Well-Quasi-Orders

Christian Sternagel*

February 23, 2021

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.
6 A Proof of Higman’s Lemma via Open Induction 23
6.1 Some facts about the suffix relation . . . . . . . . . . . . . . . 23
6.2 Lexicographic Order on Infinite Sequences . . . . . . . . . . 24

7 Almost-Full Relations 29
7.1 Adding a Bottom Element to a Set . . . . . . . . . . . . . . . 29
7.2 Adding a Bottom Element to an Almost-Full Set . . . . . . . 30
7.3 Disjoint Union of Almost-Full Sets . . . . . . . . . . . . . . . 30
7.4 Dickson’s Lemma for Almost-Full Relations . . . . . . . . . . 32
7.5 Higman’s Lemma for Almost-Full Relations . . . . . . . . . . 32
7.6 Natural Numbers . . . . . . . . . . . . . . . . . . . . . . . . . 34

8 Well-Quasi-Orders 34
8.1 Basic Definitions . . . . . . . . . . . . . . . . . . . . . . . . . 35
8.2 Equivalent Definitions . . . . . . . . . . . . . . . . . . . . . . 35
8.3 A Type Class for Well-Quasi-Orders . . . . . . . . . . . . . . 37
8.4 Dickson’s Lemma . . . . . . . . . . . . . . . . . . . . . . . . . 38
8.5 Higman’s Lemma . . . . . . . . . . . . . . . . . . . . . . . . . 39

9 Kruskal’s Tree Theorem 41

10 Instances of Well-Quasi-Orders 47
10.1 The Option Type is Well-Quasi-Ordered . . . . . . . . . . . . 48
10.2 The Sum Type is Well-Quasi-Ordered . . . . . . . . . . . . . 48
10.3 Pairs are Well-Quasi-Ordered . . . . . . . . . . . . . . . . . . 48
10.4 Lists are Well-Quasi-Ordered . . . . . . . . . . . . . . . . . . 48

11 Multiset Extension of Orders (as Binary Predicates) 49

12 Multiset Extension Preserves Well-Quasi-Orders 62

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 $SEQ A = \{f::\text{nat} \Rightarrow 'a. \forall i. f\ i \in A\}$

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** $ith B i = \{ f i | f, f \in B \}$

**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: 
$x \in ith B i \iff (\exists f \in B. x = f i)$
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 :: $(nat \Rightarrow 'a) set \Rightarrow (nat \Rightarrow 'a) set$
where 
$eq$-upto $B f i = \{ g \in B. \forall j < i. f j = g j \}$

**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 $B f i; g i = f i \implies f j = g j ] \implies Q \implies Q$
by (auto simp: $eq$-upto-def)

**lemma** $eq$-upto-Suc: 
$[ g \in eq$-upto $B f i; g i = f i \implies f j = g j ] \implies g \in eq$-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** $LEX P f g \iff (\exists i::nat. P (f i) (g i) \land (\forall j < i. f j = g j))$

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

**lemma** $LEX$-imp-not-LEX:
assumes $LEX P f g$


and \( \text{[dest]} : \forall x y z. P x y \rightarrow P y z \rightarrow P x z \)
and \( \text{[simp]} : \forall x. \neg P x x \)
shows \( \neg LEX P g f \)
proof
\begin{itemize}
\item fix \( i j :: \text{nat} \)
\item assume \( P (f i) (g i) \) and \( \forall k < i. f k = g k \)
\item and \( P (g j) (f j) \) and \( \forall k < j. g k = f k \)
\item then have False by (cases \( i < j \)) (auto simp: not-less dest: le-imp-less-or-eq)
\end{itemize}
then show \( \neg LEX P g f \) using \( \langle LEX P f g \rangle \) unfolding LEX-def by blast qed

lemma LEX-cases:
assumes \( LEX P f g \)
obtains \( (eq) f = g \mid (neq) k \) where \( \forall i < k. f i = g i \) and \( P (f i) (g i) \)
using assms by (auto simp: LEX-def)

lemma LEX-imp-less:
assumes \( \forall x \in A. \neg P x x \) and \( f \in \text{SEQ A} \lor g \in \text{SEQ A} \)
\begin{itemize}
\item and \( LEX P f g \) and \( \forall i < k. f i = g i \) and \( f k \neq g k \)
\item shows \( P (f k) (g k) \)
\end{itemize}
using assms by (auto elim!: LEX-cases) (metis linorder-neqE-nat)+
end

2 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 \( uf: \text{wfp-on P A} \)
begin

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

lemma minimal:
assumes \( x \in A \) and \( Q x \)
shows \( \exists y \in A. P^\equiv y x \land Q y \land (\forall z \in A. P z y \rightarrow \neg Q z) \)
using \( uf \) and assms
proof (induction rule: wfp-on-induct)
\begin{itemize}
\item case \( (\text{less } x) \)
\end{itemize}
A lexicographically minimal sequence w.r.t. a given set of sequences $C$
then show \(?\)case by blast

qed

lemma lexmin-SEQ-mem:
  assumes \(C \subseteq \text{SEQ } A\) and \(C \neq \{\}\)
  shows \(\text{lexmin } C \in \text{SEQ } A\)
proof –
  \{ fix \(i\)
  let \(?X = \text{ith } (\text{eq-upto } C \text{ (lexmin } C\) i) i\)
  have \(?X \subseteq A\) using assms by (auto simp: ith-def)
  moreover have \(?X \neq \{\}\) using eq-upto-lexmin-non-empty [OF assms] by auto
  ultimately have \(\text{lexmin } C \in A\) using min-elt-mem [of \(?X\)] by (subst lexmin)
  blast \}
  then show \(?\)thesis by auto
qed

lemma non-empty-ith:
  assumes \(C \subseteq \text{SEQ } A\) and \(C \neq \{\}\)
  shows \(\text{ith } (\text{eq-upto } C \text{ (lexmin } C\) i) i \subseteq A\)
and \(\text{ith } (\text{eq-upto } C \text{ (lexmin } C\) i) i \neq \{\}\)
using eq-upto-lexmin-non-empty [OF assms, of \(i\)] and assms by (auto simp: ith-def)

lemma lexmin-minimal:
  \(C \subseteq \text{SEQ } A \Rightarrow C \neq \{\}\ \Rightarrow \ y \in A \Rightarrow \ P y \text{ (lexmin } C\) i) i \Rightarrow y \notin \text{ith } (\text{eq-upto } C \text{ (lexmin } C\) i) i\)
using min-elt-minimal [OF non-empty-ith, folded lexmin].

lemma lexmin-mem:
  \(C \subseteq \text{SEQ } A \Rightarrow C \neq \{\}\ \Rightarrow \text{lexmin } C \in \text{ith } (\text{eq-upto } C \text{ (lexmin } C\) i) i\)
using min-elt-mem [OF non-empty-ith, folded lexmin].

lemma LEX-chain-on-eq-upto-imp-ith-chain-on:
  assumes chain-on (LEX \(P\) \(\text{eq-upto } C \text{ f } i\) \(\text{SEQ } A\))
  shows chain-on \(P \text{ ith } (\text{eq-upto } C \text{ f } i\) i) \(A\)
using assms
proof –
  \{ fix \(x y\) assume \(x \in \text{ith } (\text{eq-upto } C \text{ f } i\) i\) and \(y \in \text{ith } (\text{eq-upto } C \text{ f } i\) i\)
and \(\neg P x y\) and \(y \neq x\)
  then obtain \(g \ h\) where \(*: g \in \text{eq-upto } C \text{ f } i \ h \in \text{eq-upto } C \text{ f } i\)
  and \([\text{simpl}: x = g \ i \ y = h \ i\) and \(\text{eq } \forall j<i. g \ j = f \ j \ \wedge \ h \ j = f \ j\)
  by (auto simp: ith-def eq-upto-def)
  with assms and \((y \neq x)\) consider LEX \(P \ g \ h\) | LEX \(P \ h \ g\) by (force simp:
chain-on-def)
  then have \(P y x\)
  proof (cases)
   assume LEX \(P \ g \ h\)
   with eq and \((y \neq x)\) have \(P x y\) using assms and *
   by (auto simp: 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
\begin{align*}
\text{and } N :: 'a \\
\text{assumes } \infm: \forall i \geq N. \exists j > i. P \ i \ j \\
\text{begin} \\
\text{Enumerate the elements of a well-ordered infinite set that form a chain w.r.t.} \\
a \text{given predicate } P \text{ starting from a given index } N \text{ in increasing order.} \\
\text{fun enumchain :: } \text{nat } \Rightarrow \ 'a \ \text{where} \\
\quad \text{enumchain } 0 = N \\
\quad \text{enumchain } (\text{Suc } n) = (\text{LEAST } m. m > \text{enumchain } n \land P \ (\text{enumchain } n) \ m) \\
\text{lemma enumchain-mono:} \\
\text{shows } N \leq \text{enumchain } i \land \text{enumchain } i < \text{enumchain } (\text{Suc } i) \\
\text{proof (induct } i) \\
\text{case } 0 \\
\text{have enumchain } 0 \geq N \text{ by simp} \\
\text{moreover then have } \exists m > \text{enumchain } 0. P \ (\text{enumchain } 0) \ m \text{ using } \infm \text{ by blast} \\
\text{ultimately show } ?\text{case by auto (metis (lifting) LeastI)} \\
\text{next} \\
\text{case } (\text{Suc } i) \\
\text{then have } N \leq \text{enumchain } (\text{Suc } i) \text{ by auto} \\
\text{moreover then have } \exists m > \text{enumchain } (\text{Suc } i). P \ (\text{enumchain } (\text{Suc } i)) \ m \text{ using } \infm \text{ by blast} \\
\text{ultimately show } ?\text{case by (auto) (metis (lifting) LeastI)} \\
\text{qed} \\
\text{lemma enumchain-chain:} \\
\text{shows } P \ (\text{enumchain } i) \ (\text{enumchain } (\text{Suc } i)) \\
\text{proof (cases } i) \\
\text{case } 0 \\
\text{moreover have } \exists m > \text{enumchain } 0. P \ (\text{enumchain } 0) \ m \text{ using } \infm \text{ by auto} \\
\text{ultimately show } ?\text{thesis by auto (metis (lifting) LeastI)} \\
\text{next} \\
\text{case } (\text{Suc } i) \\
\text{moreover have } \text{enumchain } (\text{Suc } i) > N \text{ using } \text{enumchain-mono by (metis le-less-trans)} \\
\text{moreover then have } \exists m > \text{enumchain } (\text{Suc } i). P \ (\text{enumchain } (\text{Suc } i)) \ m \text{ using } \infm \text{ by auto} \\
\text{ultimately show } ?\text{thesis by (auto) (metis (lifting) LeastI)} \\
\text{qed} \\
\text{end} \\
\text{end} \\
\end{align*}

\section{4 The Almost-Full Property}

theory Almost-Full

imports
lemma le-Suc-eq':
\[ x \leq \text{Suc } y \iff x = 0 \lor (\exists x'. x = \text{Suc } x' \land x' \leq y) \]
by (cases x) auto

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

lemma ex-less-Suc:
\[ (\exists i < \text{Suc } j. P i) \iff P 0 \lor (\exists i < j. P (\text{Suc } i)) \]
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
\[ \text{good } P f \iff (\exists i j. i < j \land P (f i) (f j)) \]

A sequence that is not good is called *bad*.

abbreviation bad P f ≡ ¬ good P f

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

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

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

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

**Lemma** \( \text{almost-full-onI} \) [Pure.intro]:

\( (\forall f. \forall i. f i \in A \implies \text{good } P f) \implies \text{almost-full-on } PA \)

Unfolding almost-full-on-def by blast

**Lemma** \( \text{almost-full-onD} \):

fixes \( f :: nat \rightarrow 'a \) and \( A :: 'a \set \)

assumes almost-full-on \( PA \) and \( \forall i. f i \in A \)

obtains \( i j \) where \( i < j \) and \( P (f i) (f j) \)

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

### 4.2 An equivalent inductive definition

**Inductive af for A**

where

now: \( (\forall x y. x \in A \implies y \in A \implies P x y) \implies af A P \)

| later: \( (\forall x. x \in A \implies af A (\lambda y z. P y z \lor P x y)) \implies af A P \)

**Lemma** \( \text{af-imp-almost-full-on} \):

assumes \( \text{af } A P \)

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

proof

fix \( f :: nat \rightarrow 'a \) assume \( \forall i. f i \in A \)

with assms obtain \( i j \) where \( i < j \) and \( P (f i) (f j) \)

proof (induct arbitrary: \( f \) thesis)

| case (later \( P \))

\| define \( g \) where \( g i = f \) \( (\text{Suc } i) \) for \( i \)

\| have \( f 0 \in A \) and \( \forall i. g i \in A \) using later by auto

\| then obtain \( i j \) where \( i < j \) and \( P (g i) (g j) \lor P (f 0) (g i) \) using later by blast

\| then consider \( P (g i) (g j) \mid P (f 0) (g i) \) by blast

\| then show \( ?\)case using \( \langle i < j \rangle \) by (cases) (auto intro: later)

qed blast

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

qed

**Lemma** \( \text{af-mono} \):

assumes \( \text{af } A P \)

and \( \forall x y. x \in A \wedge y \in A \wedge P x y \implies Q x y \)

shows \( \text{af } A Q \)

using assms

proof (induct arbitrary: \( Q \))

| case (now \( P \))

\| then have \( \forall x y. x \in A \implies y \in A \implies Q x y \) by blast

\| then show \( ?\)case by (rule af.now)

next

| case (later \( P \))

\| show \( ?\)case
proof (intro af later [of A Q])
fix x assume x ∈ A
then show af A (λ y z. Q y z ∨ Q x y)
  using later(3) by (intro later(2) [of x]) auto
qed
qed

lemma accessible-on-imp-af:
  assumes accessible-on P A x
  shows af A (λ u v. ¬ P v u ∨ ¬ P u x)
  using assms
proof (induct)
case (1 x)
then have af A (λ u v. (¬ P v u ∨ ¬ P u x) ∨ ¬ P u y ∨ ¬ P y x) if y ∈ A for y
  using that by (cases P y x) (auto intro: af.now af-mono)
then show ?case by (rule af.later)
qed

lemma wfp-on-imp-af:
  assumes wfp-on P A
  shows af A (λ x y. ¬ P y x)
  using assms by (auto simp: wfp-on-accessible-on-iff intro: accessible-on-imp-af af.later)

lemma af-leq:
  af UNIV ((≤) :: nat ⇒ nat ⇒ bool)
  using wf-less [folded afP-def wfp-on-UNIV, THEN wfp-on-imp-af] by (simp add: not-less)

definition NOTAF A P = (SOME x. x ∈ A ∧ ¬ af A (λ y z. P y z ∨ P x y))

lemma not-af:
  ¬ af A P ⊢ (∃ x y. x ∈ A ∧ y ∈ A ∧ ¬ P x y) ∨ (∃ x ∈ A. ¬ af A (λ y z. P y z ∨ P x y))
  unfolding af.simps [of A P] by blast

fun F
where
  F A P 0 = NOTAF A P
| F A P (Suc i) = (let x = NOTAF A P in F A (λ y z. P y z ∨ P x y) i)

lemma almost-full-on-imp-af:
  assumes af: almost-full-on P A
  shows af A P
proof (rule ccontr)
assume ¬ af A P
then have ∗: F A P n ∈ A ∧ ¬ af A (λ y z. P y z ∨ (∃ i≤n. P (F A P i) y) ∨ (∃ j≤n. ∃ i. i < j ∧ P (F A P i) (F A P j))) for n
proof (induct n arbitrary: P)
case 0
  from (¬ af A P) have ∃ x. x ∈ A ∧ ¬ af A (λy z. P y z ∨ P x y) by (auto intro: af.intros)
  then have NOTAF A P ∈ A ∧ ¬ af A (λy z. P y z ∨ P (NOTAF A P) y)
unfolding NOTAF-def by (rule someI-ex)
  with 0 show ?case by simp
next
case (Suc n)
  from (¬ af A P) have ∃ x. x ∈ A ∧ ¬ af A (λy z. P y z ∨ P x y) by (auto intro: af.intros)
  then have NOTAF A P ∈ A ∧ ¬ af A (λy z. P y z ∨ P (NOTAF A P) y)
unfolding NOTAF-def by (rule someI-ex)
  from Suc(1) [OF this [THEN conjunct2]]
  show ?case
    by (fastforce simp: ex-leq-Suc ex-less-Suc elim: back-subst [where P = λx. ¬ af A x])
qed

hide-const NOTAF F

lemma almost-full-on-UNIV:
  almost-full-on (λ- _. 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 ⊆ B ⇒ almost-full-on P B ⇒ almost-full-on P A
by (auto simp: almost-full-on-def)

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

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

lemma almost-full-on-imp-homogeneous-subseq:
  assumes almost-full-on P A
and \( \forall i :: \text{nat}. \ f \ i \in A \)
shows \( \exists \varphi :: \text{nat}. \ \forall i \ j. \ i < j \rightarrow \varphi \ i < \varphi \ j \land P (f (\varphi \ i)) (f (\varphi \ j)) \)
proof
- 
  define \( X \) where \( X = \{ (i, j) \mid i :: \text{nat}. \ i < j \land P (f \ i) (f \ j) \} \)
  define \( Y \) where \( Y = \neg X \)
  define \( h \) where \( h = (\lambda Z. \text{if } Z \in X \text{ then } 0 \text{ else } Suc 0) \)
  have \( \text{iff} : \forall x. y. \ h \ {x, y} = 0 \longleftrightarrow \{x, y\} \in X \text{ by } (\text{auto simp: h-def}) \)
  have \( \text{iff} : \forall x. y. \ h \ {x, y} = Suc 0 \longleftrightarrow \{x, y\} \in Y \text{ by } (\text{auto simp: h-def Y-def}) \)

have \( \forall x \in UNIV. \forall y \in UNIV. \ x \neq y \rightarrow h \ {x, y} < 2 \text{ by } (\text{simp add: h-def}) \)
from Ramsey2 \( [\text{OF infinite-UNIV-nat this}] \) obtain \( I \ c \)
  where \( \text{infinite } I \text{ and } c < 2 \)
  and \( * : \forall x \in I. \forall y \in I. \ x \neq y \rightarrow h \ {x, y} = c \text{ by blast} \)
then interpret infinitely-many1 \( \lambda i. \ i \in I \)
  by \( (\text{unfold-locales}) \) \( (\text{simp add: infinite-nat-iff-unbounded}) \)

have \( c = 0 \lor c = 1 \text{ using } c < 2 \) by \( \text{arith} \)
then show \( \text{thesis} \)
proof
  assume \( \text{simp} : c = 0 \)
  have \( \forall i \ j. \ i < j \rightarrow P (f (\text{enum} \ i)) (f (\text{enum} \ j)) \)
  proof \( (\text{intro allII impI}) \)
    fix \( i \ j :: \text{nat} \)
    assume \( i < j \)
    from \( * \) and \( \text{enum-P and enum-less [OF } i < j \text{]} \) have \( \{\text{enum} \ i, \text{enum} \ j\} \in X \) by \( \text{auto} \)
    with \( \text{enum-less [OF } i < j \text{]} \)
    show \( P (f (\text{enum} \ i)) (f (\text{enum} \ j)) \) by \( (\text{auto simp: X-def doubleton-eq-iff}) \)
  qed
then show \( \text{thesis using enum-less by blast} \)
next
  assume \( \text{simp} : c = 1 \)
  have \( \forall i \ j. \ i < j \rightarrow \neg P (f (\text{enum} \ i)) (f (\text{enum} \ j)) \)
  proof \( (\text{intro allII impI}) \)
    fix \( i \ j :: \text{nat} \)
    assume \( i < j \)
    from \( * \) and \( \text{enum-P and enum-less [OF } i < j \text{]} \) have \( \{\text{enum} \ i, \text{enum} \ j\} \in Y \) by \( \text{auto} \)
    with \( \text{enum-less [OF } i < j \text{]} \)
    show \( \neg P (f (\text{enum} \ i)) (f (\text{enum} \ j)) \) by \( (\text{auto simp: Y-def doubleton-eq-iff}) \)
  qed
then have \( \neg \text{good } P (f \circ \text{enum}) \) by \( \text{auto} \)
moreover have \( \forall i. \ f (\text{enum} \ i) \in A \) using \( \text{assms by auto} \)
ultimately show \( \text{thesis using } (\text{almost-full-on } P \ A) \text{ by } (\text{simp add: almost-full-on-def}) \)
  qed
qed
Almost full relations do not admit infinite antichains.

**Lemma** almost-full-on-imp-no-antichain-on:
- **Assumes** almost-full-on P A
- **Shows** \( \neg \text{antichain-on} P f A \)

**Proof**
- Assume \( *: \text{antichain-on} P f A \)
- Then have \( \forall i. f i \in A \) by simp
- With assms have good P f by (auto simp: almost-full-on-def)
- Unfolding good-def by auto
- Moreover with \( * \) have incomparable \( P (f i) (f j) \) by auto
- Ultimately show False by blast

qed

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

**Lemma** almost-full-on-map:
- **Assumes** almost-full-on Q B and \( h ': A \subseteq B \)
- **Shows** almost-full-on \((\lambda x y. Q (h x) (h y)) A \) (is almost-full-on ?P A)

**Proof**
- Fix \( f \)
- Assume \( \forall i :: \text{nat}. f i \in A \)
- Then have \( \\land i. h (f i) \in B \) using \( h ': A \subseteq B \) by auto
- With \{almost-full-on Q B; \[unfolded almost-full-on-def, THEN bspec, of h \circ \] \}
- 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: \( \forall x y. [x \in A; y \in A; P x y] \Rightarrow Q (h x) (h y) \)
- **And** af: almost-full-on P A
- **Shows** almost-full-on Q \( h ' A \)

**Proof**
- Fix \( f :: \text{nat} \Rightarrow 'b \)
- Assume \( \forall i. f i \in h ' A \)
- Then have \( \forall i. \exists x. x \in A \land f i = h x \) by (auto simp: image-def)
- From choice [OF this] obtain \( g \)
  - Where \( *: \forall i. g i \in A \land f i = h (g i) \) by blast
- Show good \( Q f \)
- Proof (rule ccontr)
  - Assume bad: bad \( Q f \)
    - \{ fix \( i j :: \text{nat} \)
      - Assume \( i < j \)
        - From bad have \( \neg Q (f i) (f j) \) using \( i < j \) by (auto simp: good-def)
        - With hom have \( \neg P (g i) (g j) \) using \(* \) by auto \}
    - Then have \( \neg P g \) by (auto simp: good-def)
      - With of \( \) and \( * \) show False by (auto simp: good-def almost-full-on-def)

qed
The monomorphic preimage of an almost-full set is almost-full.

**Lemma** almost-full-on-mon:
- **Assumes** `mon: ∀x y. [x ∈ A; y ∈ A] ⇒ P x y = Q (h x) (h y)` (bij-betw h A B)
  - and `af: almost-full-on Q B`
- **Shows** `almost-full-on P A`

**Proof**
- Fix `f :: nat ⇒ 'a`
- Assume `∀i. f i ∈ A`
- Then have `∀i. (h ∘ f) i ∈ B` using `mon`
- Show `good P f`
  - Rule `ccontr`
  - Assume `¬ almost-full-on P A`
    - Then obtain `f :: nat ⇒ 'a` where `∀i. f i ∈ A` (bij-betw-def)
  - With `mon` have `∀i. (h ∘ f) i ∈ B` (bij-betw-def)
    - Using `i < j` by (auto simp: good-def)
  - With `af` and `¬` show `False` by (auto simp: good-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 `¬ almost-full-on P A`
  - Then obtain `f :: nat ⇒ 'a` where `∀i. f i ∈ A` (bij-betw-def)
  - With `mon` have `∀i. (h ∘ f) i ∈ B` (bij-betw-def)
    - Using `i < j` by (auto simp: good-def)
  - With `af` and `¬` show `False` by (auto simp: good-def)

**Qed**

**Lemma** Nil-imp-good-list-emb [simp]:
- **Assumes** `f i = []`
- **Shows** `good (list-emb P) f`

**Proof** (rule `ccontr`)
- Assume `bad (list-emb P) f`
  - Moreover have `¬ list-emb P A` (bij-betw-def)
    - Using `i < j` by (auto simp: good-def)
  - With `af` and `¬` show `False` by (auto simp: good-def)

**Qed**
lemma ne-lists:
assumes $xs \neq []$ and $xs \in \text{lists } A$
shows $\text{hd } xs \in A$ and $\text{tl } xs \in \text{lists } A$
using assms by (case-tac [] $xs$) simp-all

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

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

4.3 Special Case: Finite Sets

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

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

qed

**Lemma** \( \text{eq-almost-full-on-finite-set} \):
- **Assumes** finite \( A \)
- **Shows** almost-full-on \( (=) A \)
- **Using** finite-almost-full-on \( \text{OF assms, of } (=) \)
- **By** (auto simp: reflp-on-def)

### 4.4 Further Results

**Lemma** \( \text{af-trans-extension-imp-wf} \):
- **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 \( \text{strict } Q A \)

**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 \( \forall i. f i \in A \land ((\text{strict } Q)^{\sim-1}) (f i) (f (\text{Suc } i)) \) by blast
- from chain-transp-on-less \( \text{OF this} \)
- and transp-on-strict \( \text{THEN transp-on-converse, OF trans} \)
- **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** \( \text{af-trans-imp-wf} \):
- **Assumes** almost-full-on \( P A \)
- and transp-on \( P A \)
- **Shows** wfp-on \( \text{strict } P A \)
- **Using** assms by (intro af-trans-extension-imp-wf)

**Lemma** \( \text{wf-and-no-antichain-imp-qo-extension-wf} \):
- **Assumes** wf: wfp-on \( \text{strict } P A \)
- 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 \( \text{strict } Q A \)

**Proof** (rule ccontr)
- **Have** transp-on \( \text{strict } Q A \)
- using go unfolding go-on-def transp-on-def by blast
- **Then have** \( * \) transp-on \( ((\text{strict } Q)^{\sim-1}) A \) by (rule transp-on-converse)
- assume \( \neg \text{wpf-on } (\text{strict } Q) A \)
- **Then obtain** \( f :: \text{nat} \Rightarrow 'a \) 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** \( \forall i. f i \in A \land ((\text{strict } Q)^{\sim-1}) (f i) (f (\text{Suc } i)) \) by auto
- from chain-transp-on-less \( \text{OF this } * \)
- **Have** \( \forall i j. i < j \Rightarrow \neg P (f i) (f j) \)
proof (cases)
  assume \( \exists k. \forall i > k. \exists j > i. \ P (f j) (f i) \)
  then obtain \& where \( \forall i > k. \exists j > i. \ P (f j) (f i) \) by auto
from subchain [of \( k - f \), OF this] obtain \( g \)
  where \( \forall i. j. \ i < j \Rightarrow g i < g j \)
  and \( \forall i. P (g (Suc i)) (f (g i)) \) by auto
with * have \( \forall i. \ P (f (g (Suc i))) (f (g i)) \) by blast
with wf [unfolded wfp-on-def not-ex, THEN spec, of \( \lambda i. \ f (g i) \)] and \( A \)
  show False by fast
next
  assume \( \neg (\exists k. \forall i > k. \exists j > i. \ P (f j) (f i)) \)
  then have \( \forall k. \exists i > k. \forall j > i. \neg P (f j) (f i) \) 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: \( \varphi = (\lambda i. (h \, ^{\sim} \, Suc i) \, 0) \)
have \( \forall i. \varphi i < \varphi (Suc i) \)
  using \( \forall k. h k > k \) by (induct-tac \( i \)) auto
then have mon: \( \forall i. j. \ i < j \Rightarrow \varphi i < \varphi j \) by (metis lift-Suc-mono-less)
then have \( \forall i. j. \ i < j \Rightarrow \neg P (f (\varphi j)) (f (\varphi i)) \)
  using ** by auto
with mono [THEN *]
  have \( \forall i. j. \ i < j \Rightarrow \neg \text{inc}\text{omp} P (f (\varphi j)) (f (\varphi i)) \) by blast
moreover have \( \exists i. j. \ i < j \land \neg \text{inc}\text{omp} P (f (\varphi j)) (f (\varphi i)) \)
  using anti [unfolded not-ex, THEN spec, of \( \lambda i. \ f (\varphi i) \)] and \( A \) by blast
ultimately show False by blast
qed

lemma every-qo-extension-wf-imp-af:
  assumes ext: \( \forall Q. \ (\forall x \in A. \forall y \in A. \ P x y \Rightarrow Q x y) \land \)
  \( \text{qo-on} \ Q A \Rightarrow \text{wf-on} \ (\text{strict} \ Q) \ A \)
  and \( \text{qo-on} \ P A \)
  shows \( \text{almost-full-on} \ P A \)
proof
from (qo-on P A)
  have refl: reflp-on P A
  and trans: transp-on P A
  by (auto intro: qo-on-imp-reflp-on qo-on-imp-transp-on)

fix \( f :: \text{nat} \Rightarrow 'a \)
assume \( \forall i. f i \in A \)
then have \( A: \land i. f i \in A \) ..
  show \( \text{good} \ P f \)
proof (rule ccontr)
  assume \( \neg \text{thesis} \)
  then have \( \text{bad: } \forall i. j. \ i < j \Rightarrow \neg P (f i) (f j) \) 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)

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

define \( P' \) where \( P' = restrict-to P A \)

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

have \( \forall i j. (D OO P') (f i) (f j) \implies i > j \)

proof -
  fix \( i j \)
  assume \( (D OO P') (f i) (f j) \)
  then show \( i > j \)
    apply (induct \( f i f j \) arbitrary: \( j \))
    apply (insert \( A \), auto dest: \( \ast \))
    apply (metis \( \ast \) dual-order.strict-trans1 less-Suc-eq-le refl reflp-on-def)
    by (metis le-imp-less-Suc less-trans)

  ultimately have \( \sup (P') (\sup P') (f i) (f (Suc i)) \)
    by simp
  then have \( (P') (\sup P') (f i) (f (Suc i)) \)
    by auto
  then have \( Suc i < i \)
    using \( \ast \) apply auto
    by (metis (lifting, mono_tags) less-le recomp.pp.recompI tranclp-into-tranclp2)
  then show \( False \) by auto

  with \( A \) [of \( i \)] show \( f i \in A \land strict Q (f (Suc i)) (f i) \)
  then have \( False \) unfolding wfp-on-def by blast

  qed

  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. $\text{size}$ of an element.

locale mbs =
  fixes $A$ :: (\'a :: $size$) set
begin

Since the $\text{size}$ 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 $\exists y \in A. \text{size} \; y \leq \text{size} \; x \land (\forall z \in A. \text{size} \; z < \text{size} \; y \longrightarrow \neg P \; z)$
using assms
proof (induction $x$ taking: $\text{size}$ rule: measure-induct)
  case (1 $x$)
  then show $?$ case
  proof (cases $\forall y \in A. \text{size} \; y < \text{size} \; x \longrightarrow \neg P \; y$)
    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]:
  $x \in A \Longrightarrow \text{size} \; x < \text{size} \; y \Longrightarrow x = y \Longrightarrow \text{False}$
by simp

The set of all bad sequences over $A$.

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

lemma $\text{BAD}-\text{iff}$ [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 geseq :: ((nat $\Rightarrow$ \'a) $\times$ (nat $\Rightarrow$ \'a)) set
where
geseq =
\{(f, g). f \in \text{SEQ } A \land g \in \text{SEQ } A \land (f = g \lor (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j)))\}\}

The strict part of the above order.

**definition** gseq ::= ((\text{nat} \Rightarrow 'a) \times (\text{nat} \Rightarrow 'a)) \text{ set where}

\[
gseq = \{(f, g). f \in \text{SEQ } A \land g \in \text{SEQ } A \land (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j))\}\]

**lemma** gseq-iff:

\((f, g) \in gseq \iff f \in \text{SEQ } A \land g \in \text{SEQ } A \land (f = g \lor (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j)))\)

**lemma** gseq-iff:

\((f, g) \in gseq \iff f \in \text{SEQ } A \land g \in \text{SEQ } A \land (\exists i. \text{size } (g i) < \text{size } (f i) \land (\forall j < i. f j = g j))\)

**lemma** geseqE:

assumes \((f, g) \in geseq \land (\forall i. f i \in A; \forall i. g i \in A; f = g) \Rightarrow Q\)

and \(\land i. [\forall i. f i \in A; \forall i. g i \in A; \text{size } (g i) < \text{size } (f i); \forall j < i. f j = g j] \Rightarrow Q\)

shows \(Q\)

using **assms** by (auto simp: geseq-iff)

**lemma** geseqE:

assumes \((f, g) \in geseq \land (\forall i. f i \in A; \forall i. g i \in A; \text{size } (g i) < \text{size } (f i); \forall j < i. f j = g j) \Rightarrow Q\)

shows \(Q\)

using **assms** by (auto simp: geseq-iff)

**sublocale** min-elt-size?: minimal-element measure-on size UNIV A

rewrites measure-on size UNIV \(\equiv \lambda x y. \text{size } x < \text{size } y\)

apply (unfold-locales)

apply (auto simp: po-on-def irreflp-on-def transp-on-def simp del: wfp-on-UNIV.intro: wfp-on-subset)

apply (auto simp: measure-on-def inv-image-betw-def)

done

context

fixes \text{P} :: 'a \Rightarrow 'a \Rightarrow bool

begin

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

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

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

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

begin

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

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

lemma non-empty-ith:
shows $\text{ith} (\text{eq-upto} (\text{BAD } P) \ lb \ i) \ i \subseteq A$
and $\text{ith} (\text{eq-upto} (\text{BAD } P) \ lb \ i) \ i \neq \{\}$
using eq-upto-BAD-non-empty [of $i$] by auto

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

$lb$ is an infinite bad sequence.

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

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

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

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

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

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

There is a minimal bad sequence.

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

end

end

end

end

6 A Proof of Higman’s Lemma via Open Induction

theory \( \text{Higman-OI} \)
imports
  \( \text{Open-Induction, Open-Induction} \)
  \( \text{Minimal-Elements} \)
  \( \text{Almost-Full} \)
begin

6.1 Some facts about the suffix relation

lemma \( \text{wfp-on-strict-suffix}: \)
\( \text{wfp-on strict-suffix A} \)
by \( \text{(rule wfp-on-mono \{OF subset-refl, of - - measure-on length A\})} \)
(\( \text{(auto simp: strict-suffix-def suffix-def)} \))

lemma \( \text{po-on-strict-suffix}: \)
po-on strict-suffix A
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 ∈ SEQ A g ∈ 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 ∀k<i. f k = g k and ∀k<j. g k = f k by (auto simp: LEX-def)
then have P (f (min i j)) (f (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 ∈ SEQ A and g ∈ SEQ A and h ∈ 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 ⇒ 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 ≠ {}
**shows** glb (LEX P) C (lexmin C)
**proof**
have C ⊆ SEQ A using assms by (auto simp: chain-on-def)
then have lexmin C ∈ SEQ A using C ≠ {} by (intro lexmin-SEQ-mem)
note * = ℐ C ⊆ SEQ A ; C ≠ {} ;
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 ∈ C
then show LEXEQ P (lexmin C) f
**proof** (cases f = lexmin C)
  define i where i = (LEAST i. f i ≠ lexmin C i)
case False
then have neq: ∃ i. f i ≠ lexmin C i by blast
from LeastI-ex [OF this, folded i-def]
and not-less-Least [where \( P = \lambda i. f \neq \operatorname{lexmin} C i \), folded i-def]

have neq: \( f \neq \operatorname{lexmin} C i \) and eq: \( \forall j<i. f j = \operatorname{lexmin} C j \) by auto

then have \(*\): \( f \in \text{eq-upto} C (\operatorname{lexmin} C) \) if \( f \in \text{ith} (\text{eq-upto} C (\operatorname{lexmin} C) i) \) i

using \( f \in C \) by force+

moreover from \(*\) have \( \neg P (f i) (\operatorname{lexmin} C i) \)

using \( \operatorname{lexmin}\)-minimal [OF \(*\), of \( f i \)] and \( f \in C \) and \( \{C \subseteq \text{SEQ} A\} \)

by blast

moreover obtain \( g \) where \( g \in \text{eq-upto} C (\operatorname{lexmin} C) (\operatorname{Suc} i) \)

using \( \text{eq-upto-xm-min} \)-non-empty [OF \(*\)] by blast

ultimately have \( P (\operatorname{lexmin} C i) (f i) \)

using \( \text{neq} \) and \( \{C \subseteq \text{SEQ} A\} \) and \( \text{assms}(1) \) and \( \text{lex} \) [of \( g f i \)] and \( \text{lex} \) [of \( f g i \)]

by (auto simp: \( \text{eq-upto-def} \) \( \text{chain-on-def} \))

with eq show \(?\)thesis by (auto simp: \( \text{LEX-def} \))

qed simp

qed

— \( \operatorname{lexmin} C \) is greater than or equal to any other lower bound

fix \( f \) assume \( \text{lb}: \text{lb} (\text{LEX} P) C f \)

then show \( \text{LEXEQ} P f (\text{lexmin} C) \)

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 neq: \( f \neq \text{lexmin} C i \) 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 \( \text{eq-upto-xm-min} \)-non-empty [OF \(*\)] by (auto simp: \( \text{eq-upto-def} \))

then have \( \text{simp}: \forall j. j < \text{Suc} i \imp h j = \text{lexmin} C j \) by auto

with \( \text{lb} \) and \( h \in C \) have \( \text{LEX} P f h \) using \( \text{neq} \) by (auto simp: \( \text{lb-def} \))

then have \( P (f i) (h i) \)

using \( \text{neq} \) and \( \text{eq} \) and \( \{C \subseteq \text{SEQ} A\} \) and \( h \in C \) by (intro lex) auto

with eq show \(?\)thesis by (auto simp: \( \text{LEX-def} \))

qed simp

qed

lemma \( \text{dc-on-LEXEQ} \):

\( \text{dc-on} (\text{LEXEQ} P) (\text{SEQ} A) \)

proof

fix \( C \) assume \( \text{chain-on} (\text{LEXEQ} P) C (\text{SEQ} A) \) and \( C \neq \{\} \)

then have \( \text{chain}: \text{chain-on} (\text{LEX} P) C (\text{SEQ} A) \) by (auto simp: \( \text{chain-on-def} \))

then have \( C \subseteq \text{SEQ} A \) by (auto simp: \( \text{chain-on-def} \))

then have \( \text{lexmin} C \in \text{SEQ} A \) using \( \{C \neq \{\}\} \) by (intro \( \text{lexmin} \)-SEQ-mem)

have \( \text{lb} (\text{LEX} P) C (\text{lexmin} C) \) by (rule \( \text{lb-LEX-lexmin} \) [OF chain \( \{C \neq \{\}\}\)])

then have \( \text{lb} (\text{LEXEQ} P) C (\text{lexmin} C) \) by (auto simp: \( \text{lb-def} \))

with \( \text{lexmin} C \in \text{SEQ} A \) show \( \exists f \in \text{SEQ} A . \text{lb} (\text{LEXEQ} P) C f \) 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**

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

**Lemma** \( pt\text{-}open\text{-}on\ D: \)

\( pt\text{-}open\text{-}on\ Q\ A \rightarrow Q f \rightarrow f \in A \rightarrow (\exists n.\ (\forall i < n.\ g i = f i) \rightarrow Q g)) \)

unfolding \( pt\text{-}open\text{-}on\text{-}def\) by blast

**Lemma** \( pt\text{-}open\text{-}on\text{-}good: \)

\( pt\text{-}open\text{-}on\ (good\ Q)\ (SEQ\ A) \)

**Proof** (unfold \( pt\text{-}open\text{-}on\text{-}def\), intro ballI)

fix \( f \) assume \( f \in\ SEQ\ A \)

show \( good\ Q\ f = (\exists n.\ g \in\ SEQ\ A.\ (\forall i < n.\ g i = f i) \rightarrow good\ Q\ g) \)

proof

assume \( good\ Q\ f \)

then obtain \( i \) and \( j \) where \( *: i < j \) \( Q (f i \ f j) \) by auto

have \( \forall g \in\ SEQ\ A.\ (\forall i < \text{Suc} j.\ g i = f i) \rightarrow good\ Q\ g \)

proof (intro ballI impI)

fix \( g \) assume \( g \in\ SEQ\ A\ and\ \forall i < \text{Suc} j.\ g i = f i \)

then show \( good\ Q\ g\ using\ *\ by\ (force\ simp: good-def)\)

qed

then show \( \exists n.\ \forall g \in\ SEQ\ A.\ (\forall i < n.\ g i = f i) \rightarrow good\ Q\ g \) ..

next

assume \( \exists n.\ \forall g \in\ SEQ\ A.\ (\forall i < n.\ g i = f i) \rightarrow good\ Q\ g \)

with \( f \) show \( good\ Q\ f\ by\ blast\)

qed

qed

context minimal-element begin

**Lemma** \( pt\text{-}open\text{-}on\text{-}imp\text{-}open\text{-}on\-LEXEQ: \)

assumes \( pt\text{-}open\text{-}on\ Q\ (SEQ\ A) \)

shows \( open\text{-}on\ (LEXEQ\ P)\ Q\ (SEQ\ A) \)

**Proof**

fix \( C \) assume chain: \( chain\text{-}on\ (LEXEQ\ P)\ C\ (SEQ\ A)\ and\ ne: C \neq \{} \)

and \( \exists g \in\ SEQ\ A.\ \text{glb}\ (LEXEQ\ P)\ C g \land Q g \)

then obtain \( g\ where\ g:\ g \in\ SEQ\ A\ and\ \text{glb}\ (LEXEQ\ P)\ C g \)

and \( Q: Q\ g\ by\ blast\)

then have \( \text{glb}: \text{glb}\ (LEX\ P)\ C g\ by\ (auto\ simp: \text{glb-def lb-def})\)

from \( \text{chain}\ have\ \text{chain}\text{-}on\ (LEX\ P)\ C\ (SEQ\ A)\ and\ C: C \subseteq\ SEQ\ A\ by\ (auto\ simp: chain\text{-}on\text{-}def)\)

note \( * = \text{glb\text{-}LEX\text{-}lexmin}[OF\ this(1)\ ne]\)

have \text{lexmin} \( C \in\ SEQ\ A\ using\ ne\ and\ C\ by\ (intro\ lexmin\text{-}SEQ\text{-}mem)\)

from \( \text{glb\text{-}unique}[OF\ -\ g\ this\ glb\ *]\)
and antisym-on-LEX [OF po-on-imp-irrefl-on [OF po] po-on-imp-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:
oppenon (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)+

27
fix \( f \) presume \( f \in \text{SEQ} \) (lists \( A \))
with \( \text{go-on-LEXEQ} \) \( \{ \text{OF po-on-imp-transp-on} \ [ \text{OF po-on-strict-suffix} \} \) and \( \text{dc-on-LEXEQ} \)
and \( \text{open-on-good} \)
show \( \text{good} \) (list-emb \( P \)) \( f \)
proof
(induct rule: open-induct-on)
case (less \( f \))
define \( h \) where \( h \cdot i = \text{hd} \ ( f \cdot i ) \) for \( i \)
show \( ?\text{case} \)
proof
(cases \( \exists i \cdot f \cdot i = [] \))
case False
then have \( \text{ne} : \forall i. \ f \cdot i \neq [] \) by auto
with \( f \in \text{SEQ} \) (lists \( A \))
\( \forall i. \ h \cdot i \in A \) by (auto simp: \( h \)-def \( \text{ne-lists} \))
from \( \text{almost-full-on-imp-homogeneous-subseq} \) \( \{ \text{OF assms this} \} \) obtain \( \varphi : \forall \cdot \text{nat} \Rightarrow \text{nat} \) where mono: \( \forall i. \ j. \ i < j \Rightarrow \varphi \cdot i < \varphi \cdot j \)
and \( P : \underbrace{\forall i. \ j. \ i < j \Rightarrow P \ ( h \ ( \varphi \ i )) \ ( h \ ( \varphi \ j ))} \) by blast
define \( f' \) where \( f' \cdot i = ( \text{if } i < \varphi \cdot 0 \text{ then } f \ ( \varphi \ ( i - \varphi \ 0 )) \text{ else } \text{tl} \ ( \varphi \ ( i - \varphi \ 0 ))) \) for \( i \)
have \( f' : f' \in \text{SEQ} \) (lists \( A \)) using \( \text{ne} \) and \( f \in \text{SEQ} \) (lists \( A \))
by (auto simp: \( f'\)-def dest: list-setSel)
have \( \text{simp} : \forall i. \ j. \ \varphi \cdot 0 \leq i \Rightarrow h \ ( \varphi \ ( i - \varphi \ 0 )) \# f' \cdot i = f \ ( \varphi \ ( i - \varphi \ 0 )) \)
\( \wedge i. \ i < \varphi \cdot 0 \Rightarrow f' \cdot i = f \cdot i \) using \( \text{ne} \) by (auto simp: \( f'\)-def h-def)
moreover have \( \text{strict-suffix} \ ( f' \ ( \varphi \ 0 )) \) (\( f \ ( \varphi \ 0 ) \)) using \( \text{ne} \) by (auto simp: \( f'\)-def)
ultimately have \( \text{LEX} \) \( \text{strict-suffix} \ \varphi \cdot f \cdot f' \)
with \( \text{LEX-imp-not-LEX} \) \( \{ \text{OF this} \} \) have \( \text{strict} \) (\( \text{LEXEQ} \) \( \text{strict-suffix} \)) \( f' \cdot f \)
using \( \text{po-on-strict-suffix} \) [\( \text{of UNIV} \)] unfolding \( \text{po-on-strict-suffix} \) irreflp-on-def transp-on-def by blast
from \( \text{less(2)} \) [\( \text{OF } f' \ this \) ] have \( \text{good} \) (list-emb \( P \)) \( f' \).
then obtain \( i \cdot j \) where \( i < j \) and emb: list-emb \( P \) \( ( f' \cdot i ) \) \( ( f' \cdot j ) \) by (auto simp: \( \text{good-def} \))
consider \( j < \varphi \cdot 0 \mid \varphi \cdot 0 \leq i \mid i < \varphi \cdot 0 \) and \( \varphi \cdot 0 \leq j \) by arith
then show \( \text{?thesis} \)
proof
(cases)
case 1 with \( \cdot i < j \) and emb show \( \text{?thesis} \) by (auto simp: \( \text{good-def} \))
next
case 2
with \( \cdot i < j \) and \( P \) have \( P \ ( h \ ( \varphi \ ( i - \varphi \ 0 ))) \ ( h \ ( \varphi \ ( j - \varphi \ 0 ))) \) by auto
with emb have list-emb \( P \) \( ( h \ ( \varphi \ ( i - \varphi \ 0 ))) \# f' \cdot i ) \ ( h \ ( \varphi \ ( j - \varphi \ 0 ))) \# f' \cdot j ) \) by auto
then have list-emb \( P \) \( f \ ( \varphi \ ( i - \varphi \ 0 ))) \ ( f \ ( \varphi \ ( j - \varphi \ 0 ))) \) using 2 and \( \cdot i < j \) by auto
moreover with 2 and \( \cdot i < j \) have \( \varphi \ ( i - \varphi \ 0 ) < \varphi \ ( j - \varphi \ 0 ) \) using mono
by auto
ultimately show \( \text{?thesis} \) by (auto simp: \( \text{good-def} \))
next
case 3
with emb have list-emb \( P \) \( f \cdot i ) \) \( f' \cdot j ) \) by auto
moreover have \( f \ ( \varphi \ ( j - \varphi \ 0 ))) = h \ ( \varphi \ ( j - \varphi \ 0 ))) \# f' \cdot j \) using 3 by auto
ultimately have list-emb \( P \) \( f \cdot i ) \) \( f \ ( \varphi \ ( j - \varphi \ 0 ))) \) by auto
moreover have \( i < \varphi \ ( j - \varphi \ 0 ) \) using mono [\( \text{of } 0 \cdot j - \varphi \ 0 \) ] and \( 3 \) by force

28
ultimately show \( \varphi \)thesis by (auto simp: good-def)
Qed
Qed auto
Qed
Qed blast
end

7 Almost-Full Relations

theory Almost-Full-Relations
imports Minimal-Bad-Sequences
begin
lemma (in mbs) mbs':
  assumes \( \neg \) almost-full-on \( P \) \( A \)
  shows \( \exists m \in \text{BAD} \ P. \ \forall g. (m, g) \in gseq \rightarrow \text{good} \ P \ g \)
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 \Rightarrow 'a option set (\⊥ [1000] 1000)
where
\( A_\bot = \{ \text{None} \} \cup \text{Some} ' A \)

lemma with-bot-iff [iff]:
  Some \( x \in A_\bot \leftrightarrow x \in A \)
by (auto simp: with-bot-def)

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

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

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

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

lemma with-bot-UNIV [simp]:
  \( \text{UNIV}_\bot = \text{UNIV} \)
proof (rule set-eql)
  fix \( x :: 'a \ \text{option} \)
show $x \in \text{UNIV}_\bot \leftrightarrow x \in \text{UNIV}$ by (cases $x$) auto

qed

7.2 Adding a Bottom Element to an Almost-Full Set

fun
   \text{option-le} :: \('a \Rightarrow 'a \Rightarrow \text{bool}' \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow \text{bool}'
where
   \text{option-le} P \text{ None} y = \text{True} |
   \text{option-le} P (\text{Some} x) \text{ None} = \text{False} |
   \text{option-le} P (\text{Some} x) (\text{Some} y) = P x y

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

lemma \text{almost-full-on-with-bot}:
   assumes \text{almost-full-on} P A
   shows \text{almost-full-on} (\text{option-le} P) A_\bot (\text{is \ almost-full-on} ?P ?A)
proof
   fix \text{f} :: \text{nat} \Rightarrow 'a \text{ option}
   assume $\ast$ : $\forall i. \text{f} i \in ?A$
   show \text{good} ?P $f$
   proof (cases $\forall i. \text{f} i \neq \text{None}$)
      case True
      then have $\ast\ast$: $\forall i. \text{Some} (\text{the} (\text{f} i)) = \text{f} i$
      and $\forall i. \text{the} (\text{f} i) \in A$ using $\ast$ by auto
      with \text{almost-full-onD} (\text{OF} \text{assms}, \text{of} \text{the} \circ \text{f}) \text{obtain} i \ j \text{ where} i < j
      and $\text{P} (\text{the} (\text{f} i)) (\text{the} (\text{f} j))$ by auto
      then have $?P (\text{Some} (\text{the} (\text{f} i))) (\text{Some} (\text{the} (\text{f} j)))$ by simp
      then have $?P (\text{f} i) (\text{f} j)$ unfolding $\ast\ast$.
      with $(i < j)$ show \text{good} $?P f$ by (auto simp: \text{good-def})
   qed auto
qed

7.3 Disjoint Union of Almost-Full Sets

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

lemma \text{not-sum-le-cases}:
   assumes $\sim \text{sum-le} P Q a b$
   and $\forall x \ y. [a = \text{Inl} x; b = \text{Inl} y; \sim P x y] \Longrightarrow \text{thesis}$
   and $\forall x \ y. [a = \text{Inr} x; b = \text{Inr} y; \sim Q x y] \Longrightarrow \text{thesis}$
   and $\forall x \ y. [a = \text{Inl} x; b = \text{Inr} y] \Longrightarrow \text{thesis}$
   qed
and $\forall x y. [a = \text{Inr } x; b = \text{Inl } y] \implies \text{thesis}$

shows thesis

using asms 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 $P Q$) ($A <\leftrightarrow B$) (is almost-full-on $?P \ ?A$

**Proof**

fix $f :: \text{nat} \Rightarrow (\text{'}a + \text{'}b)$

let $?I = f - \text{'}\text{Inl } A$

let $?J = f - \text{'}\text{Inr } B$

assume $\forall i. f i \in ?A$

then have $\ast$: $\forall i. f i \in ?A$

by (fastforce)

show good $?P f$

proof (rule ccontr)

assume bad $?P f$

show False

proof (cases finite $?I$

assume finite $?I$

then have infinite $?J$ by (auto simp: $\ast$)

then interpret infinitely-many1 $\lambda i. f i \in \text{Inr } B$

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

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

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

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

have $B: \forall i. f i \in B$ 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 $\neg ?F (f (\text{enum } i)) (f (\text{enum } j))$ by (auto simp: good-def)

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

then have bad Q $?F$ by (auto simp: good-def)

moreover from almost-full-on $Q B$ and $B$

have good Q $?F$ by (auto simp: good-def almost-full-on-def)

ultimately show False by blast

next

assume infinite $?I$

then interpret infinitely-many1 $\lambda i. f i \in \text{Inr } A$

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

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

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

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

have $A: \forall i. f i \in A$ using enum-P by (auto simp: image-iff) (metis sum.sel(I))

\{ fix i j :: nat

assume $i < j$

then have enum i < enum j using enum-less by auto
with \( \text{bad have } \neg \forall P \ (f \ (\text{enum } i)) \ (f \ (\text{enum } j)) \ \text{by (auto simp: good-def}) \)
then have \( \neg P \ (f \ i \ (f \ j)) \ \text{by (auto elim: not-sum-le-cases}) \) 
then have \( \text{bad } P \ (\forall f \ i) \ (\forall f \ j) \ \text{by (auto simp: good-def}) \)

moreover from \( \text{almost-full-on } P \ A \ \text{and } A \ 
\text{have good } P \ (\forall f \ i) \ (\forall f \ j) \ \text{by (auto simp: good-def almost-full-on-def}) \)
ultimately show \( \text{False} \ \text{by blast} \)

\[ \text{qed} \]
\[ \text{qed} \]
\[ \text{qed} \]

7.4 Dickson’s Lemma for Almost-Full Relations

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

definition prod-le :: \( '(a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow \text{bool} \)
where 
prod-le P1 P2 = \( \lambda \ (p1, p2) \ (q1, q2). \ P1 \ p1 \ q1 \ \land \ P2 \ p2 \ q2 \)

lemma prod-le-True [simp]:
prod-le P (\( \lambda \ -. \ True \)) a b = \( P \ (\text{fst } a) \ (\text{fst } b) \) 
by (auto simp: prod-le-def)

lemma almost-full-on-Sigma:
assumes almost-full-on P1 A1 and almost-full-on P2 A2
shows almost-full-on (prod-le P1 P2) (A1 \times A2) (is 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: \( \text{bad } P \ (\forall f \ i) \) 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 fst: \( \forall i. \ \text{fst } (f \ i) \in A1 \) and snd: \( \forall i. \ \text{snd } (f \ i) \in A2 \)
by (metis SigmaE fst-conv, metis SigmaE snd-conv)
from almost-full-on-imp-homogeneous-subseq [OF assms(1) fst]
obtain \( \varphi :: \text{nat} \Rightarrow \text{nat} \) where mono: \( \forall i. \ j. \ i < j \Rightarrow \varphi \ i < \varphi \ j \) 
\( \land \ast: \forall i. \ j. \ i < j \Rightarrow ?W \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by auto
from 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 assms(2) have \( \text{good } P2 \ (\text{snd } \circ \ f \circ \varphi) \) by (auto simp: almost-full-on-def)
then obtain i j :: nat
where \( i < j \) and \( \exists B \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by auto
with \( \ast \ [OF \ i < j]\) have \( \forall P \ (f \ (\varphi \ i)) \ (f \ (\varphi \ j)) \) by (simp add: case-prod-beta prod-le-def)
with mono [OF \ i < j] and bad show \( \text{False} \) 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 \( \neg P \) \( \neg A \))
proof (rule ccontr)
\begin{align*}
  &\text{interpret mbs \( ?A \) .} \\
  &\text{assume } \neg \text{thesis} \\
  &\text{from mbs' \( [OF \ this] \) obtain } m \\
  &\quad \text{where } \text{bad}: m \in \text{BAD} \ \neg P \\
  &\quad \text{and } \text{min}: \forall g. \ (m, g) \in \text{gseq} \longrightarrow \text{good} \ \neg P \ g \ldots \\
  &\quad \text{then have lists: } \bigwedge i. m \ i \in \text{lists} A \\
  &\quad \quad \text{and ne: } \bigwedge i. m \ i \ \neq \ [] \ \text{by auto} \\
\end{align*}

\begin{align*}
  \text{define } h \ t \ \text{where } h &= (\lambda i. \ \text{hd} \ (m \ i)) \ \text{and } t = (\lambda i. \ \text{tl} \ (m \ i)) \\
  \text{have } m: \bigwedge i. m \ i = h \ i \ # \ t \ i \ \text{using ne by (simp add: h-def t-def)}
\end{align*}

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

\begin{align*}
  &\quad \text{where } \text{less}: \bigwedge i. j. \ i < j \ \Longrightarrow \ \varphi \ i < \varphi \ j \\
  &\quad \text{and } P: \forall i. j. \ i < j \longrightarrow P \ (h \ (\varphi \ i)) \ (h \ (\varphi \ j)) \ \text{by blast} \\
\end{align*}

have \( \text{bad-t: } \text{bad} \ ?P \ (t \circ \varphi) \)
proof
\begin{align*}
  &\quad \text{assume } \text{good} \ ?P \ (t \circ \varphi) \\
  &\quad \text{then obtain } i \ j \ \text{where } i < j \ \text{and } \ ?P \ (t \ (\varphi \ i)) \ (t \ (\varphi \ j)) \ \text{by auto} \\
  &\quad \text{moreover with } P \ \text{have } P \ (h \ (\varphi \ i)) \ (h \ (\varphi \ j)) \ \text{by blast} \\
  &\quad \text{ultimately have } \ ?P \ (m \ (\varphi \ i)) \ (m \ (\varphi \ j)) \\
  &\quad \quad \text{by (subst (1 2) } m) \ \text{rule list-emb-Cons2, auto) \\
  &\quad \text{with less and } (i < j) \ \text{have good } \ ?P \ m \ \text{by (auto simp: good-def)} \\
  &\quad \text{with bad show False by blast} \\
\end{align*}
qed

\begin{align*}
  \text{define } m' \ \text{where } m' &= (\lambda i. \ \text{if } i < \varphi \ 0 \ \text{then } m \ i \ \text{else } t \ (\varphi \ (i - \varphi \ 0))) \\
  \text{have } m'^{-}\text{less}: \bigwedge i. i < \varphi \ 0 \ \Longrightarrow \ m' \ i = m \ i \ \text{by (simp add: m'-def)} \\
  \text{have } m'^{-}\text{geq}: \bigwedge i. i \geq \varphi \ 0 \ \Longrightarrow \ m' \ i = t \ (\varphi \ (i - \varphi \ 0)) \ \text{by (simp add: m'-def)}
\end{align*}

have \( \forall i. \ m' \ i \in \text{lists} A \ \text{using ne-lists} \ [OF \ ne] \ \text{and lists by (auto simp: m'-def t-def)} \)
moreover have \( \text{length} \ (m' \ (\varphi \ 0)) < \text{length} \ (m \ (\varphi \ 0)) \ \text{using ne by (simp add: t-def m'-geq)} \)
moreover have \( \forall j < \varphi \ 0. \ m' \ j = m \ j \ \text{by (auto simp: m'-less)} \)
ultimately have \( (m, m') \in \text{gseq} \ \text{using lists by (auto simp: gseq-def)} \)
moreover have \( \text{bad} \ ?P \ m' \)
proof
\begin{align*}
  &\quad \text{assume } \text{good} \ ?P \ m' \\
  &\quad \text{then obtain } i \ j \ \text{where } i < j \ \text{and } \text{emb: } ?P \ (m' \ i) \ (m' \ j) \ \text{by (auto simp: good-def)} \\
  &\quad \quad \{ \ \text{assume } j < \varphi \ 0 \\
  &\quad \quad \quad \text{with } (i < j) \ \text{and } \text{emb have } ?P \ (m \ i) \ (m \ j) \ \text{by (auto simp: m'-less)} \}
\end{align*}
with \((i < j)\) and \(bad\) 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: \(m^\prime\)-geq)
  with \(bad-t\) have \(False\) by auto }

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: \(m^\prime\)-less \(m^\prime\)-geq)
  from \(\text{list-emb-Cons} \ [OF\ this,\ of\ h\ (\varphi \ (j - \varphi \ 0))]\)
  have \(?P\ (m \ i)\) (\(m \ (\varphi \ (j - \varphi \ 0)))\) using \(ne\) by (simp add: \(h\)-def \(t\)-def)
  moreover have \(i < \varphi \ (j - \varphi \ 0)\)
  using \(less\ [of\ 0\ j - \varphi \ 0]\) and \((i < \varphi \ 0,\ and\ \varphi \ 0 \leq \ j)\)
  by (cases \(j = \varphi \ 0\)) auto
  ultimately have \(False\) using \(bad\) by \(blast\) }
  ultimately show \(False\) using \((i < j)\) by \(arith\)
  qed

ultimately show \(False\) using \(\min\) by \(blast\)
  qed

7.6 Natural Numbers

lemma \(\text{almost-full-on-UNIV-nat:}\)
  \(\text{almost-full-on} \ (\leq) \ (\text{UNIV} :: \text{nat set})\)

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

8 Well-Quasi-Orders

theory \(\text{Well-Quasi-Orders}\)
imports \(\text{Almost-Full-Relations}\)
begin

8.1 Basic Definitions

definition wqo-on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
wqo-on P A \iff 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] \implies wqo-on P A
unfolding wqo-on-def almost-full-on-def by blast

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

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

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

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

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

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

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

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

lemma wqo-on-imp-wfp-on:
wqo-on P A \Rightarrow wfp-on (strict P) A
by (metis (no-types) wqo-on-imp-qo-on wqo-wf-and-no-antichain-conv)

The homomorphic image of a wqo set is wqo.

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

The monomorphic preimage of a wqo set is wqo.

lemma wqo-on-mon:
assumes \forall x \in A. \forall y \in A. \ P x y \imp Q (h x) (h y)
and bij: bij-betw h A B
and wqo: wqo-on Q B
shows wqo-on P A
proof
have transp-on P A
proof
fix x y z assume [intro]: x \in A y \in A z \in A
and \( P \ x \ y \) and \( P \ y \ z \)
with \* have \( Q \ (h \ x) \ (h \ y) \) and \( Q \ (h \ y) \ (h \ z) \) by blast+
with \ wqo-on-imp-transp-on \ [OF \ wqo] \ have \( Q \ (h \ x) \ (h \ z) \)
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.

\[
\begin{align*}
\text{class} & \ wqo = \ preorder + \\
& \text{assumes} \ good: \ good \ (\leq) \ f
\end{align*}
\]

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

\textbf{lemma} \ wqo-on-UNIV-class-wqo \ [intro!]:
\( wqo-on \ P \ UNIV =\Rightarrow \ \text{class.wqo} \ P \) (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 \( \text{class.wqo} \).

\textbf{lemma} \ wqo-on-UNIV-class-wqo \ [intro!]:
\( wqo-on \ P \ UNIV \iff \ \text{class.wqo} \ P \) (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.

\textbf{lemma} \ (in \ wqo) \ wfp \ (<)
\proof
have \ class.wqo \ (\leq) \ (<) .
hence \ wqo-on \ (\leq) \ UNIV
unfolding \ less-le-not-le \ [abs-def] \ wqo-on-UNIV-conv \ [symmetric] ,
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 wqo-on (prod-le P1 P2) (A1 × A2) (is wqo-on ?P ?A)
proof –
  { from assms have transp-on P1 A1 and transp-on P2 A2 by (auto simp: wqo-on-def)
    hence transp-on ?P ?A unfolding transp-on-def prod-le-def by blast }

38
moreover
  \{ from \textit{assms and almost-full-on-Sigma [of P1 A1 P2 A2]}
  have almost-full-on \textit{?P \,?A by (auto simp: wqo-on-def) } \}\nultimately
show \textit{?thesis by (auto simp: wqo-on-def)}
qed

lemmas dickson = wqo-on-Sigma

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

\textbf{8.5 Higman’s Lemma}

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

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

lemmas higman = wqo-on-lists

\textbf{lemma wqo-on-list-UNIV [intro]:}
  wqo-on P UNIV \implies wqo-on (list-emb P) UNIV
using wqo-on-lists [of P UNIV] by simp

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

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

\textbf{lemma finite-eq-wqo-on:}
  \textbf{assumes} finite A
  \textbf{shows} wqo-on (=) A
using \textit{finite-wqo-on [OF assms, of (=)]}
  by (auto simp: reflp-on-def transp-on-def)

\textbf{lemma wqo-on-lists-over-finite-sets:}
  wqo-on (list-emb (=)) (UNIV::('a::finite) list set)
using \textit{wqo-on-lists [OF finite-eq-wqo-on [OF finite [of UNIV::('a::finite) set]]] by simp}
lemma \textit{wqo-on-map}:
 fixes \( P \) and \( Q \) and \( h \)
defines \( P' \equiv \lambda x \ y. P \ x \ y \land Q \ (h \ x) \ (h \ y) \)
assumes \textit{wqo-on} \( P \ A \)
 and \textit{wqo-on} \( Q \ B \)
 and \textit{subset}: \( h ' A \subseteq B \)
shows \textit{wqo-on} \( P' \ A \)
proof
\begin{itemize}
  \item let \( \forall x \ y. Q \ (h \ x) \ (h \ y) \)
  \item from \textit{⟨wqo-on \( P \ A \)⟩} have \textit{transp-on} \( P \ A \)
    by (rule \textit{wqo-on-imp-transp-on})
  \item then show \textit{transp-on} \( P' \ A \)
    using \textit{⟨wqo-on \( Q \ B \)⟩} and \textit{subset}
    unfolding \textit{wqo-on-def transp-on-def} \( P' \)-def by blast
\end{itemize}

\begin{itemize}
  \item \textit{from} \textit{⟨wqo-on \( P \ A \)⟩} have \textit{almost-full-on} \( P \ A \)
    by (rule \textit{wqo-on-imp-almost-full-on})
  \item \textit{from} \textit{⟨wqo-on \( Q \ B \)⟩} have \textit{almost-full-on} \( Q \ B \)
    by (rule \textit{wqo-on-imp-almost-full-on})
\end{itemize}

shows \textit{almost-full-on} \( P' \ A \)
proof
\begin{itemize}
  \item fix \( f \)
  \item assume \( \forall i :: \text{nat}. f i \in A \)
  \item from \textit{⟨almost-full-on-imp-homogeneous-subseq} \( \text{OF} \langle \text{almost-full-on} \ P \ A \rangle \ this \rangle \)
    obtain \( g :: \text{nat} \Rightarrow \text{nat} \)
    where \( g. \forall i j. i < j \implies g \ i < g \ j \)
    and \( \forall i. f \ (g \ i) \in A \land P \ (f \ (g \ i)) \ (f \ (g \ \text{Suc} \ i)) \)
    using \( * \) by \text{auto}
  \item from \textit{chain-transp-on-less} \( \langle \text{OF} \ \langle \text{transp-on} \ P \ A \rangle \ \rangle \)
    have \( \forall i j. i < j \implies P \ (f \ (g \ i)) \ (f \ (g \ j)) \)
  \item let \( g = \lambda i. h \ (f \ (g \ i)) \)
  \item from \( * \) and \textit{subset} have \( B :: \forall i :: \text{nat}. \ ?g \ i \in B \) by \text{auto}
    with \( \langle \text{almost-full-on} \ Q \ B \rangle \) \[\text{unfolded \textit{almost-full-on-def} \( \text{good-def, THEN} \ \text{bspec, OF} \ ?g \rangle} \]
    obtain \( i j :: \text{nat} \)
    where \( i < j \) and \( Q \ (\ ?g \ i) \ (\ ?g \ j) \) by \text{blast}
    with \( \langle \text{OF} \ \langle \ ?g \ i < j \rangle \rangle \) have \textit{P'} \( f \ (g \ i)) \ (f \ (g \ j)) \)
    by (auto simp: \( \text{P'} \)-def)
  \item with \( g \ (\ ?g \ i < j) \) show \textit{good} \( P' \ f \) by (auto simp: \textit{good-def})
\end{itemize}
qed

end
9 Kruskal’s Tree Theorem

theory Kruskal
imports Well-Quasi-Orders
begin

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 (emb P) trees (is almost-full-on ?P ?A)
proof (rule ccontr)
interpret mbs ?A .
assume ¬ ?thesis
from mbs" [OF this] obtain m

41
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 trees: \( \lambda i. \ m i \in \text{trees} \) 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 \( S = \bigcup \{ \text{set} \ (a \ i) \mid i. \ True \} \)

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

have almost-full-on \( \ ?P \ S \)
proof (rule ccontr)
assume \( \neg \ ?thesis \)
then obtain \( s \ : \ \text{nat} \Rightarrow 'a \)
  where \( S = \lambda i. \ s \ i \in S \) and \( \text{bad-s} : \ ?P \ s \) by (auto simp: 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 LeastI-ex [OF this] obtain \( k \)
where \( \text{sk} : s \ k \in \text{set} \ (a \ n) \) by (auto simp: \( \text{n-def} \)
have \( \text{args} : \lambda k. \ \exists m \geq n. \ s \ k \in \text{set} \ (a \ m) \)
using \( S \) by (auto simp: \( \text{S-def} \) (metis 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 \( m'\text{-less} : \lambda i. \ i < n \Rightarrow m' \ i = m \ i \) by (simp add: \( \text{m'-def} \)
have \( m'\text{-geq} : \lambda i. \ i \geq n \Rightarrow m' \ i = s \ (k + (i - n)) \) 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 }
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: \( \text{m'-geq} \)
    with \( \text{bad-s} \) have \( \text{False} \) by auto }
moreover
\{ \text{assume } i < n \text{ and } n \leq j \}
\begin{align*}
\text{with } &i' < j \text{ and } \text{emb have } *: ?P (m \, i') (s \, (k + (j - n))) \text{ by (auto simp: m'-less m'-geq)} \\
\text{with } &\text{args obtain } l \text{ where } l \geq n \text{ and } **: s \, (k + (j - n)) \in \text{set } (a \, l) \text{ by blast} \\
\text{from } &\text{emb,ary } [\text{OF sig of } l] - \text{a-trees of } l \text{ ** *] } \\
\text{have } &?P (m \, i') (m \, l) \text{ by (simp add: } m) \\
\text{moreover have } &i < l \text{ using } (i < n) \text{ and } (n \leq l) \text{ by auto} \\
\text{ultimately have } &\text{False using bad by blast } \\
\text{ultimately show } &\text{False using } (i < j) \text{ by arith} \\
\text{qed} \\
\text{moreover have } & (m, m') \in \text{gseq} \\
\text{proof } - \\
\text{have } &m \in \text{SEQ } ?A \text{ using trees by auto} \\
\text{moreover have } &m' \in \text{SEQ } ?A \\
\text{using } &\text{trees and } S \text{ and } \text{trees-args } [\text{OF trees}] \text{ by (auto simp: m'-def a-def S-def)} \\
\text{moreover have } &\forall i < n. \, m \, i = m' \, i \text{ by (auto simp: m'-less)} \\
\text{moreover have } &\text{size } (m' \, n) < \text{size } (m \, n) \\
\text{using } &\text{sk and } \text{size-ary } [\text{OF trees, unfolded } m] \\
\text{by } &\text{(auto simp: m'-geq root-mk } [\text{OF sig}] \text{ args-mk } [\text{OF sig}]) \\
\text{ultimately show } &?\text{thesis by (auto simp: gseq-def)} \\
\text{qed} \\
\text{ultimately show } &\text{False using min by blast} \\
\text{qed} \\
\text{from } &\text{almost-full-on-lists } [\text{OF this, THEN almost-full-on-imp-homogeneous-subseq, OF lists}] \\
\text{obtain } &\varphi :: \text{nat } \Rightarrow \text{nat} \\
\text{where } &\text{less: } \land \, i \, j. \, i < j \Rightarrow \varphi \, i < \varphi \, j \\
\text{and } &\text{lemb: } \land \, i \, j. \, i < j \Rightarrow \text{list-emb } ?P (a \, (\varphi \, i)) (a \, (\varphi \, j)) \text{ by blast} \\
\text{have } &\text{roots: } \land \, i. \, r \, (\varphi \, i) \in \text{F using trees } [\text{THEN trees-root}] \text{ by (auto simp: r-def)} \\
\text{then have } &r \circ \varphi \in \text{SEQ } F \text{ by auto} \\
\text{with } &\text{assms have } \text{good } P \, (r \circ \varphi) \text{ by (auto simp: almost-full-on-def)} \\
\text{then obtain } &i \, j \\
\text{where } &\text{i < j and } P \, (r \, (\varphi \, i)) \, (r \, (\varphi \, j)) \text{ by auto} \\
\text{with } &\text{lemb } [\text{OF } i < j] \text{ have } ?P (m \, (\varphi \, i)) (m \, (\varphi \, j)) \\
\text{using } &\text{sig and arity and } \text{a-trees by (auto simp: m intro: emb.list-emb)} \\
\text{with } &\text{less } [\text{OF } i < j] \text{ and bad show } \text{False by blast} \\
\text{qed} \\
\text{inductive-cases} \\
\text{emb-mk2 [consumes 1, case-names arg list-emb]: emb } P \, s \, (mk \, y \, ts) \\
\text{inductive-cases} \\
\text{list-emb-nil2-cases: list-emb } P \, xs \, [] \text{ and} \\
\text{list-emb-cons-cases: list-emb } P \, xs \, (y\#ys) \\
\text{lemma } &\text{list-emb-trans-right:} \\
\text{assumes } &\text{list-emb } P \, xs \, ys \text{ and } \text{list-emb } (\lambda y \, z. \, P \, y \, z \, \land \, (\forall x. \, P \, x \, y \, \rightarrow \, P \, x \, z)) \, ys
shows \( \text{list-emb} \ P \ x s \ z s \)

using \( \text{assms}(2, \ 1) \) by (induct arbitrary: \( x s \)) (auto elim!: list-emb-Nil2-cases list-emb-Cons-cases)

**lemma** \( \text{emb-trans} \):

assumes \( \text{trans} : \bigwedge f \ g \ h. \ f \in F \implies g \in F \implies h \in F \implies P \ f \ g \implies P \ g \ h \implies P \ f \ h \)

shows \( \text{emb} \ P \ s \ t \) and \( \text{emb} \ P \ t \ u \)

using \( \text{assms}(3, \ 2) \)

proof (induct arbitrary: \( s \))

case \( \text{arg} \ f \ m \ t s \ v \)

then show \( \text{?case by (auto intro: emb.arg)} \)

next

case \( \text{list-emb} \ f \ m \ g \ n \ s s \ t s \)

note \( \text{IH = this} \)

from \( \langle \text{emb} \ P \ s \ (\text{mk} \ f \ s s) \rangle \)

show \( \text{?case} \)

proof (cases rule: emb-mk2)

case \( \text{arg} \)

then show \( \text{?thesis using IH by (auto elim!: list-emb-set intro: emb.arg)} \)

next

case \( \text{list-emb} \)

then show \( \text{?thesis using IH by (auto intro: emb.intros dest: trans list-emb-trans-right)} \)

qed

**lemma** \( \text{transp-on-emb} \):

assumes \( \text{transp-on} \ P \ F \)

shows \( \text{transp-on} \ (\text{emb} \ P) \ \text{trees} \)

using \( \text{assms and emb-trans [of P]} \) unfolding \( \text{transp-on-def} \) by blast

**lemma** \( \text{kruskal} \):

assumes \( \text{wqo-on} \ P \ F \)

shows \( \text{wqo-on} \ (\text{emb} \ P) \ \text{trees} \)

using \( \text{almost-full-on-trees [of P]} \) and \( \text{assms by (metis transp-on-emb wqo-on-def)} \)

end

end

theory \( \text{Kruskal-Examples} \)

imports \( \text{Kruskal} \)

begin

datatype 'a tree = Node 'a 'a tree list

fun node

where
\[
\text{node} (\text{Node } f \ t s) = (f, \text{length } t s)
\]

\textbf{fun} \text{succs} \\
\textbf{where} \\
\text{succs} (\text{Node } f \ t s) = t s

\textbf{inductive-set} trees \text{ for } A \\
\textbf{where} \\
f \in A \implies \forall t \in \text{set } t s. \ t \in \text{trees } A \implies \text{Node } f \ t s \in \text{trees } A

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

\textbf{interpretation} kruskal-tree-tree: kruskal-tree A \times \text{UNIV \ Node node succs trees } A \\
\text{for } A \\
\text{apply } (\text{unfold-locales}) \\
\text{apply auto} \\
\text{apply} (\text{case-tac }[!] \ t \text{ rule: trees.cases}) \\
\text{apply auto} \\
\text{by } (\text{metis less-not-refl not-less-eq size-list-estimation})

\textbf{thm} kruskal-tree-tree.almost-full-on-trees \\
\textbf{thm} kruskal-tree-tree.kruskal

\textbf{definition} tree-emb A P = kruskal-tree-tree.emb A (prod-le P (\lambda - -. True))

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

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

\textbf{instantiation} tree :: (wqo) wqo \\
\textbf{begin} \\
\textbf{definition} s \leq t \longleftrightarrow \text{tree-emb UNIV } (\leq) \ s t \\
\textbf{definition} (s :: \text{'a tree}) < t \longleftrightarrow s \leq t \land \neg (t \leq s) \\
\textbf{instance} \\
\textbf{by } (\text{rule class.wqo.of-class.intro}) \\
\textbf{(auto simp: less-eq-tree-def [abs-def] less-tree-def [abs-def] } \\
\text{intro: wqo-on-trees [of - UNIV, simplified])}

45
end

datatype ('f, 'v) term = Var 'v | Fun 'f ('f, '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) ∈ F → length ts = n → ∀ s ∈ set ts. s ∈ gterms F → Fun f ts ∈ 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
  ∀ t ∈ set ts. t ∈ terms → Fun f ts ∈ 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 'a exp = V 'a | C nat | Plus 'a exp 'a exp

datatype 'a symb = v 'a | c nat | p

fun mk
where
  mk (v x) [] = V x |
  mk (c n) [] = C n |
  mk p [a, b] = Plus a b
fun \( rt \) where
\[
rt \ (V \ x) = (v \ x, 0 \ :: \ \text{nat}) \ |
\]
\[
rt \ (C \ n) = (c \ n, 0) \ |
\]
\[
rt \ (Plus \ a \ b) = (p, 2)
\]

fun \( ags \) where
\[
ags \ (V \ x) = [] \ |
\]
\[
ags \ (C \ n) = [] \ |
\]
\[
ags \ (Plus \ a \ b) = [a, b]
\]

inductive-set \( exps \)
where
\[
V \ x \in \ exps \ |
\]
\[
C \ n \in \ exps \ |
\]
\[
a \in \ exps \Rightarrow b \in \ exps \Rightarrow Plus \ a \ b \in \ exps
\]

lemma [simp]:
assumes \( length \ ts = 2 \)
shows \( rt \ (mk \ p \ ts) = (p, 2) \)
using \( assms \) by (induct \( ts \)) (auto, case-tac \( ts \), auto)

lemma [simp]:
assumes \( length \ ts = 2 \)
shows \( ags \ (mk \ p \ ts) = ts \)
using \( assms \) by (induct \( ts \)) (auto, case-tac \( ts \), auto)

interpretation kruskal-exp: kruskal-tree
\[
\{(v \ x, 0) \mid x. \ True\} \cup \{(c \ n, 0) \mid n. \ True\} \cup \{(p, 2)\}
\]
\( mk \ rt \ ags \ exps \)
apply (unfold-locales)
apply auto
apply (case-tac []) rule: exps.cases
by auto

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

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

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** \( (x :: 'a \text{ option}) < y \iff x \leq y \land \neg (y \leq x) \)

**instance**

\( \text{by (rule class.wqo.of-class.intro)} \)

\( \text{(auto simp: less-eq-option-def [abs-def] less-option-def [abs-def])} \)

**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** \( (x :: 'a + 'b) < y \iff x \leq y \land \neg (y \leq x) \)

**instance**

\( \text{by (rule class.wqo.of-class.intro)} \)

\( \text{(auto simp: less-eq-sum-def [abs-def] less-sum-def [abs-def])} \)

**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** \( (p :: 'a \times 'b) < q \iff p \leq q \land \neg (q \leq p) \)

**instance**

\( \text{by (rule class.wqo.of-class.intro)} \)

\( \text{(auto simp: less-eq-prod-def [abs-def] less-prod-def [abs-def])} \)

**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** list :: (wqo) wqo

**begin**

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

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

**instance**
11 Multiset Extension of Orders (as Binary Predicates)

theory Multiset-Extension
imports
  Open-Induction.Restricted-Predicates
  HOL-Library.Multiset
begin

definition multisets :: 'a set ⇒ 'a multiset set where
  multisets A = {M. set-mset M ⊆ A}

lemma in-multisets-iff: M ∈ multisets A ←→ set-mset M ⊆ A
  by (simp add: multisets-def)

lemma empty-multisets [simp]: {#} ∈ multisets F
  by (simp add: in-multisets-iff)

lemma multisets-union [simp]: M ∈ multisets A =⇒ N ∈ multisets A =⇒ M + N ∈ multisets A
  by (auto simp add: in-multisets-iff)

definition mulex1 :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool where
  mulex1 P = (λM N. (λ(x, y). P x y))

lemma mulex1-empty [iff]:
  mulex1 P M {#} =⇒ 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#}) =⇒
  (∃M. mulex1 P M M0 ∧ N = M + {#a#}) ∨
  (∃K. (∀b. b ∈ #. K =⇒ P b a) ∧ N = M0 + K)
  using less-add [of N a M0 {(x, y). P x y}]
  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 ∀ 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 ∀ b. b ∈ {#} → (b, a) ∈ \? R by simp
  ultimately show ?thesis unfolding mult1-def by force
qed

lemma empty-mulex1 [simp]:
mulex P {#} {#a#}
using empty-mult1 of a {x, y}. P x y by (simp add: mulex1-def)
definition mulex-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ 'a multiset ⇒ bool
  where mulex-on P A = (restrict-to (mulex1 P) (multisets A))++
abbreviation mulex :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ 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 \ ∀ M N. \[ M ∈ multisets A; N ∈ multisets A; mulex1 P M N \] ⇒ \ Q M N
  and \ ∀ L M N. \[ mulex-on P A L M; Q L M; N ∈ multisets A; mulex1 P M N \] ⇒ \ 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]:

50
\{\#x\#\} \in \text{multisets } A \iff x \in A
by \ (\text{auto simp: multisets-def})

\textbf{lemma union-multisetsD:}
assumes \(M + N \in \text{multisets } A\)
s\textbf{shows} \(M \in \text{multisets } A \land N \in \text{multisets } A\)
\textbf{using} 
assms \ by \ (\text{auto simp: multisets-def})

\textbf{lemma mulex-on-multisetsD} \ [\text{dest}]:
assumes \(\text{mulex-on } P F M N\)
s\textbf{shows} \(M \in \text{multisets } F \text{ and } N \in \text{multisets } F\)
\textbf{using} assms \ by \ (\text{induct}) \ auto

\textbf{lemma union-multisets-iff} \ [\text{iff}]:
\(M + N \in \text{multisets } A \iff M \in \text{multisets } A \land N \in \text{multisets } A\)
by \ (\text{auto dest: union-multisetsD})

\textbf{lemma add-mset-multisets-iff} \ [\text{iff}]:
\(\text{add-mset } a M \in \text{multisets } A \iff a \in A \land M \in \text{multisets } A\)
\textbf{unfolding} \(\text{add-mset-add-single[of a } M\) \ union-multisets-iff \ \text{by} \ \text{auto}

\textbf{lemma mulex-on-trans:}
mulex-on \(P A L M \rightarrow mulex-on \ P A M N \rightarrow mulex-on \ P A L N\)
by \ (\text{auto simp: mulex-on-def})

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

\textbf{lemma mulex-on-add-right} \ [\text{simp}]:
assumes \(\text{mulex-on } P A M N \text{ and } a \in A\)
s\textbf{shows} \(\text{mulex-on } P A M (\text{add-mset } a N)\)
\textbf{proof} –
\textbf{from} \ assms \ \textbf{have} \(a \in A \land N \in \text{multisets } A\) \ \textbf{by} \ \text{auto}
\textbf{then have} \(\text{mulex-on } P A N (\text{add-mset } a N)\) \ \textbf{by} \ \text{simp}
\textbf{with} \(\text{mulex-on } P A M N\) \ \textbf{show} \ ?thesis \ \textbf{by} \ (\text{rule mulex-on-trans})
\textbf{qed}

\textbf{lemma empty-mulex-on} \ [\text{simp}]:
assumes \(M \neq \{\#\} \text{ and } M \in \text{multisets } A\)
s\textbf{shows} \(\text{mulex-on } P A \{\#\} M\)
\textbf{using} assms
\textbf{proof} \ (\text{induct } M)\n\textbf{case} \ (\text{add } a M)\n\textbf{show} \ ?case
\textbf{proof} \ (\text{cases } M = \{\#\})\n\textbf{assume} \(M = \{\#\}\)
\textbf{with add show} \ ?thesis \ \textbf{by} \ (\text{auto simp: mulex-on-def})
\textbf{next}
\textbf{lemma} mulex-on-self-add-right [simp]:
\begin{itemize}
  \item \textbf{assumes} $M \in \text{multisets A and } K \in \text{multisets A and } K \neq \{\#\}$
  \item \textbf{shows} mulex-on $P A M (M + K)$
\end{itemize}
\textbf{using} \text{assms}
\textbf{proof (induct K)}
\begin{itemize}
  \item \textbf{case empty}
  \begin{itemize}
    \item \textbf{then show} \text{?case by (cases $K = \{\#\}$) auto}
  \end{itemize}
  \item \textbf{next}
    \begin{itemize}
      \item \textbf{case} (add $a M$
        \begin{itemize}
          \item \textbf{show} \text{?case by (cases $M = \{\#\}$) auto}
        \end{itemize}
      \end{itemize}
      \item \textbf{next}
        \begin{itemize}
          \item \textbf{assume} $M \neq \{\#\}$ \text{with add show} \text{?thesis by auto}
          \item \textbf{by (auto dest: mulex-on-add-right simp add: ac-simps)}
        \end{itemize}
  \end{itemize}
\end{itemize}
\textbf{qed}
\textbf{qed simp}

\textbf{lemma} mult1-singleton [iff]:
\begin{itemize}
  \item $(\{\#\}, \{\#\}) \in \text{mult1 R} \iff (x, y) \in R$
\end{itemize}
\textbf{proof}
\begin{itemize}
  \item \textbf{assume} $(x, y) \in R$
  \begin{itemize}
    \item \textbf{then have} $\{\#\} = \{\#\} + \{\#\}$
    \item \text{and} $\{\#\} = \{\#\} + \{\#\}$
    \item \text{and} $\forall b. \ b \in \# \{\#\} \rightarrow (b, y) \in R$ \text{by auto}
  \end{itemize}
  \item \textbf{then show} $(\{\#\}, \{\#\}) \in \text{mult1 R unfolding mult1-def by blast}$
  \item \textbf{next}
    \begin{itemize}
      \item \textbf{assume} $(\{\#\}, \{\#\}) \in \text{mult1 R}$
      \item \textbf{then obtain} $M0 K a$
      \begin{itemize}
        \item \textbf{where} $\{\#\} = \text{add-mset a M0}$
        \item \text{and} $\{\#\} = M0 + K$
        \item \text{and} $\forall b. \ b \in \# \ K \rightarrow (b, a) \in R$
      \end{itemize}
      \item \textbf{unfolding mult1-def by blast}
      \item \textbf{then show} $(x, y) \in R$ \text{by (auto simp: add-eq-conv-diff)}
    \end{itemize}
\end{itemize}
\textbf{qed}

\textbf{lemma} mulex1-singleton [iff]:
\begin{itemize}
  \item mulex1 $P \{\#\} \{\#\} \leftrightarrow P x y$
\end{itemize}
\textbf{using} \text{mult1-singleton [of $x y \{(x, y), P x y\}$] by (simp add: mulex1-def)}

\textbf{lemma} singleton-mulex-onI:
\begin{itemize}
  \item $P x y \Rightarrow x \in A \Rightarrow y \in A \Rightarrow \text{mulex-on} P A \{\#\} \{\#\}$
\end{itemize}
\textbf{by (auto simp: mulex-on-def)}
lemma reflclp-mulex-on-add-right [simp]:
assumes \( \text{mulex-on } P A \Rightarrow M N \) and \( M \in \text{multisets } A \) and \( a \in A \)
shows \( \text{mulex-on } P A M (N + \{\#a\}) \)
using assms by (cases \( M = N \)) simp-all

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

lemma mulex-on-union-right [simp]:
assumes \( \text{mulex-on } P F A B \) 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 \( a \in F \) 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

lemma mulex-on-union-right' [simp]:
assumes \( \text{mulex-on } P F A B \) and \( K \in \text{multisets } F \)
shows \( \text{mulex-on } P F A (B + K) \)
using mulex-on-union-right [OF assms] by (simp add: ac-simps)

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

lemma accessible-on-mulex1-multisets:
assumes \( \text{wf } : \langle P A \rangle \) shows \( \forall M \in \text{multisets } A. \text{accessible-on } \langle \text{mulex1 } P \rangle (\text{multisets } A) M \)
proof
let \( ?P = \text{mulex1 } P \)
let \( ?A = \text{multisets } A \)
let \( ?\text{acc} = \text{accessible-on } ?P \) ?A
{
fix \( M M0 a \)
assume \( M0: ?\text{acc } M0 \)
and \( a \in A \)
and \( M0 \in ?A \)
and wfhyp: \( \forall b. [b \in A; P b a] \Rightarrow (\forall M. ?\text{acc } (M) \rightarrow ?\text{acc } (M + \{\#b\}) \))
and acc-hyp: \( \forall M. M \in ?A \land \forall P M M0 \rightarrow ?\text{acc } (M + \{\#a\}) \))
then have \( \text{add-mset a } M0 \in ?A \) by (auto simp: multisets-def)
then have \( ?\text{acc } (\text{add-mset a } M0) \)
proof (rule accessible-onI [of \( \text{add-mset a } M0 \)])
fix \( N \)
assume \( N \in ?A \)
and \( ?P N (\text{add-mset a } M0) \)
then have \( (\exists M. M \in ?A \\ \forall P M M0 \land N = M + \{\#a\}) \lor \)

53
(∃ K. (∀ b ∈ # K → P b a) ∧ N = M₀ + K))

using mulex1-add [of P N M₀ a] by (auto simp: multisets-def)
then show ?acc (N)
proof (elim exE disjE conjE)
  fix M assume M ∈ ?A and ?P M M₀ and N: N = M + {#a#}
  from acc-hyp have M ∈ ?A ∧ ?P M M₀ → ?acc (M + {#a#}) ..
  with M ∈ ?A and ?P M M₀ have ?acc (M + {#a#}) by blast
  then show ?acc (N) by (simp only: N)
next
  fix K
  assume N: N = M₀ + K
  assume ∀ b ∈ # K → P b a
  moreover from N and (N ∈ ?A) have K ∈ ?A by (auto simp: multisets-def)
  ultimately have ?acc (M₀ + K)
  proof (induct K)
    case empty
    from M₀ show ?acc (M₀ + {#}) by simp
  next
    case (add x K)
    from add.prems have x ∈ A and P x a by (auto simp: multisets-def)
    with wf-hyp have ∀ M. ?acc M → ?acc (M + {#x#}) by blast
    moreover from add have ?acc (M₀ + K) by (auto simp: multisets-def)
    ultimately show ?acc (M₀ + (add-mset x K)) by simp
  qed
  then show ?acc N by (simp only: N)
  qed
qed

} note tedious-reasoning = this

fix M
assume M ∈ ?A
then show ?acc M
proof (induct M)
  show ?acc {#}
  proof (rule accessible-onI)
    show {#} ∈ ?A by (auto simp: multisets-def)
  next
    fix b assume ?P b {#} then show ?acc b by simp
  qed
next
  case (add a M)
  then have ?acc M by (auto simp: multisets-def)
  from add have a ∈ A by (auto simp: multisets-def)
  with wf have ∀ M. ?acc M → ?acc (add-mset a M)
  proof (induct)
    case (less a)
    then have r: ∃ b. [b ∈ A; P b a] → (∀ M. ?acc M → ?acc (M + {#b#}))
    by auto
    show ∀ M. ?acc M → ?acc (add-mset a M)

54
proof (intro allI impI)
  fix M'
  assume ?acc M'
moreover then have M' ∈ ?A by (blast dest: accessible-on-imp-mem)
ultimately show ?acc (add-mset a M')
    by (induct) (rule tedious-reasoning [OF - ⟨a ∈ A⟩ - r], auto)
qed
qed
with ⟨?acc (M)⟩ show ?acc (add-mset a M) by blast
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 (mulex-on P A) (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 ⇒ 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 ∈ multisets A
  shows mulex-on P A (K + M) (K + N)
  using 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) ∈ multisets A
  and (K + N) ∈ multisets A by (auto simp: multisets-def)
ultimately have restrict-to (mulex1 P) (multisets A) (K + M) (K + N) by auto
then show ?case by (auto simp: mulex-on-def)
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-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-union)

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)
by (auto intro: union-mulex-on-mono mulex-on-union')

lemma mult1-mono:
assumes A x y. [x ∈ A; y ∈ A; (x, y) ∈ R] ⇒ (x, y) ∈ S
and M ∈ multisets A
and N ∈ multisets A
and $(M, N) \in \text{mult1 } R$
shows $(M, N) \in \text{mult1 } S$
using \text{assms unfolding mult1-def multisets-def}
by \text{auto (metis (full-types) subsetD)}

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

\text{lemma mulex-on-mono:}
\begin{align*}
\text{assumes } & \ast: \forall x y. \left[ x \in A; y \in A; P x y \right] \implies Q x y \\
& \text{and } \text{mulex-on } P A M N \\
\text{shows } \text{mulex-on } Q A M N \\
\text{proof –}
\begin{align*}
& \text{let } \text{rel } = \lambda P. \text{(restrict-to (mulex1 } P \text{) (multisets } A\}) \\
& \text{from } \langle \text{mulex-on } P A M N \rangle \text{ have } (\text{rel } P)^++ M N \text{ by (simp add: mulex-on-def)} \\
& \text{then have } (\text{rel } Q)^++ M N \\
& \text{proof (induct rule: tranclp.induct)} \\
& \text{case } \langle \text{r-into-trancl } M N \rangle \\
& \text{then have } M \in \text{multisets } A \text{ and } N \in \text{multisets } A \text{ by auto} \\
& \text{from } \text{mulex1-mono [OF } \ast \text{ this] and r-into-trancl} \\
& \text{show } \text{?case by auto} \\
& \text{next} \\
& \text{case } \langle \text{trancl-into-trancl } L M N \rangle \\
& \text{then have } M \in \text{multisets } A \text{ and } N \in \text{multisets } A \text{ by auto} \\
& \text{from } \text{mulex1-mono [OF } \ast \text{ this] and trancl-into-trancl} \\
& \text{have } \text{rel } Q M N \text{ by auto} \\
& \text{with } (\text{rel } Q)^++ L M; \text{ show } \text{?case by (rule tranclp.trancl-into-trancl)} \\
& \text{qed} \\
& \text{then show } \text{?thesis by (simp add: mulex-on-def)} \\
& \text{qed}
\end{align*}
\text{lemma mult1-reflcl:}
\begin{align*}
\text{assumes } & (M, N) \in \text{mult1 } R \\
\text{shows } & (M, N) \in \text{mult1 } (R^=) \\
\text{using } & \text{assms by (auto simp: mult1-def)}
\end{align*}

\text{lemma mulex1-reflclp:}
\begin{align*}
\text{assumes } & \text{mulex1 } P M N \\
\text{shows } & \text{mulex1 } (P^=) M N \\
\text{using } & \text{mulex1-mono [of } \text{UNIV } P \text{ P } \ast \text{ M N, OF } \ast \text{ - - } \text{assms]} \\
& \text{by (auto simp: multisets-def)}
\end{align*}

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

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

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

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

\textbf{lemma} non-empty-multiset-induct [consumes 1, case-names singleton add]:
\textbf{assumes} \(M \neq \{\#\}\)
\textbf{and} \(\forall x. P \{\#x\}\)
\textbf{and} \(\forall x M. P M \Rightarrow P (\text{add-mset } x M)\)
\textbf{shows} \(P M\)
\textbf{using} assms by (induct M) auto

\textbf{lemma} mulex-on-all-strict:
\textbf{assumes} \(X \neq \{\#\}\)
\textbf{assumes} \(X \in \text{multisets } A\) and \(Y \in \text{multisets } A\)
\textbf{and} \(*: \forall y. y \in \# Y \rightarrow (\exists x. x \in \# X \land P y x)\)
\textbf{shows} mulex-on P A Y X
\textbf{using} assms
\textbf{proof (induction } X\ \text{arbitrary: } Y\ \text{rule: } \text{non-empty-multiset-induct})
  case (singleton x)
  then \textbf{have} \(\text{mulex1 } P Y \{\#x\}\)
  unfolding mulex1-def mult1-def
  by auto
  with singleton \textbf{show} ?case \textbf{by} (auto simp: mulex-on-def)
next
\begin{verbatim}
case (add x M)
let \( \mathcal{Y} = \{ \#y \in \mathcal{Y} \mid \exists x. x \in \# M \land P y x \} \)
let \( \mathcal{Z} = \mathcal{Y} - \mathcal{Y} \)

have \( \mathcal{Y} : \mathcal{Y} = \mathcal{Z} + \mathcal{Y} \) by (metis multiset-partition union-multisets-iff)

ultimately have mulex-on P A \( \mathcal{Y} \) M using add by blast
ultimately have mulex-on P A \( \mathcal{Z} \) \{\#x\} using add by blast
ultimately have mulex-on P A \( \mathcal{Z} \) \{\#x\} unfolding mulex1-def mult1-def by blast
ultimately have \( \mathcal{Z} \) \{\#x\} \in multisets A by auto
ultimately have \( \mathcal{Z} \) \{\#x\} \in multisets A using ⟨ Y \in multisets A ⟩ by (metis diff-union-cancelL union-multisetsD)
ultimately show \( \mathfrak{thesis} \) by (auto simp: mulex-on-def)
qed

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

\textbf{lemma} \texttt{diff-set-Ex-iff}:
\( X \neq \{\#\} \land X \subseteq \# M \land N = (M - X) + Y \leftrightarrow X \neq \{\#\} \land (\exists Z. M = Z + X \land N = Z + Y) \);

by (auto) (metis add-diff-cancel-left' multiset-diff-union-assoc union-commute)

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

\textbf{lemma} \texttt{mulex-on-alt-def}:
\[\text{assumes trans: } \texttt{trans-on P A} \]
\[\text{shows } \texttt{mulex-on P A M N } \leftrightarrow \text{ } M \in \texttt{multisets A } \land \text{ N } \in \texttt{multisets A } \land (\exists X Y Z. X \neq \{\#\} \land N = Z + X \land M = Z + Y \land (\forall y. y \in \# Y \rightarrow (\exists x. x \in \# X \land P y x)) \)
\[\text{is } \texttt{?P M N } \leftrightarrow \texttt{?Q M N} \]

\textbf{proof}
\[\text{assume } \texttt{?P M N then show } \texttt{?Q M N} \]
\[\text{proof } (\texttt{induct M N}) \]
\[\text{case } \texttt{base M N} \]
\[\text{then obtain } a \texttt{ M0 K where } N: N = M0 + \{\#a\} \]
\[\text{and } M: M = M0 + K \]
\[\text{and } \ast: \forall b. b \in \# K \rightarrow P b a \]
\end{verbatim}
and \( M \in \text{multisets} \ A \) and \( N \in \text{multisets} \ A \) by (auto simp: mulex1-def mult1-def)

moreover have \( \{\#\#\}\in \text{multisets} \ A \) and \( K \in \text{multisets} \ A \) by auto
moreover have \( \{\#\#\} \neq \{\#\} \) by auto
moreover have \( N = M0 + \{\#\#\} \) by fact
moreover have \( M = M0 + K \) by fact
moreover have \( \forall y. y \in \# K \longrightarrow (\exists x. x \in\# \{\#\#\} \land P \ y \ x) \) using \* by auto
ultimately show \( \text{case by blast} \)

next
case \( \text{(step } L \ M \ N) \)
then obtain \( X \ Y \ Z \)
where \( L \in \text{multisets} \ A \) and \( M \in \text{multisets} \ A \) and \( N \in \text{multisets} \ A \)
and \( X \in \text{multisets} \ A \) and \( Y \in \text{multisets} \ A \)
and \( M' = M + X \)

\( L : L = Z + Y \) and \( X \neq \{\#\} \)

and \( Y : \forall y. y \in \# Y \longrightarrow (\exists x. x \in\# \ X \land P \ y \ x) \)
and \( \text{mulex1} \ P \ M \ N \)
by blast
from \( \text{mulex1} \ P \ M \ N \) obtain \( a \)
where \( N : N = \text{add-mset} \ a \) and \( M0 \)
and \( \ast : \forall b. b \in \# K \longrightarrow P \ b \ a \) unfolding \( \text{mulex1-def} \) \( \text{mult1-def} \) by blast
have \( L' : L = (M - X) + Y \) by (simp add: \( L \ M \))
have \( K : \forall y. y \in \# K \longrightarrow (\exists x. x \in\# \{\#\#\} \land P \ y \ x) \) using \* 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 \( \ast \ X = \text{add-mset} \ a \ (X - K) \)
let \( \ast \ Y = (K - X) + Y \)

have \( L \in \text{multisets} \ A \) and \( N \in \text{multisets} \ A \) by fact+
m
moreover have \( \ast \ X \neq \{\#\} \) and \( (\exists Z. N = Z + \ast \ X \land L = Z + \ast \ Y) \)

proof –
have \( \ast \ X \neq \{\#\} \) by auto
moreover have \( \ast \ X \subseteq \# N \)
using \( M \ N \ M' \) by (simp add: \( \text{add-commute} \) \mid \{\#\#\}\)
(mmetis \( \text{Multiset.diff-subset-eq-self} \) \( \text{add-commute} \) \( \text{add-diff-cancel-right} \))
m
moreover have \( L = (N - \ast \ X) + \ast \ Y \)

proof (rule \( \text{multiset-eqI} \))

fix \( x :: \ast \ a \)
let \( \ast \ c = \lambda M. \ \text{count} \ M \ x \)
let \( \ast \ ic = \lambda x. \ \text{int} \ (\ast \ x) \)
from \( \ast \ X \subseteq \# N \) have \( \ast : \ast \ X \neq \{\#\#\} + \ast \ (X - K) \leq \ast \ N \)
by (auto simp add: \( \text{subseteq-mset-def} \) \( \text{split: if-splits} \))
from \( \ast \ X \neq \{\#\#\} \) unfolding \( N \) by (auto split: if-splits)

have \( \ast \ ic \ N = \ast \ ic (N - \ast \ X + \ast \ Y) = \text{int} \ (\ast \ ic N \ - \ast \ ic \ ?X) + \ast \ ic \ ?Y \) by simp
also have \( \ast \ ic \ N \ - \ast \ ic (\ast \ ?X) + \ast \ ic (X - K)) + \ast \ ic (K - X) + \ast \ ic \ Y \)

60
using of-nat-diff [OF *] by simp
also have \ldots = (\text{?ic } N - \text{?ic } \{\text{#a}\}) - \text{?ic } (X - K) + \text{?ic } (K - X) +
\text{?ic } Y by simp
also have \ldots = (\text{?ic } N - \text{?ic } \{\text{#a}\}) + (\text{?ic } K - \text{?ic } X) + \text{?ic } Y by simp
also have \ldots = (\text{?ic } N - \text{?ic } \{\text{#a}\}) + \text{?ic } Y by simp
also have \ldots = \text{?ic } L

unfolding \text{L'} \text{M'} \text{N}
using \ast\ast by (simp add: algebra-simps)
finally show \text{?c } L = \text{?c } (N - X + Y) by simp
qed
ultimately show \text{?thesis} by (metis diff-set-Ex-iff)
qed
moreover have \forall y. y \in \# Y \longrightarrow (\exists x. x \in \# X \land P y x)
proof (intro allI impI)
fix y assume y \in \# Y
then have y \in \# K - X \lor y \in \# Y by auto
then show \exists x. x \in \# X \land P y x
proof
assume y \in \# K - X
then show y \in \# K by (rule in-diffD)
with K show \text{?thesis} by auto
next
assume y \in \# Y
with Y obtain x where x \in \# X and P y x by blast
{ assume x \in \# X - K with \langle P y x \rangle have \text{?thesis} by auto }
moreover
{ assume x \in \# K with \ast have P x a by auto
moreover have y \in A using \langle Y \in \text{multisets } A \rangle and \langle y \in \# Y \rangle by (auto simp: multisets-def)
moreover have a \in A using \langle N \in \text{multisets } A \rangle by (auto simp: N)
moreover have x \in A using \langle M \in \text{multisets } A \rangle and \langle x \in \# K \rangle by (auto simp: M' multisets-def)
ultimately have P y a using \langle P y x \rangle and trans unfolding transp-on-def
by blast
then have \text{?thesis} by force }
moreover from \langle x \in \# X \rangle have x \in \# X - K \lor x \in \# K
by (auto simp add: in-diff-count not-in-iff)
ultimately show \text{?thesis} by auto
qed
qed
ultimately show \text{?case} by blast
qed
next
assume \?Q M N
then obtain X Y Z where M \in \text{multisets } A and N \in \text{multisets } A
and X \neq \{\#\} and N: N = Z + X and M: M = Z + Y
and \ast\ast: \forall y. y \in \# Y \longrightarrow (\exists x. x \in \# X \land 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
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 ?P = (mulex-on P A) ==
from almost-full-on-hom [OF almost-full-on-lists, of A P ?P mset, OF list-emb-imp-reflclp-mulex-on, simplified]
show ?thesis using assms by blast
qed

lemma wqo-on-multisets:
assumes \textit{wqo-on }$P \ A$

shows \textit{wqo-on} $(\text{mulex-on } P \ A)^== (\text{multisets } A)$

proof

from \textit{transp-on-mulex-on} [of $P \ A$ multisets $A$]

show \textit{transp-on} $(\text{mulex-on } P \ A)^== (\text{multisets } A)$

unfolding \textit{transp-on-def} by \textit{blast}

next

from \textit{almost-full-on-multisets} [OF \text{assms} [\text{THEN wqo-on-imp-almost-full-on}]]

show \textit{almost-full-on} $(\text{mulex-on } P \ A)^== (\text{multisets } A)$.  

qed

end

References