The Myhill-Nerode Theorem
Based on Regular Expressions

Chunhan Wu, Xingyuan Zhang and Christian Urban

May 28, 2015

Abstract

There are many proofs of the Myhill-Nerode theorem using automata. In this library we give a proof entirely based on regular expressions, since regularity of languages can be conveniently defined using regular expressions (it is more painful in HOL to define regularity in terms of automata). We prove the first direction of the Myhill-Nerode theorem by solving equational systems that involve regular expressions. For the second direction we give two proofs: one using tagging-functions and another using partial derivatives. We also establish various closure properties of regular languages. 1

Contents

1 “Summation” for regular expressions 3

2 First direction of MN: finite partition ⇒ regular language 4
   2.1 Equational systems ................................. 5
   2.2 Arden Operation on equations ...................... 5
   2.3 Substitution Operation on equations .............. 6
   2.4 While-combinator and invariants ................. 6
   2.5 Initial Equational Systems ....................... 8
   2.6 Interations ........................................ 10
   2.7 The conclusion of the first direction ............. 17

3 Second direction of MN: regular language ⇒ finite partition 19
   3.1 Tagging functions ................................. 19
   3.2 Base cases: Zero, One and Atom .................. 21
   3.3 Case for Plus ..................................... 23
   3.4 Case for Times ................................... 23
   3.5 Case for Star .................................... 25
   3.6 The conclusion of the second direction .......... 29

1Most details of the theories are described in the paper [2].
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>The theorem</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Second direction proved using partial derivatives</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Closure properties of regular languages</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Closure under $\cup$, $\cdot$ and $\star$</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Closure under complementation</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Closure under $-$ and $\cap$</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>Closure under string reversal</td>
<td>31</td>
</tr>
<tr>
<td>5.5</td>
<td>Closure under left-quotients</td>
<td>33</td>
</tr>
<tr>
<td>5.6</td>
<td>Finite and co-finite sets are regular</td>
<td>33</td>
</tr>
<tr>
<td>5.7</td>
<td>Continuation lemma for showing non-regularity of languages</td>
<td>34</td>
</tr>
<tr>
<td>5.8</td>
<td>The language $a^n b^n$ is not regular</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>Normalizing Derivative</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Normalizing operations</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>Deciding Regular Expression Equivalence</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Bisimulation between languages and regular expressions</td>
<td>38</td>
</tr>
<tr>
<td>7.2</td>
<td>Closure computation</td>
<td>39</td>
</tr>
<tr>
<td>7.3</td>
<td>Bisimulation-free proof of closure computation</td>
<td>40</td>
</tr>
<tr>
<td>7.4</td>
<td>The overall procedure</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Regular Expressions as Homogeneous Binary Relations</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>Proving Relation (In)equalities via Regular Expressions</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Infinite Sequences</td>
<td>44</td>
</tr>
<tr>
<td>10.1</td>
<td>Operations on Infinite Sequences</td>
<td>45</td>
</tr>
<tr>
<td>10.2</td>
<td>Predicates on Natural Numbers</td>
<td>47</td>
</tr>
<tr>
<td>10.3</td>
<td>Assembling Infinite Words from Finite Words</td>
<td>49</td>
</tr>
<tr>
<td>11</td>
<td>Enumerations of Well-Ordered Sets in Increasing Order</td>
<td>56</td>
</tr>
<tr>
<td>12</td>
<td>Binary Predicates Restricted to Elements of a Given Set</td>
<td>57</td>
</tr>
<tr>
<td>12.1</td>
<td>Measures on Sets (Instead of Full Types)</td>
<td>64</td>
</tr>
<tr>
<td>12.2</td>
<td>Facts About Predecessor Sets</td>
<td>70</td>
</tr>
<tr>
<td>13</td>
<td>Constructing Minimal Bad Sequences</td>
<td>71</td>
</tr>
<tr>
<td>14</td>
<td>Almost-Full Relations</td>
<td>76</td>
</tr>
<tr>
<td>14.1</td>
<td>Basic Definitions and Facts</td>
<td>77</td>
</tr>
<tr>
<td>14.2</td>
<td>Adding a Bottom Element to a Set</td>
<td>80</td>
</tr>
<tr>
<td>14.3</td>
<td>Adding a Bottom Element to an Almost-Full Set</td>
<td>81</td>
</tr>
<tr>
<td>14.4</td>
<td>Disjoint Union of Almost-Full Sets</td>
<td>82</td>
</tr>
<tr>
<td>14.5</td>
<td>Dickson’s Lemma for Almost-Full Relations</td>
<td>83</td>
</tr>
</tbody>
</table>
1 “Summation” for regular expressions

To obtain equational system out of finite set of equivalence classes, a fold operation on finite sets \( \text{folds} \) is defined. The use of \( \text{SOME} \) makes \( \text{folds} \) more robust than the \( \text{fold} \) in the Isabelle library. The expression \( \text{folds} \) \( f \) makes sense when \( f \) is not associative and commutitive, while \( \text{fold} \) \( f \) does not.

**definition**

\[
\text{folds} :: (\text{\'}a \Rightarrow \text{\'}b \Rightarrow \text{\'}b) \Rightarrow \text{\'}a \text{ set} \Rightarrow \text{\'}b
\]

**where**

\[
\text{folds} \ z \ S \equiv \text{SOME} \ x. \ \text{fold-graph} \ f \ z \ S \ x
\]

Plus-combination for a set of regular expressions

**abbreviation**

\[
\text{Setalt} :: \text{\'}a \text{ rexp set} \Rightarrow \text{\'}a \text{ rexp} (\bigcup [1000] 999)
\]

**where**

\[
\bigcup A \equiv \text{folds Plus Zero} \ A
\]

For finite sets, \( \text{Setalt} \) is preserved under \( \text{lang} \).

**lemma** \( \text{folds-plus-simp} \) [simp]:

**fixes** \( rs :: (\text{\'}a \text{ rexp} \text{ set}) \)

**assumes** \( a :: \text{finite} \text{ rs} \)

**shows** \( \text{lang} (\bigcup rs) = \bigcup (\text{lang} \cdot rs) \)

**unfolding** \( \text{folds-def} \)

**apply** (rule set-eqI)
apply (rule someI2-ex)
apply (rule-tac finite-imp-fold-graph [OF a])
apply (erule fold-graph.induct)
apply (auto)
done

theory Myhill-1
imports Folds
  ~/src/HOL/Library/While-Combinator
begin

2 First direction of MN: finite partition $\Rightarrow$ regular language

notation
conc (infixr 100) and
star (-• 101 102)

lemma Pair-Collect [simp]:
  shows $(x, y) \in \{(x, y). P x y\} \iff P x y$
by simp

Myhill-Nerode relation

definition
str-eq :: `'a lang $\Rightarrow$ ('a list $\times$ 'a list) set ($\approx$ 100 100)
where
$\approx A \equiv \{(x, y). (\forall z. x \odot z \in A \iff y \odot z \in A)\}$

abbreviation
str-eq-applied :: 'a list $\Rightarrow$ 'a lang $\Rightarrow$ 'a list $\Rightarrow$ bool ($\approx$ - -)
where
$x \approx A y \equiv (x, y) \in \approx A$

definition
finals :: 'a lang $\Rightarrow$ 'a lang set
where
finals A $\equiv \{\approx A {""} \{s\} \mid s . s \in A\}$

lemma lang-is-union-of-finals:
  shows $A = \bigcup$ finals A
unfolding finals-def
unfolding Image-def
unfolding str-eq-def
by (auto) (metis append-Nil2)

lemma finals-in-partitions:
shows finals $A \subseteq (UNIV // \approx A)$

unfolding finals-def quotient-def
by auto

2.1 Equational systems
The two kinds of terms in the rhs of equations.

datatype 'a trm =
    Lam 'a rexp
  | Trn 'a lang 'a rexp

fun
    lang-trm::'a trm ⇒ 'a lang
where
    lang-trm (Lam r) = lang r
  | lang-trm (Trn X r) = X · lang r

fun
    lang-rhs::('a trm) set ⇒ 'a lang
where
    lang-rhs rhs = \bigcup (lang-trm ' rhs)

lemma lang-rhs-set:
    shows lang-rhs {Trn X r | r. P r} = \bigcup {lang-trm (Trn X r) | r. P r}
by (auto)

lemma lang-rhs-union-distrib:
    shows lang-rhs A ∪ lang-rhs B = lang-rhs (A ∪ B)
by simp

Transitions between equivalence classes

definition
transition :: 'a lang ⇒ 'a ⇒ 'a lang ⇒ bool (- |-:- [100,100,100] 100)
where
    Y |-c⇒ X ≡ Y · {c} ⊆ X

    Initial equational system

definition
    Init-rhs CS X ≡
        if ([] ∈ X) then
            \{Lam One\} ∪ \{Trn Y (Atom c) | Y c. Y ∈ CS ∧ Y |-c⇒ X\}
        else
            \{Trn Y (Atom c)| Y c. Y ∈ CS ∧ Y |-c⇒ X\}

definition
    Init CS ≡ \{(X, Init-rhs CS X) | X. X ∈ CS\}

2.2 Arden Operation on equations

fun
Append-rexp :: 'a rexp ⇒ 'a trm ⇒ 'a trm
where
Append-rexp r (Lam rexp) = Lam (Times rexp r)
| Append-rexp r (Trn X rexp) = Trn X (Times rexp r)

definition
Append-rexp-rhs rhs rexp ≡ (Append-rexp rexp) · rhs

definition
Arden X rhs ≡
Append-rexp-rhs (rhs − {Trn X r | r. Trn X r ∈ rhs}) (Star (U {r. Trn X r ∈ rhs}))

2.3 Substitution Operation on equations

definition
Subst rhs X xrhs ≡
(rhs − {Trn X r | r. Trn X r ∈ rhs}) ∪ (Append-rexp-rhs xrhs (U {r. Trn X r ∈ rhs}))

definition
Subst-all :: ('a lang × ('a trm) set) set ⇒ ('a lang ⇒ ('a trm) set) set
where
Subst-all ES X xrhs ≡
{(Y, Subst yrhs X xrhs) | Y yrhs. (Y, yrhs) ∈ ES}

definition
Remove ES X xrhs ≡
Subst-all (ES − {(X, xrhs)}) X (Arden X xrhs)

2.4 While-combinator and invariants

definition
Iter X ES ≡ (let (Y, yrhs) = SOME (Y, yrhs). (Y, yrhs) ∈ ES ∧ X ≠ Y in Remove ES Y yrhs)

lemma IterI2:
  assumes (Y, yrhs) ∈ ES
  and X ≠ Y
  and Y yrhs. [(Y, yrhs) ∈ ES; X ≠ Y] ⇒ Q (Remove ES Y yrhs)
  shows Q (Iter X ES)
unfolding Iter-def using assms
by (rule-tac a=(Y, yrhs) in someI2) (auto)

abbreviation
Cond ES ≡ card ES ≠ 1

definition
Solve X ES ≡ while Cond (Iter X) ES
definition

distinctness \( ES \equiv \forall X \text{ rhs} \text{ rhs}' \). \((X, \text{ rhs}) \in ES \land (X, \text{ rhs}') \in ES \rightarrow \text{ rhs} = \text{ rhs}'\)

definition

soundness \( ES \equiv \forall (X, \text{ rhs}) \in ES. X = \text{ lang-rhs} \text{ rhs} \)

definition

ardenable \( \text{ rhs} \equiv (\forall Y r. \text{ Trn} Y r \in \text{ rhs} \rightarrow [] \notin \text{ lang} r) \)

definition

ardenable-all \( ES \equiv \forall (X, \text{ rhs}) \in ES. \text{ ardenable} \text{ rhs} \)

definition

finite-rhs \( ES \equiv \forall (X, \text{ rhs}) \in ES. \text{ finite} \text{ rhs} \)

lemma finite-rhs-def2:

finite-rhs \( ES = (\forall X \text{ rhs}. (X, \text{ rhs}) \in ES \rightarrow \text{ finite} \text{ rhs}) \)

unfolding finite-rhs-def by auto

definition

\( \text{ rhss} \text{ rhs} \equiv \{X \mid X r. \text{ Trn} X r \in \text{ rhs}\} \)

definition

\( \text{ lhss} \text{ ES} \equiv \{Y \mid Y \text{ yrhs}. (Y, \text{ yrhs}) \in ES\} \)

definition

validity \( ES \equiv \forall (X, \text{ rhs}) \in ES. \text{ rhss} \text{ rhs} \subseteq \text{ lhss} \text{ ES} \)

lemma rhss-union-distrib:

shows \( \text{ rhss} (A \cup B) = \text{ rhss} A \cup \text{ rhss} B \)

by (auto simp add: rhss-def)

lemma lhss-union-distrib:

shows \( \text{ lhss} (A \cup B) = \text{ lhss} A \cup \text{ lhss} B \)

by (auto simp add: lhss-def)

definition

\( \text{ invariant} \text{ ES} \equiv \text{ finite} \text{ ES} \land \text{ finite-rhs} \text{ ES} \land \text{ soundness} \text{ ES} \land \text{ distinctness} \text{ ES} \land \text{ ardenable-all} \text{ ES} \land \text{ validity} \text{ ES} \)
lemma invariantI:
  assumes soundness ES finite ES distinctness ES ardenable-all ES
  finite-rhs ES validity ES
  shows invariant ES
using assms by (simp add: invariant-def)

declare [[simproc add: finite-Collect]]

lemma finite-Trn:
  assumes fin: finite rhs
  shows finite \{ r. Trn Y r ∈ rhs \}
using assms by (auto intro!: finite-vimageI simp add: inj-on-def)

lemma finite-Lam:
  assumes fin: finite rhs
  shows finite \{ r. Lam r ∈ rhs \}
using assms by (auto intro!: finite-vimageI simp add: inj-on-def)

lemma trm-soundness:
  assumes finite: finite rhs
  shows lang-rhs (\{ Trn X r | r. Trn X r ∈ rhs \}) = X · (lang (\bigcup \{ r. Trn X r ∈ rhs \}))
proof –
  have finite \{ r. Trn X r ∈ rhs \}
    by (rule finite-Trn[OF finite])
  then show lang-rhs (\{ Trn X r | r. Trn X r ∈ rhs \}) = X · (lang (\bigcup \{ r. Trn X r ∈ rhs \}))
    by (simp only: lang-rhs-set lang-trm.simps) (auto simp add: conc-def)
qed

lemma lang-of-append-rexp:
  lang-trm (Append-rexp r trm) = lang-trm trm · lang r
by (induct rule: Append-rexp.induct) (auto simp add: conc-assoc)

lemma lang-of-append-rexp-rhs:
  lang-rhs (Append-rexp-rhs rhs r) = lang-rhs rhs · lang r
unfolding Append-rexp-rhs-def
by (auto simp add: conc-def lang-of-append-rexp)

2.5 Initial Equational Systems

lemma defined-by-str:
  assumes s ∈ X X ∈ UNIV // ≈A
  shows X = ≈A " " \{s\}
using assms
unfolding quotient-def Image-def str-eq-def
by auto
lemma \textit{every-eqclass-has-transition}: assumes has-str: \( s \circ [c] \in X \) and in-CS: \( X \in \text{UNIV} /\!
\text{\textasciitilde}A \) obtains \( Y \) where \( Y \in \text{UNIV} /\!
\text{\textasciitilde}A \) and \( Y \cdot \{[c]\} \subseteq X \) and \( s \in Y \) proof –
def \( Y \equiv \text{\textasciitilde}A \ '' \{s\} \)
have \( Y \in \text{UNIV} /\!
\text{\textasciitilde}A \)
unfolding \( Y\)-def quotient-def by auto
moreover
have \( X = \text{\textasciitilde}A \ '' \{s \circ [c]\} \)
using has-str in-CS defined-by-str by blast
then have \( Y \cdot \{[c]\} \subseteq X \)
unfolding \( Y\)-def Image-def conc-def
by clarsimp
moreover
have \( s \in Y \)
  unfolding \( Y\)-def
unfolding Image-def str-eq-def by simp
ultimately show \( \text{thesis} \) using that by blast
qed

lemma \textit{l-eq-r-in-eqs}: assumes X-in-eqs: \((X, \text{rhs})\) \(\in\) \(\text{Init} \ (\text{UNIV} /\!
\text{\textasciitilde}A)\)
shows \( X = \text{lang-rhs} \text{\ rhs} \) proof
show \( X \subseteq \text{lang-rhs} \text{\ rhs} \) proof
fix \( x \)
assume in-X: \( x \in X \)
\{ assume empty: \( x = [] \)
then have \( x \in \text{lang-rhs} \text{\ rhs} \) using X-in-eqs in-X
  unfolding Init-def Init-rhs-def
by auto
\}
moreover
\{ assume not-empty: \( x \neq [] \)
then obtain \( s\ c \) where decom: \( x = s \circ [c]\)
  using rev-cases by blast
have \( X \in \text{UNIV} /\!
\text{\textasciitilde}A \) using X-in-eqs unfolding Init-def by auto
then obtain \( Y \) where \( Y \in \text{UNIV} /\!
\text{\textasciitilde}A \) \( Y \cdot \{[c]\} \subseteq X \) \( s \in Y \)
  using decom in-X every-eqclass-has-transition by metis
then have \( x \in \text{lang-rhs} \) \(\{\text{Trn} \ Y \ (\text{Atom} \ c)| \ Y\ c \ \in \text{\UNIV} /\!
\text{\textasciitilde}A \ \land \ Y \}\) \( s \Rightarrow X \)
unfolding transition-def
using decom by (force simp add: conc-def)
then have \( x \in \text{lang-rhs} \) using X-in-eqs in-X
  unfolding Init-def Init-rhs-def by simp
\}

9
ultimately show $x \in \text{lang-rhs \ rhs}$ by blast
qed

next

show $\text{lang-rhs \ rhs} \subseteq X$ using $X\text{-in-eqs}$
  unfolding \text{Init-def Init-rhs-def transition-def}
  by auto
qed

lemma \text{finite-Init-rhs}:
  fixes $CS :: (\langle a::\text{finite} \rangle \ \text{lang})$ set
  assumes finite: finite $CS$
  shows finite ($\text{Init-rhs} \ CS \ X$)
  using \text{assms unfolding Init-rhs-def transition-def} by \text{simp}

lemma \text{Init-ES-satisfies-invariant}:
  fixes $A :: (\langle a::\text{finite} \rangle \ \text{lang})$
  assumes finite-CS: finite ($\text{UNIV} \ // \ \approx A$)
  shows invariant ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
  proof (rule invariantI)
    show soundness ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
      unfolding soundness-def
      using l-eq-r-in-eqs by auto
    show finite ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$) using finite-CS
      unfolding Init-def by simp
    show distinctness ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
      unfolding distinctness-def Init-def by simp
    show ardenable-all ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
      unfolding ardenable-all-def Init-def Init-rhs-def ardenable-def
      by auto
    show finite-rhs ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
      using finite-Init-rhs[OF finite-CS]
      unfolding finite-rhs-def Init-def by auto
    show validity ($\text{Init} \ (\text{UNIV} \ // \ \approx A)$)
      unfolding validity-def Init-def Init-rhs-def rhss-def lhss-def
      by auto
  qed

2.6 Interations

lemma \text{Arden-preserves-soundness}:
  assumes l-eq-r: $X = \text{lang-rhs} \ \text{rhs}$
  and not-empty: ardenable \text{rhs}
  and finite: finite \text{rhs}
  shows $X = \text{lang-rhs} \ (\text{Arden} \ X \ \text{rhs})$
  proof
    def $A \equiv \text{lang} \ (\langle \{ r. \ \text{Trn} \ X \ r \in \text{rhs} \} \rangle)$
    def $b \equiv \{ \text{Trn} \ X \ r \ | \ r. \ \text{Trn} \ X \ r \in \text{rhs} \}$

\[ \text{def } B \equiv \text{lang-rhs} \ (\text{rhs} - b) \]

have \( \text{not-empty2: } [] \notin A \)
  using finite-Trn[OF finite] not-empty
  unfolding A-def ardenable-def by simp
have \( X = \text{lang-rhs rhs using l-eq-r by simp} \)
also have \( \ldots = \text{lang-rhs } (b \cup (\text{rhs} - b)) \) unfolding b-def by auto
also have \( \ldots = \text{lang-rhs } b \cup B \) unfolding B-def by (simp only: lang-rhs-union-distrib)
also have \( \ldots = X \cdot A \cup B \)
  unfolding b-def
  unfolding trm-soundness[OF finite]
  unfolding A-def by blast
finally have \( X = X \cdot A \cup B \).
then have \( X = B \cdot A^* \)
  by (simp add: reversed-Arden[OF not-empty2])
also have \( \ldots = \text{lang-rhs } (\text{Arden } X \text{ rhs}) \)
  unfolding Arden-def A-def b-def
  by (simp only: lang-of-append-rexp-rhs lang.simps)
finally show \( X = \text{lang-rhs } (\text{Arden } X \text{ rhs}) \) by simp
qed

lemma Append-preserves-finite:
  \( \text{finite rhs } \implies \text{finite } (\text{Append-rexp-rhs rhs r}) \)
by (auto simp: Append-rexp-rhs-def)

lemma Arden-preserves-finite:
  \( \text{finite rhs } \implies \text{finite } (\text{Arden } X \text{ rhs}) \)
by (auto simp: Arden-def Append-preserves-finite)

lemma Append-preserves-ardenable:
  \( \text{ardenable rhs } \implies \text{ardenable } (\text{Append-rexp-rhs rhs r}) \)
apply (auto simp: ardenable-def Append-rexp-rhs-def)
by (case-tac x, auto simp: conc-def)

lemma ardenable-set-sub:
  \( \text{ardenable rhs } \implies \text{ardenable } (\text{rhs} - A) \)
by (auto simp: ardenable-def)

lemma ardenable-set-union:
  \( \text{ardenable rhs; ardenable rhs} \) \( \implies \) \( \text{ardenable } (\text{rhs} \cup \text{rhs'}) \)
by (auto simp: ardenable-def)

lemma Arden-preserves-ardenable:
  \( \text{ardenable rhs } \implies \text{ardenable } (\text{Arden } X \text{ rhs}) \)
by (simp only:Arden-def Append-preserves-ardenable ardenable-set-sub)

lemma Subst-preserves-ardenable:
  \( \text{ardenable rhs; ardenable } x\text{rhs} \) \( \implies \) \( \text{ardenable } (\text{Subst } rhs \ X \ x\text{rhs}) \)
by (simp only: Subst-def Append-preserves-ardenable ardenable-set-union ardenable-set-sub)

lemma Subst-preserves-soundness:
assumes substor: X = lang-rhs xrhs
and finite: finite rhs
shows lang-rhs (Subst rhs X xrhs) = lang-rhs rhs (is ?Left = ?Right)
proof –
def A ≡ lang-rhs (rhs − {Trn X r | r. Trn X r ∈ rhs})
have ?Left = A ∪ lang-rhs (Append-rexp-rhs xrhs (⨄ (r. Trn X r ∈ rhs))")
  unfolding Subst-def
  unfolding lang-rhs-union-distrib[symmetric]
  by (simp add: A-def)
moreover have ?Right = A ∪ lang-rhs {Trn X r | r. Trn X r ∈ rhs}
proof –
have rhs = (rhs − {Trn X r | r. Trn X r ∈ rhs}) ∪ ({Trn X r | r. Trn X r ∈ rhs}) by auto
  thus ?thesis
  unfolding A-def
  unfolding lang-rhs-union-distrib
  by simp
qed
moreover
have lang-rhs (Append-rexp-rhs xrhs (⨄ (r. Trn X r ∈ rhs))) = lang-rhs {Trn X r | r. Trn X r ∈ rhs}
  using finite substor by (simp only: lang-of-append-rexp-rhs trm-soundness)
ultimately show ?thesis by simp
qed

lemma Subst-preserves-finite-rhs:
[finite rhs; finite yrhs] ⇒ finite (Subst rhs Y yrhs)
by (auto simp: Subst-def Append-preserves-finite)

lemma Subst-all-preserves-finite:
assumes finite: finite ES
shows finite (Subst-all ES Y yrhs)
using assms unfolding Subst-all-def by simp

declare [[simproc del: finite-Collect]]

lemma Subst-all-preserves-finite-rhs:
[finite-rhs ES; finite yrhs] ⇒ finite-rhs (Subst-all ES Y yrhs)
by (auto intro:Subst-preserves-finite-rhs simp add:Subst-all-def finite-rhs-def)

lemma append-rhs-preserves-cls:
rhss (Append-rexp-rhs rhs r) = rhss rhs
apply (auto simp: rhss-def Append-rexp-rhs-def)
apply (case-tac xa, auto simp: image-def)
by (rule-tac x = Times ra r in exI, rule-tac x = Trn x ra in bexI, simp)
lemma Arden-removes-cl:
\[
\text{rhss (Arden Y yrhs)} = \text{rhss yrhs } \setminus \{Y\}
\]
apply (simp add:Arden-def append-rhs-preserves-cl)
by (auto simp: rhss-def)

lemma lhss-preserves-cl:
\[
\text{lhss (Subst-all ES Y yrhs)} = \text{lhss ES}
\]
by (auto simp: lhss-def Subst-all-def)

lemma Subst-updates-cl:
\[
X \notin \text{rhss xrhs } \implies \text{rhss (Subst rhs X xrhs)} = \text{rhss rhs } \cup \text{rhss xrhs } \setminus \{X\}
\]
apply (simp only: Subst-def append-rhs-preserves-cl rhss-union-distrib)
by (auto simp: rhss-def)

lemma Subst-all-preserves-validity:
assumes sc: validity (ES \cup \{(Y, yrhs)\})
shows validity (Subst-all ES Y (Arden Y yrhs))
proof –
{ fix X xrhs'
  assume (X, xrhs') \in ?B
  then obtain xrhs
    where xrhs-xrhs': xrhs' = Subst xrhs Y (Arden Y yrhs)
    and X-in: (X, xrhs) \in ES by (simp add: Subst-all-def, blast)
  have rhss xrhs' \subseteq lhss ?B
    proof –
    have lhss ?B = lhss ES by (auto simp add: lhss-def Subst-all-def)
    moreover have rhss xrhs' \subseteq lhss ES
      proof –
      have rhss xrhs' \subseteq rhss xrhs \cup rhss (Arden Y yrhs) \setminus \{Y\}
        proof –
        have Y \notin rhss (Arden Y yrhs)
          using Arden-removes-cl by auto
          thus ?thesis using xrhs-xrhs' by (auto simp: Subst-updates-cl)
        qed
      moreover have rhss xrhs \subseteq lhss ES \cup \{Y\} using X-in sc
        apply (simp only: validity-def lhss-union-distrib)
        by (drule-tac x = (X, xrhs) in bspec, auto simp: lhss-def)
      moreover have rhss (Arden Y yrhs) \subseteq lhss ES \cup \{Y\}
        using sc
        by (auto simp add: Arden-removes-cl validity-def lhss-def)
      ultimately show ?thesis by auto
    qed
  ultimately show ?thesis by simp
  qed
} thus ?thesis by (auto simp only: Subst-all-def validity-def)
qed

lemma Subst-all-satisfies-invariant:
assumes invariant-ES: invariant \((ES \cup \{(Y, \text{yrhs})\})\)

shows invariant \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)

proof (rule invariantI)

have Y-eq-yrhs: \(Y = \text{lang-rhs} \ \text{yrhs}\)
  using invariant-ES by (simp only: invariant-def soundness-def, blast)

have finite-yrhs: finite \(\text{yrhs}\)
  using invariant-ES by (auto simp: invariant-def finite-rhs-def)

have ardenable-yrhs: ardenable \(\text{yrhs}\)
  using invariant-ES by (auto simp: invariant-def ardenable-all-def)

show soundness \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)

proof –

have \(Y = \text{lang-rhs} \ (\text{Arden} \ Y \ \text{yrhs})\)
  using Y-eq-yrhs invariant-ES finite-yrhs
  using finite-Trn[OF finite-yrhs]
  apply(rule-tac Arden-preserves-soundness)
  apply(simp-all)
  unfolding invariant-def ardenable-all-def ardenable-def
  apply(auto)
  done

thus ?thesis using invariant-ES
  unfolding invariant-def finite-rhs-def2 soundness-def Subst-all-def
  by (auto simp add: Subst-preserves-soundness simp del: lang-rhs.simps)

qed

show finite \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)
  using invariant-ES by (simp add: invariant-def Subst-all-preserves-finite)

show distinctness \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)
  using invariant-ES
  unfolding distinctness-def Subst-all-def invariant-def by auto

show ardenable-all \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)

proof –

\{ fix X rhs
  assume \((X, \ rhs) \in ES\)
  hence ardenable rhs using invariant-ES
    by (auto simp add: invariant-def ardenable-all-def)
  with ardenable-yrhs
  have ardenable \((\text{Subst} rhs \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)
    by (simp add: ardenable-yrhs
        Subst-preserves-ardenable Arden-preserves-ardenable)
  \} thus ?thesis by (auto simp add: ardenable-all-def Subst-all-def)

qed

show finite-rhs \((\text{Subst-all} \ ES \ Y \ (\text{Arden} \ Y \ \text{yrhs}))\)

proof –

have finite-rhs ES using invariant-ES
  by (simp add: invariant-def finite-rhs-def)

moreover have finite \((\text{Arden} \ Y \ \text{yrhs})\)

proof –

have finite yrhs using invariant-ES
  by (auto simp: invariant-def finite-rhs-def)

thus ?thesis using Arden-preserves-finite by auto
ultimately show thesis by (simp add: Subst-all-preserves-finite-rhs)

show validity (Subst-all ES Y (Arden Y yrhs)) using invariant-ES Subst-all-preserves-validity by (auto simp add: invariant-def)

lemma Remove-in-card-measure:
assumes finite: finite ES and in-ES: (X, rhs) ∈ ES
shows (Remove ES X rhs, ES) ∈ measure card
proof –
def f ≡ λ x. ((fst x)::‘a lang, Subst (snd x) X (Arden X rhs))
def ES′ ≡ ES −{(X, rhs)}
have Subst-all ES′ X (Arden X rhs) = f ' ES'
  apply (auto simp: Subst-all-def f-def image-def)
  by (rule-tac x = (Y, yrhs) in bexI, simp+)
then have card (Subst-all ES′ X (Arden X rhs)) ≤ card ES'
  unfolding ES'-'def using finite by (auto intro: card-image-le)
also have ... < card ES unfolding ES'-'def
  using in-ES finite by (rule-tac card-Diff1-less)
finally show (Remove ES X rhs, ES) ∈ measure card
  unfolding Remove-def ES'-'def by simp
qed

lemma Subst-all-cls-remains:
(X, xrhs) ∈ ES =⇒ ∃ xrhs'. (X, xrhs') ∈ (Subst-all ES Y yrhs)
by (auto simp: Subst-all-def)

lemma card-noteq-1-has-more:
assumes card: Cond ES and e-in: (X, xrhs) ∈ ES and finite: finite ES
shows ∃(Y, yrhs) ∈ ES. (X, xrhs) ≠ (Y, yrhs)
proof –
have card ES > 1 using card e-in finite
  by (cases card ES) (auto)
then have card (ES −{(X, xrhs)}) > 0
  using finite e-in by auto
then have (ES −{(X, xrhs)}) ≠ {} using finite by (rule-tac notI, simp)
then show ∃(Y, yrhs) ∈ ES. (X, xrhs) ≠ (Y, yrhs)
  by auto
qed

lemma iteration-step-measure:
assumes Inv-ES: invariant ES and X-in-ES: (X, xrhs) ∈ ES
and \( \text{Cond: Cond ES} \)
shows \( (\text{Iter X ES}, \text{ES}) \in \text{measure card} \)

**proof** –

have \( \text{fin: finite ES using Inv-ES unfolding invariant-def by simp} \)
then obtain \( Y \text{ yrhs} \)
where \( \text{Y-in-ES: (Y, yrhs) \in ES and not-eq: (X, xrhs) \neq (Y, yrhs)} \)
using \( \text{Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)} \)
then have \( (Y, yrhs) \in ES X \neq Y \)
using \( \text{X-in-ES Inv-ES unfolding invariant-def distinctness-def} \)
by auto
then show \( (\text{Iter X ES}, \text{ES}) \in \text{measure card} \)
apply(\text{rule IterI2})
apply(\text{rule Remove-in-card-measure})
apply(simp-all add: fin)
done
qed

**lemma** iteration-step-invariant:
assumes \( \text{Inv-ES: invariant ES} \)
and \( \text{X-in-ES: (X, xrhs) \in ES} \)
and \( \text{Cnd: Cond ES} \)
shows invariant \( (\text{Iter X ES}) \)
**proof** –

have \( \text{finite-ES: finite ES using Inv-ES by (simp add: invariant-def)} \)
then obtain \( Y \text{ yrhs} \)
where \( \text{Y-in-ES: (Y, yrhs) \in ES and not-eq: (X, xrhs) \neq (Y, yrhs)} \)
using \( \text{Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)} \)
then have \( (Y, yrhs) \in ES X \neq Y \)
using \( \text{X-in-ES Inv-ES unfolding invariant-def distinctness-def} \)
by auto
then show invariant \( (\text{Iter X ES}) \)
proof(\text{rule IterI2})
fix \( Y \text{ yrhs} \)
assume \( h: (Y, yrhs) \in ES X \neq Y \)
then have \( ES = \{(Y, yrhs)\} \cup \{(Y, yrhs)\} = ES \) by auto
then show invariant \( (\text{Remove ES Y yrhs}) \) unfolding \( \text{Remove-def} \)
using \( \text{Inv-ES} \)
by (\text{rule-tac Subst-all-satisfies-invariant}) (simp)
qed

**lemma** iteration-step-ex:
assumes \( \text{Inv-ES: invariant ES} \)
and \( \text{X-in-ES: (X, xrhs) \in ES} \)
and \( \text{Cnd: Cond ES} \)
shows \( \exists xrhs': (X, xrhs') \in (\text{Iter X ES}) \)
**proof** –

have \( \text{finite-ES: finite ES using Inv-ES by (simp add: invariant-def)} \)
then obtain \( Y \text{ yrhs} \)
where \( (Y, yrhs) \in ES (X, xrhs) \neq (Y, yrhs) \)
using Cnd X-in-ES by (drule-tac card-noteq-1-has-more) (auto)
then have \( (Y, yrhs) \in ES X \neq Y \)
using X-in-ES unfolding invariant-def distinctness-def
by auto
then show \( \exists xrhs'. (X, xrhs') \in (Iter X ES) \)
apply(rule IterI2)
unfolding Remove-def
apply(rule Subst-all-cls-remains)
using X-in-ES
apply(auto)
done
qed

2.7 The conclusion of the first direction

lemma Solve:
fixes A::('a::finite) lang
assumes fin: finite (UNIV // \approx A)
and X-in: X \in (UNIV // \approx A)
shows \( \exists rhs. \text{Solve} X (\text{Init} (\text{UNIV} // \approx A)) = \{(X, rhs)\} \land \text{invariant} \{(X, rhs)\} \)
proof
\[\text{def } \text{Inv} \equiv \lambda ES. \text{invariant } ES \land (\exists rhs. (X, rhs) \in ES)\]
have Inv (Init (UNIV // \approx A)) unfolding Inv-def
using fin X-in by (simp add: Init-ES-satisfies-invariant, simp add: Init-def)
moreover
\{ fix ES
assume inv: Inv ES and crd: Cnd ES
then have Inv (Iter X ES)
unfolding Inv-def
by (auto simp add: iteration-step-invariant iteration-step-ex) \}
moreover
\{ fix ES
assume inv: Inv ES and not-crd: \text{\neg} Cnd ES
from inv obtain rhs where \( (X, rhs) \in ES) \) unfolding Inv-def by auto
moreover
from not-crd have card ES = 1 by simp
ultimately
have ES = \{(X, rhs)\} by (auto simp add: card-Suc-eq)
than have \( \exists rhs'. ES = \{(X, rhs')\} \land \text{invariant} \{(X, rhs')\} \) using inv
unfolding Inv-def by auto \}
moreover
have wf (measure card) by simp
moreover
\{ fix ES
assume inv: Inv ES and crd: Cnd ES
then have (Iter X ES, ES) \in measure card
unfolding Inv-def
apply(clarify)
\}
apply(rule-tac iteration-step-measure)
apply(auto)
done
ultimately
show \( \exists \text{rhs}. \) Solve \( X \) (Init (UNIV // \approx A)) = \{(X, \text{rhs})\} \land invariant \{(X, \text{rhs})\}

unfolding Solve-def by (rule while-rule)

qed

lemma every-eqcl-has-reg:
fixes A::('a::finite) lang
assumes finite-CS: finite (UNIV // \approx A)
and X-in-CS: X \in (UNIV // \approx A)
sows \( \exists r. \) X = lang r

proof –
from finite-CS X-in-CS
obtain xrhs where Inv-ES: invariant \{(X, xrhs)\}
using Solve by metis

def A \equiv Arden X xrhs
have rhss xrhs \subseteq \{X\} using Inv-ES
  unfolding validity-def invariant-def rhss-def lhss-def
  by auto
then have rhss A = {} unfolding A-def
  by (simp add: Arden-removes-cl)
then have eq: \{Lam r \mid r. Lam r \in A\} = A unfolding rhss-def
  by (auto, case-tac x, auto)

have finite A using Inv-ES unfolding A-def invariant-def finite-rhs-def
  using Arden-preserves-finite by auto
then have fin: finite \{r. Lam r \in A\} by (rule finite-Lam)

have X = lang-rhs xrhs using Inv-ES unfolding invariant-def soundness-def
  by simp
then have X = lang-rhs A using Inv-ES
  unfolding A-def invariant-def ardenable-all-def finite-rhs-def
  by (rule-tac Arden-preserves-soundness) (simp-all add: finite-Trn)
then have X = lang-rhs \{Lam r \mid r. Lam r \in A\} using eq by simp
then have X = lang (\cup \{r. Lam r \in A\}) using fin by auto
then show \( \exists r. \) X = lang r by blast

qed

lemma bchoice-finite-set:
  assumes a: \forall x \in S. \exists y. x = f y
  and b: finite S
  shows \( \exists ys. \) (\bigcup S) = \bigcup f ' ys \land finite ys
using bchoice[OF a] b
apply(erule-tac exE)
apply(rule-tac x=f a \cdot S in exI)
apply(auto)
done

theorem Myhill-Nerode1:
  fixes A::('a::finite) lang
  assumes finite-CS: finite (UNIV // ≈A)
  shows ∃r. A = lang r
proof −
  have fin: finite (finals A)
    using finals-in-partitions finite-CS by (rule finite-subset)
  have ∀X ∈ (UNIV // ≈A). ∃r. X = lang r
    using finite-CS every-eqcl-has-reg by blast
  then have a: ∀X ∈ finals A. ∃r. X = lang r
    using finals-in-partitions by auto
  then obtain rs::('a rexp) set where ⋃ (finals A) = ⋃ (lang ' rs) finite rs
    using fin by (auto dest: bchoice-finite-set)
  then have A = lang (⋃ rs)
    unfolding lang-is-union-of-finals[symmetric] by simp
  then show ∃r. A = lang r by blast
qed

theory Myhill-2
  imports Myhill-1 ~~/src/HOL/Library/Sublist
begin

3 Second direction of MN: regular language ⇒ finite partition

3.1 Tagging functions

definition tag-eq :: ('a list ⇒ 'b) ⇒ ('a list × 'a list) set (=.=)
where
  =tag≡ ≡ {(x, y). tag x = tag y}

abbreviation
tag-eq-applied :: 'a list ⇒ ('a list ⇒ 'b) ⇒ 'a list ⇒ bool (.=.=. -)
where
  x =tag y ≡ (x, y) ∈ =tag=

lemma [simp]:
  shows (≈A) " {x} = (≈A) " {y} ⟷ x ≈A y
unfolding str-eq-def by auto

lemma refined-intro:
  assumes ∀x y z. [x =tag= y; x @ z ∈ A] ⟹ y @ z ∈ A
shows \(\tag = \subseteq \approx A\)
using \texttt{assms unfolding str-eq-def tag-eq-def}
apply(\texttt{clarify, simp (no-asm-use)})
by \texttt{metis}

\textbf{lemma} \texttt{finite-eq-tag-rel:}
\begin{itemize}
\item \texttt{assumes rng-fnt: finite (range \tag)}
\item \texttt{shows finite (UNIV // =\tag=)}
\end{itemize}
\textbf{proof –}
\begin{itemize}
\item \texttt{let} \(\?f = \lambda X. \tag ' X\) \texttt{and} \(\?A = (\UNIV // =\tag=)\)
\item \texttt{have} \texttt{finite (\?f ' \?A)}
\end{itemize}
\textbf{proof –}
\begin{itemize}
\item \texttt{have} \texttt{range \?f \subseteq (Pow (range \tag)) unfolding Pow-def by auto}
\item \texttt{moreover}
\item \texttt{have} \texttt{finite (Pow (range \tag)) using rng-fnt by simp}
\item \texttt{ultimately}
\item \texttt{have} \texttt{finite (range \?f) unfolding image-def by (blast intro: finite-subset)}
\item \texttt{moreover}
\item \texttt{have} \texttt{\?f ' \?A \subseteq range \?f by auto}
\item \texttt{ultimately show} \texttt{finite (\?f ' \?A) by (rule rev-finite-subset)}
\end{itemize}
\textbf{qed}
\textbf{moreover}
\textbf{have} \texttt{inj-on \?f \?A}
\textbf{proof –}
\begin{itemize}
\item \texttt{\{} \texttt{fix} X Y
\item \texttt{assume} X-in: \(X \in \?A\)
\item \texttt{and} Y-in: \(Y \in \?A\)
\item \texttt{and} tag-eq: \(\?f X = \?f Y\)
\item \texttt{then obtain} x y
\item \texttt{where} \(x \in X \land y \in Y\) \(\tag x = \tag y\)
\item \texttt{unfolding quotient-def Image-def image-def tag-eq-def}
\item \texttt{by (simp) (blast)}
\item \texttt{with} X-in Y-in
\item \texttt{have} \(X = Y\)
\item \texttt{unfolding quotient-def tag-eq-def by auto}
\item \texttt{\}}
\item \texttt{then show} \texttt{inj-on \?f \?A unfolding inj-on-def by auto}
\end{itemize}
\textbf{qed}
\textbf{ultimately show} \texttt{finite (UNIV // =\tag=) by (rule finite-imageD)}
\textbf{qed}

\textbf{lemma} \texttt{refined-partition-finite:}
\begin{itemize}
\item \texttt{assumes fnt: finite (UNIV // R1)}
\item \texttt{and refined: R1 \subseteq R2}
\item \texttt{and eq1: equiv UNIV R1 and eq2: equiv UNIV R2}
\item \texttt{shows finite (UNIV // R2)}
\end{itemize}
\textbf{proof –}
\begin{itemize}
\item \texttt{let} \(\?f = \lambda X. \{ R1 " \{ x \mid x \in X \}\} \) \texttt{and} \(\?A = UNIV // R2\) \texttt{and} \(\?B = UNIV // R1\)
\end{itemize}
have \( f \subseteq \text{Pow} \ ?B \)

unfolding image-def Pow-def quotient-def by auto

moreover

have finite (\text{Pow} \ ?B) using fnt by simp

ultimately

have finite (\( f \subseteq \?A \)) by (rule finite-subset)

moreover

have inj-on \( f \subseteq \?A \)

proof 

\{ fix \ X \ Y \\
assume X-in: \( X \subseteq \?A \) and Y-in: \( Y \subseteq \?A \) and eq-f: \( f \! = \! g \) \\
from quotientE [OF X-in] 

obtain \( x \) where \( X = R2 \{ \{ x \} \} \) by blast 

with equiv-class-self [OF eq2] have x-in: \( x \in X \) by simp 

then have \( R1 \{ \{ x \} \} \subseteq \?A \) by auto 

with eq-f have \( R1 \{ \{ x \} \} \subseteq \?A \) by simp 

then obtain \( y \) 

where y-in: \( y \in Y \) and eq-r1-xy: \( R1 \{ \{ x \} \} = R1 \{ \{ y \} \} \) by auto 

with eq-equiv-class [OF - eq1] have \( (x, y) \in R1 \) by blast 

with refined have \( (x, y) \in R2 \) by auto 

with quotient-eqI [OF eq2 X-in Y-in x-in y-in] have \( X = Y \).

\}

then show inj-on \( f \subseteq \?A \) unfolding inj-on-def by blast

qed

ultimately show finite \( (\text{UNIV} // R2) \) by (rule finite-imageD)

qed

lemma tag-finite-imageD:

assumes rng-fnt: finite (range tag)

and refined: \( =\text{tag}= \subseteq \approx \?A \)

shows finite \( (\text{UNIV} // \approx \?A) \)

proof (rule-tac refined-partition-finite [of \( =\text{tag}= \)])

show finite \( (\text{UNIV} // =\text{tag}=) \) by (rule finite-eq-tag-rel [OF rng-fnt])

next

show \( =\text{tag}= \subseteq \approx \?A \) using refined .

next

show equiv UNIV =\text{tag}= 

and equiv UNIV \( (\approx \?A) \)

unfolding equiv-def str-eq-def tag-eq-def refl-on-def sym-def trans-def 

by auto

qed

3.2 Base cases: Zero, One and Atom

lemma quot-zero-eq:

shows \( \text{UNIV} // =\{} = \{\text{UNIV} \}

unfolding quotient-def Image-def str-eq-def by auto
lemma quot-zero-finiteI [intro]:
  shows finite (UNIV // ≈{[]})
unfolding quot-zero-eq by simp

lemma quot-one-subset:
  shows UNIV // ≈{[]} ⊆ {{{}}, UNIV − {[]}}
proof
  fix x
  assume x ∈ UNIV // ≈{[]}
  then obtain y where h: x = {z. y ≈{[]} z}
  unfolding quotient-def Image-def by blast
  { assume y = []
    with h have x = {[]} by (auto simp: str-eq-def)
    then have x ∈ {{{}}, UNIV − {[]}} by simp }
  moreover
  { assume y ≠ []
    with h have x = UNIV − {[]} by (auto simp: str-eq-def)
    then have x ∈ {{{}}, UNIV − {[]}} by simp }
  ultimately show x ∈ {{{}}, UNIV − {[]}} by blast
qed

lemma quot-one-finiteI [intro]:
  shows finite (UNIV // ≈{[]})
by (rule finite-subset[OF quot-one-subset]) (simp)

lemma quot-atom-subset:
  UNIV // (≈{[c]}) ⊆ {{{},[c]}, UNIV − {[}, [c]}}
proof
  fix x
  assume x ∈ UNIV // ≈{[c]}
  then obtain y where h: x = {z. (y @ z) ∈ ≈{[c]}}
  unfolding quotient-def Image-def by blast
  show x ∈ {{{},[c]}, UNIV − {[}, [c]}}
  proof −
    { assume y = [] hence x = {[]} using h
      by (auto simp: str-eq-def) }
  moreover
    { assume y = [c] hence x = {[c]} using h
      by (auto dest!: spec[where x = [] simp: str-eq-def] ) }
  moreover
    { assume y ≠ [] and y ≠ [c]
      hence ∀ z. (y @ z) ≠ [c] by (case-tac y, auto)
      moreover have ∨ p. (p ≠ [] ∧ p ≠ [c]) = (∨ q. p @ q ≠ [c])
        by (case-tac p, auto)
      ultimately have x = UNIV − {[},[c]] using h
        by (auto simp add: str-eq-def) }
ultimately show thesis by blast
qed

lemma quot-atom-finiteI [intro]:
 shows finite (UNIV // ≈{c})
by (rule finite-subset[OF quot-atom-subset]) (simp)

3.3 Case for Plus
definition
tag-Plus :: 'a lang ⇒ 'a lang ⇒ 'a list ⇒ ('a lang × 'a lang)
where
tag-Plus A B ≡ λx. (≈ A "{x}, ≈ B "{x})

lemma quot-plus-finiteI [intro]:
 assumes finite1: finite (UNIV // ≈A)
 and finite2: finite (UNIV // ≈B)
 shows finite (UNIV // ≈(A ∪ B))
proof (rule_tac tag = tag-Plus A B in tag-finite-imageD)
have finite ((UNIV // ≈A) × (UNIV // ≈B))
  using finite1 finite2 by auto
then show finite (range (tag-Plus A B))
  unfolding tag-Plus-def quotient-def
  by (rule rev-finite-subset) (auto)
next
show ≈tag-Plus A B= ⊆≈ (A ∪ B)
  unfolding tag-eq-def tag-Plus-def str-eq-def by auto
qed

3.4 Case for Times
definition
Partitions x ≡ {(xp, xs). xp @ xs = x}

lemma conc-partitions-elim:
 assumes x ∈ A · B
 shows ∃(u, v) ∈ Partitions x. u ∈ A ∧ v ∈ B
using assms unfolding conc-def Partitions-def
by auto

lemma conc-partitions-intro:
 assumes (u, v) ∈ Partitions x ∧ u ∈ A ∧ v ∈ B
 shows x ∈ A · B
using assms unfolding conc-def Partitions-def
by auto

lemma equiv-class-member:
 assumes x ∈ A

and \( \approx A \{ x \} = \approx A \{ y \} \)
shows \( y \in A \)
using assms
apply(simp)
apply(simp add: str-eq-def)
apply(metis append-Nil2)
done

definition
\[ \text{tag-Times} :: \\
\text{'}a \text{ lang} \Rightarrow \\
\text{'}a \text{ lang} \Rightarrow \\
\text{'}a \text{ list} \Rightarrow \\
\text{'}a \text{ lang} \times \\
\text{'}a \text{ lang set} \]
where
\[ \text{tag-Times} \ A \ B \equiv \lambda x. (\approx A \{ x \}, \{(\approx B \{ x \}) \mid x_p x_s, x_p \in A \land (x_p, x_s) \in \text{Partitions} \ x \}) \]

lemma tag-Times-injI:
assumes \( a \): \( \text{tag-Times} \ A \ B \ x = \text{tag-Times} \ A \ B \ y \)
and \( c \): \( x @ z \in A \cdot B \)
shows \( y @ z \in A \cdot B \)
proof
- from \( c \) obtain \( u v \) where
  \( h1 \): \( (u, v) \in \text{Partitions} \ (x @ z) \) and
  \( h2 \): \( u \in A \) and
  \( h3 \): \( v \in B \) by (auto dest: conc-partitions-elim)
from \( h1 \) have \( x @ z = u @ v \) unfolding Partitions-def by simp
then obtain \( u^\prime \)\( u^\prime \) where
  \( x = u @ u^\prime \) and \( \approx A \{ u^\prime \} \) = \( \approx B \{ u^\prime \} \) and
  \( u^\prime \in A \) and \( v \in B \) by (auto simp add: append-eq-append-conv2)
moreover
\{ assume eq: \( x = u @ u \) \( u \) \( @ z = v \)
have \( (\approx B \{ u \}) \in \text{snd} \ (\text{tag-Times} \ A \ B \ x) \)
  unfolding Partitions-def tag-Times-def using \( h2 \) \( eq \)
  by (auto simp add: str-eq-def)
then have \( (\approx B \{ u \}) \in \text{snd} \ (\text{tag-Times} \ A \ B \ y) \)
  unfolding Partitions-def by auto
  using \( a \) by simp
then obtain \( u' \)\( u' \)\( u' \) where
  \( q1 \): \( u' \in A \) and
  \( q2 \): \( \approx B \{ u' \} = \approx B \{ u \} \) and
  \( q3 \): \( (u', u') \in \text{Partitions} \ y \)
  unfolding tag-Times-def by auto
from \( q2 \) \( h3 \) \( eq \)
have \( \approx B \{ u' \} \in B \)
  unfolding Image-def str-eq-def by auto
then have \( y @ z \in A \cdot B \) using \( q1 \) \( q3 \)
  unfolding Partitions-def by auto
\} moreover
\{ assume eq: \( x @ u \)\( u \) \( = u z = us @ v \)
have \( (\approx A \{ x \}) = \text{fst} \ (\text{tag-Times} \ A \ B \ x) \)
  by (simp add: tag-Times-def)
then have \( \approx \{ x \} = \text{fst} (\text{tag-Times} \ A \ B \ y) \)
using \( a \) by simp
then have \( \approx \{ x \} = \approx \{ y \} \)
by \( (\text{simp add: tag-Times-def}) \)
moreover
have \( x \ast us \in A \) using \( h2 \) eq by simp
ultimately
have \( y \ast us \in A \) using equiv-class-member
unfolding Image-def str-eq-def by blast
then have \( y \ast z \in A \cdot B \) using \( h3 \) unfolding conc-def by blast
then have \( y \ast z \in A \cdot B \) using eq by simp
}
ultimately show \( y \ast z \in A \cdot B \) by blast
qed

lemma \( \text{quot-conc-finiteI} \) [intro]:
assumes \( \text{fin1: finite (UNIV} /\!\!\!/ \approx A) \)
and \( \text{fin2: finite (UNIV} /\!\!\!/ \approx B) \)
shows \( \text{finite (UNIV} /\!\!\!/ \approx (A \cdot B)) \)
proof \( (\text{rule-tac tag = tag-Times} \ A \ B \ \text{in tag-finite-imageD}) \)
have \( \forall x \ y z . [\text{tag-Times} \ A \ B \ x = \text{tag-Times} \ A \ B \ y; x \ast z \in A \cdot B] \Longrightarrow y \ast z \in A \cdot B \)
by \( (\text{rule tag-Times-injI}) \)
(auto simp add: tag-Times-def tag-eq-def)
then show \( =\text{tag-Times} \ A \ B = \subseteq \approx (A \cdot B) \)
by \( (\text{rule refined-intro}) \)
(auto simp add: tag-eq-def)
next
have \( \ast: \text{finite ((UNIV} /\!\!\!/ \approx A) \times (\text{Pow} \ (\text{UNIV} /\!\!\!/ \approx B))) \)
using \( \text{fin1 fin2 by auto} \)
show \( \text{finite (range (tag-Times A B))} \)
unfolding tag-Times-def
apply \( (\text{rule finite-subset[OF - \ast])}) \)
unfolding quotient-def
by auto
qed

3.5 Case for \( \text{Star} \)

lemma \( \text{star-partitions-elim}: \)
assumes \( x \ast z \in A \ast \ x \neq [] \)
shows \( \exists (u, v) \in \text{Partitions} \ (x \ast z), \text{prefix} \ \text{u} \ \text{x} \ \wedge \ \text{u} \ \in A \ast \ \wedge \ \text{v} \ \in A \ast \)
proof \( - \)
have \( (], x \ast z) \in \text{Partitions} \ (x \ast z) \ \text{prefix} \ [] \ \text{x} \ [] \ \in A \ast \ x \ @ z \in A \ast \)
using \( \text{assms by} \ (\text{auto simp add: Partitions-def prefix-def}) \)
then show \( \exists (u, v) \in \text{Partitions} \ (x \ast z), \text{prefix} \ \text{u} \ \text{x} \ \wedge \ \text{u} \ \in A \ast \ \wedge \ \text{v} \ \in A \ast \)
by blast
qed
lemma finite-set-has-max2:
\[
\text{[finite } A; A \neq \{\}] \implies \exists \ \text{max } \in A. \ \forall \ a \in A. \ \text{length } a \leq \text{length max}
\]
apply(induct rule:finite.induct)
apply(simp)
by (metis (no-types) all-not-in-conv insert-iff linorder-le-cases order-trans)

lemma finite-prefix-set:
shows finite \{xa. prefix xa (x::'a list)\}
apply (induct rule:rev-induct, simp)
apply (subgoal-tac \{xa. prefix xa xs \} = \{xa. prefix xa xs\} \cup \{xs\})
by (auto simp)

lemma append-eq-cases:
assumes a: x @ y = m @ n m \neq []
shows prefixeq x m \lor prefix m x
unfolding prefixeq-def prefix-def using a
by (auto simp add: append-eq-append-conv2)

lemma star-partitions-elim2:
assumes a: x @ z \in A
and b: x \neq []
shows \exists \ (u, v) \in Partitions x. \exists (u', v') \in Partitions z. \prefix u x \land u \in A^* \land v @ z \in A^*
proof –
def S \equiv \{u \mid u v. (u, v) \in \text{Partitions } x \land \prefix u x \land u \in A^* \land v @ z \in A^*\}
have finite S unfolding S-def
by (rule rev-finite-subset) (auto)
moreover
have S \neq {} using a b unfolding S-def Partitions-def
by (auto simp: prefix-def)
ultimately have \exists u-max \in S. \forall u \in S. \text{length } u \leq \text{length u-max}
using finite-set-has-max2 by blast
then obtain u-max v
where h0: (u-max, v) \in Partitions x
and h1: prefix u-max x
and h2: u-max \in A^*
and h3: v @ z \in A^*
and h4: \forall u v. (u, v) \in Partitions x \land \prefix u x \land u \in A^* \land v @ z \in A^* \implies
\text{length } u \leq \text{length } u-max
unfolding S-def Partitions-def by blast
have q: v \neq [] using h0 h1 b unfolding Partitions-def by auto
from h5 obtain a b
where i1: (a, b) \in Partitions (v @ z)
and i2: a \in A
and i3: b \in A^*
and i4: a \neq []
unfolding Partitions-def
using q by (auto dest: star-decom)

have prefixeq v a
proof (rule ccontr)
  assume a: ¬(prefixeq v a)
  from i1 have i1': a ⊕ b = v ⊕ z unfolding Partitions-def by simp
  then have prefixeq a v ∨ prefix v a using append-eq-cases q by blast
  then have q: prefix a v using a unfolding prefix-def prefixeq-def by auto
  then obtain as where eq: a ⊕ as = v unfolding prefix-def prefixeq-def by auto
  then have (u-max ⊕ a, as) ∈ Partitions x using eq h0 unfolding Partitions-def
  moreover have prefix (u-max ⊕ a) x using h0 eq unfolding Partitions-def prefix-def
  prefixeq-def by auto
  moreover have u-max ⊕ a ∈ A* using i2 h2 by simp
  moreover have as ⊕ z ∈ A* using i1' i2 i3 eq by auto
  ultimately have length (u-max ⊕ a) ≤ length u-max using h4 by blast
  with i4 show False by auto
qed

definition
tag-Star :: 'a lang ⇒ 'a list ⇒ ('a lang) set
where
tag-Star A ≡ λx. {≈ A ⊕ {v} | u v. prefix u x ∧ u ∈ A* ∧ (u, v) ∈ Partitions x}

lemma tag-Star-non-empty-injI:
assumes a: tag-Star A x = tag-Star A y
and c: x ⊕ z ∈ A*
and d: x ≠ []
show y ⊕ z ∈ A*
proof 
  obtain u v u' v'
    where a1: (u, v) ∈ Partitions x (u', v') ∈ Partitions z
    and a2: prefix u x
    and a3: u ∈ A*
    and a4: v ⊕ u' ∈ A
    and a5: v' ∈ A*

  with i1 obtain za zb
    where k1: v ⊕ za = a
    and k2: (za, zb) ∈ Partitions z
    and k4: zb = b
    unfolding Partitions-def prefix-def
    by (auto simp add: append-eq-append-conv2)
  show ∃ (u, v) ∈ Partitions x. ∃ (u', v') ∈ Partitions z. prefix u x ∧ u ∈ A* ∧ v ⊕ u' ∈ A ∧ v' ∈ A*
    using h0 h1 h2 i2 i3 k1 k2 k4 unfolding Partitions-def by blast
qed
using c d by (auto dest: star-partitions-elim2)

have \((\approx A) \ " \ \{v\} \in \text{tag-Star} A x\)
apply(simp add: tag-Star_def Partitions_def str-eq-def)
using a1 a2 a3 by (auto simp add: Partitions_def)
then have \((\approx A) \ " \ \{v\} \in \text{tag-Star} A y\) using a by simp
then obtain u1 v1
where b1: \(v \approx A v1\)
and b3: u1 \in A*
and b4: (u1, v1) \in \text{Partitions} y
unfolding tag-Star_def by auto
have c: v1 @ u' \in A* using b1 a4 unfolding str-eq-def by simp
have u1 @ (v1 @ u') @ v' \in A*
using b3 c a5 by (simp only: append-in-StarI)
then show y @ z \in A* using b4 a1 unfolding Partitions_def by auto

qed

lemma tag-Star-empty-injI:
assumes a: tag-Star A x = tag-Star A y
and c: x @ z \in A*
and d: x = []
shows y @ z \in A*
proof
from a have \{} = tag-Star A y unfolding tag-Star_def using d by auto
then have y = []
unfolding tag-Star-def Partitions-def prefix-def prefixeq-def
by (auto) (metis Nil-in-star append-self-conv2)
then show y @ z \in A* using c d by simp

qed

lemma quot-star-finiteI [intro]:
assumes finite1: finite (UNIV // \approx A)
show finite (UNIV // \approx (A*))
proof (rule_tac tag = tag-Star A in tag-finite-imageD)
have \(\forall x y z. [\text{tag-Star} A x = \text{tag-Star} A y; x @ z \in A*] \implies y @ z \in A*\)
by (case_tac x = []) (blast intro: tag-Star-empty-injI tag-Star-non-empty-injI)+
then show \=(tag-Star A)= \subseteq \approx (A*)
by (rule refined-intro) (auto simp add: tag-eq-def)

next
have *: finite (Pow (UNIV // \approx A))
  using finite1 by auto
show finite (range (tag-Star A))
  unfolding tag-Star-def
  by (rule finite-subset[OF - *])
    (auto simp add: quotient-def)

qed
3.6 The conclusion of the second direction

lemma Myhill-Nerode2:
  fixes r::'a rexpr
  shows finite (UNIV // ≈(lang r))
by (induct r) (auto)
end

theory Myhill
  imports Myhill-2 ../Regular-sets/Derivatives
begin

4 The theorem

theorem Myhill-Nerode:
  fixes A::('a::finite) lang
  shows (∃r. A = lang r) ⟷ finite (UNIV // ≈A)
using Myhill-Nerode1 Myhill-Nerode2 by auto

4.1 Second direction proved using partial derivatives

An alternative proof using the notion of partial derivatives for regular expressions due to Antimirov [1].

lemma MN-Rel-Derivs:
  shows x ≈ A y ⟷ Derivs x A = Derivs y A
unfolding Derivs-def str-eq-def
by auto

lemma Myhill-Nerode3:
  fixes r::'a rexpr
  shows finite (UNIV // ≈(lang r))
proof –
  have finite (UNIV // = (λx. pderivs x r) =)
  proof –
    have range (λx. pderivs x r) ⊆ Pow (pderivs-lang UNIV r)
      unfolding pderivs-lang-def by auto
    moreover
    have finite (Pow (pderivs-lang UNIV r)) by (simp add: finite-pderivs-lang)
    ultimately
    have finite (range (λx. pderivs x r))
      by (simp add: finite-subset)
    then show finite (UNIV // = (λx. pderivs x r) =)
      by (rule finite-eq-tag-rel)
  qed
  moreover
  have =(λx. pderivs x r) = ⊆ ≈(lang r)
  unfolding tag-eq-def
by (auto simp add: MN-Rel-Derivs Derivs-pderivs)
moreover
have equiv UNIV = (λx. pderiv x r) =
  unfolding equiv-def refl-on-def sym-def trans-def
  unfolding tag-eq-def str-eq-def
  by auto
ultimately show finite (UNIV // r)
  by (rule refined-partition-finite)
qed
end

theory Closures
import Myhill ~~/src/HOL/Library/Infinite-Set
begin

5 Closure properties of regular languages

abbreviation
  regular :: 'a lang ⇒ bool
where
  regular A ≡ ∃ r. A = lang r

5.1 Closure under ∪, · and *

lemma closure-union [intro]:
  assumes regular A regular B
  shows regular (A ∪ B)
proof –
  from assms obtain r1 r2 :: 'a rexp where lang r1 = A lang r2 = B by auto
  then have A ∪ B = lang (Plus r1 r2) by simp
  then show regular (A ∪ B) by blast
qed

lemma closure-seq [intro]:
  assumes regular A regular B
  shows regular (A · B)
proof –
  from assms obtain r1 r2 :: 'a rexp where lang r1 = A lang r2 = B by auto
  then have A · B = lang (Times r1 r2) by simp
  then show regular (A · B) by blast
qed

lemma closure-star [intro]:
  assumes regular A
  shows regular (A*)
proof –
  from assms obtain r :: 'a rexp where lang r = A by auto
then have $A^* = \text{lang } (\text{Star } r)$ by simp
then show regular $(A^*)$ by blast
qed

5.2 Closure under complementation

Closure under complementation is proved via the Myhill-Nerode theorem

lemma closure-complement [intro]:
  fixes $A::('a::finite) \text{ lang}$
  assumes regular $A$
  shows regular $(\sim A)$
proof –
  from assms have finite $(\text{UNIV } / \sim A)$ by (simp add: Myhill-Nerode)
  then have finite $(\text{UNIV } / \sim \sim A)$ by (simp add: str-eq-def)
  then show regular $(\sim A)$ by (simp add: Myhill-Nerode)
qed

5.3 Closure under $-$ and $\cap$

lemma closure-difference [intro]:
  fixes $A::('a::finite) \text{ lang}$
  assumes regular $A$ regular $B$
  shows regular $(A - B)$
proof –
  have $A - B = \sim (\sim A \cup B)$ by blast
moreover
  have regular $(\sim (\sim A \cup B))$
    using assms by blast
ultimately show regular $(A - B)$ by simp
qed

lemma closure-intersection [intro]:
  fixes $A::('a::finite) \text{ lang}$
  assumes regular $A$ regular $B$
  shows regular $(A \cap B)$
proof –
  have $A \cap B = \sim (\sim A \cup \sim B)$ by blast
moreover
  have regular $(\sim (\sim A \cup \sim B))$
    using assms by blast
ultimately show regular $(A \cap B)$ by simp
qed

5.4 Closure under string reversal

fun
  $\text{Rev} :: 'a \text{ rexp } \Rightarrow 'a \text{ rexp}$
where
  $\text{Rev} \text{ Zero } = \text{ Zero}$

31
lemma rev-seq[simp]:
  shows \( \text{rev} \cdot (B \cdot A) = (\text{rev} \cdot A) \cdot (\text{rev} \cdot B) \)
unfolding conc-def image-def

by (auto) (metis rev-append)

lemma rev-star1:
  assumes a: \( s \in (\text{rev} \cdot A)^* \)
  shows \( s \in \text{rev} \cdot (A^*) \)
using a
proof (induct rule: star-induct)
case (append s1 s2)
  have inj: inj (rev::\text{a list} \Rightarrow \text{a list}) unfolding inj-on-def by auto
  have s1 \( \in \text{rev} \cdot A \) s2 \( \in \text{rev} \cdot (A^*) \) by fact
  then obtain x1 x2 where x1 \( \in A \) x2 \( \in A^* \) and eqs: s1 = rev x1 s2 = rev x2
  by auto
  then have x1 \( \in A^* \) x2 \( \in A^* \) by (auto)
  then have rev s1 \( \in \text{rev} \cdot A^* \) by (auto)
  then have \( \text{rev} \cdot s2 \in (\text{rev} \cdot A^*)^* \) by (auto)
  moreover
  have \( \text{rev} \cdot s2 \in (\text{rev} \cdot A)^* \) by fact
  ultimately show \( \text{rev} \cdot (s1 \circ s2) \in (\text{rev} \cdot A^*)^* \) by (auto)
qed (auto)

lemma rev-star2:
  assumes a: \( s \in A^* \)
  shows \( \text{rev} s \in (\text{rev} \cdot A)^* \)
using a
proof (induct rule: star-induct)
case (append s1 s2)
  have inj: inj (rev::\text{a list} \Rightarrow \text{a list}) unfolding inj-on-def by auto
  have s1 \( \in \text{Aby} \) fact
  then have \( \text{rev} \cdot s1 \in \text{rev} \cdot A \) using inj by (simp only: inj-image-mem-iff)
  then have \( \text{rev} \cdot s1 \in (\text{rev} \cdot A^*)^* \) by (auto)
  moreover
  have \( \text{rev} \cdot s2 \in (\text{rev} \cdot A^*)^* \) by fact
  ultimately show \( \text{rev} \cdot (s1 \circ s2) \in (\text{rev} \cdot A^*)^* \) by (auto)
qed (auto)

lemma rev-star [simp]:
  shows \( \text{rev} \cdot (A^*) = (\text{rev} \cdot A)^* \)
using rev-star1 rev-star2 by auto

lemma rev-lang:
  shows \( \text{rev} \cdot (\text{lang} r) = \text{lang} (\text{Rev} r) \)
by (induct r) (simp-all add: image-Un)
lemma closure-reversal [intro]:
  assumes regular A
  shows regular (rev ' A)
proof –
  from assms obtain r::'a rexp where A = lang r by auto
  then have lang (Rev r) = rev ' A by (simp add: rev-lang)
  then show regular (rev' A) by blast
qed

5.5 Closure under left-quotients
abbreviation
  Deriv-lang A B ≡ ∪ x ∈ A. Derivs x B

lemma closure-left-quotient:
  assumes regular A
  shows regular (Deriv-lang B A)
proof –
  from assms obtain r::'a rexp where eq: lang r = A by auto
  have fin: finite (pderivs-lang B r) by (rule finite-pderivs-lang)
  have Deriv-lang B (lang r) = (∪ (lang ' pderivs-lang B r))
    by (simp add: Derivs-pderivs pderivs-lang-def)
  also have . . . = lang (∪ (pderivs-lang B r)) using fin by simp
  finally have Deriv-lang B A = lang (∪ (pderivs-lang B r)) using eq
    by simp
  then show regular (Deriv-lang B A) by auto
qed

5.6 Finite and co-finite sets are regular

lemma singleton-regular:
  shows regular {s}
proof (induct s)
case Nil
  have [[]] = lang (One) by simp
  then show regular {[]} by blast
next
case (Cons c s)
  have regular {s} by fact
  then obtain r where {s} = lang r by blast
  then have {c # s} = lang (Times (Atom c) r)
    by (auto simp add: conc-def)
  then show regular {c # s} by blast
qed

lemma finite-regular:
  assumes finite A
  shows regular A
using assms
proof (induct)
  case empty
  have {} = lang (Zero) by simp
  then show regular {} by blast
next
  case (insert s A)
  have regular {s} by (simp add: singleton-regular)
  moreover
  have regular A by fact
  ultimately have regular ({s} ∪ A) by (rule closure-union)
  then show regular (insert s A) by simp
qed

lemma cofinite-regular:
  fixes A :: 'a::finite lang
  assumes finite (− A)
  shows regular A
proof −
  from assms have regular (− A) by (simp add: finite-regular)
  then have regular (−(− A)) by (rule closure-complement)
  then show regular A by simp
qed

5.7 Continuation lemma for showing non-regularity of languages

lemma continuation-lemma:
  fixes A B :: 'a::finite lang
  assumes reg: regular A
  and inf: infinite B
  shows ∃x ∈ B. ∃y ∈ B. x ≠ y ∧ x ≈ A y
proof −
  def eqfun ≡ λA x::('a::finite list). (≈A) " {x}
  have finite (UNIV // ≈A) using reg by (simp add: Myhill-Nerode)
  moreover
  have (eqfun A) * B ⊆ UNIV // (≈A)
    unfolding eqfun-def quotient-def by auto
  ultimately have finite ((eqfun A) * B) by (rule rev-finite-subset)
  with inf have ∃a ∈ B. infinite {b ∈ B. eqfun A b = eqfun A a}
    by (rule pigeonhole-infinite)
  then obtain a where in-a: a ∈ B and infinite {b ∈ B. eqfun A b = eqfun A a}
    by blast
  moreover
  have {b ∈ B. eqfun A b = eqfun A a} = {b ∈ B. b ≈ A a}
    unfolding eqfun-def Image-def str-eq-def by auto
  ultimately have infinite {b ∈ B. b ≈ A a} by simp
  then have infinite ( {b ∈ B. b ≈ A a} − {a}) by simp
moreover
have \{ b \in B. b \approx_A a \} - \{ a \} = \{ b \in B. b \approx a \land b \neq a \} by auto
ultimately have infinite \{ b \in B. b \approx_A a \land b \neq a \} by simp
then have \{ b \in B. b \approx_A a \land b \neq a \} \neq \{
  by (metis finite.emptyI)
then obtain b where b \in B b \neq a b \approx_A a by blast
with in-a show \exists x \in B. \exists y \in B. x \neq y \land x \approx_A y
  by blast
qed

5.8 The language $a^n b^n$ is not regular

abbreviation
replicate-rev (- `^-` [100, 100] 100)
where
  a `^-` n ≡ replicate n a

lemma an-bn-not-regular:
shows \neg regular (\bigcup n. \{ CHR "a" `^-` n @ CHR "b" `^-` n \})
proof
def A ≡\bigcup n. \{ CHR "a" `^-` n @ CHR "b" `^-` n \}
def B ≡\bigcup n. \{ CHR "a" `^-` n \}
assume as: regular A
def B ≡\bigcup n. \{ CHR "a" `^-` n \}

have sameness: \( \forall i j. CHR "a" `^-` i \land CHR "b" `^-` j \in A \leftrightarrow i = j \)
  unfolding A-def
  apply auto
  apply (drule-tac f=\lambda s. length (filter (op= (CHR "a") ) s) = length (filter (op=
                         (CHR "b") ) s)
                      in arg-cong)
  apply(simp)
done

have b: infinite B
  unfolding infinite-if-countable-subset
  unfolding inj-on-def B-def
  by (rule-tac x=\lambda n. CHR "a" `^-` n in exI) (auto)
moreover
have \( \forall x \in B. \forall y \in B. x \neq y \rightarrow \neg (x \approx_A y) \)
  apply(auto)
  unfolding B-def
  apply(auto)
  apply(simp add: str-eq-def)
  apply(drule-tac x=CHR "b" `^-` n in spec)
  apply(simp add: sameness)
done
ultimately
show False using continuation-lemma[OF as] by blast
6 Normalizing Derivative

theory NDerivative
imports Regular-Exp
begin

6.1 Normalizing operations

associativity, commutativity, idempotence, zero

fun nPlus :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
nPlus Zero r = r
| nPlus r Zero = r
| nPlus (Plus r s) t = nPlus r (nPlus s t)
| nPlus r (Plus s t) =
  (if r = s then (Plus s t)
   else if le-rexp r s then Plus r (Plus s t)
   else Plus s (nPlus r t))
| nPlus r s =
  (if r = s then r
   else if le-rexp r s then Plus r s
   else Plus s r)

lemma lang-nPlus[simp]: lang (nPlus r s) = lang (Plus r s)
by (induction r s rule: nPlus.induct) auto

associativity, zero, one

fun nTimes :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
nTimes Zero - = Zero
| nTimes - Zero = Zero
| nTimes One r = r
| nTimes r One = r
| nTimes (Times r s) t = Times r (nTimes s t)
| nTimes r s = Times r s

lemma lang-nTimes[simp]: lang (nTimes r s) = lang (Times r s)
by (induction r s rule: nTimes.induct) (auto simp: conc-assoc)

primrec norm :: 'a::order rexp ⇒ 'a rexp
where
norm Zero = Zero

qed

end
\[ \text{norm One} = \text{One} \]
\[ \text{norm} (\text{Atom}\ a) = \text{Atom}\ a \]
\[ \text{norm} (\text{Plus}\ r\ s) = n\text{Plus} \left( \text{norm}\ r \right) \left( \text{norm}\ s \right) \]
\[ \text{norm} (\text{Times}\ r\ s) = n\text{Times} \left( \text{norm}\ r \right) \left( \text{norm}\ s \right) \]
\[ \text{norm} (\text{Star}\ r) = \text{Star} \left( \text{norm}\ r \right) \]

\textbf{lemma} \ \text{lang-norm[simp]}: \ \text{lang} \left( \text{norm}\ r \right) = \text{lang}\ r \\
\text{by} \ (\text{induct}\ r) \ \text{auto}

\textbf{primrec} \ \textit{nderiv} :: \ 'a::order ⇒ 'a\ rexp ⇒ 'a\ rexp \\
\text{where} \\
\ \textit{nderiv}\ -\ Zero = \text{Zero} \\
\ \textit{nderiv}\ -\ One = \text{Zero} \\
\ \textit{nderiv}\ a\ (\text{Atom}\ b) = (\text{if}\ a = b\ \text{then}\ \text{One}\ \text{else}\ \text{Zero}) \\
\ \textit{nderiv}\ a\ (\text{Plus}\ r\ s) = n\text{Plus} \left( \text{nderiv}\ a\ r \right) \left( \text{nderiv}\ a\ s \right) \\
\ \textit{nderiv}\ a\ (\text{Times}\ r\ s) = \\
\ \ \quad (\text{let}\ r′′s = n\text{Times} \left( \text{nderiv}\ a\ r \right) s \\
\ \ \ \ \quad \text{if}\ \text{nullable}\ r\ \text{then}\ n\text{Plus} r′′s \left( \text{nderiv}\ a\ s \right) \text{else}\ r′′s) \\
\ \textit{nderiv}\ a\ (\text{Star}\ r) = n\text{Times} \left( \text{nderiv}\ a\ r \right) (\text{Star}\ r)

\textbf{lemma} \ \text{lang-nderiv}: \ \text{lang} \left( \text{nderiv}\ a\ r \right) = \text{Deriv}\ a \left( \text{lang}\ r \right) \\
\text{by} \ (\text{induction}\ r) \ (\text{auto}\ \text{simp}: \ \text{Let-def}\ \text{nullable-iff})

\textbf{lemma} \ \text{deriv-no-occurrence}: \\
\ \ x \notin\ \text{atoms}\ r \implies \ \text{nderiv}\ x\ r = \text{Zero} \\
\text{by} \ (\text{induction}\ r) \ \text{auto}

\textbf{lemma} \ \text{atoms-nPlus[simp]}: \ \text{atoms} \left( n\text{Plus}\ r\ s \right) = \text{atoms}\ r \cup \text{atoms}\ s \\
\text{by} \ (\text{induction}\ r\ s\ \text{rule}: n\text{Plus.induct}) \ \text{auto}

\textbf{lemma} \ \text{atoms-nTimes}: \ \text{atoms} \left( n\text{Times}\ r\ s \right) \subseteq \text{atoms}\ r \cup \text{atoms}\ s \\
\text{by} \ (\text{induction}\ r\ s\ \text{rule}: n\text{Times.induct}) \ \text{auto}

\textbf{lemma} \ \text{atoms-norm}: \ \text{atoms} \left( \text{norm}\ r \right) \subseteq \text{atoms}\ r \\
\text{by} \ (\text{induction}\ r) \ (\text{auto\ dest!:subsetD}[OF\ \text{atoms-nTimes}])

\textbf{lemma} \ \text{atoms-nderiv}: \ \text{atoms} \left( \text{nderiv}\ a\ r \right) \subseteq \text{atoms}\ r \\
\text{by} \ (\text{induction}\ r) \ (\text{auto\ simp}: \ \text{Let-def}\ \text{dest!:subsetD}[OF\ \text{atoms-nTimes}])

\textbf{end}

7 Deciding Regular Expression Equivalence

\textbf{theory} \ \textit{Equivalence-Checking} \\
\textbf{imports} \ \textit{NDerivative} \\
\ \ \quad \sim\sim/src/HOL/Library/While-Combinator \\
\textbf{begin}

37


7.1 Bisimulation between languages and regular expressions

coinductive bisimilar :: 'a lang ⇒ 'a lang ⇒ bool where
([] ∈ K ◄⇒ [] ∈ L)
⇒ (\x. bisimilar (Deriv x K) (Deriv x L))
⇒ bisimilar K L

lemma equal-if-bisimilar:
assumes bisimilar K L shows K = L
proof (rule set-eqI)
fix w
from (bisimilar K L) show w ∈ K ◄⇒ w ∈ L
proof (induct w arbitrary: K L)
case Nil thus \_case by (auto elim: bisimilar.cases)
next
case (Cons a w K L) from (bisimilar K L) have bisimilar (Deriv a K) (Deriv a L)
by (auto elim: bisimilar.cases)
then have w ∈ Deriv a K ◄⇒ w ∈ Deriv a L by (rule Cons(1))
thus \_case by (auto simp: Deriv-def)
qed

lemma language-coinduct:
fixes R (infixl ~ 50)
assumes K ~ L
assumes (\K L. K ~ L ⇒ ([] ∈ K ◄⇒ []) ∈ L)
assumes (\K L x. K ~ L ⇒ Deriv x K ~ Deriv x L)
shows K = L
apply (rule equal-if-bisimilar)
apply (rule bisimilar.coinduct[of R, OF 'K ~ L])
apply (auto simp: assms)
done

type-synonym 'a rexp-pair = 'a rexp × 'a rexp

type-synonym 'a rexp-pairs = 'a rexp-pair list

definition is-bisimulation :: 'a::order list ⇒ 'a rexp-pair set ⇒ bool
where
is-bisimulation as R =
(∀ (r,s)∈ R. (atoms r ∪ atoms s ⊆ set as) ∧ (nullable r ◄⇒ nullable s) ∧
(∀ a∈ set as. (nderiv a r, nderiv a s) ∈ R))

lemma bisim-lang-eq:
assumes bisim: is-bisimulation as ps
assumes (r, s) ∈ ps
shows lang r = lang s
proof –
def \p = insert (Zero, Zero) ps
from bisim have bisim\p: is-bisimulation as \p

38
by (auto simp: ps'-def is-bisimulation-def)
let ?R = \lambda K L. (\exists (r,s) \in ps'. K = \text{lang } r \land L = \text{lang } s)
show ?thesis
proof (rule language-coinduct[where R=?R])
  from (r, s) \in ps
  have (r, s) \in ps' by (auto simp: ps'-def)
  thus ?thesis by auto
next
  fix K L assume ?R K L
  then obtain r s where rs: (r, s) \in ps'
    and KL: K = \text{lang } r L = \text{lang } s by auto
  with bisim' have nullable r \leftrightarrow nullable s
    by (auto simp: is-bisimulation-def)
  thus \[] \in K \leftrightarrow \[] \in L by (auto simp: nullable-iff KL)
  fix a
  show ?R (\text{Deriv } a K) (\text{Deriv } a L)
  proof cases
    assume a \in set as
    with rs bisim'
    have (nderiv a r, nderiv a s) \in ps'
      by (auto simp: is-bisimulation-def)
    thus ?thesis by (force simp: KL lang-nderiv)
  next
    assume a /\in set as
    with bisim' rs
    have a /\in atoms r a /\in atoms s by (auto simp: is-bisimulation-def)
    then have nderiv a r = Zero nderiv a s = Zero
      by (auto intro: deriv-no-occurrence)
    then have \text{Deriv } a K = \text{lang } Zero
      Deriv a L = \text{lang } Zero
      unfolding KL lang-nderiv[symmetric] by auto
    thus ?thesis by (auto simp: ps'-def)
  qed
  qed

7.2 Closure computation

definition closure :: 'a::order list \Rightarrow 'a rexp-pair \Rightarrow ('a rexp-pairs + 'a rexp-pair set) option
where
closure as = rtrancl-while (%(r,s). nullable r = nullable s)
  (%(r,s). map (\lambda a. (nderiv a r, nderiv a s)) as)

definition pre-bisim :: 'a::order list \Rightarrow 'a rexp \Rightarrow 'a rexp \Rightarrow 'a rexp-pairs + 'a rexp-pair set \Rightarrow bool
where
pre-bisim as r s = (\lambda (ws,R).
  (r,s) \in R \land \text{set } ws \subseteq R \land
\( \forall (r, s) \in R, \text{ atoms } r \cup \text{ atoms } s \subseteq \text{ set as} \land \\
(\forall (r, s) \in R - \text{ set ws, (nullable } r \leftrightarrow \text{ nullable } s) \land \\
(\forall a \in \text{ set as, (nderiv } a \text{ r, nderiv } a \text{ s) } \in R)) \)

**Theorem closure-sound:**

**Assumes result:** closure as \((r, s) = \text{ Some}([], R)\) and atoms: atoms \(r \cup \text{ atoms } s \subseteq \text{ set as}\)

**Shows** lang \(r = \text{ lang } s\)

**Proof**

- **Let** \(?test = \text{ While-Combinator.rtrancl-while-test } \% (r, s). \text{ nullable } r = \text{ nullable } s\)
- **Let** \(?step = \text{ While-Combinator.rtrancl-while-step } \% (r, s). \text{ map } (\lambda a. (\text{nderiv } a \text{ r, nderiv } a \text{ s})) \) as

  
  \[
  \{ \text{ fix } st \text{ assume inv: pre-bisim as } r \text{ s } st \text{ and test: } ?test st \\
  \text{ have pre-bisim as } r \text{ s } (?step st) \\
  \text{ proof (cases st) }
  \text{ fix ws } R \text{ assume st } = (\text{ ws, } R) \\
  \text{ with test obtain } r \text{ s t where st: } st = ((r, s) \# t, R) \text{ and nullable } r = \text{ nullable s} \\
  \text{ by (cases ws) auto }
  \text{ with inv show } \text{ ?thesis using atoms-nderiv[of - r] atoms-nderiv[of - s]}
  \text{ unfolding st rtrancl-while-test.simps rtrancl-while-step.simps pre-bisim-def Ball-def }
  \text{ by simp-all blast+ } \\
  \text{ qed } \}

  \text{ moreover from atoms}
  \text{ have pre-bisim as } r \text{ s } ([[(r, s)], \{(r, s)\}) \text{ by (simp add: pre-bisim-def)}
  \text{ ultimately have pre-bisim-ps: pre-bisim as } r \text{ s } ([[], R)]
  \text{ by (rule while-option-rule[of - result[unfolded closure-def rtrancl-while-def]])}
  \text{ then have is-bisimulation as } R \text{ (r, s) } \in R \\
  \text{ by (auto simp: pre-bisim-def is-bisimulation-def)}
  \text{ thus lang } r = \text{ lang } s \text{ by (rule bisim-lang-eq)}

**QED**

### 7.3 Bisimulation-free proof of closure computation

The equivalence check can be viewed as the product construction of two automata. The state space is the reflexive transitive closure of the pair of next-state functions, i.e. derivatives.

**Lemma rtrancl-nderiv-nderivs:**

**Defines** nderivs == foldl \(\% r a. \text{ nderiv a } r\)

**Shows** \({((r, s), (\text{nderiv a r, nderiv a s})) | r s a. a : A}^{*} = \\
{\{(r, s), (\text{nderivs r w, nderivs s w}) | r s w. w : \text{ lists } A}\) (is \(?L = \text{ ?R})

**Proof**

- **Note** [simp] = nderivs-def
  - **Fix** \(r s r' s'\)
  - **Have** \(\{(r, s), (r', s')\) : ?L \(\rightarrow (r, s), (r', s')\) : ?R
  - **Proof** (induction rule: converse-rtrancl-induct2)

**Case refl show** [case by (force intro: foldl.simps(1)[symmetric])]
next
  case step thus ?case by (force intro!: foldl.simps(2)[symmetric])
qed

moreover
  { fix r s r' s' 
    { fix w have \( \forall x \in \text{set } w. x \in A \implies ((r, s), \text{nderivs } r \text{ } w, \text{nderivs } s \text{ } w) : ?L \) 
      proof (induction w rule: rev-induct) 
        case Nil show ?case by simp 
      next 
        case snoc thus ?case by (auto elim!: rtrancl-into-rtrancl) 
      qed 
    } 
    hence \( ((r, s), (r', s')) : ?R = ((r, s), (r', s')) : ?L \) by auto 
  } 
  ultimately show ?thesis by (auto simp add: in-lists-conv-set blast) 
qed

lemma nullable-nderivs: 
  nullable (foldl (%r a. nderv a r) r w) = (w : lang r) 
by (induct w arbitrary: r) (simp-all add: nullable-iff lang-nderiv Deriv-def)

theorem closure-sound-complete:
assumes result: closure as (r, s) = Some(ws,R) 
and atoms: set as = atoms r \cup atoms s 
shows ws = [] \iff lang r = lang s 
proof – 
  have leq: (lang r = lang s) = 
    \( \forall (r',s') \in \{(r0, s0), (\text{nderiv a r0, nderv a s0})| r0 s0 a. a : \text{set as}\} \implies ((r,s)). \) 
    nullable r' = nullable s' 
    by(simp add: atoms rtrancl-nderiv-nderivs Ball-def lang-eq-ext imp-ex nullable-nderivs del:Un-iff) 
  have \{((r,s), (r',s')) \in set (\lambda(p,q). \text{map (\lambdaa. (nderiv a p, nderv a q) as) x}) = 
    \{((r,s), (\text{nderiv a r, nderv a s}) | r s a. a \in \text{set as}\} 
    by auto 
  with atoms rtrancl While-Some[OF result|unfolded closure-def]] 
  show ?thesis by (auto simp add: leq Ball-def split: if-splits) 
qed

7.4 The overall procedure

primrec add-atoms :: 'a rexp \Rightarrow 'a list \Rightarrow 'a list 
where 
  add-atoms Zero = id 
  | add-atoms One = id 
  | add-atoms (Atom a) = List.insert a 
  | add-atoms (Plus r s) = add-atoms s o add-atoms r 
  | add-atoms (Times r s) = add-atoms s o add-atoms r 
  | add-atoms (Star r) = add-atoms r 

lemma set-add-atoms: set (add-atoms r as) = atoms r \cup set as
by (induct r arbitrary: as) auto

**definition** check-eqv :: nat rexp ⇒ nat rexp ⇒ bool where
check-eqv r s =
  (let nr = norm r; ns = norm s; as = add-atoms nr (add-atoms ns [])
   in case closure as (nr, ns) of
     Some([],-) ⇒ True | _ ⇒ False)

**lemma** soundness:
**assumes** check-eqv r s **shows** lang r = lang s

**proof** –
  let ?nr = norm r let ?ns = norm s
  let ?as = add-atoms ?nr (add-atoms ?ns [])
  obtain R where I: closure ?as (?nr,?ns) = Some([],R)
    using assms by (auto simp: check-eqv-def Let-def split:option.splits list.splits)
  then have lang (norm r) = lang (norm s)
    by (rule closure-sound) (auto simp: set-add-atoms dest:subsetD[OF atoms-norm])
  thus lang r = lang s by simp
qed

Test:

**lemma** check-eqv (Plus One (Times (Atom 0) (Star (Atom 0)))) (Star (Atom 0))
by eval

end

8 Regular Expressions as Homogeneous Binary Relations

theory Relation-Interpretation
imports Regular-Exp
begin

primrec rel :: ('a ⇒ ('b * 'b) set) ⇒ 'a rexp ⇒ ('b * 'b) set
where
  rel v Zero = { } |
  rel v One = Id |
  rel v (Atom a) = v a |
  rel v (Plus r s) = rel v r ∪ rel v s |
  rel v (Times r s) = rel v r O rel v s |
  rel v (Star r) = (rel v r)^∗

primrec word-rel :: ('a ⇒ ('b * 'b) set) ⇒ 'a list ⇒ ('b * 'b) set
where
  word-rel v [] = Id |
  word-rel v (a#as) = v a O word-rel v as
lemma word-rel-append:
word-rel v w O word-rel v w′ = word-rel v (w @ w′)
by (rule sym) (induct w, auto)

lemma rel-word-rel: rel v r = (∪ w∈lang r. word-rel v w)
proof (induct r)
case Times thus ?case
  by (auto simp: rel-def word-rel-append conc-def relcomp-UNION-distrib relcomp-UNION-distrib2)
next
case (Star r)
  { fix n
  have (rel v r) ^ n = (∪ w ∈ lang r ^ n. word-rel v w)
  proof (induct n)
    case 0 show ?case by simp
  next
    case Suc n thus ?case
    unfolding relpow.simps relpow-commute[symmetric]
    by (auto simp add: Star conc-def word-rel-append relcomp-UNION-distrib relcomp-UNION-distrib2)
  qed }
thus ?case unfolding rel.simps
  by (force simp: rtrancl-power star-def)
qed auto

Soundness:

lemma soundness:
lang r = lang s ⇒ rel v r = rel v s
unfolding rel-word-rel by auto

end

9 Proving Relation (In)equalities via Regular Expressions

theory Regexp-Method
imports Equivalence-Checking Relation-Interpretation
begin

primrec rel-of-regexp :: ('a * 'a) set list ⇒ nat rexp ⇒ ('a * 'a) set where
rel-of-regexp vs Zero = {}
rel-of-regexp vs One = Id
rel-of-regexp vs (Atom i) = vs ! i
rel-of-regexp vs (Plus r s) = rel-of-regexp vs r ∪ rel-of-regexp vs s
rel-of-regexp vs (Times r s) = rel-of-regexp vs r O rel-of-regexp vs s
rel-of-regexp vs (Star r) = (rel-of-regexp vs r) ^*

lemma rel-of-regexp-rel: rel-of-regexp vs r = rel (λ i. vs ! i) r
primrec rel-eq where
rel-eq (r, s) vs = (rel-of-regexp vs r = rel-of-regexp vs s)

lemma rel-eqI: check-eqv r s ⇒ rel-eq (r, s) vs
unfolding rel-eq.simps rel-of-regexp-rel
by (rule Relation-Interpretation.soundness)
(rule Equivalence-Checking.soundness)

lemmas regexp-reify = rel-of-regexp.simps rel-eq.simps
lemmas regexp-unfold = trancl-unfold-left subset-Un-eq

method-setup regexp = ⟨⟨
let
val regexp-conv = Code-Runtime.static-holds-conv { ctxt = @{context},
consts = [@{const-name Nat.zero-nat-inst.zero-nat}, @{const-name Suc},
@{const-name Zero}, @{const-name One}, @{const-name Atom},
@{const-name Plus}, @{const-name Times}, @{const-name Star},
@{const-name check-eqv}, @{const-name Trueprop}]
⟩⟩
in Scan.succeed (fn ctxt =>
SIMPLE-METHOD’ (TRY o etac @{thm rev-subsetD})
THEN’ (Subgoal.FOCUS-PARAMS (fn {context = ctxt’, ...} =>
TRY (Local-Defs.unfold-tac ctxt’ @{thms regexp-unfold}))
THEN Reification.tac ctxt’ @{thms regexp-reify} NONE 1
THEN rtac @{thm rel-eqI} 1
THEN CONVERSION (regexp-cone ctxt’) 1
THEN rtac TrueI 1 ctxt)⟩⟩
⟩⟩
end

hide-const (open) le-rexp nPlus nTimes norm nullable bisimilar is-bisimulation
closure
pre-bisim add-atoms check-eqv rel word-rel rel-eq

Example:

lemma (r ∪ s ^+) ^* = (r ∪ s) ^*
  by regexp

end

10 Infinite Sequences

theory Seq
imports
Main
~~/src/HOL/Library/Infinite-Set
begin
Infinite sequences are represented by functions of type \( \text{nat} \Rightarrow 'a \). 

**type-synonym** \( 'a \text{ seq} = \text{nat} \Rightarrow 'a \) 

### 10.1 Operations on Infinite Sequences 

An infinite sequence is *linked* by a binary predicate \( P \) if every two consecutive elements satisfy it. Such a sequence is called a \( P \)-chain.

**abbreviation** *(input) chainp :: ('a ⇒ 'a ⇒ bool) ⇒ 'a seq ⇒ bool where*  
\( \text{chainp } P \ S \equiv \forall i. \ P \ (S \ i) \ (S \ (\text{Suc } i)) \) 

Special version for relations.

**abbreviation** *(input) chain :: 'a rel ⇒ 'a seq ⇒bool where*  
\( \text{chain } r \ S \equiv \text{chainp } (λ x y. \ (x, y) \in r) \ S \) 

Extending a chain at the front.

**lemma** \( \text{cons-chainp} \):  
**assumes** \( P \ x \ (S \ 0) \ \text{and } \text{chainp } P \ S \)  
**shows** \( \text{chainp } P \ (\text{case-nat } x \ S) \ (\text{is } \text{chainp } P \ ?S) \)  
**proof**  
\( \text{fix } i \ \text{show } P \ (?S \ i) \ (?S \ (\text{Suc } i)) \ \text{using } \text{assms} \ \text{by } (\text{cases } i) \ \text{simp-all} \) 
\( \text{qed} \) 

Special version for relations.

**lemma** \( \text{cons-chain} \):  
**assumes** \( (x, S \ 0) \in r \ \text{and } \text{chain } r \ S \)  
**shows** \( \text{chain } r \ (\text{case-nat } x \ S) \)  
**using** \( \text{cons-chainp [of } λx y. \ (x, y) \in r, \ \text{OF assms} \) . 

A chain admits arbitrary transitive steps.

**lemma** \( \text{chainp-imp-relpowp} \):  
**assumes** \( \text{chainp } P \ S \)  
**shows** \( (P \ j) \ (S \ i) \ (S \ (i + j)) \))  
**proof**  
\( \text{(induct } i + j \ \text{arbitrary: } j) \)  
\( \text{case } (\text{Suc } n) \ \text{thus } \text{?case using } \text{assms by } (\text{cases } j) \ \text{auto} \) 
\( \text{qed simp} \) 

**lemma** \( \text{chain-imp-relpow} \):  
**assumes** \( \text{chain } r \ S \)  
**shows** \( (S \ i, S \ (i + j)) \in r \ j \))  
**proof**  
\( \text{(induct } i + j \ \text{arbitrary: } j) \)  
\( \text{case } (\text{Suc } n) \ \text{thus } \text{?case using } \text{assms by } (\text{cases } j) \ \text{auto} \) 
\( \text{qed simp} \) 

**lemma** \( \text{chainp-imp-tranclp} \):  
**assumes** \( \text{chainp } P \ S \ \text{and } i < j \)  
**shows** \( P \ j \ (S \ i) \ (S \ j) \))  
**proof**  
\( \text{from } \text{less-imp-Suc-add[OF assms(2)] obtain } n \ \text{where } j = i + \text{Suc } n \ \text{by auto} \)  
\( \text{with } \text{chainp-imp-relpowp[of } P \ S \ \text{Suc } n \ i, \ \text{OF assms(1)} \] \)  
\( \text{show } \?\text{thesis} \)  
\( \text{unfolding } \text{trancl-power[of } (S \ i, S \ j), \text{to-pred]} \)  
\( \text{by force} \)
lemma chain-imp-trancl:
  assumes chain r S and i < j shows \((S i, S j) \in r^*\)
proof –
  from less-imp-Suc-add[OF assms(2)] obtain n where j = i + Suc n by auto
  with chain-imp-relpow[OF assms(1), of i Suc n]
  show ?thesis unfolding trancl-power by force
qed

A chain admits arbitrary reflexive and transitive steps.

lemma chainp-imp-rtranclp:
  assumes chainp P S and i ≤ j shows \(P^{**}(S i) (S j)\)
proof –
  from assms(2) obtain n where j = i + n by (induct j - i arbitrary: j) force+
  with chainp-imp-relpowp[of P S, OF assms(1), of n i]
  show ?thesis by (simp add: relpow-imp-rtrancl)
qed

lemma chain-imp-rtrancl:
  assumes chain r S and i ≤ j shows \((S i, S j) \in r^*\)
proof –
  from assms(2) obtain n where j = i + n by (induct j - i arbitrary: j) force+
  with chain-imp-relpow[OF assms(1), of i n]
  show ?thesis by (simp add: relpow-imp-rtrancl)
qed

If for every \(i\) there is a later index \(f i\) such that the corresponding elements satisfy the predicate \(P\), then there is a \(P\)-chain.

lemma stepfun-imp-chainp'
  assumes \(\forall i \geq n :: \text{nat}. \ f i \geq i \land P (S i) (S (f i))\)
  shows chainp P (\lambda i. S ((f ^^ i) n)) (is chainp P ?T)
proof
  fix i
  from assms have \((f ^^ i) n \geq n\) by (induct i) auto
  with assms THEN spec[of -(f ^^ i) n]]
  show \((? T i) (? T (Suc i))\) by simp
qed

lemma stepfun-imp-chainp:
  assumes \(\forall i \geq n :: \text{nat}. f i > i \land P (S i) (S (f i))\)
  shows chainp P (\lambda i. S ((f ^^ i) n)) (is chainp P ?T)
using stepfun-imp-chainp'[of n f P S] and assms by force

lemma subchain:
  assumes \(\forall i :: \text{nat} > n. \exists j > i. \ P (f i) (f j)\)
  shows \(\exists \varphi. \ \forall i \ j. i < j \longrightarrow \varphi i < \varphi j\) \land \(\forall i. P (f (\varphi i)) (f (\varphi (Suc i))))\)
proof –
  from assms have \(\forall i \in \{i. i > n\}. \exists j > i. \ P (f i) (f j)\) by simp
  from bchoice [OF this] obtain g
  qed
where \(*\): \(\forall i > n. \ g \ i \ > \ i\)

and \(*\): \(\forall i > n. \ P \ (f \ i) \ (f \ (g \ i)) \) by auto

def \[simp\]: \(\varphi \equiv \lambda i. \ (g \ i) \ (S u \ n)\)

from * have ***: \(\forall i > n. \varphi \ i \ > \ n\) by (induct-tac \ i\) auto

then have \(\forall i. \varphi \ i \ < \ (S u \ i)\) using * by (induct-tac \ i\) auto

then have \(\forall i. \ i \ < \ j \ \Longrightarrow \varphi \ i \ < \ \varphi \ j\) by (rule lift-Suc-mono-less)

moreover have \(\forall i. \ P \ (f \ (\varphi \ i)) \ (f \ (\varphi \ (S u \ i)))\) using ** and *** by simp

ultimately show \(?\)thesis by blast

qed

If for every \(i\) there is a later index \(j\) such that the corresponding elements satisfy the predicate \(P\), then there is a \(P\)-chain.

lemma steps-imp-chainp':

assumes \(\forall i \geq n::nat. \ \exists j \geq i. \ P \ (S u \ i) \ (S j)\) shows \(\exists T. \ chainp \ P \ T\)

proof –

from assms have \(\forall i \in \{i. \ i \geq n\}. \ \exists j \geq i. \ P \ (S u \ i) \ (S j)\) by auto

from bchoice [OF this]

obtain \(f\) where \(\forall i \geq n. \ f \ i \ \geq \ i \ \wedge \ P \ (S u \ i) \ (S (f \ i))\) by auto

from stepfun-imp-chainp' [of \(n\ \ f\ \ P\ \ S\), OF this] show \(?\)thesis by fast

qed

lemma steps-imp-chainp:

assumes \(\forall i \geq n::nat. \ \exists j > i. \ P \ (S u \ i) \ (S j)\) shows \(\exists T. \ chainp \ P \ T\)

using steps-imp-chainp' [of \(n\ \ P\ \ S\) and assms] by force

10.2 Predicates on Natural Numbers

If some property holds for infinitely many natural numbers, obtain an index function that points to these numbers in increasing order.

locale infinitely-many =

fixes \(p :: nat \Rightarrow bool\)

assumes infinite: \(INFM \ j. \ p \ j\)

begin

lemma inf: \(\exists j \geq i. \ p \ j\) using infinite[unfolded INFM-nat-le\] by auto

fun index :: nat seq where

index \(0\) = (LEAST \(n\). \(p\ \ n\))

| index \((S u \ n)\) = (LEAST \(k. \ p \ \ k \ \wedge \ k \ > \ index \ n)\)

lemma index-p: \(p \ (index \ n)\)

proof (induct \(n\))

case \(0\)

from inf obtain \(j\) where \(p \ j\) by auto

with LeastI[of \(p\ \ j\)] show \(?\)case by auto

next

case \((S u \ n)\)

from inf obtain \(k\) where \(k \ \geq \ S u \ (index \ n) \ \wedge \ p \ k\) by auto

with LeastI[of \(\lambda k. \ p \ \ k \ \wedge \ k \ > \ index \ n \ k\)] show \(?\)case by auto
qed

lemma index-ordered: index n < index (Suc n)
proof -
  from inf obtain k where k ≥ Suc (index n) ∧ p k by auto
  with LeastI[of λ k. p k ∧ k > index n k] show ?thesis by auto
qed

lemma index-not-p-between:
  assumes i1: index n < i
      and i2: i < index (Suc n)
  shows ¬ p i
proof -
  from not-less-Least[OF i2[simplified]] i1 show ?thesis by auto
qed

lemma index-ordered-le:
  assumes i ≤ j shows index i ≤ index j
proof -
  from assms have j = i + (j - i) by auto
  then obtain k where j: j = index i + k by auto
  have index i ≤ index (i + k)
    proof (induct k)
      case (Suc k)
      with index-ordered[of i + k]
      show ?case by auto
    qed simp
    thus ?thesis unfolding j .
qed

lemma index-surj:
  assumes k ≥ index l
  shows ∃ i j. k = index i + j ∧ index i + j < index (Suc i)
proof -
  from assms have k = index l + (k - index l) by auto
  then obtain u where k: k = index l + u by auto
  show ?thesis unfolding k
    proof (induct u)
      case 0
      show ?case
        by (intro exI conjI, rule refl, insert index-ordered[of l], simp)
    next
      case (Suc u)
      then obtain i j
        where lu: index l + u = index i + j and lt: index i + j < index (Suc i)
        by auto
      hence index l + u < index (Suc i) by auto
      show ?case
        proof (cases index l + (Suc u) = index (Suc i))
case False
  show ?thesis
    by (rule exI[of - i], rule exI[of - Suc j], insert lt lt False, auto)
next
case True
  show ?thesis
    by (rule exI[of - Suc i], rule exI[of - 0], insert True index-ordered[of Suc i],
      auto)
qed
qed
qed

lemma index-ordered-less:
  assumes i < j shows index i < index j
proof –
  from assms have Suc i ≤ j by auto
  from index-ordered-le[OF this] have index (Suc i) ≤ index j .
  with index-ordered[of i] show ?thesis by auto
qed

lemma index-not-p-start: assumes i: i < index 0 shows ¬ p i
proof –
  from i[simplified index.simps] have i < Least p .
  from not-less-Least[OF this] show ?thesis .
qed
end

10.3 Assembling Infinite Words from Finite Words

Concatenate infinitely many non-empty words to an infinite word.

fun inf-concat-simple :: (nat ⇒ nat) ⇒ nat ⇒ (nat × nat) where
  inf-concat-simple f 0 = (0, 0)
| inf-concat-simple f (Suc n) = (let (i, j) = inf-concat-simple f n in
  if Suc j < f i then (i, Suc j)
    else (Suc i, 0))

lemma inf-concat-simple-add:
  assumes ck: inf-concat-simple f k = (i, j)
    and jl: j + l < f i
  shows inf-concat-simple f (k + l) = (i, j + l)
using jl
proof (induct l)
case 0
  thus ?case using ck by simp
next
case (Suc l)
hence $c$: \( \inf\text{-}concat\text{-}simple\ f\ (k + l) = (i,\ j + l) \) by \textit{auto}

\textbf{show} \ ?\text{case}

\text{by} \ (\textit{simp add: } c, \ \textit{insert} \ Suc(2), \ \textit{auto})

\textbf{qed}

\textbf{lemma} \ \inf\text{-}concat\text{-}simple\text{-}surj-zero: $\exists\ k. \ \inf\text{-}concat\text{-}simple\ f\ k = (i,0)$

\textbf{proof} (\textit{induct } i)

\begin{enumerate}
\item \textbf{case } 0
\item \textbf{show} \ ?\text{case}
\text{by} \ (\textit{rule exI[of - 0]}, \ \textit{simp})
\end{enumerate}

\textbf{next}

\begin{enumerate}
\item \textbf{case } (Suc \ i)
\item \textbf{then} \ \textbf{obtain} \ k \ \textbf{where} \ ck: \ \inf\text{-}concat\text{-}simple\ f\ k = (i,0) \ \textbf{by} \ \textit{auto}
\item \textbf{show} \ ?\text{case}
\textbf{proof} (\textit{cases } f\ i)
\item \textbf{case } 0
\item \textbf{show} \ ?\text{thesis}
\text{by} \ (\textit{rule exI[of - Suc k]}, \ \textit{simp add: } ck\ 0)
\end{enumerate}

\textbf{qed}

\textbf{lemma} \ \inf\text{-}concat\text{-}simple\text{-}surj:

\textbf{assumes} \ j < f \ i

\textbf{shows} \ $\exists\ k. \ \inf\text{-}concat\text{-}simple\ f\ k = (i,j)$

\textbf{proof} –

\begin{enumerate}
\item \textbf{from} \ \textbf{assms} \ \textbf{have} \ j: \ 0 + j < f \ i \ \textbf{by} \ \textit{auto}
\item \textbf{from} \ \textit{inf\text{-}concat\text{-}simple\text{-}surj\text{-}zero} \ \textbf{obtain} \ k \ \textbf{where} \ \inf\text{-}concat\text{-}simple\ f\ k = (i,0)
\item \textbf{by} \ \textit{auto}
\item \textbf{from} \ \textit{inf\text{-}concat\text{-}simple\text{-}add}[OF \ this, \ OF\ j] \ \textbf{show} \ ?\text{thesis} \ \textbf{by} \ \textit{auto}
\end{enumerate}

\textbf{qed}

\textbf{lemma} \ \inf\text{-}concat\text{-}simple\text{-}mono:

\textbf{assumes} \ k ≤ k’

\textbf{shows} \ \textit{fst} \ (\textit{inf\text{-}concat\text{-}simple} \ f\ k) ≤ \textit{fst} \ (\textit{inf\text{-}concat\text{-}simple} \ f\ k’)

\textbf{proof} –

\begin{enumerate}
\item \textbf{from} \ \textbf{assms} \ \textbf{have} \ k’: \ k + (k’ - k) \ \textbf{by} \ \textit{auto}
\item \textbf{then} \ \textbf{obtain} \ l \ \textbf{where} \ k’: \ k’ = k + l \ \textbf{by} \ \textit{auto}
\item \textbf{show} \ ?\text{thesis} \ \textbf{unfolding} \ k’
\item \textbf{proof} (\textit{induct } l)
\item \textbf{case} \ (Suc \ l)
\item \textbf{obtain} \ i \ j \ \textbf{where} \ ckl: \ \textit{inf\text{-}concat\text{-}simple} \ f\ (k+l) = (i,j) \ \textbf{by} \ (\textit{cases} \ \textit{inf\text{-}concat\text{-}simple} \ f\ (k+l), \ \textit{auto})
\item \textbf{with} \ Suc \ \textbf{have} \ \textit{fst} \ (\textit{inf\text{-}concat\text{-}simple} \ f\ k) ≤ i \ \textbf{by} \ \textit{auto}
\item \textbf{also} \ \textbf{have} \ ... \ ≤ \ \textit{fst} \ (\textit{inf\text{-}concat\text{-}simple} \ f\ (k + Suc \ l))
\end{enumerate}

50
by (simp add: ckl)
finally show ?case .
qed simp
qed

fun inf-concat :: (nat ⇒ nat) ⇒ nat ⇒ nat × nat where
inf-concat n 0 = (LEAST j. n j > 0, 0)
| inf-concat n (Suc k) = (let (i, j) = inf-concat n k in (if Suc j < n i then (i, Suc j) else (LEAST i'. i' > i ∧ n i' > 0, 0)))

lemma inf-concat-bounds:
assumes inf: INFM i. n i > 0
and res: inf-concat n k = (i,j)
shows j < n i
proof (cases k)
  case 0
  with res have i: i = (LEAST i. n i > 0) and j: j = 0 by auto
  from inf[unfolded INFM-nat-le] obtain i' where i': 0 < n i' by auto
  have 0 < n (LEAST i. n i > 0)
    by (rule LeastI, rule i')
  with i j show ?thesis by auto
next
  case (Suc k')
  obtain i' j' where res': inf-concat n k' = (i' j') by force
  note res = res[unfolded Suc inf-concat.simps res'[Let-def split]
  show ?thesis
    proof (cases Suc j' < n i')
      case True
      with res show ?thesis by auto
    next
      case False
      with res have i: i = (LEAST f. i' < f ∧ 0 < n f) and j: j = 0 by auto
      from inf[unfolded INFM-nat] obtain f where f: i' < f ∧ 0 < n f by auto
      have 0 < n (LEAST f. i' < f ∧ 0 < n f)
        using LeastI[of λ f. i' < f ∧ 0 < n f, OF f]
        by auto
      with i j show ?thesis by auto
    qed
    qed

lemma inf-concat-add:
assumes res: inf-concat n k = (i,j)
and j: j + m < n i
shows inf-concat n (k + m) = (i,j+m)
using j
proof (induct m)
  case 0 show ?case using res by auto
next
  case (Suc m)
  hence inf-concat n (k + m) = (i, j + m) by auto
with Suc(2)
show ?case by auto
qed

lemma inf-concat-step:
  assumes res: inf-concat n k = (i, j)
  and j: Suc (j + m) = n i
  shows inf-concat n (k + Suc m) = (LEAST i'. i' > i ∧ 0 < n i', 0)
proof –
  from j have j + m < n i by auto
note res = inf-concat-add[OF res, OF this]
  show ?thesis by (simp add: res j)
qed

lemma inf-concat-surj-zero:
  assumes 0 < n i
  shows ∃k. inf-concat n k = (i, 0)
proof –
  { fix l
    have ∀ j. j < l ∧ 0 < n j → (∃ k. inf-concat n k = (j, 0))
    proof (induct l)
      case 0
      thus ?case by auto
    next
      case (Suc l)
      show ?case
      proof (intro allI impI, elim conjE)
        fix j
        assume j: j < Suc l and nj: 0 < n j
        show ∃ k. inf-concat n k = (j, 0)
        proof (cases j < l)
          case True
          from Suc[THEN spec[of - j]] True nj show ?thesis by auto
          next
          case False
          with j have j: j = l by auto
          show ?thesis
          proof (cases ∃ j'. j' < l ∧ 0 < n j')
            case False
            have l: (LEAST i. 0 < n i) = l
            proof (rule Least-equality, rule nj[unfolded j])
              fix l'
              assume 0 < n l'
              with False have ¬ l' < l by auto
              thus l ≤ l' by auto
          qed
        qed
      qed
    qed
  }
shows 

\[ \text{thesis} \]

by (rule exI[of - 0], simp add: j)

next

case True

then obtain lll where lll: lll < l and nlll: lll < n lll by auto

then obtain ll where l: l = Suc ll by (cases l, auto)

from lll l have lll: lll = ll - (ll - lll) by auto

let \(?l'\) = LEAST d. \(0 < n (ll - d)\)

have nl': \(?l'\) < n (ll - nl')

proof (rule LeastI)

show \(?l'\) < n (ll - (ll - lll)) using lll nlll by auto

qed

with Suc[THEN spec[of - ll - \(?l'\)]] obtain k where:

inf-concat n k = (ll - \(?l'\), 0) unfolding l by auto

from nl' obtain off where off: Suc (0 + off) = n (ll - \(?l'\)) by (cases n (ll - \(?l'\)), auto)

have ll: \(?l'\) = l unfolding l

proof (rule Least-equality)

show ll - \(?l'\) < Suc ll ∧ \(0 < n (ll - \(?l'\))\) using nj[unfolded j ll] by simp

next

fix l'

assume ass: ll - \(?l'\) < l' ∧ \(0 < n l'\)

show Suc ll ≤ l'

proof (rule ccontr)

assume not: ¬ \(?thesis\)

hence l' ≤ ll by auto

hence ll = l' + (ll - l') by auto

then obtain k where ll: ll = l' + k by auto

from ass have l' + k - \(?l'\) < l' unfolding ll by auto

hence kl': k < \(?l'\) by auto

have 0 < n (ll - k) using ass unfolding ll by simp

from Least-le[of \(\lambda k. 0 < n (ll - k)\), OF this] kl'

show False by auto

qed

qed

show \(?thesis\) unfolding j

by (rule exI[of - k + Suc off], unfold id ll, simp)

qed

qed

qed
lemma inf-concat-surj:
assumes \( j : j < n \ i \)
shows \( \exists k. \infcon\ n \ k = (i, j) \)
proof
  from \( j \) have \( 0 < n \ i \) by auto
  from inf-concat-surj-zero[of \( n \), OF this] obtain \( k \) where \( \infcon\ n \ k = (i,0) \) by auto
  from inf-concat-add[OF this, of \( j \)] \( j \) show \( \text{thesis} \) by auto
qed

lemma inf-concat-mono:
assumes \( \text{inf} : \text{INFM} \ i. \ n \ i > 0 \)
and \( \text{resk} : \infcon\ n \ k = (i, j) \)
and \( \text{reskp} : \infcon\ n \ k' = (i', j') \)
and \( \text{lt} : i < i' \)
shows \( k < k' \)
proof
  note \( \text{bounds} = \infcon\text{-bounds}[\text{OF inf}] \)
  { assume \( k' \leq k \)
    hence \( k = k' + (k - k') \) by auto
    then obtain \( l \) where \( k = k' + l \) by auto
    have \( i' \leq \text{fst} (\infcon\ n (k' + l)) \)
    proof (induct \( l \))
      case 0
      with \( \text{reskp} \) show \( \text{case} \) by auto
    next
      case (Suc \( l \))
      obtain \( i'' j'' \) where \( l : \infcon\ n (k' + l) = (i'' j'') \) by force
      with Suc have \( \text{one} : i' \leq i'' \) by auto
      from \( \text{bounds}[\text{OF l}] \) have \( j'', j'' < n i'' \) by auto
      show \( \text{case} \)
      proof (cases Suc \( j'' < n i'' \))
        case True
        show \( \text{thesis} \) by (simp add: \( l \) True one)
      next
        case False
        let \( \approx i = \text{LEAST} i'. i'' < i' \land 0 < n i' \)
        from \( \text{inf}[\text{unfolded INFM-nat}] \) obtain \( k \) where \( i'' < k < 0 < n k \) by auto
        from LeastI[of \( \lambda k. i'' < k \land 0 < n k \), OF this] have \( i'' < \approx i \) by auto
        with one show \( \text{thesis} \) by (simp add: \( l \) False)
      qed
    qed
    with \( \text{resk} \) \( \text{lt} \) have \( \text{False} \) by auto
  }
  thus \( \text{thesis} \) by arith
qed
lemma inf-concat-Suc:
assumes inf: INFM i. n i > 0
and f: \( \bigwedge i. f i (n i) = f (Suc i) 0 \)
and resk: inf-concat n k = (i, j)
and ressk: inf-concat n (Suc k) = (i', j')
shows f i' j' = f i (Suc j)
proof –
   note bounds = inf-concat-bounds[OF inf]
from bounds[OF resk] have j: j < n i .
show ?thesis
proof (cases Suc j < n i)
   case True
   with ressk resk show ?thesis by simp
next
case False
let ?p = \( \lambda i'. i < i' \land 0 < n i' \)
let ?i' = LEAST i'. ?p i'
from False j have id: Suc (j + 0) = n i by auto
from inf-concat-step[OF resk, OF id] ressk
have i'; i' = ?i' and j'; j' = 0 by auto
from id have j: Suc j = n i by simp
from inf[unfolded INFM-nat] obtain k where ?p k by auto
from LeastI[of ?p, OF this] have ?p ?i'.
   hence ?i' = Suc i + (?i' - Suc i) by simp
then obtain d where ii': ?i' = Suc i + d by auto
from not-less-Least[of ?p, unfolded ii'] have d': \( \lambda d'. d' < d \Rightarrow n (Suc i + d') = 0 \) by auto
have f (Suc i) 0 = f ?i' 0 unfolding ii' using d'
proof (induct d)
   case 0
   show ?case by simp
next
case (Suc d)
   hence f (Suc i) 0 = f (Suc i + d) 0 by auto
   also have ... = f (Suc (Suc i + d)) 0
   unfolding f[symmetric]
   using Suc(2)[of d] by simp
finally show ?case by simp
qed
thus ?thesis unfolding i' j' j f by simp
qed
qed
end
11 Enumerations of Well-Ordered Sets in Increasing Order

theory Least-Enum
imports Main
begin

locale infinitely-many1 =
  fixes P :: 'a :: wellorder ⇒ bool
  assumes infm: ∀i. ∃j>i. P j
begin

Enumerate the elements of a well-ordered infinite set in increasing order.

fun enum :: nat ⇒ 'a where
enum 0 = (LEAST n. P n) |
enum (Suc i) = (LEAST n. n > enum i ∧ P n)

lemma enum-mono:
  shows enum i < enum (Suc i)
  using infm by (cases i, auto) (metis (lifting) LeastI)+

lemma enum-less:
i < j ⇒ enum i < enum j
  using enum-mono by (metis lift-Suc-mono-less)

lemma enum-P:
  shows P (enum i)
  using infm by (cases i, auto) (metis (lifting) LeastI)+
end

locale infinitely-many2 =
  fixes P :: 'a :: wellorder ⇒ 'a ⇒ bool
  and N :: 'a
  assumes infm: ∀i≥N. ∃j>i. P i j
begin

Enumerate the elements of a well-ordered infinite set that form a chain w.r.t. a given predicate P starting from a given index N in increasing order.

fun enumchain :: nat ⇒ 'a where
enumchain 0 = N |
enumchain (Suc n) = (LEAST m. m > enumchain n ∧ P (enumchain n) m)

lemma enumchain-mono:
  shows N ≤ enumchain i ∧ enumchain i < enumchain (Suc i)
proof (induct i)
  case 0
  have enumchain 0 ≥ N by simp
moreover then have \( \exists m > \text{enumchain } 0 \). \( P (\text{enumchain } 0) \) \( m \) using \( \text{in} \text{fm} \) by \( \text{blast} \)
ultimately show \( ?\text{case by auto (metis (lifting) LeastI)} \)
next
case \( (\text{Suc } i) \)
then have \( N \leq \text{enumchain } (\text{Suc } i) \) by \( \text{auto} \)
moreover then have \( \exists m > \text{enumchain } (\text{Suc } i) \). \( P (\text{enumchain } (\text{Suc } i)) \) \( m \) using \( \text{in} \text{fm} \) by \( \text{blast} \)
ultimately show \( ?\text{case by (auto) (metis (lifting) LeastI)} \)
qed

lemma \text{enumchain-chain}:
shows \( P (\text{enumchain } i) (\text{enumchain } (\text{Suc } i)) \)
proof \( \text{(cases } i) \)
case \( 0 \)
moreover have \( \exists m > \text{enumchain } 0 \). \( P (\text{enumchain } 0) \) \( m \) using \( \text{in} \text{fm} \) by \( \text{auto} \)
ultimately show \( ?\text{thesis by auto (metis (lifting) LeastI)} \)
next
case \( (\text{Suc } i) \)
moreover have \( \text{enumchain } (\text{Suc } i) > N \) using \( \text{enumchain-mono} \) by \( \text{metis le-less-trans} \)
moreover then have \( \exists m > \text{enumchain } (\text{Suc } i) \). \( P (\text{enumchain } (\text{Suc } i)) \) \( m \) using \( \text{in} \text{fm} \) by \( \text{auto} \)
ultimately show \( ?\text{thesis by (auto) (metis (lifting) LeastI)} \)
qed

end

12 Binary Predicates Restricted to Elements of a Given Set

theory \text{Restricted-Predicates}
imports \text{Main}
begin

definition \text{restrict-to} :: \( ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool}) \) where
\text{restrict-to} \( P \ A = (\lambda x \ y. \ x \in A \land y \in A \land P \ x \ y) \)
definition \text{reflp-on} :: \( ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \) where
\text{reflp-on} \( P \ A \longleftrightarrow (\forall a \in A. \ P \ a \ a) \)
definition \text{transp-on} :: \( ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \) where
\text{transp-on} \( P \ A \longleftrightarrow (\forall x \in A. \ \forall y \in A. \ \forall z \in A. \ P \ x \ y \land P \ y \ z \longrightarrow P \ x \ z) \)
definition \text{total-on} :: \( ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \) where
\text{total-on} \( P \ A \longleftrightarrow (\forall x \in A. \ \forall y \in A. \ x \ = \ y \lor P \ x \ y \lor P \ y \ x) \)
abbreviation strict $P \equiv \lambda x y. \ P x y \land \neg (P y x)$

abbreviation chain-on $P f A \equiv \forall i. \ f i \in A \land P (f i) (f (Suc i))$

abbreviation incomparable $P \equiv \lambda x y. \ \neg P x y \land \neg P y x$

abbreviation antichain-on $P f A \equiv \forall (i::nat) j. \ f i \in A \land (i < j \rightarrow \text{incomparable } P (f i) (f j))$

lemma strict-reflclp-conv [simp]:
strict $(P=\!) = \text{strict } P$ by auto

lemma reflp-onI [Pure.intro]:
$(\forall a. \ a \in A \implies P a a) \implies \text{reflp-on } P A$
unfolding reflp-on-def by blast

lemma transp-onI [Pure.intro]:
$(\forall x y z. \ [x \in A; \ y \in A; \ z \in A; \ P x y; \ P y z] \implies P x z) \implies \text{transp-on } P A$
unfolding transp-on-def by blast

lemma total-onI [Pure.intro]:
$(\forall x y. \ [x \in A; \ y \in A] \implies x = y \lor P x y \lor P y x) \implies \text{total-on } P A$
unfolding total-on-def by blast

lemma reflp-on-reflclp-simp [simp]:
assumes reflp-on $P A$ and $a \in A$ and $b \in A$
shows $P=\!\!\!\!\!\!\!a b = P a b$
using assms by (auto simp: reflp-on-def)

lemma reflp-on-reflclp:
reflp-on $(P=\!) \ A$
by (auto simp: reflp-on-def)

lemma reflp-on-converse-simp [simp]:
reflp-on $P^{\sim\sim} A \iff \text{reflp-on } P A$
by (auto simp: reflp-on-def)

lemma transp-on-converse:
transp-on $P A \implies \text{transp-on } P^{\sim\sim} A$
unfolding transp-on-def by blast

lemma transp-on-converse-simp [simp]:
transp-on $P^{\sim\sim} A \iff \text{transp-on } P A$
unfolding transp-on-def by blast

lemma transp-on-reflclp:
transp-on $P A \implies \text{transp-on } P=\!\!\!\!\!\!\! A$
unfolding transp-on-def by blast
lemma transp-on-strict:
\[\text{transp-on } P \ A \Rightarrow \text{transp-on} \ (\text{strict } P) \ A\]
unfolding transp-on-def by blast

lemma reflp-on-subset:
\[A \subseteq B \Rightarrow \text{reflp-on } P \ B \Rightarrow \text{reflp-on } P \ A\]
by (auto simp: reflp-on-def)

lemma transp-on-subset:
\[A \subseteq B \Rightarrow \text{transp-on } P \ B \Rightarrow \text{transp-on } P \ A\]
by (auto simp: transp-on-def)

definition wfp-on :: 
\[\forall a. \forall a. \text{bool} \Rightarrow \forall a. \set \Rightarrow \text{bool}\]
where
\[\text{wfp-on } P \ A \iff \neg (\exists f. \forall i. f \ i \in A \land P (f \ (\text{Suc } i)) (f \ i))\]

definition inductive-on ::
\[\forall a. \forall a. \text{bool} \Rightarrow \forall a. \set \Rightarrow \text{bool}\]
where
\[\text{inductive-on } P \ A \iff (\forall Q. (\forall y \in A. (\forall x \in A. P \ x \ y \Rightarrow Q \ x) \Rightarrow Q \ y) \Rightarrow (\forall x \in A. Q \ x))\]

lemma inductive-onI [Pure.intro]:
assumes \[\forall x. ((\forall y. (\forall x. (P \ x \ y \Rightarrow Q \ x) \Rightarrow Q \ y)) \Rightarrow Q \ y)\]
shows inductive-on P A using assms unfolding inductive-on-def by metis

If \(P\) is well-founded on \(A\) then every non-empty subset \(Q\) of \(A\) has a minimal element \(z\) w.r.t. \(P\), i.e., all elements that are \(P\)-smaller than \(z\) are not in \(Q\).

lemma wfp-on-imp-minimal:
assumes wfp-on P A
shows \[\forall Q \ x. (x \in Q \land Q \subseteq A \Rightarrow (\exists z \in Q. \forall y. P \ y \ z \Rightarrow y \notin Q))\]
proof (rule ccontr)
assume \[\neg \thesis\]
then obtain \(Q \ x\) where \[*: x \in Q \land Q \subseteq A\]
and \[\forall z. \exists y. z \in Q \Rightarrow P y z \land y \in Q \land Q y\]
by metis
from choice [OF this(3)] obtain \(f\)
where \[**: \forall x \in Q. P \ (f \ x) \ x \land f \ x \in Q\] by blast
let \[?S = \lambda i. (f ^^ i) \ x\]
have \[***: \forall i. ?S i \in Q\]
proof
fix \(i\) show \[?S i \in Q\] by (induct \(i\)) (auto simp: \(*\) \ **) qed
then have \[\forall i. \ ?S i \in A\] using \(*\) by blast
moreover have \[\forall i. P \ (?S \ (\text{Suc } i)) (?S i)\]
proof
fix \(i\) show \[P \ (?S \ (\text{Suc } i)) \ (?S i)\]
by (induct \(i\)) (auto simp: \(*\) \ ** \ ***)
ultimately have \( \forall i. \ ?S \ i \in A \land P \ (?S \ (Suc \ i)) \ (\ ?S \ i) \) by blast

unfolding wfp-on-def by fast

\[ \text{lemma minimal-imp-inductive-on:} \]
\[ \text{assumes} \ \forall Q \ x. \ x \in Q \land Q \subseteq A \rightarrow (\exists z \in Q. \ \forall y. \ P \ y \ z \rightarrow y \notin Q) \]
\[ \text{shows} \ \text{inductive-on} \ P \ A \]

\[ \text{proof (rule ccontr)} \]
\[ \text{assume} \ \neg \ ?thesis \]
\[ \text{then obtain} \ Q \ x \]
\[ \text{where} \ \star: \ \forall y \in A. \ (\forall x \in A. \ P \ x \ y \rightarrow Q \ x) \rightarrow Q \ y \]
\[ \text{and} \ \star\star: \ x \in A \rightarrow \neg Q \ x \]
\[ \text{by} \ \{\text{auto simp: inductive-on-def}\} \]
\[ \text{let} \ \varnothing = \{x \in A. \ \neg Q \ x\} \]
\[ \text{from} \ \star\star \ \text{have} \ x \in \varnothing \text{ by auto} \]
\[ \text{moreover have} \ \varnothing \subseteq A \text{ by auto} \]
\[ \text{ultimately obtain} \ z \text{ where} \ z \in \varnothing \]
\[ \text{and} \ \min: \ \forall y. \ P \ y \ z \rightarrow y \notin \varnothing \]
\[ \text{using} \ \text{assms [THEN spec [of - \ ?Q], THEN spec [of - x]] by blast} \]
\[ \text{from} \ \star \ \text{have} \ z \in A \text{ and } \neg Q \ z \text{ by auto} \]
\[ \text{with } \star \ \text{obtain} \ y \text{ where} \ y \in A \text{ and } P \ y \ z \text{ and } \neg Q \ y \text{ by auto} \]
\[ \text{then have} \ y \in \varnothing \text{ by auto} \]
\[ \text{with } \langle P \ y \ z \rangle \text{ and } \min \text{ show } \text{False by auto} \]

\[ \text{qed} \]

\[ \text{lemmas} \ wfp-on-imp-inductive-on = \]
\[ \text{wfp-on-imp-minimal [THEN minimal-imp-inductive-on]} \]

\[ \text{lemma inductive-on-induct [consumes 2, case-names less, induct pred: inductive-on]}: \]
\[ \text{assumes} \ \text{inductive-on} \ P \ A \text{ and } x \in A \]
\[ \text{and } \Lambda y. \ [y \in A; \ \Lambda x. \ [x \in A; P \ x \ y] \implies Q \ x] \implies Q \ y \]
\[ \text{shows} \ Q \ x \]
\[ \text{using} \ \text{assms unfolding inductive-on-def by metis} \]

\[ \text{lemma inductive-on-imp-wfp-on}: \]
\[ \text{assumes} \ \text{inductive-on} \ P \ A \]
\[ \text{shows} \ \text{wfp-on} \ P \ A \]

\[ \text{proof} - \]
\[ \text{let} \ ?Q = \lambda x. \ \neg (\exists f. \ f \ 0 = x \land (\forall i. \ f \ i \in A \land P \ (f \ (Suc \ i)) \ (f \ i))) \]
\[ \{\ \text{fix x assume x \in A} \}
\[ \text{with } \text{assms have} \ ?Q \ x \]
\[ \text{proof (induct rule: inductive-on-induct) } \]
\[ \text{fix } y \text{ assume } y \in A \text{ and IH: } \Lambda x. \ x \in A \implies P \ x \ y \implies ?Q \ x \]
\[ \text{show } ?Q \ y \]
\[ \text{proof (rule ccontr) } \]
\[ \text{assume } \neg ?Q \ y \]
\[ \text{then obtain } f \text{ where } \star: \ f \ 0 = y \]

60
∀ i. f i ∈ A ∧ P (f (Suc i)) then have P (f (Suc i)) by auto

then have P (f (Suc 0)) and f (Suc 0) ∈ A by auto

with IH and * have ?Q (f (Suc 0)) by auto

with * show False by auto

qed

then show thesis unfolding wfp-on-def by blast

qed

definition antisym on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
antisym on P A ←→ (∀ a ∈ A. ∀ b ∈ A. P a b ∧ P b a → a = b)

lemma antisym on I [Pure.intro]:
(∀ a b. [a ∈ A; b ∈ A; P a b; P b a] → a = b) → antisym on P A
by (auto simp: antisym on-def)

lemma antisym on-refl cl p [simp]:
antisym on P = antisym on P A
by (auto simp: antisym on-def)

definition qo on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
qo on P A ←→ reflp on P A ∧ transp on P A

definition irreflp on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
irreflp on P A ←→ (∀ a ∈ A. ¬ P a a)

definition po on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
po on P A ←→ (irreflp on P A ∧ transp on P A)

lemma po on I [Pure.intro]:
[irreflp on P A; transp on P A] → po on P A
by (auto simp: po on-def)

lemma irreflp on I [Pure.intro]:
(∀ a ∈ A ⇒ ¬ P a a) → irreflp on P A
unfolding irreflp on-def by blast

lemma irreflp on converse:
irreflp on P A ⇒ irreflp on P ^ −1−1 A
unfolding irreflp on-def by blast

lemma irreflp on converse simp [simp]:
irreflp on P ^ −1−1 A ←→ irreflp on P A
by (auto simp: irreflp on-def)

lemma po on converse simp [simp]:
po on P ^ −1−1 A ←→ po on P A
by (simp add: po on-def)
lemma po-on-imp-go-on:
    po-on P A → go-on (P\text{==}) A
unfolding po-on-def go-on-def
by (metis reflp-on-reflclp transp-on-reflclp)

lemma po-on-imp-irreflp-on:
    po-on P A → irreflp-on P A
by (auto simp: po-on-def)

lemma po-on-imp-transp-on:
    po-on P A → transp-on P A
by (auto simp: po-on-def)

lemma irreflp-on-subset:
    assumes A ⊆ B and irreflp-on P B
shows irreflp-on P A
using assms by (auto simp: irreflp-on-def)

lemma po-on-subset:
    assumes A ⊆ B and po-on P B
shows po-on P A
using transp-on-subset and irreflp-on-subset and assms
unfolding po-on-def by blast

lemma transp-on-irreflp-on-imp-antisym-on:
    assumes transp-on P A and irreflp-on P A
shows antisym-on (P\text{==}) A
proof
  fix a b assume a ∈ A
    and b ∈ A and P\text{==} a b and P\text{==} b a
  show a = b
proof (rule ccontr)
  assume a ≠ b
    with (P\text{==} a b) and \(\text{=}\)P\text{==} b a) have P a b and P b a by auto
    with (transp-on P A) and \(a ∈ A\) and \(b ∈ A\) have P a a unfolding transp-on-def by blast
    with (irreflp-on P A) and \(a ∈ A\) show False unfolding irreflp-on-def by blast
qed
qed

lemma po-on-imp-antisym-on:
    assumes po-on P A
shows antisym-on (P\text{==}) A
using transp-on-irreflp-on-imp-antisym-on \(\text{af} P A\)
  and assms
unfolding po-on-def by blast

lemma strict-reflclp [simp]:
assumes \( x \in A \) and \( y \in A \)
and transp-on \( P A \) and irreflp-on \( P A \)
shows strict \( (\text{strict } P \equiv \) \( x \neq y \))
using assms unfolding transp-on-def irreflp-on-def
by blast

lemma \( \text{qo-on-imp-reflp-on} \):
\( \text{qo-on } P A \imp \text{reflp-on } P A \)
by (auto simp: qo-on-def)

lemma \( \text{qo-on-imp-transp-on} \):
\( \text{qo-on } P A \imp \text{transp-on } P A \)
by (auto simp: qo-on-def)

lemma \( \text{qo-on-subset} \):
\( A \subseteq B \imp \text{qo-on } P B \imp \text{qo-on } P A \)
unfolding qo-on
using reflp-on-subset
and transp-on-subset by blast

Quasi-orders are instances of the preorder class.

lemma \( \text{qo-on-UNIV-conv} \):
\( \text{qo-on } P \text{ UNIV} \iff \text{class.preorder } P \ (\text{strict } P) \ (\text{is } \text{lhs} = \text{rhs}) \)
proof
assume \( \text{lhs} \) then show \( \text{rhs} \)
  unfolding qo-on-def class.preorder-def
  and qo-on-imp-reflp-on [of P UNIV]
  and qo-on-imp-transp-on [of P UNIV]
  by (auto simp: reflp-on-def, blast)
next
assume \( \text{rhs} \) then show \( \text{lhs} \)
  unfolding class.preorder-def
  by (auto simp: qo-on-def reflp-on-def transp-on-def)
qed

lemma \( \text{wfp-on-iff-inductive-on} \):
\( \text{wfp-on } P A \iff \text{inductive-on } P A \)
by (blast intro: inductive-on-imp-wfp-on wfp-on-imp-inductive-on)

lemma \( \text{wfp-on-iff-minimal} \):
\( \text{wfp-on } P A \iff \ (\forall Q x. \ x \in Q \land Q \subseteq A \imp \ (\exists z \in Q. \ \forall y. \ P y z \imp y \notin Q)) \)
using wfp-on-imp-minimal [of P A]
and minimal-imp-inductive-on [of A P]
and inductive-on-imp-wfp-on [of P A]
by blast

Every non-empty well-founded set \( A \) has a minimal element, i.e., an element that is not greater than any other element.
lemma \textit{wfp-on-imp-has-min-elt}:
assumes \textit{wfp-on P A and} \( A \neq \{\} \)
shows \( \exists x \in A. \ \forall y \in A. \ \neg P y x \)
using assms unfolding \textit{wfp-on-iff-minimal} by force

lemma \textit{wfp-on-induct} [\textit{consumes 2, case-names less, induct pred: wfp-on}]:
assumes \textit{wfp-on P A and} \( x \in A \)
and \( \forall y. \ [ y \in A; \ \forall x. \ [ x \in A; \ P x y ] \ \Rightarrow \ Q x ] \ \Rightarrow \ Q y \)
shows \( Q x \)
using assms and inductive-on-induct [\textit{of P A x}]
unfolding \textit{wfp-on-iff-inductive-on} by blast

lemma \textit{wfp-on-UNIV} [simp]:
\textit{wfp-on P UNIV} \iff \textit{wfP P}
unfolding \textit{wfp-on-iff-inductive-on} inductive-on-def wfP-def \textit{wf-def} by force

**12.1 Measures on Sets (Instead of Full Types)**

definition \textit{inv-image-betw} :: \( \lambda \text{betw} \Rightarrow \) \( \Rightarrow \text{set} \Rightarrow \text{set} \Rightarrow \) \( \Rightarrow \) \( \Rightarrow \) \( \Rightarrow \)
\begin{align*}
& \text{inv-image-betw} P f A B = \lambda x y. \ x \in A \wedge y \in A \wedge f x \in B \wedge f y \in B \wedge P (f x) (f y))
\end{align*}

where

definition \textit{measure-on} :: \( \Rightarrow \text{nat} \Rightarrow \text{set} \Rightarrow \text{set} \Rightarrow \text{bool} \)
\begin{align*}
& \text{measure-on} f A = \text{inv-image-betw} (\text{op} <) f A \text{ UNIV}
\end{align*}

lemma \textit{in-inv-image-betw} [simp]:
\textit{inv-image-betw} \( P f A B \) \( x y \) \iff \( x \in A \wedge y \in A \wedge f x \in B \wedge f y \in B \wedge P (f x) (f y)
\) 
by (auto simp: inv-image-betw-def)

lemma \textit{in-measure-on} [simp, code-unfold]:
\textit{measure-on} \( f A x y \) \iff \( x \in A \wedge y \in A \wedge f x < f y \)
by (simp add: measure-on-def)

lemma \textit{wfp-on-inv-image-betw} [simp, intro!]:
assumes \textit{wfp-on P B}
shows \textit{wfp-on (inv-image-betw P f A B) A} (\textit{is wfp-on ?P A})
proof (rule contr)
assume \( \neg \ ?thesis \)
then obtain \( g \) where \( \forall i. \ g i \in A \wedge \ ?P (g (\text{Suc} i)) \) (\( g i \)) by (auto simp: wfp-on-def)
with assms show \textit{False} by (auto simp: wfp-on-def)
qed
lemma wfp-less:
\[
\text{wfp-on } (\text{op } <) (\text{UNIV :: nat set})
\]
using wfp-less by (auto simp: wfP-def)

lemma wfp-on-measure-on [iff]:
\[
\text{wfp-on } (\text{measure-on } f A) A
\]
unfolding measure-on-def by (rule wfp-less [THEN wfp-on-inv-image-betw])

lemma wfp-on-mono:
\[
A \subseteq B \implies (\forall x y. x \in A \implies y \in A \implies P x y \implies Q x y) \implies \text{wfp-on } Q B \implies \text{wfp-on } P A
\]
unfolding wfp-on-def by (metis set-mp)

lemma wfp-on-subset:
\[
A \subseteq B \implies \text{wfp-on } P B \implies \text{wfp-on } P A
\]
using wfp-on-mono by blast

lemma restrict-to-iff [iff]:
\[
\text{restrict-to } P A x y \iff x \in A \land y \in A \land P x y
\]
by (simp add: restrict-to-def)

lemma wfp-on-restrict-to [simp]:
\[
\text{wfp-on } (\text{restrict-to } P A) A = \text{wfp-on } P A
\]
by (auto simp: wfp-on-def)

lemma irreflp-on-strict [simp, intro]:
\[
\text{irreflp-on } (\text{strict } P) A
\]
by (auto simp: irreflp-on-def)

lemma transp-on-map':
assumes transp-on Q B
and g ' A \subseteq B
and h ' A \subseteq B
and \(x. x \in A \implies Q = (h x) (g x)\)
shows transp-on (\(\lambda x y. Q (g x) (h y)\)) A
using assms unfolding transp-on-def
by auto (metis imageI set-mp)

lemma transp-on-map:
assumes transp-on Q B
and h ' A \subseteq B
shows transp-on (\(\lambda x y. Q (h x) (h y)\)) A
using transp-on-map' [of Q B h A h, simplified, OF assms] by blast

lemma irreflp-on-map:
assumes irreflp-on Q B
and h ' A \subseteq B
shows irreflp-on (\(\lambda x y. Q (h x) (h y)\)) A
lemma po-on-map:
  assumes po-on Q B
  and h : A ⊆ B
  shows po-on (λx y. Q (h x) (h y)) A
  using assms and transp-on-map and irreflp-on-map
  unfolding po-on-def by auto

lemma chain-on-transp-on-less:
  assumes chain-on P f A and transp P A and i < j
  shows P (f i) (f j)
  using ⟨i < j⟩
  proof (induct j)
    case 0 then show ?case by simp
    next
    case (Suc j)
    show ?case
    proof (cases i = j)
    case True
    with Suc show ?thesis using assms(1) by simp
    next
    case False
    with Suc have P (f i) (f j) by force
    moreover from assms have P (f j) (f (Suc j)) by auto
    ultimately show ?thesis using assms(1, 2) unfolding transp-on-def by blast
    qed
  qed

lemma wfp-on-imp-irreflp-on:
  assumes wfp-on P A
  shows irreflp-on P A
  proof
    fix x
    assume x ∈ A
    show ¬ P x x
    proof
      let ?f = λ-- x
      assume P x x
      then have ∀ i. P (?f (Suc i)) (?f i) by blast
      with ⟨x ∈ A⟩ have ¬ wfp-on P A by (auto simp: wfp-on-def)
      with assms show False by contradiction
    qed
  qed

inductive accessible-on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ ′a ⇒ bool
  for P and A
  where
accessible-onI [Pure.intro]:
\[ x \in A; \forall y. [y \in A; P y x] \implies \text{accessible-on } P A y \implies \text{accessible-on } P A x \]

\text{lemma accessible-on-imp-mem:}
\text{assumes accessible-on } P A a
\text{shows } a \in A
\text{using assms by (induct) auto}

\text{lemma accessible-on-induct [consumes 1, induct pred: accessible-on]:}
\text{assumes } \star: \text{accessible-on } P A a
\text{and } IH: \forall x. [\text{accessible-on } P A x; \forall y. [y \in A; P y x] \implies Q y] \implies Q x
\text{shows } Q a
\text{by (rule } \star [\text{THEN accessible-on.induct}]\text{) (auto intro: } IH \text{ accessible-onI)}

\text{lemma accessible-on-downward:}
\text{accessible-on } P A b \implies a \in A \implies P a b \implies \text{accessible-on } P A a
\text{by (cases rule: accessible-on.cases) fast}

\text{lemma accessible-on-restrict-to-downwards:}
\text{assumes (restrict-to } P A)++ a b \text{ and accessible-on } P A b
\text{shows accessible-on } P A a
\text{using assms by (induct) (auto dest: accessible-on-imp-mem accessible-on-downward)}

\text{lemma accessible-on-inductive-on:}
\text{assumes } \forall x \in A. \text{accessible-on } P A x
\text{shows inductive-on } P A
\text{proof}
\text{fix } Q x
\text{assume } x \in A
\text{and } \star: \forall y. [y \in A; \forall x. [x \in A; P x y] \implies Q x] \implies Q y
\text{with assms have accessible-on } P A x \text{ by auto}
\text{then show } Q x
\text{proof (induct)}
\text{case (1 z)}
\text{then have } z \in A \text{ by (blast dest: accessible-on-imp-mem)}
\text{show } ?case \text{ by (rule } \star \text{) fact+}
\text{qed}
\text{qed}

\text{lemmas accessible-on-imp-wfp-on = accessible-on-imp-inductive-on [THEN inductive-on-imp-wfp-on]}

\text{lemma wfp-on-tranclp-imp-wfp-on:}
\text{assumes wfp-on } (P^{++}) A
\text{shows wfp-on } P A
\text{by (rule ccontr) (insert assms, auto simp: wfp-on-def)}

\text{lemma inductive-on-imp-accessible-on:}
\text{assumes inductive-on } P A
\text{shows } \forall x \in A. \text{accessible-on } P A x

67
proof
fix x
assume \( x \in A \)
with \( \text{assms} \) show accessible-on \( P \ A \ x \)
  by (induct) (auto intro: accessible-onI)
qed

lemma inductive-on-accessible-on-cone:
inductive-on \( P \ A \leftarrow (\forall x \in A. \text{accessible-on} \ P \ A \ x) \)
using inductive-on-imp-accessible-on
  and accessible-on-imp-inductive-on
by blast

lemmas wfp-on-imp-accessible-on =
wfp-on-imp-inductive-on [THEN inductive-on-imp-accessible-on]

lemma accessible-on-tranclp:
assumes accessible-on \( P \ A \ x \)
shows accessible-on \((\text{restrict-to} \ P \ A)^{++} \) \( A \ x \)
  (is accessible-on \(?P \ A \ x\) )
using assms
proof (induct)
case (1 \( x \))
  then have \( x \in A \) by (blast dest: accessible-on-imp-mem)
then show \(?\)case
proof (rule accessible-onI)
  fix \( y \)
  assume \( y \in A \)
  assume \(?P \ y \ x\) 
  then show accessible-on \(?P \ A \ y\)
proof (cases)
  assume restrict-to \( P \ A \ y \ x \)
  with \( 1 \) and \( y \in A \) show \(?\)thesis by blast
next
fix \( z \)
  assume \(?P \ y \ z\) and restrict-to \( P \ A \ z \ x \)
  with \( 1 \) have accessible-on \(?P \ A \ z\) by (auto simp: restrict-to-def)
  from accessible-on-downward \([OF this \ y \in A \ (?P \ y \ z)]\)
  show \(?\)thesis .
qed
qed

lemma wfp-on-restrict-to-tranclp:
assumes wfp-on \( P \ A \)
shows wfp-on \((\text{restrict-to} \ P \ A)^{++} \) \( A \)
using wfp-on-imp-accessible-on \([OF \ assms]\)
  and accessible-on-tranclp \([of \ P \ A]\)
  and accessible-on-imp-wfp-on \([of \ A \ (\text{restrict-to} \ P \ A)^{++}]\)
by blast

lemma wfp-on-restrict-to-tranclp':
  assumes wfp-on (restrict-to P A)++ A
  shows wfp-on P A
  by (rule ccontr) (insert assms, auto simp: wfp-on-def)

lemma wfp-on-restrict-to-tranclp-wfp-on-conv:
  wfp-on (restrict-to P A)++ A ↔ wfp-on P A
  using wfp-on-restrict-to-tranclp [of P A]
  and wfp-on-restrict-to-tranclp' [of P A]
  by blast

lemma tranclp-idemp [simp]:
  (P++)++ = P++ (is ?l = ?r)
proof (intro ext)
  fix x y
  show ?l x y = ?r x y
  proof
    assume ?l x y then show ?r x y by (induct) auto
  next
    assume ?r x y then show ?l x y by (induct) auto
  qed
  qed

lemma stepfun-imp-tranclp:
  assumes f 0 = x and f (Suc n) = z
  and ∀ i ≤ n. P (f i) (f (Suc i))
  shows P+++ x z
  using assms
  by (induct n arbitrary: x z)
    (auto intro: tranclp.trancl-into-trancl)

lemma tranclp-imp-stepfun:
  assumes P+++ x z
  shows ∃ f n. f 0 = x ∧ f (Suc n) = z ∧ (∀ i ≤ n. P (f i) (f (Suc i)))
  (is ∃ f n. ?P x z f n)
  using assms
proof (induct rule: tranclp-induct)
  case (base y)
  let ?f = (λ-. y)(0 := x)
  have ?f 0 = x and ?f (Suc 0) = y by auto
  moreover have ∀ i ≤ 0. P (?f i) (?f (Suc i))
    using base by auto
  ultimately show ?case by blast
next
  case (step y z)
  then obtain f n where IH: ?P x y f n by blast
then have \( \forall i \leq n. P(f(i)(f(Suc\ i))) \)
and \([\text{simp}]: f\ 0 = x\ f\ (Suc\ n) = y\)
by auto

let \(?n = Suc\ n\)
let \(?f = f(Suc\ ?n) = z\)
have \(if\ 0 = x\ and\ if\ (Suc\ ?n) = z\ by\ auto\)
moreover have \(\forall i \leq ?n. P(if\ i)(if\ (Suc\ i))\)
using \(<P\ y z>\) and *
by auto
ultimately show \(?case\ by\ blast\)
qed

**lemma** tranclp-stepfun-conv:
\(P^{+} x y \iff (\exists f\ n. f\ 0 = x \land f(Suc\ n) = y \land \forall i \leq n. P(f i)(f(Suc\ i)))\)
using tranclp-imp-stepfun and stepfun-imp-tranclp by metis

**12.2 Facts About Predecessor Sets**

**lemma** qo-on-predecessor-subset-conv:
assumes qo-on P A and B \(\subseteq\ A\) and \(C \subseteq A\)
shows \(\{x \in A. \exists y \in B. P\ x y\} \subseteq \{x \in A. \exists y \in C. P\ x y\}\)
using assms
by (auto simp: subset-eq qo-on-def reflp-on-def, unfold transp-on-def)

**lemma** qo-on-predecessor-subset-conv':
\([qo-on P A; x \in A; y \in A] \Rightarrow \{z \in A. P z x\} \subseteq \{z \in A. P z y\}\)
using qo-on-predecessor-subset-conv \{of P A \{x\} \{y\}\}
by simp

**lemma** po-on-predecessors-eq-conv:
assumes po-on P A and \(x \in A\) and \(y \in A\)
shows \(\{z \in A. P^{=} z x\} = \{z \in A. P^{=} z y\}\)
using assms(2\−)
and reflp-on-reflclp [of P A]
and po-on-imp-antisym-on [OF \{po-on P A\] unfolding antisym-on-def reflp-on-def
by blast

**lemma** restrict-to-rtranclp:
assumes transp-on P A
and \(x \in A\) and \(y \in A\)
shows \((restrict-to P A)^{\ast\ast} x y \iff P^{=} x y\)

proof --
{ assume (restrict-to P A)^{\ast\ast} x y
then have \(P^{=} x y\) using assms
by (induct) (auto, unfold transp-on-def, blast) }
with assms show \(\text{thesis}\ by\ auto\)
qed

**lemma** reflp-on-restrict-to-rtranclp:
assumes reflp-on $P$ and tranp-on $P$
and $x \in A$ and $y \in A$
shows $(\text{restrict-to}\ P\ A)^*\ x\ y \iff P\ x\ y$
unfolding restrict-to-tranclp [OF assms(2-)]
unfolding reflp-on-reflclp-simp [OF assms(1-3)] ..

end

13 Constructing Minimal Bad Sequences

theory Minimal-Bad-Sequences
imports Restricted-Predicates
begin

The set of all infinite sequences over elements from $A$.
definition $\text{SEQ}\ A = \{f::\text{nat} \Rightarrow 'a. \forall i. f\ i \in A\}$

lemma $\text{SEQ-iff}$ [iff]:
$f \in \text{SEQ}\ A \iff (\forall i. f\ i \in A)$
by (auto simp: $\text{SEQ-def}$)

An infinite sequence is good whenever there are indices $i < j$ such that $P\ (f\ i)\ (f\ j)$.
definition good :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow (\text{nat} \Rightarrow 'a \Rightarrow \text{bool})$ where
good $P\ f \iff (\exists i\ j. i < j \land P\ (f\ i)\ (f\ j))$

A sequence that is not good is called bad.
abbreviation bad $P\ f \equiv \neg\ \text{good}\ P\ f$

lemma goodI:
$[i < j; P\ (f\ i)\ (f\ j)] \Longrightarrow \text{good}\ P\ f$
by (auto simp: $\text{good-def}$)

lemma goodE [elim]:
good $P\ f \Longrightarrow ((\forall i\ j. i < j \land P\ (f\ i)\ (f\ j)) \Longrightarrow Q) \Longrightarrow Q$
by (auto simp: $\text{good-def}$)

lemma badE [elim]:
bad $P\ f \Longrightarrow (\neg\ P\ (f\ i)\ (f\ j)) \Longrightarrow Q \Longrightarrow Q$
by (auto simp: $\text{good-def}$)

A locale capturing the construction of minimal bad sequences over values from $A$. Where minimality is to be understood w.r.t. size of an element.
locale mbs =
fixes $A::('a::\text{size})\ \text{set}$
begin

Since the size is a well-founded measure, whenever some element satisfies a property $P$, then there is a size-minimal such element.
lemma minimal:
  assumes \( x \in A \) and \( P x \)
  shows \( \exists y \in A. \) \( \text{size } y \leq \text{size } x \wedge (\forall z \in A. \) \( \text{size } z < \text{size } y \rightarrow \neg P z) \)
using assms
proof (induction \( x \) taking: size rule: measure-induct)
case \( 1 x \)
  then show \(?case \) by (cases \( \forall y \in A. \) \( \text{size } y < \text{size } x \rightarrow \neg P y \))
next
case False
  then obtain \( y \) where \( y \in A \) and \( \text{size } y < \text{size } x \) and \( P y \) by blast
  with \( 1.\text{IH} \) show \(?thesis \) by (fastforce elim!: order-trans)
qed

lemma less-not-eq [simp]:
  \( x \in A \Rightarrow \text{size } x < \text{size } y \Rightarrow x = y \Rightarrow False \)
by simp

The set of all bad sequences over \( A \).
definition BAD \( P \) = \( \{ f \in \text{SEQ } A. \) bad \( P f \} \)

lemma BAD-iff [iff]:
  \( f \in \text{BAD } P \iff (\forall i. f i \in A) \wedge \text{bad } P f \)
by (auto simp: BAD-def)

A partial order on infinite bad sequences.
definition geseq :: \((\text{nat } \Rightarrow 'a) \times (\text{nat } \Rightarrow 'a)\) set
where
geseq =
  \{ (f, g). f \in \text{SEQ } A \land g \in \text{SEQ } A \land (f = g \lor (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j))) \}

The strict part of the above order.
definition gseq :: \((\text{nat } \Rightarrow 'a) \times (\text{nat } \Rightarrow 'a)\) set where
gseq = \{ (f, g). f \in \text{SEQ } A \land g \in \text{SEQ } A \land (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j))) \}

lemma gseq-iff:
  \( (f, g) \in \text{gseq} \iff \)
f \( \in \text{SEQ } A \land g \in \text{SEQ } A \land (f = g \lor (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j)) \))
by (auto simp: geseq-def)

lemma gseq-iff:
  \( (f, g) \in \text{gseq} \iff (f \in \text{SEQ } A \land g \in \text{SEQ } A \land (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j)))) \)
by (auto simp: gseq-def)

lemma gseqE:
assumes \((f, g) \in gseq\)
and \(\forall i. f i \in A; \forall i. g i \in A; f = g\) \(\Rightarrow Q\)
and \(\exists i. \forall i. f i \in A; \forall i. g i \in A; \text{size} (g i) < \text{size} (f i); \forall j < i. f j = g j\)
\(\Rightarrow Q\)
shows \(Q\)
using \(assms\) by (auto simp: gseq-iff)

lemma gseqE:
assumes \((f, g) \in gseq\)
and \(\forall i. f i \in A; \forall i. g i \in A; \text{size} (g i) < \text{size} (f i); \forall j < i. f j = g j\)
\(\Rightarrow Q\)
shows \(Q\)
using \(assms\) by (auto simp: gseq-iff)

The \(i\)-th ”column” of a set \(B\) of infinite sequences.

definition \(ith\ B i = \{ f i | f. f \in B \}\)

lemma \(ithI\) [intro]:
\(f \in B \Rightarrow f i = x \Rightarrow x \in ith B i\)
by (auto simp: ith-def)

lemma \(ithE\) [elim]:
\([x \in ith B i; \bigwedge f. [f \in B; f i = x] \Rightarrow Q]\ \Rightarrow Q\)
by (auto simp: ith-def)

lemma \(ith\-conv\):
\(x \in ith B i \leftrightarrow (\exists f \in B. x = f i)\)
by auto

context
fixes \(B :: \text{'a set}\)
assumes \(\text{subset-A}: B \subseteq A\) and \(\text{ne}: B \neq \{\}\)
begin

A minimal element (w.r.t. \(\text{size}\)) from a set.

definition \(\text{min-elt} = (SOME x. x \in B \land (\forall y \in A. \text{size} y < \text{size} x \rightarrow y \notin B))\)

lemma \(\text{min-elt-ex}\):
\(\exists x. x \in B \land (\forall y \in A. \text{size} y < \text{size} x \rightarrow y \notin B)\)
using \(\text{subset-A}\) and \(\text{ne}\) using \(\text{minimal}\) [of \(\lambda x. x \in B\)] by auto

lemma \(\text{min-elt-mem}\):
\(\text{min-elt} \in B\)
using \(\text{someI-ex}\) [OF \(\text{min-elt-ex}\)] by (auto simp: \(\text{min-elt-def}\))

lemma \(\text{min-elt-minimal}\):
assumes $y \in A$ and $\text{size } y < \text{size } \text{min-elt}$
shows $y \notin B$
using $\text{someI-ex [OF min-elt-ex]}$ and $\text{assms by (auto simp: min-elt-def)}$
end
end

The restriction of a set $B$ of sequences to sequences that are equal to a given sequence $f$ up to position $i$.

**definition** $\text{eq-upto} :: (\text{nat } \Rightarrow 'a) \Rightarrow (\text{nat } \Rightarrow 'a) \Rightarrow \text{nat } \Rightarrow (\text{nat } \Rightarrow 'a) \Rightarrow \text{set}$
where
\[
\text{eq-upto } B \ f \ i = \{ g \in B. \forall j < i. \ f j = g j \}
\]

**lemma** $\text{eq-uptoI} \ [\text{intro}]$:
\[
[\forall j. j < i \Rightarrow f j = g j] \Rightarrow g \in \text{eq-upto } B \ f \ i
\]
by (auto simp: $\text{eq-upto-def}$)

**lemma** $\text{eq-uptoE} \ [\text{elim}]$:
\[
[\forall j. j < i \Rightarrow f j = g j] \Rightarrow Q \]\[
\Rightarrow Q
\]
by (auto simp: $\text{eq-upto-def}$)

**lemma** $\text{eq-upto-Suc}$:
\[
[\forall j. j < i \Rightarrow f j = g j] \Rightarrow g \in \text{eq-upto } B \ f \ (\text{Suc } i)
\]
by (auto simp: $\text{eq-upto-def}$ less-Suc-eq)

**lemma** $\text{eq-upto-0} \ [\text{simp}]$:
\[
\text{eq-upto } B \ f \ 0 = B
\]
by (auto simp: $\text{eq-upto-def}$)

**lemma** $\text{eq-upto-cong} \ [\text{fundef-cong}]$:
assumes $\forall j. j < i \Rightarrow f j = g j$ and $B = C$
shows $\text{eq-upto } B \ f \ i = \text{eq-upto } C \ g \ i$
using $\text{assms by (auto simp: eq-upto-def)}$

context $\text{mbs}$
begin

A lower bound to all sequences in a set of sequences $B$.

**fun** $\text{lb} :: \text{nat } \Rightarrow 'a$ where
\[
\text{lb} \colon \text{lb } i = \text{min-elt } (\text{ith } (\text{eq-upto } (BAD \ P) \ \text{lb } i) \ i)
\]
declare $\text{lb.simps [simp del]}$

**lemma** $\text{eq-upto-BAD-mem}$:
assumes $f \in \text{eq-upto} (\text{BAD P})$ $g i$
shows $f j \in A$
using assms by (auto)

Assume that there is some infinite bad sequence $h$.

context
fixes $h :: \text{nat} \Rightarrow 'a$
assumes BAD-ex: $h \in \text{BAD P}$
begin

When there is a bad sequence, then filtering $\text{BAD P}$ w.r.t. positions in $lb$ never yields an empty set of sequences.

lemma eq-upto-BAD-non-empty:

\[ \text{eq-upto} (\text{BAD P}) \; \text{lb} \; i \neq \{\} \]

proof (induct $i$)

case 0
show \(?case\) using BAD-ex by auto

next
let \(?A\) = $\lambda i \cdot \text{ith} (\text{eq-upto} (\text{BAD P}) \; \text{lb} \; i)$

case (Suc $i$)
then have \(?A \; i \neq \{\}\) by auto
moreover have \(\text{eq-upto} (\text{BAD P}) \; \text{lb} \; 0 \subseteq \text{eq-upto} (\text{BAD P}) \; \text{lb} \; \theta\) by auto
ultimately have \(?A \; i \subseteq A\) and \(?A \; i \neq \{\}\) by (auto simp: \text{ith-def})

from min-elt-mem [OF this, \text{folded} \text{lb}] obtain \(f\)
where \(f \in \text{eq-upto} (\text{BAD P}) \; \text{lb} \; (\text{Suc} \; i)\) by (auto dest: eq-upto-Suc)
then show \(?case\) by blast
qed

lemma non-empty-ith:

shows \(\text{ith} (\text{eq-upto} (\text{BAD P}) \; \text{lb} \; i) \subseteq A\)
and \(\text{ith} (\text{eq-upto} (\text{BAD P}) \; \text{lb} \; i) \; i \neq \{\}\)

using eq-upto-BAD-non-empty [of $i$] by auto

lemmas
\(\text{lb-minimal} = \text{min-elt-minimal} [\text{OF non-empty-ith, \text{folded} \text{lb}] \text{ and}\)
\(\text{lb-mem} = \text{min-elt-mem} [\text{OF non-empty-ith, \text{folded} \text{lb}]\}

\(\text{lb}\) is a infinite bad sequence.

lemma lb-BAD:
\(\text{lb} \in \text{BAD P}\)

proof –

have \(\ast\): \(\forall j . \; \text{lb} \; j \in \text{ith} (\text{eq-upto} (\text{BAD P}) \; \text{lb} \; j) \; j\) by (rule lb-mem)
then have \(\forall i . \; \text{lb} \; i \in A\) by (auto simp: \text{ith-conv}) (metis eq-upto-BAD-mem)
moreover
\{ assume \(\text{good P \; lb}\)
then obtain \(i \; j\) where \(i \lt j\) and \(P \; (\text{lb} \; i) \; (\text{lb} \; j)\) by (auto simp: \text{good-def})
from \(\ast\) have \(\text{lb} \; j \in \text{ith} (\text{eq-upto} (\text{BAD P}) \; \text{lb} \; j) \; j\) by (auto)
then obtain \(g\) where \(g \in \text{eq-upto} (\text{BAD P}) \; \text{lb} \; j\) and \(g \; j = \text{lb} \; j\) by force
then have \(\forall k \leq j . \; g \; k = \text{lb} \; k\) by (auto simp: \text{order-le-less})
with \( i < j \) and \( P(\langle lb\ i \rangle \ (lb\ j)) \) have \( P\ (g\ i)\ (g\ j) \) by auto
with \( i < j \) have good \( P\ g \) by (auto simp: good-def)
with \( g \in \text{eq-upto}\ (BAD\ P)\ lb\ j \) have False by auto
ultimately show \( \text{thesis} \) by blast
qed

There is no infinite bad sequence that is strictly smaller than \( lb \).

lemma lb-lower-bound:
\(~\forall\ g.\ (lb,\ g) \in gseq \rightarrow g \notin BAD\ P~\)
proof (intro allI impI)
fix \( g \)
assume \( (lb,\ g) \in gseq \)
then obtain \( i \) where \( g\ i \in A\ \text{and size}\ (g\ i) < \text{size}\ (lb\ i) \)
and \( \forall j < i.\ lb\ j = g\ j \) by (auto simp: gseq-iff)
moreover with \( lb\)-minimal
have \( g\ i \notin \text{i-th}\ (eq-upto\ (BAD\ P)\ lb\ i)\ i \) by auto
ultimately show \( g \notin BAD\ P \) by blast
qed

If there is at least one bad sequence, then there is also a minimal one.

lemma lower-bound-ex:
\(~\exists f \in BAD\ P.\ \forall g.\ (f,\ g) \in gseq \rightarrow g \notin BAD\ P~\)
using lb-BAD and lb-lower-bound by blast

lemma gseq-conv:
\( (f,\ g) \in gseq \leftrightarrow f \neq g \land (f,\ g) \in gseq \)
by (auto simp: gseq-def geseq-def dest: less-not-eq)

There is a minimal bad sequence.

lemma mbs:
\(~\exists f \in BAD\ P.\ \forall g.\ (f,\ g) \in gseq \rightarrow \text{good}\ P\ g~\)
using lower-bound-ex by (auto simp: gseq-conv geseq-iff)

end

end
end

end

14 Almost-Full Relations

theory Almost-Full-Relations
imports
  ~~/src/HOL/Library/Sublist
  ~~/src/HOL/Library/Ramsey
  ../Regular-Sets/Regexp-Method
  ../Abstract-Rewriting/Seq
14.1 Basic Definitions and Facts

definition almost-full-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool where
  almost-full-on P A ⟷ (∀ f ∈ SEQ A. good P f)

lemma almost-full-on-UNIV:
  almost-full-on (λ - -. True) UNIV
  by (auto simp: almost-full-on-def good-def)

lemma (in mbs) mbs':
  assumes ¬ almost-full-on P A
  shows ∃ m ∈ BAD P. ∀ g. (m, g) ∈ gseq ⟷ good P g
  using assms and mbs
  unfolding almost-full-on-def by blast

lemma almost-full-onD:
  fixes f :: nat ⇒ 'a and A :: 'a set
  assumes almost-full-on P A and ∀ i. f i ∈ A
  obtains i j where i < j and P (f i) (f j)
  using assms unfolding almost-full-on-def by blast

lemma almost-full-onI [Pure.intro]:
  (∀ f. ∀ i. f i ∈ A =⇒ good P f) =⇒ almost-full-on P A
  unfolding almost-full-on-def by blast

lemma almost-full-on-imp-reflp-on:
  assumes almost-full-on P A
  shows reflp-on P A
  using assms by (auto simp: almost-full-on-def reflp-on-def)

lemma almost-full-on-subset:
  A ⊆ B =⇒ almost-full-on P B =⇒ almost-full-on P A
  by (auto simp: almost-full-on-def)

lemma almost-full-on-mono:
  assumes A ⊆ B and ∀ x y. Q x y =⇒ P x y
  and almost-full-on Q B
  shows almost-full-on P A
  using assms by (metis almost-full-on-def almost-full-on-subset good-def)

  Every sequence over elements of an almost-full set has a homogeneous subsequence.

lemma almost-full-on-imp-homogeneous-subseq:
  assumes almost-full-on P A
  and ∀ i::nat. f i ∈ A
shows $\exists \varphi :: \text{nat} \Rightarrow \text{nat}. \forall i j. i < j \rightarrow \varphi i < \varphi j \land P (f \varphi i) (f \varphi j)$

proof –

- def $X \equiv \{i, j\mid i < j \land P (f i) (f j)\}$
- def $Y \equiv - X$
- def $h \equiv \lambda Z. \text{if } Z \in X \text{ then } 0 \text{ else } \text{Suc } 0$

have [iff]: $\forall x y. h \{x, y\} = 0 \leftrightarrow \{x, y\} \in X$ by (auto simp: h-def)

have [iff]: $\forall x y. h \{x, y\} = \text{Suc } 0 \leftrightarrow \{x, y\} \in Y$ by (auto simp: h-def Y-def)

have $\forall x \in \text{UNIV}. \forall y \in \text{UNIV}. x \neq y \rightarrow h \{x, y\} < 2$ by (simp add: h-def)

from Ramsey2 [OF infinite-UNIV-nat this] obtain $I \ c$

where infinite $I$ and $c < 2$

and $*: \forall x \in I. \forall y \in I. x \neq y \rightarrow h \{x, y\} = c$ by blast

then interpret infinitely-many1 $\lambda i. i \in I$

by (unfold-locales) (simp add: infinite-nat-iff-unbounded)

have $c = 0 \lor c = 1$ using $\langle c < 2 \rangle$ by arith

then show ?thesis

proof

assume [simp]: $c = 0$

have $\forall i j. i < j \rightarrow P (f \text{enum } i) (f \text{enum } j)$

proof (intro allI impI)

fix $i j :: \text{nat}$

assume $i < j$

from $*$ and enum-P and enum-less [OF $\langle i < j \rangle$] have $\{\text{enum } i, \text{enum } j\} \in X$ by auto

with enum-less [OF $\langle i < j \rangle$]

show $P (f \text{enum } i) (f \text{enum } j)$ by (auto simp: X-def doubleton-eq-iff)

qed

then show ?thesis using enum-less by blast

next

assume [simp]: $c = 1$

have $\forall i j. i < j \rightarrow \neg P (f \text{enum } i) (f \text{enum } j)$

proof (intro allI impI)

fix $i j :: \text{nat}$

assume $i < j$

from $*$ and enum-P and enum-less [OF $\langle i < j \rangle$] have $\{\text{enum } i, \text{enum } j\} \in Y$ by auto

with enum-less [OF $\langle i < j \rangle$]

show $\neg P (f \text{enum } i) (f \text{enum } j)$ by (auto simp: Y-def X-def doubleton-eq-iff)

qed

then have $\neg \text{good } P (f \circ \text{enum})$ by auto

moreover have $\forall i. f (\text{enum } i) \in A$ using assms by auto

ultimately show ?thesis using almost-full-on $P A$ by (simp add: almost-full-on-def)

qed

Almost full relations do not admit infinite antichains.

lemma almost-full-on-imp-no-antichain-on:
assumes almost-full-on $P A$
shows $\neg$ antichain-on $P f A$

proof
assume $*$: antichain-on $P f A$
then have $\forall i. f i \in A$ by simp
with assms have good $P f$ by (auto simp: almost-full-on-def)
then obtain $i j$ where $i < j$ and $P (f i) (f j)$
  unfolding good-def by auto
moreover with $*$ have incomparable $P (f i) (f j)$ by auto
ultimately show False by blast
qed

If the image of a function is almost-full then also its preimage is almost-full.

lemma almost-full-on-map:
assumes almost-full-on $Q B$
and $h ' A \subseteq B$
shows almost-full-on $(\lambda x y. Q (h x) (h y)) A$ (is almost-full-on $?P A$)

proof
fix $f$
assume $\forall i::\text{nat}. f i \in A$
then have $\forall i. \exists x. x \in A \land f i = h (g i)$ by (auto simp: image-def)
from choice [OF this] obtain $g$
  where $*: \forall i. g i \in A \land f i = h (g i)$ by blast
show good $?P f$
proof (rule ccontr)
assume bad: bad $Q f$
{ fix $i j :: \text{nat}$
  assume $i < j$
  from bad have $\neg Q (f i) (f j)$ using $i < j$ by (auto simp: good-def)
  with hom have $\neg P (g i) (g j)$ using $*$ by auto }
then have bad $P g$ by (auto simp: good-def)
with $af$ and $*$ show False by (auto simp: good-def almost-full-on-def)
qed

The homomorphic image of an almost-full set is almost-full.

lemma almost-full-on-hom:
fixes $h :: 'a \Rightarrow 'b$
assumes hom: $\forall x y. [x \in A; y \in A; P x y] \Longrightarrow Q (h x) (h y)$
and af: almost-full-on $P A$
shows almost-full-on $Q (h ' A)$

proof
fix $f :: \text{nat} \Rightarrow 'b$
assume $\forall i. f i \in h ' A$
then have $\forall i. \exists x. x \in A \land f i = h x$ by (auto simp: image-def)
from choice [OF this] obtain $g$
  where $*: \forall i. g i \in A \land f i = h (g i)$ by blast
show good $Q f$
proof (rule ccontr)
assume bad: bad $Q f$
{ fix $i j :: \text{nat}$
  assume $i < j$
  from bad have $\neg Q (f i) (f j)$ using $i < j$ by (auto simp: good-def)
  with hom have $\neg P (g i) (g j)$ using $*$ by auto }
then have bad $P g$ by (auto simp: good-def)
with $af$ and $*$ show False by (auto simp: good-def almost-full-on-def)
qed
The monomorphic preimage of an almost-full set is almost-full.

**Lemma:** *almost-full-on-mon*:

**Assumes:**
- \(\forall x y. \{ x \in A ; y \in A \} \Rightarrow P x y = Q (h x) (h y)\) bij_betw \(h\) \(A\) \(B\)
- \(\text{af}: \text{almost-full-on} Q B\)

**Shows:** almost-full-on \(P A\)

**Proof**

\[
\text{fix } f :: \text{nat} \Rightarrow \text{'}a
\]
\[
\text{assume } \ast; \forall i, f i \in A
\]
\[
\text{then have } \ast\ast; \forall i, (h \circ f) i \in B \text{ using } \text{mon by (auto simp: bij_betw-def)}
\]
\[
\text{show } \text{good } P f
\]

**Proof** (rule ccontr)

\[
\text{assume } \neg \text{almost-full-on } P == A
\]
\[
\text{then obtain } f :: \text{nat} \Rightarrow \text{'}a \text{ where } \ast; \forall i, f i \in A
\]
\[
\text{and } \forall i j, i < j \quad \neg P (f i) (f j) \text{ using } \ast\ast\ast; \forall i, (h \circ f) i \in B \text{ by (auto simp: good-def)}
\]
\[
\text{with } \text{mon have } \neg Q (h (f i)) (h (f j)) \text{ using } \ast\ast\ast\ast; \forall i, (h \circ f) i \in B \text{ by (auto simp: inj-on-def)}
\]
\[
\text{then have } \neg Q (h \circ f) (h \circ f) \text{ by (auto simp: good-def)}
\]
\[
\text{with } \text{af and } \ast\ast\ast\ast\ast\ast\ast; \forall i, (h \circ f) i \in B \text{ by (auto simp: good-def almost-full-on-def)}
\]

\[
\text{qed}
\]

\[
\text{qed}
\]

Every total and well-founded relation is almost-full.

**Lemma:** *total-on-and-wfp-on-imp-almost-full-on*:

**Assumes:**
- \(\text{total-on } P A\)
- \(\text{wfp-on } P A\)

**Shows:** almost-full-on \(P == A\)

**Proof** (rule ccontr)

\[
\text{assume } \neg \text{almost-full-on } P == A
\]
\[
\text{then obtain } f :: \text{nat} \Rightarrow \text{'}a \text{ where } \ast; \forall i, f i \in A
\]
\[
\text{and } \forall i j, i < j \quad \neg P == (f i) (f j) \quad \text{by (auto dest: badE)}
\]
\[
\text{unfolding } \text{almost-full-on-def by (auto dest: badE)}
\]
\[
\text{with } \text{total-on } P A \text{ have } \forall i j, i < j \quad \neg P (f j) (f i)
\]
\[
\text{unfolding } \text{total-on-def by blast}
\]
\[
\text{then have } \forall i, (f (\text{Suc } i)) (f i) \text{ by auto}
\]
\[
\text{with } \text{wfp-on } P A \text{ and } \ast \quad \text{show } \text{False}
\]
\[
\text{unfolding } \text{wfp-on-def by blast}
\]

\[
\text{qed}
\]

**14.2 Adding a Bottom Element to a Set**

**Definition:** *with-bot :: 'a set ⇒ 'a option set (-⊥ \([1000]\) 1000)*

**Where**
- \(A_\bot = \{\text{None}\} \cup \text{Some } ' A\)

**Lemma:** *with-bot-iff [iff]*:

\[
\text{Some } x \in A_\bot \iff x \in A
\]
\[
\text{by (auto simp: with-bot-def)}
\]

**Lemma:** *NoneI [simp, intro]:*

80
None ∈ A⊥
by (simp add: with-bot-def)

lemma not-None-the-mem [simp]:
x ≠ None ⇒ the x ∈ A ←→ x ∈ A⊥
by auto

lemma with-bot-cases:
u ∈ A⊥ ⇒ (∀x. x ∈ A ⇒ u = Some x ⇒ P) ⇒ (u = None ⇒ P) ⇒ P
by auto

lemma with-bot-empty-conv [iff]:
A⊥ = {None} ←→ A = {}
by (auto elim: with-bot-cases)

lemma with-bot-UNIV [simp]:
UNIV⊥ = UNIV
proof (rule set-eqI)
fix x :: 'a option
show x ∈ UNIV⊥ ←→ x ∈ UNIV by (cases x) auto
qed

14.3 Adding a Bottom Element to an Almost-Full Set
fun
  option-le :: ('a ⇒ 'a ⇒ bool) ⇒ 'a option ⇒ 'a option ⇒ bool
where
  option-le P None y = True |
  option-le P (Some x) None = False |
  option-le P (Some x) (Some y) = P x y

lemma None-imp-good-option-le [simp]:
assumes f i = None
shows good (option-le P) f
by (rule goodI [of i Suc i]) (auto simp: assms)

lemma almost-full-on-with-bot:
assumes almost-full-on P A
shows almost-full-on (option-le P) A⊥ (is almost-full-on ?P ?A)
proof
fix f :: nat ⇒ 'a option
assume *: ∀i. f i ∈ ?A
show good ?P f
proof (cases ∀i. f i ≠ None)
  case True
  then have **: ∀i. Some (the (f i)) = f i
  and ∃i. the (f i) ∈ A using * by auto
  with almost-full-onD [OF assms, of the ∘ f] obtain i j where i < j
  and P (the (f i)) (the (f j)) by auto
then have \( \?P \ (\text{Some} \ (\text{the} \ (f \ i))) \ (\text{Some} \ (\text{the} \ (f \ j))) \) by simp
then have \( \?P \ (f \ i) \ (f \ j) \) unfolding \(*\).
with \( i < j \) show good \( \?P \ f \) by (auto simp: good-def)
qed auto

disjoint sum of sets.

\section{Disjoint Union of Almost-Full Sets}

fun
\[
\text{sum-le} :: \ ('a \Rightarrow \ 'b) \Rightarrow \ ('b \Rightarrow \ 'c) \Rightarrow \ 'a + \ 'b \Rightarrow \ 'b + \ 'c
\]
where
\[
\text{sum-le} \ P \ Q \ (\text{Inl} \ x) \ (\text{Inl} \ y) = P x y
\]
\[
\text{sum-le} \ P \ Q \ (\text{Inr} \ x) \ (\text{Inr} \ y) = Q x y
\]
\[
\text{sum-le} \ P \ Q x y = False
\]

lemma not-sum-le-cases:
assumes \( \neg \text{sum-le} \ P \ Q a b \) and \( \forall x y. \ [a = \text{Inl} \ x; \ b = \text{Inl} \ y; \ \neg \ P x y] \Rightarrow \text{thesis} \)
and \( \forall x y. \ [a = \text{Inr} \ x; \ b = \text{Inr} \ y; \ \neg \ Q x y] \Rightarrow \text{thesis} \)
and \( \forall x y. \ [a = \text{Inl} \ x; \ b = \text{Inr} \ y] \Rightarrow \text{thesis} \)
and \( \forall x y. \ [a = \text{Inr} \ x; \ b = \text{Inl} \ y] \Rightarrow \text{thesis} \)
shows thesis
using assms by (cases a b rule: sum.exhaust [case-product sum.exhaust]) auto

When two sets are almost-full, then their disjoint sum is almost-full.

lemma almost-full-on-Plus:
assumes almost-full-on \( P \ A \) and almost-full-on \( Q \ B \)
shows almost-full-on \( \text{sum-le} \ P \ Q \) \((A \leftrightarrow B)\) (is almost-full-on \( \?P \ ?A \))
proof
fix \( f :: \nat \Rightarrow \ ('a + \ 'b) \)
let \( ?I = f - \text{Inl} \ A \)
let \( ?J = f - \text{Inr} \ B \)
assume \( \forall i. \ f i \in \ ?A \)
then have \( * \): \( ?J = (\text{UNIV}::\nat \ set) - \ ?I \) by (fastforce)
show good \( \?P \ f \)
proof (rule ccontr)
assume bad: bad \( \?P \ f \)
show False
proof (cases finite \( \?I \))
assume finite \( \?I \)
then have infinite \( \?J \) by (auto simp: *)
then interpret infinitely-many \1 \ \( \lambda i. \ f i \in \text{Inr} \ B \)
by (unfold-locales) (simp add: infinite-nat_iff_unbounded)
have \( [\text{dest}]: \forall i. \ f (\text{enum} \ i) = \text{Inl} \ x \Rightarrow \text{False} \)
using enum-P by (auto simp: image_iff) (metis Inr-InlFalse)
let \( \forall f = \lambda i. \ \text{projr} \ (f (\text{enum} \ i)) \)
have \( B: \forall i. \ \forall i \in B \) using enum-P by (auto simp: image_iff) (metis enum.sel(2))
{ fix \( i \ j :: \nat \)
assume \( i < j \)
}
then have \( \text{enum } i < \text{enum } j \) using \text{enum-less} by auto

with \( \text{bad} \) have \( \neg \ ?P \ (f \ (\text{enum } i)) (f \ (\text{enum } j)) \) by (auto simp: good-def)

then have \( \neg \ Q \ (\ ?f \ i \ ) (\ ?f \ j \ ) \) by (auto elim: not-sam-le-cases)

then have \( \text{bad } Q \ ?f \) by (auto simp: good-def)

moreover from \( \text{almost-full-on } Q \ B \ ) \ and \ B

have \( \text{good } Q \ ?f \) by (auto simp: good-def almost-full-on-def)

ultimately show \( \text{False} \) by blast

next

assume infinite \( ?I \)

then interpret \( \text{infinitely-many1 } \lambda i. \ f \ i \in \text{Inl } A \)

by (unfold-locales) (simp add: infinite-nat-iff-unbounded)

have [dest]: \( \forall i. \ f \ (\text{enum } i) = \text{Inr} \ x \Longrightarrow \text{False} \)

using \text{enum-P} by (auto simp: image-iff) (metis Inr-Inl-False)

let \( ?f = \lambda i. \ \text{projl} (f \ (\text{enum } i)) \)

have \( A: \forall i. \ ?f \ i \in A \) using \text{enum-P} by (auto simp: image-iff) (metis sum.sel(1))

{ fix \( i \ j :: \text{nat} \)

assume \( i < j \)

then have \( \text{enum } i < \text{enum } j \) using \text{enum-less} by auto

with \( \text{bad} \) have \( \neg \ ?P \ (f \ (\text{enum } i)) (f \ (\text{enum } j)) \) by (auto simp: good-def)

then have \( \neg \ P \ (\ ?f \ i \ ) (\ ?f \ j \ ) \) by (auto elim: not-sam-le-cases)

then have \( \text{bad } P \ ?f \) by (auto simp: good-def)

moreover from \( \text{almost-full-on } P \ A \ ) \ and \ A

have \( \text{good } P \ ?f \) by (auto simp: good-def almost-full-on-def)

ultimately show \( \text{False} \) by blast

qed

qed

14.5 Dickson’s Lemma for Almost-Full Relations

When two sets are almost-full, then their Cartesian product is almost-full.

definition prod-le :: \( 'a \Rightarrow 'a \Rightarrow \text{bool} \Rightarrow 'b \Rightarrow 'b \Rightarrow \text{bool} \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow \text{bool} \)

where

\[ \text{prod-le } P1 \ P2 = (\lambda(p1, p2) \ q1, q2). \ P1 \ p1 \ q1 \land \ P2 \ p2 \ q2) \]

lemma prod-le-True [simp]:

\( \text{prod-le } P \ (\lambda- \ . \ \text{True}) \ a \ b = P \ (\text{fst } a) \ (\text{fst } b) \)

by (auto simp: prod-le-def)

lemma almost-full-on-Sigma:

assumes \( \text{almost-full-on } P1 \ A1 \ \text{and } \text{almost-full-on } P2 \ A2 \)

shows \( \text{almost-full-on } (\text{prod-le } P1 \ P2) \ (A1 \times A2) \) (is \( \text{almost-full-on ?P ?A} \))

proof (rule ccontr)

assume \( \neg \ \text{almost-full-on ?P ?A} \)

then obtain \( f \) where \( f: \forall i. \ f \ i \in ?A \)

and \( \text{bad} \) : \( \text{bad } ?P \ f \) by (auto simp: almost-full-on-def)

let \( ?W = \lambda x. \ y. \ P1 \ (\text{fst } x) \ (\text{fst } y) \)

83
let \( ?B = \lambda x. y. P^2 \ (\text{snd} \ x) \ (\text{snd} \ y) \)

from \( f \) have \( \text{fst} \ (\forall i. \ (f \ i) \in A) \) and \( \text{snd} \ (\forall i. \ (f \ i) \in A) \)
  by (metis SigmaE \text{fst-conv}, \text{metis SigmaE \text{snd-conv}})

from \text{almost-full-on-imp-homogeneous-subseq} \ [\text{OF}\ \text{assms}(1) \ \text{fst}]
  obtain \( \varphi :: \text{nat} \Rightarrow \text{nat} \) where \( \lambda i. j < j \Rightarrow \varphi \ i < \varphi \ j \)
  and \( \ast : \lambda i. j < j \Rightarrow ?W \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by \text{auto}

from \text{snd} have \( \forall i. \ (\text{snd} \ (f \ (\varphi \ i))) \in A \) by \text{auto}

then have \( \text{snd} \circ f \circ \varphi \in \text{SEQ} A \) by \text{auto}

with \text{assms} \ (2) have \( \text{good} \ (\text{P}^2) \ (\text{snd} \circ f \circ \varphi) \) by \text{auto}

then obtain \( i \ j :: \text{nat} \)
  where \( i < j \) and \( ?B \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by \text{auto}

with \text{mono} \ [\text{OF} \ (i < j)] have \( ?P \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by \text{simp; case-prod-beta prod-le-def}

with \text{mono} \ [\text{OF} \ (i < j)] \text{ and bad show False by auto}

\text{qed}

14.6 Higman’s Lemma for Almost-Full Relations

\text{lemma Nil-imp-good-list-emb} \ [\text{simp}]:

assumes \( f \ i = [] \)

shows \( \text{good} \ (\text{list-emb} \ P) \ f \)

proof (rule \text{contr})

assume \( \text{bad} \ (\text{list-emb} \ P) \ f \)

moreover have \( (\text{list-emb} \ P) \ (f \ (\text{Suc} \ i)) \)

unfolding \text{assms} by \text{auto}

ultimately show \( \text{False} \)

unfolding \text{good-def} by \text{auto}

\text{qed}

\text{lemma ne-lists}:

assumes \( xs \neq [] \) and \( xs \in \text{lists} \ A \)

shows \( \text{hd} \ xs \in A \) and \( \text{tl} \ xs \in \text{lists} \ A \)

using \text{assms} by (case-tac \[!\] \( xs \)) \text{simp-all}

\text{lemma almost-full-on-lists}:

assumes \( \text{almost-full-on} \ P \ A \)

shows \( \text{almost-full-on} \ (\text{list-emb} \ P) \ (\text{lists} \ A) \) \text{ (is almost-full-on \( ?P \ A \))}

proof (rule \text{contr})

interpret \( \text{mbs} \ ?A \).

assume \( \neg \ ?\text{thesis} \)

from \( \text{mbs'} \ [\text{OF} \ \text{this}] \) obtain \( m \)

where \( \text{bad:} \ m \in \text{BAD} \ ?P \)

and \( \text{min:} \ \forall g. \ (m, g) \in \text{gseq} \rightarrow \text{good} \ ?P \ g \ .. \)

then have \( \text{lists:} \ \lambda i. \ (m \ i) \in \text{lists} \ A \)

and \( \text{ne:} \ \lambda i. \ (m \ i) \neq [] \) by \text{auto}

\( \text{def} \ h \equiv \lambda i. \ \text{hd} \ (m \ i) \)

\( \text{def} \ t \equiv \lambda i. \ \text{tl} \ (m \ i) \)
have ⋀i. m i = h i \# t i using ne by (simp add: h-def t-def)

have \(\forall i. h i \in A\) using ne-lists [OF ne] and lists by (auto simp add: h-def)
from almost-full-on-imp-homogeneous-subseq [OF assms this] obtain \(\varphi :: \text{nat} \Rightarrow \text{nat}\)

where less: \(\forall i j. i < j \Rightarrow \varphi i < \varphi j\)
and \(P: \forall i j. i < j \rightarrow P (h (\varphi i)) (h (\varphi j))\) by blast

have bad-t: bad \(\exists P\) (t \circ \varphi)
proof
  assume good \(\exists P\) (t \circ \varphi)
  then obtain i j where i < j and \(\exists P\) (t (\varphi i)) (t (\varphi j)) by auto
  moreover with \(P\) have \(P (h (\varphi i)) (h (\varphi j))\) by blast
  ultimately have \(\exists P (m (\varphi i)) (m (\varphi j))\)
    by (subst (1 2) m) (rule list-emb-Cons2, auto)
  with less and (i < j) have good \(\exists P\) m by (auto simp: good-def)
  with bad show False by blast
qed

def m' \(\equiv \lambda x_i. \text{if } i < \varphi 0 \text{ then } m i \text{ else } t (\varphi (i - \varphi 0))\)

have m'-less: \(\forall i. i < \varphi 0 \Rightarrow m' i = m i\) by (simp add: m'-def)
have m'-geq: \(\forall i. i \geq \varphi 0 \Rightarrow m' i = t (\varphi (i - \varphi 0))\) by (simp add: m'-def)

have \(\forall i. m' i \in \text{lists } A\) using ne-lists [OF ne] and lists by (auto simp: m'-def t-def)
moreover have length (m' (\varphi 0)) < length (m (\varphi 0)) using ne by (simp add: t-def m'-geq)
moreover have \(\forall j < p 0. m' j = m j\) by (auto simp: m'-less)
ultimately have \((m, m') \in \text{gseq using lists by (auto simp: gseq-def)}\)
moreover have bad \(\exists P\) m'
proof
  assume good \(\exists P\) m'
  then obtain i j where i < j and emb: \(\exists P (m' i) (m' j)\) by (auto simp: good-def)
    
    \{ assume j < \varphi 0
    with \(i < j\) and emb have \(\exists P (m i) (m j)\) by (auto simp: m'-less)
    with \(i < j\) and bad have False by blast \}
  moreover
    \{ assume \(\varphi 0 \leq i\)
    with \(i < j\) and emb have \(\exists P (t (\varphi (i - \varphi 0))) (t (\varphi (j - \varphi 0)))\)
    and \(i - \varphi 0 < j - \varphi 0\) by (auto simp: m'-geq)
    with bad-t have False by auto \}
  moreover
    \{ assume \(i < \varphi 0\) and \(\varphi 0 \leq j\)
    with \(i < j\) and emb have \(\exists P (m i) (t (\varphi (j - \varphi 0)))\) by (simp add: m'-less m'-geq)
    from list-emb-Cons [OF this, of h (\varphi (j - \varphi 0))] have \(\exists P (m i) (m (\varphi (j - \varphi 0)))\) using ne by (simp add: h-def t-def)
moreover have $i < \varphi (j - \varphi 0) \land i < \varphi 0 \land \varphi 0 \leq j$
by \(\text{cases } j = \varphi 0\) \text{ auto}
ultimately have False \text{ using } \text{bad by blast }
ultimately show False \text{ using } (i < j) \text{ by } \text{arith}
qeda
ultimately show False \text{ using } \text{min by blast}
qed

\textbf{lemma} \texttt{list-emb-eq-length-induct} \texttt{[consumes 2, case-names Nil Cons]}
\texttt{[consumes 2, case-names Nil Cons]}:
assumes length \(xs = length ys\)
and \texttt{list-emb P xs ys}
and \(Q \emptyset \emptyset\)
and \(\forall x y. x \in xs. \ [P x y; \texttt{list-emb P xs ys}; Q xs ys] \implies Q (x\#xs) (y\#ys)\)
shows \(Q xs ys\)
using \texttt{assms(2, 1, 3)} by \(\text{induct} \) \(\text{(auto dest: list-emb-length)}\)

\textbf{lemma} \texttt{list-emb-eq-length-P}:
assumes length \(xs = length ys\)
and \texttt{list-emb P xs ys}
shows \(\forall i < \text{length } xs. P \ (xs ! i) \ (ys ! i)\)
using \texttt{assms}
proof \(\text{(induct rule: list-emb-eq-length-induct)}\)
case \(\text{(Cons x y xs ys)}\)
show \(?case\)
proof \(\text{(intro allI impI)}\)
fix \(i\) assume \(i < \text{length } (x \# xs)\)
with \texttt{Cons} show \(P (((x\#xs)\!i) \ ((y\#ys)\!i))\)
by \(\text{(cases } i\) simp-all
qed
qed simp

14.7 Special Case: Finite Sets

Every reflexive relation on a finite set is almost-full.

\textbf{lemma} \texttt{finite-almost-full-on}:
assumes finite: \(\text{finite } A\)
and refl: \(\text{refl-p-on } P A\)
shows \(\text{almost-full-on } P A\)
proof
fix \(f :: \text{nat } \Rightarrow 'a\)
assume \(*:: \forall i. f i \in A\)
let \(\forall I = \text{UNIV}::\text{nat set}\)
have \(f \ i \ \forall I \subseteq A\) using \(*\) by \(\text{auto}\)
with \texttt{finite} and \texttt{finite-subset} have \(1:: \text{finite } (f \ i \ \forall I)\) by \texttt{blast}
have \(\text{infinite } \forall I\) by \texttt{auto}
from \texttt{pigeonhole-infinite} \(\texttt{[OF this 1]}\)
obtain \(k\) where \(\text{infinite } \{j. f j = f k\}\) by \(\texttt{auto}\)
then obtain \(l\) where \(k < l\) and \(f l = f k\)
unfolding infinite-nat-iff-unbounded by auto
then have \( P (f \, k) \, (f \, l) \) using refl and \( \star \) by (auto simp: reflp-on-def)
with \( k < l \) show good \( P \, f \) by (auto simp: good-def)
qed

**lemma** wf-and-no-antichain-imp-qo-extension-wf:

assumes **wf**: wfp-on (strict P) A
and **anti**: ¬ (∃ f. antichain-on P f A)
and **subrel**: ∀ x ∈ A. ∀ y ∈ A. P x y → Q x y
and **qo**: qo-on Q A

shows wfp-on (strict Q) A

**proof** (rule ccontr)

have transp-on (strict Q) A
  using **qo** unfolding go-on-def transp-on-def by blast
then have ∗: transp-on ((strict Q)⁻¹⁻¹) A by (rule transp-on-converse)
assume ¬ wfp-on (strict Q) A
then obtain f :: nat ⇒ 'a where A: ∃ i. f i ∈ A
and ∀ i. strict Q (f (Suc i)) (f i) unfolding wfp-on-def by blast+
then have chain-on ((strict Q)⁻¹⁻¹) f A by auto
from chain-on-transp-on-less [OF this ∗]
  have ∗: ∀ i j. i < j ⇒ ¬ P (f i) (f j)
using subrel and A by blast
show False

**proof** (cases)

assume ∃ k. ∀ i > k. ∃ j > i. P (f j) (f i)
then obtain k where ∀ i > k. ∃ j > i. P (f j) (f i) by auto
from subchain [of k - f, OF this] obtain g
  where ∀ i j. i < j ⇒ g i < g j
and ∀ i. P (f (g (Suc i))) (f (g i)) by auto
with ∗ have ∀ i. strict P (f (g (Suc i))) (f (g i)) by blast
with **wf** [unfolded wfp-on-def not-ex, THEN spec, of λ i. f (g i)] and A
  show False by fast

next
assume ¬ (∃ k. ∀ i > k. ∃ j > i. P (f j) (f i))
then have ∀ k. ∃ i > k. ∀ j > i. ¬ P (f j) (f i) by auto
from choice [OF this] obtain h
  where ∀ k. h k > k
and ∗∗: ∀ k. (∃ j > h k. ¬ P (f j) (f (h k))) by auto
def [simp]: ϕ ≡ λ i. (h "Suc i") 0
have λ i. ϕ i < ϕ (Suc i)
  using ∀ k. h k > k by (induct-tac i) auto
then have mono: λ i j. i < j ⇒ ϕ i < ϕ j by (metis lift-Suc-mono-less)
then have ∀ i j. i < j → ¬ P (f (ϕ j)) (f (ϕ i))
  using ∗∗ by auto
with mono [THEN ∗]
  have ∀ i j. i < j → incomparable P (f (ϕ j)) (f (ϕ i)) by blast
moreover have ∃ i j. i < j ∧ ¬ incomparable P (f (ϕ i)) (f (ϕ j))
  using anti [unfolded not-ex, THEN spec, of λ i. f (ϕ i)] and A by blast
ultimately show False by blast

qed
lemma every-quo-extension-wf-imp-af:

assumes ext: \( \forall Q. (\forall x \in A. \forall y \in A. P x y \rightarrow Q x y) \wedge \\
qo-on Q A \rightarrow wfp-on (\text{strict} Q) A \)

and qo-on P A

shows almost-full-on P A

proof

from (qo-on P A)

have refl: reflp-on P A

and trans: transp-on P A

by (auto intro: qo-on-imp-reflp-on qo-on-imp-transp-on)

fix f :: nat \Rightarrow 'a

assume \( \forall i. f i \in A \).

then have A: \( \forall i. f i \in A \).

show good P f

proof (rule ccontr)

assume \( \neg \)thesis

then have bad: \( \forall i j. i < j \rightarrow \neg P (f i) (f j) \) by (auto simp: good-def)

then have \( \star \): \( \forall i j. P (f i) (f j) \rightarrow i \geq j \) by (metis not-leE)

def [simp]: D \( \equiv \lambda x y. \exists i. x = f \text{Suc} i \wedge y = f i \)

def P' \( \equiv \text{restrict-to P} A \)

def [simp]: Q \( \equiv (\sup P')** \)

have \( \star \star \): \( \forall i j. (D \text{OO} P'**)++ (f i) (f j) \rightarrow i > j \)

proof

fix i j

assume \( (D \text{OO} P'**)+++ (f i) (f j) \)

then show \( i > j \)

apply (induct f i f j arbitrary: j)

apply (insert A, auto dest!: \( \star \) simp: P'-def reflp-on-restrict-to-rtranclp [OF refl trans])

apply (metis \( \star \) dual-order.strict-trans1 less-Suc-eq-le refl reflp-on-def)

by (metis le-imp-less-Suc less-trans)

qed

have \( \forall x \in A. \forall y \in A. P x y \rightarrow Q x y \) by (auto simp: P'-def)

moreover have qo-on Q A by (auto simp: qo-on-def reflp-on-def transp-on-def)

ultimately have wfp-on (strict Q) A

using ext [THEN spec, of Q] by blast

moreover have \( \forall i. f i \in A \wedge \text{strict} Q (f \text{Suc} i) (f i) \)

proof

fix i

have \( \neg Q (f i) (f \text{Suc} i) \)

proof

assume \( Q (f i) (f \text{Suc} i) \)

then have \( (\sup P')** (f i) (f \text{Suc} i) \) by auto

moreover have \( (\sup P')**(\sup P'OO (D OO P'**))+++ \)

proof


have \( \bigwedge A B. (A \cup B)^* = A^* \cup A^* O (B O A^*)^+ \) by regexp
from this [to-pred] show \(?thesis\) by blast
qed
ultimately have \( sup (P^{**}) (P^{**} OO (D OO P^{**})^{++}) (f i) (f (Suc i)) \)
by simp
then have \( (P^{**} OO (D OO P^{**})^{++}) (f i) (f (Suc i)) \) by auto
then have \( Suc i < i \)
using ** apply auto
by (metis (lifting, mono-tags) less-le relcompp.relcompI tranclp-into-tranclp2)
then show \( False \) by auto
qed
with \( A [of \ i] \) show \( f i \in A \land strict Q (f (Suc i)) (f i) \) by auto
qed
ultimately show \( False \) unfolding \( wfp-on-def \) by blast
qed
qed
end

15 Well-Quasi-Orders

theory Well-Quasi-Orders
imports Almost-Full-Relations
begin

15.1 Basic Definitions

definition wqo-on :: \((\forall a. \forall a. \Rightarrow bool) \Rightarrow \forall a. set \Rightarrow bool\) where
wqo-on \( P \ A \leftarrow transp-on \ P \ A \land almost-full-on \ P \ A \)

lemma wqo-on-UNIV:
wqo-on \( (\lambda x. True) \) UNIV
using almost-full-on-UNIV by (auto simp: wqo-on-def transp-on-def)

lemma wqo-onI [Pure.intro]:
\[ \text{transp-on } P \ A; \text{ almost-full-on } P \ A \] \( \Rightarrow \) wqo-on \( P \ A \)
unfolding wqo-on-def almost-full-on-def by blast

lemma wqo-on-imp-reflp-on:
wqo-on \( P \ A \Rightarrow reflp-on \ P \ A \)
using almost-full-on-imp-reflp-on by (auto simp: wqo-on-def)

lemma wqo-on-imp-transp-on:
wqo-on \( P \ A \Rightarrow transp-on \ P \ A \)
by (auto simp: wqo-on-def)

lemma wqo-on-imp-almost-full-on:
wqo-on \( P \ A \Rightarrow almost-full-on \ P \ A \)
by (auto simp: wqo-on-def)

90
lemma wqo-on-imp-qo-on:
\[ wqo-on P A \implies qo-on P A \]
by (metis qo-on-def wqo-on-imp-reflp-on wqo-on-imp-transp-on)

lemma wqo-on-imp-good:
\[ wqo-on P A \implies \forall i. f i \in A \implies good P f \]
by (auto simp: wqo-on-def almost-full-on-def)

lemma wqo-on-subset:
\[ A \subseteq B \implies wqo-on P B \implies wqo-on P A \]
using almost-full-on-subset [of A B P]
and transp-on-subset [of A B P]
unfolding wqo-on-def by blast

15.2 Equivalent Definitions

Given a quasi-order \( P \), the following statements are equivalent:

1. \( P \) is a almost-full.
2. \( P \) does neither allow decreasing chains nor antichains.
3. Every quasi-order extending \( P \) is well-founded.

lemma wqo-af-conv:
assumes qo-on P A
shows  wqo-on P A \iff almost-full-on P A
using assms by (metis qo-on-def wqo-on-def)

lemma wqo-wf-and-no-antichain-conv:
assumes qo-on P A
shows  wqo-on P A \iff wfp-on (strict P) A \land \neg (\exists f. antichain-on P f A)
unfolding wqo-af-conv [OF assms]
using af-trans-imp-wf [OF - assms [THEN qo-on-imp-transp-on]]
and almost-full-on-imp-no-antichain-on [of P A]
and wf-and-no-antichain-imp-qo-extension-wf [of P A]
and every-qo-extension-wf-imp-af [OF - assms]
by blast

lemma wqo-extensions-wf-conv:
assumes qo-on P A
shows  wqo-on P A \iff (\forall Q. (\forall x \in A. \forall y \in A. P x y \implies Q x y) \land qo-on Q A)
\implies wfp-on (strict Q) A
unfolding wqo-af-conv [OF assms]
using af-trans-imp-wf [OF - assms [THEN qo-on-imp-transp-on]]
and almost-full-on-imp-no-antichain-on [of P A]
and wf-and-no-antichain-imp-qo-extension-wf [of P A]
and every-qo-extension-wf-imp-af [OF - assms]
by blast
lemma wqo-on-imp-wfp-on:
  \[ wqo-on \ P \ A \implies wfp-on \ (\text{strict } P) \ A \]
by (metis (no-types) wqo-on-imp-qo-on wqo-wf-and-no-antichain-cone)

The homomorphic image of a wqo set is wqo.

lemma wqo-on-hom:
assumes transp-on Q (h ' A)
  and \( \forall x \in A, \forall y \in A. \ P x y \implies Q (h x) (h y) \)
  and wqo-on P A
shows wqo-on Q (h ' A)
using assms and almost-full-on-hom [of A P Q h]
unfolding wqo-on-def by blast

The monomorphic preimage of a wqo set is wqo.

lemma wqo-on-mon:
assumes \( \forall x \in A, \forall y \in A. \ P x y \iff Q (h x) (h y) \)
  and bij: bij-betw h A B
  and wqo: wqo-on Q B
shows wqo-on P A
proof -
have transp-on P A
proof
  fix x y z assume [intro!]: \( x \in A \ y \in A \ z \in A \)
  and P x y and P y z
  with * have Q (h x) (h y) and Q (h y) (h z) by blast+
  with wqo-on-imp-transp-on [OF wqo] have Q (h x) (h z) by blast
  using bij by (auto simp: bij-betw_def transp-on-def)
  with * show P x z by blast
qed
with assms and almost-full-on-mon [of A P Q h]
show ?thesis unfolding wqo-on-def by blast
qed

15.3 A Type Class for Well-Quasi-Orders

In a well-quasi-order (wqo) every infinite sequence is good.

class wqo = preorder +
  assumes good: good (op \( \leq \)) f

lemma wqo-on-class [simp, intro]:
  wqo-on (op \( \leq \)) (UNIV :: ('a :: wqo) set)
  using good by (auto simp: wqo-on-def transp-on-def almost-full-on-def dest: order-trans)

lemma wqo-on-UNIV-class-wqo [intro!]:
  wqo-on P UNIV \implies class.wqo P (strict P)
by (unfold-locales) (auto simp: wqo-on-def almost-full-on-def, unfold transp-on-def, blast)
The following lemma converts between \textit{wqo-on} (for the special case that the domain is the universe of a type) and the class predicate \textit{class.wqo}.

\begin{lemma}\textit{wqo-on-UNIV-conv:}\end{lemma}
\begin{align*}
\text{wqo-on P UNIV } & \iff \text{class.wqo P (strict P)} \ (\text{is } \text{?lhs = ?rhs}) \\
\text{proof} & \\
\text{assume } \text{?lhs then show } \text{?rhs by auto} \\
\text{next} & \\
\text{assume } \text{?rhs then show } \text{?lhs by auto} \\
\text{unfolding } & \text{class.wqo-def class.preorder-def class.wqo-axioms-def} \\
\text{by } & \text{(auto simp: wqo-on-def almost-full-on-def transp-on-def)}
\end{align*}
\begin{proof}
\text{The strict part of a wqo is well-founded.}
\begin{lemma}\textit{(in wqo) wfP (op <)}\end{lemma}
\begin{proof}
\text{−}
\begin{align*}
\text{have class.wqo (op \leq) (op <) ..} \\
\text{hence wqo-on (op \leq) UNIV} \\
\text{unfolding less-le-not-le [abs-def] wqo-on-UNIV-conv [symmetric] .} \\
\text{from wqo-on-imp-wfp-on [OF this]} \\
\text{show ?thesis unfolding less-le-not-le [abs-def] wfp-on-UNIV .}
\end{align*}
\begin{qed}
\end{proof}
\begin{lemma}\textit{wqo-on-with-bot:}\end{lemma}
\begin{align*}
\text{assumes wqo-on P A} \\
\text{shows wqo-on (option-le P) A\bot (is wqo-on ?P ?A)} \\
\text{proof } \\
\{ \text{from assms have trans [unfolded transp-on-def]: transp-on P A} \\
\text{by (auto simp: wqo-on-def)} \\
\text{have transp-on ?P ?A} \\
\text{by (auto simp: transp-on-def elim!: with-bot-cases, insert trans) blast } \}
\text{moreover} \\
\{ \text{from assms and almost-full-on-with-bot} \\
\text{have almost-full-on ?P ?A by (auto simp: wqo-on-def)} \}
\text{ultimately} \\
\text{show ?thesis by (auto simp: wqo-on-def)}
\end{align*}
\begin{qed}
\end{proof}
\begin{lemma}\textit{wqo-on-option-UNIV [intro]:}\end{lemma}
\begin{align*}
\text{wqo-on P UNIV } & \implies \text{wqo-on (option-le P) UNIV} \\
\text{using wqo-on-with-bot [of P UNIV] by simp}
\end{align*}
\begin{proof}
\text{When two sets are wqo, then their disjoint sum is wqo.}
\begin{lemma}\textit{wqo-on-Plus:}\end{lemma}
\begin{align*}
\text{assumes wqo-on P A and wqo-on Q B} \\
\text{shows wqo-on (sum-le P Q) (A <+> B) (is wqo-on ?P ?A)} \\
\text{proof } \\
\{ \text{from assms have trans [unfolded transp-on-def]: transp-on P A transp-on Q} \\
\text{B} \\
\text{by (auto simp: wqo-on-def)}
\end{align*}
\end{proof}
have transp-on ?P ?A
  unfolding transp-on-def by (auto, insert trans) (blast+ )
moreover
{ from assms and almost-full-on-Plus have almost-full-on ?P ?A by (auto simp: wqo-on-def) }
ultimately
show ?thesis by (auto simp: wqo-on-def)
qed

lemma wqo-on-sum-UNIV [intro]:
  wqo-on P UNIV ⊢ wqo-on Q UNIV ⊢ wqo-on (sum-le P Q) UNIV
using wqo-on-Plus [of P UNIV Q UNIV] by simp

15.4 Dickson’s Lemma

lemma wqo-on-Sigma:
  fixes A1 :: 'a set and A2 :: 'b set
  assumes wqo-on P1 A1 and wqo-on P2 A2
  shows wqo-on (prod-le P1 P2) (A1 × A2) (is wqo-on ?P ?A)
proof –
{ from assms have transp-on P1 A1 and transp-on P2 A2 by (auto simp: wqo-on-def)
  hence transp-on ?P ?A unfolding transp-on-def prod-le-def by blast }
moreover
{ from assms and almost-full-on-Sigma [of P1 A1 P2 A2]
  have almost-full-on ?P ?A by (auto simp: wqo-on-def) }
ultimately
show ?thesis by (auto simp: wqo-on-def)
qed

lemmas dickson = wqo-on-Sigma

lemma wqo-on-prod-UNIV [intro]:
  wqo-on P UNIV ⊢ wqo-on Q UNIV ⊢ wqo-on (prod-le P Q) UNIV
using wqo-on-Sigma [of P UNIV Q UNIV] by simp

15.5 Higman’s Lemma

lemma transp-on-list-emb:
  assumes transp-on P A
  shows transp-on (list-emb P) (lists A)
using assms and list-emb-trans [of - - - P]
unfolding transp-on-def by blast

lemma wqo-on-lists:
  assumes wqo-on P A shows wqo-on (list-emb P) (lists A)
using assms and almost-full-on-lists
  and transp-on-list-emb by (auto simp: wqo-on-def)

lemmas higman = wqo-on-lists
lemma wqo-on-list-UNIV [intro]:
\[
\text{wqo-on } P \text{ UNIV } \Rightarrow \text{wqo-on } (\text{list-emb } P) \text{ UNIV}
\]
using wqo-on-lists [of P UNIV] by simp

Every reflexive and transitive relation on a finite set is a wqo.

lemma finite-wqo-on:
assumes finite A and refl: reflp-on P A and transp-on P A
shows wqo-on P A
using assms and finite-almost-full-on
by (auto simp: wqo-on-def)

lemma finite-eq-wqo-on:
assumes finite A
shows wqo-on (op =) A
using finite-wqo-on OF assms
by (auto simp: reflp-on-def transp-on-def)

lemma wqo-on-lists-over-finite-sets:
\[
\text{wqo-on } (\text{list-emb } (\text{op }=)) \text{ (UNIV::('a::finite) list)} \text{ UNIV}
\]
using wqo-on-lists [OF finite-eq-wqo-on OF finite]
by simp

lemma wqo-on-map:
fixes P and Q and h
defines P' \equiv \lambda x y. P x y \wedge Q (h x) (h y)
assumes wqo-on P A
and wqo-on Q B
and subset: h ' A \subseteq B
shows wqo-on P' A
proof
let \(\forall Q = \lambda x y. Q (h x) (h y)\)
from \(\langle wqo-on P A \rangle\) have transp-on P A
by (rule wqo-on-imp-transp-on)
then show transp-on P' A
using \(\langle wqo-on Q B \rangle\) and subset
unfolding wqo-on-def transp-on-def P'-def by blast
from \(\langle wqo-on P A \rangle\) have almost-full-on P A
by (rule wqo-on-imp-almost-full-on)
from \(\langle wqo-on Q B \rangle\) have almost-full-on Q B
by (rule wqo-on-imp-almost-full-on)
show almost-full-on P' A
proof
fix f
assume \(\forall i :: \text{nat}. f i \in A\)
from almost-full-on-imp-homogeneous-subseq [OF \(\langle \text{almost-full-on } P A \rangle\) this]
obtain g :: nat \Rightarrow nat
where g: \(\forall i j. i < j \Rightarrow g i < g j\)
\[\forall i. f (g i) \in A \land P (f (g i)) (f (g (Suc i)))\]

using \texttt{* by auto}

from \texttt{chain-on-transp-on-less [OF \texttt{*} (transp-on P A)]}

have \texttt{\texttt{*} \land i < j \implies P (f (g i)) (f (g j))}.

let \texttt{?g = \lambda i. h (f (g i))}

from \texttt{* and subset have B: \ \land i. ?g i \in B by auto}

with \texttt{(almost-full-on Q B) [unfolded almost-full-on-def good-def, THEN bspec, OF \texttt{*}]}

\begin{itemize}
    \item obtain \texttt{i j :: nat}
    \item where \texttt{i < j and Q (?g i) (?g j) by blast}
    \item with \texttt{OF \texttt{*} \land i < j] have \texttt{P' (f (g i)) (f (g j))}
    \item by (auto simp: \texttt{P'-def})
    \item with \texttt{g [OF \texttt{*} \land i < j] show \texttt{good P' f by (auto simp: good-def})}
\end{itemize}

qed

\texttt{lemma wqo-on-UNIV-nat:}

\texttt{wqo-on (op \texttt{\le}) (UNIV :: nat set) [unfolded wqo-on-def transp-on-def]}

using \texttt{almost-full-on-UNIV-nat by simp}

end

theory Closures2
imports Closures ../Well-Quasi-Orders/Well-Quasi-Orders
begin

\section{Closure under \texttt{SUBSEQ} and \texttt{SUPSEQ}}

Properties about the embedding relation

\texttt{lemma sublisteq-strict-length:}

\begin{itemize}
    \item assumes \texttt{a: sublisteq x y x \neq y}
    \item shows \texttt{length x < length y}
    \item using \texttt{a}
    \item by (induct) (auto simp add: less-Suc-eq)
\end{itemize}

\texttt{lemma sublisteq-wf:}

\begin{itemize}
    \item shows \texttt{wf \{ (x, y). sublisteq x y \land x \neq y\}}
    \item proof --
    \item have \texttt{wf (measure length) by simp}
    \item moreover
    \item have \texttt{\{ (x, y). sublisteq x y \land x \neq y\} \subseteq measure length}
    \item unfolding \texttt{measure-def by (auto simp add: sublisteq-strict-length)}
    \item ultimately
    \item show \texttt{wf \{ (x, y). sublisteq x y \land x \neq y\} by (rule wf-subset)}
\end{itemize}

qed
lemma sublisteq-good:
  shows good sublisteq (f :: nat ⇒ ('a::finite) list)
using wqo-on-imp-good[where f=f, OF wqo-on-lists-over-finite-sets]
by simp

lemma sublisteq-Higman-antichains:
  assumes a: ∀(x::('a::finite) list) ∈ A. ∀ y ∈ A. x ≠ y \implies ¬(sublisteq x y) ∧ ¬(sublisteq y x)
  shows finite A
proof (rule ccontr)
  assume infinite A
  then obtain f::nat ⇒ 'a::finite list where b: inj f and c: range f ⊆ A
  by (auto simp add: infinite-iff-countable-subset)
  from sublisteq-good[where f=f]
  obtain i j where d: i < j and e: sublisteq (f i) (f j) ∨ f i = f j
  unfolding good-def
  by auto
  have f i ≠ f j using b d by (auto simp add: inj-on-def)
  moreover
  have f i ∈ A using c by auto
  moreover
  have f j ∈ A using c by auto
  ultimately have ¬(sublisteq (f i) (f j)) using a by simp
  with e show False by auto
qed

16.1 Sub- and Supersequences

definition SUBSEQ A ≡ {x::('a::finite) list. ∃ y ∈ A. sublisteq x y}

definition SUPSEQ A ≡ {x::('a::finite) list. ∃ y ∈ A. sublisteq y x}

lemma SUPSEQ-simps [simp]:
  shows SUPSEQ {} = {}
  and SUPSEQ {{}} = UNIV
unfolding SUPSEQ-def by auto

lemma SUPSEQ-atom [simp]:
  shows SUPSEQ {[c]} = UNIV · {[c]} · UNIV
unfolding SUPSEQ-def conc-def
by (auto dest: list-emb-ConsD)

lemma SUPSEQ-union [simp]:
  shows SUPSEQ (A ∪ B) = SUPSEQ A ∪ SUPSEQ B
unfolding SUPSEQ-def by auto

lemma SUPSEQ-conc [simp]:

97
shows \( \text{SUPSEQ} (A \cdot B) = \text{SUPSEQ} A \cdot \text{SUPSEQ} B \)
unfolding \( \text{SUPSEQ-def conc-def} \)
apply(auto)
apply(drule list-emb-appendD)
apply(auto)
by (metis list-emb-append-mono)

lemma \( \text{SUPSEQ-star [simp]} \):
  shows \( \text{SUPSEQ} (A^*) = \text{UNIV} \)
apply(subst star-unfold-left)
apply(simp only: SUPSEQ-union)
apply(simp)
done

16.2 Regular expression that recognises every character

definition
Allreg :: 'a::finite rexp
where
Allreg ≡ \( \biguplus \) (Atom \( \cdot \) UNIV)

lemma Allreg-lang [simp]:
  shows \( \text{lang Allreg} = (\bigcup a. \{[a]\}) \)
unfolding Allreg-def by auto

lemma [simp]:
  shows \( (\bigcup a. \{[a]\})^* = \text{UNIV} \)
apply(auto)
apply(induct-tac x)
apply(auto)
apply(subgoal-tac [a] \in (\bigcup a. \{[a]\})^*)
apply(simp)
apply(rule append-in-starI)
apply(auto)
done

lemma Star-Allreg-lang [simp]:
  shows \( \text{lang} (\text{Star Allreg}) = \text{UNIV} \)
by simp

fun
  UP :: 'a::finite rexp ⇒ 'a rexp
where
  UP Zero = Zero |
  UP (One) = Star Allreg |
  UP (Atom c) = Times (Star Allreg) (Times (Atom c) (Star Allreg)) |
  UP (Plus r1 r2) = Plus (UP r1) (UP r2) |
  UP (Times r1 r2) = Times (UP r1) (UP r2) |
  UP (Star r) = Star Allreg
lemma `lang-UP`:
  fixes `r::'a::finite rexp`
  shows `lang (UP r) = SUPSEQ (lang r)`
  by (induct `r`) (simp-all)

lemma `SUPSEQ-regular`:
  fixes `A::'a::finite lang`
  assumes `regular A`
  shows `regular (SUPSEQ A)`
  proof –
    from `assms` obtain `r::'a::finite rexp` where `lang r = A` by auto
    then have `lang (UP r) = SUPSEQ A` by (simp add: `lang-UP`) 
    then show `regular (SUPSEQ A)` by auto
  qed

lemma `SUPSEQ-subset`:
  fixes `A::'a::finite list set`
  shows `A ⊆ SUPSEQ A`
  unfolding `SUPSEQ-def` by auto

lemma `SUBSEQ-complement`:
  shows `− (SUBSEQ A) = SUPSEQ (− (SUBSEQ A))`
  proof –
    have `− (SUBSEQ A) ⊆ SUPSEQ (− (SUBSEQ A))`
      by (rule `SUPSEQ-subset`)
    moreover
    have `SUPSEQ (− (SUBSEQ A)) ⊆ − (SUBSEQ A)`
      proof (rule ccontr)
        assume `¬ (SUPSEQ (− (SUBSEQ A)) ⊆ − (SUBSEQ A))`
        then obtain `x` where `a: x ∈ SUPSEQ (− (SUBSEQ A))` and
          `b: x /∈ − (SUBSEQ A)` by auto
        from `a` obtain `y` where `c: y ∈ − (SUBSEQ A)` and `d: sublisteq y x`
          by (auto simp add: `SUPSEQ-def`)
        from `b` have `x ∈ SUBSEQ A` by simp
        then obtain `x'` where `f: x' ∈ A` and `e: sublisteq x x'`
          by (auto simp add: `SUBSEQ-def`)
        from `d e` have `sublisteq y x'`
          by (rule `sublisteq-trans`)
        then have `y ∈ SUBSEQ A` using `f`
          by (auto simp add: `SUBSEQ-def`)
        with `e` show `False` by simp
      qed
    ultimately show `− (SUBSEQ A) = SUPSEQ (− (SUBSEQ A))` by simp
  qed
definition

\texttt{minimal} :: 'a::finite list \Rightarrow 'a lang \Rightarrow bool

where

\texttt{minimal x A} \equiv (\forall y \in A. \texttt{sublisteq y x} \rightarrow \texttt{sublisteq x y})

lemma \texttt{main-lemma}:

shows \( \exists M. \text{finite M} \land \texttt{SUPSEQ A} = \texttt{SUPSEQ M} \)

proof –

definition \texttt{M} \equiv \{ x \in A. \texttt{minimal x A} \}

have \texttt{finite M}

unfolding \texttt{M-def minimal-def}

by (rule sublisteq-Higman-antichains) (auto simp add: sublisteq-antisym)

moreover

have \texttt{SUPSEQ A} \subseteq \texttt{SUPSEQ M}

proof

fix \( x \)

assume \( x \in \texttt{SUPSEQ A} \)

then obtain \( y \) where \( y \in A \) and \( \texttt{sublisteq y x} \)

by (auto simp add: SUPSEQ-def)

then have \( a \): \( y \in \{ y' \in A. \texttt{sublisteq y'} x \} \)

by simp

obtain \( z \) where \( b \): \( z \in A \texttt{ sublisteq z x} \) and \( c \): \( \forall y. \texttt{sublisteq y z} \land y \neq z \rightarrow y \notin \{ y' \in A. \texttt{sublisteq y'} x \} \)

using \texttt{wfE-min[OF sublisteq-wf a]} by auto

then have \( z \in M \)

unfolding \texttt{M-def minimal-def}

by (auto intro: sublisteq-trans)

with \( b(2) \) show \( x \in \texttt{SUPSEQ M} \)

by (auto simp add: SUPSEQ-def)

qed

moreover

have \texttt{SUPSEQ M} \subseteq \texttt{SUPSEQ A}

by (auto simp add: SUPSEQ-def M-def)

ultimately

show \( \exists M. \text{finite M} \land \texttt{SUPSEQ A} = \texttt{SUPSEQ M} \)

by blast

qed

16.3 Closure of \texttt{SUBSEQ} and \texttt{SUPSEQ}

lemma \texttt{closure-SUPSEQ}:

fixes A::'a::finite lang

shows \texttt{regular (SUPSEQ A)}

proof –

obtain \( M \) where \( a \): \texttt{finite M} and \( b \): \texttt{SUPSEQ A} = \texttt{SUPSEQ M}

using \texttt{main-lemma} by blast

have \texttt{regular M} using \( a \) by (rule finite-regular)

then have \texttt{regular (SUPSEQ M)} by (rule SUPSEQ-regular)

then show \texttt{regular (SUPSEQ A)} using \( b \) by simp

qed
lemma closure-SUBSEQ:
  fixes A::'a::finite lang
  shows regular (SUBSEQ A)
proof –
  have regular (SUPSEQ (¬ SUBSEQ A)) by (rule closure-SUPSEQ)
  then have regular (¬ SUBSEQ A) by (subst SUBSEQ-complement) (simp)
  then have regular (¬ (¬ (SUBSEQ A))) by (rule closure-complement)
  then show regular (SUBSEQ A) by simp
qed

References

     Automata Constructions. Theoretical Computer Science, 155:291–319,
     1995.

     Theorem based on Regular Expressions (Proof Pearl). In Proc. of the
     2nd International Conference on Interactive Theorem Proving, volume