->
  cproper-interval (Some x1) (Some y1) ∨
  cless x1 y1 ∧ (cproper-interval (Some x2) None ∨ cproper-interval None (Some y2)) ∨
  ¬ cless y1 x1 ∧ cproper-interval (Some x2) (Some y2)
instance ⟨proof⟩
end

instantiation list :: (ccompare) cproper-interval begin
definition cproper-interval-list :: 'a list proper-interval where
  cproper-interval-list xso yso = undefined
instance ⟨proof⟩
end

lemma UNIV-literal-eq-range-STR: UNIV = range STR 
⟨proof⟩

lemma infinite-UNIV-literal: ¬ finite (UNIV :: String.literal set) 
⟨proof⟩

instantiation String.literal :: cproper-interval begin
definition cproper-interval-literal :: String.literal proper-interval where
  cproper-interval-literal xso yso = undefined
instance ⟨proof⟩
end

lemma lt-of-comp-option: lt-of-comp (comparator-option c) sx sy = 
  (case sx of None ⇒ (case sy of None ⇒ False | Some y ⇒ True)
  | Some x ⇒ (case sy of None ⇒ False | Some y ⇒ lt-of-comp c x y)) 
⟨proof⟩

instantiation option :: (cproper-interval) cproper-interval begin
fun cproper-interval-option :: 'a option proper-interval where
  cproper-interval-option None None ←→ True
| cproper-interval-option None (Some x) ←→ x ≠ None
| cproper-interval-option (Some x) None ←→ cproper-interval x None
| cproper-interval-option (Some x) (Some None) ←→ False
| cproper-interval-option (Some x) (Some (Some y)) ←→ cproper-interval x (Some y)
instance ⟨proof⟩
end

instantiation set :: (cproper-interval) cproper-interval begin
fun cproper-interval-set :: 'a set proper-interval where
  [code]: cproper-interval-set None None ←→ True
| [code]: cproper-interval-set None (Some B) ←→ (B ≠ {}) | cproper-interval-set-Some-Some [code del]: — Refine for concrete implementations
| [code]: cproper-interval-set (Some A) None ←→ (A ≠ UNIV)
| cproper-interval-set-Some-Some

  cproper-interval-set (Some A) (Some B) ←→ finite (UNIV :: 'a set) ∧ (∃ C. cless A C ∧ cless C B)
instance ⟨proof⟩
end

lemma Complement-cproper-interval-set-Complement:
  fixes A B :: 'a set
  assumes order: ID CCOMPARE('a) ≠ None
  shows cproper-interval (Some (− A)) (Some (− B)) = cproper-interval (Some B) (Some A)
⟨proof⟩
end

instantiation fun :: (type, type) cproper-interval begin

No interval checks on functions needed because we have not defined an order on them.

definition cproper-interval = (undefined :: ('a ⇒ 'b) proper-interval)
instance ⟨proof⟩
end

end

theory List-Proper-Interval imports
  ~~/src/HOL/Library/List-lexord
  Collection-Order
3.2 Instantiate proper-interval of for 'a list

lemma Nil-less-conv-neq-Nil: [] < xs ←→ xs ≠ []
⟨proof⟩

lemma less-append-same-iff:
  fixes xs :: 'a :: preorder list
  shows xs < xs @ ys ←→ [] < ys
⟨proof⟩

lemma less-append-same2-iff:
  fixes xs :: 'a :: preorder list
  shows xs @ ys < xs @ zs ←→ ys < zs
⟨proof⟩

lemma Cons-less-iff:
  fixes x :: 'a :: preorder shows 
x # xs < ys ←→ (∃ y ys'. ys = y # ys' ∧ (x < y ∨ x = y ∧ xs < ys'))
⟨proof⟩

instantiation list :: ({proper-interval, order}) proper-interval begin

definition proper-interval-list-aux :: 'a list ⇒ 'a list ⇒ bool
  where proper-interval-list-aux-correct:
    proper-interval-list-aux xs ys ←→ (∃ zs. xs < zs ∧ zs < ys)

lemma proper-interval-list-aux-simps [code]:
  proper-interval-list-aux [] [] ←→ False
  proper-interval-list-aux [] (y # ys) ←→ ys ≠ [] ∨ proper-interval None (Some y)
  proper-interval-list-aux (x # xs) (y # ys) ←→ x < y ∨ x = y ∧ proper-interval-list-aux xs ys
⟨proof⟩

fun proper-interval-list :: 'a list option ⇒ 'a list option ⇒ bool where
  proper-interval-list None None ←→ True
  | proper-interval-list None (Some xs) ←→ (xs ≠ [])
  | proper-interval-list (Some xs) None ←→ True
  | proper-interval-list (Some xs) (Some ys) ←→ proper-interval-list-aux xs ys

instance
⟨proof⟩
end

end

theory Collection-Eq imports
  Containers-Auxiliary
  Containers-Generator
3.3. A TYPE CLASS FOR OPTIONAL EQUALITY TESTING

../Deriving/Equality-Generator/Equality-Instances
begin

3.3 A type class for optional equality testing

class ceq =
  fixes ceq :: ('a ⇒ 'a ⇒ bool) option
  assumes ceq: ceq = Some eq =⇒ eq = op =
begin

lemma ceq-equality: ceq = Some eq =⇒ equality eq
⟨proof⟩

lemma ID-ceq: ID ceq = Some eq =⇒ eq = op =
⟨proof⟩

abbreviation ceq′ :: 'a ⇒ 'a ⇒ bool where ceq′ ≡ the (ID ceq)
end

syntax -CEQ :: type => logic ( (1CEQ/(1'(¬))))

⟨ML⟩

definition is-ceq :: 'a :: ceq itself ⇒ bool
where is-ceq -<⇒ ID CEQ('a) ≠ None

3.3.1 Generator for the ceq-class

This generator registers itself at the derive-manager for the class ceq. To be more precise, one can choose whether one wants to take op = as function for CEQ('a) by passing ”eq” as parameter, whether equality should not be supported by passing ”no” as parameter, or whether an own definition for equality should be derived by not passing any parameters. The last possibility only works for datatypes.

- instantiation type :: (type,...,type) (eq) ceq
- instantiation type :: (type,...,type) (no) ceq
- instantiation datatype :: (ceq,...,ceq) ceq

If the parameter ”no” is not used, then the corresponding is-ceq-theorem is also automatically generated and attributed with [simp, code-post].
This generator can be used for arbitrary types, not just datatypes.

lemma equality-subst: c1 = c2 =⇒ equality c1 =⇒ equality c2 ⟨proof⟩

⟨ML⟩
3.3.2 Type class instances for HOL types

derive \((eq) \ ceq \ unit\)

lemma [code]: \(CEQ(\text{unit}) = \text{Some} (\lambda - \cdot \ True)\)  
(proof)

derive \((eq) \ ceq\)
ool

nat

int

Enum.finite-1

Enum.finite-2

Enum.finite-3

Enum.finite-4

Enum.finite-5

integer

natural

nibble

char

String.literal
derive ceq sum prod list option

derive \((\text{no}) \ ceq \ fun\)

lemma \(\text{is-ceq-fun}\) [simp]: \(- \text{is-ceq TYPE}('a \Rightarrow 'b)\)
(proof)

definition \(\text{set-eq} :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}\)

where [code del]: \(\text{set-eq} = \text{op} =\)

lemma \(\text{set-eq-code}\):

shows [code]: \(\text{set-eq A B} \longleftrightarrow A \subseteq B \land B \subseteq A\)

and [code-unfold]: \(\text{op} = = \text{set-eq}\)
(proof)

instantiation \(\text{set} :: (ceq) \ ceq \begin{proof}\)
definition \(\text{CEQ}('a \text{ set}) = \text{case ID CEQ('a) of None \Rightarrow None | Some - \Rightarrow Some set-eq}\)

instance (proof)
\end

lemma \(\text{is-ceq-set}\) [simp, code-post]: \(\text{is-ceq TYPE('a \text{ set})} \longleftrightarrow \text{is-ceq TYPE('a :: ceq)}\)
(proof)

lemma \(\text{ID-ceq-set-not-None-iff}\) [simp]: \(\text{ID CEQ('a \text{ set})} \neq \text{None} \longleftrightarrow \text{ID CEQ('a :: ceq)} \neq \text{None}\)
(proof)

Instantiation for 'a Predicate.pred

correct fixes eq :: 'a \Rightarrow 'a \Rightarrow bool begin
3.3. A TYPE CLASS FOR OPTIONAL EQUALITY TESTING

definition member-pred : 'a Predicate.pred ⇒ 'a ⇒ bool
where member-pred P x ←→ (∃ y. eq x y ∧ Predicate.eval P y)

definition member-seq :: 'a Predicate.seq ⇒ 'a ⇒ bool
where member-seq xp = member-pred (Predicate.pred-of-seq xp)

lemma member-seq-code [code]:
member-seq seq.Empty x ←→ False
member-seq (seq.Insert y P) x ←→ eq x y ∨ member-pred P x
member-seq (seq.Join Q xq) x ←→ member-pred Q x ∨ member-seq xq x
⟨proof⟩

lemma member-pred-code [code]:
member-pred (Predicate.Seq f) = member-seq (f ())
⟨proof⟩

definition leq-pred :: 'a Predicate.pred ⇒ 'a Predicate.pred ⇒ bool
where leq-pred P Q ←→ (∀ x. Predicate.eval P x →→ (∃ y. eq x y ∧ Predicate.eval Q y))

definition leq-seq :: 'a Predicate.seq ⇒ 'a Predicate.pred ⇒ bool
where leq-seq xp Q ←→ leq-pred (Predicate.pred-of-seq xp) Q

lemma leq-seq-code [code]:
leq-seq seq.Empty Q ←→ True
leq-seq (seq.Insert x P) Q ←→ member-pred Q x ∧ leq-pred P Q
leq-seq (seq.Join P xp) Q ←→ leq-pred P Q ∧ leq-seq xp Q
⟨proof⟩

lemma leq-pred-code [code]:
leq-pred (Predicate.Seq f) Q ←→ leq-seq (f ()) Q
⟨proof⟩

definition predicate-eq :: 'a Predicate.pred ⇒ 'a Predicate.pred ⇒ bool
where predicate-eq P Q ←→ leq-pred P Q ∧ leq-pred Q P

context assumes eq: eq = op = begin

lemma member-pred-eq: member-pred = Predicate.eval
⟨proof⟩

lemma member-seq-eq: member-seq = Predicate.member
⟨proof⟩

lemma leq-pred-eq: leq-pred = op ≤
⟨proof⟩

lemma predicate-eq-eq: predicate-eq = op =
⟨proof⟩
3.4 A type class for optional enumerations

3.4.1 Definition

class cenum =
  fixes cEnum :: ('a list × ('a ⇒ bool) ⇒ bool) × (('a ⇒ bool) ⇒ bool)) option
  assumes UNIV-cenum: cEnum = Some (enum, enum-all, enum-ex) ⇒ UNIV = set enum
  and cenum-all-UNIV: cEnum = Some (enum, enum-all, enum-ex) ⇒ enum-all P = Ball UNIV P
  and cenum-ex-UNIV: cEnum = Some (enum, enum-all, enum-ex) ⇒ enum-ex P = Bex UNIV P
begin

lemma ID-cEnum:
  ID cEnum = Some (enum, enum-all, enum-ex)
  ⇒ UNIV = set enum ∧ enum-all = Ball UNIV ∧ enum-ex = Bex UNIV
<proof>

lemma in-cenum: ID cEnum = Some (enum, rest) ⇒ f ∈ set enum
<proof>

abbreviation cenum :: 'a list
where cenum ≡ fst (the (ID cEnum))

abbreviation cenum-all :: ('a ⇒ bool) ⇒ bool
where cenum-all ≡ fst (snd (the (ID cEnum)))

abbreviation cenum-ex :: ('a ⇒ bool) ⇒ bool
where cenum-ex ≡ snd (snd (the (ID cEnum)))
3.4. A TYPE CLASS FOR OPTIONAL ENUMERATIONS

syntax -CENUM :: type => logic ((1CENUM/(1'(-))))

⟨ML⟩

3.4.2 Generator for the cenum-class

This generator registers itself at the derive-manager for the class cenum. To be more precise, one can currently only choose to not support enumeration by passing "no" as parameter.

• instantiation type :: (type,...,type) (no) cenum

This generator can be used for arbitrary types, not just datatypes.

⟨ML⟩

3.4.3 Instantiations

context fixes cenum-all :: ('a => bool) => bool begin
fun all-n-lists :: ('a list => bool) => nat => bool
where [simp del]:
  all-n-lists P n = (if n = 0 then P [] else cenum-all (\x. all-n-lists (\xs. P (x # xs)) (n - 1)))
end

context fixes cenum-ex :: ('a => bool) => bool begin
fun ex-n-lists :: ('a list => bool) => nat => bool
where [simp del]:
ex-n-lists P n ←→ (if n = 0 then P [] else cenum-ex (%x. ex-n-lists (%xs. P (x # xs)) (n - 1)))
end

lemma all-n-lists-iff: fixes cenum shows
  all-n-lists (Ball (set cenum)) P n ←→ (∀ xs ∈ set (List.n-lists n cenum). P xs)
  ⟨proof⟩

lemma ex-n-lists-iff: fixes cenum shows
  ex-n-lists (Bex (set cenum)) P n ←→ (∃ xs ∈ set (List.n-lists n cenum). P xs)
  ⟨proof⟩

instantiation fun :: (cenum, cenum) cenum begin

definition
  CENUM('a => 'b) =
  (case ID CENUM('a) of None => None | Some (enum-a, enum-all-a, enum-ex-a) ⇒
case ID CENUM ('a) of None ⇒ None | Some (enum-a, enum-all-a, enum-ex-a)
⇒ Some
  (map (λys. the o map-of (zip enum-a ys)) (List.n-lists (length enum-a) enum-b),
  λP. all-n-lists enum-all-b (λbs. P (the o map-of (zip enum-a bs))) (length enum-a),
  λP. ex-n-lists enum-ex-b (λbs. P (the o map-of (zip enum-a bs))) (length enum-a))
instance ⟨proof⟩
end

instantiation set :: (cenum) cenum begin
definition CENUM ('a set) =
  (case ID CENUM ('a) of None ⇒ None | Some (enum-a, enum-all-a, enum-ex-a)
⇒ Some
  (map set (sublists enum-a),
  λP. list-all P (map set (sublists enum-a)),
  λP. list-ex P (map set (sublists enum-a))))
instance ⟨proof⟩
end

instantiation unit :: cenum begin
definition CENUM (unit) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance ⟨proof⟩
end

instantiation bool :: cenum begin
definition CENUM (bool) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance ⟨proof⟩
end

instantiation prod :: (cenum, cenum) cenum begin
definition CENUM ('a × 'b) =
  (case ID CENUM ('a) of None ⇒ None | Some (enum-a, enum-all-a, enum-ex-a)
⇒ Some
  (List.product enum-a enum-b, 
  AP. enum-all-a (%x. enum-all-b (%y. P (x, y))),
  AP. enum-ex-a (%x. enum-ex-b (%y. P (x, y))))
instance ⟨proof⟩
end

instantiation sum :: (cenum, cenum) cenum begin
definition
3.4. A TYPE CLASS FOR OPTIONAL ENUMERATIONS

\[
CENUM(\texttt{a} + \texttt{b}) = \\
\text{case ID } CENUM(\texttt{a}) \text{ of None } \Rightarrow \text{ None } | \text{ Some (enum-a, enum-all-a, enum-ex-a)} \\
\Rightarrow \\
\text{case ID } CENUM(\texttt{b}) \text{ of None } \Rightarrow \text{ None } | \text{ Some (enum-b, enum-all-b, enum-ex-b)} \\
\Rightarrow \\
\text{Some (map Inl enum-a @ map Inr enum-b,} \\
\lambda P. \text{enum-all-a (lx. } P \text{ (Inl x)) } \land \text{ enum-all-b (lx. } P \text{ (Inr x)),} \\
\lambda P. \text{enum-ex-a (lx. } P \text{ (Inl x)) } \lor \text{ enum-ex-b (lx. } P \text{ (Inr x)))}
\]

\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation } \text{option} :: (cenum) \text{ cenum begin}
\text{definition } CENUM(\texttt{a option}) = \\
\text{case ID } CENUM(\texttt{a}) \text{ of None } \Rightarrow \text{ None } | \text{ Some (enum-a, enum-all-a, enum-ex-a)} \\
\Rightarrow \\
\text{Some (None } \# \text{ map Some enum-a,} \\
\lambda P. P \text{ None } \land \text{ enum-all-a (lx. } P \text{ (Some x)),} \\
\lambda P. P \text{ None } \lor \text{ enum-ex-a (lx. } P \text{ (Some x)))}
\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation Enum.finite-1 :: cenum begin}
\text{definition } CENUM(\text{Enum.finite-1}) = \text{ Some (enum-class.enum, enum-class.enum-all,} \\
\text{enum-class.enum-ex)}
\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation Enum.finite-2 :: cenum begin}
\text{definition } CENUM(\text{Enum.finite-2}) = \text{ Some (enum-class.enum, enum-class.enum-all,} \\
\text{enum-class.enum-ex)}
\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation Enum.finite-3 :: cenum begin}
\text{definition } CENUM(\text{Enum.finite-3}) = \text{ Some (enum-class.enum, enum-class.enum-all,} \\
\text{enum-class.enum-ex)}
\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation Enum.finite-4 :: cenum begin}
\text{definition } CENUM(\text{Enum.finite-4}) = \text{ Some (enum-class.enum, enum-class.enum-all,} \\
\text{enum-class.enum-ex)}
\text{instance \langle \text{proof} \rangle}
\end

\text{instantiation Enum.finite-5 :: cenum begin}
definition CENUM (Enum.finite-5) = Some (enum-class.enum, enum-class.enum-all, 
enum-class.enum-ex)
instance ⟨proof⟩
end

instantiation nibble :: cenum begin
definition CENUM (nibble) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance ⟨proof⟩
end

instantiation char :: cenum begin
definition CENUM (char) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance ⟨proof⟩
end

derive (no) cenum list nat int integer natural String.literal
end

theory Equal imports Main begin

3.5 Locales to abstract over HOL equality

locale equal-base = fixes equal :: 'a ⇒ 'a ⇒ bool

locale equal = equal-base +
  assumes equal-eq: equal = op =
begin
lemma equal-conv-eq: equal x y ⟷ x = y
⟨proof⟩
end
end

theory RBT-ext imports
~~/src/HOL/Library/RBT-Impl
Containers-Auxiliary
List-Fusion
begin
3.6. MORE ON RED-BLACK TREES

3.6 More on red-black trees

3.6.1 More lemmas

context linorder begin

lemma is-rbt-fold-rbt-insert:
  is-rbt t =⇒ is-rbt (RBT-Impl.fold rbt-insert t' t)
⟨proof⟩

lemma rbt-sorted-fold-insert:
  rbt-sorted t =⇒ rbt-sorted (RBT-Impl.fold rbt-insert t' t)
⟨proof⟩

lemma rbt-lookup-rbt-insert':
  rbt-sorted t =⇒ rbt-lookup (rbt-insert k v t) = rbt-lookup t(k := v)
⟨proof⟩

lemma rbt-lookup-fold-rbt-insert:
  rbt-sorted t2 =⇒ rbt-lookup (RBT-Impl.fold rbt-insert t1 t2) = rbt-lookup t2 ++ map-of (rev (RBT-Impl.entries t1))
⟨proof⟩

end

3.6.2 Build the cross product of two RBTs

context fixes f :: 'a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e begin

definition alist-product :: ('a × 'b) list ⇒ ('c × 'd) list ⇒ (('a × 'c) × 'e) list
where alist-product xs ys = concat (map (λ(a, b). map (λ(c, d). ((a, c), f a b c d)) ys) xs)

lemma alist-product-simps [simp]:
  alist-product [] ys = []
alist-product xs [] = []
alist-product ((a, b) # xs) ys = map (λ(c, d). ((a, c), f a b c d)) ys @ alist-product xs ys
⟨proof⟩

lemma append-alist-product-conv-fold:
  zs @ alist-product xs ys = rev (fold (λ(a, b). fold (λ(c, d) rest. ((a, c), f a b c d) # rest) ys) xs (rev zs))
⟨proof⟩

lemma alist-product-code [code]:
alist-product xs ys =
  rev (fold (λ(a, b). fold (λ(c, d) rest. ((a, c), f a b c d) # rest) ys) xs [])
⟨proof⟩
**Lemma** set-alist-product:

\[
\text{set } (\text{alist-product } xs \ ys) = \\
(\lambda ((a, b), (c, d)). ((a, c), f a b c d)) \ (\text{set } xs \times \text{set } ys)
\]

**Lemma** distinct-alist-product:

\[
[ \quad \text{distinct } (\text{map fst } xs); \text{distinct } (\text{map fst } ys) \quad ] \\
\implies \text{distinct } (\text{map fst } (\text{alist-product } xs \ ys))
\]

**Lemma** map-of-alist-product:

\[
\text{map-of } (\text{alist-product } xs \ ys) (a, c) = \\
(\text{case map-of } xs a \ of \text{None } \Rightarrow \text{None} \\
| \text{Some } b \Rightarrow \text{map-option } (f a b c) (\text{map-of } ys c))
\]

**Definition** rbt-product :: \((\mathbf{a}, \mathbf{b})\) rbt \Rightarrow \((\mathbf{c}, \mathbf{d})\) rbt \Rightarrow \((\mathbf{a} \times \mathbf{c}, \mathbf{e})\) rbt

\[
\text{where } \\
\text{rbt-product } rbt1 \ rbt2 = \text{rbtreeify } (\text{alist-product } (\text{RBT-Impl. entries } rbt1) (\text{RBT-Impl. entries } rbt2))
\]

**Lemma** rbt-product-code [code]:

\[
\text{rbt-product } rbt1 \ rbt2 = \\
\text{rbtreeify } (\text{rev } (\text{RBT-Impl. fold } (\lambda a b. RBT-Impl. fold (\lambda c d. rest. ((a, c), f a b c d) \neq \text{rest}) \ rbt2) \ rbt1 []))
\]

**Context**

\[
\text{fixes } \leq-a :: \mathbf{a} \Rightarrow \mathbf{a} \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50) \\
\text{and } \less-a :: \mathbf{a} \Rightarrow \mathbf{a} \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50) \\
\text{and } \leq-b :: \mathbf{b} \Rightarrow \mathbf{b} \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50) \\
\text{and } \less-b :: \mathbf{b} \Rightarrow \mathbf{b} \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50)
\]

\[
\text{assumes } \text{lin-a: class.linorder } \leq-a \ \\
\text{and } \text{lin-b: class.linorder } \leq-b \ \\
\text{begin}
\]

\[
\text{abbreviation } (\text{input}) \text{less-eq-prod'} :: (\mathbf{a} \times \mathbf{b}) \Rightarrow (\mathbf{a} \times \mathbf{b}) \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50)
\]

\[
\text{where } \text{less-eq-prod'} \equiv \text{less-eq-prod } \leq-a \ \less-a \ \leq-b
\]

\[
\text{abbreviation } (\text{input}) \text{less-prod'} :: (\mathbf{a} \times \mathbf{b}) \Rightarrow (\mathbf{a} \times \mathbf{b}) \Rightarrow \text{bool } (\text{infix } \sqsubseteq 50)
\]

\[
\text{where } \text{less-prod'} \equiv \text{less-prod } \leq-a \ \less-a \ \less-b
\]

\[
\text{lemmas } \text{linorder-prod} = \text{linorder-prod}[\text{OF } \text{lin-a } \text{lin-b}]
\]

**Lemma** sorted-alist-product:

\[
\text{assumes } xs: \text{linorder.sorted } \leq-a (\text{map fst } xs) \quad \text{distinct } (\text{map fst } xs)
\]
and ys: linorder.sorted op ⊒ (map fst ys)

shows linorder.sorted op ⊒ (map fst (alist-product f xs ys))

⟨proof⟩

lemma is-rbt-rbt-product:
[ ord.is-rbt op ⊏ a rbt1; ord.is-rbt op ⊏ b rbt2 ]

⇒ ord.is-rbt op ⊏ (rbt-product f rbt1 rbt2)

⟨proof⟩

lemma rbt-lookup-rbt-product:
[ ord.is-rbt op ⊏ a rbt1; ord.is-rbt op ⊏ b rbt2 ]

⇒ ord.rbt-lookup op ⊏ (rbt-product f rbt1 rbt2) (a, c) =
    (case ord.rbt-lookup op ⊏ a rbt1 a of None ⇒ None
        | Some b ⇒ map-option (f a b c) (ord.rbt-lookup op ⊏ b rbt2 c))

⟨proof⟩

end

hide-const less-eq-prod' less-prod'

3.6.3 Build an RBT where keys are paired with themselves

primrec RBT-Impl-diag :: ('a', 'b) rbt ⇒ ('a × 'a, 'b) rbt
where
    RBT-Impl-diag rbt.Empty = rbt.Empty
| RBT-Impl-diag (rbt.Branch c l k v r) = rbt.Branch c (RBT-Impl-diag l) (k, k) v (RBT-Impl-diag r)

lemma entries-RBT-Impl-diag:
    RBT-Impl.entries (RBT-Impl-diag t) = map (λ(k, v). ((k, k), v)) (RBT-Impl.entries t)

⟨proof⟩

lemma keys-RBT-Impl-diag:
    RBT-Impl.keys (RBT-Impl-diag t) = map (λk. (k, k)) (RBT-Impl.keys t)

⟨proof⟩

lemma rbt-sorted-RBT-Impl-diag:
    ord.rbt-sorted lt t ⇒ ord.rbt-sorted (less-prod leq lt lt) (RBT-Impl-diag t)

⟨proof⟩

lemma bheight-RBT-Impl-diag:
    bheight (RBT-Impl-diag t) = bheight t

⟨proof⟩

lemma inv-RBT-Impl-diag:
    assumes inv1 t inv2 t
    shows inv1 (RBT-Impl-diag t) inv2 (RBT-Impl-diag t)
    and color-of t = color.B ⇒ color-of (RBT-Impl-diag t) = color.B
(proof)

lemma is-rbt-RBT-Impl-diag:
ord.is-rbt lt t =⇒ ord.is-rbt (less-prod leq lt lt) (RBT-Impl-diag t)
(proof)

lemma (in linorder) rbt-lookup-RBT-Impl-diag:
ord.rbt-lookup (less-prod op ≤ op < op <) (RBT-Impl-diag t) =
(λ(k, k'). if k = k' then ord.rbt-lookup op < t k else None)
(proof)

3.6.4 Folding and quantifiers over RBTs

definition RBT-Impl-fold1 :: ('a ⇒ 'a ⇒ 'a) ⇒ ('a × 'a) RBT-Impl.rbta ⇒ 'a
where RBT-Impl-fold1 f rbt = fold f (tl (RBT-Impl.keys rbt)) (hd (RBT-Impl.keys rbt))

lemma RBT-Impl-fold1-simps [simp, code]:
RBT-Impl-fold1 f rbt.Empty = undefined
RBT-Impl-fold1 f (Branch c rbt.Empty k v r) = RBT-Impl.fold (λk v. f k) r k
RBT-Impl-fold1 f (Branch c (Branch c' l' k' v' r') k v r) =
RBT-Impl.fold (λk v. f k) r (f k (RBT-Impl-fold1 f (Branch c' l' k' v' r')))}
(proof)

definition RBT-Impl-rbt-all :: ('a ⇒ 'b ⇒ bool) ⇒ ('a × 'b) RBT-Impl.rbta ⇒ bool
where [code del]: RBT-Impl-rbt-all P rbt = (∀(k, v) ∈ set (RBT-Impl.entries rbt). P k v)

lemma RBT-Impl-rbt-all-simps [simp, code]:
RBT-Impl-rbt-all P rbt.Empty = True
RBT-Impl-rbt-all P (Branch c l k v r) = P k v ∧ RBT-Impl-rbt-all P l ∧
RBT-Impl-rbt-all P r
(proof)

definition RBT-Impl-rbt-ex :: ('a ⇒ 'b ⇒ bool) ⇒ ('a × 'b) RBT-Impl.rbta ⇒ bool
where [code del]: RBT-Impl-rbt-ex P rbt = (∃(k, v) ∈ set (RBT-Impl.entries rbt). P k v)

lemma RBT-Impl-rbt-ex-simps [simp, code]:
RBT-Impl-rbt-ex P rbt.Empty = False
RBT-Impl-rbt-ex P (Branch c l k v r) = P k v ∨ RBT-Impl-rbt-ex P l ∨
RBT-Impl-rbt-ex P r
(proof)

3.6.5 List fusion for RBTs

type-synonym ('a, 'b, 'c) rbt-generator-state = ('c × ('a, 'b) RBT-Impl.rbta) list
× ('a, 'b) RBT-Impl.rbta

fun rbt-has-next :: ('a, 'b, 'c) rbt-generator-state ⇒ bool
3.6. MORE ON RED-BLACK TREES

where
\[ \text{rbt-has-next} ([], \text{rbt.} \text{Empty}) = \text{False} \]
\[ \text{rbt-has-next} = \text{True} \]

fun rbt-keys-next :: ('a, 'b, 'a) rbt-generator-state \[\Rightarrow\] 'a × ('a, 'b, 'a) rbt-generator-state
where
\[ \text{rbt-keys-next} ((k, t) \# \text{kt}, \text{rbt.} \text{Empty}) = (k, \text{kt}, t) \]
\[ \text{rbt-keys-next} (\text{kt}, \text{rbt.} \text{Branch} c l k v r) = \text{rbt-keys-next} ((k, v) \# \text{kt}, l) \]

lemma rbt-generator-induct [case-names empty split shuffle]:
assumes \[ P ([], \text{rbt.} \text{Empty}) \]
and \[ \forall k \ t \ \text{kt}. \ P (\text{kt}, t) \Rightarrow P ((k, t) \# \text{kt}, \text{rbt.} \text{Empty}) \]
and \[ \forall \text{kt} \ c \ l \ k \ v \ r. \ P ((f k v, r) \# \text{kt}, l) \Rightarrow P (\text{kt}, \text{Branch} c l k v r) \]
shows \[ P \text{kt} \]

proof

lemma terminates-rbt-keys-generator:
terminates (rbt-has-next, rbt-keys-next)

proof

lift-definition rbt-keys-generator :: ('a, ('a, 'b, 'a) rbt-generator-state) generator
is (rbt-has-next, rbt-keys-next)

proof

definition rbt-init :: ('a, 'b) rbt \[\Rightarrow\] ('a, 'b, 'c) rbt-generator-state
where
\[ \text{rbt-init} = \text{Pair} [] \]

lemma has-next-rbt-keys-generator [simp]:
\[ \text{list.has-next rbt-keys-generator} = \text{rbt-has-next} \]

proof

lemma next-rbt-keys-generator [simp]:
\[ \text{list.next rbt-keys-generator} = \text{rbt-keys-next} \]

proof

lemma unfoldr-rbt-keys-generator-aux:
\[ \text{list.unfoldr rbt-keys-generator} (\text{kt}, t) = \text{RBT-Impl.keys} t \& \text{concat} (\map (\lambda (k, t). \text{kt} \text{RBT-Impl.keys} t) \text{kt}) \]

proof

corollary unfoldr-rbt-keys-generator:
\[ \text{list.unfoldr rbt-keys-generator} (\text{rbt-init} t) = \text{RBT-Impl.keys} t \]

proof

fun rbt-entries-next ::
\[ ('a, 'b, 'a \times 'b) rbt-generator-state \[\Rightarrow\] ('a \times 'b) \times ('a, 'b, 'a \times 'b) rbt-generator-state \]
where
\[ \text{rbt-entries-next} ((\text{kv}, t) \# \text{kt}, \text{rbt.} \text{Empty}) = (\text{kv}, \text{kt}, t) \]
\[ \text{rbt-entries-next} (\text{kt}, \text{rbt.} \text{Branch} c l k v r) = \text{rbt-entries-next} (((k, v), r) \# \text{kt}, l) \]
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

lemma terminates-rbt-entries-generator:
  terminates (rbt-has-next, rbt-entries-next)
⟨proof⟩

lift-definition rbt-entries-generator :: ('a × 'b, ('a, 'b, 'a × 'b) rbt-generator-state) generator
  is (rbt-has-next, rbt-entries-next)
⟨proof⟩

lemma has-next-rbt-entries-generator [simp]:
  list.has-next rbt-entries-generator = rbt-has-next
⟨proof⟩

lemma next-rbt-entries-generator [simp]:
  list.next rbt-entries-generator = rbt-entries-next
⟨proof⟩

lemma unfoldr-rbt-entries-generator-aux:
  list.unfoldr rbt-entries-generator (kts, t) =
  RBT-Impl.entries t @ concat (map (λ(k, t). k # RBT-Impl.entries t) kts)
⟨proof⟩

corollary unfoldr-rbt-entries-generator:
  list.unfoldr rbt-entries-generator (rbt-init t) = RBT-Impl.entries t
⟨proof⟩

end

theory RBT-Mapping2
imports
  Collection-Order
  RBT-ext
  ../Deriving/Comparator-Generator/RBT-Comparator-Impl
begin

3.7 Mappings implemented by red-black trees

lemma distinct-map-filterI: distinct (map f xs) ⇒ distinct (map f (filter P xs))
⟨proof⟩

lemma map-of-filter-apply:
  distinct (map fst xs)
  ⇒ map-of (filter P xs) k =
  (case map-of xs k of None ⇒ None | Some v ⇒ if P (k, v) then Some v else None)
⟨proof⟩
3.7. MAPPINGS IMPLEMENTED BY RED-BLACK TREES

3.7.1 Type definition

typedef ('a, 'b) mapping-rbt
    = { t :: ('a :: ccompare, 'b) RBT-Impl.rbt. ord.rbt cless t ∨ ID CCOMPARE('a) = None }

  morphisms impl-of Mapping-RBT'

  definition Mapping-RBT :: ('a :: ccompare, 'b) rbt ⇒ ('a, 'b) mapping-rbt
    where
      Mapping-RBT t = Mapping-RBT'
      (if ord.is-rbt cless t ∨ ID CCOMPARE('a) = None then t
       else RBT-Impl.fold (ord.rbt-insert cless) t rbt.Empty)

  lemma Mapping-RBT-inverse:
    fixes y :: ('a :: ccompare, 'b) rbt
    assumes y ∈ { t. ord.is-rbt cless t ∨ ID CCOMPARE('a) = None }
    shows impl-of (Mapping-RBT y) = y

  lemma impl-of-inverse: Mapping-RBT (impl-of t) = t

  lemma type-definition-mapping-rbt'::
    type-definition impl-of Mapping-RBT
    { t :: ('a, 'b) rbt. ord.is-rbt cless t ∨ ID CCOMPARE('a :: ccompare) = None }

  lemmas Mapping-RBT-cases[cases type: mapping-rbt] =
    type-definition.Abs-cases[OF type-definition-mapping-rbt']
  and Mapping-RBT-induct[induct type: mapping-rbt] =
    type-definition.Abs-induct[OF type-definition-mapping-rbt'] and
    Mapping-RBT-inject = type-definition.Abs-inject[OF type-definition-mapping-rbt']

  lemma rbt-eq-iff:
    t1 = t2 ←→ impl-of t1 = impl-of t2

  lemma rbt-eqI:
    impl-of t1 = impl-of t2 ⇒ t1 = t2

  lemma Mapping-RBT-impl-of [simp]:
    Mapping-RBT (impl-of t) = t

3.7.2 Operations

setup-lifting type-definition-mapping-rbt'
context fixes dummy :: 'a :: ccompare begin

lift-definition lookup :: (′a, ′b) mapping-rbt ⇒ ′a ⇒ ′b is rbt-comp-lookup ccomp
⟨proof⟩

lift-definition empty :: (′a, ′b) mapping-rbt is RBT-Impl.Empty
⟨proof⟩

lift-definition insert :: ′a ⇒ ′b ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b) mapping-rbt is
rbt-comp-insert ccomp
⟨proof⟩

lift-definition delete :: ′a ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b) mapping-rbt is
rbt-comp-delete ccomp
⟨proof⟩

lift-definition bulkload :: (′a × ′b) list ⇒ (′a, ′b) mapping-rbt is
rbt-comp-bulkload ccomp
⟨proof⟩

lift-definition map-entry :: ′a ⇒ (′b ⇒ ′b) ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b) mapping-rbt is
rbt-comp-map-entry ccomp
⟨proof⟩

lift-definition map :: (′a ⇒ ′b ⇒ ′c ⇒ ′c ⇒ ′c) ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′c) mapping-rbt
is RBT-Impl.map
⟨proof⟩

lift-definition entries :: (′a, ′b) mapping-rbt ⇒ (′a × ′b) list is RBT-Impl.entries
⟨proof⟩

lift-definition keys :: (′a, ′b) mapping-rbt ⇒ ′a set is set ◦ RBT-Impl.keys ⟨proof⟩

lift-definition fold :: (′a ⇒ ′b ⇒ ′c ⇒ ′c ⇒ ′c) ⇒ (′a, ′b) mapping-rbt ⇒ ′c ⇒ ′c is
RBT-Impl.fold ⟨proof⟩

lift-definition is-empty :: (′a, ′b) mapping-rbt ⇒ bool is case-rbt True (λ- - - - -.
False) ⟨proof⟩

lift-definition filter :: (′a × ′b ⇒ bool) ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b) mapping-rbt
is λP t. rbtreify (List.filter P (RBT-Impl.entries t))
⟨proof⟩

lift-definition join ::
(′a ⇒ ′b ⇒ ′b ⇒ ′b) ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b) mapping-rbt ⇒ (′a, ′b)
mapping-rbt
is rbt-comp-union-with-key ccomp
3.7. MAPPINGS IMPLEMENTED BY RED-BLACK TREES

⟨proof⟩

lift-definition meet ::
  (′a ⇒  ′b ⇒  ′b ⇒  ′b) ⇒  (′a,  ′b) mapping-rbt ⇒  (′a,  ′b)
is rbt-comp-inter-with-key ccomp
⟨proof⟩

lift-definition all :: (′a ⇒  ′b ⇒  bool) ⇒  (′a,  ′b) mapping-rbt ⇒  bool
is RBT-Impl-rbt-all ⟨proof⟩

lift-definition ex :: (′a ⇒  ′b ⇒  bool) ⇒  (′a,  ′b) mapping-rbt ⇒  bool
is RBT-Impl-rbt-ex ⟨proof⟩

lift-definition product ::
  (′a ⇒  ′b ⇒  ′c ⇒  ′d ⇒  ′e) ⇒  (′c :: ccompare,  ′d) mapping-rbt ⇒  (′a ×  ′c,  ′e) mapping-rbt
is rbt-product
⟨proof⟩

lift-definition diag ::
  (′a,  ′b) mapping-rbt ⇒  (′a ×  ′a,  ′b) mapping-rbt
is RBT-Impl-diag
⟨proof⟩

lift-definition init :: (′a,  ′b) mapping-rbt ⇒  (′a,  ′b,  ′c) rbt-generator-state
is rbt-init ⟨proof⟩

end

3.7.3 Properties

lemma unfoldr-rbt-entries-generator:
  list.unfoldr rbt-entries-generator (init t) = entries t
⟨proof⟩

lemma lookup-RBT:
  ord.is-rbt cless t \[⇒\]
  lookup (Mapping-RBT t) = rbt-comp-lookup ccomp t
⟨proof⟩

lemma lookup-impl-of:
  rbt-comp-lookup ccomp (impl-of t) = lookup t
⟨proof⟩

lemma entries-impl-of:
  RBT-Impl.entries (impl-of t) = entries t
⟨proof⟩
**CHAPTER 3. LIGHT-WEIGHT CONTAINERS**

**lemma** `keys-impl-of`:

```plaintext
  set (RBT-Impl.keys (impl-of t)) = keys t
  ⟨proof⟩
```

**lemma** `lookup-empty` [simp]:

```plaintext
  lookup empty = Map.empty
  ⟨proof⟩
```

**lemma** `fold-conv-fold`:

```plaintext
  fold f t = List.fold (case-prod f) (entries t)
  ⟨proof⟩
```

**lemma** `is-empty-empty` [simp]:

```plaintext
  is-empty t ←→ t = empty
  ⟨proof⟩
```

**context** assumes `ID-ccompare-neq-None`:

```plaintext
  ID CCOMPARE (′a :: ccompare) ≠ None
```

**begin**

**lemma** `mapping-linorder`:

```plaintext
  class.linorder (cless-eq :: ′a ⇒ ′a ⇒ bool) cless
  ⟨proof⟩
```

**lemma** `mapping-comparator`:

```plaintext
  comparator (ccomp :: ′a comparator)
  ⟨proof⟩
```

**lemmas** `rbt-comp` [simp] = `rbt-comp-simps` [OF `mapping-comparator`]

**lemma** `is-rbt-impl-of` [simp, intro]:

```plaintext
  fixes t :: ′a, ′b mapping-rbt
  shows ord.is-rbt cless (impl-of t)
  ⟨proof⟩
```

**lemma** `lookup-insert` [simp]:

```plaintext
  lookup (insert (k :: ′a) v t) = (lookup t)(k ↦ v)
  ⟨proof⟩
```

**lemma** `lookup-delete` [simp]:

```plaintext
  lookup (delete (k :: ′a) t) = (lookup t)(k := None)
  ⟨proof⟩
```

**lemma** `map-of-entries` [simp]:

```plaintext
  map-of (entries (t :: ′a, ′b mapping-rbt)) = lookup t
  ⟨proof⟩
```

**lemma** `entries-lookup`:

```plaintext
  entries (t1 :: ′a, ′b mapping-rbt) = entries t2 ←→ lookup t1 = lookup t2
  ⟨proof⟩
```

**lemma** `lookup-bulkload` [simp]:
3.7. MAPPINGS IMPLEMENTED BY RED-BLACK TREES

lookup (bulkload xs) = map-of (xs :: ('a × 'b) list)
⟨proof⟩

lemma lookup-map-entry [simp]:
lookup (map-entry (k :: 'a) f t) = (lookup t)(k := map-option f (lookup t k))
⟨proof⟩

lemma lookup-map [simp]:
lookup (map f t) (k :: 'a) = map-option (f k) (lookup t k)
⟨proof⟩

lemma RBT-lookup-empty [simp]:
ord.rbt-lookup cless (t :: ('a, 'b) RBT-Impl.rbt) = Map.empty ←→ t = RBT-Impl.Empty
⟨proof⟩

lemma lookup-empty-empty [simp]:
lookup t = Map.empty ←→ (t :: ('a, 'b) mapping-rbt) = empty
⟨proof⟩

lemma finite-dom-lookup [simp]: finite (dom (lookup (t :: ('a, 'b) mapping-rbt)))
⟨proof⟩

lemma card-com-lookup [unfolded length-map, simp]:
card (dom (lookup (t :: ('a, 'b) mapping-rbt))) = length (List.map fst (entries t))
⟨proof⟩

lemma lookup-join:
lookup (join f (t1 :: ('a, 'b) mapping-rbt) t2) =
(λk. case lookup t1 k of None ⇒ lookup t2 k | Some v1 ⇒ Some (case lookup t2 k of None ⇒ v1 | Some v2 ⇒ f k v1 v2))
⟨proof⟩

lemma lookup-meet:
lookup (meet f (t1 :: ('a, 'b) mapping-rbt) t2) =
(λk. case lookup t1 k of None ⇒ None | Some v1 ⇒ case lookup t2 k of None ⇒ None | Some v2 ⇒ Some (f k v1 v2))
⟨proof⟩

lemma lookup-filter [simp]:
lookup (filter P (t :: ('a, 'b) mapping-rbt)) k =
(case lookup t k of None ⇒ None | Some v ⇒ if P (k, v) then Some v else None)
⟨proof⟩

lemma all-conv-all-lookup:
all P t ←→ (∀ (k :: 'a) v. lookup t k = Some v → P k v)
⟨proof⟩

lemma ex-conv-ex-lookup:
ex P t \rightarrow (\exists (k :: 'a) v. \text{lookup } t \ k = \text{Some } v \land P \ k \ v)

\langle \text{proof} \rangle

\textbf{lemma diag-lookup:}
\text{lookup} (\text{diag } t) = (\lambda (k :: 'a, k'). \text{if } k = k' \text{ then lookup } t \ k \text{ else None})
\langle \text{proof} \rangle

\textbf{context assumes ID-ccompare-neq-None':} ID \ CCOMPARE('b :: ccompare) \neq None
\begin{description}
\item[begin]
\textbf{lemma mapping-linorder':} class.linorder (cless-eq :: 'b \Rightarrow 'b \Rightarrow bool) cless
\langle \text{proof} \rangle
\textbf{lemma mapping-comparator':} comparator (ccomp :: 'b comparator)
\langle \text{proof} \rangle
\item[lemmas rbt-comp'[simp] = rbt-comp-simps[OF mapping-comparator']]
\textbf{lemma ccomp-comparator-prod':}
ccomp = (\text{comparator-prod ccomp ccomp :: ('a} \times 'b)comparator)
\langle \text{proof} \rangle
\item[lemma lookup-product:]
\text{lookup} (\text{product } f \ rbt1 \ rbt2) (a :: 'a, b :: 'b) =
(case \text{lookup} rbt1 \ a \text{ of None} \Rightarrow \text{None}
| \text{Some } c \Rightarrow \text{map-option } (f a c b) (\text{lookup} rbt2 \ b))
\langle \text{proof} \rangle
\end{description}
\textbf{end}
\textbf{hide-const (open)} impl-of lookup empty insert delete
\text{entries keys bulkload map-entry map fold join meet filter all ex product diag init}
\textbf{end}

\textbf{theory AssocList imports}
\texttt{~/src/HOL/Library/DAList}
\begin{description}
\item[begin]
\item[3.8 Additional operations for associative lists]
\item[3.8.1 Operations on the raw type]
\textbf{primrec update-with-aux :: 'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list\where}
3.8. ADDITIONAL OPERATIONS FOR ASSOCIATIVE LISTS

```
update-with-aux v k f [] = [(k, f v)]
| update-with-aux v k f (p # ps) = (if (fst p = k) then (k, f (snd p)) # ps
| # update-with-aux v k f ps
```

Do not use AList.delete because this traverses all the list even if it has found
the key. We do not have to keep going because we use the invariant that
keys are distinct.

```
fun delete-aux :: 'key ⇒ ('key × 'val) list ⇒ ('key × 'val) list
where
  delete-aux k [] = []
| delete-aux k ((k', v) # xs) = (if k = k' then xs else (k', v) # delete-aux k xs)
```

```
lemma update-conv-update-with-aux:
  AList.update k v xs = update-with-aux v k (λ- v) xs
⟨proof⟩
```

```
lemma map-of-update-with-aux:
  map-of (update-with-aux v k f ps) k' = ((map-of ps)(k ↦ (case map-of ps k of
None ⇒ f v | Some v ⇒ f v))) k'
⟨proof⟩
```

```
lemma map-of-update-with-aux:
  map-of (update-with-aux v k f ps) = (map-of ps)(k ↦ (case map-of ps k of None
⇒ f v | Some v ⇒ f v))
⟨proof⟩
```

```
lemma dom-update-with-aux: fst ' set (update-with-aux v k f ps) = {k} ∪ fst ' set ps
⟨proof⟩
```

```
lemma distinct-update-with-aux [simp]:
  distinct (map fst (update-with-aux v k f ps)) = distinct (map fst ps)
⟨proof⟩
```

```
lemma set-update-with-aux:
  distinct (map fst xs)
⇒ set (update-with-aux v k f xs) = (set xs - {k} × UNIV ∪ {(k, f (case map-of xs k of
None ⇒ v | Some v ⇒ v))})
⟨proof⟩
```

```
lemma set-delete-aux: distinct (map fst xs) ⇒ set (delete-aux k xs) = set xs -
{k} × UNIV
⟨proof⟩
```

```
lemma dom-delete-aux: distinct (map fst ps) ⇒ fst ' set (delete-aux k ps) = fst
' set ps - {k}
⟨proof⟩
```

```
lemma distinct-delete-aux [simp]:
```
distinct \((\text{map } \text{fst } ps)\) \implies \text{distinct } (\text{map } \text{fst } (\text{delete-aux } k \ ps))

\langle \text{proof} \rangle

\textbf{lemma map-of-delete-aux':}
\text{distinct } (\text{map } \text{fst } xs) \implies \text{map-of } (\text{delete-aux } k \ xs) = (\text{map-of } xs)(k := \text{None})

\langle \text{proof} \rangle

\textbf{lemma map-of-delete-aux:}
\text{distinct } (\text{map } \text{fst } xs) \implies \text{map-of } (\text{delete-aux } k \ xs) k' = ((\text{map-of } xs)(k := \text{None}))

\langle \text{proof} \rangle

\textbf{lemma delete-aux-eq-Nil-conv: delete-aux } k \ ts = [] \iff ts = [] \lor (\exists v. \ ts = [(k, v)])

\langle \text{proof} \rangle

\textbf{3.8.2 Operations on the abstract type } (\text{'}a, 'b\text{) \text{alist}}

\textbf{lift-definition update-with :} 'v \Rightarrow 'k \Rightarrow ('v \Rightarrow 'v) \Rightarrow ('k, 'v) \text{\text{alist} } \Rightarrow ('k, 'v) \text{\text{alist}}

\text{is update-with-aux} \langle \text{proof} \rangle

\textbf{lift-definition delete :} 'k \Rightarrow ('k, 'v) \text{\text{alist}} \Rightarrow ('k, 'v) \text{\text{alist}} \text{is delete-aux}

\langle \text{proof} \rangle

\textbf{lift-definition keys :} ('k, 'v) \text{\text{alist} } \Rightarrow 'k \ \text{set} \text{ is set } \circ \ \text{map} \text{ } \text{fst} \langle \text{proof} \rangle

\textbf{lift-definition set :} ('key, 'val) \text{\text{alist}} \Rightarrow ('key \times 'val) \text{set}

\text{is List.set} \langle \text{proof} \rangle

\textbf{lemma lookup-update-with [simp]:}
\text{DAList.lookup } (\text{update-with } v \ k \ f \ \text{al}) = (\text{DAList.lookup al})(k \mapsto \text{case } \text{DAList.lookup al } k \text{ of } \text{None } \Rightarrow f \ v \mid \text{Some } v \Rightarrow f \ v)

\langle \text{proof} \rangle

\textbf{lemma lookup-delete [simp]: DAList.lookup } (\text{delete } k \ \text{al}) = (\text{DAList.lookup al})(k := \text{None})

\langle \text{proof} \rangle

\textbf{lemma finite-dom-lookup [simp, intro!]: finite } (\text{dom } (\text{DAList.lookup } m))

\langle \text{proof} \rangle

\textbf{lemma update-conv-update-with: DAList.update } k \ v = \text{update-with } v \ k \ (\lambda-. v)

\langle \text{proof} \rangle

\textbf{lemma lookup-update [simp]: DAList.lookup } (\text{DAList.update } k \ v \ \text{al}) = (\text{DAList.lookup al})(k \mapsto v)

\langle \text{proof} \rangle

\textbf{lemma dom-lookup-keys: dom } (\text{DAList.lookup al}) = \text{keys } \text{al}
3.8. ADDITIONAL OPERATIONS FOR ASSOCIATIVE LISTS

⟨proof⟩

lemma keys-empty [simp]: keys DList.empty = {}
⟨proof⟩

lemma keys-update-with [simp]: keys (update-with v k f al) = insert k (keys al)
⟨proof⟩

lemma keys-update [simp]: keys (DList.update k v al) = insert k (keys al)
⟨proof⟩

lemma keys-delete [simp]: keys (delete k al) = keys al − {k}
⟨proof⟩

lemma set-empty [simp]: set DList.empty = {}
⟨proof⟩

lemma set-update-with:
  set (update-with v k f al) =
  (set al − {k} × UNIV ∪ {(k, f (case DList.lookup al k of None ⇒ v | Some v ⇒ v)})
⟨proof⟩

lemma set-update: set (DList.update k v al) = (set al − {k} × UNIV ∪ {(k, v)})
⟨proof⟩

lemma set-delete: set (delete k al) = set al − {k} × UNIV
⟨proof⟩

class context begin
interpretation lifting-syntax ⟨proof⟩

lemma size-dalist-transfer [transfer-rule]:
  (pcr-alist op= op= === op =) length size
⟨proof⟩

end

lemma size-eq-card-dom-lookup: size al = card (dom (DList.lookup al))
⟨proof⟩

hide-const (open) update-with keys set delete

end

theory DList-Set imports
3.9 Sets implemented by distinct lists

3.9.1 Operations on the raw type with parametrised equality

collection-eq
equal
begin

primrec list-member :: 'a list ⇒ 'a ⇒ bool
where
  list-member [] y ←→ False
| list-member (x # xs) y ←→ equal x y ∨ list-member xs y

primrec list-distinct :: 'a list ⇒ bool
where
  list-distinct [] ←→ True
| list-distinct (x # xs) ←→ ¬ list-member xs x ∧ list-distinct xs

definition list-insert :: 'a ⇒ 'a list ⇒ 'a list
where
  list-insert x xs = (if list-member xs x then xs else x # xs)

primrec list-remove1 :: 'a ⇒ 'a list ⇒ 'a list
where
  list-remove1 x [] = []
| list-remove1 x (y # xs) = (if equal x y then xs else y # list-remove1 x xs)

primrec list-remdups :: 'a list ⇒ 'a list
where
  list-remdups [] = []
| list-remdups (x # xs) = (if list-member xs x then list-remdups xs else x # list-remdups xs)

lemma list-member-filterD: list-member (filter P xs) x ⇒ list-member xs x
⟨proof⟩

lemma list-distinct-filter [simp]: list-distinct xs ⇒ list-distinct (filter P xs)
⟨proof⟩

lemma list-distinct-tl [simp]: list-distinct xs ⇒ list-distinct (tl xs)
⟨proof⟩
end

lemmas [code] =
equal-base.list-member.simps
equal-base.list-distinct.simps
equal-base.list-insert-def
equal-base.list-remove1.simps
equal-base.list-remdups.simps
3.9. SETS IMPLEMENTED BY DISTINCT LISTS

lemmas [simp] =
  \text{equal-base.list-member.simps}
  \text{equal-base.list-distinct.simps}
  \text{equal-base.list-remove1.simps}
  \text{equal-base.list-remdups.simps}

\text{lemma list-member-core-member [simp]:}
  \text{equal-base.list-member op = = List.member}
  \langle \text{proof} \rangle

\text{lemma list-distinct-core-distinct [simp]:}
  \text{equal-base.list-distinct op = = List.distinct}
  \langle \text{proof} \rangle

\text{lemma list-insert-core-insert [simp]:}
  \text{equal-base.list-insert op = = List.insert}
  \langle \text{proof} \rangle

\text{lemma list-remove1-core-remove1 [simp]:}
  \text{equal-base.list-remove1 op = = List.remove1}
  \langle \text{proof} \rangle

\text{lemma list-remdups-core-remdups [simp]:}
  \text{equal-base.list-remdups op = = List.remdups}
  \langle \text{proof} \rangle

context equal begin

\text{lemma member-insert [simp]: list-member (list-insert x xs) y \longleftrightarrow equal x y \lor list-member xs y}
  \langle \text{proof} \rangle

\text{lemma member-remove1 [simp]:}
  \neg equal x y \implies list-member (list-remove1 x xs) y = list-member xs y
  \langle \text{proof} \rangle

\text{lemma distinct-remove1:}
  list-distinct xs \implies list-distinct (list-remove1 x xs)
  \langle \text{proof} \rangle

\text{lemma distinct-member-remove1 [simp]:}
  list-distinct xs \implies list-member (list-remove1 x xs) = (list-member xs)(x := \text{False})
  \langle \text{proof} \rangle

end

\text{lemma ID-ceq:}
ID CEQ('a :: ceq) = Some eq \Rightarrow equal eq
⟨proof⟩

3.9.2 The type of distinct lists

typedef 'a :: ceq set-dlist =
{xs::'a list. equal-base.list-distinct ceq' xs \lor ID CEQ('a) = None}

morphisms list-of-dlist Abs-dlist'
⟨proof⟩

definition Abs-dlist :: 'a :: ceq list \Rightarrow 'a set-dlist

where
Abs-dlist xs = Abs-dlist'
(if equal-base.list-distinct ceq' xs \lor ID CEQ('a) = None then xs
else equal-base.list-remdups ceq' xs)

lemma Abs-dlist-inverse:
fixes y :: 'a :: ceq list
assumes y \in \{xs. equal-base.list-distinct ceq' xs \lor ID CEQ('a) = None\}
shows list-of-dlist (Abs-dlist y) = y
⟨proof⟩

lemma list-of-dlist-inverse: Abs-dlist (list-of-dlist dxs) = dxs
⟨proof⟩

lemma type-definition-set-dlist' :

  type-definition list-of-dlist Abs-dlist

{xs :: 'a :: ceq list. equal-base.list-distinct ceq' xs \lor ID CEQ('a) = None}
⟨proof⟩

lemmas Abs-dlist-cases[cases type: set-dlist] =
  type-definition.Abs-cases[OF type-definition-set-dlist']

and Abs-dlist-induct[induct type: set-dlist] =
  type-definition.Abs-induct[OF type-definition-set-dlist'] and
Abs-dlist-inject = type-definition.Abs-inject[OF type-definition-set-dlist']

setup-lifting type-definition-set-dlist'

3.9.3 Operations

lift-definition empty :: 'a :: ceq set-dlist is []
⟨proof⟩

lift-definition insert :: 'a :: ceq \Rightarrow 'a set-dlist \Rightarrow 'a set-dlist is
equal-base.list-insert ceq'
⟨proof⟩

lift-definition remove :: 'a :: ceq \Rightarrow 'a set-dlist \Rightarrow 'a set-dlist is
equal-base.list-remove1 ceq'
⟨proof⟩
3.9. SETS IMPLEMENTED BY DISTINCT LISTS

lift-definition filter :: (′a :: ceq ⇒ bool) ⇒ ′a set-dlist ⇒ ′a set-dlist is List.filter
⟨proof⟩

Derived operations:

lift-definition null :: ′a :: ceq set-dlist ⇒ bool is List.null (proof)

lift-definition member :: ′a :: ceq set-dlist ⇒ ′a ⇒ bool is equal-base.list-member ceq
⟨proof⟩

lift-definition length :: ′a :: ceq set-dlist ⇒ nat is List.length (proof)

lift-definition fold :: (′a :: ceq ⇒ ′b ⇒ ′b) ⇒ ′a set-dlist ⇒ ′b ⇒ ′b is List.fold
⟨proof⟩

lift-definition foldr :: (′a :: ceq ⇒ ′b ⇒ ′b) ⇒ ′a set-dlist ⇒ ′b ⇒ ′b is List.foldr
⟨proof⟩

lift-definition hd :: ′a :: ceq set-dlist ⇒ ′a is List.hd (proof)

lift-definition tl :: ′a :: ceq set-dlist ⇒ ′a set-dlist is List.tl
⟨proof⟩

lift-definition dlist-all :: (′a ⇒ bool) ⇒ ′a :: ceq set-dlist ⇒ bool is list-all (proof)

lift-definition dlist-ex :: (′a ⇒ bool) ⇒ ′a :: ceq set-dlist ⇒ bool is list-ex (proof)

definition union :: ′a :: ceq set-dlist ⇒ ′a set-dlist ⇒ ′a set-dlist
where
  union = fold insert

lift-definition product :: ′a :: ceq set-dlist ⇒ ′b :: ceq set-dlist ⇒ (′a × ′b) set-dlist
  is λxs ys. rev (concat (map (λx. map (Pair x) ys) xs))
⟨proof⟩

lift-definition Id-on :: ′a :: ceq set-dlist ⇒ (′a × ′a) set-dlist
  is map (λx. (x, x))
⟨proof⟩

3.9.4 Properties

lemma member-empty [simp]: member empty = (λ-. False)
⟨proof⟩

lemma null-iff [simp]: null xs ↔ xs = empty
⟨proof⟩

lemma list-of-dlist-empty [simp]: list-of-dlist DList-Set.empty = []
⟨proof⟩
lemma list-of-dlist-insert [simp]: \(\neg \text{member } dxs \ x \implies \text{list-of-dlist (insert } x \ dxs) = x \# \text{list-of-dlist } dxs\)  
(proof)

lemma list-of-dlist-eq-Nil-iff [simp]: \(\text{list-of-dlist } dxs = [] \iff dxs = \text{empty}\)  
(proof)

lemma fold-empty [simp]: \(\text{DList-Set.fold } f \text{ empty } b = b\)  
(proof)

lemma fold-insert [simp]: \(\neg \text{member } dxs \ x \implies \text{DList-Set.fold } f (\text{insert } x \ dxs) \ b = \text{DList-Set.fold } f \ dxs \ (f \ x \ b)\)  
(proof)

lemma no-memb-fold-insert: \(\neg \text{member } dxs \ x \implies \text{fold } f (\text{insert } x \ dxs) \ b = \text{fold } f \ dxs \ (f \ x \ b)\)  
(proof)

lemma set-fold-insert: \(\text{set } (\text{List.fold } \text{List.insert } xs1 \ xs2) = \text{set } xs1 \cup \text{set } xs2\)  
(proof)

lemma list-of-dlist-eq-singleton-conv: \(\text{list-of-dlist } dxs = [x] \iff dxs = \text{DList-Set.insert } x \text{DList-Set.empty}\)  
(proof)

lemma product-code [code abstract]: \(\text{list-of-dlist } (\text{product } dxs1 \ dxs2) = \text{fold } (\lambda a. \text{fold } (\lambda c \ \text{rest}. \ (a, c) \# \rest) \ dxs2) \ dxs1 \ []\)  
(proof)

lemma set-list-of-dlist-Abs-dlist: \(\text{set } (\text{list-of-dlist } (\text{Abs-dlist } xs)) = \text{set } xs\)  
(proof)

context assumes ID-ceq-neq-None: \(\text{ID CEQ('a :: ceq) \neq None}\)  
begin

lemma equal-ceq: \(\text{equal } (\text{ceq' :: 'a \Rightarrow 'a \Rightarrow bool})\)  
(proof)

declare Domainp-forall-transfer[where A = pcr-set-dlist op=, simplified set-dlist.domain-eq, transfer-rule]

lemma set-dlist-induct [case-names Nil insert, induct type: set-dlist]:  
fixes dxs :: 'a :: ceq set-dlist  
assumes Nil: \(P \text{ empty and Cons: } \forall a \ dxs. \ [\neg \text{member } dxs \ a; \ P \ dxs] \implies P\)  
(insert a dxs)  
shows P dxs
3.9. SETS IMPLEMENTED BY DISTINCT LISTS

⟨proof⟩

context
begin
interpretation lifting-syntax ⟨proof⟩

lemma fold-transfer2 [transfer-rule]:
    assumes is-equality A
    shows \((A \Longrightarrow pcr-set-dlist op = \Longrightarrow pcr-set-dlist op =) \Longrightarrow\)
        \((pcr-set-dlist op = :: 'a list ⇒ 'a set-dlist ⇒ bool) \Longrightarrow pcr-set-dlist op =\)\)
        \(\Longrightarrow List.fold DList-Set.fold\)
⟨proof⟩

end

lemma distinct-list-of-dlist:
    distinct (list-of-dlist (dxs :: 'a set-dlist))
⟨proof⟩

lemma member-empty-empty: \((∀ x :: 'a. ¬ member dxs x) \longleftrightarrow dxs = empty\)
⟨proof⟩

lemma Collect-member: Collect (member (dxs :: 'a set-dlist)) = set (list-of-dlist dxs)
⟨proof⟩

lemma member-insert: member (insert (x :: 'a) xs) = (member xs)(x := True)
⟨proof⟩

lemma member-remove:
    member (remove (x :: 'a) xs) = (member xs)(x := False)
⟨proof⟩

lemma member-union: member (union (xs1 :: 'a set-dlist) xs2) x \longleftrightarrow member
    xs1 x ∨ member xs2 x
⟨proof⟩

lemma member-fold-insert: member (List.fold insert xs dxs) (x :: 'a) \longleftrightarrow member
    dxs x ∨ x \in set xs
⟨proof⟩

lemma card-eq-length [simp]:
    card (Collect (member (dxs :: 'a set-dlist))) = length dxs
⟨proof⟩

lemma finite-member [simp]:
    finite (Collect (member (dxs :: 'a set-dlist)))
⟨proof⟩
lemma member-filter [simp]: \text{member} \ (\text{filter} \ P \ xs) = (\lambda x :: 'a. \text{member} \ xs \ x \land P \ x) \\
\langle \text{proof} \rangle

lemma dlist-all-conv-member: \text{dlist-all} \ P \ dxs \iff (\forall x :: 'a. \text{member} \ dxs \ x \rightarrow P \ x) \\
\langle \text{proof} \rangle

lemma dlist-ex-conv-member: \text{dlist-ex} \ P \ dxs \iff (\exists x :: 'a. \text{member} \ dxs \ x \land P x) \\
\langle \text{proof} \rangle

lemma member-Id-on: \text{member} \ (\text{Id-on} \ dxs) = (\lambda x :: 'a. y. x = y \land \text{member} \ dxs \ x) \\
\langle \text{proof} \rangle
end

lemma product-member: \\
\text{assumes} \ ID \ CEQ (\cdot \ a :: ceq) \neq \text{None} \quad ID \ CEQ (\cdot \ b :: ceq) \neq \text{None} \\
\text{shows} \ \text{member} \ (\text{product} \ dxs1 \ dxs2) = (\lambda(a :: 'a, b :: 'b). \text{member} \ dxs1 \ a \land \text{member} \ dxs2 \ b) \\
\langle \text{proof} \rangle

hide-const (open) \text{empty} \ \text{insert} \ \text{remove} \ \text{null} \ \text{member} \ \text{length} \ \text{fold} \ \text{foldr} \ \text{union} \ \text{filter} \ \text{hd} \ \text{tl} \ \text{dlist-all} \ \text{product} \ \text{Id-on}
end

theory RBT-Set2 
imports RBT-Mapping2 
begin

3.10 Sets implemented by red-black trees

lemma map-of-map-Pair-const: \\
map-of \ (\map (\lambda x. (x, v)) \ xs) = (\lambda x. \text{if} \ x \in \text{set} \ xs \ \text{then} \text{Some} \ v \ \text{else} \text{None}) \\
\langle \text{proof} \rangle

lemma map-of-rev-unit [simp]: \\
fixes xs :: ('a * unit) list \\
shows map-of \ (rev \ xs) = map-of \ xs \\
\langle \text{proof} \rangle

lemma fold-split-conv-map-fst: fold \ (\lambda(x, y). f x) \ xs = fold f \ (\map \text{fst} \ xs) \\
\langle \text{proof} \rangle
3.10. SETS IMPLEMENTED BY RED-BLACK TREES

**lemma** foldr-split-conv-map-fst: \( \text{foldr} \ (\lambda(x, y). f x) \ xs = \text{foldr} f \ (\text{map} \ \text{fst} \ xs) \)

**lemma** set-foldr-Cons:
\[ \text{set} \ (\text{foldr} \ (\lambda x \ xs. \text{if} \ P \ x \ xs \ \text{then} \ x \ # \ xs \ \text{else} \ xs) \ \text{as} \ []) \subseteq \text{set as} \]

**lemma** distinct-fst-foldr-Cons:
\[ \text{distinct} \ (\text{map} \ f \ \text{as}) \implies \text{distinct} \ (\text{foldr} \ (\lambda x \ xs. \text{if} \ P \ x \ xs \ \text{then} \ x \ # \ xs \ \text{else} \ xs) \ \text{as} \ []) \]

**lemma** filter-conv-foldr:
\[ \text{filter} \ P \ xs = \text{foldr} \ (\lambda x \ xs. \text{if} \ P \ x \ \text{then} \ x \ # \ xs \ \text{else} \ xs) \ [] \]

**lemma** map-of-filter:
\[ \text{map-of} \ (\text{filter} \ (\lambda x. P (\text{fst} x)) \ xs) = \text{map-of} \ xs \ \mid \ \text{'Collect} \ P \]

**lemma** neq-Empty-conv:
\[ t \neq \text{rbt.Empty} \iff (\exists \ c \ l \ k \ v \ r. \ t = \text{Branch} \ c \ l \ k \ v \ r) \]

**context** linorder begin

**lemma** is-rbt-RBT-fold-rbt-insert [simp]:
\[ \text{is-rbt} \ t \implies \text{is-rbt} \ (\text{fold} \ (\lambda(k, v). rbt-insert \ k \ v) \ xs \ t) \]

**lemma** rbt-lookup-RBT-fold-rbt-insert [simp]:
\[ \text{is-rbt} \ t \implies \text{rbt-lookup} \ (\text{fold} \ (\lambda(k, v). rbt-insert \ k \ v) \ xs \ t) = \text{rbt-lookup} \ t \ \mid \ \text{map-of} \ (\text{rev} \ xs) \]

**lemma** is-rbt-fold-rbt-delete [simp]:
\[ \text{is-rbt} \ t \implies \text{is-rbt} \ (\text{fold} \ rbt-delete \ xs \ t) \]

**lemma** rbt-lookup-fold-rbt-delete [simp]:
\[ \text{is-rbt} \ t \implies \text{rbt-lookup} \ (\text{fold} \ rbt-delete \ xs \ t) = \text{rbt-lookup} \ t \ \mid \ \text{map-of} \ (\text{rev} \ xs) \]

**lemma** is-rbt-fold-rbt-insert: \[ \text{is-rbt} \ t \implies \text{is-rbt} \ (\text{fold} \ (\lambda k. rbt-insert \ k \ (f k)) \ xs \ t) \]

**lemma** rbt-lookup-fold-rbt-insert:
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

\[ is\text{-}rbt\ t \implies \]
\[ \text{rbt}\text{-lookup}\ (\text{fold}\ (\lambda k.\ \text{rbt}\text{-insert}\ k\ (f\ k))\ xs\ t) = \]
\[ \text{rbt}\text{-lookup}\ t\ +\ map\text{-}of\ (\text{map}\ (\lambda k.\ (k,\ f\ k))\ xs) \]
\langle proof\rangle

end

definition fold-rev :: (\text{a} \Rightarrow \text{b} \Rightarrow \text{c} \Rightarrow \text{d}) \Rightarrow (\text{a},\ \text{b})\ \text{rbt} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} where
\[ \text{fold}\text{-rev}\ f\ t = \text{List}\\text{-foldr}\ (\lambda (k,\ v).\ f\ k\ v)\ (\text{RBT}\text{-Impl}\.\text{entries}\ t) \]

lemma fold-rev-simps [simp, code]:
\[ \text{fold}\text{-rev}\ f\ \text{RBT}\text{-Impl}\.\text{Empty} = \text{id} \]
\[ \text{fold}\text{-rev}\ f\ (\text{Branch}\ c\ l\ k\ v\ r) = \text{fold}\text{-rev}\ f\ l\ \circ\ f\ k\ v\ \circ\ \text{fold}\text{-rev}\ f\ r \]
\langle proof\rangle

definition (in ord) rbt\text{-minus} :: (\text{a},\ \text{b})\ \text{rbt} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} where
\[ \text{rbt}\text{-minus}\ t1\ t2 = \]
\[ \text{(let}\ h1 = \text{bheight}\ t1;\ h2 = \text{bheight}\ t2 \]
\[ \text{in}\ \text{if} 2 * h1 \leq h2\ \text{then}\ \text{rbtreeify}\ (\text{fold}\text{-rev}\ (\lambda k\ v\ kvs.\ \text{if}\ \text{rbt}\text{-lookup}\ t2\ k = \text{None}\ \text{then}\ (k,\ v)\ \#\ kvs\ \text{else}\ kvs)\ t1\ []) \]
\[ \text{else}\ \text{RBT}\text{-Impl}\.\text{fold}\ (\lambda k\ v.\ \text{rbt}\text{-delete}\ k)\ t2\ t1) \]

definition rbt\text{-comp}\text{-minus} :: \text{a}\ \text{comparator} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} \Rightarrow (\text{a},\ \text{b})\ \text{rbt} where
\[ \text{rbt}\text{-comp}\text{-minus}\ c\ t1\ t2 = \]
\[ \text{(let}\ h1 = \text{bheight}\ t1;\ h2 = \text{bheight}\ t2 \]
\[ \text{in}\ \text{if} 2 * h1 \leq h2\ \text{then}\ \text{rbtreeify}\ (\text{fold}\text{-rev}\ (\lambda k\ v\ kvs.\ \text{if}\ \text{rbt}\text{-comp}\text{-lookup}\ c\ t2\ k = \text{None}\ \text{then}\ (k,\ v)\ \#\ kvs\ \text{else}\ kvs)\ t1\ []) \]
\[ \text{else}\ \text{RBT}\text{-Impl}\.\text{fold}\ (\lambda k\ v.\ \text{rbt}\text{-comp}\text{-delete}\ c\ k)\ t2\ t1) \]

lemma rbt\text{-comp}\text{-minus}: assumes c: \text{comparator}\ c
shows rbt\text{-comp}\text{-minus}\ c = \text{ord}\text{-rbt}\text{-minus}\ (\text{lt}\text{-of}\text{-comp}\ c) \]
\langle proof\rangle

context linorder begin

lemma sorted-fst-foldr-Cons:
\[ \text{sorted}\ (\text{map}\ f\ \text{as}) \implies \text{sorted}\ (\text{map}\ f\ (\text{foldr}\ (\lambda x\ xs.\ \text{if}\ P\ x\ xs\ \text{then} x\ \#\ xs\ \text{else}\ xs)\ \text{as}\ []))) \]
\langle proof\rangle

lemma is-rbt-rbt\text{-minus}:
\[ [\text{is-rbt}\ t1;\ \text{is-rbt}\ t2] \implies \text{is-rbt}\ (\text{rbt}\text{-minus}\ t1\ t2) \]
\langle proof\rangle

lemma rbt\text{-lookup}\text{-rbt}\text{-minus}:
\[ [\text{is-rbt}\ t1;\ \text{is-rbt}\ t2] \]
3.10. SETS IMPLEMENTED BY RED-BLACK TREES

⇒ rbt-lookup (rbt-minus t1 t2) = rbt-lookup t1 \setminus (\text{dom} \ (rbt-lookup t2))

end

3.10.1 Type and operations

type-synonym \(\text{'a set-rbt} = (\text{'a, unit})\)
mapping-rbt

translations

\((\text{type}) \ 'a \text{ set-rbt} <\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\![\text{proof}]

3.10.2 Primitive operations

lift-definition member :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{'a} \Rightarrow \text{bool}\) is
\(\lambda x. x \in \text{dom} \ (rbt-comp-lookup ccomp t)\) (\text{proof})

abbreviation empty :: \(\text{'a :: ccompare set-rbt}\)
where empty \(\equiv\) RBT-Mapping2.empty

abbreviation insert :: \(\text{'a :: ccompare} \Rightarrow \text{'a set-rbt} \Rightarrow \text{'a set-rbt}\)
where insert k \(\equiv\) RBT-Mapping2.insert k ()

abbreviation remove :: \(\text{'a :: ccompare} \Rightarrow \text{'a set-rbt} \Rightarrow \text{'a set-rbt}\)
where remove \(\equiv\) RBT-Mapping2.delete

lift-definition bulkload :: \(\text{'a :: ccompare list} \Rightarrow \text{'a set-rbt}\)
\(\Rightarrow\) \(\text{'a set-rbt}\) is
\(\text{rbt-comp-bulkload ccomp} \circ \text{map}(\lambda x. (x, ()))\) (\text{proof})

abbreviation is-empty :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{bool}\)
where is-empty \(\equiv\) RBT-Mapping2.is-empty

abbreviation union :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{'a set-rbt} \Rightarrow \text{'a set-rbt}\)
where union \(\equiv\) RBT-Mapping2.join (\lambda - -. id)

abbreviation inter :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{'a set-rbt} \Rightarrow \text{'a set-rbt}\)
where inter \(\equiv\) RBT-Mapping2.meet (\lambda - -. id)

lift-definition inter-list :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{'a list} \Rightarrow \text{'a set-rbt}\)
\(\Rightarrow\) \(\text{'a set-rbt}\) is
\(\text{At } xs. \text{fold}(\lambda k. \text{rbt-comp-insert ccomp k ()}) \ [x \leftarrow xs. \text{rbt-comp-lookup ccomp t x } \neq \text{None}]\) RBT-Impl.Empty (\text{proof})

lift-definition minus :: \(\text{'a :: ccompare set-rbt} \Rightarrow \text{'a set-rbt} \Rightarrow \text{'a set-rbt}\)
\(\Rightarrow\) \(\text{'a set-rbt}\) is
\(\text{rbt-comp-minus ccomp}\) (\text{proof})
abbreviation filter :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ 'a set-rbt
where filter P ≡ RBT-Mapping2.filter (P ∘ fst)

lift-definition fold :: ('a :: ccompare ⇒ 'b ⇒ 'b) ⇒ 'a set-rbt ⇒ 'b ⇒ 'b is λf.
RBT-Impl.fold (λa - f a) ⟨proof⟩

lift-definition fold1 :: ('a :: ccompare ⇒ 'a ⇒ 'a) ⇒ 'a set-rbt ⇒ 'a is RBT-Impl-fold1
⟨proof⟩

lift-definition keys :: 'a :: ccompare set-rbt ⇒ 'a list is RBT-Impl.keys ⟨proof⟩

abbreviation all :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ bool
where all P ≡ RBT-Mapping2.all (λk - P k)

abbreviation ex :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ bool
where ex P ≡ RBT-Mapping2.ex (λk - P k)

definition product :: 'a :: ccompare set-rbt ⇒ 'b :: ccompare set-rbt ⇒ ('a × 'b) set-rbt
where product rbt1 rbt2 = RBT-Mapping2.product (λ- - . ()) rbt1 rbt2

abbreviation Id-on :: 'a :: ccompare set-rbt ⇒ ('a × 'a) set-rbt
where Id-on ≡ RBT-Mapping2.diag

abbreviation init :: 'a :: ccompare set-rbt ⇒ ('a, unit, 'a) rbt-generator-state
where init ≡ RBT-Mapping2.init

3.10.3 Properties

lemma member-empty [simp]:
  member empty = (λ_. False)
  ⟨proof⟩

lemma fold-conv-fold-keys: RBT-Set2.fold f rbt b = List.fold f (RBT-Set2.keys rbt) b
  ⟨proof⟩

lemma fold-conv-fold-keys':
  fold f t = List.fold f (RBT-Impl.keys (RBT-Mapping2.impl-of t))
  ⟨proof⟩

lemma member-lookup [code]: member t x ←→ RBT-Mapping2.lookup t x = Some ()
  ⟨proof⟩

lemma unfoldr-rbt-keys-generator:
  list.unfoldr rbt-keys-generator (init t) = keys t
  ⟨proof⟩
3.10. SETS IMPLEMENTED BY RED-BLACK TREES

lemma keys-eq-nil-iff [simp]: keys rbt = [] ↔ rbt = empty
⟨proof⟩

lemma fold1-conv-fold: fold1 f rbt = List.fold f (tl (keys rbt)) (hd (keys rbt))
⟨proof⟩

context assumes ID-ccompare-neq-None: ID CCOMPARE ('a :: ccompare) ≠ None
begin
lemma set-linorder: class.linorder (cless-eq :: 'a ⇒ 'a ⇒ bool) cless
⟨proof⟩

lemma ccomp-comparator: comparator (ccomp :: 'a comparator)
⟨proof⟩

lemmas rbt-comps = rbt-comp-simps[OF ccomp-comparator] rbt-comp-minus[OF ccomp-comparator]

lemma is-rbt-impl-of [simp, intro]:
fixes t :: 'a set-rbt
shows ord.is-rbt cless (RBT-Mapping2.impl-of t)
⟨proof⟩

lemma member-RBT:
ord.is-rbt cless t ⟹ member (Set-RBT t) (x :: 'a) ⟷ ord.rbt-lookup cless t x = Some ()
⟨proof⟩

lemma member-impl-of:
ord.rbt-lookup cless (RBT-Mapping2.impl-of t) (x :: 'a) = Some () ⟷ member t x
⟨proof⟩

lemma member-insert [simp]:
member (insert x (t :: 'a set-rbt)) = (member t)(x := True)
⟨proof⟩

lemma member-fold-insert [simp]:
member (List.fold insert xs (t :: 'a set-rbt)) = (λx. member t x ∨ x ∈ set xs)
⟨proof⟩

lemma member-remove [simp]:
member (remove (x :: 'a) t) = (member t)(x := False)
⟨proof⟩

lemma member-bulkload [simp]:
member (bulkload xs) (x :: 'a) ⟷ x ∈ set xs
⟨proof⟩
\textbf{lemma} member-cone-keys: member \( t = (\lambda x :: 'a. x \in \text{set} (\text{keys} t)) \)
\langle proof \rangle
\textbf{lemma} is-empty-empty [simp]:
\begin{align*}
\text{is-empty} \ t & \iff t = \text{empty} \\
\langle proof \rangle
\end{align*}
\textbf{lemma} RBT-lookup-empty [simp]:
\begin{align*}
\text{ord.rbt-lookup} \ \text{clesss} \ (t :: ('a, \text{unit}) \ rbt) & = \text{Map.empty} \iff t = \text{RBT-Impl.Empty} \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-empty-empty [simp]:
\begin{align*}
\text{member} \ t & = (\lambda -. \text{False}) \iff (t :: 'a \text{set-rbt}) = \text{empty} \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-union [simp]:
\begin{align*}
\text{member} \ (\text{union} \ (t1 :: 'a \text{set-rbt}) \ t2) & = (\lambda x. \text{member} \ t1 \ x \lor \text{member} \ t2 \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-minus [simp]:
\begin{align*}
\text{member} \ (\text{minus} \ (t1 :: 'a \text{set-rbt}) \ t2) & = (\lambda x. \text{member} \ t1 \ x \land \neg \text{member} \ t2 \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-inter [simp]:
\begin{align*}
\text{member} \ (\text{inter} \ (t1 :: 'a \text{set-rbt}) \ t2) & = (\lambda x. \text{member} \ t1 \ x \land \text{member} \ t2 \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-inter-list [simp]:
\begin{align*}
\text{member} \ (\text{inter-list} \ (t :: 'a \text{set-rbt}) \ xs) & = (\lambda x. \text{member} \ t \ x \land x \in \text{set} \ xs) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} member-filter [simp]:
\begin{align*}
\text{member} \ (\text{filter} \ P \ (t :: 'a \text{set-rbt})) & = (\lambda x. \text{member} \ t \ x \land P \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} distinct-keys [simp]:
\begin{align*}
\text{distinct} \ (\text{keys} \ (\text{rbt} :: 'a \text{set-rbt})) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} all-conv-all-member:
\begin{align*}
\text{all} \ P \ t & \iff (\forall x :: 'a. \text{member} \ t \ x \longrightarrow P \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} ex-conv-ex-member:
\begin{align*}
\text{ex} \ P \ t & \iff (\exists x :: 'a. \text{member} \ t \ x \land P \ x) \\
\langle proof \rangle
\end{align*}
\textbf{lemma} finite-member: finite (Collect (RBT-Set2.member \ (t :: 'a \text{set-rbt})))
3.10. SETS IMPLEMENTED BY RED-BLACK TREES

lemma member-Id-on: member (Id-on t) = (\lambda (k :: 'a, k'). k = k' \land member t k)

context assumes ID-ccompare-neq-None': ID CCOMPARE('b :: ccompare) \neq None
begin

lemma set-linorder': class.linorder (cless-eq :: 'b => 'b => bool) cless

lemma member-product:
  member (product rbt1 rbt2) = (\lambda ab :: 'a x 'b. ab \in Collect (member rbt1) \times Collect (member rbt2))

end

end

lemma sorted-RBT-Set-keys:
  ID CCOMPARE('a :: ccompare) = Some c
  \implies linorder.sorted (le-of-comp c) (RBT-Set2.keys rbt)

context assumes ID-ccompare-neq-None: ID CCOMPARE('a :: {ccompare, lattice}) \neq None
begin

lemma set-linorder2: class.linorder (cless-eq :: 'a => 'a => bool) cless

end

lemma set-keys-Mapping-RBT: set (keys (Mapping-RBT t)) = set (RBT-Impl.keys t)

hide-const (open) member empty insert remove bulkload union keys fold fold-rev filter all ex product Id-on init

end

theory Closure-Set imports Equal begin
3.11 Sets implemented as Closures

class equal-base begin

definition fun-upd :: (′a ⇒ ′b) ⇒ ′a ⇒ ′b ⇒ ′a ⇒ ′b
where fun-upd-apply: fun-upd f a b a′ = (if equal a a′ then b else f a′)
end

lemmas [code] = equal-base.fun-upd-apply
lemmas [simp] = equal-base.fun-upd-apply

lemma fun-upd-conv-fun-upd: equal-base.fun-upd op = = fun-upd
⟨proof⟩
end

theory Set-Impl imports
Collection-Enum
DList-Set
RBT-Set2
Closure-Set
Containers-Generator
begin

3.12 Different implementations of sets

3.12.1 Auxiliary functions

A simple quicksort implementation

class ord begin

function (sequential) quicksort-acc :: ′a list ⇒ ′a list ⇒ ′a list ⇒ ′a list
and quicksort-part :: ′a list ⇒ ′a ⇒ ′a list ⇒ ′a list ⇒ ′a list ⇒ ′a list ⇒ ′a list ⇒ ′a list
where
  quicksort-acc ac [] = ac
  | quicksort-acc ac [x] = x # ac
  | quicksort-acc ac (x # xs) = quicksort-part ac x [] [] [] xs
  | quicksort-part ac x lts eqs gts [] = quicksort-acc (eqs @ x # quicksort-acc ac gts)
  | lts
  | quicksort-part ac x lts eqs gts (z # zs) =
    (if z > x then quicksort-part ac x lts eqs (z # gts) zs
     else if z < x then quicksort-part ac x (z # lts) eqs gts zs
     else quicksort-part ac x lts (z # eqs) gts zs)
⟨proof⟩

lemma length-quicksort-accp:
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

quicksort-acc-quicksort-part-dom \((\text{Inl } (ac, xs))\) \(\implies\) length \((\text{quicksort-acc } ac \text{ } xs)\)
= length \(ac\) + length \(xs\)
and length-quicksort-partp:
quicksort-acc-quicksort-part-dom \((\text{Inr } (ac, x, lts, eqs, gts, zs))\) 
\(\implies\) length \((\text{quicksort-part } ac \text{ } x \text{ } lts \text{ } eqs \text{ } gts \text{ } zs)\) = length \(ac\) + 1 + length \(lts\) + length \(eqs\) + length \(gts\) + length \(zs\)
⟨proof⟩

termination
⟨proof⟩
definition quicksort :: 'a list \(\implies\) 'a list
where quicksort = quicksort-acc []
lemma set-quicksort-acc [simp]: set \((\text{quicksort-acc } ac \text{ } xs)\) = set \(ac\) \(\cup\) set \(xs\)
and set-quicksort-part [simp]:
set \((\text{quicksort-part } ac \text{ } x \text{ } lts \text{ } eqs \text{ } gts \text{ } zs)\) =
set \(ac\) \(\cup\) \{(x}\) \(\cup\) set \(lts\) \(\cup\) set \(eqs\) \(\cup\) set \(gts\) \(\cup\) set \(zs\)
⟨proof⟩
lemma set-quicksort [simp]: set \((\text{quicksort } xs)\) = set \(xs\)
⟨proof⟩
lemma distinct-quicksort-acc:
distinct \((\text{quicksort-acc } ac \text{ } xs)\) = distinct \((ac \text{ } @ \text{ } xs)\)
and distinct-quicksort-part:
distinct \((\text{quicksort-part } ac \text{ } x \text{ } lts \text{ } eqs \text{ } gts \text{ } zs)\) = distinct \((ac \text{ } @ \text{ } x\text{ } @ \text{ } lts \text{ } @ \text{ } eqs \text{ } @ \text{ } gts \text{ } @ \text{ } zs)\)
⟨proof⟩
lemma distinct-quicksort [simp]: distinct \((\text{quicksort } xs)\) = distinct \(xs\)
⟨proof⟩
end
lemmas [code] =
ord.quicksort-acc.simps quicksort-acc.simps
ord.quicksort-part.simps quicksort-part.simps
ord.quicksort-def quicksort-def
context linorder begin
lemma sorted-quicksort-acc:
\[ \text{sorted } ac; \forall x \in \text{ set } xs, \forall a \in \text{ set } ac. \ x < a \] 
\(\implies\) sorted \((\text{quicksort-acc } ac \text{ } xs)\)
and sorted-quicksort-part:
\[ \text{sorted } ac; \forall y \in \text{ set } lts \cup \{ x\} \cup \text{ set } eqs \cup \text{ set } gts \cup \text{ set } zs, \forall a \in \text{ set } ac. \ y < a; \] 
\(\forall y \in \text{ set } lts. \ y < x; \forall y \in \text{ set } eqs. \ y = x; \forall y \in \text{ set } gts. \ y > x \] 
\(\implies\) sorted \((\text{quicksort-part } ac \text{ } x \text{ } lts \text{ } eqs \text{ } gts \text{ } zs)\)
\textbf{CHAPTER 3. LIGHT-WEIGHT CONTAINERS}

\begin{quote}
\begin{proof}
\end{proof}
\end{quote}

\textbf{lemma \texttt{sorted-quicksort}} [simp]: \texttt{sorted (quicksort xs)}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{insort-key-append1}}:
\begin{proof}
\end{proof}

\textbf{lemma \texttt{insort-key-append2}}:
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-key-append}}:
\begin{proof}
\end{proof}

\textbf{definition \texttt{single-list} :: 'a ⇒ 'a list} where \texttt{single-list a = [a]}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{to-single-list}}: \texttt{x ≠ xs = single-list x @ xs}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-snoc}}: \texttt{sort (xs @ [x]) = insort x (sort xs)}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-append-swap}}: \texttt{sort (xs @ ys) = sort (ys @ xs)}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-append-swap2}}: \texttt{sort (xs @ ys @ zs) = sort (ys @ xs @ zs)}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-Cons-append-swap}}: \texttt{sort (x # xs) = sort (xs @ [x])}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{sort-append-Cons-swap}}: \texttt{sort (ys @ x # zs) = sort (ys @ xs @ [x])}
\begin{proof}
\end{proof}

\textbf{lemma \texttt{quicksort-acc-conv-sort}}:
\begin{proof}
\end{proof}

\textbf{and \texttt{quicksort-part-conv-sort}}:
\begin{proof}
\end{proof}

\textbf{lemma \texttt{quicksort-conv-sort}}: \texttt{quicksort xs = sort xs}
\begin{proof}
\end{proof}
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

lemma sort-remdups: sort (remdups xs) = remdups (sort xs)
⟨proof⟩
end

Removing duplicates from a sorted list

context ord begin
fun remdups-sorted :: 'a list ⇒ 'a list
where
  remdups-sorted [] = []
| remdups-sorted [x] = [x]
| remdups-sorted (x # y # xs) = (if x < y then x # remdups-sorted (y # xs) else remdups-sorted (y # xs))
end

lemmas [code] = ord.remdups-sorted.simps

context linorder begin
lemma [simp]:
  assumes sorted xs
  shows sorted-remdups-sorted: sorted (remdups-sorted xs)
  and set-remdups-sorted: set (remdups-sorted xs) = set xs
⟨proof⟩
lemma distinct-remdups-sorted [simp]: sorted xs =⇒ distinct (remdups-sorted xs)
⟨proof⟩
lemma remdups-sorted-conv-remdups: sorted xs =⇒ remdups-sorted xs = remdups xs
⟨proof⟩
end

An specialised operation to convert a finite set into a sorted list

definition esorted-list-of-set :: 'a :: ccompare set ⇒ 'a list
where [code del]:
esorted-list-of-set A =
  (if ID CCOMPARE (⊤ :: ccompare) = None ∨ ¬ finite A then undefined else linorder.sorted-list-of-set cless-eq A)

lemma esorted-list-of-set-set [simp]:
  [ ID CCOMPARE (⊤ :: ccompare) = Some c; linorder.sorted (le-of-comp c) xs; distinct xs ]
  =⇒ linorder.sorted-list-of-set (le-of-comp c) (set xs) = xs
⟨proof⟩
lemma csorted-list-of-set-split:
fixes A :: 'a :: ccompare set
shows \( P \langle \text{csorted-list-of-set } A \rangle \leftrightarrow \langle \forall \text{xs. ID CCOMPARE('a) \neq None \rightarrow finite } A \rightarrow A = \text{set } \text{xs } \rightarrow \text{distinct } \text{xs } \rightarrow \text{linorder.sorted } \text{cless-eq } \text{xs } \rightarrow \text{P } \text{xs } \rangle \land \langle \text{ID CCOMPARE('a) = None } \lor \neg \text{finite } A \rightarrow \text{P undefined} \rangle \rangle

\langle \text{proof} \rangle

code-identifier code-module Set \rightarrow (SML) Set-Impl
| code-module Set-Impl \rightarrow (SML) Set-Impl

3.12.2 Delete code equation with set as constructor

lemma is-empty-unfold [code-unfold]:
set-eq A { } = Set.is-empty A
set-eq { } A = Set.is-empty A
\langle \text{proof} \rangle

definition is-UNIV :: 'a set \Rightarrow bool
where [code del]: is-UNIV A \leftarrow A = UNIV

lemma is-UNIV-unfold [code-unfold]:
A = UNIV \longleftrightarrow is-UNIV A
UNIV = A \longleftrightarrow is-UNIV A
set-eq A UNIV \longleftrightarrow is-UNIV A
set-eq UNIV A \longleftrightarrow is-UNIV A
\langle \text{proof} \rangle

lemma [code-unfold del, symmetric, code-post del]:
x \in \text{set } \text{xs } \equiv \text{List.member } \text{xs } x
\langle \text{proof} \rangle

lemma [code-unfold del, symmetric, code-post del]:
finite \equiv \text{Cardinality.finite'} \langle \text{proof} \rangle

lemma [code-unfold del, symmetric, code-post del]:
card \equiv \text{Cardinality.card'} \langle \text{proof} \rangle
declare [\langle \text{code drop} \rangle:
Set.empty
Set.is-empty
uminus-set-inst.uminus-set
Set.member
Set.insert
Set.remove
UNIV
Set.filter
image
Set.subset-eq
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

Ball
Bex
Set.union
minus-set-inst.minus-set
Set.inter
card
Set.bind
the-elem
Pow
setsum
Product-Type.product
Id-on
Image
trancl
relcomp
wf
Min
Inf-fin
INFIMUM
Max
Sup-fin
SUPREMUM
Inf :: 'a set set ⇒ 'a set
Sup :: 'a set set ⇒ 'a set
sorted-list-of-set
List.map-project
Sup-pred-inst.Sup-pred
finite
Cardinality.finite'
card
Cardinality.card'
Inf-pred-inst.Inf-pred
pred-of-set
Cardinality.subset'
Cardinality.eq-set
Wellfounded.acc
Bleast
can-select
set-eq :: 'a set ⇒ 'a set ⇒ bool
irrefl
bacc
set-of-pred
set-of-seq
]

declare
finite'-def[code]
card'=def[code]
3.12.3 Set implementations

definition Collect-set :: ('a ⇒ bool) ⇒ 'a set
where [simp]: Collect-set = Collect

definition DList-set :: 'a :: ceq set-dlist ⇒ 'a set
where DList-set = Collect o DList-Set.member

definition RBT-set :: 'a :: ccompare set-rbt ⇒ 'a set
where RBT-set = Collect o RBT-Set2.member

definition Complement :: 'a set ⇒ 'a set
where [simp]: Complement A = − A

definition Set-Monad :: 'a list ⇒ 'a set
where [simp]: Set-Monad = set

code-datatype Collect-set DList-set RBT-set Set-Monad Complement

lemma DList-set-empty [simp]: DList-set DList-Set.empty = {}
⟨proof⟩

lemma RBT-set-empty [simp]: RBT-set RBT-Set2.empty = {}
⟨proof⟩

lemma RBT-set-conv-keys:
  ID CCOMPARE('a :: ccompare) ≠ None
  ⇒ RBT-set (t :: 'a set-rbt) = set (RBT-Set2.keys t)
⟨proof⟩

3.12.4 Set operations

A collection of all the theorems about Complement.
⟨ML⟩

Various fold operations over sets

typedef ('a, 'b) comp-fun-commute = {f :: 'a ⇒ 'b ⇒ 'b. comp-fun-commute f}
morphisms comp-fun-commute-apply Abs-comp-fun-commute
⟨proof⟩

setup-lifting type-definition-comp-fun-commute

lemma comp-fun-commute-apply' [simp]:
  comp-fun-commute (comp-fun-commute-apply f)
⟨proof⟩

lift-definition set-fold-cfc :: ('a, 'b) comp-fun-commute ⇒ 'b ⇒ 'a set ⇒ 'b is
Finite-Set.fold ⟨proof⟩
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

declare [[code drop: set-fold-cfc]]

lemma set-fold-cfc-code [code]:
fixes xs :: 'a :: ceq list
and dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt
shows set-fold-cfc-Complement[set-complement-code]:
set-fold-cfc f'''' b (Complement A) = Code.abort (STR "set-fold-cfc not supported on Complement") (λ- set-fold-cfc f'''' b (Complement A))
and set-fold-cfc f'''' b (Collect-set P) = Code.abort (STR "set-fold-cfc Collect-set: ceq = None") (λ- set-fold-cfc f'''' b (Collect-set P))
set-fold-cfc f b (Set-Monad xs) = (case ID CEQ('a) of None ⇒ Code.abort (STR "set-fold-cfc Set-Monad: ceq = None") (λ- set-fold-cfc f b (Set-Monad xs)))
| Some eq ⇒ List.fold (comp-fun-commute-apply f) (equal-base.list-remdups eq xs) b)
(is ?Set-Monad)
| Some - ⇒ DList-Set.fold (comp-fun-commute-apply f') dxs b)
(is ?DList-set)
set-fold-cfc f'' b (RBT-set rbt) = (case ID CCOMPARE('b) of None ⇒ Code.abort (STR "set-fold-cfc RBT-set: ccompare = None") (λ- set-fold-cfc f'' b (RBT-set rbt)))
| Some - ⇒ RBT-Set2.fold (comp-fun-commute-apply f'') rbt b)
(is ?RBT-set)
⟨proof⟩

typedef ('a, 'b) comp-fun-idem = {f :: 'a ⇒ 'b ⇒ 'b. comp-fun-idem f}
morphisms comp-fun-idem-apply Abs-comp-fun-idem
⟨proof⟩

setup-lifting type-definition-comp-fun-idem

lemma comp-fun-idem-apply' [simp]:
comp-fun-idem (comp-fun-idem-apply f)
⟨proof⟩

lift-definition set-fold-cfi :: ('a, 'b) comp-fun-idem ⇒ 'b ⇒ 'a set ⇒ 'b is Finite-Set.fold
⟨proof⟩

declare [[code drop: set-fold-cfi]]

lemma set-fold-cfi-code [code]:
fixes xs :: 'a list
and dxs :: 'b :: ceq set-dlist
and rbt :: 'c :: ccompare set-rbt shows
set-fold-cf \( f \) \( b \) (Complement \( A \)) = Code.abort (STR "set-fold-cf not supported on Complement") \( (\lambda\cdot \text{set-fold-cf} \ f \ b \ (\text{Complement} \ A)) \)

set-fold-cf \( f \) \( b \) (Collect \( P \)) = Code.abort (STR "set-fold-cf not supported on Collect") \( (\lambda\cdot \text{set-fold-cf} \ f \ b \ (\text{Collect} \ P)) \)

set-fold-cf \( f \) \( b \) (Set-Monad \( xs \)) = \( \text{List.fold} \ (\text{comp-fun-idem-apply} \ f) \) \( xs \) \( b \) \( (\text{is ?Set-Monad}) \)

set-fold-cf \( f' \) \( b \) (DList-set \( dxs \)) =

\[
\begin{align*}
\text{(case ID CEQ('b) of None ⇒ Code.abort (STR "set-fold-cf DList-set: ceq = None") (\lambda\cdot \text{set-fold-cf} \ f' \ b \ (\text{DList-set} \ dxs))} \\
\text{| Some - ⇒ DList-Set.fold} \ (\text{comp-fun-idem-apply} \ f') \ dxs \ b
\end{align*}
\]

\( (\text{is ?DList-set}) \)

set-fold-cf \( f'' \) \( b \) (RBT-set \( rbt \)) =

\[
\begin{align*}
\text{(case ID CCOMPARE('c) of None ⇒ Code.abort (STR "set-fold-cf RBT-set: ccompare = None") (\lambda\cdot \text{set-fold-cf} \ f'' \ b \ (\text{RBT-set} \ rbt))} \\
\text{| Some - ⇒ RBT-Set2.fold} \ (\text{comp-fun-idem-apply} \ f'') \ rbt \ b
\end{align*}
\]

\( (\text{is ?RBT-set}) \)

\langle proof \rangle

typedef 'a semilattice-set = \{ f :: 'a ⇒ 'a ⇒ 'a. semilattice-set \( f \) \}

\langle proof \rangle

setup-lifting type-definition-semilattice-set

\langle proof \rangle

lemma semilattice-set-apply' [simp]:

\langle proof \rangle

lemma comp-fun-idem-semilattice-set-apply [simp]:

\langle proof \rangle

lift-definition set-fold1 :: 'a semilattice-set ⇒ 'a set ⇒ 'a is semilattice-set.F

\langle proof \rangle

lemma (in semilattice-set) F-set-conv-fold:

\[
xs \not= [] \Rightarrow F \ (set \ xs) = \text{Finite-Set.fold} \ f \ (hd \ xs) \ (set \ (tl \ xs))
\]

\langle proof \rangle

lemma set-fold1-code [code]:

\langle proof \rangle

\langle proof \rangle

\langle proof \rangle
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

and

\text{set-fold1 } f' \text{ (DList-set } dxs) =
\begin{align*}
\text{(case ID CEQ('b) of None } & \Rightarrow \text{ Code.abort (STR "set-fold1 DList-set: ceq = None") (λ- set-fold1 } f' \text{ (DList-set } dxs)) \\
| \text{ Some } - & \Rightarrow \text{ if DList-Set.null } dxs \text{ then Code.abort (STR "set-fold1 DList-set: empty set") (λ- set-fold1 } f' \text{ (DList-set } dxs) \\
& \text{ else DList-Set.fold (semilattice-set-apply } f') \text{ (DList-Set } tl \\
\text{ dxs) (DList-Set.hd } dxs))}
\end{align*}

\text{(is ?DList-set)

and

\text{set-fold1 } f'' \text{ (RBT-set } rbt) =
\begin{align*}
\text{(case ID CCOMPARE('a) of None } & \Rightarrow \text{ Code.abort (STR "set-fold1 RBT-set: ccompare = None") (λ- set-fold1 } f'' \text{ (RBT-set } rbt)) \\
| \text{ Some } - & \Rightarrow \text{ if RBT-Set2.is-empty } rbt \text{ then Code.abort (STR "set-fold1 RBT-set: empty set") (λ- set-fold1 } f'' \text{ (RBT-set } rbt) \\
& \text{ else RBT-Set2.fold1 (semilattice-set-apply } f'') \text{ rbt)}
\end{align*}

\text{(is ?RBT-set)

(⟨proof⟩

Implementation of set operations

\textbf{lemma} \text{Collect-code [code]:}

\text{fixes } P : : 'a :: cenum ⇒ bool \text{ shows}

\text{Collect } P =
\begin{align*}
\text{(case ID CENUM('a) of None } & \Rightarrow \text{ Collect-set } P \\
| \text{ Some } (\text{enum, }-) & \Rightarrow \text{ Set-Monad } (\text{filter } P \text{ enum})
\end{align*}

(⟨proof⟩

\textbf{lemma} \text{finite-code [code]:}

\text{fixes } dxs :: 'a :: ceq set-dlist \\
\text{and } rbt :: 'b :: ccompare set-rbt \\
\text{and } A :: 'c :: finite-UNIV set \text{ and } P :: 'c ⇒ bool \text{ shows}

\text{finite } (\text{DList-set } dxs) =
\begin{align*}
\text{(case ID CEQ('a) of None } & \Rightarrow \text{ Code.abort (STR "finite DList-set: ceq = None") (λ- finite } (\text{DList-set } dxs)) \\
| \text{ Some } - & \Rightarrow \text{ True})
\end{align*}

\text{finite } (\text{RBT-set } rbt) =
\begin{align*}
\text{(case ID CCOMPARE('b) of None } & \Rightarrow \text{ Code.abort (STR "finite RBT-set: ccompare = None") (λ- finite } (\text{RBT-set } rbt)) \\
| \text{ Some } - & \Rightarrow \text{ True})
\end{align*}

\text{and finite-Complement [set-complement-code]:}

\text{finite } (\text{Complement } A) \leftrightarrow
\begin{align*}
\text{(if of-phantom } (\text{finite-UNIV :: 'c finite-UNIV}) \text{ then True) \\
& \text{ else if finite } A \text{ then False)}
\end{align*}

\text{else Code.abort (STR "finite Complement: infinite set") (λ- finite } (\text{Complement } A))

\text{and}

\text{finite } (\text{Set-Monad } xs) = \text{ True}

\text{finite } (\text{Collect-set } P) \leftrightarrow
\begin{align*}
\text{of-phantom } (\text{finite-UNIV :: 'c finite-UNIV}) \lor \text{ Code.abort (STR "finite Collect-set")}
\end{align*}
104

CHAPTER 3. LIGHT-WEIGHT CONTAINERS

(\lambda.- finite (Collect-set P))

(proof)

lemma card-code [code]:
  fixes dxs :: 'a :: ceq set-dlist and xs :: 'a list
  and A :: 'c :: card-UNIV set shows
  card (DList-set dxs) =
  (case ID CEQ('a) of None \Rightarrow Code.abort (STR "card DList-set: ceq = None")
  (\lambda.- card (DList-set dxs)))
  | Some \Rightarrow \lambda.- card (DList-set dxs)

  card (RBT-set rbt) =
  (case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "card RBT-set: ccompare = None")
  (\lambda.- card (RBT-set rbt)))
  | Some \Rightarrow length (RBT-Set2.keys rbt)

  card (Set-Monad xs) =
  (case ID CEQ('a) of None \Rightarrow Code.abort (STR "card Set-Monad: ceq = None")
  (\lambda.- card (Set-Monad xs)))
  | Some eq \Rightarrow length (equal-base.list-remdups eq xs)

  and card-Complement [set-complement-code]:
  card (Complement A) =
  (let a = card A; s = CARD('c)
    in if s > 0 then s - a
    else if finite A then 0
    else Code.abort (STR "card Complement: infinite") (\lambda.- card (Complement A)))

(proof)

lemma is-UNIV-code [code]:
  fixes rbt :: 'a :: {cproper-interval, finite-UNIV} set-rbt
  and A :: 'b :: card-UNIV set shows
  is-UNIV A \leftrightarrow
  (let a = CARD('b);
    b = card A
    in if a > 0 then a = b
    else if b > 0 then False
    else Code.abort (STR "is-UNIV called on infinite type and set") (\lambda.- is-UNIV A))

(is ?generic)
  is-UNIV (RBT-set rbt) =
  (case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "is-UNIV RBT-set: ccompare = None")
  (\lambda.- is-UNIV (RBT-set rbt)))
  | Some \Rightarrow of-phantom (finite-UNIV :: 'a finite-UNIV) \land
  proper-intrvl.exhaustive-fusion cproper-interval rbt-keys-generator (RBT-Set2.init rbt)

(is ?rbt)

(proof)

lemma is-empty-code [code]:
\textbf{3.12. DIFFERENT IMPLEMENTATIONS OF SETS}

\begin{verbatim}
fixes \( dxs :: 'a :: coq set-dlist \)
\textbf{and} \( rbt :: 'b :: ccompare set-rbt \)
\textbf{and} \( A :: 'c set \)
\textbf{shows}
\( \text{Set.is-empty (Set-Monad xs)} \leftrightarrow xs = [] \)
\( \text{Set.is-empty (DList-set dxs)} \leftrightarrow \)
\( \text{(case ID CEQ('a) of None \Rightarrow Code.abort (STR "is-empty DList-set: ceq = None") (λ- Set.is-empty (RBT-set rbt))}
\quad | \text{Some - ⇒ DList-Set.null dxs) (is ?DList-set)} \)
\( \text{Set.is-empty (RBT-set rbt)} \leftrightarrow \)
\( \text{(case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "is-empty RBT-set: ccompare = None") (λ- Set.is-empty (RBT-set rbt))}
\quad | \text{Some - ⇒ RBT-Set2.is-empty rbt) (is ?RBT-set)} \)
\text{and is-empty-Complement [set-complement-code]:}
\( \text{Set.is-empty (Complement A) \leftrightarrow is-UNIV A (is ?Complement)} \)
<proof>
\end{verbatim}

\textbf{lemma Set-insert-code \[\text{[code]}\]:}
\begin{verbatim}
fixes \( dxs :: 'a :: coq set-dlist \)
\textbf{and} \( rbt :: 'b :: ccompare set-rbt \)
\textbf{shows}
\( \text{\( \forall x. \text{Set.insert x (Collect-set A) =}
\text{(case ID CEQ('a) of None \Rightarrow Code.abort (STR "insert Collect-set: ceq = None")}
\text{(λ- Set.insert x (Collect-set A))}
\text{Some eq ⇒ Collect-set (equal-base.fun-upd eq A x True))}
\text{\( \forall x. \text{Set.insert x (DList-set dxs)} \text{=} (case ID CEQ('a) of None \Rightarrow Code.abort (STR "insert DList-set: ceq = None")}
\text{(λ- Set.insert x (DList-set dxs))}
\text{Some - ⇒ DList-set (DList-Set.insert x dxs))}
\text{\( \forall x. \text{Set.insert x (RBT-set rbt) =}
\text{(case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "insert RBT-set: ccompare = None")}
\text{(λ- Set.insert x (RBT-set rbt))}
\text{Some - ⇒ RBT-set (RBT-Set2.insert x rbt))}
\text{\( \text{and insert-Complement [set-complement-code]:}
\text{\( \forall x. \text{Set.insert x (Complement X) = Complement (Set.remove x X)} \)
\text{<proof>}
\end{verbatim}

\textbf{lemma Set-member-code \[\text{[code]}\]:}
\begin{verbatim}
fixes \( xs :: 'a :: coq list \)
\textbf{shows}
\( \text{\( \forall x. x \in \text{Collect-set A} \leftrightarrow A x \)
\text{\( \forall x. x \in \text{DList-set dxs} \leftrightarrow \text{DList-Set.member dxs x}
\text{\( \forall x. x \in \text{RBT-set rbt} \leftrightarrow \text{RBT-Set2.member rbt x}
\text{\( \text{and mem-Complement [set-complement-code]:}
\text{\( \forall x. x \in \text{Complement X} \leftrightarrow x \notin X \)
\text{\( \text{\( \forall x. x \in \text{Set-Monad xs} \leftrightarrow}
\text{(case ID CEQ('a) of None \Rightarrow Code.abort (STR "member Set-Monad: ceq = None")}
\text{(λ- x \in \text{Set-Monad xs)}
\text{Some eq ⇒ equal-base.list-member eq xs x)}
\text{<proof>}
\end{verbatim}
"CHAPTER 3. LIGHT-WEIGHT CONTAINERS"

lemma Set-remove-code [code]:
fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: ceq set-dlist shows
\( \forall x. \text{Set.remove} \ x \ (\text{Collect-set} \ A) = \)
\( (\text{case} \ \text{ID} \ \text{CEQ}(\ 'b \ ) \ \text{of} \ \text{None} \Rightarrow \text{Code.abort} \ (\text{STR} \ "\text{remove} \ \text{Collect} : \ ceq = \ \text{None}")) \)
\( (\lambda -. \text{Set.remove} \ x \ (\text{Collect-set} \ A)) \)
\( | \ \text{Some} \ \text{eq} \Rightarrow \text{Collect-set} \ (\text{equal-base.fun-upd} \ \text{eq} \ A \ \text{x False}) \)
\( \forall x. \text{Set.remove} \ x \ (\text{DList-set} \ dxs) = \)
\( (\text{case} \ \text{ID} \ \text{CEQ}(\ 'b \ ) \ \text{of} \ \text{None} \Rightarrow \text{Code.abort} \ (\text{STR} \ "\text{remove} \ \text{DList-set} : \ ceq = \ \text{None}")) \)
\( (\lambda -. \text{Set.remove} \ x \ (\text{DList-set} \ dxs)) \)
\( | \ \text{Some} -. \Rightarrow \text{DList-set} \ (\text{DList-Set.remove} \ dxs) \)
\( \forall x. \text{Set.remove} \ x \ (\text{RBT-set} \ rbt) = \)
\( (\text{case} \ \text{ID} \ \text{CCOMPARE}(\ 'a \ ) \ \text{of} \ \text{None} \Rightarrow \text{Code.abort} \ (\text{STR} \ "\text{remove} \ \text{RBT-set} : \ ccompare = \ \text{None}")) \)
\( (\lambda -. \text{Set.remove} \ x \ (\text{RBT-set} \ rbt)) \)
\( | \ \text{Some} -. \Rightarrow \text{RBT-set} \ (\text{RBT-Set2.remove} \ x \ rbt) \)
and remove-Complement [set-complement-code]:
\( \forall x A. \text{Set.remove} \ x \ (\text{Complement} \ A) = \text{Complement} \ (\text{Set.insert} \ x \ A) \)
⟨proof⟩

lemma Set-uminus-code [code, set-complement-code]:
\(- A = \text{Complement} \ A \)
\(- (\text{Collect-set} \ P) = \text{Collect-set} \ (\lambda x. \neg P \ x) \)
\(- (\text{Complement} \ B) = B \)
⟨proof⟩

These equations represent complements as true complements. If you want that the complement operations returns an explicit enumeration of the elements, use the following set of equations which use cenum.

lemma Set-uminus-cenum:
fixes A :: 'a :: cenum set shows
\(- A = \)
\( (\text{case} \ \text{ID} \ \text{CENUM}(\ 'a \ ) \ \text{of} \ \text{None} \Rightarrow \text{Complement} \ A \)
\( | \ \text{Some} \ \text{enum, -} \Rightarrow \text{Set-Monad} \ (\text{filter} \ (\lambda x. \ x /\in A) \ \text{enum}) \)
\( \and - (\text{Complement} \ B) = B \)
⟨proof⟩

lemma Set-minus-code [code]: \(- A = A \cap - B \)
⟨proof⟩

lemma Set-union-code [code]:
fixes rbt1 rbt2 :: 'a :: ccompare set-rbt
and rbt :: 'b :: {ccompare, ceq} set-rbt
and dxs :: 'b set-dlist
and dxs1 dxs2 :: 'c :: ceq set-dlist shows
\( \text{RBT-set} \ rbt1 \cup \text{RBT-set} \ rbt2 = \)
\( (\text{case} \ \text{ID} \ \text{CCOMPARE}(\ 'a \ ) \ \text{of} \ \text{None} \Rightarrow \text{Code.abort} \ (\text{STR} \ "\text{union} \ \text{RBT-set} \ \text{RBT-set} : \ ccompare = \ \text{None}")) \)
\( (\lambda -. \text{RBT-set} \ rbt1 \cup \text{RBT-set} \ rbt2) \)
\( | \ \text{Some} -. \Rightarrow \text{RBT-set} \ (\text{RBT-Set2.union} \ rbt1 \ rbt2) \)
⟨is
### 3.12. DIFFERENT IMPLEMENTATIONS OF SETS

\[
\text{RBT-set-RBT-set} = \\
\text{case ID CCOMPARE('b) of None ⇒ Code.abort } \text{(STR "union RBT-set DList-set: ccompare = None") (λ- RBT-set rbt ∪ DList-set dxs)} \\
\text{ Some - ⇒ } \\
\text{case ID CEQ('b) of None ⇒ Code.abort } \text{(STR "union RBT-set DList-set: ceq = None") (λ- RBT-set rbt ∪ DList-set dxs)} \\
\text{ Some - ⇒ RBT-set (DList-Set.fold RBT-Set2.insert dxs rbt)} \\
\text{is RBT-set-DList-set} \\
\text{DList-set dxs ∪ RBT-set rbt = } \\
\text{case ID CCOMPARE('b) of None ⇒ Code.abort } \text{(STR "union DList-set RBT-set: ccompare = None") (λ- RBT-set rbt ∪ DList-set dxs)} \\
\text{ Some - ⇒ RBT-set (DList-Set.fold RBT-Set2.insert dxs rbt)} \\
\text{is DList-set-RBT-set} \\
\text{DList-set dxs1 ∪ DList-set dxs2 = } \\
\text{case ID CEQ('c) of None ⇒ Code.abort } \text{(STR "union DList-set DList-set: ceq = None") (λ- DList-set dxs1 ∪ DList-set dxs2)} \\
\text{ Some - ⇒ DList-set (DList-Set.union dxs1 dxs2)} \text{ is DList-set-DList-set} \\
\text{Set-Monad zs ∪ RBT-set rbt2 = } \\
\text{case ID CCOMPARE('a) of None ⇒ Code.abort } \text{(STR "union Set-Monad RBT-set: ccompare = None") (λ- Set-Monad zs ∪ RBT-set rbt2)} \\
\text{ Some - ⇒ RBT-set (fold RBT-Set2.insert zs rbt2)} \text{ is Set-Monad-RBT-set} \\
\text{RBT-set rbt1 ∪ Set-Monad zs = } \\
\text{case ID CCOMPARE('a) of None ⇒ Code.abort } \text{(STR "union RBT-set Set-Monad: ccompare = None") (λ- RBT-set rbt1 ∪ Set-Monad zs)} \\
\text{ Some - ⇒ RBT-set (fold RBT-Set2.insert rs rbt1)} \text{ is RBT-set-Set-Monad} \\
\text{Set-Monad ws ∪ DList-set dxs2 = } \\
\text{case ID CEQ('c) of None ⇒ Code.abort } \text{(STR "union Set-Monad DList-set: ceq = None") (λ- Set-Monad ws ∪ DList-set dxs2)} \\
\text{ Some - ⇒ DList-set (fold DList-Set.insert ws dxs2)} \text{ is Set-Monad-DList-set} \\
\text{DList-set dxs1 ∪ Set-Monad ws = } \\
\text{case ID CEQ('c) of None ⇒ Code.abort } \text{(STR "union DList-set Set-Monad: ceq = None") (λ- DList-set dxs1 ∪ Set-Monad ws)} \\
\text{ Some - ⇒ DList-set (fold DList-Set.insert ws dxs1)} \text{ is Set-Monad-DList-set} \\
\text{Set-Monad xs ∪ Set-Monad ys = Set-Monad (xs @ ys)} \\
\text{Collect-set A ∪ B = Collect-set (λx. A x ∨ x ∈ B)} \\
\text{B ∪ Collect-set A = Collect-set (λx. A x ∨ x ∈ B)} \\
\text{and Set-union-Complement [set-complement-code]:} \\
\text{Complement B ∪ B' = Complement (B ∩ − B')} \\
\text{B' ∪ Complement B = Complement (− B' ∩ B)} \\
\text{proof}
lemma Set-inter-code [code]:

\[\text{fixes } \text{rbt1 rbt2 :: 'a :: compare set-rbt} \]
\[\text{and } \text{rbt :: 'b :: \{compare, ceq\} set-rbt} \]
\[\text{and } \text{dxs :: 'b set-dlist} \]
\[\text{and } \text{dxs1 dxs2 :: 'c :: ceq set-dlist} \]
\[\text{and } \text{xs1 xs2 :: 'c list} \]

\text{shows}
\[\text{Collect-set } A'' \cap J = \text{Collect-set } (\lambda x. A'' x \land x \in J) \text{ (is ?collect1)} \]
\[J \cap \text{Collect-set } A'' = \text{Collect-set } (\lambda x. A'' x \land x \in J) \text{ (is ?collect2)} \]
\[\text{Set-Monad } xs'' \cap I = \text{Set-Monad } (\text{filter } (\lambda x. x \in I) \text{ xs''}) \text{ (is ?monad1)} \]
\[I \cap \text{Set-Monad } xs'' = \text{Set-Monad } (\text{filter } (\lambda x. x \in I) \text{ xs''}) \text{ (is ?monad2)} \]
\[\text{DList-set } dxs1 \cap H = \]
\[\text{(case ID CEQ('c) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter DList-set1: ceq = None"}) \]
\[\text{(\lambda -. DList-set dxs1 \cap H)} \]
\[\mid \text{Some eq } \Rightarrow \text{DList-set } (\text{DList-Set.filter } (\lambda x. x \in H) \text{ dxs1}) \text{ (is ?dlist1)} \]
\[H \cap \text{DList-set } dxs2 = \]
\[\text{(case ID CEQ('c) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter DList-set2: ceq = None"}) \]
\[\text{(\lambda -. H \cap \text{DList-set } dxs2)} \]
\[\mid \text{Some eq } \Rightarrow \text{DList-set } (\text{DList-Set.filter } (\lambda x. x \in H) \text{ dxs2}) \text{ (is ?dlist2)} \]
\[\text{RBT-set } rbt1 \cap G = \]
\[\text{(case ID CCOMPARE('a) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter RBT-set1: compare = None"}) \]
\[\text{(\lambda -. RBT-set rbt1 \cap G)} \]
\[\mid \text{Some - } \Rightarrow \text{RBT-set } (\text{RBT-Set2.filter } (\lambda x. x \in G) \text{ rbt1}) \text{ (is ?rbt1)} \]
\[G \cap \text{RBT-set rbt2} = \]
\[\text{(case ID CCOMPARE('a) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter RBT-set2: compare = None"}) \]
\[\text{(\lambda -. G \cap \text{RBT-set } rbt2)} \]
\[\mid \text{Some - } \Rightarrow \text{RBT-set } (\text{RBT-Set2.filter } (\lambda x. x \in G) \text{ rbt2}) \text{ (is ?rbt2)} \]

\text{and Set-inter-Complement [set-complement-code]:}
\[\text{Complement } B'' \cap \text{Complement } B''' = \text{Complement } (B'' \cup B''') \text{ (is ?complement)}\]

\text{and}
\[\text{Set-Monad } xs \cap \text{RBT-set } rbt1 = \]
\[\text{(case ID CCOMPARE('a) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter Set-Monad RBT-set: compare = None"}) \]
\[\text{(\lambda -. RBT-set rbt1 \cap \text{Set-Monad } xs)} \]
\[\mid \text{Some - } \Rightarrow \text{RBT-set } (\text{RBT-Set2.inter-list } \text{rbt1 } \text{xs}) \text{ (is ?monad-rbt)} \]
\[\text{Set-Monad } xs' \cap \text{DList-set } dxs2 = \]
\[\text{(case ID CEQ('c) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter Set-Monad DList-set: ceq = None"}) \]
\[\text{(\lambda -. Set-Monad } xs' \cap \text{DList-set } dxs2) \]
\[\mid \text{Some eq } \Rightarrow \text{DList-set } (\text{DList-Set.filter } (\text{equal-base.list-member eq xs'} \text{ dxs2}) \text{ (is ?monad-dlist)} \]
\[\text{Set-Monad } xs1 \cap \text{Set-Monad } xs2 = \]
\[\text{(case ID CEQ('c) of None } \Rightarrow \text{Code.abort } (\text{STR } \text{"inter Set-Monad Set-Monad:} \]

\text{...}
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

\(ceq = None'\) (\(\lambda-.\) Set-Monad \(xs1 \cap\) Set-Monad \(xs2\))  
\(|\) Some \(eq \Rightarrow\) Set-Monad (filter (equal-base.list-member \(eq\) \(xs2\) \(xs1\)))  
(is ?monad)

\(DL\text{-}list\text{-}set\) \(dxs1 \cap\) RBT-set \(rbt\) =  
(case ID CCOMPARE('b') of None ⇒ Code.abort (STR "inter DList-set RBT-set: ccompare = None") (\(\lambda-.\) DList-set \(dxs1 \cap\) RBT-set \(rbt\))  
| Some - ⇒  
\(\lambda-.\) DList-set \(dxs1 \cap\) RBT-set \(rbt\)  
| Some - ⇒ DList-set (DListSet.filter (DListSet.member \(dxs2\)))  
(is ?dlist-rbt)  
\(DL\text{-}list\text{-}set\) \(dxs2\) =  
(case ID CEQ('c') of None ⇒ Code.abort (STR "inter DList-set DList-set: ceq = None") (\(\lambda-.\) DList-set \(dxs1 \cap\) DList-set \(dxs2\))  
| Some - ⇒ DList-set (DListSet.filter (equal-base.list-member eq \(xs1\) \(dxs1\)))  
(is ?dlist-monad)  
\(RBT\text{-}set\) \(rbt1 \cap\) RBT-set \(rbt2\) =  
(case ID CCOMPARE('a') of None ⇒ Code.abort (STR "inter RBT-set RBT-set: ccompare = None") (\(\lambda-.\) RBT-set \(rbt1 \cap\) RBT-set \(rbt2\))  
| Some - ⇒ RBT-set (RBTSet2.inter-list rbt1 rbt2)  
(is ?rbt-rbt)  
\(RBT\text{-}set\) \(rbt\) \(\cap\) DList-set \(dxs\) =  
(case ID CCOMPARE('b') of None ⇒ Code.abort (STR "inter RBT-set DList-set: ccompare = None") (\(\lambda-.\) RBT-set \(rbt\) \(\cap\) DList-set \(dxs\))  
| Some - ⇒  
\(\lambda-.\) RBT-set \(rbt\) \(\cap\) DList-set \(dxs\)  
| Some - ⇒ RBT-set (RBTSet2.inter-list rbt (list-of-dlist \(dxs\)))  
(is ?rbt-dlist)  
\(RBT\text{-}set\) \(rbt1 \cap\) Set-Monad \(xs\) =  
(case ID CCOMPARE('a') of None ⇒ Code.abort (STR "inter RBT-set Set-Monad: ccompare = None") (\(\lambda-.\) RBT-set \(rbt1 \cap\) Set-Monad \(xs\))  
\(|\) Some - ⇒ RBT-set (RBTSet2.inter-list \(rbt1\) \(xs\))  
(is ?rbt-monad)

(proof)

**Lemma** Set-bind-code [code];

**fixes** \(dxs::'a::\) ccompare set-dlist  
and \(rbt::'b::\) ccompare set-rbt shows  
Set.bind (Set-Monad \(xs\)) \(f =\) fold \((op \cup \circ f)\) \(xs\) (Set-Monad []) (is ?Set-Monad)  
Set.bind (DList-set \(dxs\)) \(f\) =  
(case ID CEQ('a') of None ⇒ Code.abort (STR "bind DList-set: ceq = None")  
\(\lambda-.\) Set.bind (DList-set \(dxs\)) \(f\))
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

| Some - ⇒ DList-Set.fold (union o f') dxs {} | \( \text{is } ?DList \)
\[
\text{Set.bind (RBT-set rbt) f''} =
\begin{cases}
\text{case ID CCOMPARE('b) of None ⇒ Code.abort (STR "bind RBT-set: ccompare = None") (λ- Set.bind (RBT-set rbt) f'')} \\
\text{Some - ⇒ RBT-Set2.fold (union o f'') rbt {}} \text{ (is } ?RBT) 
\end{cases}
\]

\langle proof \rangle

**lemma** UNIV-code [code]: UNIV = – {}
\langle proof \rangle

**lift-definition** inf-sls :: 'a :: lattice semilattice-set is inf  \langle proof \rangle

**lemma** Inf-fin-code [code]: Inf-fin A = set-fold1 inf-sls A  \langle proof \rangle

**lift-definition** sup-sls :: 'a :: lattice semilattice-set is sup  \langle proof \rangle

**lemma** Sup-fin-code [code]: Sup-fin A = set-fold1 sup-sls A  \langle proof \rangle

**lift-definition** inf-cfi :: ('a :: lattice, 'a) comp-fun-idem is inf 
\langle proof \rangle

**lemma** Inf-code:
\[
\text{fixes } A :: 'a :: complete-lattice set shows}
\text{Inf A = (if finite A then set-fold-cfi inf-cfi top A else Code.abort (STR "Inf: infinite") (λ- Inf A))}
\langle proof \rangle

**lift-definition** sup-cfi :: ('a :: lattice, 'a) comp-fun-idem is sup  \langle proof \rangle

**lemma** Sup-code:
\[
\text{fixes } A :: 'a :: complete-lattice set shows}
\text{Sup A = (if finite A then set-fold-cfi sup-cfi bot A else Code.abort (STR "Sup: infinite") (λ- Sup A))}
\langle proof \rangle

**lemmas** Inter-code [code] = Inf-code[where ?'a = - :: type set]
**lemmas** Union-code [code] = Sup-code[where ?'a = - :: type set]
**lemmas** Predicate-Inf-code [code] = Inf-code[where ?'a = - :: type Predicate.pred]
**lemmas** Predicate-Sup-code [code] = Sup-code[where ?'a = - :: type Predicate.pred]
**lemmas** Inf-fun-code [code] = Inf-code[where ?'a = - :: type ⇒ - :: complete-lattice]
**lemmas** Sup-fun-code [code] = Sup-code[where ?'a = - :: type ⇒ - :: complete-lattice]

**lemma** INF-code [code]: INFIMUM A f = Inf (f ' A)  \langle proof \rangle

**lemma** SUP-code [code]: SUPREMUM A f = Sup (f ' A)
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

⟨proof⟩

**lift-definition** min-sls :: 'a :: linorder semilattice-set is min ⟨proof⟩

**lemma** Min-code [code]: Min A = set-fold1 min-sls A ⟨proof⟩

**lift-definition** max-sls :: 'a :: linorder semilattice-set is max ⟨proof⟩

**lemma** Max-code [code]: Max A = set-fold1 max-sls A ⟨proof⟩

We do not implement Ball, Bex, and sorted-list-of-set for Collect-set using CENUM('a), because it should already have been converted to an explicit list of elements if that is possible.

**lemma** Ball-code [code]:

fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: ceq set-dlist shows
Ball (Set-Monad xs) P = list-all P xs
Ball (DList-set dxs) P' =
  (case ID CEQ('b) of None ⇒ Code.abort (STR "Ball DList-set: ceq = None")
   (λ-. Ball (DList-set dxs) P'))
   | Some - ⇒ DList-Set.dlist-all P' dxs)
Ball (RBT-set rbt) P'' =
  (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "Ball RBT-set: ccompare = None")
   (λ-. Ball (RBT-set rbt) P''))
   | Some - ⇒ RBT-Set2.all P'' rbt)
⟨proof⟩

**lemma** Bex-code [code]:

fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: {ccompare, ceq} set-dlist shows
Bex (Set-Monad xs) P = list-ex P xs
Bex (DList-set dxs) P' =
  (case ID CEQ('b) of None ⇒ Code.abort (STR "Bex DList-set: ceq = None")
   (λ-. Bex (DList-set dxs) P'))
   | Some - ⇒ DList-Set.dlist-ex P' dxs)
Bex (RBT-set rbt) P'' =
  (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "Bex RBT-set: ccompare = None")
   (λ-. Bex (RBT-set rbt) P''))
   | Some - ⇒ RBT-Set2.ex P'' rbt)
⟨proof⟩

**lemma** csorted-list-of-set-code [code]:

fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: {ccompare, ceq} set-dlist
and xs :: 'a :: ccompare list shows
csorted-list-of-set (RBT-set rbt) =
  (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "csorted-list-of-set RBT-set:

⟨proof⟩
\[\text{ccompare} = \text{None}''\) (\(-\), \text{csorted-list-of-set} (\text{RBT-set rbt})))
\[\; | \; \text{Some c} \Rightarrow \text{RBT-Set2 keys rbt})
\[\text{csorted-list-of-set} (\text{DList-set dxs}) =
\quad (\text{case ID CEQ(''b'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{csorted-list-of-set DList-set: ceq} = \text{None}'') (\(-\), \text{csorted-list-of-set} (\text{DList-set dxs})))
\[\; | \; \text{Some c} \Rightarrow \text{RBT-Set2 keys rbt})
\[\text{csorted-list-of-set} (\text{DList-set dxs}) =
\quad (\text{case ID CCOMPARE(''b'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{csorted-list-of-set DList-set: ccompare} = \text{None}'') (\(-\), \text{csorted-list-of-set} (\text{DList-set dxs})))
\[\; | \; \text{Some c} \Rightarrow \text{RBT-Set2 keys rbt})
\quad \text{csorted-list-of-set} (\text{Set-Monad xs}) =
\quad (\text{case ID CCOMPARE(''a'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{csorted-list-of-set Set-Monad: ccompare} = \text{None}'') (\(-\), \text{csorted-list-of-set} (\text{Set-Monad xs})))
\[\; | \; \text{Some c} \Rightarrow \text{ord.remdups-sorted} (\text{lt-of-comp c}) (\text{ord.quicksort} (\text{lt-of-comp c}) \text{xs}))
\quad \langle \text{proof} \rangle
\]

\text{lemma cless-set-code [code]}:
\quad \text{fixes rbt rbt' :: 'a :: compare set-rbt}
\quad \text{and rbt1 rbt2 :: 'b :: cproper-interval set-rbt}
\quad \text{and A B :: 'a set}
\quad \text{and A' B' :: 'b set shows}
\quad \text{cless-set A B} \mapsfrom
\quad (\text{case ID CCOMPARE(''a'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{cless-set: ccompare} = \text{None}'') (\(-\), \text{cless-set A B}))
\[\; | \; \text{Some c} \Rightarrow \]
\quad \text{if finite A} \land \text{finite B then ord.lexordp} (\lambda x y. \text{lt-of-comp c y x}) (\text{csorted-list-of-set A}) (\text{csorted-list-of-set B})
\quad \text{else Code.abort} (\text{STR } ''\text{cless-set: infinite set}'') (\(-\), \text{cless-set A B}))
\quad (\text{is ?fin-fin})
\quad \text{and cless-set-Complement2 [set-complement-code]}:
\quad \text{cless-set A' (Complement B')} \mapsfrom
\quad (\text{case ID CCOMPARE(''b'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{cless-set Complement2: ccompare} = \text{None}'') (\(-\), \text{cless-set A' (Complement B'}))
\[\; | \; \text{Some c} \Rightarrow \]
\quad \text{if finite A' \land finite B' then}
\quad \text{finite} (\text{UNIV :: 'b set} \mapsto
\quad \text{proper-intrel.set-less-aux-Compl} (\text{lt-of-comp c}) \text{cproper-interval None} (\text{csorted-list-of-set A'}) (\text{csorted-list-of-set B'})
\quad \text{else Code.abort} (\text{STR } ''\text{cless-set Complement2: infinite set}'') (\(-\), \text{cless-set A'} (\text{Complement B'}))
\quad (\text{is ?fin-Compl-fin})
\quad \text{and cless-set-Complement1 [set-complement-code]}:
\quad \text{cless-set} (\text{Complement A'}) B' \mapsfrom
\quad (\text{case ID CCOMPARE(''b'') of None} \Rightarrow \text{Code.abort} (\text{STR } ''\text{cless-set Complement1: ccompare} = \text{None}'') (\(-\), \text{cless-set} (\text{Complement A'}) B')
\[\; | \; \text{Some c} \Rightarrow \]
\quad \text{if finite A' \land finite B' then}
\quad \text{finite} (\text{UNIV :: 'b set} \land
\quad \text{proper-intrel.Compl-set-less-aux} (\text{lt-of-comp c}) \text{cproper-interval None} (\text{csorted-list-of-set A'}) (\text{csorted-list-of-set B'})
\quad \text{else Code.abort} (\text{STR } ''\text{cless-set Complement1: infinite set}'') (\(-\), \text{cless-set} (\text{Complement A'}) B')
\quad (\text{is ?fin-Compl-fin})
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

A′) (csorted-list-of-set B′)

else Code.abort (STR "cless-set Complement1: infinite set") (λ· cless-set
(Complement A′) B′))
(is ?Compl-fin-fin)
and cless-set-Complement12 [set-complement-code]:
class-set (Complement A) (Complement B) ←→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cless-set Complement
Complement: compare = None") (λ· cless-set (Complement A) (Complement B))
| Some c ⇒ cless B A) (is ?Compl-Compl)
and
class-set (RBT-set rbt) (RBT-set rbt′) ←→
(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "cless-set RBT-set
RBT-set: ccompare = None") (λ· cless-set (RBT-set rbt) (RBT-set rbt′))
| Some c ⇒ Ord.le-satur=Ord.le
rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))
(is ?rbl-rbt)
and cless-set-rbt-Complement1 [set-complement-code]:
class-set (RBT-set rbt1) (Complement (RBT-set rbt2)) ←→
(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "cless-set RBT-set
(Complement RBT-set): ccompare = None") (λ· cless-set (RBT-set rbt1) (Complement
(RBT-set rbt2)))
| Some c ⇒
finite (UNIV :: 'a set) ∧
proper-intervl.Compl-set-less-aux-fusion (lt-comp c) cproper-interval rbt-keys-generator
rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))
(is ?Compl-rbt)
⟨proof⟩

lemma le-of-comp-set-less-eq:
le-of-comp (comp-of-ords (ord.set-less-eq le) (ord.set-less le)) = ord.set-less-eq le
⟨proof⟩

lemma cless-eq-set-code [code]:
fixes rbt rbt′ :: 'a :: ccompare set-rbt
and rbt1 rbt2 :: 'b :: cproper-interval set-rbt
and A B :: 'a set
and A′ B′ :: 'b set shows
class-eq-set A B ←→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cless-eq-set: ccompare
\[= \text{None} (\lambda . \text{cless-eq-set } A \ B)\]
\[
| \text{Some } c \Rightarrow \\
\quad \text{if finite } A \land \text{finite } B \text{ then} \\
\quad \text{ord.lexord-eq } (\lambda x y. \text{lt-of-comp } c \ x \ y) (\text{csorted-list-of-set } A) (\text{csorted-list-of-set } B)
\]
\[
\quad \text{else Code.abort } (\text{STR } "\text{cless-eq-set: infinite set}" (\lambda . \text{cless-eq-set } A \ B))
\]
\[
(\text{is } \text{?fin-fin})
\quad \text{and } \text{cless-eq-set-Complement2 } [\text{set-complement-code}]:
\quad \text{cless-eq-set } A' (\text{Complement } B') \leftarrow \rightarrow
\quad (\text{case ID CCOMPARE}'(b) \text{ of None } \Rightarrow \text{Code.abort } (\text{STR } "\text{cless-eq-set Complement2: compare } = \text{None}" (\lambda . \text{cless-eq-set } A' (\text{Complement } B')))
\]
\[
| \text{Some } c \Rightarrow \\
\quad \text{if finite } A' \land \text{finite } B' \text{ then} \\
\quad \text{finite } (\text{UNIV } :: 'b \text{ set}) \rightarrow \\
\quad \text{proper-intrel.set-less-eq-aux-Compl } (\text{lt-of-comp } c) \text{ cproper-interval } \text{None} (\text{csorted-list-of-set } A') (\text{csorted-list-of-set } B')
\]
\[
\quad \text{else Code.abort } (\text{STR } "\text{cless-eq-set Complement2: infinite set}" (\lambda . \text{cless-eq-set } A' (\text{Complement } B')))
\]
\[
(\text{is } \text{?fin-Compl-fin})
\quad \text{and } \text{cless-eq-set-Complement1 } [\text{set-complement-code}]:
\quad \text{cless-eq-set } (\text{Complement } A') B' \leftarrow \rightarrow
\quad (\text{case ID CCOMPARE}'(b) \text{ of None } \Rightarrow \text{Code.abort } (\text{STR } "\text{cless-eq-set Complement1: compare } = \text{None}" (\lambda . \text{cless-eq-set } (\text{Complement } A') B'))
\]
\[
| \text{Some } c \Rightarrow \\
\quad \text{if finite } A' \land \text{finite } B' \text{ then} \\
\quad \text{finite } (\text{UNIV } :: 'b \text{ set}) \land \\
\quad \text{proper-intrel.Compl-set-less-eq-aux } (\text{lt-of-comp } c) \text{ cproper-interval } \text{None} (\text{csorted-list-of-set } A') (\text{csorted-list-of-set } B')
\]
\[
\quad \text{else Code.abort } (\text{STR } "\text{cless-eq-set Complement1: infinite set}" (\lambda . \text{cless-eq-set } (\text{Complement } A') B'))
\]
\[
(\text{is } \text{?Compl-fin-fin})
\quad \text{and } \text{cless-eq-set-Complement12 } [\text{set-complement-code}]:
\quad \text{cless-eq-set } (\text{Complement } A) (\text{Complement } B) \leftarrow \rightarrow
\quad (\text{case ID CCOMPARE}'(a) \text{ of None } \Rightarrow \text{Code.abort } (\text{STR } "\text{cless-eq-set Complement: compare } = \text{None}" (\lambda . \text{cless-eq } (\text{Complement } A) (\text{Complement } B)))
\]
\[
| \text{Some } c \Rightarrow \text{cless-eq-set } B A
\]
\[
(\text{is } \text{?Compl-Compl})
\quad \text{cless-eq-set } (\text{RBT-set } rbt) (\text{RBT-set } rbt') \leftarrow \rightarrow
\quad (\text{case ID CCOMPARE}'(a) \text{ of None } \Rightarrow \text{Code.abort } (\text{STR } "\text{cless-eq-set RBT-set RBT-set: compare } = \text{None}" (\lambda . \text{cless-eq-set } (\text{RBT-set } rbt) (\text{RBT-set } rbt'))
\]
\[
| \text{Some } c \Rightarrow \text{ord.lexord-eq-fusion } (\lambda x y. \text{lt-of-comp } c \ y x) \text{ rbt-keys-generator } (\text{RBT-Set2.init } rbt) (\text{RBT-Set2.init } rbt')
\]
\[
(\text{is } \text{?rbt-rbt})
\quad \text{and } \text{cless-eq-set-rbt-Complement2 } [\text{set-complement-code}]:
\quad \text{cless-eq-set } (\text{RBT-set } rbt1) (\text{Complement } (\text{RBT-set } rbt2)) \leftarrow \rightarrow
\quad (\text{case ID CCOMPARE}'(b) \text{ of None } \Rightarrow \text{Code.abort } (\text{STR } "\text{cless-eq-set RBT-set (Complement RBT-set): compare } = \text{None}" (\lambda . \text{cless-eq-set } (\text{RBT-set } rbt1) (\text{Complement } (\text{RBT-set } rbt2))))
\]
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

| Some c ⇒
| finite (UNIV :: 'b set) →
| proper-intrvl.set-less-eq-aux-Compl-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))
(is ?rbt-Compl)
and cless-eq-set-rbt-Complement1 [set-complement-code]:
cless-eq-set (Complement (RBT-set rbt1)) (RBT-set rbt2) ⌋→
(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "cproper-interval: compare = None") (λ-. cless-eq-set (Complement (RBT-set rbt1)) (RBT-set rbt2))
(is ?Compl-rbt)
(proof)

lemma cproper-interval-set-Some-Complement [code]:
fixes rbt1 rbt2 :: 'a :: cproper-interval set-rbt
and A B :: 'a set shows
cproper-interval (Some A) (Some B) ⌋→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cproper-interval: compare = None") (λ-. cproper-interval (Some A) (Some B))
| Some c ⇒
| finite (UNIV :: 'a set) ∧ proper-intrvl.proper-interval-set-aux (lt-of-comp c) cproper-interval (csorted-list-of-set A) (csorted-list-of-set B))
(is ?fin-fin)
and cproper-interval-set-Some-Complement [set-complement-code]:
cproper-interval (Some (Complement A)) (Some (Complement B)) ⌋→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cproper-interval Complement2: compare = None") (λ-. cproper-interval (Some (Complement A)) (Some (Complement B)))
| Some c ⇒
| finite (UNIV :: 'a set) ∧ proper-intrvl.proper-interval-set-Compl-aux (lt-of-comp c) cproper-interval None 0 (csorted-list-of-set A) (csorted-list-of-set B))
(is ?Compl-fin-fin)
and cproper-interval-set-Some-Complement [set-complement-code]:
cproper-interval (Some (Complement A)) (Some (Complement B)) ⌋→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cproper-interval Complement2: compare = None") (λ-. cproper-interval (Some (Complement A)) (Some (Complement B)))
| Some c ⇒
| finite (UNIV :: 'a set) ∧ proper-intrvl.proper-interval-Compl-set-aux (lt-of-comp c) cproper-interval None (csorted-list-of-set A) (csorted-list-of-set B))
(is ?Compl-fin-fin)
and cproper-interval-set-Some-Complement-Some-Complement [set-complement-code]:
cproper-interval (Some (Complement A)) (Some (Complement B)) ⌋→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cproper-interval Complement2: compare = None") (λ-. cproper-interval (Some (Complement A)) (Some (Complement B)))
| Some c ⇒
| finite (UNIV :: 'a set) ∧ proper-intrvl.proper-interval-Compl-set-aux (lt-of-comp c) cproper-interval None (csorted-list-of-set A) (csorted-list-of-set B))
(is ?Compl-fin-fin)
and cproper-interval-set-Some-Complement-Some-Complement-Some-Complement [set-complement-code]:
cproper-interval (Some (Complement A)) (Some (Complement B)) ⌋→
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cproper-interval Complement2: compare = None") (λ-. cproper-interval (Some (Complement A)) (Some (Complement B)))
| Some c ⇒
| finite (UNIV :: 'a set) ∧ proper-intrvl.proper-interval-Compl-set-aux (lt-of-comp c) cproper-interval None (csorted-list-of-set A) (csorted-list-of-set B))
(is ?Compl-fin-fin)
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

\[\text{implement Complement: ccompare = None} \] (\(\lambda\). cproper-interval (Some (Complement A)) (Some (Complement B)))

\[\text{Some -} \Rightarrow \text{cproper-interval (Some B) (Some A)}\]

\[\text{(is ?Compl-Compl)}\]

\[\text{cproper-interval (Some (RBT-set rbt1)) (Some (RBT-set rbt2)) } \mapsto \]

\[\text{(case ID CCOMPARE(\text{\textit{\textquotesingle\textit{a}}}) of None } \Rightarrow \text{Code.abort (STR "cproper-interval RBT-set: ccompare = None") (\(\lambda\). cproper-interval (Some (RBT-set rbt1)) (Some (RBT-set rbt2))}\]  

\[\text{| Some } \Rightarrow \text{cproper-interval (Some B) (Some A)}\]

\[\text{(is ?Rbt-rbt)}\]

\[\text{and cproper-interval-set-Some-rbt-Complement \{set-complement-code\}:} \]

\[\text{cproper-interval (Some (RBT-set rbt1)) (Some (Complement (RBT-set rbt2))) } \mapsto \]

\[\text{(case ID CCOMPARE(\text{\textit{\textquotesingle\textit{a}}}) of None } \Rightarrow \text{Code.abort (STR "cproper-interval RBT-set: ccompare = None") (\(\lambda\). cproper-interval (Some (RBT-set rbt1)) (Some (Complement (RBT-set rbt2))}\]  

\[\text{| Some } \Rightarrow \text{finite (UNIV :: \textit{\textquotesingle\textit{a}} set) } \wedge \text{proper-interval-set-aux-fusion (lt-of-comp c)} \]

\[\text{cproper-interval-set-subset-rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2)}\]

\[\text{(is ?Rbt-Compl-rbt)}\]

\[\text{and cproper-interval-set-Complement-Compl-rbt [set-complement-code]:} \]

\[\text{cproper-interval (Some (Complement (RBT-set rbt1))) (Some (RBT-set rbt2)) } \mapsto \]

\[\text{(case ID CCOMPARE(\text{\textit{\textquotesingle\textit{a}}}) of None } \Rightarrow \text{Code.abort (STR "cproper-interval (Complement RBT-set) RBT-set: ccompare = None") (\(\lambda\). cproper-interval (Some (Complement (RBT-set rbt1))) (Some (RBT-set rbt2))}\]  

\[\text{| Some } \Rightarrow \text{finite (UNIV :: \textit{\textquotesingle\textit{a}} set) } \wedge \text{proper-interval-Compl-set-aux-fusion (lt-of-comp c)} \]

\[\text{cproper-interval-set-Compl-rbt-rbt-keys-generator rbt-keys-generator None 0 (RBT-Set2.init rbt1) (RBT-Set2.init rbt2)}\]

\[\text{(is ?Compl-rbt-rbt)}\]

\[\langle \text{proof} \rangle\]

\textit{context ord begin}

\textit{fun sorted-list-subset :: (\textit{\textquotesingle\textit{a}} \Rightarrow \textit{\textquotesingle\textit{b}} \Rightarrow \textit{bool}) \Rightarrow \textit{\textquotesingle\textit{a}} \textit{list } \Rightarrow \textit{\textquotesingle\textit{a}} \textit{list } \Rightarrow \textit{bool}}

\textit{where}

\[\text{sorted-list-subset eq [] ys = True}\]

\[\text{| sorted-list-subset eq (x \# xs) [] = False}\]

\[\text{| sorted-list-subset eq (x \# xs) (y \# ys) } \mapsto \]

\[\text{(if eq x y then sorted-list-subset eq xs ys}\]

\[\text{else x > y } \wedge \text{sorted-list-subset eq (x \# xs) ys)}\]

\textit{end}
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

context linorder begin

lemma sorted-list-subset-correct:
  \[ \text{sorted } xs \land \text{distinct } xs \land \text{sorted } ys \land \text{distinct } ys \]
  \[ \implies \text{sorted-list-subset } op = xs \iff \text{set } xs \subseteq \text{set } ys \]
⟨proof⟩
end

context ord begin

definition sorted-list-subset-fusion :: (\'a \Rightarrow \'a \Rightarrow \text{bool}) \Rightarrow (\'a, \'s1) \text{ generator } \Rightarrow (\'a, \'s2) \text{ generator } \Rightarrow \text{\'s1 } \Rightarrow \text{\'s2 } \Rightarrow \text{bool}
where
  sorted-list-subset-fusion eq g1 g2 s1 s2 = sorted-list-subset eq (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma sorted-list-subset-fusion-code:
  \[
  \begin{align*}
  &\text{sorted-list-subset-fusion eq g1 g2 s1 s2} = \\
  &\quad \text{if list.has-next g1 s1 then} \\
  &\hspace{1em} \text{let } (x, s1') = \text{list.next g1 s1} \\
  &\hspace{1em} \text{in list.has-next g2 s2 } \land \quad ( \\
  &\hspace{2em} \text{let } (y, s2') = \text{list.next g2 s2} \\
  &\hspace{2em} \text{in if eq x y then sorted-list-subset-fusion eq g1 g2 s1' s2' } \\
  &\hspace{2em} \text{else } y < x \land \text{sorted-list-subset-fusion eq g1 g2 s1 s2} \\
  &\quad \text{else True}
\end{align*}
\]
⟨proof⟩
end

lemmas [code] = ord.sorted-list-subset-fusion-code

Define a new constant for the subset operation because \text{Cardinality} introduces \text{Cardinality.subset}' and rewrites \text{op } \subseteq \text{ to } \text{Cardinality.subset}' based on the sort of the element type.

definition subset-eq :: (\'a set) \Rightarrow (\'a set) \Rightarrow bool
where [simp, code del]: subset-eq = op \subseteq

lemma subseteq-code [code]: op \subseteq = subset-eq
⟨proof⟩

lemma subset'-code [code]: \text{Cardinality.subset}' = subset-eq
⟨proof⟩

lemma subset-eq-code [folded subset-eq-def, code]:
  fixes A1 A2 :: \'a set
  and rbt :: \'b \Rightarrow \text{ccompare set-rbt}
  and rbt1 rbt2 :: \'d :: \{\text{ccompare, ceq}\} \text{set-rbt}
  and dxs :: \'c :: \text{ceq set-dlist}
and xs :: 'c list shows
RBT-set rbt ⊆ B ←→
(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "subset RBT-set1: ccompare = None") (λ-. RBT-set rbt ⊆ B)
| Some e ⇒ list-all-fusion rbt-keys-generator (λx. x ∈ B)
(RBT-Set2.init rbt) (is ?rbt)
DList-set dxs ⊆ C ←→
(case ID CEQ('c) of None ⇒ Code.abort (STR "subset DList-set1: ceq = None")
(λ-. DList-set dxs ⊆ C)
| Some e ⇒ DList-Set.dlist-all (λx. x ∈ C) dxs (is ?dlist)
Set-Monad xs ⊆ C ←→ list-all (λx. x ∈ C) xs (is ?Set-Monad)
and Collect-subset-eq-Complement [folded subset-eq-def, set-complement-code]:
Collect-set P ⊆ Complement A ←→ A ⊆ \{x. ¬ P x\} (is ?Collect-Set-Compl)
and Complement-subset-eq-Complement [folded subset-eq-def, set-complement-code]:
Complement A1 ⊆ Complement A2 ←→ A2 ⊆ A1 (is ?Compl)
and
RBT-set rbt1 ⊆ RBT-set rbt2 ←→
(case ID CCOMPARE('d) of None ⇒ Code.abort (STR "subset RBT-set RBT-set: ccompare = None") (λ-. RBT-set rbt1 ⊆ RBT-set rbt2)
| Some e ⇒
(case ID CEQ('d) of None ⇒ ord.sorted-list-subset-fusion (lt-of-comp c) (λ x y. c x y = Eq) rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2)
| Some eq ⇒ ord.sorted-list-subset-fusion (lt-of-comp c) eq rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))
(is ?rbt-rbt)
(proof)

hide-const (open) subset-eq
hide-fact (open) subset-eq-def

lemma eq-set-code [code]: Cardinality.eq-set = set-eq
(proof)

lemma set-eq-code [code]:
fixes rbt1 rbt2 :: 'b :: {ccompare, ceq} set-rbt shows
set-eq A B ←→ A ⊆ B ∧ B ⊆ A
and set-eq-Complement-Complement [set-complement-code]:
set-eq (Complement A) (Complement B) = set-eq A B
and
set-eq (RBT-set rbt1) (RBT-set rbt2) =
(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "set-eq RBT-set RBT-set: ccompare = None") (λ-. set-eq (RBT-set rbt1) (RBT-set rbt2))
| Some e ⇒
(case ID CEQ('b) of None ⇒ list-all2-fusion (λ x y. c x y = Eq) rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2)
| Some eq ⇒ list-all2-fusion eq rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

(is ?rht-rbt)
⟨proof⟩

lemma Set-project-code [code]:
Set.filter P A = A ∩ Collect-set P
⟨proof⟩

lemma Set-image-code [code]:
fixes dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt shows
image f (Set-Monad xs) = Set-Monad (map f xs)
image f (Collect-set A) = Code.abort (STR "image Collect-set") (λ-. image f (Collect-set A))
and image-Complement-Complement [set-complement-code]:
image f (Complement (Complement B)) = image f B
⟨proof⟩

lemma the-elem-code [code]:
fixes dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt shows
the-elem (Set-Monad [x]) = x
the-elem (DList-set dxs) =
  (case ID CEQ("a") of None ⇒ Code.abort (STR "image DList-set: ceq = None") (λ-. image g (DList-set dxs))
   | Some ⇒ DList-Set.fold (insert o g) dxs {})
the-elem (RBT-set rbt) =
  (case ID CCOMPARE("b") of None ⇒ Code.abort (STR "image RBT-set: ccompare = None") (λ-. image h (RBT-set rbt))
   | Some ⇒ RBT-Set2.fold (insert o h) rbt {})
⟨proof⟩
lemma **Pow-set-conv-fold**:  
\[ \text{Pow} (\text{set } xs \cup A) = \text{fold} (\lambda x. A \cup \text{insert } x \cdot A) \hspace{1mm} xs \]  
\( \text{(proof)} \)

lemma **Pow-code [code]**:  
\[ \text{fixes } dxs :: 'a :: \text{ceq set-dlist} \]  
\[ \text{and } rbt :: 'b :: \text{ccompare set-rbt} \]  
\[ \text{shows} \]  
\[ \text{Pow} (\text{Set-Monad } xs) = \text{fold} (\lambda x A. A \cup \text{insert } x \cdot A) \hspace{1mm} xs \{\} \]  
\( \text{(proof)} \)

lemma **fold-singleton**:  
\[ \text{Finite-Set. fold } f x \{y\} = f \hspace{1mm} y \hspace{1mm} x \]  
\( \text{(proof)} \)

lift-definition **setsum-cfc :: ('a => 'b :: comm-monoid-add) => ('a, 'b) comp-fun-commute**  
is \( \lambda f :: 'a => 'b. \hspace{1mm} \text{plus} \circ f \)  
\( \text{(proof)} \)

lemma **setsum-code [code]**:  
\[ \text{setsum } f A = (\text{if finite } A \text{ then set-fold-cfc (setsum-cfc } f \hspace{1mm} 0 \hspace{1mm} A \hspace{1mm} \text{ else } 0) \]  
\( \text{(proof)} \)

lemma **product-code [code]**:  
\[ \text{fixes } dxs :: 'a :: \text{ceq set-dlist} \]  
\[ \text{and } dys :: 'b :: \text{ceq set-dlist} \]  
\[ \text{and } rbt1 :: 'c :: \text{ccompare set-rbt} \]  
\[ \text{and } rbt2 :: 'd :: \text{ccompare set-rbt} \]  
\[ \text{shows} \]  
\[ \text{Product-Type. product } A \hspace{1mm} B = \text{Collect-set } (\lambda(x, y). x \in A \land y \in B) \]  
\( \text{(is ?Set-Monad)} \)

\[ \text{Product-Type. product } (\text{Set-Monad } xs) \hspace{1mm} (\text{Set-Monad } ys) = \]  
\[ \text{Set-Monad (fold } (\lambda x. \text{fold } (\lambda y \text{ rest. } (x, y) \neq \text{ rest}) \hspace{1mm} ys \hspace{1mm} []) \hspace{1mm} ys \hspace{1mm} []) \]  
\( \text{(is ?Set-Monad)} \)

\[ \text{Product-Type. product } (\text{DLList-set } dxs) \hspace{1mm} B1 = \]  
\[ (\text{case ID CEQ('a of None => Code.abort (STR "product DList-set1: ceq = None") (\lambda. Product-Type. product } (\text{DLList-set } dxs) \hspace{1mm} B1 \hspace{1mm}) \]  
\[ \hspace{1mm} | \hspace{1mm} \text{Some - => DList-Set.fold } (\lambda x. \text{Pair } x \hspace{1mm} B1 \hspace{1mm} \text{ rest}) \hspace{1mm} dxs \hspace{1mm} \{\} \hspace{1mm}) \]  
\( \text{(is ?dlist1)} \)

\[ \text{Product-Type. product } A1 \hspace{1mm} (\text{DLList-set } dys) = \]  
\[ (\text{case ID CEQ('b of None => Code.abort (STR "product DList-set2: ceq = None") (\lambda. Product-Type. product } (\text{DLList-set } dys) \hspace{1mm} A1 \hspace{1mm}) \]  
\( \hspace{1mm} | \hspace{1mm} \text{Some - => DList-Set.fold } (\lambda x. \text{Pair } x \hspace{1mm} A1 \hspace{1mm} \text{ rest}) \hspace{1mm} dys \hspace{1mm} \{\} \hspace{1mm}) \]  
\( \text{(is ?dlist1)} \)
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

None’’ (\(\lambda\). Product-Type.product A1 (DList-set dys))
\(
\text{Some } \Rightarrow \text{DList-Set.fold} (\lambda y \text{ rest. (}\lambda x. (x, y)) ^{'} A1 \cup \text{rest}) \text{ dys}
\)
\{\}
(is ?dlist2)

Product-Type.product (DList-set dxs) (DList-set dys) =
\(\text{(case ID CEQ('a) of None } \Rightarrow \text{Code.abort (STR "product DList-set DList-set: ceq1 } = \text{None’’}) (\lambda\). Product-Type.product (DList-set dxs) (DList-set dys))\)
\| Some - \Rightarrow
\text{case ID CEQ('b) of None } \Rightarrow \text{Code.abort (STR "product DList-set DList-set: ceq2 } = \text{None’’}) (\lambda\). Product-Type.product (DList-set dxs) (DList-set dys))
\| Some - \Rightarrow \text{DList-set (DList-Set.product dxs dys)}\)

Product-Type.product (RBT-set rbt1) B2 =
\(\text{(case ID CCOMPARE('c) of None } \Rightarrow \text{Code.abort (STR "product RBT-set: compare1 } = \text{None’’}) (\lambda\). Product-Type.product (RBT-set rbt1) B2)}\)
\| Some - \Rightarrow \text{RBT-Set2.fold} (\lambda y \text{ rest. Pair x ^} B2 \cup \text{rest}) rbt1 \{\}\n(is ?rbt1)

Product-Type.product A2 (RBT-set rbt2) =
\(\text{(case ID CCOMPARE('d) of None } \Rightarrow \text{Code.abort (STR "product RBT-set: compare2 } = \text{None’’}) (\lambda\). Product-Type.product A2 (RBT-set rbt2)}\)
\| Some - \Rightarrow \text{RBT-Set2.fold} (\lambda y \text{ rest. (}\lambda x. (x, y)) ^{'} A2 \cup \text{rest}) rbt2 \{\}\n(is ?rbt2)

Product-Type.product (RBT-set rbt1) (RBT-set rbt2) =
\(\text{(case ID CCOMPARE('c) of None } \Rightarrow \text{Code.abort (STR "product RBT-set RBT-set: compare1 } = \text{None’’}) (\lambda\). Product-Type.product (RBT-set rbt1) (RBT-set rbt2)}\)
\| Some - \Rightarrow
\text{case ID CCOMPARE('d) of None } \Rightarrow \text{Code.abort (STR "product RBT-set RBT-set: compare2 } = \text{None’’}) (\lambda\). Product-Type.product (RBT-set rbt1) (RBT-set rbt2)}\)
\| Some - \Rightarrow \text{RBT-set (RBT-Set2.product rbt1 rbt2)}\)
(proof)

lemma Id-on-code [code];
fixes A :: 'a :: ceq set
and dxs :: 'a set-dlist
and P :: 'a \Rightarrow \text{bool}
and rbt :: 'b :: ccompare set-rbt shows
Id-on B = (\lambda x. (x, x)) ^ B
and Id-on-Complement [set-complement-code];
Id-on (Complement A) =
\(\text{(case ID CEQ('a) of None } \Rightarrow \text{Code.abort (STR "Id-on Complement: ceq } = \text{None’’}) (\lambda\). Id-on (Complement A)}\)
\| Some eq \Rightarrow \text{Collect-set (}\lambda (x, y). eq x y \land x \notin A))\)
and
Id-on (Collect-set P) =
\begin{align*}
\text{(case ID CEQ('a') of None ⇒ Code.abort (STR "Id-on Collect-set: ceq = None")}
\end{align*}
\begin{align*}
\text{(λ-. Id-on (Collect-set P))}
\end{align*}
\begin{align*}
\text{Id-on (DLIST-set dxs)} =
\end{align*}
\begin{align*}
\text{(case ID CEQ('a') of None ⇒ Code.abort (STR "Id-on DLIST-set: ceq = None")}
\end{align*}
\begin{align*}
\text{(λ-. Id-on (DLIST-set dxs))}
\end{align*}
\begin{align*}
\text{Id-on (RBT-set rbt)} =
\end{align*}
\begin{align*}
\text{(case ID CCOMPARE('b') of None ⇒ Code.abort (STR "Id-on RBT-set: ccompare = None") (λ-. Id-on (RBT-set rbt))}
\end{align*}
\begin{align*}
\text{| Some - ⇒ RBT-set (RBT-Set2.Id-on rbt)}
\end{align*}
\begin{align*}
\langle \text{proof} \rangle
\end{align*}

\text{lemma Image-code [code]:}
\begin{align*}
\text{fixes dxs :: ('a :: ceq × 'b :: ceq) set-dlist}
\end{align*}
\begin{align*}
\text{and rbt :: ('c :: ccompare × 'd :: ccompare) set-rbt shows}
\end{align*}
\begin{align*}
X \backslash Y = \text{snd \; Set.filter (λ(x, y). x ∈ Y) X}
\end{align*}
\begin{align*}
\text{(is ?generic)}
\end{align*}
\begin{align*}
\text{Set-Monad rxs " A = Set-Monad (fold (λ(x, y). rest. if x ∈ A then y # rest else rest) rxs [])}
\end{align*}
\begin{align*}
\text{(is ?Set-Monad)}
\end{align*}
\begin{align*}
\text{DLIST-set dxs " B =}
\end{align*}
\begin{align*}
\text{(case ID CEQ('a') of None ⇒ Code.abort (STR "Image DLIST-set: ceq1 = None")}
\end{align*}
\begin{align*}
\text{(λ-. DLIST-set dxs " B)}
\end{align*}
\begin{align*}
\text{| Some - ⇒}
\end{align*}
\begin{align*}
\text{case ID CEQ('b') of None ⇒ Code.abort (STR "Image DLIST-set: ceq2 = None") (λ-. DLIST-set dxs " B)}
\end{align*}
\begin{align*}
\text{| Some - ⇒}
\end{align*}
\begin{align*}
\text{DLIST-Set.fold (λ(x, y). acc. if x ∈ B then insert y acc else acc) dxs {}}
\end{align*}
\begin{align*}
\text{(is ?DLIST-set)}
\end{align*}
\begin{align*}
\text{RBT-set rbt " C =}
\end{align*}
\begin{align*}
\text{(case ID CCOMPARE('c') of None ⇒ Code.abort (STR "Image RBT-set: ccompare1 = None") (λ-. RBT-set rbt " C)}
\end{align*}
\begin{align*}
\text{| Some - ⇒}
\end{align*}
\begin{align*}
\text{case ID CCOMPARE('d') of None ⇒ Code.abort (STR "Image RBT-set: ccompare2 = None") (λ-. RBT-set rbt " C)}
\end{align*}
\begin{align*}
\text{| Some - ⇒}
\end{align*}
\begin{align*}
\text{RBT-Set2.fold (λ(x, y). acc. if x ∈ C then insert y acc else acc) rbt {}}
\end{align*}
\begin{align*}
\text{(is ?RBT-set)}
\end{align*}
\begin{align*}
\langle \text{proof} \rangle
\end{align*}

\text{lemma insert-relcomp: insert (a, b) A O B = A O B ∪ \{a\} × \{c. (b, c) ∈ B\}}
\begin{align*}
\langle \text{proof} \rangle
\end{align*}

\text{lemma trancl-code [code]:}
\begin{align*}
\text{trancl A =}
\end{align*}
\begin{align*}
\text{(if finite A then ntrancl (card A - 1) A else Code.abort (STR \"transl: infinite set\") (λ-. trancl A))}
\end{align*}
\section{Different Implementations of Sets}

\begin{proof}

\begin{lemma}
\begin{proof}
\end{proof}
\end{lemma}

\begin{lemma}
\end{lemma}

\begin{lemma}
\end{lemma}

\begin{proof}
\end{proof}
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

```plaintext
| Some eq ⇒
\begin{align*}
&\text{case ID CEQ('c) of None ⇒ Code.abort (STR "relcomp RBT-set DList-set: ceq3 = None") (λ- RBT-set rbt3 O DList-set dxs1)} \\
&\text{Some - ⇒ RBT-Set2 fold (λ(x, y). DList-Set.fold (λ(y', z)
A. if eq y y' then insert (x, z) A else A) rbt3 {})} \\
&\text{(is ?rbt-dlist)}
\end{align*}
```

```plaintext
\begin{align*}
\text{DList-set dxs2 O RBT-set rbt4} &= \\
\text{(case ID CEQ('e) of None ⇒ Code.abort (STR "relcomp DList-set RBT-set: ceq1 = None") (λ- DList-set dxs2 O RBT-set rbt4)} & | \text{Some - ⇒}
\text{case ID CCOMPARE('d) of None ⇒ Code.abort (STR "relcomp DList-set RBT-set: ceq2 = None") (λ- DList-set dxs2 O RBT-set rbt4)}
& | \text{Some - ⇒}
\text{case ID CEQ('a) of None ⇒ Code.abort (STR "relcomp DList-set RBT-set: ceq2 = None") (λ- DList-set dxs2 O RBT-set rbt4)}
& | \text{Some eq ⇒}
\text{case ID CCOMPARE('e) of None ⇒ Code.abort (STR "relcomp DList-set DList-set: ceq1 = None") (λ- DList-set dxs3 O DList-set dxs4)}
\text{Some - ⇒}
\text{Some - ⇒}
\text{case ID CEQ('g) of None ⇒ Code.abort (STR "relcomp DList-set DList-set: ceq3 = None") (λ- DList-set dxs3 O DList-set dxs4)}
& | \text{Some - ⇒ DList-Set.fold (λ(x, y). DList-Set.fold (λ(y', z)
A. if eq y y' then insert (x, z) A else A) dxs2 {}}) \\
&\text{(is ?dlist-dlist)}
\end{align*}
```

```plaintext
\begin{align*}
\text{DList-set dxs3 O DList-set dxs4} &= \\
\text{(case ID CEQ('i) of None ⇒ Code.abort (STR "relcomp DList-set DList-set: ceq1 = None") (λ- DList-set dxs3 O DList-set dxs4)}
\text{Some - ⇒}
\text{Some eq ⇒}
& | \text{Some - ⇒ DList-Set.fold (λ(x, y). DList-Set.fold (λ(y', z)
A. if eq y y' then insert (x, z) A else A) dxs3 {}}) \\
&\text{(is ?dlist-dlist)}
\end{align*}
```

```plaintext
\begin{align*}
\text{Set-Monad xs1 O Set-Monad xs2} &= \\
\text{(case ID CEQ('i) of None ⇒ Code.abort (STR "relcomp Set-Monad Set-Monad: ceq1 = None") (λ- Set-Monad xs1 O Set-Monad xs2)}
\text{Some eq ⇒ fold (λ(x, y). fold (λ(y', z) A. if eq y y' then insert (x, z) A else A) xs2 {x})} \\
&\text{(is ?monad-monad)}
\end{align*}
```

```plaintext
\begin{align*}
\text{RBT-set rbt1 O Set-Monad xs3} &= \\
\text{(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "relcomp RBT-set Set-Monad: ccompare1 = None") (λ-. RBT-set rbt1 O Set-Monad xs3)}
\text{Some - ⇒}
\text{case ID CCOMPARE('b) of None ⇒ Code.abort (STR "relcomp RBT-set Set-Monad: ccompare2 = None") (λ-. RBT-set rbt1 O Set-Monad xs3)}
\end{align*}
```
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

\[
\text{Set-Monad } \texttt{xs4} \text{ O } \texttt{RBT-set rbt5} =
\begin{align*}
(\text{case ID } \texttt{CCOMPARE}('u) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp Set-Monad RBT-set: ccompare1 = None"}) (\lambda\text{- Set-Monad } \texttt{xs4} \text{ O } \texttt{RBT-set rbt5}) \\
\text{ | Some } - &\Rightarrow \\
\text{case ID } \texttt{CCOMPARE}('b) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp Set-Monad RBT-set: ccompare2 = None"}) (\lambda\text{- Set-Monad } \texttt{xs4} \text{ O } \texttt{RBT-set rbt5}) \\
\text{ | Some } c-b &\Rightarrow \text{fold} (\lambda(x, y). \text{RBT-Set2.fold} (\lambda(y', z) A. \text{if } c-b \text{ y y' } \neq \text{Eq} \\
\text{then } A \text{ else insert } (x, z) A) \text{ rbt5} \text{ xs4 } \{})) \\
(\text{is } \texttt{?monad-rbt})
\end{align*}
\]

\[
\text{DLList-set } \texttt{dxs3} \text{ O } \texttt{Set-Monad xs5} =
\begin{align*}
(\text{case ID } \texttt{CEQ}('e) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp DLList-set Set-Monad: ceq1 = None"}) (\lambda\text{- DLList-set } \texttt{dxs3} \text{ O } \texttt{Set-Monad xs5}) \\
\text{ | Some } - &\Rightarrow \\
\text{case ID } \texttt{CEQ}('f) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp DLList-set Set-Monad: ceq2 = None"}) (\lambda\text{- DLList-set } \texttt{dxs3} \text{ O } \texttt{Set-Monad xs5}) \\
\text{ | Some eq } &\Rightarrow \text{DLList-Set.fold} (\lambda(x, y). \text{fold} (\lambda(y', z) A. \text{if } eq \text{ y y' then insert } (x, z) A \text{ else A}) \text{ xs5 } \{})))
(\text{is } \texttt{?dlisan-dlist})
\end{align*}
\]

\[
\text{Set-Monad } \texttt{xs6} \text{ O } \texttt{DLList-set dxs4} =
\begin{align*}
(\text{case ID } \texttt{CEQ}('f) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp Set-Monad DLList-set: ceq1 = None"}) (\lambda\text{- Set-Monad } \texttt{xs6} \text{ O } \texttt{DLList-set dxs4}) \\
\text{ | Some eq } &\Rightarrow \\
\text{case ID } \texttt{CEQ}('g) \text{ of None } &\Rightarrow \text{Code.abort (STR "relcomp Set-Monad DLList-set: ceq2 = None"}) (\lambda\text{- Set-Monad } \texttt{xs6} \text{ O } \texttt{DLList-set dxs4}) \\
\text{ | Some - } &\Rightarrow \text{fold} (\lambda(x, y). \text{DLList-Set.fold} (\lambda(y', z) A. \text{if } eq \text{ y y' then insert } (x, z) A \text{ else A}) \text{ dxs4 } \{}))
(\text{is } \texttt{?monad-dlist})
\end{align*}
\]

\[
\text{lemma } \texttt{irrefl-code [code]}:
\begin{align*}
\text{fixes } r &:: ('a :: \texttt{ceq, compare}) \times ('a) \texttt{set shows} \\
\text{irrefl } r &\leftrightarrow \\
(\text{case ID } \texttt{CEQ('a) of Some eq } &\Rightarrow (\forall (x, y) \in r. \neg eq \ x \ y) \text{ | None } \Rightarrow \\
\text{case ID } \texttt{CCOMPARE('a) of None } &\Rightarrow \text{Code.abort (STR "irrefl: ceq = None & compare = None") (\lambda\text{- irrefl } r) \\
\text{ | Some c } &\Rightarrow (\forall (x, y) \in r. \ c \ x \ y \neq \text{Eq})}
\end{align*}
\]

\[
\text{lemma } \texttt{wf-code [code]}:
\begin{align*}
\text{fixes } \texttt{rbt} &:: ('a :: \texttt{compare} \times ('a) \texttt{set-rbt} \\
\text{and } \texttt{dxs} &:: ('b :: \texttt{ceq \times b) set-dlist shows} \\
\text{wf } (\texttt{Set-Monad xs}) &\Rightarrow \text{acyclic (Set-Monad xs)} \\
\text{wf } (\texttt{RBT-set rbt}) &\Rightarrow
\end{align*}
\]
(case ID CCOMPARE\('a\) of None ⇒ Code.abort (STR "wf RBT-set: ccompare = None") \(λ\cdot wf\ (RBT-set rbt))
  \[ Some \to acyclic (RBT-set rbt) \]
\(wf\ (DList-set dxs) =
\)
(case ID CEQ\('b\) of None ⇒ Code.abort (STR "wf DList-set: ceq = None")
\(λ\cdot wf\ (DList-set dxs))
  \[ Some \to acyclic (DList-set dxs) \]

\[ proof \]

\textbf{lemma bacc-code [code]:}
\[
bacc R 0 = \text{snd } ' R
bacc R (Suc n) = (let rec = bacc R n in rec \u2229 \text{snd } ' (Set.filter \(λ(y, x). y \not\in\ rec) R))
\]
\[ proof \]

\textbf{lemma acc-code [code]:}
\[
\text{fixes } A :: \('a :: \{\text{finite, card-UNIV}\} \times \{a\} \\text{set shows}}
\text{Wellfounded.acc } A = \text{bacc } A \text{ (of-phantom (card-UNIV :: 'a card-UNIV))}
\]
\[ proof \]

\textbf{lemma sorted-list-of-set-code [code]:}
\[
\text{fixes } dxs :: \('a :: \{\text{linorder, ceq}\} \\text{set-dlist and}}\ rbt :: \('b :: \{\text{linorder, ccompare}\} \\text{set-rbt shows}}
\text{sorted-list-of-set (Set-Monad xs) = sort (remdups xs)}
\text{sorted-list-of-set (DList-set dxs) =}
\text{\(\text{(case ID CEQ('a) of None ⇒ Code.abort (STR "sorted-list-of-set DList-set: ceq = None") (λ\cdot sorted-list-of-set (DList-set dxs))}
\text{\[ Some \to sort (list-of-dlist dxs) \]}}
\text{sorted-list-of-set (RBT-set rbt) =}
\text{\(\text{(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "sorted-list-of-set RBT-set: ccompare = None") (λ\cdot sorted-list-of-set (RBT-set rbt))}
\text{\[ Some \to sort (RBT-Set2.keys rbt) \]}}
\]

— We must sort the keys because ccompare’s ordering need not coincide with \text{linorder’s.}
\]
\[ proof \]

\textbf{lemma map-project-set: List.map-project f (set xs) = set (List.map-filter f xs)}
\[ proof \]

\textbf{lemma map-project-simps:}
\[
\text{shows map-project-empty: List.map-project f } \{} = \{}
\text{and map-project-insert:}
\text{List.map-project f (insert x A) =}
\text{\(\text{(case f x of None ⇒ List.map-project f A}
\text{\[ Some y ⇒ insert y (List.map-project f A) \]}}
\]
\[ proof \]
lemma map-project-conv-fold:
List.map-project f (set xs) =
fold (λx A. case f x of None ⇒ A | Some y ⇒ insert y A) xs {}
Some singleton-list-fusion (filter-generator R rbt-keys-generator) (RBT-Set2.init rbt)) (is ?rbt)
⟨proof⟩

lemma pred-of-set-code [code]:

fixes dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt shows
pred-of-set (Set-Monad xs) = fold (sup o Predicate.single) xs bot
pred-of-set (DList-set dxs) =
  (case ID CEQ('a) of None ⇒ Code.abort (STR "pred-of-set DList-set: ceq = None") (λ-, pred-of-set (DList-set dxs))
  | Some - ⇒ DList-Set.fold (sup o Predicate.single) dxs bot)
pred-of-set (RBT-set rbt) =
  (case ID CCOMPARE('b) of None ⇒ Code.abort (STR "pred-of-set RBT-set: ccompare = None") (λ-, pred-of-set (RBT-set rbt))
  | Some - ⇒ RBT-Set2.fold (sup o Predicate.single) rbt bot)
⟨proof⟩

'a Predicate.pred is implemented as a monad, so we keep the monad when converting to 'a set. For this case, insert-monad and union-monad avoid the unnecessary dictionary construction.

definition insert-monad :: 'a ⇒ 'a set ⇒ 'a set
where [simp]: insert-monad = insert

definition union-monad :: 'a set ⇒ 'a set ⇒ 'a set
where [simp]: union-monad = op ∪

lemma insert-monad-code [code]:
insert-monad x (Set-Monad xs) = Set-Monad (x # xs)
⟨proof⟩

lemma union-monad-code [code]:
union-monad (Set-Monad xs) (Set-Monad ys) = Set-Monad (xs @ ys)
⟨proof⟩

lemma set-of-pred-code [code]:
set-of-pred (Predicate.Seq f) =
  (case f () of seq.Empty ⇒ Set-Monad []
  | seq.Insert x P ⇒ insert-monad x (set-of-pred P)
  | seq.Join P xq ⇒ union-monad (set-of-pred P) (set-of-seq xq))
⟨proof⟩

lemma set-of-seq-code [code]:
set-of-seq seq.Empty = Set-Monad []
set-of-seq (seq.Insert x P) = insert-monad x (set-of-pred P)
⟨proof⟩
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

hide-const (open) insert-monad union-monad

3.12.5 Type class instantiations

datatype set-impl = Set-IMPL
declare
  set-impl.eq.simps [code del]
  set-impl.size [code del]
  set-impl.rec [code del]
  set-impl.casc [code del]

lemma [code]:
  fixes x :: set-impl
  shows size x = 0
  and size-set-impl x = 0
⟨proof⟩

definition set-Choose :: set-impl where [simp]: set-Choose = Set-IMPL
definition set-Collect :: set-impl where [simp]: set-Collect = Set-IMPL
definition set-DList :: set-impl where [simp]: set-DList = Set-IMPL
definition set-RBT :: set-impl where [simp]: set-RBT = Set-IMPL
definition set-Monad :: set-impl where [simp]: set-Monad = Set-IMPL
code-datatype set-Choose set-Collect set-DList set-RBT set-Monad
definition set-empty-choose :: 'a set where [simp]: set-empty-choose = {}

lemma set-empty-choose-code [code]:
  (set-empty-choose :: 'a :: {ceq, ccompare} set) =
  (case CCOMPARE('a) of Some - ⇒ RBT-set RBT-Set2.empty 
  | None ⇒ case CEQ('a) of None ⇒ Set-Monad [] | Some - ⇒ DList-set
  (DList-Set.empty))
⟨proof⟩

definition set-impl-choose2 :: set-impl ⇒ set-impl ⇒ set-impl
  where [simp]: set-impl-choose2 = (λ- -. Set-IMPL)

lemma set-impl-choose2-code [code]:
  set-impl-choose2 x y = set-Choose
  set-impl-choose2 set-Collect set-Collect = set-Collect
  set-impl-choose2 set-DList set-DList = set-DList
  set-impl-choose2 set-RBT set-RBT = set-RBT
  set-impl-choose2 set-Monad set-Monad = set-Monad
⟨proof⟩

definition set-empty :: set-impl ⇒ 'a set
  where [simp]: set-empty = (λ- -. {})

lemma set-empty-code [code]:
\texttt{set-empty set-Collect} = \texttt{Collect-set (\lambda. False)}
\texttt{set-empty set-DList} = \texttt{DList-set DList-Set.empty}
\texttt{set-empty set-RBT} = \texttt{RBT-set RBT-Set2.empty}
\texttt{set-empty set-Monad} = \texttt{Set-Monad []}
\texttt{set-empty set-Choose} = \texttt{set-empty-choose}

\langle\text{proof}\rangle

class \texttt{set-impl} =
fixes \texttt{set-impl :: ('a, set-impl) phantom}

\textbf{syntax (input)}
\langle\text{ML}\rangle
\texttt{-SET-IMPL :: type => logic ( \(1\text{SET'-IMPL}/(1'('))\))}
\langle\text{ML}\rangle

declare [[\text{code drop: \{}]]

\textbf{lemma empty-code [code, code-unfold]}:
\langle\text{proof}\rangle
\langle\text{ML}\rangle

3.12.6 \textbf{Generator for the set-impl-class}

This generator registers itself at the derive-manager for the classes \texttt{set-impl}.
Here, one can choose the desired implementation via the parameter.

- \textbf{instantiation type :: (type,...,type) (rbt,dlist,collect,monad,choose, or arbitrary constant name) set-impl}

This generator can be used for arbitrary types, not just datatypes.

\langle\text{ML}\rangle

derive (dlist) set-impl unit bool
derive (rbt) set-impl nat
derive (set-RBT) set-impl int
derive (dlist) set-impl Enum.finite-1 Enum.finite-2 Enum.finite-3
derive (rbt) set-impl integer natural
derive (dlist) set-impl nibble
derive (rbt) set-impl char

\textbf{instantiation sum :: (set-impl, set-impl) set-impl begin}
definition \texttt{SET-IMPL('a + 'b) = Phantom('a + 'b)}
\langle\text{proof}\rangle
end

\textbf{instantiation prod :: (set-impl, set-impl) set-impl begin}
definition \texttt{SET-IMPL('a * 'b) = Phantom('a * 'b)}
3.12. DIFFERENT IMPLEMENTATIONS OF SETS

(set-impl-choose2 (of-phantom SET-IMPL('a)) (of-phantom SET-IMPL('b)))
instance ⟨proof⟩
end

derive (choose) set-impl list
derive (rbl) set-impl String.literal

instantiation option :: (set-impl) set-impl begin
definition SET-IMPL('a option) = Phantom('a option) (of-phantom SET-IMPL('a))
instance ⟨proof⟩
end
derive (monad) set-impl fun
derive (choose) set-impl set

instantiation phantom :: (type, set-impl) set-impl begin
definition SET-IMPL(('a, 'b) phantom) = Phantom (('a, 'b) phantom) (of-phantom SET-IMPL('b))
instance ⟨proof⟩
end

We enable automatic implementation selection for sets constructed by set, although they could be directly converted using Set-Monad in constant time. However, then it is more likely that the parameters of binary operators have different implementations, which can lead to less efficient execution. However, we test whether automatic selection picks Set-Monad anyway and take a short-cut.

definition set-aux :: set-impl ⇒ 'a list ⇒ 'a set
where [simp, code del]: set-aux - = set

lemma set-aux-code [code]:
defines conv ≡ foldl (λs (x :: 'a), insert x s)
shows
set-aux impl = conv (set-empty impl) (is ?thesis1)
set-aux set-Choose =
  (case CCOMPARE('a :: {ccompare, ceq}) of Some - ⇒ conv (RBT-set RBT-Set2.empty)
  | None ⇒ case CEQ('a) of None ⇒ Set-Monad
  | Some - ⇒ conv (DList-set DList-Set.empty)) (is ?thesis2)
set-aux set-Monad = Set-Monad
⟨proof⟩

lemma set-code [code]:
fixes xs :: 'a :: set-impl list
shows set xs = set-aux (of-phantom (ID SET-IMPL('a))) xs
⟨proof⟩
3.12.7 Pretty printing for sets

code-post marks contexts (as hypothesis) in which we use code_post as a decision procedure rather than a pretty-printing engine. The intended use is to enable more rules when proving assumptions of rewrite rules.

definition code-post :: bool where code-post = True

lemma conj-code-post [code-post]:
  assumes code-post
  shows True & x ←→ x  False & x ←→ False
  ⟨proof⟩

A flag to switch post-processing of sets on and off. Use declare pretty_sets[code_post del] to disable pretty printing of sets in value.

definition code-post-set :: bool where pretty-sets [code-post, simp]: code-post-set = True

definition collapse-RBT-set :: 'a set-rbt ⇒ 'a :: ccompare set ⇒ 'a set
  where collapse-RBT-set r M = set (RBT-Set2.keys r) ∪ M

lemma RBT-set-collapse-RBT-set [code-post]:
  fixes r :: 'a :: ccompare set-rbt
  assumes code-post ⇒ is-ccompare TYPE('a) and code-post-set
  shows RBT-set r = collapse-RBT-set r {}
  ⟨proof⟩

lemma collapse-RBT-set-Branch [code-post]:
  collapse-RBT-set (Mapping-RBT (Branch c l x v r)) M =
  collapse-RBT-set (Mapping-RBT l) (insert x (collapse-RBT-set (Mapping-RBT r) M))
  ⟨proof⟩

lemma collapse-RBT-set-Empty [code-post]:
  collapse-RBT-set (Mapping-RBT rbt.Empty) M = M
  ⟨proof⟩

definition collapse-DList-set :: 'a :: ceq set-dlist ⇒ 'a set
  where collapse-DList-set dxs = set (DList-Set.list-of-dlist dxs)

lemma DList-set-collapse-DList-set [code-post]:
  fixes dxs :: 'a :: ceq set-dlist
  assumes code-post ⇒ is-ceq TYPE('a) and code-post-set
  ⟨proof⟩

lemma collapse-DList-set-empty [code-post]: collapse-DList-set (Abs-dlist []) = {}
  ⟨proof⟩

lemma collapse-DList-set-Cons [code-post]:
3.13. DIFFERENT IMPLEMENTATIONS OF MAPS

\[ \text{collapse-DList-set} (\text{Abs-dlist} (x \# xs)) = \text{insert} x (\text{collapse-DList-set} (\text{Abs-dlist} xs)) \]

\langle proof \rangle

\text{lemma Set-Monad-code-post} [\text{code-post}]:
\begin{align*}
\text{assumes & code-post-set} \\
\text{shows & Set-Monad} [] = {} \\
\text{and & Set-Monad} (x\#xs) = \text{insert} x (\text{Set-Monad} xs)
\end{align*}

\langle proof \rangle

end

theory Mapping-Impl imports 
RBT-Mapping2 
AssocList 
\sim\sim{/src/HOL/Library/Mapping 
Set-Impl 
Containers-Generator 
begin

3.13. Different implementations of maps

code-identifier 

\text{code-module} Mapping \rightarrow (SML) Mapping-Impl 
| \text{code-module} Mapping-Impl \rightarrow (SML) Mapping-Impl

3.13.1 Map implementations

definition Assoc-List-Mapping :: ('a, 'b) alist \Rightarrow ('a, 'b) mapping 

definition RBT-Mapping :: ('a :: ccompare, 'b) mapping-rbt \Rightarrow ('a, 'b) mapping 
where [simp]: RBT-Mapping t = Mapping.Mapping (RBT-Mapping2.lookup t)

\text{code-datatype} Assoc-List-Mapping RBT-Mapping Mapping

3.13.2 Map operations

\text{declare} [\text{code drop: Mapping.lookup}]

\text{lemma lookup-Mapping-code} [\text{code}]:
\begin{align*}
\text{Mapping.lookup} (\text{Assoc-List-Mapping} al) &= \text{DAList.lookup} al \\
\text{Mapping.lookup} (\text{RBT-Mapping} t) &= \text{RBT-Mapping2.lookup} t
\end{align*}
\langle proof \rangle

\text{declare} [\text{code drop: Mapping.is-empty}]

context
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

begin
interpretation lifting-syntax ⟨proof⟩

lemma is-empty-transfer [transfer-rule]:
(∀ mapping op = op = ===> op =) (λ m = empty) Mapping.is-empty ⟨proof⟩

end

lemma is-empty-Mapping [code]:
fixes t :: ('a :: ccompare, 'b) mapping-rbt shows
Mapping.is-empty (Assoc-List-Mapping al) ◻−→ al = DAList.empty
Mapping.is-empty (RBT-Mapping t) ◻−→
(case ID CCOMPARE ('a) of None ⇒ Code.abort (STR "is-empty RBT-Mapping: ccompare = None") (λ. Mapping.is-empty (RBT-Mapping t))
  | Some - ⇒ RBT-Mapping2.is-empty t) ⟨proof⟩

declare [[code drop: Mapping.update]]

lemma update-Mapping [code]:
fixes t :: ('a :: ccompare, 'b) mapping-rbt shows
Mapping.update k v (Mapping m) = Mapping (m(k := v))
Mapping.update k v (RBT-Mapping t) =
(case ID CCOMPARE ('a) of None ⇒ Code.abort (STR "update RBT-Mapping: ccompare = None") (λ. Mapping.update k v (RBT-Mapping t))
  | Some - ⇒ RBT-Mapping (RBT-Mapping2.insert k v t)) (is ?RBT)
⟨proof⟩

declare [[code drop: Mapping.delete]]

lemma delete-Mapping [code]:
fixes t :: ('a :: ccompare, 'b) mapping-rbt shows
Mapping.delete k (Mapping m) = Mapping (m(k := None))
Mapping.delete k (RBT-Mapping t) =
(case ID CCOMPARE ('a) of None ⇒ Code.abort (STR "delete RBT-Mapping: ccompare = None") (λ. Mapping.delete k (RBT-Mapping t))
  | Some - ⇒ RBT-Mapping (RBT-Mapping2.delete k t)) ⟨proof⟩

declare [[code drop: Mapping.keys]]

theorem rbt-comp-lookup-map-const: rbt-comp-lookup c (RBT-Impl.map (λ. f) t) = map-option f ◦ rbt-comp-lookup c t
3.13. DIFFERENT IMPLEMENTATIONS OF MAPS

⟨proof⟩

lemma keys-Mapping [code];
  fixes t :: (′a :: ccompare, ′b) mapping-rbt shows
  Mapping.keys (Mapping m) = Collect (λk. m k ≠ None) (is ?Mapping)
  Mapping.keys (RBT-Mapping t) = RBT-set (RBT-Mapping2.map (λ- -. ()) t)
  (is ?RBT)
⟨proof⟩

declare [[code drop: Mapping.size]]

context
begin
interpretation lifting-syntax ⟨proof⟩

lemma Mapping-size-transfer [transfer-rule]:
  (pcr-mapping op = op = ===> op =) (card ∘ dom) Mapping.size
⟨proof⟩
end

lemma size-Mapping [code];
  fixes t :: (′a :: ccompare, ′b) mapping-rbt shows
  Mapping.size (Assoc-List-Mapping al) = size al
  Mapping.size (RBT-Mapping t) =
  (case ID CCOMPARE(′a) of None ⇒ Code.abort (STR "size RBT-Mapping:
  ccompare = None") (λ-. Mapping.size (RBT-Mapping t))
  | Some - ⇒ length (RBT-Mapping2.entries t))
⟨proof⟩

declare [[code drop: Mapping.tabulate]]
declare tabulate-fold [code]

datatype mapping-impl = Mapping-IMPL
declare
  mapping-impl.eq.simps [code del]
  mapping-impl.rec [code del]
  mapping-impl.case [code del]

lemma [code]:
  fixes x :: mapping-impl
  shows size x = 0
  and size-mapping-impl x = 0
⟨proof⟩

definition mapping-Choose :: mapping-impl where [simp]: mapping-Choose =
Mapping-IMPL
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

**definition** mapping-Assoc-List :: mapping-impl where [simp]: mapping-Assoc-List = Mapping-IMPL

**definition** mapping-RBT :: mapping-impl where [simp]: mapping-RBT = Mapping-IMPL

**definition** mapping-Mapping :: mapping-impl where [simp]: mapping-Mapping = Mapping-IMPL

**code-datatype** mapping-Choose mapping-Assoc-List mapping-RBT mapping-Mapping

**definition** mapping-empty-choose :: ('a, 'b) mapping where [simp]: mapping-empty-choose = Mapping.empty

**lemma** mapping-empty-choose-code [code]:
(mapping-empty-choose :: ('a :: ccompare, 'b) mapping) =
(case ID CCOMPARE('a) of Some - ⇒ RBT-Mapping RBT-Mapping2.empty | None ⇒ Assoc-List-Mapping DAList.empty)
⟨proof⟩

**definition** mapping-impl-choose2 :: mapping-impl ⇒ mapping-impl ⇒ mapping-impl where [simp]: mapping-impl-choose2 = (λ- -. Mapping-IMPL)

**lemma** mapping-impl-choose2-code [code]:
mapping-impl-choose2 mapping-Mapping mapping-Mapping = mapping-Mapping
mapping-impl-choose2 mapping-Assoc-List mapping-Assoc-List = mapping-Assoc-List
mapping-impl-choose2 mapping-RBT mapping-RBT = mapping-RBT
⟨proof⟩

**definition** mapping-empty :: mapping-impl ⇒ ('a, 'b) mapping where [simp]: mapping-empty = (λ- -. Mapping.empty)

**lemma** mapping-empty-code [code]:
mapping-empty mapping-Choose = mapping-empty-choose
mapping-empty mapping-Mapping = Mapping (λ- -. None)
mapping-empty mapping-Assoc-List = Assoc-List-Mapping DAList.empty
mapping-empty mapping-RBT = RBT-Mapping RBT-Mapping2.empty
⟨proof⟩

### 3.13.3 Type classes

**class** mapping-impl =

**fixes** mapping-impl :: ('a, mapping-impl) phantom

**syntax** (input)

- `MAPPING-IMPL` :: type => logic ( (IMAPPING'-IMPL/(1'("-"))))

⟨ML⟩

**declare** [[code drop: Mapping.empty]]
3.13. DIFFERENT IMPLEMENTATIONS OF MAPS

3.13.1 Lemma

(Mapping-empty-code :: (′a :: mapping-impl, ′b) mapping) =
  mapping-empty (of-phantom MAPPING-IMPL(′a))

⟨proof⟩

3.13.4 Generator for the mapping-impl-class

This generator registers itself at the derive-manager for the classes mapping-impl.
Here, one can choose the desired implementation via the parameter.

- instantiation type :: (type,...,type) (rbt,assoclist,mapping,choose, or arbitrary constant name) mapping-impl

This generator can be used for arbitrary types, not just datatypes.

⟨ML⟩

derive (assoclist) mapping-impl unit bool
derive (rbt) mapping-impl nat
derive (mapping-RBT) mapping-impl int
derive (assoclist) mapping-impl Enum.finite-1 Enum.finite-2 Enum.finite-3
derive (rbt) mapping-impl integer natural
derive (assoclist) mapping-impl nibble
derive (rbt) mapping-impl char

instantiation sum :: (mapping-impl, mapping-impl) mapping-impl begin
definition MAPPING-IMPL(′a + ′b) = Phantom(′a + ′b)
  (mapping-impl-choose2 (of-phantom MAPPING-IMPL(′a)) (of-phantom MAPPING-IMPL(′b)))
instance ⟨proof⟩
end

instantiation prod :: (mapping-impl, mapping-impl) mapping-impl begin
definition MAPPING-IMPL(′a * ′b) = Phantom(′a * ′b)
  (mapping-impl-choose2 (of-phantom MAPPING-IMPL(′a)) (of-phantom MAPPING-IMPL(′b)))
instance ⟨proof⟩
end

derive (choose) mapping-impl list
derive (rbt) mapping-impl String.literal

instantiation option :: (mapping-impl) mapping-impl begin
definition MAPPING-IMPL(′a option) = Phantom(′a option) (of-phantom MAPPING-IMPL(′a))
instance ⟨proof⟩
end

derive (choose) mapping-impl set

instantiation phantom :: (type, mapping-impl) mapping-impl begin
definition MAPPING-IMPL((′a, ′b) phantom) = Phantom ((′a, ′b) phantom)
CHAPTER 3. LIGHT-WEIGHT CONTAINERS

(of-phantom MAPPING-IMPL('b))

instance ⟨proof⟩
end

end

theory Map-To-Mapping imports
Mapping-Impl
begin

3.14 Infrastructure for operation identification

To convert theorems from 'a ⇒ 'b option to ('a, 'b) mapping using lifting / transfer, we first introduce constants for the empty map and map lookup, then apply lifting / transfer, and finally eliminate the non-converted constants again.

Dynamic theorem list of rewrite rules that are applied before Transfer.transferred ⟨ML⟩
Dynamic theorem list of rewrite rules that are applied after Transfer.transferred ⟨ML⟩

context begin interpretation lifting-syntax ⟨proof⟩

definition map-empty :: 'a ⇒ 'b option
where [code-unfold]: map-empty = Map.empty
declare map-empty-def[containers-post, symmetric, containers-pre]
declare Mapping.empty.transfer[transfer-rule del]
lemma map-empty-transfer [transfer-rule]:
  (pcr-mapping A B) map-empty Mapping.empty ⟨proof⟩

definition map-apply :: ('a ⇒ 'b option) ⇒ 'a ⇒ 'b option
where [code-unfold]: map-apply = (λm. m)
lemma eq-map-apply: m x ≡ map-apply m x ⟨proof⟩
declare eq-map-apply[symmetric, abs-def, containers-post]

We cannot use eq-map-apply as a fold rule for operator identification, because it would loop. We use a simp proc instead.
3.14. INFRASTRUCTURE FOR OPERATION IDENTIFICATION

\langle ML \rangle

lemma map-apply-parametric [transfer-rule]:
   \((A \implies B) \implies A \implies B\) map-apply map-apply
\langle proof \rangle

lemma map-apply-transfer [transfer-rule]:
   \(\text{pcr-mapping } A B \implies A \implies \text{rel-option } B\) map-apply Mapping.lookup
\langle proof \rangle

definition map-update :: 'a \Rightarrow 'b option \Rightarrow ('a \Rightarrow 'b option)
where
map-update x y f = f(x := y)

lemma map-update-parametric [transfer-rule]:
   assumes [transfer-rule]: bi-unique A
   shows \((A \implies \text{rel-option } B \implies (A \implies \text{rel-option } B) \implies (A \implies \text{rel-option } B))\) map-update map-update
\langle proof \rangle

context begin
\langle ML \rangle

lift-definition update' :: 'a \Rightarrow 'b option \Rightarrow ('a, 'b) mapping \Rightarrow ('a, 'b) mapping
is map-update parametric map-update-parametric \langle proof \rangle

lemma update'-code [simp, code, code-unfold]:
   update' x None = Mapping.delete x
   update' x (Some y) = Mapping.update x y
\langle proof \rangle

end

declare map-update-def [abs-def, containers-post] map-update-def [symmetric, containers-pre]

definition map-is-empty :: ('a \Rightarrow 'b option) \Rightarrow bool
where
map-is-empty m \iff m = Map.empty

lemma map-is-empty-folds:
   m = map-empty \iff map-is-empty m
   map-empty = m \iff map-is-empty m
\langle proof \rangle

declare map-is-empty-folds [containers-pre]

lemma map-is-empty-def [abs-def, containers-post]
\langle proof \rangle

lemma map-is-empty-transfer [transfer-rule]:
   assumes bi-total A
shows \( (\text{per-mapping} \ A \ B \implies \text{op} =) \ \text{map-is-empty} \ \text{Mapping} \ .\text{is-empty} \)
(proof)
end

⟨ML⟩

hide-const (open) map-apply map-empty map-is-empty map-update 
hide-fact (open) map-apply-def map-empty-def eq-map-apply
end

theory Containers imports
  
  Set-Liorder
  Collection-Order
  Collection-Eq
  Collection-Enum
  Equal
  Mapping-Impl
  Map-To-Mapping

begin

end
Compatibility with Regular-Sets theory Compatibility-Containers-Regular-Sets imports
  Containers
  ../Regular-Sets/Regexp-Method

begin

  Adaptation theory to make \text{regexp} work when Containers are loaded.
  Warning: Each invocation of \text{regexp} takes longer than without Containers
  because the code generator takes longer to generate the evaluation code for \text{regexp}.

datatype-compat \textvariable{rexp}
derive \text{ceq} \textvariable{rexp}
derive \text{ccompare} \textvariable{rexp}
derive (\text{choose}) \text{set-impl} \textvariable{rexp}

notepad begin
(proof)
end

end
Chapter 4

User guide

This user guide shows how to use and extend the lightweight containers framework (LC). For a more theoretical discussion, see [5]. This user guide assumes that you are familiar with refinement in the code generator [1, 2]. The theory Containers-Userguide generates it; so if you want to experiment with the examples, you can find their source code there.

4.1 Characteristics

- **Separate type classes for code generation**
  LC follows the ideal that type classes for code generation should be separate from the standard type classes in Isabelle. LC’s type classes are designed such that every type can become an instance, so well-sortedness errors during code generation can always be remedied.

- **Multiple implementations**
  LC supports multiple simultaneous implementations of the same container type. For example, the following implements at the same time (i) the set of bool as a distinct list of the elements, (ii) int set as a RBT of the elements or as the RBT of the complement, and (iii) sets of functions as monad-style lists:

  \[
  \text{value} \left( \{\text{True}\}, \{1 :: \text{int}\}, \{2 :: \text{int} \land 3\}, \{\lambda x :: \text{int}. x \times x, \lambda y. y + 1\} \right)
  \]

  The LC type classes are the key to simultaneously supporting different implementations.

- **Extensibility**
  The LC framework is designed for being extensible. You can add new containers, implementations and element types any time.
4.2 Getting started

Add the entry theory `Containers` for LC to the end of your imports. This will reconfigure the code generator such that it implements the types ‘a set for sets and (‘a, ‘b) mapping for maps with one of the data structures supported. As with all the theories that adapt the code generator setup, it is important that `Containers` comes at the end of the imports.

Run the following command, e.g., to check that LC works correctly and implements sets of ints as red-black trees (RBT):

```haskell
value {1 :: int}
```

This should produce `{1}`. Without LC, sets are represented as (complements of) a list of elements, i.e., `set [1]` in the example.

If your exported code does not use your own types as elements of sets or maps and you have not declared any code equation for these containers, then your `export-code` command will use LC to implement ‘a set and (‘a, ’b) mapping.

Our running example will be arithmetic expressions. The function `vars e` computes the variables that occur in the expression `e`

```haskell
type-synonym vname = string
datatype expr = Var vname | Lit int | Add expr expr

fun vars :: expr ⇒ vname set where
  vars (Var v) = {v}
  | vars (Lit i) = {}
  | vars (Add e1 e2) = vars e1 ∪ vars e2

value vars (Var "x")
```

To illustrate how to deal with type variables, we will use the following variant where variable names are polymorphic:

```haskell
datatype ‘a expr = Var ‘a | Lit int | Add ‘a expr ‘a expr

fun vars’ :: ‘a expr ⇒ ‘a set where
  vars’ (Var ‘a) = {‘a}
  | vars’ (Lit i) = {}
  | vars’ (Add e1 e2) = vars’ e1 ∪ vars’ e2

value vars’ (Var’ (1 :: int))
```

4.3 New types as elements

This section explains LC’s type classes and shows how to instantiate them. If you want to use your own types as the elements of sets or the keys of maps,
4.3. NEW TYPES AS ELEMENTS

you must instantiate up to eight type classes: ceq (§4.3.1), ccompare (§4.3.2),
set-impl (§4.3.3), mapping-impl (§4.3.3), cenum (§4.3.4), finite-UNIV (§4.3.5),
card-UNIV (§4.3.5), and cproper-interval (§4.3.5). Otherwise, well-sortedness
errors like the following will occur:

*** Wellsortedness error:
*** Type expr not of sort \{ceq,ccompare\}
*** No type arity expr :: ceq
*** At command "value"

In detail, the sort requirements on the element type \('a\) are:

- ceq (§4.3.1), ccompare (§4.3.2), and set-impl (§4.3.3) for \('a\) set in general

- cenum (§4.3.4) for set comprehensions \{x. P x\},

- card-UNIV, cproper-interval for \('a\) set set and any deeper nesting of sets (§4.3.5),

- equal, ccompare (§4.3.2) and mapping-impl (§4.3.3) for (\('a, 'b\) mapping.

4.3.1 Equality testing

The type class ceq defines the operation $CEQ('a) : \ ('a \Rightarrow 'a \Rightarrow \text{bool}) \text{ option}$
for testing whether two elements are equal. The test is embedded in an option value to allow for types that do not support executable equality test such as \('a \Rightarrow 'b\). Whenever possible, $CEQ('a)$ should provide an executable
equality operator. Otherwise, membership tests on such sets will raise an
exception at run-time.

For data types, the derive command can automatically instantiates of ceq,
we only have to tell it whether an equality operation should be provided or
not (parameter no).

---

1These type classes are only required for set complements (see §4.7.2).
2We deviate here from the strict separation of type classes, because it does not make
sense to store types in a map on which we do not have equality, because the most basic
operation Mapping.lookup inherently requires equality.
3Technically, the type class ceq defines the operation ceq. As usage often does not fully
determine ceq's type, we use the notation $CEQ('a)$ that explicitly mentions the type. In
detail, $CEQ('a)$ is translated to $CEQ('a) : \ ('a \Rightarrow 'a \Rightarrow \text{bool}) \text{ option}$ including the type
constraint. We do the same for the other type class operators: ccompare constrains the
operation ccompare (§4.3.2), SET-IMPL('a) constrains the operation set-impl, (§4.3.3),
MAPPING-IMPL('a) (constrains the operation mapping-impl, (§4.3.3), and CENUM('a)
constrains the operation cenum, §4.3.4.
**CHAPTER 4. USER GUIDE**

derive (eq) ceq expr

datatype example = Example
derive (no) ceq example

In the remainder of this subsection, we look at how to manually instantiate a type for `ceq`. First, the simple case of a type constructor `simple-tycon` without parameters that already is an instance of `equal`:

typedec simple-tycon
axiomatization where simple-tycon-equal: OFCLASS(simple-tycon, equal-class)
instance simple-tycon :: equal ⟨proof⟩

instantiation simple-tycon :: ceq begin
definition CEQ(simple-tycon) = Some op =
instance ⟨proof⟩
end

For polymorphic types, this is a bit more involved, as the next example with `′a expr` illustrates (note that we could have delegated all this to `derive`). First, we need an operation that implements equality tests with respect to a given equality operation on the polymorphic type. For data types, we can use the relator which the transfer package (method `transfer`) requires and the BNF package generates automatically. As we have used the old datatype package for `′a expr`, we must define it manually:

context fixes R :: ′a ⇒ ′b ⇒ bool begin
fun expr′-rel :: ′a expr′ ⇒ ′b expr′ ⇒ bool
where
  expr′-rel (Var′ v) (Var′ v′) ←→ R v v′
  | expr′-rel (Lit′ i) (Lit′ i′) ←→ i = i′
  | expr′-rel (Add′ e1 e2) (Add′ e1′ e2′) ←→ expr′-rel e1 e1′ ∧ expr′-rel e2 e2′
  | expr′-rel - - ←→ False
end

If we give HOL equality as parameter, the relator is equality:

lemma expr′-rel-eq: expr′-rel op = e1 e2 ←→ e1 = e2
⟨proof⟩

Then, the instantiation is again canonical:

instantiation expr′ :: (ceq) ceq begin
definition
  CEQ(′a expr′) =
  (case ID CEQ(′a) of None ⇒ None | Some eq ⇒ Some (expr′-rel eq))
instance ⟨proof⟩
4.3. NEW TYPES AS ELEMENTS

Note the following two points: First, the instantiation should avoid to use $op$ = on terms of the polymorphic type. This keeps the LC framework separate from the type class _equal_, i.e., every choice of $'a$ in $'a$ _expr_ can be of sort _ceq_. The easiest way to achieve this is to obtain the equality test from _CEQ_('a_). Second, we use _ID CEQ_('a_) instead of _CEQ_('a_). In proofs, we want that the simplifier uses assumptions like _CEQ_('a_)$ = Some \ldots$ for rewriting. However, _CEQ_('a_) is a nullary constant, so the simplifier reverses such an equation, i.e., it only rewrites _Some \ldots_ to _CEQ_('a_). Applying the identity function _ID_ to _CEQ_('a_) avoids this, and the code generator eliminates all occurrences of _ID_. Although _ID_ = _id_ by definition, do not use the conventional _id_ instead of _ID_, because _id CEQ_('a_) immediately simplifies to _CEQ_('a_).

4.3.2 Ordering

LC takes the order for storing elements in search trees from the type class _ccompare_ rather than _compare_, because we cannot instantiate _compare_ for some types (e.g., $'a$ _set_ as _op_ $\subseteq$ is not linear). Similar to _CEQ_('a_) in class _CEQ_('b_), the class _ccompare_ specifies an optional comparator _CCOMPARE_('a_) :: (('a $\Rightarrow$ 'a $\Rightarrow$ _order_)) _option_. If you cannot or do not want to implement a comparator on your type, you can default to _None_. In that case, you will not be able to use your type as elements of sets or as keys in maps implemented by search trees.

If the type is a data type or instantiates _compare_ and we wish to use that comparator also for the search tree, instantiation is again canonical: For our data type _expr_, derive does everything!

derive ccompare expr

In general, the pattern for type constructors without parameters looks as follows:

axiomatization where simple-tycon-compare: OFCLASS(simple-tycon, compare-class)
instance simple-tycon :: compare ⟨proof⟩

derive (compare) ccompare simple-tycon

For polymorphic types like $'a$ _expr_’, we should not do everything manually: First, we must define a comparator that takes the comparator on the type variable $'a$ as a parameter. This is necessary to maintain the separation between Isabelle/HOL’s type classes (like _compare_) and LC’s. Such a comparator is again easily defined by derive.

derive ccompare expr'

thm ccompare-expr'-def comparator-expr'-simps
4.3.3 Heuristics for picking an implementation

Now, we have defined the necessary operations on $expr$ and $'a expr'$ to store them in a set or use them as the keys in a map. But before we can actually do so, we also have to say which data structure to use. The type classes $set-impl$ and $mapping-impl$ are used for this.

They define the overloaded operations $SET-IMPL('a) :: ('a, set-impl) phantom$ and $MAPPING-IMPL('a) :: ('a, mapping-impl) phantom$, respectively. The phantom type $('a, 'b) phantom$ from theory Phantom-Type is isomorphic to $'b$, but formally depends on $'a$. This way, the type class operations meet the requirement that their type contains exactly one type variable. The Haskell and ML compiler will get rid of the extra type constructor again.

For sets, you can choose between $set-Collect$ (characteristic function $P$ like in $\{x. P x\}$), $set-DList$ (distinct list), $set-RBT$ (red-black tree), and $set-Monad$ (list with duplicates). Additionally, you can define $set-impl$ as $set-Choose$ which picks the implementation based on the available operations (RBT if $compare$ provides a linear order, else distinct lists if $CEQ('a)$ provides equality testing, and lists with duplicates otherwise). $set-Choose$ is the safest choice because it picks only a data structure when the required operations are actually available. If $set-impl$ picks a specific implementation, Isabelle does not ensure that all required operations are indeed available.

For maps, the choices are $mapping-Assoc-List$ (associative list without duplicates), $mapping-RBT$ (red-black tree), and $mapping-Mapping$ (closures with function update). Again, there is also the $mapping-Choose$ heuristics. For simple cases, $derive$ can be used again (even if the type is not a data type). Consider, e.g., the following instantiations: $expr set$ uses RBTs, $(expr, -) mapping$ and $'a expr'$ $set$ use the heuristics, and $(a expr', -) mapping$ uses the same implementation as $(a, -) mapping$.

\begin{verbatim}
derive (rbt) set-impl expr
derive (choose) mapping-impl expr
derive (choose) set-impl expr'
\end{verbatim}

More complex cases such as taking the implementation preference of a type parameter must be done manually.

\begin{verbatim}
instantiation expr' :: (mapping-impl) mapping-impl begin
definition
  MAPPING-IMPL('a expr') =
  Phantom('a expr') (of-phantom MAPPING-IMPL('a))
instance (proof)
end
\end{verbatim}

To see the effect of the different configurations, consider the following ex-
4.3. NEW TYPES AS ELEMENTS

amples where empty refers to Mapping.empty. For that, we must disable pretty printing for sets as follows:

\textbf{declare} pretty-sets [code-post del]

<table>
<thead>
<tr>
<th>value [code]</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>{} :: expr set</td>
<td>RBT-set (Mapping-RBT Empty)</td>
</tr>
<tr>
<td>empty :: (expr, unit) mapping</td>
<td>RBT-Mapping (Mapping-RBT Empty)</td>
</tr>
<tr>
<td>{} :: string expr set</td>
<td>RBT-set (Mapping-RBT Empty)</td>
</tr>
<tr>
<td>{} :: (nat \to\ nat) expr' set</td>
<td>Set-Monad []</td>
</tr>
<tr>
<td>{} :: bool expr' set</td>
<td>RBT-set (Mapping-RBT Empty)</td>
</tr>
<tr>
<td>empty :: (bool expr', unit) mapping</td>
<td>Assoc-List-Mapping (Alist [])</td>
</tr>
</tbody>
</table>

For expr, mapping-Choose picks RBTs, because ccompare provides a comparison operation for expr. For 'a expr', the effect of set-Choose is more pronounced: ccompare is not None, so neither is ccompare, and set-Choose picks RBTs. As \(nat \to nat\) neither provides equality tests (ceq) nor comparisons (ccompare), neither does \(nat \to nat\) expr', so we use lists with duplicates. The last two examples show the difference between inheriting a choice and choosing freshly: By default, bool prefers distinct (associative) lists over RBTs, because there are just two elements. As bool expr' inherits the choice for maps from bool, an associative list implements empty :: (bool expr', unit) mapping. For sets, in contrast, set-impl discards 'a's preferences and picks RBTs, because there is a comparison operation.

Finally, let’s enable pretty-printing for sets again:

\textbf{declare} pretty-sets [code-post]

4.3.4 Set comprehensions

If you use the default code generator setup that comes with Isabelle, set comprehensions \(\{ x. P x \} \) :: 'a set are only executable if the type 'a has sort \textit{enum}. Internally, Isabelle’s code generator transforms set comprehensions into an explicit list of elements which it obtains from the list \textit{enum} of all of 'a’s elements. Thus, the type must be an instance of \textit{enum}, i.e., finite in particular. For example, \(\{ c. CHR "A" \leq c \land c \leq CHR "D" \} \) evaluates to \textit{set} "ABCD", the set of the characters A, B, C, and D.

For compatibility, LC also implements such an enumeration strategy, but avoids the finiteness restriction. The type class \textit{cenum} mimicks \textit{enum}, but its single parameter \textit{cEnum} :: ('a list \times (('a \Rightarrow bool) \Rightarrow bool) \times (('a \Rightarrow bool) \Rightarrow bool)) option combines all of \textit{enum}'s parameters, namely a list of all elements, a universal and an existential quantifier. \textit{option} ensures that every type can be an instance as CENUM('a) can always default to None.
For types that define $CENUM(\tau)$, set comprehensions evaluate to a list of their elements. Otherwise, set comprehensions are represented as a closure. This means that if the generated code contains at least one set comprehension, all element types of a set must instantiate $cenum$. Infinite types default to None, and enumerations for finite types are canonical, see Collection-Enum for examples.

\begin{verbatim}
instantiation expr :: cenum begin
definition CENUM(expr) = None
instance ⟨proof⟩
end

derive (no) cenum expr'
derive compare-order expr
\end{verbatim}

For example, \texttt{value} (\{b. b = True\}, \{x. compare x (Lit 0) = Lt\}) yields (\{True\}, \texttt{Collect-set -})

LC keeps complements of such enumerated set comprehensions, i.e., $-\{b. b = True\}$ evaluates to $Complement \{True\}$. If you want that the complement operation actually computes the elements of the complements, you have to replace the code equations for \texttt{uminus} as follows:

\begin{verbatim}
declare Set-uminus-code[\texttt{code del}] Set-uminus-cenum[\texttt{code}]
\end{verbatim}

Then, $-\{b. b = True\}$ becomes $\{False\}$, but this applies to all complement invocations. For example, UNIV :: bool set becomes $\{False, True\}$.

4.3.5 Nested sets

To deal with nested sets such as \texttt{expr set set}, the element type must provide three operations from three type classes:

- $finite-UNIV$ from theory Cardinality defines the constant $finite-UNIV :: (\tau\', bool) phantom$ which designates whether the type is finite.

- $card-UNIV$ from theory Cardinality defines the constant $card-UNIV :: (\tau\', nat) phantom$ which returns $CARD(\tau)$, i.e., the number of values in $\tau$. If $\tau$ is infinite, $CARD(\tau) = 0$.

- $cproper-interval$ from theory Collection-Order defines the function $cproper-interval :: \tau\' option ⇒ \tau\' option ⇒ bool$. If the type $\tau'$ is finite and $ccompare$ yields a linear order on $\tau'$, then $cproper-interval x y$ returns whether the open interval between $x$ and $y$ is non-empty. The bound $None$ denotes unboundedness.
4.3. NEW TYPES AS ELEMENTS

Note that the type class `finite-UNIV` must not be confused with the type class `finite`. `finite-UNIV` allows the generated code to examine whether a type is finite whereas `finite` requires that the type in fact is finite.

For datatypes, the theory `Card-Datatype` defines some machinery to assist in proving that the type is (in)finite and has a given number of elements – see `Card_Datatype_Ex.thy` for examples. With this, it is easy to instantiate `card-UNIV` for our running examples:

```plaintext
lemma inj-expr [simp]: inj Lit inj Var inj Add inj (Add e)
⟨proof⟩

lemma infinite-UNIV-expr: ¬ finite (UNIV :: expr set)
  including card-datatype
⟨proof⟩

instantiation expr :: card-UNIV begin
definition finite-UNIV = Phantom(expr) False
definition card-UNIV = Phantom(expr) 0
instance ⟨proof⟩
end

lemma inj-expr'[simp]: inj Lit' inj Var' inj Add' inj (Add' e)
⟨proof⟩

lemma infinite-UNIV-expr': ¬ finite (UNIV :: 'a expr' set)
  including card-datatype
⟨proof⟩

instantiation expr' :: (type) card-UNIV begin
definition finite-UNIV = Phantom('a expr') False
definition card-UNIV = Phantom('a expr') 0
instance ⟨proof⟩
end

As `expr` and `expr'` are infinite, instantiating `cproper-interval` is trivial, because `cproper-interval` only makes assumptions about its parameters for finite types. Nevertheless, it is important to actually define `cproper-interval`, because the code generator requires a code equation.

```
Instantiation of \textit{proper-interval}

To illustrate what to do with finite types, we instantiate \textit{proper-interval} for \textit{expr}. Like \textit{ccompare} relates to \textit{compare}, the class \textit{cproper-interval} has a counterpart \textit{proper-interval} without the finiteness assumption. Here, we first have to gather the simplification rules of the comparator from the derive invocation, especially, how the strict order of the comparator, \textit{lt-of-comp}, can be defined.

Since the order on lists is not yet shown to be consistent with the comparators that are used for lists, this part of the userguide is currently not available.

\section{New implementations for containers}

This section explains how to add a new implementation for a container type. If you do so, please consider to add your implementation to this AFP entry.

\subsection{Model and verify the data structure}

First, you of course have to define the data structure and verify that it has the required properties. As our running example, we use a trie to implement \((a, b)\) mapping. A trie is a binary tree whose the nodes store the values, the keys are the paths from the root to the given node. We use lists of \textit{bool}ans for the keys where the \textit{boolean} indicates whether we should go to the left or right child.

For brevity, we skip this step and rather assume that the type \(\forall v\) \textit{trie-raw} of tries has following operations and properties:

\begin{itemize}
  \item \textbf{type-synonym} \textit{trie-key} = \textit{bool} list
  \item \textbf{axiomatization}
    \begin{itemize}
      \item \textit{trie-empty} :: \(\forall v\) \textit{trie-raw} \textbf{and}
      \item \textit{trie-update} :: \textit{trie-key} \Rightarrow \(\forall v\) \textit{trie-raw} \Rightarrow \(\forall v\) \textit{trie-raw} \textbf{and}
      \item \textit{trie-lookup} :: \(\forall v\) \textit{trie-raw} \Rightarrow \textit{trie-key} \Rightarrow \(\forall v\) \textbf{option}
      \item \textit{trie-keys} :: \(\forall v\) \textit{trie-raw} \Rightarrow \textit{trie-key} set
    \end{itemize}
  \end{itemize}

\textbf{where} \textit{trie-lookup-empty}: \textit{trie-lookup} \textit{trie-empty} = \textit{Map.empty}
and trie-lookup-update:
  trie-lookup (trie-update k v t) = (trie-lookup t)(k ↦ v)
and trie-keys-dom-lookup: trie-keys t = dom (trie-lookup t)

This is only a minimal example. A full-fledged implementation has to provide more operations and – for efficiency – should use more than just booleans for the keys.

(proof)(proof)

4.4.2 Generalise the data structure

As (’k, ’v) mapping store keys of arbitrary type ’k, not just trie-key, we cannot use ’v trie-raw directly. Instead, we must first convert arbitrary types ’k into trie-key. Of course, this is not always possible, but we only have to make sure that we pick tries as implementation only if the types do. This is similar to red-black trees which require an order. Hence, we introduce a type class to convert arbitrary keys into trie keys. We make the conversions optional such that every type can instantiate the type class, just as LC does for ceq and ccompare.

type-synonym ’a cbl = (’a ⇒ bool list) × (bool list ⇒ ’a) option
class cbl =
  fixes cbl :: ’a cbl
  assumes inj-to-bl: ID cbl = Some (to-bl, from-bl) ⇒ inj to-bl
  and to-bl-inverse: ID cbl = Some (to-bl, from-bl) ⇒ from-bl (to-bl a) = a
begin
abbreviation from-bl where from-bl ≡ snd (the (ID cbl))
abbreviation to-bl where to-bl ≡ fst (the (ID cbl))
end

It is best to immediately provide the instances for as many types as possible. Here, we only present two examples: unit provides conversion functions, ’a ⇒ ’b does not.

instantiation unit :: cbl begin
definition cbl = Some (λ_. [], λ_. ())
instance ⟨proof⟩
end

instantiation fun :: (type, type) cbl begin
definition cbl = (None :: (’a ⇒ ’b) cbl)
instance ⟨proof⟩
end
4.4.3 Hide the invariants of the data structure

Many data structures have invariants on which the operations rely. You must hide such invariants in a `typedef` before connecting to the container, because the code generator cannot handle explicit invariants. The type must be inhabited even if the types of the elements do not provide the required operations. The easiest way is often to ignore all invariants in that case.

In our example, we require that all keys in the trie represent encoded values.

```
typedef ('k :: cbl, 'v) trie =
  { t :: 'v trie-raw.
    trie-keys t ⊆ range (to-bl :: 'k ⇒ trie-key) ∨ ID (cbl :: 'k cbl) = None }
⟨proof⟩
```

Next, transfer the operations to the new type. The transfer package does a good job here.

```
setup-lifting type-definition-trie — also sets up code generation

lift-definition empty :: ('k :: cbl, 'v) trie
  is trie-empty
⟨proof⟩

lift-definition lookup :: ('k :: cbl, 'v) trie ⇒ 'k ⇒ 'v option
  is λ t. trie-lookup t ◦ to-bl ⟨proof⟩

lift-definition update :: 'k ⇒ 'v ⇒ ('k :: cbl, 'v) trie ⇒ ('k, 'v) trie
  is trie-update ◦ to-bl ⟨proof⟩

lift-definition keys :: ('k :: cbl, 'v) trie ⇒ 'k set
  is λ t. from-bl ' trie-keys t ⟨proof⟩
```

And now we go for the properties. Note that some properties hold only if the type class operations are actually provided, i.e., `cbl ≠ None` in our example.

```
lemma lookup-empty: lookup empty = Map.empty
⟨proof⟩

context
  fixes t :: ('k :: cbl, 'v) trie
  assumes ID-cbl: ID (cbl :: 'k cbl) ≠ None
begin

lemma lookup-update: lookup (update k v t) = (lookup t)(k ↦ v)
⟨proof⟩
```

```
4.4. NEW IMPLEMENTATIONS FOR CONTAINERS

lemma keys-conv-dom-lookup: keys t = dom (lookup t)
(proof)

end

4.4.4 Connecting to the container

Connecting to the container (('a, 'b) mapping in our example) takes three steps:

1. Define a new pseudo-constructor
2. Implement the container operations for the new type
3. Configure the heuristics to automatically pick an implementation
4. Test thoroughly

Thorough testing is particularly important, because Isabelle does not check whether you have implemented all your operations, whether you have configured your heuristics sensibly, nor whether your implementation always terminates.

Define a new pseudo-constructor

Define a function that returns the abstract container view for a data structure value, and declare it as a datatype constructor for code generation with code-datatype. Unfortunately, you have to repeat all existing pseudo-constructors, because there is no way to extract the current set of pseudo-constructors from the code generator. We call them pseudo-constructors, because they do not behave like datatype constructors in the logic. For example, ours are neither injective nor disjoint.

definition Trie-Mapping :: ('a :: cbl, 'v) trie ⇒ ('a, 'v) mapping
where [simp, code del]: Trie-Mapping t = Mapping.Mapping (lookup t)


Implement the operations

Next, you have to prove and declare code equations that implement the container operations for the new implementation. Typically, these just dispatch to the operations on the type from §4.4.3. Some operations depend on the type class operations from §4.4.2 being defined; then, the code equation must check that the operations are indeed defined. If not, there is usually no way
to implement the operation, so the code should raise an exception. Logically, we use the function `Code.abort` of type `String.literal ⇒ (unit ⇒ 'a) ⇒ 'a` with definition `λ- f. f ()`, but the generated code raises an exception `Fail` with the given message (the unit closure avoids non-termination in strict languages). This function gets the exception message and the unit-closure of the equation’s left-hand side as argument, because it is then trivial to prove equality.

Again, we only show a small set of operations; a realistic implementation should cover as many as possible.

```
context fixes t :: ('k :: cbl, 'v) trie begin

lemma lookup-Trie-Mapping [code]:
  Mapping.lookup (Trie-Mapping t) = lookup t
  — Lookup does not need the check on cbl, because we have defined the
  pseudo-constructor `Trie-Mapping` in terms of `lookup

⟨proof⟩

lemma update-Trie-Mapping [code]:
  Mapping.update k v (Trie-Mapping t) =
  (case ID cbl :: 'k cbl of
    None ⇒ Code.abort (STR "update Trie-Mapping: cbl = None") (λ-.
    Mapping.update k v (Trie-Mapping t))
    | Some - ⇒ Trie-Mapping (update k v t))

⟨proof⟩

lemma keys-Trie-Mapping [code]:
  Mapping.keys (Trie-Mapping t) =
  (case ID cbl :: 'k cbl of
    None ⇒ Code.abort (STR "keys Trie-Mapping: cbl = None") (λ-.
    Mapping.keys (Trie-Mapping t))
    | Some - ⇒ keys t)

⟨proof⟩

end
```

These equations do not replace the existing equations for the other constructors, but they do take precedence over them. If there is already a generic implementation for an operation `foo`, say `foo A = gen-foo A`, and you prove a specialised equation `foo (Trie-Mapping t) = trie-foo t`, then when you call `foo` on some `Trie-Mapping t`, your equation will kick in. LC exploits this sequentiality especially for binary operators on sets like `op ∩`, where there are generic implementations and faster specialised ones.
Configure the heuristics

Finally, you should setup the heuristics that automatically picks a container implementation based on the types of the elements (§4.3.3).

The heuristics uses a type with a single value, e.g., `mapping-impl` with value `Mapping-IMPL`, but there is one pseudo-constructor for each container implementation in the generated code. All these pseudo-constructors are the same in the logic, but they are different in the generated code. Hence, the generated code can distinguish them, but we do not have to commit to anything in the logic. This allows to reconfigure and extend the heuristic at any time.

First, define and declare a new pseudo-constructor for the heuristics. Again, be sure to redeclare all previous pseudo-constructors.

```ml
definition mapping-Trie :: mapping-impl
where [simp]: mapping-Trie = Mapping-IMPL
```

Then, adjust the implementation of the automatic choice. For every initial value of the container (such as the empty map or the empty set), there is one new constant (e.g., `mapping-empty-choose` and `set-empty-choose`) equivalent to it. Its code equation, however, checks the available operations from the type classes and picks an appropriate implementation.

For example, the following prefers red-black trees over tries, but tries over associative lists:

```ml
lemma mapping-empty-choose-code [code]:
(mapping-empty-choose :: ('a :: {ccompare, cbl}, 'b) mapping) =
  (case ID CCOMPARE('a) of Some - ⇒ RBT-Mapping RBT-Mapping2.empty
   | None ⇒
     case ID (cbl :: 'a cbl) of Some - ⇒ Trie-Mapping empty
   | None ⇒ Assoc-List-Mapping DAList.empty)
⟨proof⟩
```

There is also a second function for every such initial value that dispatches on the pseudo-constructors for `mapping-impl`. This function is used to pick the right implementation for types that specify a preference.

```ml
lemma mapping-empty-code [code]:
  mapping-empty mapping-Trie = Trie-Mapping empty
⟨proof⟩
```

For (`'k, 'v) mapping, LC also has a function `mapping-impl-choose2` which is given two preferences and returns one (for `'a set, it is called `set-impl-choose2`). Polymorphic type constructors like `'a + `'b use it to pick an implementation
based on the preferences of \( a \) and \( b \). By default, it returns \( \text{mapping-Choose} \), i.e., ignore the preferences. You should add a code equation like the following that overrides this choice if both preferences are your new data structure:

\[
\text{lemma mapping-impl-choose2-Trie [code]:} \\
\text{mapping-impl-choose2 mapping-Trie mapping-Trie = mapping-Trie}
\]

\( \langle \text{proof} \rangle \)

If your new data structure is better than the existing ones for some element type, you should reconfigure the type’s preference. As all preferences are logically equal, you can prove (and declare) the appropriate code equation. For example, the following prefers tries for keys of type \( \text{unit} \):

\[
\text{lemma mapping-impl-unit-Trie [code]:} \\
\text{MAPPING-IMPL(unit) = Phantom(unit) mapping-Trie}
\]

\( \langle \text{proof} \rangle \)

\text{value Mapping.empty :: (unit, int) mapping}

You can also use your new pseudo-constructor with \( \text{derive} \) in instantiations, just give its name as option:

\text{derive (mapping-Trie) mapping-impl simple-tycon}

\section*{4.5 Changing the configuration}

As containers are connected to data structures only by refinement in the code generator, this can always be adapted later on. You can add new data structures as explained in §4.4. If you want to drop one, you redeclare the remaining pseudo-constructors with \( \text{code-datatype} \) and delete all code equations that pattern-match on the obsolete pseudo-constructors. The command \text{code-thms} will tell you which constants have such code equations. You can also freely adapt the heuristics for picking implementations as described in §4.4.4.

One thing, however, you cannot change afterwards, namely the decision whether an element type supports an operation and if so how it does, because this decision is visible in the logic.

\section*{4.6 New containers types}

We hope that the above explanations and the examples with sets and maps suffice to show what you need to do if you add a new container type, e.g., priority queues. There are three steps:

1. \textbf{Introduce a type constructor for the container.}

Your new container type must not be a composite type, like \( a \Rightarrow b \).
option for maps, because refinement for code generation only works with a single type constructor. Neither should you reuse a type constructor that is used already in other contexts, e.g., do not use 'a list to model queues.

Introduce a new type constructor if necessary (e.g., ('a, 'b) mapping for maps) – if your container type already has its own type constructor, everything is fine.

2. Implement the data structures and connect them to the container type as described in §4.4.


4.7 Troubleshooting

This section describes some difficulties in using LC that we have come across, provides some background for them, and discusses how to overcome them. If you experience other difficulties, please contact the author.

4.7.1 Nesting of mappings

Mappings can be arbitrarily nested on the value side, e.g., ('a, ('b, 'c) mapping) mapping. However, ('a, 'b) mapping cannot currently be the key of a mapping, i.e., code generation fails for (('a, 'b) mapping, 'c) mapping. Similarly, you cannot have a set of mappings like ('a, 'b) mapping set at the moment. There are no issues to make this work, we have just not seen the need for it. If you need to generate code for such types, please get in touch with the author.

4.7.2 Wellsortedness errors

LC uses its own hierarchy of type classes which is distinct from Isabelle/HOL’s. This ensures that every type can be made an instance of LC’s type classes. Consequently, you must instantiate these classes for your own types. The following lists where you can find information about the classes and examples how to instantiate them:
The type classes `card-UNIV` and `cproper-interval` are only required to implement the operations on set complements. If your code does not need complements, you can manually delete the code equations involving `Complement`, the theorem list `set-complement-code` collects them. It is also recommended that you remove the pseudo-constructor `Complement` from the code generator. Note that some set operations like \( A - B \) and \( \text{UNIV} \) have no code equations any more.

**declare** `set-complement-code[code del]`

**code-datatype** `Collect-set DList-set RBT-set Set-Monad`

### 4.7.3 Exception raised at run-time

Not all combinations of data and container implementation are possible. For example, you cannot implement a set of functions with a RBT, because there is no order on \( 'a \Rightarrow 'b \). If you try, the code will raise an exception `Fail` (with an exception message) or `Match`. They can occur in three cases:

1. You have misconfigured the heuristics that picks implementations (§4.3.3), or you have manually picked an implementation that requires an operation that the element type does not provide. Printing a stack trace for the exception may help you in locating the error.

2. You are trying to invoke an operation on a set complement which cannot be implemented on a complement representation, e.g., `op '`. If the element type is enumerable, provide an instance of `cenum` and choose to represent complements of sets of enumerable types by the elements rather than the elements of the complement (see §4.3.4 for how to do this).

3. You use set comprehensions on types which do not provide an enumeration (i.e., they are represented as closures) or you chose to represent a map as a closure.

A lot of operations are not implementable for closures, in particular those that return some element of the container.
4.7. TROUBLESHOOTING

Inspect the code equations with code-thms and look for calls to Collect-set and Mapping which are LC’s constructor for sets and maps as closures.

Note that the code generator preprocesses set comprehensions like \{i < 4 \mid i. 2 < i\} to (\lambda i. i < 4) ‘i. 2 < i\}, so this is a set comprehension over int rather than bool.

⟨ML⟩

4.7.4 LC slows down my code

Normally, this will not happen, because LC’s data structures are more efficient than Isabelle’s list-based implementations. However, in some rare cases, you can experience a slowdown:

1. Your containers contain just a few elements.
   In that case, the overhead of the heuristics to pick an implementation outweighs the benefits of efficient implementations. You should identify the tiny containers and disable the heuristics locally. You do so by replacing the initial value like {} and Mapping.empty with low-overhead constructors like Set-Monad and Mapping. For example, if tiny-set-code: tiny-set = \{1, 2\} is your code equation with a tiny set, the following changes the code equation to directly use the list-based representation, i.e., disables the heuristics:

   lemma empty-Set-Monad: {} = Set-Monad [] ⟨proof⟩
   declare tiny-set-code[code del, unfolded empty-Set-Monad, code]

   If you want to globally disable the heuristics, you can also declare an equation like empty-Set-Monad as [code].

2. The element type contains many type constructors and some type variables.
   LC heavily relies on type classes, and type classes are implemented as dictionaries if the compiler cannot statically resolve them, i.e., if there are type variables. For type constructors with type variables (like ‘a × ‘b), LC’s definitions of the type class parameters recursively calls itself on the type variables, i.e., ‘a and ‘b. If the element type is polymorphic, the compiler cannot precompute these recursive calls and therefore they have to be constructed repeatedly at run time. If you wrap your complicated type in a new type constructor, you can define optimised equations for the type class parameters.
Bibliography


