Abstract

Based on Isabelle/HOL’s type class for preorders, we introduce a type class for well-quasi-orders (wqo) which is characterized by the absence of “bad” sequences (our proofs are along the lines of the proof of Nash-Williams [1], from which we also borrow terminology). Our main results are instantiations for the product type, the list type, and a type of finite trees, which (almost) directly follow from our proofs of (1) Dickson’s Lemma, (2) Higman’s Lemma, and (3) Kruskal’s Tree Theorem. More concretely:

1. If the sets $A$ and $B$ are wqo then their Cartesian product is wqo.
2. If the set $A$ is wqo then the set of finite lists over $A$ is wqo.
3. If the set $A$ is wqo then the set of finite trees over $A$ is wqo.

Contents

1 Infinite Sequences 2
  1.1 Lexicographic Order on Infinite Sequences . . . . . . . . . . . 3

2 Minimal elements of sets w.r.t. a well-founded and transitive relation 4

3 Enumerations of Well-Ordered Sets in Increasing Order 7

4 The Almost-Full Property 8
  4.1 Basic Definitions and Facts . . . . . . . . . . . . . . . . . . . . . . . . 9
  4.2 An equivalent inductive definition . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  4.3 Special Case: Finite Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  4.4 Further Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

5 Constructing Minimal Bad Sequences 20

*The research was funded by the Austrian Science Fund (FWF): J3202.
1 Infinite Sequences

Some useful constructions on and facts about infinite sequences.

theory Infinite-Sequences
imports Main
begin

The set of all infinite sequences over elements from $A$.

definition SEQ $A$ = \{ $f$::nat $\Rightarrow \forall i. f\ i \in A$ \}

lemma SEQ-iff [iff]:
  $f \in SEQ A$ $\iff$ ($\forall i. f\ i \in A$)
by (auto simp: SEQ-def)
The $i$-th "column" of a set $B$ of infinite sequences.

**definition** $ith B i = \{ f i \mid f . f \in B \}$

**lemma** $ithI$ [intro]:

$f \in B \implies f i = x \implies x \in ith B i$

by (auto simp: $ith$-def)

**lemma** $ithE$ [elim]:

$[ x \in ith B i ; \forall f . [ f \in B ; f i = x ] \implies Q ] \implies Q$

by (auto simp: $ith$-def)

**lemma** $ith$-conv:

$x \in ith B i \iff (\exists f . f . x = f i)$

by auto

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

**definition** $eq$-upto :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{set}$

where

$eq$-upto $B f i = \{ g . \forall j < i . f j = g j \}$

**lemma** $eq$-uptoI [intro]:

$[ g \in B ; \forall j . j < i \implies f j = g j ] \implies g \in eq$-upto $B f i$

by (auto simp: $eq$-upto-def)

**lemma** $eq$-uptoE [elim]:

$[ g \in eq$-upto $B f i ; [ g \in B ; \forall j . j < i \implies f j = g j ] \implies Q ] \implies Q$

by (auto simp: $eq$-upto-def)

**lemma** $eq$-upto-Suc:

$[ g \in eq$-upto $B f i ; g i = f i ] \implies g \in eq$-upto $B f (Suc i)$

by (auto simp: $eq$-upto-def less-$Suc$-eq)

**lemma** $eq$-upto-0 [simp]:

$eq$-upto $B f 0 = B$

by (auto simp: $eq$-upto-def)

**lemma** $eq$-upto-cong [fundef-cong]:

assumes $\forall j . j < i \implies f j = g j$ and $B = C$

shows $eq$-upto $B f i = eq$-upto $C g i$

using assms by (auto simp: $eq$-upto-def)

**1.1** Lexicographic Order on Infinite Sequences

**definition** $LEX P f g \longleftrightarrow (\exists i . : \text{nat} . P (f i) (g i) \land (\forall j < i . f j = g j))$

**abbreviation** $LEXEQ P \equiv (LEX P)=$

**lemma** $LEX$-imp-not-$LEX$:

assumes $LEX P f g$

by (auto simp: $LEX$-def)
and \[ \exists ! x y z. P x y \implies P y z \implies P x z \]
and \[ \forall x. \neg P x x \]
suggests \( \neg \text{LEX } P \ g \ f \)

proof

\[
\begin{align*}
\text{fix } i & \ :: \ \text{nat} \\
\text{assume } & \ P (f \ i) (g \ i) \ \text{and } \ \forall k < i. \ f \ k = g \ k \\
\text{and } & \ P (g \ j) (f \ j) \ \text{and } \ \forall k < j. \ g \ k = f \ k \\
\text{then have } & \ \text{False by (cases } i < j \text{) (auto simp: not-less dest: le-imp-less-or-eq)}
\end{align*}
\]

\[
\text{then show } \neg \text{LEX } P \ g \ f \ \text{using } \langle \text{LEX } P \ f \ g \rangle \ \text{unfolding LEX-def by blast}
\]

qed

lemma LEX-cases:
\[
\begin{align*}
\text{assumes } & \ P f g \\
\text{obtains } & \ (eq) f = g \ | \ (neq) k \ \text{where } \ \forall i < k. \ f \ i = g \ i \ \text{and } P (f \ k) (g \ k)
\end{align*}
\]

using assms by (auto simp: LEX-def)

lemma LEX-imp-less:
\[
\begin{align*}
\text{assumes } & \ \forall x \in A. \ \neg P x x \ \text{and } f \in \text{SEQ } A \ \vee \ g \in \text{SEQ } A \\
\text{and } & \ \text{LEX } P f g \ \text{and } \ \forall i < k. \ f \ i = g \ i \ \text{and } f \ k \neq g \ k \\
\text{shows } & \ P (f \ k) (g \ k)
\end{align*}
\]

using assms by (auto elim!: LEX-cases) (metis linorder-neqE-nat)+

end

\section{Minimal elements of sets w.r.t. a well-founded and transitive relation}

theory Minimal-Elements

imports

Infinite-Sequences

Open-Induction.Restricted-Predicates

begin

locale minimal-element =

\fixes P A
\assumes po: \text{po-on } P A
\and \text{wf: wfp-on } P A

begin

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

lemma minimal:
\[
\begin{align*}
\text{assumes } & \ x \in A \ \land Q x \\
\text{shows } & \ \exists y \in A. \ P x y \land Q y \land (\forall z \in A. \ P z y \implies \neg Q z)
\end{align*}
\]

using \text{wf} \and \text{assms}

proof (induction rule: wfp-on-induct)

\text{case } (\text{less } x)

4
then show \(?case

proof (cases \(\forall y \in A.\ P\ y\ x \to \neg Q\ y\))

case True

with less show \(?thesis by blast

next

case False

then obtain \(y\) where \(y \in A\) and \(P\ y\ x\) and \(Q\ y\) by blast

with less show \(?thesis

using po \([THEN\ po-on-imp\-transp\-on,\ unfolded\ transp\-on\-def,\ rule\-format,\ of\ -\ y\ x]\) by blast

qed

qed

lemma min-elt-ex:

assumes \(B \subseteq A\) and \(B \neq \{}\)

shows \(\exists x.\ x \in B \land (\forall y \in A.\ P\ y\ x \to y \notin B)\)

using assms using minimal \([of - \lambda x.\ x\in B]\) by auto

lemma min-elt-mem:

assumes \(B \subseteq A\) and \(B \neq \{}\)

shows \(min\-elt\ B \in B\)

using someI-ex \([OF\ min\-elt\-ex\ [OF\ assms]]\) by (auto simp: min-elt-def)

lemma min-elt-minimal:

assumes *: \(B \subseteq A\) \(B \neq \{}\)

assumes \(y \in A\) and \(P\ y\ (min\-elt\ B)\)

shows \(y \notin B\)

using someI-ex \([OF\ min\-elt\-ex\ [OF\ *]]\) and assms by (auto simp: min-elt-def)

A lexicographically minimal sequence w.r.t. a given set of sequences \(C\)

fun lexmin

where

\(lexmin;\ lexmin\ C\ i = min\-elt\ (ith\ eq\-upto\ C\ (lexmin\ C)\ i)\ i)\)

declare lexmin [simp del]

lemma eq-upto-lexmin-non-empty:

assumes \(C \subseteq SEQ\ A\) and \(C \neq \{}\)

shows eq-upto \(C\) \((lexmin\ C)\ i\) \(\neq \{}\)

proof (induct i)

case 0

show \(?case using assms by auto

next

let \(A = \lambda i.\ ith\ eq\-upto\ C\ (lexmin\ C)\ i)\ i)\)

case (Suc i)

then have \(?A\ i\ \neq \{\}\) by force

moreover have eq-upto \(C\) \((\lexmin\ C)\ i\) \(\subseteq\ eq\-upto\ C\) \((\lexmin\ C)\ 0\) by auto

ultimately have \(?A\ i\ \subseteq A\) and \(?A\ i\ \neq \{\}\) using assms by (auto simp: ith-def)

from min-elt-mem \([OF\ this,\ folded\ lexmin]\)

obtain \(f\) where \(f \in eq\-upto\ C\) \((\lexmin\ C)\) \((Suc\ i)\) by (auto dest: eq-upto-Suc)
then show \( \text{case by blast} \)

qed

lemma \text{lexmin-SEQ-mem}:
  assumes \( C \subseteq \text{SEQ } A \) and \( C \neq \{\} \)
  shows \( \text{lexmin } C \in \text{SEQ } A \)

proof (cases)
  { fix \( i \)
    let \( X = \text{ith (eq-upto } C \text{ (lexmin } C \text{) } i) \)
    have \( \{X \subseteq A \text{ using assms by (auto simp: ith-def)} \}
    moreover have \( \{X \neq \{\} \text{ using eq-upto-lexmin-non-empty[OF assms] by auto} \}
    ultimately have \( \text{lexmin } C \in A \text{ using min-elt-mem[of } X \text{] by (subst lexmin) blast} \}

then show \( \text{thesis by auto} \)

qed

lemma \text{non-empty-ith}:
  assumes \( C \subseteq \text{SEQ } A \) and \( C \neq \{\} \)
  shows \( \text{ith (eq-upto } C \text{ (lexmin } C \text{) } i) \subseteq A \)
  and \( \text{ith (eq-upto } C \text{ (lexmin } C \text{) } i) \neq \{\} \)

using \( \text{eq-upto-lexmin-non-empty[OF assms, of i] and assms by (auto simp: ith-def)} \)

lemma \text{lexmin-minimal}:
  \( C \subseteq \text{SEQ } A \Rightarrow C \neq \{\} \Rightarrow y \in A \Rightarrow P y \text{ (lexmin } C \text{) } i \Rightarrow y \neq \text{ith (eq-upto } C \text{ (lexmin } C \text{) } i) \)

using \( \text{min-elt-minimal[OF non-empty-ith, folded lexmin]} \).

lemma \text{lexmin-mem}:
  \( C \subseteq \text{SEQ } A \Rightarrow C \neq \{\} \Rightarrow \text{lexmin } C \in \text{ith (eq-upto } C \text{ (lexmin } C \text{) } i) \)

using \( \text{min-elt-mem[OF non-empty-ith, folded lexmin]} \).

lemma \text{LEX-chain-on-eq-upto-imp-ith-chain-on}:
  assumes \text{chain-on } (\text{LEX } P) \text{ (eq-upto } C f i) \text{ (SEQ } A) \)
  shows \text{chain-on } P \text{ (ith (eq-upto } C f i) \text{) } A \)

using \text{assms}

proof (cases)
  { fix \( x \) \( y \) assume \( x \in \text{ith (eq-upto } C f i) \) \( y \in \text{ith (eq-upto } C f i) \)
and \( \neg P x y \) \( y \neq x \)

then obtain \( g h \) \text{ where } \(*: g \in \text{eq-upto } C f i h \in \text{eq-upto } C f i \)
and \( \text{simp}: x = g i y = h i \) \( \text{and eq: } \forall j<i. \) \( g j = f j \) \( \land h j = f j \)

by (auto simp: ith-def eq-upto-def)

with \text{assms and } \(*: y \neq x \) \text{ consider } \text{LEX } P g h \text{ | LEX } P h g \text{ by (force simp: chain-on-def)}

then have \( P y x \)

proof (cases)
  assume \( \text{LEX } P g h \)
with \text{eq and } \(*: y \neq x \) have \( P x y \) \text{ using \text{assms and } *}

by (auto simp: \text{LEX-def})
(metis SEQ-iff chain-on-imp-subset linorder-neqE-nat minimal subsetCE)
with (∼P x y) show P y x ..
next
assume LEX P h g
with eq and (y ≠ x) show P y x using assms and *
  by (auto simp: LEX-def)
  (metis SEQ-iff chain-on-imp-subset linorder-neqE-nat minimal subsetCE)
qed }
then show ?thesis using assms by (auto simp: chain-on-def) blast
qed

end
end

3 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 $\text{infm}: \forall i\geq N. \exists j>i. \ P\ i\ j$
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 $\text{enumchain} :: \text{nat} \Rightarrow 'a$ where
\begin{align*}
\text{enumchain } 0 &= N \\
\text{enumchain } (\text{Suc } n) &= (\text{LEAST } m. \ m > \text{enumchain } n \land P \ (\text{enumchain } n) \ m)
\end{align*}

lemma $\text{enumchain-mono}$:
shows $N \leq \text{enumchain } i \land \text{enumchain } i < \text{enumchain } (\text{Suc } i)$
proof (induct $i$
\begin{align*}
\text{case } 0 & \quad \text{have } \text{enumchain } 0 \geq N \quad \text{by simp} \\
& \quad \text{moreover then have } \exists m>\text{enumchain } 0. \ P \ (\text{enumchain } 0) \ m \quad \text{using } \text{infm} \quad \text{by blast} \\
& \quad \quad \text{ultimately show } \text{?case by auto (metis (lifting) LeastI)} \\
\text{next} \\
\text{case } (\text{Suc } i) & \quad \text{then have } N \leq \text{enumchain } (\text{Suc } i) \quad \text{by auto} \\
& \quad \text{moreover then have } \exists m>\text{enumchain } (\text{Suc } i). \ P \ (\text{enumchain } (\text{Suc } i)) \ m \quad \text{using } \text{infm} \quad \text{by blast} \\
& \quad \quad \text{ultimately show } \text{?case by auto (metis (lifting) LeastI)} \\
\text{qed}
\end{align*}

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

end

end

4 The Almost-Full Property

theory Almost-Full
imports
lemma le-Suc-eq':
\( x \leq \text{Suc} y \leftrightarrow x = 0 \lor (\exists x'. x = \text{Suc} x' \land x' \leq y) \)
by (cases x) auto

lemma ex-leq-Suc:
\( (\exists i \leq \text{Suc} j. P i) \leftrightarrow P 0 \lor (\exists i \leq j. P (\text{Suc} i)) \)
by (auto simp: le-Suc-eq')

lemma ex-less-Suc:
\( (\exists i < \text{Suc} j. P i) \leftrightarrow P 0 \lor (\exists i < j. P (\text{Suc} i)) \)
by (auto simp: less-Suc-eq-0-disj)

4.1 Basic Definitions and Facts

An infinite sequence is good whenever there are indices \( i < j \) such that \( P (f i) (f j) \).

definition good :: ('a ⇒ 'a ⇒ bool) ⇒ (nat ⇒ 'a) ⇒ bool
where
\[ \text{good} P f \leftrightarrow (\exists i j. i < j \land P (f i) (f j)) \]

A sequence that is not good is called bad.

abbreviation bad P f ≡ ¬ good P f

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

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

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

definition almost-full-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool
where
almost-full-on $P A \iff (\forall f \in \text{SEQ } A. \text{good } P f)$

**lemma** almost-full-onI [Pure.intro]:
\begin{align*}
(\forall f. \forall i. f i \in A \implies \text{good } P f) \implies \text{almost-full-on } P A
\end{align*}

unfolding almost-full-on-def by blast

**lemma** almost-full-onD:
\begin{align*}
\text{fixes } & f :: \text{nat} \Rightarrow 'a \text{ and } A :: 'a \text{ set} \\
\text{assumes } & \text{almost-full-on } P A \text{ and } \forall i. f i \in A \\
\text{obtains } & i j \text{ where } i < j \text{ and } P (f i) (f j)
\end{align*}

using assms unfolding almost-full-on-def by blast

4.2 An equivalent inductive definition

**inductive** af for $A$

where
\begin{align*}
\text{now: } & (\forall x y. x \in A \implies y \in A \implies P x y) \implies \text{af } A P \\
\text{later: } & (\forall x. x \in A \implies \text{af } A (\lambda y z. P y z \lor P x y)) \implies \text{af } A P
\end{align*}

**lemma** af-imp-almost-full-on:
\begin{align*}
\text{assumes } & \text{af } A P \\
\text{shows } & \text{almost-full-on } P A \\
\text{proof}
\end{align*}

\begin{align*}
\text{fix } & f :: \text{nat} \Rightarrow 'a \text{ assume } \forall i. f i \in A \\
\text{with assms obtain } & i j \text{ where } i < j \text{ and } P (f i) (f j)
\end{align*}

\begin{align*}
\text{proof} \text{ (induct arbitrary: } f \text{ thesis)}
\end{align*}

\begin{align*}
\text{case (later } P) \\
\text{define } & g \text{ where } \text{simp}: g i = f (\text{Suc } i) \text{ for } i \\
\text{have } & f 0 \in A \text{ and } \forall i. g i \in A \text{ using later by auto} \\
\text{then obtain } & i j \text{ where } i < j \text{ and } P (g i) (g j) \lor P (f 0) (g i) \text{ using later by blast} \\
\text{then consider } & P (g i) (g j) \lor P (f 0) (g i) \text{ by blast} \\
\text{then show } & ?\text{case using } (i < j) \text{ by } \text{(cases)} \text{ (auto intro: later)} \\
\text{qed blast}
\end{align*}

\begin{align*}
\text{then show } & \text{good } P f \text{ by } (\text{auto simp: good-def)} \\
\text{qed}
\end{align*}

**lemma** af-mono:
\begin{align*}
\text{assumes } & \text{af } A P \\
& \text{and } \forall x y. x \in A \land y \in A \land P x y \implies Q x y \\
\text{shows } & \text{af } A Q \\
\text{using } & \text{assms}
\end{align*}

**proof** (induct arbitrary: $Q$)

\begin{align*}
\text{case (now } P) \\
\text{then have } & (\forall x y. x \in A \implies y \in A \implies Q x y) \text{ by blast} \\
\text{then show } & ?\text{case by } \text{(rule af.now)} \\
\text{next}
\end{align*}

\begin{align*}
\text{case (later } P) \\
\text{show } & ?\text{case}
\end{align*}
proof \( (\text{intro af} \, \text{later} \, [\text{af} \ A \ Q]) \)
fix \( x \)
assume \( x \in A \)
then show \( \text{af} \ A \ (\lambda y \ z. \ Q \ y \ z \lor Q \ x \ y) \)
  \( \text{using} \ \text{later}(3) \) by \( (\text{intro later}(2) \, [\text{of} \ x]) \) auto
qed
qed

\text{lemma} \ \text{accessible-on-imp-af}:
\text{assumes} \ \text{accessible-on} \ P \ A \ x
\text{shows} \ \text{af} \ A \ (\lambda u \ v. \ \neg P \ u \ v \lor \neg P \ u \ x)
\text{using} \ \text{assms}
\text{proof} \ (\text{induct})
\text{case} \ (1 \ x)
then have \( \text{af} \ A \ (\lambda u \ v. \ (\neg P \ u \ v \lor \neg P \ u \ x) \lor \neg P \ u \ y \lor \neg P \ y \ x) \)
if \( y \in A \) for \( y \)
  \text{using} \ \text{that} \ (\text{by} \ (\text{cases} \ P \ y \ x) \ (\text{auto intro af})). \ \text{now} \ \text{af-mono})
then show \( ?\text{case} \) by \( (\text{rule af} \, \text{later}) \)
qed

\text{lemma} \ \text{wfp-on-imp-af}:
\text{assumes} \ \text{wfp-on} \ P \ A
\text{shows} \ \text{af} \ A \ (\lambda x \ y. \ \neg P \ y \ x)
\text{using} \ \text{assms} \ \text{by} \ (\text{auto simp: wfp-on-accessible-on-iff intro: accessible-on-imp-af af.next})

\text{definition} \ \text{NOTAF} \ A \ P = (\text{SOME} \ x. \ x \in A \land \neg \text{af} \ A \ (\lambda y \ z. \ P \ y \ z \lor \exists i \leq n. \ P (F \ A \ P \ i) \ y) \lor (\exists j \leq n. \ \exists i. \ i < j \land P (F \ A \ P \ i))

\text{fun} \ F
\text{where}
\ F \ A \ P \ 0 = \text{NOTAF} \ A \ P
| \ F \ A \ P \ (\text{Suc} \ i) = (\text{let} \ x = \text{NOTAF} \ A \ P \ in \ F \ A \ (\lambda y \ z. \ P \ y \ z \lor P \ x \ y) \ i)

\text{lemma} \ \text{almost-full-on-imp-af}:
\text{assumes} \ \text{af}: \ \text{almost-full-on} \ P \ A
\text{shows} \ \text{af} \ A \ P
\text{proof} \ (\text{rule ccontr})
\text{assume} \ \neg \text{af} \ A \ P
\text{then have} \ ?:\ F \ A \ P \ n \in A \land
\neg \text{af} \ A \ (\lambda y \ z. \ P \ y \ z \lor (\exists i \leq n. \ P (F \ A \ P \ i) \ y) \lor (\exists j \leq n. \ \exists i. \ i < j \land P (F \ A \ P \ i))\)

11
\( P \ i \ (F \ A \ P \ j) \) for \( n \)

proof (induct \( n \) arbitrary: \( P \))

  case 0
    from (\( \neg af \ A \ P \)) have \( \exists x. x \in A \wedge \neg af \ A \ (\lambda y. P y z \vee P x y) \) by (auto intro: af.intro)
    then have \( NOTAF \ A \ P \in A \wedge \neg af \ A \ (\lambda y. P y z \vee P (NOTAF \ A \ P) y) \)
      unfolding \( NOTAF-def \) by (rule someI-ex)
    with 0 show \( \Box \) case by simp
  next
    case (Suc \( n \))
    from (\( \neg af \ A \ P \)) have \( \exists x. x \in A \wedge \neg af \ A \ (\lambda y. P y z \vee P x y) \) by (auto intro: af.intro)
      then have \( NOTAF \ A \ P \in A \wedge \neg af \ A \ (\lambda y. P y z \vee P (NOTAF \ A \ P) y) \)
        unfolding \( NOTAF-def \) by (rule someI-ex)
    from Suc(1) [OF this [THEN conjunct2]]
      show \( \Box \) case
        by (fastforce simp: ex-leq-Suc ex-less-Suc elim!: back-subst [where \( P = \lambda x. \neg af \ A \ x \)])
    qed
  then have \( F \ A \ P \in SEQ \ A \) by auto
  from af [unfolded almost-full-on-def, THEN bspec, OF this] and not-af [OF * [THEN conjunct2]]
    show False unfolding good-def by blast
  qed

hide-const \( NOTAF \ F \)

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

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:
  \( A \subseteq B \Rightarrow \) almost-full-on \( P \ B \Rightarrow \) almost-full-on \( P \ A \)
by (auto simp: almost-full-on-def)

lemma almost-full-on-mono:
  assumes \( A \subseteq B \) and \( \forall x y. Q x y \Rightarrow P x y \)
    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:
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::nat. f i \in A
  then have \forall i. h (f i) \in B using \ h ' A \subseteq B; by auto
  with \ (almost-full-on Q B) [unfolded almost-full-on-def, THEN bspec, of h \circ f]
    show good ?P f
      unfolding good-def comp-def by blast
  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 \implies Q (h x) (h y)
    and af: almost-full-on P A
shows almost-full-on Q (h ' A)

proof
  fix f :: 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: \neg Q f
      { fix i j :: 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
  qed
The monomorphic preimage of an almost-full set is almost-full.

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

**Assumes** mon: \( \forall x. \forall y. [x \in A; y \in A] \implies P x y = Q (h x) (h y) \) biij-betw \( h \) \( A \) \( B \)

**And** af: almost-full-on \( Q \) \( B \)

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

**Proof**

Fix \( f :: \text{nat} \to 'a \)

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

Then have \( \forall i. (h \circ f) i \in B \) using mon by (auto simp: biij-betw-def)

Show good \( P f \)

Proof (rule ccontr)

Assume bad: bad \( P f \)

\[
\begin{align*}
\text{fix } i j :: \text{nat} \\
\text{assume } i < j \\
\text{from bad have } \neg P (f i) (f j) \text { using } (i < j) \text { by (auto simp: good-def)} \\
\text {with mon have } \neg Q (h (f i)) (h (f j)) \\
\text {using * by (auto simp: biij-betw-def inj-on-def) }
\end{align*}
\]

Then have bad \( Q (h \circ f) \) by (auto simp: good-def)

With af and ** show False by (auto simp: good-def almost-full-on-def)

Qed

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

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

**Assumes** total-on \( P \) \( A \) and wfp-on \( P \) \( A \)

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

**Proof** (rule ccontr)

Assume \( \neg \) almost-full-on \( P = A \)

Then obtain \( f :: \text{nat} \to 'a \) where \( \forall i. f i \in A \) and \( \forall i j. i < j \implies \neg P = (f i) (f j) \)

Unfolding almost-full-on-def by (auto dest: badE)

With \( _\text{total-on } P \) \( A \) have \( \forall i j. i < j \implies P = (f j) (f i) \)

Unfolding total-on-def by blast

Then have \( \forall i. P = (f (\text{Suc } i)) (f i) \) by auto

With \( \langle \text{wfp-on } P \rangle \) and * show False

Unfolding wfp-on-def by blast

Qed

**Lemma Nil-imp-good-list-emb [simp]:**

**Assumes** \( f i = [] \)

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

**Proof** (rule ccontr)

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

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

Unfolding assms by auto

Ultimately show False

Unfolding good-def by auto

Qed
lemma \( ne\text{-}lists \):
\[
\begin{align*}
&\text{assumes } xs \neq [] \text{ and } xs \in \text{lists } A \\
&\text{shows } \text{hd } xs \in A \text{ and } \text{tl } xs \in \text{lists } A \\
&\text{using } \text{assms } \text{by } (\text{case-tac } [[]] \hspace{1em} xs) \hspace{1em} \text{simp-all}
\end{align*}
\]

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

lemma \( \text{list-emb-eq-length-P} \):
\[
\begin{align*}
&\text{assumes } \text{length } xs = \text{length } ys \\
&\text{and } \text{list-emb } P \ xs \ ys \\
&\text{shows } \forall i<\text{length } xs. P (xs ! i) (ys ! i) \\
&\text{using } \text{assms}
\end{align*}
\]

\[
\text{proof } (\text{induct rule: } \text{list-emb-eq-length-induct})
\]
\[
\text{case } (\text{Cons } x \ y \ xs \ ys)
\]
\[
\text{show } ?\text{case}
\]
\[
\text{proof } (\text{intro allI impI})
\]
\[
\text{fix } i \ \text{assume } i<\text{length } (x \# xs)
\]
\[
\text{with } \text{Cons } \text{show } P ((x#xs)!i) ((y#ys)!i)
\]
\[
\text{by } (\text{cases } i) \ \text{simp-all}
\]
\[
\text{qed}
\]
\[
\text{qed } \ \text{simp}
\]

\[ \ \] 4.3 Special Case: Finite Sets

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

lemma \( \text{finite-almost-full-on} \):
\[
\begin{align*}
&\text{assumes } \text{finite: } \text{finite } A \\
&\text{and } \text{refl: } \text{reflp-on } P \ A \\
&\text{shows } \text{almost-full-on } P \ A
\end{align*}
\]

\[
\text{proof }
\]
\[
\text{fix } f :: \text{nat } \Rightarrow \ 'a
\]
\[
\text{assume } \ast; \forall i. \ f \ i \in A
\]
\[
\text{let } ?I = \text{UNIV::nat set}
\]
\[
\text{have } f \ ^\ast \ ?I \subseteq A \ \text{using } \ast \ \text{by } \text{auto}
\]
\[
\text{with } \text{finite } \text{and } \text{finite-subset } \text{have } \text{finite } (f \ ^\ast \ ?I) \ \text{by } \text{blast}
\]
\[
\text{have } \text{infinite } ?I \ \text{by } \text{auto}
\]
\[
\text{from } \text{pigeonhole-infinitive } \text{[OF this } 1]
\]
\[
\text{obtain } k \ \text{where } \text{infinite } \{j. \ f \ j = f \ k\} \ \text{by } \text{auto}
\]
\[
\text{then obtain } l \ \text{where } k < l \ \text{and } f \ l = f \ k
\]
\[
\text{unfolding } \text{infinite-nat-iff-unbounded } \text{by } \text{auto}
\]
\[
\text{then have } P (f \ k) (f \ l) \ \text{using } \text{refl } \text{and } \ast \ \text{by } (\text{auto simp: } \text{reflp-on-def})
\]
\[
\text{with } k<l \ \text{show } \text{good } P \ f \ \text{by } (\text{auto simp: } \text{good-def})
\]
\[
\text{qed}
\]

16
lemma eq-almost-full-on-finite-set:
  assumes finite A
  shows almost-full-on (=) A
  using finite-almost-full-on [OF assms, of (=)]
  by (auto simp: reflp-on-def)

4.4 Further Results

lemma af-trans-extension-imp-wf:
  assumes subrel: \( \forall x y. P x y \Rightarrow Q x y \)
  and af: almost-full-on P A
  and trans: transp-on Q A
  shows wfp-on (strict Q) A
  proof (unfold wfp-on-def, rule notI)
    assume \( \exists f, \forall i. f i \in A \land \text{strict } Q (f (\text{Suc } i)) (f i) \)
    then obtain f where *: \( \forall i. f i \in A \land ((\text{strict } Q)^{-1}) (f i) (f (\text{Suc } i)) \) by blast
    from chain-transp-on-less [OF this]
    and transp-on-strict [THEN transp-on-converse, OF trans]
    have \( \forall i j. i < j \rightarrow \neg Q (f i) (f j) \) by blast
    with subrel have \( \forall i j. i < j \rightarrow \neg P (f i) (f j) \) by blast
    with af show False
    using * by (auto simp: almost-full-on-def good-def)
  qed

lemma af-trans-imp-wf:
  assumes almost-full-on P A
  and transp-on P A
  shows wfp-on (strict P) A
  using assms by (intro af-trans-extension-imp-wf)

lemma wf-and-no-antichain-imp-qo-extension-wf:
  assumes af: wfp-on (strict P) A
  and anti: \( \neg (\exists f. \text{antichain-on } P f A) \)
  and subrel: \( \forall x \in A. \forall y \in A. P x y \Rightarrow Q x y \)
  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}) A by (rule transp-on-converse)
    assume \( \neg \text{wfp-on } (\text{strict } Q) A \)
    then obtain f :: nat \Rightarrow 'a where A: \( \forall i. f i \in A \)
      and \( \forall i. \text{strict } Q (f (\text{Suc } i)) (f i) \) unfolding wfp-on-def by blast+
    then have \( \forall i f i i \in A \land ((\text{strict } Q)^{-1}) (f i) (f (\text{Suc } i)) \) by auto
    from chain-transp-on-less [OF this *]
    have \( \forall i j. i < j \Rightarrow \neg P (f i) (f j) \)
      using subrel and A by blast
    show False
  qed
proof (cases)
  assume \( \exists k. \forall i > k. \exists j > i. P (f j) (f i) \)
  then obtain \( k \) where \( \forall i > k. \exists j > i. P (f j) (f i) \) by auto
from subchain [of \( k \) - \( f \)], OF this] obtain \( g \)
  where \( \\forall i. j. i < j \implies g i < g j \)
  and \( \\forall i. P (f (g (Suc i))) (f (g i)) \) by auto
with * have \( \\forall i. \text{strict} P (f (g (Suc i))) (f (g i)) \) by blast
with wf [unfolded wfp-on-def not-ex, THEN spec, of \( \lambda i. f (g i) \)] and \( A \)
  show \( \text{False} \) by fast

next
  assume \( \neg (\exists k. \forall i > k. \exists j > i. P (f j) (f i)) \)
  then have \( \forall k. \exists i > k. \exists j > i. \neg P (f j) (f i) \) by auto
from choice [OF this] obtain \( h \)
  where \( \forall k. h k > k \)
  and **: \( \forall k. (\forall j > h k. \neg P (f j) (f (h k))) \) by auto
define \( \varphi \) where [simp]: \( \varphi = (\lambda i. (h \cdot Suc i) 0) \)
  have \( \\forall i. \varphi i < \varphi (Suc i) \)
  using \( \forall k. h k > k \) by (induct-tac \( i \)) auto
  define \( \varphi \) where [simp]: \( \varphi = (\lambda i. (h \cdot Suc i) 0) \)
  have \( \forall i. \varphi i < \varphi (Suc i) \)
  using ** by auto
with mono [THEN *]
  have \( \forall i. j. i < j \implies \text{incomparable} P (f (\varphi j)) (f (\varphi i)) \) by blast
moreover have \( \exists i. j. i < j \land \neg \text{incomparable} P (f (\varphi j)) (f (\varphi i)) \)
  using anti [unfolded not-ex, THEN spec, of \( \lambda i. f (\varphi i) \)] and \( A \) by blast
  ultimately show \( \text{False} \) by blast
qed

lemma every-qo-extension-wf-imp-af:
  assumes ext: \( \forall Q. (\forall x \in A. \forall y \in A. P x y \implies Q x y) \land \)
  qo-on \( Q A \implies \text{wpf-on} (\text{strict} Q) A \)
  and qo-on \( P A \)
  shows almost-full-on \( P A \)
proof
from \( \text{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 \( f :: \text{nat} \Rightarrow 'a \)
assume \( \forall i. f i \in A \)
then have \( A: \\\forall i. f i \in A .. \)
  show \( \text{good} P f \)
proof (rule ccontr)
  assume \( \neg \text{thesis} \)
  then have \( \text{bad: } \forall i. j. i < j \implies \neg P (f i) (f j) \) by (auto simp: good-def)
  then have *: \( \\forall i. j. P (f i) (f j) \implies i \geq j \) by (metis not-le-imp-less)
qed

qed


define $D$ where [simp]: $D = (\lambda x. y. \exists i. x = f (\text{Suc } i) \land y = f i)$

define $P'$ where $P' = \text{restrict-to } P A$

define $Q$ where [simp]: $Q = (\sup P')^*$

have **: $\forall i j. (D \circ\circ P')^+ (f i) (f j) \Rightarrow i > j$
proof –
  fix $i j$
  assume $(D \circ\circ P')^+ (f i) (f j)$
  then show $i > j$
    apply (induct $f i f j$ arbitrary: $j$)
    apply (insert $A$, auto dest: *)
  simp: $P'$-def reflp-on-restrict-to-rtranclp [OF refl trans])
  apply (metis * dual-order.strict-trans1 less-Suc-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 go-on $Q A$ by (auto simp: go-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 $\forall i. f i \in A \land \text{strict } Q (f (\text{Suc } i)) (f i)$
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 $(\sup P')^* = \sup (P'^*) (P'^* \circ\circ (D \circ\circ P')^+)$
      proof –
        have $\forall A B. (A \cup B)^* = A^* \cup A^* O (B \circ\circ A^*)^+$ by regexp
        from this [to-pred] show ?thesis by blast
      qed
    ultimately have $\sup (P'^*) (P'^* \circ\circ (D \circ\circ P')^+) (f i) (f (\text{Suc } i))$
      by simp
    then have $(P'^* \circ\circ (D \circ\circ P')^+) (f i) (f (\text{Suc } i))$ by auto
    then have $\text{Suc } i < i$
      using ** apply auto
      by (metis lifting mono-tags less-le relcompp relcompl tranclp-into-tranclp2)
    then show False by auto
    qed
  with $A$ [of $i$] show $f i \in A \land \text{strict } Q (f (\text{Suc } i)) (f i)$ by auto
  qed
ultimately show False unfolding wfp-on-def by blast
qed

end
5 Constructing Minimal Bad Sequences

theory Minimal-Bad-Sequences
imports
  Almost-Full
  Minimal-Elements
begin

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 $A :: ('a :: size) set$
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 $\exists y \in A. \text{size } y \leq \text{size } x \land (\forall z \in A. \text{size } z < \text{size } y \rightarrow \neg P z)$
using assms
proof (induction $x$ taking: size rule: measure-induct)
  case (1 $x$)
  then show $?case$
  proof (cases $\forall y \in A. \text{size } y < \text{size } x \rightarrow \neg P y$)
    case True
    with 1 show $?thesis$ by blast
  next
    case False
    then obtain $y$ where $y \in A$ and $\text{size } y < \text{size } x$ and $P y$ by blast
    with 1.IH show $?thesis$ by (fastforce elim!: order-trans)
  qed
qed

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

The set of all bad sequences over $A$.

definition $\text{BAD } P = \{ f \in \text{SEQ } A. \text{bad } P f \}$

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

A partial order on infinite bad sequences.

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

lemma gseq-iff:
\((f, g) \in gseq \iff f \in SEQ A \land g \in SEQ A \land (f = g \lor (\exists i. size (g i) < size (f i) \land (\forall j < i. f j = g j)))\)

by (auto simp: gseq-def)

lemma gseq-iff:
\((f, g) \in gseq \iff f \in SEQ A \land g \in SEQ A \land (\exists i. size (g i) < size (f i) \land (\forall j < i. f j = g j)))\)

by (auto simp: gseq-def)

lemma gseqE:
assumes \((f, g) \in gseq \land \land \forall i. f i \in A; \forall i. g i \in A; f = g \implies Q\)
and \(\forall i. \forall i. f i \in A; \forall i. g i \in A; size (g i) < size (f i); \forall j < i. f j = g j\)

shows \(Q\)
using assms by (auto simp: gseq-iff)

lemma gseqE:
assumes \((f, g) \in gseq \land \land \forall i. f i \in A; \forall i. g i \in A; size (g i) < size (f i); \forall j < i. f j = g j\)

shows \(Q\)
using assms by (auto simp: gseq-iff)

sublocale min-elt-size?: minimal-element measure-on size UNIV A
rewrites measure-on size UNIV \equiv \lambda x y. size x < size y
apply (unfold-locales)
apply (auto simp: po-on-def irreflp-on-def transp-on-def simp del: wfp-on-UNIV)
apply (auto simp: measure-on-def inv-image-betw-def)
done

correctness
fixes P :: 'a ⇒ 'a ⇒ bool
begin

A lower bound to all sequences in a set of sequences \(B\).

abbreviation lb \equiv lexmin (BAD P)
lemma eq upto BAD mem:
  assumes f ∈ eq upto (BAD P) g i
  shows f j ∈ A
  using assms by (auto)

Assume that there is some infinite bad sequence h.

close context

fixes h :: nat ⇒ 'a

assumes BAD-ex: h ∈ BAD P

begin

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

lemma eq upto BAD non empty:
  eq upto (BAD P) lb i ≠ {}
  using eq upto lexmin non empty [of BAD P] and BAD-ex by auto

lemma non empty ith:
  shows ith (eq upto (BAD P) lb i) i ⊆ A
  and ith (eq upto (BAD P) lb i) i ≠ {}
  using eq upto BAD non empty [of i] by auto

lemmas
  lb minimal = min elt minimal [OF non empty ith, folded lexmin] and
  lb mem = min elt mem [OF non empty ith, folded lexmin]

lb is an infinite bad sequence.

lemma lb BAD:
  lb ∈ BAD P

proof (intro allI impI)
  have *: ∃ j. lb j ∈ ith (eq upto (BAD P) lb j) j by (rule lb mem)
  then have ∀ i. lb i ∈ A by (auto simp: ith conv) (metis eq upto BAD mem)
  moreover
  { assume good P lb
    then obtain i j where i < j and P (lb i) (lb j) by (auto simp: good def)
    from * have lb j ∈ ith (eq upto (BAD P) lb j) j by (auto)
    then obtain g where g ∈ eq upto (BAD P) lb j and g j = lb j by force
    then have ∀ k ≤ j. g k = lb k by (auto simp: order le less)
    with i < j and P (lb i) (lb j) have P (g i) (g j) by auto
    with i < j have good P g by (auto simp: good def)
    with g ∈ eq upto (BAD P) lb j have False by auto }
  ultimately show thesis by blast
  qed

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

lemma lb lower bound:
  ∀ g. (lb, g) ∈ gseq → g /∈ BAD P

proof (intro allI impI)
fix g
assume (lb, g) ∈ gseq
then obtain i where g i ∈ A and size (g i) < size (lb i)
and ∀ j < i. lb j = g j by (auto simp: gseq-iff)
moreover with lb-minimal
have g i ≠ ith (eq-uplo (BAD P) lb i) i by auto
ultimately show g i ∉ BAD P by blast
qed

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

lemma lower-bound-ex:
∃ f ∈ BAD P. ∀ g. (f, g) ∈ gseq −→ g ∉ BAD P
using lb-BAD and lb-lower-bound by blast

lemma gseq-conv:
(f, g) ∈ gseq −→ f ≠ g ∧ (f, g) ∈ geseq
by (auto simp: gseq-def geseq-def dest: less-not-eq)

There is a minimal bad sequence.

lemma mbs:
∃ f ∈ BAD P. ∀ g. (f, g) ∈ gseq −→ good P g
using lower-bound-ex by (auto simp: gseq-conv geseq-iff)

end

6 A Proof of Higman’s Lemma via Open Induction

theory Higman-OI
imports
  Open-Induction.Open-Induction
  Minimal-Elements
  Almost-Full
begin

6.1 Some facts about the suffix relation

lemma wfp-on-strict-suffix:
wfp-on strict-suffix A
by (rule wfp-on-mono [OF subset-refl, of - - measure-on length A])
  (auto simp: strict-suffix-def suffix-def)

lemma po-on-strict-suffix:
by (force simp: strict-suffix-def po-on-def transp-on-def irreflp-on-def)

6.2 Lexicographic Order on Infinite Sequences

lemma antisymp-on-LEX:
  assumes irreflp-on P A and antisymp-on P A
  shows antisymp-on (LEX P) (SEQ A)
proof
  fix f g assume SEQ: \( f \in SEQ A \) \( g \in SEQ A \) and LEX P f g and LEX P g f
  then obtain i j where \( P (f i) \) \( (g i) \) and \( P (g j) \) \( (f j) \)
  and \( \forall k<i. f k = g k \) and \( \forall k<j. g k = f k \) by (auto simp: LEX-def)
  then have LEX: \( P (f (\min i j)) \) \( (g (\min i j)) \)
    using assms(2) and SEQ by (cases i = j) (auto simp: antisymp-on-def min-def, force)
  with assms(1) and SEQ show \( f = g \) by (auto simp: irreflp-on-def)
qed

lemma LEX-trans:
  assumes transp-on P A and f \in SEQ A and g \in SEQ A and h \in SEQ A
  and LEX P f g and LEX P g h
  shows LEX P f h
using assms by (auto simp: LEX-def transp-on-def) (metis less-trans linorder-neqE-nat)

lemma qo-on-LEXEQ:
  transp-on P A \( \Rightarrow \) qo-on (LEXEQ P) (SEQ A)
by (auto simp: qo-on-def reflp-on-def transp-on-def [of LEXEQ P] dest: LEX-trans)

context minimal-element
begin

lemma glb-LEX-lexmin:
  assumes chain-on (LEX P) C (SEQ A) and C \( \neq \{\} \)
  shows glb (LEX P) C (lexmin C)
proof
  have C \( \subseteq SEQ A \) using assms by (auto simp: chain-on-def)
  then have lexmin C \( \in SEQ A \) using \( C \neq \{\} \) by (intro lexmin-SEQ-mem)
  note * = \( C \subseteq SEQ A \) \( (C \neq \{\}) \)
  note lex = LEX-imp-less [folded irreflp-on-def, OF po [THEN po-on-imp-irreflp-on]]
  — lexmin C is a lower bound
  show lb (LEX P) C (lexmin C)
proof
  fix f assume f \in C
  then show LEXEQ P (lexmin C) f
proof (cases f = lexmin C)
    define i where i = (LEAST i. f i \neq lexmin C i)
    case False
    then have neq: \( \exists i. f i \neq lexmin C i \) by blast
    from LeastI-ex [OF this, folded i-def]
and not-less-Least [where $P = \lambda i. f i \neq \text{lexmin } C i$, folded i-def]

have neg: $f i \neq \text{lexmin } C i$ and eq: $\forall j < i. f j = \text{lexmin } C j$ by auto
then have **: $f \in \text{eq-upto } C (\text{lexmin } C) i i$ $i i \in \text{ith } (\text{eq-upto } C (\text{lexmin } C))$

i i

using $(f \in C)$ by force+
moreover from ** have $\neg P \ (f i) (\text{lexmin } C i)$
using lexmin-minimal [OF *, of $f i i$] and $(f \in C)$ and $(C \subseteq \text{SEQ } A)$ by blast

moreover obtain $g$ where $g \in \text{eq-upto } C (\text{lexmin } C) \ (\text{Suc } i)$
ultimately have $P \ (\text{lexmin } C i) (f i)$
using eq and $(C \subseteq \text{SEQ } A)$ and assms(1) and lex [of $g f i$] and lex [of $f$ $g$ $i$]

by (auto simp: eq-upto-def chain-on-def)
with eq show ?thesis by (auto simp: LEX-def)

qed simp

qed

— lexmin $C$ is greater than or equal to any other lower bound

fix $f$ assume lb: $lb \ (\text{LEX } P) \ C f$
then show LEXEQ $P f \ (\text{lexmin } C)$
proof (cases $f = \text{lexmin } C$)
define $i$ where $i = (\text{LEAST } i. f i \neq \text{lexmin } C i)$
case False
then have neq: $\exists i. f i \neq \text{lexmin } C i$ by blast
from LeastI-ex [OF this, folded i-def]
and not-less-Least [where $P = \lambda i. f i \neq \text{lexmin } C i$, folded i-def]
have neg: $f i \neq \text{lexmin } C i$ and eq: $\forall j < i. f j = \text{lexmin } C j$ by auto
obtain $h$ where $h \in \text{eq-upto } C (\text{lexmin } C) \ (\text{Suc } i)$ and $h \in C$
using eq-upto-lexmin-non-empty [OF *] by (auto simp: eq-upto-def)
then have [simp]: $\forall j. j < \text{Suc } i \Longrightarrow h j = \text{lexmin } C j$ by auto
with lb and $h \in C$: have LEX $P f h$ using neq by (auto simp: lb-def)
then have $P \ (f i) \ (h i)$
using eq and eq and $(C \subseteq \text{SEQ } A)$ and $(h \in C)$ by (intro lex) auto
with eq show ?thesis by (auto simp: LEX-def)

qed simp

qed

lemma dc-on-LEXEQ:
dc-on (LEXEQ $P) \ (\text{SEQ } A)$
proof
fix $C$ assume chain-on (LEXEQ $P) \ C \ (\text{SEQ } A)$ and $C \neq \{}$
then have chain: chain-on (LEX $P) \ C \ (\text{SEQ } A)$ by (auto simp: chain-on-def)
then have $C \subseteq \text{SEQ } A$ by (auto simp: chain-on-def)
then have lexmin $C \in \text{SEQ } A$ using $(C \neq \{})$ by (intro lexmin-SEQ-mem)
have $\text{gb } (\text{LEX } P) \ C \ (\text{lexmin } C)$ by (rule $\text{gb } (\text{LEX } \text{LEXEQ } P) \ (\text{lexmin } C) \ by \ (\text{auto simp: gb-def lb-def})$
with $(\text{lexmin } C \in \text{SEQ } A)$ show $\exists f \in \text{SEQ } A. \text{gb } (\text{LEXEQ } P) \ C f$ by blast

qed
Properties that only depend on finite initial segments of a sequence (i.e.,
which are open with respect to the product topology).

**Definition** \( pt\text{-open-on} \ Q \ A \mapsto (\forall f \in A. \ Q f \iff (\exists n. (\forall g \in A. \ (\forall i < n. \ g_i = f_i) \rightarrow Q g))) \)

**Lemma** \( pt\text{-open-on}\text{-D}: \)
\( pt\text{-open-on} \ Q \ A \Rightarrow Q f \Rightarrow f \in A \Rightarrow (\exists n. (\forall g \in A. \ (\forall i < n. \ g_i = f_i) \rightarrow Q g)) \)
unfolding \( pt\text{-open-on-def} \) by blast

**Lemma** \( pt\text{-open-on-good}: \)
\( pt\text{-open-on} (\text{good } Q) (\text{SEQ } A) \)
**Proof** (unfold \( pt\text{-open-on-def} \), intro ballI)
fix \( f \) assume \( f \in \text{SEQ } A \)
show \( \text{good } Q f = (\exists n. \forall g \in \text{SEQ } A. \ (\forall i < n. \ g_i = f_i) \rightarrow \text{good } Q g) \)
**Proof**
assume \( \text{good } Q f \)
then obtain \( i \) and \( j \) where \(*: i < j \ Q (f_i) \ (f_j) \) by auto
have \( \forall g \in \text{SEQ } A. \ (\forall i < \text{Suc } j. \ g_i = f_i) \rightarrow \text{good } Q g \)
**Proof** (intro ballIImpl)
fix \( g \) assume \( g \in \text{SEQ } A \) and \( \forall i < \text{Suc } j. \ g_i = f_i \)
then show \( \text{good } Q g \) using \( * \) by (force simp: good-def)
qed
then show \( \exists n. \forall g \in \text{SEQ } A. \ (\forall i < n. \ g_i = f_i) \rightarrow \text{good } Q g \) ..
next
assume \( \exists n. \forall g \in \text{SEQ } A. \ (\forall i < n. \ g_i = f_i) \rightarrow \text{good } Q g \)
with \( f \) show \( \text{good } Q f \) by blast
qed
qed

**Context** minimal-element
**Begin**

**Lemma** \( pt\text{-open-on-imp-open-on-LEXEQ}: \)
assumes \( pt\text{-open-on} \ Q (\text{SEQ } A) \)
shows \( \text{open-on} (\text{LEXEQ } P) \ Q (\text{SEQ } A) \)
**Proof**
fix \( C \) assume \( \text{chain}: \text{chain-on} (\text{LEXEQ } P) \ C (\text{SEQ } A) \) and \( \text{ne}: C \neq \{\} \)
and \( \exists g \in \text{SEQ } A. \ \text{glb} (\text{LEXEQ } P) \ C g \land Q g \)
then obtain \( g \) where \( g: g \in \text{SEQ } A \) and \( \text{glb} (\text{LEXEQ } P) \ C g \)
and \( Q: Q g \) by blast
then have \( \text{glb}: \text{glb} (\text{LEX } P) \ C g \) by (auto simp: glb-def lb-def)
from \( \text{chain} \) have \( \text{chain-on} (\text{LEX } P) \ C (\text{SEQ } A) \) and \( C: C \subseteq \text{SEQ } A \) by (auto simp: chain-on-def)
**Note** \( * = \text{glb-LEX-lexmin} \) [OF this(1) \( \text{ne} \)]
**Have** \( \text{lexmin} \ C \in \text{SEQ } A \) using \( \text{ne} \) and \( C \) by (intro lexmin-SEQ-mem)
from glb-unique [OF - g this glb *]
    and antisym-on-LEX [OF po-on-imp-irreflp-on [OF po] po-on-antisym-on [OF po]]
    have [simp]: lexmin C = g by auto
from assms [THEN pt-open-onD, OF Q g] 
obtain n :: nat where **: \h. \ h \in SEQ A \implies (\forall i<n. h i = g i) \implies Q h by blast
from eq-upto-lexmin-non-empty [OF C ne, of n] 
obtain f where f \in eq-upto C g n by auto
then have f \in C and Q f using ** [of f] and C by force+
then show \exists f \in C. Q f by blast
qed

lemma open-on-good:
  open-on (LEXEQ P) (good Q) (SEQ A)
  by (intro pt-open-on-imp-open-on-LEXEQ pt-open-on-good)
end

lemma open-on-LEXEQ-imp-pt-open-on-counterexample:
  fixes a b :: 'a
  defines A \equiv \{a, b\} and P \equiv (\lambda x y. False) and Q \equiv (\lambda f. \forall i. f i = b)
  assumes [simp]: a \neq b
  shows minimal-element P A and open-on (LEXEQ P) Q (SEQ A)
    and \neg pt-open-on Q (SEQ A)
proof -
  show minimal-element P A 
    by standard (auto simp: P-def po-on-def irrefl-on-def transp-on-def wfp-on-def)
  show open-on (LEXEQ P) Q (SEQ A)
    by (auto simp: P-def open-on-def chain-on-def SEQ-def glb-def lb-def LEX-def)
  show \neg pt-open-on Q (SEQ A)
proof 
  define f :: nat \Rightarrow 'a where f \equiv (\lambda x. b)
  have f \in SEQ A by (auto simp: A-def f-def)
  moreover assume pt-open-on Q (SEQ A)
  ultimately have Q f \iff (\exists n. (\forall g \in SEQ A. (\forall i<n. g i = f i) \implies Q g))
    unfolding pt-open-on-def by blast
  moreover have Q f by (auto simp: Q-def f-def)
  moreover have \exists g \in SEQ A. (\forall i<n. g i = f i) \land \neg Q g for n
    by (intro bexI [of - f(n := a)]) (auto simp: f-def Q-def A-def)
  ultimately show False by blast
qed

qed

lemma higman:
  assumes almost-full-on P A
  shows almost-full-on (list-emb P) (lists A)
proof
  interpret minimal-element strict-suffix lists A
by (unfold-locales) (intro po-on-strict-suffix wfp-on-strict-suffix)+
fix f presume f ∈ SEQ (lists A)
with po-on-LEXEQ [OF po-on-imp-transp-on [OF po-on-strict-suffix]] and dc-on-LEXEQ
and open-on-good
  show good (list-emb P) f
proof (induct rule: open-induct-on)
case (less f)
define h where h i = hd (f i) for i
show ?case
proof (cases \exists i. f i = [])
case False
  then have ne: \forall i. f i ≠ [] by auto
  with (f ∈ SEQ (lists A)) have \forall i. h i ∈ A by (auto simp: h-def ne-lists)
  from almost-full-on-imp-homogeneous-subseq [OF assms this]
  obtain \varphi :: nat ⇒ nat where mono: \forall i j. i < j ⇒ \varphi i < \varphi j
    and P: \forall i j. i < j ⇒ P (h (\varphi i)) (h (\varphi j)) by blast
  define f' where f' i = (if i < \varphi 0 then f i else tl (f (\varphi (i - \varphi 0)))) for i
  have f': f' ∈ SEQ (lists A) using ne and (f ∈ SEQ (lists A))
    by (auto simp: f'-def dest: list-set-select)
  moreover have strict-suffix (f' (\varphi 0)) (f (\varphi 0)) using ne by (auto simp: f'-def)
  ultimately have LEX strict-suffix f' f by (auto simp: LEX-def)
    with LEX-imp-not-LEX [OF this] have strict (LEXEQ strict-suffix) f' f
      using po-on-strict-suffix [of UNIV] unfolding po-on-def irreflp-on-def
      transp-on-def by blast
  from less(2) [OF f' this] have good (list-emb P) f',
  then obtain i j where i < j and emb: list-emb P (f' i) (f' j) by (auto simp: good-def)
  consider j < \varphi 0 | \varphi 0 ≤ i | i < \varphi 0 and \varphi 0 ≤ j by arith
  then show ?thesis
  proof (cases)
    case 1 with (i < j) and emb show ?thesis by (auto simp: good-def)
  next
    case 2
    with (i < j) and P have P (h (\varphi (i - \varphi 0))) (h (\varphi (j - \varphi 0))) by auto
    with emb have list-emb P (h (\varphi (i - \varphi 0))) (h (\varphi (j - \varphi 0))) # f' i (h (\varphi (j - \varphi 0))) # f' j by auto
    then have list-emb P (f (\varphi (i - \varphi 0))) (f (\varphi (j - \varphi 0))) using 2 and (i < j) by auto
    moreover with 2 and i < j; have \varphi (i - \varphi 0) < \varphi (j - \varphi 0) using mono by auto
    ultimately show ?thesis by (auto simp: good-def)
  next
    case 3
    with emb have list-emb P (f i) (f' j) by auto
    moreover have f (\varphi (j - \varphi 0)) = h (\varphi (j - \varphi 0)) # f' j using 3 by auto
    ultimately have list-emb P (f i) (f (\varphi (j - \varphi 0))) by auto
  qed auto
ultimately show ?thesis by (auto simp: good-def)
next
case 3
with emb have list-emb P (f i) (f' j) by auto
moreover have f (\varphi (j - \varphi 0)) = h (\varphi (j - \varphi 0)) # f' j using 3 by auto
ultimately have list-emb P (f i) (f (\varphi (j - \varphi 0))) by auto
ultimately show ?thesis by (auto simp: good-def)
next

moreover have \( i < \varphi (j - \varphi 0) \) using mono [of \( j - \varphi 0 \)] and 3 by force ultimately show thesis by (auto simp: good-def)

qed
qed auto
qed
qed blast

7 Almost-Full Relations

theory Almost-Full-Relations
imports Minimal-Bad-Sequences
begin

lemma (in mbs) mbs':
  assumes ¬almost-full-on P A
  shows \( \exists m \in \text{BAD } P \). \( \forall \quad g \cdot \quad (m, g) \in \text{gseq} \longrightarrow \text{good } P \ g \)
  using assms and mbs unfolding almost-full-on-def by blast

7.1 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]:
  Some x \( \in \ A_{\bot} \quad \iff \quad x \in \text{A} \)
  by (auto simp: with-bot-def)

lemma NoneI [simp, intro]:
  None \( \in \ A_{\bot} \)
  by (simp add: with-bot-def)

lemma not-None-the-mem [simp]:
  \( x \neq \text{None} \quad \Longrightarrow \quad \text{the } x \in \text{A} \quad \iff \quad x \in \ A_{\bot} \)
  by auto

lemma with-bot-cases:
  \( u \in \ A_{\bot} \quad \Longrightarrow \quad (\forall x. x \in \text{A} \quad \Longrightarrow \quad u = \text{Some } x \quad \Longrightarrow \quad P) \quad \Longrightarrow \quad (u = \text{None} \quad \Longrightarrow \quad P) \quad \Longrightarrow \quad P \)
  by auto

lemma with-bot-empty-conv [iff]:
  \( A_{\bot} = \{ \text{None} \} \quad \iff \quad A = \{ \} \)
  by (auto elim: with-bot-cases)

lemma with-bot-UNIV [simp]:
  \( \text{UNIV}_{\bot} = \text{UNIV} \)

29
proof (rule set-eqI)
fix x :: 'a option
show x ∈ UNIV ⊥ ←→ x ∈ UNIV by (cases x) auto
qed

7.2 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 *: ∀. f i ∈ ?A
  show good ?P f by (cases *: ∀ i. f i ≠ None)
    case True
    then have **: ∀. Some (the (f i)) = f i
    and ∀. 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 (Some (the (f i))) (Some (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
qed

7.3 Disjoint Union of Almost-Full Sets

fun
  sum-le :: ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ 'a + 'b ⇒ 'a + 'b ⇒ bool
where
  sum-le P Q (Inl x) (Inl y) = P x y |
  sum-le P Q (Inr x) (Inr y) = Q x y |
  sum-le P Q x y = False

lemma not-sum-le-cases:
  assumes ¬ sum-le P Q a b
  and ∃ x y. [a = Inl x; b = Inl y; ¬ P x y] ⇒ thesis
and \( \forall x, y. [a = Inr x; b = Inr y; \neg Q x y] \rightarrow \text{thesis} \)
and \( \forall x, y. [a = Inl x; b = Inr y] \rightarrow \text{thesis} \)
and \( \forall x, y. [a = Inr x; b = Inl y] \rightarrow \text{thesis} \)
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 (sum-le P Q) (A <+> B) (is almost-full-on ?P ?A)
proof
fix f :: nat \( \Rightarrow (\text{'}a + \text{'}b) \)
let \( \tilde{?I} = f - \text{'}Inl \text{'}A \)
let \( \tilde{?J} = f - \text{'}Inr \text{'}B \)
assume \( \forall i. f i \in \tilde{?A} \)
then have \( \ast: \tilde{?J} = (\text{UNIV::nat set}) - \tilde{?I} \) by (fastforce)
show good ?P f
proof (rule ccontr)
assume bad: bad ?P f
show False
proof (cases finite \( \tilde{?I} \))
assume finite \( \tilde{?I} \)
then have infinite \( \tilde{?J} \) by (auto simp: \( \ast \))
then interpret infinitely-many1 \( \lambda i. f i \in \text{Inr} \text{'}B \)
by (unfold-locales) (simp add: infinite-nat-iff-unbounded)
have \( \text{dest}: \forall i x. f (\text{enum i}) = \text{Inl x} \Rightarrow \text{False} \)
using enum-P by (auto simp: image-iff) (metis Inr-Inl-False)
let \( \tilde{?f} = \lambda i. \text{projr} (f (\text{enum i})) \)
have \( B: \forall i. \tilde{?f} i \in B \) using enum-P by (auto simp: image-iff) (metis sum.sel (2))
\{ fix i j :: nat
assume \( i < j \)
then have enum i < enum j using enum-less by auto
with bad have \( \neg ?P \ (f \ (\text{enum i})) \ (f \ (\text{enum j})) \) by (auto simp: good-def)
then have \( \neg Q \ (\tilde{?f} i) \ (\tilde{?f} j) \) by (auto elim: not-sum-le-cases) \}
then have bad Q \( \tilde{?f} \) by (auto simp: good-def)
moreover from almost-full-on Q B; and B
have good Q \( \tilde{?f} \) by (auto simp: good-def almost-full-on-def)
ultimately show False by blast
next
assume infinite \( \tilde{?I} \)
then interpret infinitely-many1 \( \lambda i. f i \in \text{Inl} \text{'}A \)
by (unfold-locales) (simp add: infinite-nat-iff-unbounded)
have \( \text{dest}: \forall i x. f (\text{enum i}) = \text{Inr x} \Rightarrow \text{False} \)
using enum-P by (auto simp: image-iff) (metis Inr-Inl-False)
let \( \tilde{?f} = \lambda i. \text{projl} (f (\text{enum i})) \)
have \( A: \forall i. \tilde{?f} i \in A \) using enum-P by (auto simp: image-iff) (metis sum.sel (1))
\{ fix i j :: nat

assume \( i < j \)
then have \( \text{enum } i < \text{enum } j \) using enum-less by auto
with bad have \( \neg P \ (f \ (\text{enum } i)) \ (f \ (\text{enum } j)) \) by (auto simp: good-def)
then have \( \neg P \ (if \ i) \ (if \ j) \) by (auto elim: not-sum-le-cases)
then have bad \( P \ if \) by (auto simp: good-def)
moreover from \( \text{almost-full-on } P \ A \) and \( A \)
have good \( P \ if \) by (auto simp: good-def almost-full-on-def)
ultimately show False by blast
qed

\section{Dickson’s Lemma for Almost-Full Relations}

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

**definition**

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

where

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

**lemma** \( \text{prod-le-True} \ [\text{simp}]: \)

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

by (auto simp: prod-le-def)

**lemma** \( \text{almost-full-on-Sigma}: \)

assumes \( \text{almost-full-on } P1 \ A1 \) 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 bad: bad \( ?P \ f \) by (auto simp: almost-full-on-def)

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

let \( ?B = \lambda x \ y. \ P2 \ (\text{snd} \ x) \ (\text{snd} \ y) \)

from \( f \) have \( \\text{fst} : \forall i. \ \text{fst} \ (f \ i) \in A1 \) and \( \text{snd} : \forall i. \ \text{snd} \ (f \ i) \in A2 \)

by (metis SigmaE fst-conv, metis SigmaE 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 \( \text{mono}: \forall i \ j. \ i \ < \ j \ \Rightarrow \varphi \ i \ < \varphi \ j \)

and \( *: \forall i \ j. \ i \ < \ j \ \Rightarrow \ ?W \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by auto

from \( \text{snd} \) have \( \forall i. \ \text{snd} \ (f \ (\varphi \ i)) \in A2 \) by auto

then have \( \text{snd} \circ f \circ \varphi \in \text{SEQ} \ A2 \) by auto

with \( \text{assms}(2) \) have \( \text{good} \ P2 \ (\text{snd} \circ f \circ \varphi) \) by (auto simp: almost-full-on-def)

then obtain \( i \ j :: \text{nat} \)

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

with \(*: \text{OF } i \ < \ j \) have \( \text{?P} \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by (simp add: case-prod-beta prod-le-def)

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

qed
7.5 Higman’s Lemma for Almost-Full Relations

lemma almost-full-on-lists:
  assumes almost-full-on P A
  shows almost-full-on (list-emb P) (lists A) (is almost-full-on ?P ?A)
proof (rule ccontr)
interpret mbs ?A .
assume ¬ thesis
from mbs¹ [OF this] obtain m
  where bad: m ∈ BAD ?P
      and min: ∀ g. (m, g) ∈ gseq → good ?P g ..
then have lists: ∧ i. m i ∈ lists A
  and ne: ∧ i. m i ≠ [] by auto

define h t where h = (λi. hd (m i)) and t = (λi. tl (m i))
have m: ∧ i. m i = h i # t i using ne by (simp add: h-def t-def)

have ∀ i. h i ∈ 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 ϑ :: nat ⇒ nat
  where less: ∧ i j. i < j ⇒ ϑ i < ϑ j
  and P: ∀ i j. i < j → P (h (ϑ i)) (h (ϑ j)) by blast
have bad-t: bad ?P (t ∘ ϑ)
proof
  assume good ?P (t ∘ ϑ)
  then obtain i j where i < j and ?P (t (ϑ i)) (t (ϑ j)) by auto
  moreover with P have ?P (h (ϑ i)) (h (ϑ j)) by blast
  ultimately have ?P (m (ϑ i)) (m (ϑ j)) by (subst (1 2) m) (rule list-emb-Cons2, auto)
  with less and (i < j): have good ?P m by (auto simp: good-def)
  with bad show False by blast
qed

define m' where m' = (λi. if i < ϑ 0 then m i else t (ϑ (i - ϑ 0)))

have m'-less: ∧ i. i < ϑ 0 ⇒ m' i = m i by (simp add: m'-def)
have m'-geq: ∧ i. i ≥ ϑ 0 ⇒ m' i = t (ϑ (i - ϑ 0)) by (simp add: m'-def)

have ∀ i. m' i ∈ lists A using ne-lists [OF ne] and lists by (auto simp: m'-def t-def)
moreover have length (m' (ϑ 0)) < length (m (ϑ 0)) using ne by (simp add: t-def m'-geq)
moreover have ∀ j<ϑ 0. m' j = m j by (auto simp: m'-less)
ultimately have (m, m') ∈ gseq using lists by (auto simp: gseq-def)
moreover have bad ?P m'
proof
  assume good ?P m'
  then obtain i j where i < j and emb: ?P (m' i) (m' j) by (auto simp: good-def)

33
{ assume \( j < \varphi 0 \)
  with \((i < j)\) and \(\text{emb}\) have \(?P (m \ i \ (m \ j))\) by (auto simp: \(m'\text{-less}\))
  with \((i < j)\) and \(\text{bad}\) have \(False\) by blast }

moreover
{ assume \( \varphi 0 \leq i \)
  with \((i < j)\) and \(\text{emb}\) have \(?P (t (\varphi (i - \varphi 0))) (t (\varphi (j - \varphi 0)))\)
  and \(i - \varphi 0 < j - \varphi 0\) by (auto simp: \(m'\text{-geq}\))
  with \(\text{bad-t}\) have \(False\) by blast }

moreover
{ assume \( i < \varphi 0 \) and \(\varphi 0 \leq j\)
  with \((i < j)\) and \(\text{emb}\) have \(?P (m \ i) (t (\varphi (j - \varphi 0))))\) by (simp add: \(m'\text{-less} m'\text{-geq}\))
  from \(\text{list-emb-Cons}\) [OF this, of \(h (\varphi (j - \varphi 0))\)]
  have \(?P (m \ i) (m (\varphi (j - \varphi 0))))\) using \(\text{ne}\) by (simp add: \(h\text{-def} t\text{-def}\))
  moreover have \(i < \varphi (j - \varphi 0)\)
  using \(\text{less}\) [of \(0 j - \varphi 0\)] and \(i < \varphi 0\) and \(\varphi 0 \leq j\)
  by (cases \(j = \varphi 0\)) auto
  ultimately have \(False\) using \(\text{bad}\) by blast }

ultimately show \(False\) using \(\text{min}\) by blast
qed

ultimately show \(False\) using \(\text{min}\) by blast
qed

7.6 Natural Numbers

lemma almost-full-on-UNIV-nat:
  almost-full-on \((\leq)\) (UNIV :: nat set)
proof --
  let \(?P = \text{subseq} :: \text{bool list} \Rightarrow \text{bool list} \Rightarrow \text{bool}\)
  have \(*: length \cdot (\text{UNIV} :: \text{bool list set}) = (\text{UNIV} :: \text{nat set})\)
    by (metis Ex-list-of-length surj-def)
  have almost-full-on \((\leq)\) (length \cdot (\text{UNIV} :: \text{bool list set}))
    proof (rule almost-full-on-hom)
      fix \(xs \ ys :: \text{bool list}\)
      assume \(?P \ xs \ ys\)
      then show \(\text{length} \ xs \leq \text{length} \ ys\)
        by (metis list-emb-length)
    next
      have \(\text{finite} (\text{UNIV} :: \text{bool set})\) by auto
      from almost-full-on-lists [OF eq-almost-full-on-finite-set [OF this]]
      show \(\text{almost-full-on} \ ?P \ \text{UNIV}\) unfolding lists-UNIV .
    qed
  then show \(\text{thesis}\) unfolding \(*\) .
qed

end
8 Well-Quasi-Orders

definition wqo-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool where
  wqo-on P A ←→ transp-on P A ∧ almost-full-on P A

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

lemma wqo-onI [Pure.intro]:
  [transp-on P A; almost-full-on P A] ⇒ wqo-on P A
  unfolding wqo-on-def almost-full-on-def by blast

lemma wqo-on-imp-reflp-on:
  wqo-on P A ⇒ 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 ⇒ transp-on P A
  by (auto simp: wqo-on-def)

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

lemma wqo-on-imp-qo-on:
  wqo-on P A ⇒ 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 ⇒ ∀ i. f i ∈ A ⇒ good P f
  by (auto simp: wqo-on-def almost-full-on-def)

lemma wqo-on-subset:
  A ⊆ B ⇒ wqo-on P B ⇒ 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

8.2 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** \(\text{wqo-af-conv}:\)
- assumes \(qo-on\ P\ A\)
- shows \(qo-on\ P\ A \iff \text{almost-full-on}\ P\ A\)
- using \(\text{assms}\) by (metis \(qo-on-def\ \text{wqo-on-def}\))

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

**lemma** \(\text{wqo-extensions-wf-conv}:\)
- assumes \(qo-on\ P\ A\)
- shows \(qo-on\ P\ A \iff (\forall Q. (\forall x \in A. \forall y \in A. P\ x\ y \rightarrow Q\ (h\ x)\ (h\ y)) \land qo-on\ Q\ A)
- unfolding \(\text{wqo-on-def}\)
- by blast

**lemma** \(\text{wqo-on-imp-wfp-on}:\)
- \(\text{wqo-on}\ P\ A \implies \text{wfp-on}\ \text{(strict } P\ \text{) } A\)
- by (metis (no-types) \(\text{wqo-on-imp-qo-on}\ \text{wqo-wf-and-no-antichain-conv}\))

The homomorphic image of a wqo set is wqo.

**lemma** \(\text{wqo-on-hom}:\)
- assumes \(\text{transp-on}\ Q\ \text{(h }\ A)\)
- and \(\forall x\in A. \forall y\in A. P\ x\ y \rightarrow Q\ (h\ x)\ (h\ y)\)
- and \(qo-on\ P\ A\)
- shows \(qo-on\ Q\ \text{(h }\ A)\)
- using \(\text{assms}\) and \(\text{almost-full-on-hom}\ [\text{of } A\ P\ Q\ h]\)
- unfolding \(\text{wqo-on-def}\) by blast

The monomorphic preimage of a wqo set is wqo.

**lemma** \(\text{wqo-on-mon}:\)
- assumes \(*:\ \forall x\in A. \forall y\in A. P\ x\ y \iff Q\ (h\ x)\ (h\ y)\)
- and \(\text{bij}: \text{bij-betw}\ h\ A\ B\)
- and \(\text{wqo}: \text{wqo-on}\ Q\ B\)
- shows \(\text{wqo-on}\ P\ A\)
proof –

have transp-on P A
proof
  fix x y z assume [intro!]: x ∈ A y ∈ A z ∈ 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) 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

8.3 A Type Class for Well-Quasi-Orders

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

class wqo = preorder +
assumes good: good (≤) f

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

lemma wqo-on-UNIV-class-wqo [intro!]:
wqo-on P UNIV ⇒ 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 wqo-on (for the special case that the domain is the universe of a type) and the class predicate class.wqo.

lemma wqo-on-UNIV-conv:
wqo-on P UNIV ↔ class.wqo P (strict P) (is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs by auto
next
  assume ?rhs then show ?lhs
    unfolding class.wqo-def class.preorder-def class.wqo-axioms-def
    by (auto simp: wqo-on-def almost-full-on-def transp-on-def)
qed

The strict part of a wqo is well-founded.

lemma (in wqo) wfP (<)
proof –
  have class.wqo (≤) (<) ..
hence wqo-on (≤) UNIV
from wqo-on-imp-wfp-on [OF this]
  show thesis unfolding less-le-not-le [abs-def] wfp-on-UNIV .
qed

lemma wqo-on-with-bot:
  assumes wqo-on P A
  shows wqo-on (option-le P) A⊥ (is wqo-on ?P ?A)
proof –
  { from assms have trans [unfolded transp-on-def]: transp-on P A
    by (auto simp: wqo-on-def)
    have transp-on ?P ?A
      by (auto simp: transp-on-def elim: with-bot-cases, insert trans) blast }
moreover
  { from assms and almost-full-on-with-bot
    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-option-UNIV [intro]:
  wqo-on P UNIV ⇒ wqo-on (option-le P) UNIV
using wqo-on-with-bot [of P UNIV] by simp

When two sets are wqo, then their disjoint sum is wqo.

lemma wqo-on-Plus:
  assumes wqo-on P A and wqo-on Q B
  shows wqo-on (sum-le P Q) (A <+> B) (is wqo-on ?P ?A)
proof –
  { from assms have trans [unfolded transp-on-def]: transp-on P A transp-on Q B
    by (auto simp: wqo-on-def)
    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

8.4 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 \textit{wqo-on-}\((\text{prod-le } P_1 P_2)\ (A_1 \times A_2)\) \((\text{is wqo-on } ?P \ ?A)\)

proof –
\begin{itemize}
  \item \text{from assms have transp-on } P_1 A_1 \text{ and transp-on } P_2 A_2 \text{ by (auto simp: wqo-on-def)}
  \item hence transp-on ?P ?A unfolding transp-on-def prod-le-def by blast}
\end{itemize}

moreover
\begin{itemize}
  \item \text{from assms and almost-full-on-Sigma [of } P_1 A_1 P_2 A_2]\text{ have almost-full-on } ?P ?A \text{ by (auto simp: wqo-on-def)}
\end{itemize}

ultimately
\begin{itemize}
  \item show \(\)thesis by (auto simp: wqo-on-def)
\end{itemize}

qed

lemmas dickson = wqo-on-Sigma

\textbf{8.5} \textbf{Higman’s Lemma}

\textbf{lemma} transp-on-list-emb:
\begin{itemize}
  \item \text{assumes transp-on } P \ A
  \item \text{shows transp-on } (\text{list-emb } P) \ (\text{lists } A)
  \item \text{using assms and list-emb-trans [of - - - } P]
  \item unfolding transp-on-def by blast
\end{itemize}

\textbf{lemma} wqo-on-lists:
\begin{itemize}
  \item \text{assumes wqo-on } P \ A \text{ shows wqo-on } (\text{list-emb } P) \ (\text{lists } A)
  \item \text{using assms and almost-full-on-lists}
  \item and transp-on-list-emb by (auto simp: wqo-on-def)
\end{itemize}

\textbf{lemmas} higman = wqo-on-lists

\textbf{lemma} wqo-on-list-UNIV [intro]:
\begin{itemize}
  \item \text{wqo-on } P \ \text{UNIV } \Rightarrow \ wqo-on \ (\text{list-emb } P \ \text{UNIV} \Rightarrow \text{wqo-on-}\text{prod-le } P \ Q) \ \text{UNIV}
  \item \text{using wqo-on-Sigma [of } P \ \text{UNIV } Q \ \text{UNIV] by simp}
\end{itemize}

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

\textbf{lemma} finite-wqo-on:
\begin{itemize}
  \item \text{assumes finite } A \text{ and refl: reflp-on } P A \text{ and transp-on } P \ A
  \item \text{shows wqo-on } P \ A
  \item \text{using assms and finite-almost-full-on by (auto simp: wqo-on-def)}
\end{itemize}

\textbf{lemma} finite-eq-wqo-on:
\begin{itemize}
  \item \text{assumes finite } A
  \item \text{shows wqo-on } (\text{=} ) \ A
  \item \text{using finite-wqo-on [OF assms, of } (\text{=})]
  \item \text{by (auto simp: reflp-on-def transp-on-def)}
\end{itemize}
lemma wqo-on-lists-over-finite-sets:
  wqo-on (list-emb (=)) (UNIV :: ('a :: finite) list set)
  using wqo-on-lists [OF finite-eq-wqo-on [OF finite [of UNIV :: ('a :: finite) set]]]
by simp

lemma wqo-on-map:
  fixes P and Q and h
defines P' ≡ λx y. P x y ∧ Q (h x) (h y)
assumes wqo-on P A
and wqo-on Q B
and subset: h ' A ⊆ B
shows wqo-on P' A
proof
  let ?Q = λx y. Q (h x) (h y)
  from ⟨wqo-on P A⟩ have transp-on P A by (rule wqo-on-imp-transp-on)
  then show transp-on P' A
    using ⟨wqo-on Q B⟩ and subset
    unfolding wqo-on-def transp-on-def P'-def by blast
from ⟨wqo-on P A⟩ have almost-full-on P A
  by (rule wqo-on-imp-almost-full-on)
from ⟨wqo-on Q B⟩ have almost-full-on Q B
  by (rule wqo-on-imp-almost-full-on)

show almost-full-on P' A
proof
  fix f
  assume *: ∀i :: nat. f i ∈ A
  from almost-full-on-imp-homogeneous-subseq [OF ⟨almost-full-on P A: this⟩]
  obtain g :: nat ⇒ nat
    where g: ∀i j. i < j ⇒ g i < g j
    and **: ∀i. f (g i) ∈ A ∧ P (f (g i)) (f (Suc i)))
    using * by auto
  from chain-transp-on-less [OF ** ⟨transp-on P A⟩]
  have **: ∀i j. i < j ⇒ P (f (g i)) (f (g j)) .
  let ?g = λi. h (f (g i))
  from * and subset have B: ∀i. ?g i ∈ B by auto
  with ⟨almost-full-on Q B: [unfolded almost-full-on-def good-def, THEN bspec, of ?g]⟩
  obtain i j :: nat
    where i < j and Q (?g i) (?g j) by blast
  with ** [OF i < j] have P' (f (g i)) (f (g j))
    by (auto simp: P'-def)
  with g [OF i < j] show good P' f by (auto simp: good-def)
qed
qed

lemma wqo-on-UNIV-nat:
locale kruskal-tree =  
fixes F :: (′b × nat) set  
and mk :: ′b ⇒ ′a list ⇒ (′a::size)  
and root :: ′a ⇒ ′b × nat  
and args :: ′a ⇒ ′a list  
and trees :: ′a set  
assumes size-arg: t ∈ trees ⇒ s ∈ set (args t) ⇒ size s < size t  
and root-mk: (f, length ts) ∈ F ⇒ root (mk f ts) = (f, length ts)  
and args-mk: (f, length ts) ∈ F ⇒ args (mk f ts) = ts  
and mk-root-args: t ∈ trees ⇒ mk (fst (root t)) (args t) = t  
and trees-root: t ∈ trees ⇒ root t ∈ F  
and trees-arity: t ∈ trees ⇒ length (args t) = snd (root t)  
and trees-args: ∀ s. t ∈ trees ⇒ s ∈ set (args t) ⇒ s ∈ trees  
begin  
lemma mk-inject [iff]:  
assumes (f, length ss) ∈ F and (g, length ts) ∈ F  
shows mk f ss = mk g ts ⟷ f = g ∧ ss = ts  
proof −  
{ assume mk f ss = mk g ts  
  then have root (mk f ss) = root (mk g ts)  
  and args (mk f ss) = args (mk g ts) by auto }  
show ?thesis  
using root-mk [OF assms(1)] and root-mk [OF assms(2)]  
and args-mk [OF assms(1)] and args-mk [OF assms(2)] by auto  
qed  

inductive emb for P  
where  
arg: [(f, m) ∈ F; length ts = m; ∀ t ∈ set ts. t ∈ trees;  
t ∈ set ts; emb P s t] ⟷ emb P s (mk f ts)  
list-emb: [(f, m) ∈ F; (g, n) ∈ F; length ss = m; length ts = n;  
∀ s ∈ set ss. s ∈ trees; ∀ t ∈ set ts. t ∈ trees;  
P (f, m) (g, n); list-emb (emb P) ss ts] ⟷ emb P (mk f ss) (mk g ts)  
monos list-emb-mono  

lemma almost-full-on-trees:
assumes almost-full-on \( P F \)
shows almost-full-on \((\text{emb} \; P) \; \text{trees} \) (is almost-full-on \( ?P \) \( ?A \))
proof (rule ccontr)
  interpret \( \text{mbs} \; ?A \).
  assume \( \neg \; \text{thesis} \)
  from \( \text{mbs} \; \text{OF this} \) 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 \( \text{trees} : \bigwedge i. m \in \text{trees} \) by auto

  define \( r \) where \( r \; i = \text{root} \; (m \; i) \) for \( i \)
  define \( a \) where \( a \; i = \text{args} \; (m \; i) \) for \( i \)
  define \( S \) where \( S = \bigcup \{ \text{set} \; (a \; i) \mid i. \; \text{True} \} \)

  have \( m : \bigwedge i. m \; i = \text{mk} \; (\text{fst} \; (r \; i)) \; (a \; i) \)
    by (simp add: \text{r-def} \; \text{a-def} \; \text{mk-root-args} \; \text{OF} \; \text{trees})
  have \( \text{lists} : \forall i. a \; i \in \text{lists} \; S \) by (auto simp: \text{a-def} \; \text{S-def})
  have \( \text{arity} : \bigwedge i. \text{length} \; (a \; i) = \text{snd} \; (r \; i) \)
    using \( \text{trees-arity} \; \text{OF} \; \text{trees} \) by (auto simp: \text{r-def} \; \text{a-def})
  then have \( \text{sig} : \bigwedge i. (\text{fst} \; (r \; i), \text{length} \; (a \; i)) \in F \)
    using \( \text{trees-root} \; \text{OF} \; \text{trees} \) by (auto simp: \text{a-def} \; \text{r-def})
  have \( \text{a-trees} : \bigwedge i. \forall t \in \text{set} \; (a \; i). t \in \text{trees} \) by (auto simp: \text{a-def} \; \text{trees-args} \; \text{OF} \; \text{trees})

  have \( \text{almost-full-on} \; ?P \; S \)
  proof (rule ccontr)
    assume \( \neg \; \text{thesis} \)
    then obtain \( s :: \; \text{nat} \; \Rightarrow \; \text{'}a \)
      where \( S : \bigwedge i. s \; i \in S \) and \( \text{bad-s} : \forall \; s. \; \text{bad} \; ?P \; s \) by (auto simp: \text{almost-full-on-def})

    define \( n \) where \( n = (\text{LEAST} \; n. \exists k. s \; k \in \text{set} \; (a \; n)) \)
    have \( \exists n. \exists k. s \; k \in \text{set} \; (a \; n) \) using \( S \) by (force simp: \text{S-def})
    from \( \text{LeastI-ex} \; \text{OF this} \) obtain \( k \)
      where \( \text{sk} : s \; k \in \text{set} \; (a \; n) \) by (auto simp: \text{n-def})
    have \( \exists k. \exists m \geq n. s \; k \in \text{set} \; (a \; m) \)
      using \( S \) by (auto simp: \text{S-def}) (metis \text{Least-le} \; \text{n-def})

    define \( m' \) where \( m' \; i = (\text{if} \; i < n \; \text{then} \; m \; i \; \text{else} \; s \; (k + (i - n))) \) for \( i \)

    have \( m'\text{-less} : \bigwedge i. i < n \Rightarrow m' \; i = m \; i \) by (simp add: \text{m'-def})
    have \( m'\text{-geq} : \bigwedge i. i \geq n \Rightarrow m' \; i = s \; (k + (i - n)) \) by (simp add: \text{m'-def})

    have \( \text{bad} \; ?P \; m' \)
    proof
      assume \( \text{good} \; ?P \; m' \)
      then obtain \( i \; j \) where \( i < j \) and \( \text{emb} : ?P \; (m' \; i) \; (m' \; j) \) by auto
      {  assume \( j < n \)
      with \( i < j \) and \( \text{emb} \) have \( ?P \; (m \; i) \; (m' \; j) \) by (auto simp: \text{m'-less})
      with \( i < j \) and \( \text{bad} \) have \( \text{False} \) by blast }

42
moreover
\{ assume \( n \leq i \) with \( i < j \) and \( \text{emb} \) have \( ?P \ (s \ (k + (i - n))) \ (s \ (k + (j - n))) \)
and \( k + (i - n) < k + (j - n) \) by (auto simp: \( m' - \text{geq} \))
with \( \text{bad-s} \) have False by auto \}
moreover
\{ assume \( i < n \) and \( n \leq j \) with \( i < j \) and \( \text{emb} \) have \(*: ?P \ (m \ i) \ (s \ (k + (j - n))) \) by (auto simp: \( m' - \text{less} \ \text{m' - geq} \))
with \( \text{args} \) obtain \( l \) where \( l \geq n \) and \(*: s \ (k + (j - n)) \in \text{set} \ (\text{a} \ l) \) by blast
from \( \text{emb-arg} \ [OF \ \text{sig} \ [\text{of} \ l] \ \text{- a-trees} \ [\text{of} \ l] \ *] \)
have \( ?P \ (m \ i) \ (m \ l) \) by (simp add: \( m \))
moreover have \( i < l \) using \( i < n \) and \( n \leq l \) by auto
ultimately have False using \( \text{bad} \) by blast \}
ultimately show \( \text{False} \) using \( \text{False} \) (\( i < j \)) by arith
qed
moreover have \( (m, m') \in \text{gseq} \)
proof –
have \( m \in \text{SEQ} \ ?A \) using \( \text{trees} \) by auto
moreover have \( m' \in \text{SEQ} \ ?A \)
using \( \text{trees} \) and \( S \) and \( \text{trees-args} \ [OF \ \text{trees}] \) by (auto simp: \( m' - \text{def} \ \text{a-def} \ \text{S-def} \))
moreover have \( \forall i < n. m \ i = m' \ i \) by (auto simp: \( m' - \text{less} \))
moreover have \( \text{size} \ (m' \ n) \ < \ \text{size} \ (m \ n) \)
using \( \text{sk} \) and \( \text{size-arg} \ [OF \ \text{trees}, \ \text{unfolded} \ m] \)
by (auto simp: \( m \ m' - \text{geq} \ \text{root-mk} \ [OF \ \text{sig}] \ \text{args-mk} \ [OF \ \text{sig}] \))
ultimately show \( \? \text{thesis} \) by (auto simp: \( \text{gseq-def} \))
qed
ultimately show \( \text{False} \) using \( \text{min} \) by blast
qed
from \( \text{almost-full-on-lists} \ [OF \ \text{this}, \ \text{THEN} \ \text{almost-full-on-imp-homogeneous-subseq}, \ \text{OF \ lists}] \)
obtain \( \varphi :: \text{nat} \Rightarrow \text{nat} \)
where less: \( \forall i. j. i < j \Rightarrow \varphi \ i < \varphi \ j \)
and lemb: \( \forall i. j. i < j \Rightarrow \text{list-emb} \ ?P \ (a \ \varphi \ i) \ (a \ \varphi \ j) \) by blast
have roots: \( \forall i. r \ \varphi \ i \in F \) using \( \text{trees} \ [THEN \ \text{trees-root}] \) by (auto simp: \( \text{r-def} \))
then have \( r \circ \varphi \in \text{SEQ} \ F \) by auto
with \( \text{assms} \) have \( \text{good} \ P \ (r \circ \varphi) \) by (auto simp: \( \text{almost-full-on-def} \))
then obtain \( i \ j \)
where \( i < j \) and \( P \ (r \ \varphi \ i) \ (r \ \varphi \ j) \) by auto
with \( \text{lemb} \ [OF \ i < j] \) have \( ?P \ (m \ \varphi \ i) \ (m \ \varphi \ j) \)
using \( \text{sig} \) and \( \text{arity} \) and \( \text{a-trees} \) by (auto simp: \( m' \ \text{intro!} : \text{emb-list-emb} \))
with \( \text{less} \ [OF \ i < j] \) and \( \text{bad} \) show \( \text{False} \) by blast
qed
inductive-cases
\( \text{emb-mk2} \ [\text{consumes} \ 1, \ \text{case-names} \ \text{arg} \ \text{list-emb}] : \text{emb} \ P \ s \ (\text{mk} \ g \ ts) \)
inductive-cases
  list-emb-Nil2-cases: list-emb P xs [] and
  list-emb-Cons-cases: list-emb P xs (y#ys)

lemma list-emb-trans-right:
  assumes list-emb P xs ys and list-emb (λy z. P y z ∧ (∀x. P x y → P x z)) ys zs
  shows list-emb P xs zs
  using assms(2, 1) by (induct arbitrary: xs) (auto elim: list-emb-Nil2-cases
  list-emb-Cons-cases)

lemma emb-trans:
  assumes trans: (∧ f g h. f ∈ F ⇒ g ∈ F ⇒ h ∈ F ⇒ P f g ⇒ P g h ⇒ P f h)
  assumes emb P s t and emb P t u
  shows emb P s u
  using assms(3, 2)
  proof (induct arbitrary: s)
    case (arg f m ts v)
    then show ?case by (auto intro: emb.arg)
  next
  case (list-emb f m g n ss ts)
  note IH = this
  from (emb P s (mk f ss))
  show ?case
  proof (cases rule: emb-mk2)
    case arg
    then show ?thesis using IH by (auto elim!: list-emb-set intro: emb.arg)
  next
  case list-emb
  then show ?thesis using IH by (auto intro: emb.intros dest: trans list-emb-trans-right)
  qed

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

lemma kruskal:
  assumes wqo-on P F
  shows wqo-on (emb P) trees
  using almost-full-on-trees [of P] and assms by (metis transp-on-emb wqo-on-def)

end

end

theory Kruskal-Examples
imports Kruskal
datatype 'a tree = Node 'a 'a tree list

fun node where node (Node f ts) = (f, length ts)

fun succs where succs (Node f ts) = ts

inductive-set trees for A where
  f ∈ A ⇒ ∀ t ∈ set ts. t ∈ trees A ⇒ Node f ts ∈ trees A

lemma [simp]:
  trees UNIV = UNIV
proof —
  { fix t :: 'a tree
    have t ∈ trees UNIV
      by (induct t) (auto intro: trees.intros) }
  then show thesis by auto
qed

interpretation kruskal-tree-tree: kruskal-tree A × UNIV Node node succs trees A for A
  apply (unfold-locales)
  apply auto
  apply (case-tac !! t rule: trees.cases)
  apply auto
  by (metis less-not-refl not-less-eq size-list-estimation)

thm kruskal-tree-tree.almost-full-on-trees
thm kruskal-tree-tree.kruskal

definition tree-emb A P = kruskal-tree-tree.emb A (prod-le P (λ- -. True))

lemma wqo-on-trees:
  assumes wqo-on P A
  shows wqo-on (tree-emb A P) (trees A)
  using wqo-on-Sigma [OF assms wqo-on-UNIV, THEN kruskal-tree-tree.kruskal]
  by (simp add: tree-emb-def)

If the type 'a is well-quasi-ordered by P, then trees of type 'a tree are well-
quasi-ordered by the homeomorphic embedding relation.

instantiation tree :: (wqo) wqo
begin
definition s ≤ t ←→ tree-emb UNIV (≤) s t

45
definition \((s : \text{\texttt{'a tree}}) < t \iff s \leq t \land \neg (t \leq s)\)

instance
  by (rule class.wqo.of-class.intro)
    (auto simp: less-eq-tree-def [abs-def] less-tree-def [abs-def]
       intro: wqo-on-trees [of - UNIV; simplified])

end

datatype (\texttt{f}, \texttt{v}) term = \texttt{Var} \texttt{v} | \texttt{Fun} \texttt{f} (\texttt{f}, \texttt{v}) term list

fun root
where
    root (Fun f ts) = (f, length ts)

fun args
where
   args (Fun f ts) = ts

inductive-set gterms for F
where
    \((f, n) \in F \implies length ts = n \implies \forall s \in set ts. s \in gterms F \implies Fun f ts \in gterms F\)

interpretation kruskal-term: kruskal-tree F Fun root args gterms F for F
  apply (unfold-locales)
  apply auto
  apply (case-tac [|] t rule: gterms.cases)
  apply auto
  by (metis less-not-refl not-less-eq size-list-estimation)

thm kruskal-term.almost-full-on-trees

inductive-set terms
where
    \(\forall t \in set ts. t \in terms \implies Fun f ts \in terms\)

interpretation kruskal-variadic: kruskal-tree UNIV Fun root args terms
  apply (unfold-locales)
  apply auto
  apply (case-tac [|] t rule: terms.cases)
  apply auto
  by (metis less-not-refl not-less-eq size-list-estimation)

thm kruskal-variadic.almost-full-on-trees

datatype \texttt{\texttt{a exp}} = \texttt{V} \texttt{a} | \texttt{C} \texttt{nat} | \texttt{Plus} \texttt{\texttt{a exp}} \texttt{\texttt{a exp}}

datatype \texttt{\texttt{a symb}} = \texttt{v} \texttt{a} | \texttt{c} \texttt{nat} | \texttt{p}
fun \( mk \) where
\[
\begin{align*}
\text{mk} (v \, x) &\mid = V \, x \\
\text{mk} (c \, n) &\mid = C \, n \\
\text{mk} p [a, \, b] &\mid = \text{Plus} \, a \, b
\end{align*}
\]

fun \( rt \) where
\[
\begin{align*}
\text{rt} (V \, x) &\mid = (v \, x, 0::\text{nat}) \\
\text{rt} (C \, n) &\mid = (c \, n, 0) \\
\text{rt} (\text{Plus} \, a \, b) &\mid = (p, \, 2)
\end{align*}
\]

fun \( ags \) where
\[
\begin{align*}
\text{ags} (V \, x) &\mid = [] \\
\text{ags} (C \, n) &\mid = [] \\
\text{ags} (\text{Plus} \, a \, b) &\mid = [a, \, b]
\end{align*}
\]

inductive-set \( \text{exps} \) where
\[
\begin{align*}
V \, x &\in \text{exps} \\
C \, n &\in \text{exps} \\
a &\in \text{exps} \Rightarrow b &\in \text{exps} \Rightarrow \text{Plus} \, a \, b &\in \text{exps}
\end{align*}
\]

lemma \[ \text{simp} \]:
\[
\begin{align*}
\text{assumes} \ &\text{length} \ \text{ts} = 2 \\
\text{shows} \ &\text{rt} (\text{mk} \ p \ \text{ts}) = (p, \, 2) \\
\text{using} \ &\text{assms} \ \text{by} \ (\text{induct} \ \text{ts}) \ (\text{auto, case-tac} \ \text{ts, auto})
\end{align*}
\]

lemma \[ \text{simp} \]:
\[
\begin{align*}
\text{assumes} \ &\text{length} \ \text{ts} = 2 \\
\text{shows} \ &\text{ags} (\text{mk} \ p \ \text{ts}) = \text{ts} \\
\text{using} \ &\text{assms} \ \text{by} \ (\text{induct} \ \text{ts}) \ (\text{auto, case-tac} \ \text{ts, auto})
\end{align*}
\]

interpretation \( \text{kruskal-exp: kruskal-tree} \)
\[
\{(v \, x, \, 0) \mid x. \ \text{True}\} \cup \{(c \, n, \, 0) \mid n. \ \text{True}\} \cup \{(p, \, 2)\}
\]

mk \ rt \ ags \ exps
apply (unfold-locals)
apply auto
apply (case-tac \[ \] t \ rule: \ exps.cases)
by auto

thm \( \text{kruskal-exp.almost-full-on-trees} \)

hide-const \( \text{open} \) \( \text{tree-emb} \ V \ C \ \text{Plus} \ v \ c \ p \)

end
10 Instances of Well-Quasi-Orders

theory Wqo-Instances
imports Kruskal
begin

10.1 The Option Type is Well-Quasi-Ordered

instantiation option :: (wqo) wqo
begin
definition  \( x \leq y \iff \text{option-le} (\leq) x y \)
definition  \( x :: 'a \text{ option} \prec y \iff x \leq y \land \neg (y \leq x) \)

instance
  by (rule class.wqo.of-class.intro)
    (auto simp: less-eq-option-def [abs_def] less-option-def [abs_def])
end

10.2 The Sum Type is Well-Quasi-Ordered

instantiation sum :: (wqo, wqo) wqo
begin
definition  \( x \leq y \iff \text{sum-le} (\leq) (\leq) x y \)
definition  \( x :: 'a + 'b \prec y \iff x \leq y \land \neg (y \leq x) \)

instance
  by (rule class.wqo.of-class.intro)
    (auto simp: less-eq-sum-def [abs_def] less-sum-def [abs_def])
end

10.3 Pairs are Well-Quasi-Ordered

If types 'a and 'b are well-quasi-ordered by \( P \) and \( Q \), then pairs of type 'a \( \times \) 'b are well-quasi-ordered by the pointwise combination of \( P \) and \( Q \).

instantiation prod :: (wqo, wqo) wqo
begin
definition  \( p \leq q \iff \text{prod-le} (\leq) (\leq) p q \)
definition  \( p :: 'a \times 'b \prec q \iff p \leq q \land \neg (q \leq p) \)

instance
  by (rule class.wqo.of-class.intro)
    (auto simp: less-eq-prod-def [abs_def] less-prod-def [abs_def])
end

10.4 Lists are Well-Quasi-Ordered

If the type 'a is well-quasi-ordered by \( P \), then lists of type 'a list are well-quasi-ordered by the homeomorphic embedding relation.
### instantiation

\[
\text{list} :: \langle \text{wqo} \rangle \ wqo
\]

**begin**

**definition** \( xs \leq ys \iff \text{list-emb} (\leq) xs ys \)

**definition** \( (xs :: \langle \text{'a list} \rangle) < ys \iff xs \leq ys \land \neg (ys \leq xs) \)

**instance**

by (rule class.wqo.of-class.intro)

(auto simp: less-eq-list-def [abs-def] less-list-def [abs-def])

**end**

**end**

### 11 Multiset Extension of Orders (as Binary Predicates)

**theory** Multiset-Extension

**imports**

Open-Induction, Restricted-Predicates

HOL-Library.Multiset

**begin**

**definition** multisets :: \( \langle \text{'a set} \rangle \Rightarrow \langle \text{'a multiset} \rangle \)

where

\[
\text{multisets } A = \{ M. \text{set-mset } M \subseteq A \}
\]

**lemma** in-multisets-iff:

\[
M \in \text{multisets } A \iff \text{set-mset } M \subseteq A
\]

by (simp add: multisets-def)

**lemma** empty-multisets [simp]:

\[
\{\#\} \in \text{multisets } F
\]

by (simp add: in-multisets-iff)

**lemma** multisets-union [simp]:

\[
M \in \text{multisets } A \Longrightarrow N \in \text{multisets } A \Longrightarrow M + N \in \text{multisets } A
\]

by (auto simp add: in-multisets-iff)

**definition** mulex1 :: \( \langle \text{'a set} \Rightarrow \langle \text{'a multiset} \Rightarrow \langle \text{'a multiset} \Rightarrow \text{bool} \rangle \rangle \)

where

\[
mulex1 P = (\lambda M. N. (M, N) \in \text{mult1 } \{(x, y). (x \Rightarrow y)\})
\]

**lemma** mulex1-empty [iff]:

\[
mulex1 P M \{\#\} \iff False
\]

using not-less-empty [of M \{(x, y). P x y\}]

by (auto simp: mulex1-def)

**lemma** mulex1-add: mulex1 P N (M0 + \{#a#\}) \(\Longrightarrow\)

\[
(\exists M. \text{mulex1 } P M M0 \wedge N = M + \{#a#\}) \lor
(\exists K. (\forall b. b \in \# K \Longrightarrow P b a) \wedge N = M0 + K)
\]

using less-add [of N a M0 \{(x, y). P x y\}]

49
by (auto simp: mulex1-def)

lemma mulex1-self-add-right [simp]:
mulex1 P A (add-mset a A)
proof -
  let ?R = \{(x, y). P x y\}
  thm mult1-def
  have A + (#a#) = A + (#a#) by simp
  moreover have A = A + (#) by simp
  moreover have \forall b. b ∈ (#) → (b, a) ∈ ?R by simp
  ultimately have (A, add-mset a A) ∈ mult1 ?R
    unfolding mult1-def by blast
  then show ?thesis by (simp add: mulex1-def)
qed

lemma empty-mult1 [simp]:
  (\{#\}, \{#a#\}) ∈ mult1 R
proof -
  have \{#a#\} = \{#\} + \{#a#\} by simp
  moreover have \{#\} = \{#\} + (#) by simp
  moreover have \forall b. b ∈ (#) → (b, a) ∈ R by simp
  ultimately show ?thesis unfolding mult1-def by force
qed

lemma empty-mulex1 [simp]:
mulex1 P \{#\} \{#a#\}
using empty-mult1 [of a \{(x, y). P x y\}] by (simp add: mulex1-def)

definition mulex-on :: \(\:\{\:\arrow a \rightarrow (\rightarrow (\rightarrow bool)} \rightarrow \rightarrow bool\)
where
  mulex-on P A M N = (restrict-to (mulex1 P) (multisets A))++

abbreviation mulex :: \(\:\{\:\arrow a \rightarrow bool)} \rightarrow \rightarrow bool\)
where
  mulex P ≡ mulex-on P UNIV

lemma mulex-on-induct [consumes 1, case-names base step, induct pred: mulex-on]:
assumes mulex-on P A M N
  and \(\forall M N. [M ∈ multisets A; N ∈ multisets A; mulex1 P M N] \implies Q M N\)
  and \(\forall L M N. [mulex-on P A L M; Q L M; N ∈ multisets A; mulex1 P M N] \implies Q L N\)
shows Q M N
using assms unfolding mulex-on-def by (induct) blast+

lemma mulex-on-self-add-singleton-right [simp]:
assumes a ∈ A and M ∈ multisets A
shows mulex-on P A M (add-mset a M)
proof -
  have mulex1 P M (M + (#a#)) by simp
with assms have restrict-to (mulex1 P) (multisets A) M (add-mset a M)
by (auto simp: multisets-def)
then show ?thesis unfolding mulex-on-def by blast 
qed

lemma singleton-multisets [iff]:
{#x#} ∈ multisets A ←→ x ∈ A
by (auto simp: multisets-def)

lemma union-multisetsD:
assumes M + N ∈ multisets A
shows M ∈ multisets A ∧ N ∈ multisets A
using assms by (auto simp: multisets-def)

lemma mulex-on-multisetsD [dest]:
assumes mulex-on P F M N
shows M ∈ multisets F and N ∈ multisets F
using assms by (induct) auto

lemma union-multisets-iff [iff]:
M + N ∈ multisets A ←→ M ∈ multisets A ∧ N ∈ multisets A
by (auto dest: union-multisetsD)

lemma add-mset-multisets-iff [iff]:
add-mset a M ∈ multisets A ←→ a ∈ A ∧ M ∈ multisets A
unfolding add-mset-add-single[of a M] union-multisets-iff by auto

lemma mulex-on-trans:
mulex-on P A L M = ⇒ mulex-on P A M N = ⇒ mulex-on P A L N
by (auto simp: mulex-on-def)

lemma transp-on-mulex-on:
transp-on (mulex-on P A) B
using mulex-on-trans [of P A] by (auto simp: transp-on-def)

lemma mulex-on-add-right [simp]:
assumes mulex-on P A M N and a ∈ A
shows mulex-on P A M (add-mset a N)
proof –
  from assms have a ∈ A and N ∈ multisets A by auto
  then have mulex-on P A N (add-mset a N) by simp
with ⟨mulex-on P A M N⟩ show ?thesis by (rule mulex-on-trans) 
qed

lemma empty-mulex-on [simp]:
assumes M ≠ {#} and M ∈ multisets A
shows mulex-on P A {#} M
using assms
proof (induct M)
case (add a M)
show ?case
proof (cases M = {#})
  assume M = {#}
  with add show ?thesis by (auto simp: mulex-on-def)
next
  assume M ≠ {#}
  with add show ?thesis by (auto intro: mulex-on-trans)
qed
qed simp

lemma mulex-on-self-add-right [simp]:
  assumes M ∈ multisets A and K ∈ multisets A and K ≠ {#}
  shows mulex-on P A M (M + K)
using assms
proof (induct K)
  case empty
  then show ?case by (cases K = {#}) auto
next
  case (add a M)
  assume M = {#} with add show ?thesis by auto
next
  assume M ≠ {#} with add show ?thesis
    by (auto dest: mulex-on-add-right simp add: ac-simps)
qed

lemma mult1-singleton [iff]:
  ({{#x#}, {#y#}}) ∈ mult1 R ←→ (x, y) ∈ R
proof
  assume (x, y) ∈ R
  then have {#y#} = {#} + {#y#} and {#x#} = {#} + {#x#}
    and ∀ b. b ∈ # {#x#} → (b, y) ∈ R by auto
  then show ({{#x#}, {#y#}}) ∈ mult1 R unfolding mult1-def by blast
next
  assume ({{#x#}, {#y#}}) ∈ mult1 R
  then obtain M0 K a
    where {#y#} = add-mset a M0
    and {#x#} = M0 + K
    and ∀ b. b ∈ # K → (b, a) ∈ R
    unfolding mult1-def by blast
  then show (x, y) ∈ R by (auto simp: add-eq-conv-diff)
qed

lemma mulex1-singleton [iff]:
  mulex1 P {#x#} {#y#} ←→ P x y
using mult1-singleton \([of \, x \, y \, \{(x, \, y), \, P \, x \, y\}]\) by (simp add: mulex1-def)

lemma singleton-mulex-onI:
\[ P \, x \, y \implies x \in A \implies y \in A \implies \text{mulex-on} \, P \, A \, \{\#x\#\} \, \{\#y\#\} \]
by (auto simp: mulex-on-def)

lemma reflclp-mulex-on-add-right \([simp]\):
assumes \(((\text{mulex-on} \, P \, A) = M \, N) \, \text{and} \, M \in \text{multisets} \, A \, \text{and} \, a \in A\)
shows \(((\text{mulex-on} \, P \, A \, M, \, \{\#a\#\}) + N)\)
using reflclp-mulex-on-add-right \([OF \, \text{assms}]\) by (simp add: ac-simps)

lemma mulex-on-union-right \([simp]\):
assumes \(((\text{mulex-on} \, P \, F \, A \, B) \, \text{and} \, K \in \text{multisets} \, F)\)
shows \(((\text{mulex-on} \, P \, F \, A, \, K + B)\)
using assms
proof (induct K)
case (add a K)
then have \(a \in F \, \text{and} \, \text{mulex-on} \, P \, F \, A, \, (B + K)\) by (auto simp: multisets-def ac-simps)
then have \(((\text{mulex-on} \, P \, F \, A, \, (B + K) + \{\#a\#\})\) by simp
then show \(?case\) by (simp add: ac-simps)
qed simp

Adapted from \(\text{wf} \, ?r \implies \forall \, M. \, M \in \text{Wellfounded.acc} \, (\text{mult1} \, ?r)\) in \(\text{HOL-Library.Multiset}\).

lemma accessible-on-mulex1-multisets:
assumes \(((\text{wf-on} \, P \, A)\)
shows \(\forall \, M \in \text{multisets} \, A. \, \text{accessible-on} \, (\text{mulex1} \, P) \, (\text{multisets} \, A) \, M\)
proof
let \(?P = \text{mulex1} \, P\)
let \(?A = \text{multisets} \, A\)
let \(?acc = \text{accessible-on} \, ?P \, ?A\)
\{
fix \(M \, M0 \, a\)
assume \(?M0: \, \text{?acc} \, M0\)
and \(?a: \, a \in A\)
and \(?M0: \, \text{?a}\)
and \((\text{wf-hyp:} \, \forall b. \, [b \in A; \, P \, b \, a] \implies (\forall M. \, \text{?acc} \, (M) \implies \text{?acc} \, (M + \{\#b\#\})))\)
and \(\text{acc-hyp:} \, \forall M. \, M \in \, ?A \, \land \, \text{?P \, M \, M0} \implies \text{?acc} \, (M + \{\#a\#\})))\)
then have \((\text{add-mset} \, a \, M0 \in \, ?A)\) by (auto simp: multisets-def)
then have \(?\text{acc} (\text{add-mset} \ a \ \text{M}0)\)
proof (rule accessible-onI [of \text{add-mset} \ a \ \text{M}0])
  fix \(N\)
  assume \(N \in \text{?A}\) and 
  \(?P \ N \ (\text{add-mset} \ a \ \text{M}0)\)
  then have 
  \((\exists \text{M}. \ \text{M} \in \text{?A} \land \ ?P \ \text{M} \ \text{M}0 \land N = \text{M} + \{\#a\}) \lor \n\((\exists K. (\forall b. b \in\# K \longrightarrow P b \ a) \land N = \text{M}0 + K)\)\)
  using mulex1-add [of \(P \ N \ \text{M}0 \ a\)] by (auto simp: multisets-def)
then show \(?\text{acc} (N)\)
proof (elim exE disjE conjE)
  fix \(M\)
  assume \(M \in \text{?A}\) and \(?P M \ ?P M0\) and 
  \(N: N = \text{M}0 + \{\#a\}\)
  from \(\text{acc-hyp}\) have 
  \(?\text{acc} M \ ?P M \ ?P M0\) \(\longrightarrow \ ?\text{acc} (M + \{\#a\})\) ..
  with \(?M \in \text{?A}\) and \(?P M M0\) have \(?\text{acc} (M + \{\#a\})\) by blast
  moreover from \(N\) and \(\{\#a\}\) have \(K \in \text{?A}\) by (auto simp: multisets-def)
  ultimately have \(?\text{acc} (M0 + K)\)
  proof (induct \(K\))
    case empty
    from \(\text{M}0\) show \(?\text{acc} (M0 + \{\#\})\) by simp
next
  case (add \(x\) \(K\))
  from \(\text{add-prems}\) have \(x \in A\) and \(P x a\) by (auto simp: multisets-def)
  with \(?\text{hypo}\) have \(?\text{acc} M \ ?\text{acc} (M + \{\#x\})\) by blast
  moreover from \(\text{add}\) have \(?\text{acc} (M0 + K)\) by (auto simp: multisets-def)
  ultimately show \(?\text{acc} (M0 + (\text{add-mset} x K))\) by simp
  qed
  then show \(?\text{acc} N\) by (simp only: \(N\))
  qed
  qed
\}
note tedious-reasoning = this

fix \(M\)
assume \(M \in \text{?A}\)
then show \(?\text{acc} \ ?\text{acc} \ M\)
proof (induct \(M\))
  show \(?\text{acc} \{\#\}\)
  proof (rule accessible-onI)
    show \(\{\#\} \in \text{?A}\) by (auto simp: multisets-def)
  next
    fix \(b\) assume \(?P b \ ?\{\#\}\) then show \(?\text{acc} b\) by simp
  qed
  next
  case (add \(a\) \(M\))
  then have \(?\text{acc} M\) by (auto simp: multisets-def)
  from \(\text{add}\) have \(a \in A\) by (auto simp: multisets-def)
with \( w_f \) have \( \forall M. \, \text{?acc} \ M \rightarrow \text{?acc} \ (\text{add-mset} \ a \ M) \)
proof (induct)
case (less a)
then have \( r : \bigvee b. \ [b \in A; \ P \ b \ a] \implies (\forall M. \, \text{?acc} \ M \rightarrow \text{?acc} \ (M + \{ \#b\# \})) \)
by auto
show \( \forall M. \, \text{?acc} \ M \rightarrow \text{?acc} \ (\text{add-mset} \ a \ M) \)
proof (intro allI impI)
fix \( M' \)
assume \( \text{?acc} \ M' \)
moreover then have \( M' \in \ ?A \) by (blast dest: accessible-on-imp-mem)
ultimately show \( \text{?acc} \ (\text{add-mset} \ a \ M') \)
by (induct) (rule tedious-reasoning \[ \OF - (a \in A \cdot - r), \ auto \])
qed
qed
with \( \langle \text{?acc} \ (M) \rangle \) show \( \text{?acc} \ (\text{add-mset} \ a \ M) \) by blast
qed
qed

lemmas \( wfp-on-mulex1-multisets = \)
accessible-on-mulex1-multisets [THEN accessible-on-imp-wfp-on]

lemmas \( irreflp-on-mulex1 = \)
wfp-on-mulex1-multisets [THEN wfp-on-imp-irreflp-on]

lemma \( wfp-on-mulex-on-multisets: \)
assumes \( wfp-on \ P \ A \)
shows \( wfp-on \ (\text{mulex-on} \ P \ A) \ (\text{multisets} \ A) \)
using \( wfp-on-mulex1-multisets \ [\OF \ assms] \)
by (simp only: mulex-on-def wfp-on-restrict-to-tranclp-wfp-on-conv)

lemmas \( irreflp-on-mulex-on = \)
wfp-on-mulex-on-multisets [THEN wfp-on-imp-irreflp-on]

lemma \( mulex1-union: \)
mulex1 \( P \ M \ N \Rightarrow \ mulex1 \ P \ (K + M) \ (K + N) \)
by (auto simp: mulex1-def mult1-union)

lemma \( mulex-on-union: \)
assumes \( mulex-on \ P \ A \ M \ N \) and \( K \in \text{multisets} \ A \)
shows \( mulex-on \ P \ A \ (K + M) \ (K + N) \)
using \( \text{assms} \)
proof (induct)
case (base \( M \ N \))
then have \( mulex1 \ P \ (K + M) \ (K + N) \) by (blast dest: mulex1-union)
moreover from base have \( (K + M) \in \text{multisets} \ A \)
and \( (K + N) \in \text{multisets} \ A \) by (auto simp: multisets-def)
ultimately have \( \text{restrict-to} \ (mulex1 \ P) \ (\text{multisets} \ A) \ (K + M) \ (K + N) \) by auto
then show \( ?case \) by (auto simp: mulex-on-def)

55
next

case (step L M N)
then have mulex1 P (K + M) (K + N) by (blast dest: mulex1-union)
moreover from step have (K + M) ∈ multisets A and (K + N) ∈ multisets A by blast+
ultimately have (restrict-to (mulex1 P) (multisets A))++ (K + M) (K + N)
by auto
moreover have mulex-on P A (K + L) (K + M) using step by blast
ultimately show case by (auto simp: mulex-on-def)
qed

lemma mulex-on-union':
assumes mulex-on P A M N and K ∈ multisets A
shows mulex-on P A (M + K) (N + K)
using mulex-on-union [OF assms] by (simp add: ac-simps)

lemma mulex-on-add-mset:
assumes mulex-on P A M N and m ∈ A
shows mulex-on P A (add-mset m M) (add-mset m N)
unfolding add-mset-add-single[of m M] add-mset-add-single[of m N]
apply (rule mulex-on-union)
using assms by auto

lemma union-mulex-on-mono:
mulex-on P F A C ==> mulex-on P F B D ==> mulex-on P F (A + B) (C + D)
by (metis mulex-on-multisetsD mulex-on-trans mulex-on-union mulex-on-union'

lemma mulex-on-add-mset':
assumes P m n and m ∈ A and n ∈ A and M ∈ multisets A
shows mulex-on P A (add-mset m M) (add-mset n M)
unfolding add-mset-add-single[of m M] add-mset-add-single[of n M]
apply (rule mulex-on-union)
using assms by (auto simp: mulex-on-def)

lemma mulex-on-add-mset-mono:
assumes P m n and m ∈ A and n ∈ A and mulex-on P A M N
shows mulex-on P A (add-mset m M) (add-mset n N)
unfolding add-mset-add-single[of m M] add-mset-add-single[of n N]
apply (rule union-mulex-on-mono)
using assms by (auto simp: mulex-on-def)

lemma union-mulex-on-mono1:
A ∈ multisets F ===> (mulex-on P F)''= A C ==> mulex-on P F B D ==> mulex-on P F (A + B) (C + D)
by (auto intro: union-mulex-on-mono mulex-on-anion)

lemma union-mulex-on-mono2:
B ∈ multisets F ===> mulex-on P F A C ==> (mulex-on P F)''= B D ==> mulex-on P F (A + B) (C + D)
lemma \textit{mult1-mono}:
assumes \( \forall x, y. [x \in A; y \in A; (x, y) \in R] \implies (x, y) \in S \)
and \( M \in \text{multisets } A \)
and \( N \in \text{multisets } A \)
and \( (M, N) \in \text{mult1 } R \)
shows \( (M, N) \in \text{mult1 } S \)
using assms unfolding mult1-def multisets-def
by auto (metis (full-types) subsetD)

lemma \textit{mulex1-mono}:
assumes \( \forall x, y. [x \in A; y \in A; P x y] \implies Q x y \)
and \( M \in \text{multisets } A \)
and \( N \in \text{multisets } A \)
and \( \text{mulex1 } P M N \)
shows \( \text{mulex1 } Q M N \)
using \textit{mult1-mono} \( \{ \text{of } A \{ (x, y). P x y \}\} \{ (x, y). Q x y \} M N \)
and assms unfolding mulex1-def by blast

lemma \textit{mulex-on-mono}:
assumes \( \ast \): \( \forall x, y. [x \in A; y \in A; P x y] \implies Q x y \)
and \( \text{mulex-on } P A M N \)
shows \( \text{mulex-on } Q A M N \)
proof -
let \( ?\text{rel} = \lambda P. (\text{restrict-to } (\text{mulex1 } P) (\text{multisets } A)) \)
from \( \text{mulex-on } P A M N \) have \( (?\text{rel } P)^++ M N \) by (simp add: mulex-on-def)
then have \( (?\text{rel } Q)^++ M N \)
proof (induct rule: tranclp.induct)
  case (r-into-trancl M N)
  then have \( M \in \text{multisets } A \) and \( N \in \text{multisets } A \) by auto
  from \( \text{mulex1-mono } \{OF * this\} \) and r-into-trancl
  show \( ?\text{case} \) by auto
next
  case (trancl-into-trancl L M N)
  then have \( M \in \text{multisets } A \) and \( N \in \text{multisets } A \) by auto
  from \( \text{mulex1-mono } \{OF * this\} \) and trancl-into-trancl
  have \( ?\text{rel } Q M N \) by auto
  with \( (?\text{rel } Q)^++ L M \) show \( ?\text{case} \) by (rule tranclp.trancl-into-trancl)
qed
then show \( ?\text{thesis} \) by (simp add: mulex-on-def)
qed

lemma \textit{mult1-reflcl}:
assumes \( (M, N) \in \text{mult1 } R \)
shows \( (M, N) \in \text{mult1 } (R^+) \)
using assms by (auto simp: mult1-def)

lemma \textit{mulex1-reflclp}:
assumes \textit{mulex1 }P M N \\
shows \textit{mulex1 } (P==) M N \\
using \textit{mulex1-mono }[\text{of UNIV } P P== M N, OF - - - assms] \\
by (auto simp: multisets-def) \\

\textbf{lemma} \textit{mulex-on-reflclp}: \\
assumes \textit{mulex-on }P A M N \\
shows \textit{mulex-on } (P==) A M N \\
using \textit{mulex-on-mono }[OF - assms, of P==] by auto \\

\textbf{lemma} \textit{surj-on-multisets-mset}: \\
\forall M \in \text{multisets } A. \exists xs \in \text{lists } A. M = \text{mset } xs \\
proof \\
\begin{align*}
\fix M \\
\assume M \in \text{multisets } A \\
\then \show \exists xs \in \text{lists } A. M = \text{mset } xs \\
\proof (induct M) \\
\case \text{empty} \show ?case by simp \\
\next \\
\case (add a M) \\
\then obtain \begin{align*}
xs & \where \begin{align*} xs & \in \text{lists } A \land M = \text{mset } xs \end{align*} \end{align*} by auto \\
\then have add-mset a M = \text{mset } (a \# xs) by simp \\
\moreover have a \# xs \in \text{lists } A \using \begin{align*} \begin{align*} xs & \in \text{lists } A \land \text{add } \end{align*} \end{align*} by auto \\
\ultimately show ?case by blast \\
qed \\
qed \\

\textbf{lemma} \textit{image-mset-lists }[simp]: \\
\begin{align*} 
\text{mset } \text{lists } A & = \text{multisets } A \\
\using \text{surj-on-multisets-mset }[\text{of } A] \\
\by \text{auto } (\text{metis mem-Collect-eq multisets-def set-mset-mset subsetI}) \\
\end{align*} \\

\textbf{lemma} \textit{multisets-UNIV }[simp]: \text{multisets } \text{UNIV } = \text{UNIV} \\
\by \text{ (metis image-mset-lists lists-UNIV surj-mset)} \\

\textbf{lemma} \textit{non-empty-multiset-induct }[\text{consumes } 1, \text{case-names } \text{singleton add}]: \\
assumes \begin{align*}
M & \not= \{}\#\{} \land \forall x. P \begin{align*} \{\# x\} \end{align*} \\
& \land \forall x. P M \Longrightarrow P \begin{align*} \text{add-mset } x \ M \end{align*} \\
\end{align*} \\
shows P M \\
\using \text{assms } by \text{ (induct } M \text{) auto} \\

\textbf{lemma} \textit{mulex-on-all-strict}: \\
assumes \begin{align*} 
X & \not= \{}\#\{} \\
& \land \forall y. P y \begin{align*} X \land P y \end{align*} \\
\end{align*} \\
shows \textit{mulex-on }PA Y X \\
\using \text{assms} \\
\begin{align*} \begin{align*}
\text{proof } \begin{align*} \begin{align*} \text{ (induction } X \text{ arbitrary; } Y \text{ rule: non-empty-multiset-induct) } \\
\end{align*} \end{align*} \\
\end{align*}
case (singleton x)
then have mulex P Y {#x#}
  unfolding mulex-def mult1-def
  by auto
with singleton show ?case (auto simp: mulex-on-def)
next
case (add x M)
let ?Y = {? y ∈ Y. ∃ x. x ∈ M ∧ P y x #}
let ?Z = Y − ?Y
have Y: Y = ?Z + ?Y by (subst multiset-eq_iff) auto
from ?Y ∈ multisets A have ?Y ∈ multisets A by (metis multiset-partition union-multisets iff)
moreover have ∀ y. y ∈ ?Y −→ (∃ x. x ∈ M ∧ P y x)
moreover have M ∈ multisets A using add by auto
ultimately have mulex-on P A ?Y M using add by blast
moreover have mulex-on P A ?Z {#x#}
  proof −
    have {#x#} = {#} + {#x#} by simp
    moreover have ?Z = {#} + ?Z by simp
    moreover have ∀ y. y ∈ ?Z −→ P y x
      using add.prems by (auto simp add: in-diff-count split: if_splits)
    ultimately have mulex P ?Z {#x#} unfolding mulex-def mult1-def by blast
  moreover have {#x#} ∈ multisets A using add.prems by auto
  moreover have ?Z ∈ multisets A
    using ?Y ∈ multisets A by (metis diff-union-cancelL multiset-partition union-multisets D)
  ultimately show ?thesis by (auto simp: mulex-on-def)
qed
ultimately have mulex-on P A (?Y + ?Z) (M + {#x#}) by (rule union-mulex-on-mono)
then show ?case using Y by (simp add: ac-simps)
qed

The following lemma shows that the textbook definition (e.g., “Term Rewriting
and All That”) is the same as the one used below.

lemma diff-set-Ex_iff:
  X ≠ {#} ∧ X ⊆# M ∧ N = (M − X) + Y −→ X ≠ {#} ∧ (∃ Z. M = Z + X ∧ N = Z + Y)
  by (auto) (metis add-diff-cancel-left multisets-diff-union-assoc union-commute)

Show that mulex-on is equivalent to the textbook definition of multiset-
extension for transitive base orders.

lemma mulex-on-alt-def:
  assumes trans: transp-on P A
  shows mulex-on P A M N −→ M ∈ multisets A ∧ N ∈ multisets A ∧ (∃ X Y Z.
    X ≠ {#} ∧ N = Z + X ∧ M = Z + Y ∧ (∋ y. y ∈# Y −→ (∃ x. x ∈# X ∧ P y x))
    (is ?P M N −→ ?Q M N)
proof
  assume \(?P\ M\ N\) then show \(?Q\ M\ N\)
  proof (induct \(M\ N\))
    case (base \(M\ N\))
    then obtain \(a\ M0\ K\) where \(N = M0 + \{\#a\}\)
    and \(M = M0 + K\)
    and \(\forall b\ b\ \# K \rightarrow P\ b\ a\)
    and \(M \in\) multisets \(A\) and \(N \in\) multisets \(A\) by (auto simp: mulex1-def mult1-def)
    moreover then have \(\{\#a\}\) \(\in\) multisets \(A\) and \(K\) \(\in\) multisets \(A\) by auto
    moreover have \(\{\#a\}\) \(\neq\) \(\{\#\}\) by auto
    moreover have \(N = M0 + \{\#a\}\) by fact
    moreover have \(M = M0 + K\) by fact
    moreover have \(\forall y\ y\ \# K \rightarrow (\exists x\ x\ \# (\{\#a\}\) \(\wedge P\ y\ x))\) using \(\ast\) by auto
    ultimately show \(?case\ by\ blast\)
  next
  case (step \(L\ M\ N\))
  then obtain \(X\ Y\ Z\) where \(L \in\) multisets \(A\) and \(M \in\) multisets \(A\) and \(N \in\) multisets \(A\)
  and \(X \in\) multisets \(A\) and \(Y \in\) multisets \(A\)
  and \(M: M = Z + X\)
  and \(L: L = Z + Y\) and \(X \neq\ \{\#\}\)
  and \(Y: \forall y\ y\ \# Y \rightarrow (\exists x\ x\ \# (\{\#a\}\) \(\wedge P\ y\ x))\)
  and \(mulex1\ P\ M\ N\)
  by blast
  from \(mulex1\ P\ M\ N\) obtain \(a\ M0\ K\)
  where \(N: N = add\-mset\ a\ M0\ and\ M': M = M0 + K\)
  and \(\ast: \forall b\ b\ \# K \rightarrow P\ b\ a\) unfolding mulex1-def mult1-def by blast
  have \(L': L = (M - X) + Y\) by (simp add: L M)
  have \(K: \forall y\ y\ \# K \rightarrow (\exists x\ x\ \# (\{\#a\}\) \(\wedge P\ y\ x))\) using \(\ast\) by auto

  The remainder of the proof is adapted from the proof of Lemma 2.5.4. of the book “Term Rewriting and All That.”

  let \(?X = add\-mset\ a\ (X - K)\)
  let \(?Y = (K - X) + Y\)

  have \(L \in\) multisets \(A\) and \(N \in\) multisets \(A\) by fact+
  moreover have \(?X \neq\ \{\#\}\) \(\land (\exists Z. N = Z + ?X \land L = Z + ?Y)\)
  proof
  have \(?X \neq\ \{\#\}\) by auto
  moreover have \(?X \subseteq\ \# N\)
  using \(M\ N\ M'\) by (simp add: commute [of \(\{\#a\}\)])[1] (metis Multiset.diff-subset-eq_self add.commute add-diff-cancel-right)
  moreover have \(L = (N - ?X) + ?Y\)
  proof (rule multiset-eqI)
  fix \(x\ ::\ 'a\)
  let \(?c = \lambda M.\ count\ M\ x\)
  let \(?ic = \lambda x.\ int\ (?c\ x)\)

  60
from (∃X ⊆# N) have *: ?c {#a#} + ?c (X - K) ≤ ?c N
  by (auto simp add: subseteq-mset-def split: if-splits)
from * have **: ?c (X - K) ≤ ?c M0 unfolding N by (auto split: if-splits)
  also have ... = int (?c N - (?c {#a#} + ?c (X - K))) + ?ic (K - X) + ?ic Y by simp
  also have ... = ?ic N - (?ic {#a#} + ?ic (X - K)) + ?ic (K - X) + ?ic Y
    using of-nat-diff [OF *] by simp
  also have ... = (?ic N - ?ic {#a#}) + (?ic (K - X) - ?ic (X - K)) + ?ic Y by simp
  also have ... = ?ic L
    unfolding L' M' N
    using ** by (simp add: algebra-simps)
  finally show ?c L = ?c (N - ?X + ?Y) by simp
qed

ultimately show ?thesis by (metis diff-set-Ex-iff)
qed

moreover have ∀ y. y ∈# ?Y → (∃ x. x ∈# ?X ∧ P y x)
proof (intro allI simp)
  fix y assume y ∈# ?Y
  then have y ∈# K - X ∨ y ∈# Y by auto
  then show ∃ x. x ∈# ?X ∧ P y x
    proof
      assume y ∈# K - X
      then have y ∈# K by (rule in-diffD)
      with K show ?thesis by auto
    next
      assume y ∈# Y
      with Y obtain x where x ∈# ?X ∧ P y x by blast
      { assume x ∈# X - K with :P y x have ?thesis by auto }
    moreover
      { assume x ∈# K with * have P x a by auto
        moreover have y ∈ A using ⟨Y ∈ multisets A⟩ and ⟨y ∈# Y⟩ by (auto simp: multisets-def)
        moreover have a ∈ A using ⟨N ∈ multisets A⟩ by (auto simp: N)
        moreover have x ∈ A using ⟨M ∈ multisets A⟩ and ⟨x ∈# K⟩ by (auto simp: M' multisets-def)
        ultimately have P y a using ⟨P y x⟩ and trans unfolding transp-on-def
      } by blast
      then have ?thesis by force
    moreover from ⟨x ∈# X⟩ have x ∈# X - K ∨ x ∈# K
      by (auto simp add: split: if-splits)
    ultimately show ?thesis by auto
qed
ultimately show ?case by blast
qed

next
assume ?Q M N
then obtain X Y Z where M ∈ multisets A and N ∈ multisets A
and X ≠ {#} and N: N = Z + X and M: M = Z + Y
and ∀ y. y ∈# Y → (∃ x. x ∈# X ∧ P y x) by blast
with mulex-on-all-strict [of X A Y] have mulex-on P A Y X by auto
moreover from (N ∈ multisets A) have Z ∈ multisets A by (auto simp: N)
ultimately show ?P M N unfolding M N by (metis mulex-on-union)
qed
end

12 Multiset Extension Preserves Well-Quasi-Orders

theory Wqo-Multiset
imports
  Multiset-Extension
  Well-Quasi-Orders
begin

lemma list-emb-imp-reflclp-mulex-on:
  assumes xs ∈ lists A and ys ∈ lists A
  and list-emb P xs ys
  shows (mulex-on P A)== (mset xs) (mset ys)
using assms(3, 1, 2)
proof (induct)
case (list-emb-Nil ys)
then show ?case
  by (cases ys) (auto intro!: empty-mulex-on simp: multisets-def)
next
case (list-emb-Cons xs ys y)
then show ?case by (auto intro!: mulex-on-self-add-singleton-right simp: multisets-def)
next
case (list-emb-Cons2 x y xs ys)
then show ?case
  by (force intro: union-mulex-on-mono mulex-on-add-mset
  mulex-on-add-mset' mulex-on-add-mset-mono
  simp: multisets-def)
qed

The (reflexive closure of the) multiset extension of an almost-full relation is almost-full.

lemma almost-full-on-multisets:
  assumes almost-full-on P A
  shows almost-full-on (mulex-on P A)== (multisets A)
proof –
let \( \mathcal{P} = (\text{mulex-on } P A) = \)
from almost-full-on-hom [OF - almost-full-on-lists, of \( A \) \( \mathcal{P} \) \( mset \),
                           OF list-emb-imp-reflcp-mulex-on, simplified]
show \( \text{thesis using assms by blast} \)
qed

lemma wqo-on-multisets:
assumes wqo-on \( P A \)
shows wqo-on (mulex-on \( P A \)) = (\text{multisets } A)
proof
from transp-on-mulex-on [of \( P A \) \text{multisets } A]
show transp-on (mulex-on \( P A \)) = (\text{multisets } A)
    unfolding transp-on-def by blast
next
from almost-full-on-multisets [OF assms [\( \text{THEN wqo-on-imp-almost-full-on} \)]]
show almost-full-on (mulex-on \( P A \)) = (\text{multisets } A) .
qed

end

References