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 Binary Predicates Restricted to Elements of a Given Set 2
1.1 Measures on Sets (Instead of Full Types) . . . . . . . . . . . 9
1.2 Facts About Predecessor Sets . . . . . . . . . . . . . . . . . 15

2 Constructing Minimal Bad Sequences 16

3 Enumerations of Well-Ordered Sets in Increasing Order 21

4 Almost-Full Relations 23
4.1 Basic Definitions and Facts . . . . . . . . . . . . . . . . . . . 23
4.2 Adding a Bottom Element to a Set . . . . . . . . . . . . . . . 27
4.3 Adding a Bottom Element to an Almost-Full Set . . . . . . . 27
4.4 Disjoint Union of Almost-Full Sets . . . . . . . . . . . . . . . 28
4.5 Dickson’s Lemma for Almost-Full Relations . . . . . . . . . . 30
4.6 Higman’s Lemma for Almost-Full Relations . . . . . . . . . . 30
4.7 Special Case: Finite Sets . . . . . . . . . . . . . . . . . . . . . 33
4.8 Natural Numbers . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.9 Further Results . . . . . . . . . . . . . . . . . . . . . . . . . . 34

*The research was funded by the Austrian Science Fund (FWF): J3202.
5 Well-Quasi-Orders 36
5.1 Basic Definitions 37
5.2 Equivalent Definitions 37
5.3 A Type Class for Well-Quasi-Orders 39
5.4 Dickson’s Lemma 40
5.5 Higman’s Lemma 41

6 Kruskal’s Tree Theorem 43

7 Instances of Well-Quasi-Orders 49
7.1 The Option Type is Well-Quasi-Ordered 50
7.2 The Sum Type is Well-Quasi-Ordered 50
7.3 Pairs are Well-Quasi-Ordered 50
7.4 Lists are Well-Quasi-Ordered 50

8 Multiset Extension of Orders (as Binary Predicates) 51

9 Multiset Extension Preserves Well-Quasi-Orders 63

1 Binary Predicates Restricted to Elements of a Given Set

theory Restricted-Predicates
imports Main
begin

definition restrict-to :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ (′a ⇒ ′a ⇒ bool) where
  restrict-to P A = (λx y. x ∈ A ∧ y ∈ A ∧ P x y)

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

definition transp-on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
  transp-on P A ←→ (∀x∈A. ∀y∈A. ∀z∈A. P x y ∧ P y z → P x z)

definition total-on :: (′a ⇒ ′a ⇒ bool) ⇒ ′a set ⇒ bool where
  total-on P A ←→ (∀x∈A. ∀y∈A. x = y ∨ P x y ∨ P y x)

abbreviation strict P ≡ λx y. P x y ∧ ¬(P y x)

abbreviation chain-on P f A ≡ ∀i. f i ∈ A ∧ P (f i) (f (Suc i))

abbreviation incomparable P ≡ λx y. ¬P x y ∧ ¬P y x

abbreviation antichain-on P f A ≡ ∀(i::nat) j. f i ∈ A ∧ (i < j → incomparable P (f i) (f j))
lemma strict-reflclp-conv [simp]:
strict \((P^=)\) = strict P by auto

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

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

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

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

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

lemma reflp-on-converse-simp [simp]:
reflp-on P\(^{-1}\) A \longleftrightarrow reflp-on P A
by (auto simp: reflp-on-def)

lemma transp-on-converse:
transp-on P A \Rightarrow transp-on P\(^{-1}\) A
unfolding transp-on-def by blast

lemma transp-on-converse-simp [simp]:
transp-on P\(^{-1}\) A \longleftrightarrow transp-on P A
unfolding transp-on-def by blast

lemma transp-on-reflclp:
transp-on P A \Rightarrow transp-on \((P^=)\) A
unfolding transp-on-def by blast

lemma transp-on-strict:
transp-on P A \Rightarrow transp-on (strict P) A
unfolding transp-on-def by blast

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

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

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

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

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

- **assumes** \(\land Q x. [x \in A; (\land y. [y \in A; \land x. [x \in A; P x y] \implies Q x]] \implies Q y)] \implies Q x\)
- **shows** \(\text{inductive-on } P A\)
- **using** \(\text{assms unfolding inductive-on-def by metis}\)

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

**Lemma** \(\text{wfp-on-imp-minimal}:\)

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

  **then have** \(\forall i. ?S i \in A\) **using** \(*\text{**}\) by blast

  **moreover have** \(\forall i. P (?S (\text{Suc } i)) (?S i)\)
- **proof**
  - **fix** \(i\) **show** \(P (?S (\text{Suc } i)) (?S i)\)
  - **by** (induct \(i\)) (auto simp: \(*\text{** ***}\))
- **qed**

  **ultimately have** \(\forall i. ?S i \in A \land P (?S (\text{Suc } i)) (?S i)\) by blast
  **with** \(\text{assms(1)}\) **show** \(\text{False}\)
- **unfolding** \(\text{wfp-on-def}\) **by** fast
- **qed**

**Lemma** \(\text{minimal-imp-inductive-on}:\)

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

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

lemma inductive-on-induct [consumes 2, case-names less, induct pred: inductive-on]:
assumes inductive-on \( P A \) and \( x \in A \)
and \( \forall y. [ y \in A; \forall x. [ x \in A; P x y ] \Rightarrow Q x ] \Rightarrow Q y \)
shows \( Q x \)
using assms unfolding inductive-on-def by metis

lemma inductive-on-imp-wfp-on:
assumes inductive-on \( P A \)
shows \( wfp-on P A \)
proof –
let \( ?Q = \lambda x. \neg (\exists f. f 0 = x \land (\forall i. f i \in A \land P (f (Suc i)) (f i))) \)
{ fix \( x \) assume \( x \in A \)
  with assms have \( \neg Q x \)
  proof (induct rule: inductive-on-induct)
    fix \( y \) assume \( y \in A \) and \( IH: \forall x. x \in A \Rightarrow P x y \Rightarrow \neg Q x \)
    show \( \neg Q y \)
    proof (rule ccontr)
      assume \( \neg Q y \)
      then obtain \( f \) where \( \star: f 0 = y \)
      \( \forall i. f i \in A \land P (f (Suc i)) (f i) \) by auto
      then have \( P (f (Suc 0)) (f 0) \) and \( f (Suc 0) \in A \) by auto
      with \( IH \) and \( \star \) have \( \neg Q (f (Suc 0)) \) by auto
      with \( \star \) show \( False \) by auto
    qed
  qed }
then show \( \neg \thesis unfolding wfp-on-def by blast 
qed

definition antisymp-on :: \( ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a set \Rightarrow bool \)
where
antisymp-on \( P A \) \( \longleftrightarrow \forall a \in A. \forall b \in A. (P a b \land P b a \Rightarrow a = b) \)
lemma antisym-onI [Pure.intro]:
\((\forall a. b. a \in A; b \in A; P a b; P b a) \implies a = b) \implies antisym-on P A
by (auto simp: antisym-on-def)

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

definition qo-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool where
qo-on P A = reflp-on P A ∧ transp-on P A

definition irreflp-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool where
irreflp-on P A = (∀a∈A. ~P a a)

definition po-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool where
po-on P A = (irreflp-on P A ∧ transp-on P A)

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

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

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

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

lemma po-on-converse-simp [simp]:
po-on P⁻¹⁻¹ A = po-on P A
by (simp add: po-on-def)

lemma po-on-imp-go-on:
po-on P A ⇒ qo-on (P==) A
unfolding po-on-def qo-on-def by (metis reflp-on-refclp transp-on-refclp)

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

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

lemma irreflp-on-subset:
assumes \( A \subseteq B \) and \( \text{irreflp-on} \ P \ B \)
shows \( \text{irreflp-on} \ P \ A \)
using assms by (auto simp: irreflp-on-def)

lemma po-on-subset:
assumes \( A \subseteq B \) and \( \text{po-on} \ P \ B \)
shows \( \text{po-on} \ P \ A \)
using transp-on-subset and irreflp-on-subset and assms
unfolding po-on-def by blast

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

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

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

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

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

lemma qo-on-subset:
$A \subseteq B \Rightarrow qo-on \ P \ B \Rightarrow qo-on \ P \ A$

unfolding qo-on-def
using reflp-on-subset
and transp-on-subset by blast

Quasi-orders are instances of the preorder class.

lemma qo-on-UNIV-conv:
$qo-on \ P \ \text{UNIV} \longleftrightarrow \ \text{class.preorder} \ P \ (\text{strict} \ P) \ (\text{is} \ ?\text{lhs} = ?\text{rhs})$

proof
assume ?lhs then show ?rhs
  unfolding qo-on-def class.preorder-def
  using qo-on-imp-reflp-on [of P UNIV]
  and qo-on-imp-transp-on [of P UNIV]
  by (auto simp: reflp-on-def)
next
assume ?rhs then show ?lhs
  unfolding class.preorder-def
  by (auto simp: qo-on-def)
qed

lemma wfp-on-iff-inductive-on:
wfp-on $P \ A \longleftrightarrow \ \text{inductive-on} \ P \ A$

by (blast intro: inductive-on-imp-wfp-on wfp-on-imp-inductive-on)

lemma wfp-on-iff-minimal:
wfp-on $P \ A \longleftrightarrow (\forall Q \ x. \ x \in Q \land Q \subseteq A \Rightarrow (\exists z \in Q. \ \forall y. \ P y z \Rightarrow y \notin Q))$

using wfp-on-imp-minimal [of P A]
and minimal-imp-inductive-on [of A P]
and inductive-on-imp-wfp-on [of P A]
by blast

Every non-empty well-founded set $A$ has a minimal element, i.e., an element that is not greater than any other element.

lemma wfp-on-imp-has-min-elt:
assumes wfp-on $P \ A \ and \ A \neq \ \{\}$
shows $\exists x \in A. \ \forall y \in A. \ P y x$

using assms unfolding wfp-on-iff-minimal by force

lemma wfp-on-induct [consumes 2, case-names less, induct pred: wfp-on]:
assumes wfp-on $P \ A \ and \ x \in A$
and $\forall y. \ y \in A. \ \forall x. \ [x \in A; \ P x y] \Longrightarrow Q x \Longrightarrow Q y$
shows $Q x$

using assms and inductive-on-induct [of P A x]
unfolding wfp-on-iff-inductive-on by blast

lemma wfp-on-UNIV [simp]:
  wfp-on P UNIV ←→ wfP P
unfolding wfp-on-iff-inductive-on inductive-on-def wfP-def wf-def by force

1.1 Measures on Sets (Instead of Full Types)

definition
  inv-image-betw ::
    ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ ('a ⇒ bool)
where
  inv-image-betw P f A B = (λx y. x ∈ A ∧ y ∈ A ∧ f x ∈ B ∧ f y ∈ B ∧ P (f x) (f y))

definition
  measure-on :: ('a ⇒ nat) ⇒ 'a set ⇒ 'a ⇒ 'a ⇒ bool
where
  measure-on f A = inv-image-betw (op <) f A UNIV

lemma in-inv-image-betw [simp]:
  inv-image-betw P f A B x y ←→ x ∈ A ∧ y ∈ A ∧ f x ∈ B ∧ f y ∈ B ∧ P (f x) (f y)
  by (auto simp: inv-image-betw-def)

lemma in-measure-on [simp, code-unfold]:
  measure-on f A x y ←→ x ∈ A ∧ y ∈ A ∧ f x < f y
  by (simp add: measure-on-def)

lemma wfp-on-inv-image-betw [simp, intro!]:
  assumes wfp-on P B
  shows wfp-on (inv-image-betw P f A B) A (is wfp-on ?P A)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain g where ∀ i. g i ∈ A ∧ ?P (g (Suc i)) (g i) by (auto simp: wfp-on-def)
  with assms show False by (auto simp: wfp-on-def)
qed

lemma wfp-on-less:
  wfp-on (op <) (UNIV :: nat set)
  using wfp-less by (auto simp: wfP-def)

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

lemma wfp-on-mono:
\[ A \subseteq B \implies (\forall x \ y. \ x \in A \implies y \in A \implies P \ x \ y \implies Q \ x \ y) \implies \operatorname{wfp-on} \ Q \ B \implies \operatorname{wfp-on} \ P \ A \]

**Lemma**: \( \operatorname{wfp-on-subset} \):
\[ A \subseteq B \implies \operatorname{wfp-on} \ P \ B \implies \operatorname{wfp-on} \ P \ A \]
using \( \operatorname{wfp-on-mono} \) by blast

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

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

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

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

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

**Lemma**: \( \operatorname{irreflp-on-map} \)
assumes \( \operatorname{irreflp-on} \ Q \ B \)
and \( h \ ' \ A \subseteq B \)
shows \( \operatorname{irreflp-on} \ (\lambda x \ y. \ Q \ (h x) \ (h y)) \ A \)
using \( \text{assms unfolding \ operatorname{irreflp-on-def}} \) by auto

**Lemma**: \( \operatorname{po-on-map} \)
assumes \( \operatorname{po-on} \ Q \ B \)
and \( h \ ' \ A \subseteq B \)
shows \( \operatorname{po-on} \ (\lambda x \ y. \ Q \ (h x) \ (h y)) \ A \)
using \( \text{assms and \ operatorname{transp-on-map} and \ operatorname{irreflp-on-map}} \)
unfolding \( \text{po-on-def} \) by auto

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

lemma \( \text{wfp-on-imp-irreflp-on:} \)
assumes \( \text{wfp-on } P A \)
shows \( \text{irreflp-on } P A \)
proof
  fix \( x \)
  assume \( x \in A \)
  show \( \neg P x x \)
  proof
    let \( \text{if } = \lambda_. \ x \)
    assume \( P x x \)
    then have \( \forall i. P (\text{if } (\text{Suc } i)) (\text{if } i) \) by blast
    with \( x \in A \) have \( \neg \text{wfp-on } P A \) by (auto simp: wfp-on-def)
    with assms show False by contradiction
  qed
qed

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

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

lemma \( \text{accessible-on-induct [consumes 1, induct pred: accessible-on]:} \)
assumes \( *: \text{accessible-on } P A a \)
and IH: \( \forall x. [\text{accessible-on } P A x; \forall y. [y \in A; P y x] \implies Q y] \implies Q x \)

shows Q a
by (rule * [THEN accessible-on.induct]) (auto intro: IH accessible-onI)

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

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

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

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

**lemma wfp-on-translp-imp-wfp-on:**
assumes wfp-on (P++) A
shows wfp-on P A
by (rule ccontr) (insert assms, auto simp: wfp-on-def)

**lemma inductive-on-imp-accessible-on:**
assumes inductive-on P A
shows \( \forall x \in A. \text{accessible-on } P A x \)
proof
fix x
assume \( x \in A \)
with assms show accessible-on P A x
by (induct) (auto intro: accessible-onI)
qed

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

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

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

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

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

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

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

lemma stepfun-imp-tranclp:
assumes \(f 0 = x\) and \(f (\text{Suc } n) = z\)
and \(\forall i \leq n. P (f i) (f (\text{Suc } i))\)
shows \(P^{++} x z\)
using assms
by (induct n arbitrary: \(x z\))
  (auto intro: tranclp.trancl-into-trancl)

lemma tranclp-imp-stepfun:
assumes \(P^{++} x z\)
shows \(\exists f n. f 0 = x \land f (\text{Suc } n) = z \land (\forall i \leq n. P (f i) (f (\text{Suc } i)))\)
(is \(\exists f n. ?P x z f n\))
using assms
proof (induct rule: tranclp-induct)
case (base y)
  let \(?f = (\lambda y. 0 := x)\)
  have \(?f 0 = x\) and \(?f (\text{Suc } 0) = y\) by auto
  moreover have \(\forall i \leq 0. P (?f i) (?f (\text{Suc } i))\)
    using base by auto
  ultimately show \(?case by blast\)
next
case (step y z)
  then obtain \(f n\) where IH: \(?P x y f n\) by blast
  then have \(\forall i \leq n. P (f i) (f (\text{Suc } i))\)
    and \([\text{simp}]: f 0 = x f (\text{Suc } n) = y\)
    by auto
  let \(?n = \text{Suc } n\)
  let \(?f = f (\text{Suc } ?n := z)\)
  have \(?f 0 = x\) and \(?f (\text{Suc } ?n) = z\) by auto
  moreover have \(\forall i \leq ?n. P (?f i) (?f (\text{Suc } i))\)
    using \(P y z\) and \(\ast\) by auto
  ultimately show \(?case by blast\)
qed
lemma tranclp-stepfun-conv:
\[ P^{++} \ x \ y \iff (\exists \ f \ n. \ f \ 0 = x \land f \ (Suc \ n) = y \land (\forall i \leq n. \ P \ (f \ i) \ (f \ (Suc \ i)))) \]
using tranclp-imp-stepfun and stepfun-imp-tranclp by metis

1.2 Facts About Predecessor Sets

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

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

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

lemma restrict-to-rtranclp:
assumes transp-on P A
and x \in A and y \in A
shows (restrict-to P A)** x y \iff P** x y
proof –
\{ assume (restrict-to P A)** x y
then have P** x y using assms
by (induct) (auto, unfold transp-on-def, blast) \}
with assms show ?thesis by auto
qed

lemma reflp-on-restrict-to-rtranclp:
assumes reflp-on P A and transp-on P A
and x \in A and y \in A
shows (restrict-to P A)** x y \iff P x y
unfolding restrict-to-rtranclp [OF assms(2−)]
unfolding reflp-on-reflclp-simp [OF assms(1, 3−)] ..

end
2 Constructing Minimal Bad Sequences

theory Minimal-Bad-Sequences
imports Restricted-Predicates
begin

The set of all infinite sequences over elements from \( A \).

definition \( SEQ \ A = \{ f :: \mathbb{N} \Rightarrow 'a. \forall i. f i \in A \} \)

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

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

definition \textit{good} :: \( ('a \Rightarrow 'a) \Rightarrow (\mathbb{N} \Rightarrow 'a) \Rightarrow \text{bool} \) 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 \textit{bad}.

abbreviation \textit{bad} \( P \ f \equiv \neg \text{good} \ P \ f \)

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

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

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

A locale capturing the construction of minimal bad sequences over values from \( A \). Where minimality is to be understood w.r.t. \textit{size} of an element.

locale mbs =
fixes \( A :: ('a :: \text{size}) \text{ set} \)
begin

Since the \textit{size} is a well-founded measure, whenever some element satisfies a property \( P \), then there is a size-minimal such element.

lemma \textit{minimal}:
\( \text{assumes} \ x \in A \ \text{and} \ P \ x \)
\( \text{shows} \ \exists y \in A. \ \text{size} \ y \leq \text{size} \ x \land P \ y \land (\forall z \in A. \ \text{size} \ z < \text{size} \ y \Rightarrow \neg P \ z) \)
using assms
proof (induction \( x \) taking: size rule: measure-induct)
\( \text{case} (1 \ x) \)
then show \( \vdash \)
proof \((\text{cases } \forall y \in A. \text{ size } y < \text{ size } x \rightarrow \neg P y)\)

\text{case } True
with \(I\) show \(?thesis\) by blast
next
\text{case } False
then obtain \(y\) where \(y \in A\) and \(\text{size } y < \text{ size } x\) and \(P y\) by blast
with \(I\,IH\) show \(?thesis\) by (fastforce elim!: order-trans)
qed

\text{qed}

\text{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 \(A\).

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

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

A partial order on infinite bad sequences.

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

The strict part of the above order.

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

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

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

\text{lemma} \ geseqE:
assumes \((f, g) \in \text{geseq} \land \big[\forall i. f i \in A; \forall i. g i \in A; f = g\big] \Longrightarrow Q\)
and \(\bigwedge i. [\forall i. f i \in A; \forall i. g i \in A; \text{size } (g i) < \text{size } (f i); \forall j < i. f j = g j]\) \\
\Longrightarrow Q
shows \( Q \)
using assms by (auto simp: gseq_iff)

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

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

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

lemma ithI [intro]:
\( f \in B \Rightarrow f i = x \Rightarrow x \in \text{ith} B i \)
by (auto simp: ith_def)

lemma ithE [elim]:
\( x \in \text{ith} B i ; \ \forall f. \forall f. f i = x \Rightarrow Q \) \[ \Rightarrow Q \]
by (auto simp: ith_def)

lemma ith-conv:
\( x \in \text{ith} B i \ \longleftrightarrow (\exists f. f. x = f i) \)
by auto

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

A minimal element (w.r.t. size) from a set.

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

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

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

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

end

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

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

where

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

lemma eq-uptoI [intro]:

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

by (auto simp: eq-upto-def)

lemma eq-uptoE [elim]:

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

by (auto simp: eq-upto-def)

lemma eq-upto-Suc:

$$[ g \in eq-upto\ B\ f\ i; g i = f i ] \Rightarrow g \in eq-upto\ B\ f\ (\text{Suc}\ i)$$

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 $\land\ j. j < i \Rightarrow f j = g j$ and $B = C$

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

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

context mbs
begin

context

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

begin

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

fun $lb :: \text{nat} \Rightarrow 'a$ where

$lb: lb\ i = \text{min-elt} \ (i\text{th}\ (eq-upto\ (BAD\ P)\ lb\ i)\ i)$

declare $lb$.sims [simp del]

lemma eq-upto-BAD-mem:

assumes $f \in eq-upto\ (BAD\ P)\ g\ i$

shows $f\ j \in A$

using assms by (auto)

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

context

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

begin

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

lemma eq-upto-BAD-non-empty:

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

proof (induct \( i \))

case 0

show \( ?\text{case} \) using BAD-ex by auto

next

let \( ?A = \lambda i. \text{ith} (eq\text{-}upto (BAD P) \, lb \, i) \, i \)

case (Suc \( i \))

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

moreover have eq-upto (BAD P) \( lb \, i \subseteq eq\text{-}upto (BAD P) \, lb \, 0 \) by auto

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

from min-elt-mem [OF this, folded lb] obtain \( f \)

then show \( ?\text{case} \) by blast

qed

lemma non-empty-ith:

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

and \( \text{ith} (eq\text{-}upto (BAD P) \, lb \, i) \, i \neq \{\} \)

using eq-upto-BAD-non-empty [of \( i \)] by auto

lemmas

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

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

\( lb \) is a infinite bad sequence.

lemma lb-BAD:

\( lb \in BAD P \)

proof –

have \( \ast: \forall j. \, lb \, j \in \text{ith} (eq\text{-}upto (BAD P) \, lb \, j) \, j \) by (rule lb-mem)

then have \( \forall i. \, lb \, i \in A \) by (auto simp: ith-conv) (metis eq-upto-BAD-mem)

moreover

{ assume good P \( lb \)

then obtain \( i \, j \) where \( i < j \) and \( P \, (lb \, i) \, (lb \, j) \) by (auto simp: good-def)

from \( \ast \) have \( lb \, j \in \text{ith} (eq\text{-}upto (BAD P) \, lb \, j) \, j \) by (auto)

then obtain \( g \) where \( g \in eq\text{-}upto (BAD P) \, lb \, j \) 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 \( \text{P} \, (lb \, i) \, (lb \, j) \) have \( P \, (g \, i) \, (g \, j) \) by auto

with \( i < j \) have good P \( g \) by (auto simp: good-def)

with \( g \in eq\text{-}upto (BAD P) \, lb \, j \) have False by auto }

ultimately show \( ?\text{thesis} \) by blast

qed

There is no infinite bad sequence that is strictly smaller than \( lb \).
lemma lb-lower-bound:
∀ g. (lb, g) ∈ gseq → g ∉ BAD P
proof (intro allI impI)
  fix g
  assume (lb, g) ∈ gseq
  then obtain i where g i ∈ A and size (g i) < size (lb i)
  and ∀ j < i. lb j = g j by (auto simp: gseq-iff)
  moreover with lb-minimal
  have g i ∉ ith (eq-upto (BAD P) lb i) i by auto
  ultimately show g ∉ BAD P by blast
qed

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

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

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

There is a minimal bad sequence.

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

end

end

end

3 Enumerations of Well-Ordered Sets in Increasing Order

theory Least-Enum
imports Main
begin

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

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

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

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

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

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

end

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

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

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

lemma enumchain-mono:
  shows N ≤ enumchain i ∧ enumchain i < enumchain (Suc i)
proof (induct i)
  case 0
  then have enumchain 0 ≥ N by simp
  moreover then have ∃m>enumchain 0. P (enumchain 0) m using infm by blast
  ultimately show ?case by auto (metis (lifting) LeastI)
next
  case (Suc i)
  then have N ≤ enumchain (Suc i) by auto
  moreover then have ∃m>enumchain (Suc i). P (enumchain (Suc i)) m using infm by blast
  ultimately show ?case by (auto) (metis (lifting) LeastI)
qed

lemma enumchain-chain:
  shows P (enumchain i) (enumchain (Suc i))
proof (cases i)
  case 0

22
moreover have \( \exists m > \text{enumchain } 0 \cdot P \ (\text{enumchain } 0) \) \( m \) using \text{infdm} by \text{auto}
ultimately show ?thesis by \text{auto} (metis (lifting) \text{LeastI})
next
case \((\text{Suc } i)\)
moreover have \text{enumchain} (\text{Suc } i) > N \text{ using enumchain-mono by (metis le-less-trans)}
moreover then have \( \exists m > \text{enumchain} (\text{Suc } i) . P \ (\text{enumchain} (\text{Suc } i)) \) \( m \) using \text{infdm} by \text{auto}
ultimately show ?thesis by (auto) (metis (lifting) \text{LeastI})
qed
end
end

4 Almost-Full Relations

theory Almost-Full-Relations
imports
  "~/src/HOL/Library/Sublist"
  "~/src/HOL/Library/Ramsey"
  ../Regular-Sets/Regexp-Method
  ../Abstract-Rewriting/Seq
  \text{Least-Enum}
  \text{Minimal-Bad-Sequences}
begin

4.1 Basic Definitions and Facts
definition \text{almost-full-on} :: \(\forall' a \Rightarrow 'a \Rightarrow \text{bool} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \) where
\text{almost-full-on} \( P \ A \leftrightarrow (\forall f \in \text{SEQ } A. \text{good } P \ f)\)

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

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

lemma \text{almost-full-onD}:
\text{fixes} f :: \text{nat} \Rightarrow 'a \text{ and } A :: 'a \text{ set}
\text{assumes} \text{almost-full-on} \ P \ A \text{ and } \bigwedge i. f \ i \in A
\text{obtains} i \ j \text{ where } i < j \text{ and } P (f \ i) (f \ j)
\text{using} \text{assms unfolding} \text{almost-full-on-def} \text{by blast}

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

**unfolding** almost-full-on-def by blast

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

**lemma** almost-full-on-subset:
- **assumes** \( A \subseteq B \Rightarrow \text{almost-full-on } P B \Rightarrow \text{almost-full-on } P A \)
- **by** (auto simp: almost-full-on-def)

**lemma** almost-full-on-mono:
- **assumes** \( A \subseteq B \) and \( \forall x y. Q x y \Rightarrow P x y \)
- and almost-full-on Q B
- **shows** almost-full-on P A
- **using** assms by (metis almost-full-on-def almost-full-on-subset good-def)

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

**lemma** almost-full-on-imp-homogeneous-subseq:
- **assumes** almost-full-on P A and \( \forall i :: \text{nat}. f i \in A \)
- **shows** \( \exists \phi :: \text{nat}. \forall i j. i < j \rightarrow \phi i < \phi j \land P (f (\phi i)) (f (\phi j)) \)

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

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

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

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

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

**have** \( c = 0 \lor c = 1 \) **using** \( c < 2 \) by arith

**then** **show** \( \text{thesis} \)

**proof**
- **assume** \([simp]: c = 0 \)
- **have** \( \forall i j. i < j \rightarrow P (f (\text{enum } i)) (f (\text{enum } j)) \)

**proof** (intro allI impI)
- **fix** i j :: nat
- **assume** i < j
- **from** * and enum-P and enum-less [OF \( i < j \)] **have** \( \{\text{enum } i, \text{enum } j\} \in X \) by auto

24
with enum-less [OF ⟨i < j⟩]

show P (f (enum i)) (f (enum j)) by (auto simp: X-def doubleton-eq-iff)

qed

then show ?thesis using enum-less by blast

next

assume [simp]: c = 1

have ∀i j. i < j ⟹ ¬P (f (enum i)) (f (enum j))

proof (intro allI impl)

fix i j :: nat

assume i < j

from * and enum-P and enum-less [OF ⟨i < j⟩] have \{enum i, enum j\} ∈ Y

by auto

with enum-less [OF ⟨i < j⟩]

show ¬P (f (enum i)) (f (enum j)) by (auto simp: Y-def X-def doubleton-eq-iff)

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 ¬antichain-on P \ f A

proof

assume *: antichain-on P \ f A

then have ∀i. f i ∈ A by simp

with assms have good P \ f by (auto simp: almost-full-on-def)

then obtain i j where i < j and P (f i) (f j)

unfolding good-def by auto

moreover with * have incomparable P (f i) (f j) by auto

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

qed

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 ⊆ B

shows almost-full-on (λx y. Q (h x) (h y)) A (is almost-full-on \ P A)

proof

fix f

assume ∀i::nat. f i ∈ A

then have \bigwedge i. h (f i) ∈ B using \ h ' A ⊆ B: by auto

with [unfolded almost-full-on-def, THEN bspec, of h \ f]

show good ?P f unfolding good-def comp-def by blast

qed

The homomorphic image of an almost-full set is almost-full.
lemma almost-full-on-hom:
  fixes h :: 'a ⇒ 'b
  assumes hom: ∀x y. [x ∈ A; y ∈ A; P x y] ⟹ Q (h x) (h y)
  and af: almost-full-on P A
  shows almost-full-on Q (h ' A)
proof
  fix f :: nat ⇒ 'b
  assume ∀i. f i ∈ h ' A
  then have ∀i. ∃x. x ∈ A ∧ f i = h x by (auto simp: image_def)
  from choice [OF this] obtain g
    where *: ∀i. g i ∈ A ∧ f i = h (g i) by blast
  show good Q f
proof (rule ccontr)
  assume bad: bad Q f
  { fix i j :: nat
    assume i < j
    from bad have ¬ Q (f i) (f j) using ⟨i < j⟩ by (auto simp: good_def)
    with hom have ¬ P (g i) (g j) using * by auto }
  then have bad P g by (auto simp: good_def)
  with af and * show False by (auto simp: good_def almost-full-on_def)
qed
qed

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

lemma almost-full-on-mon:
  assumes mon: ∀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 by (auto simp: bij_betw_def)
  show good P f
proof (rule ccontr)
  assume bad: bad P f
  { fix i j :: nat
    assume i < j
    from bad have ¬ P (f i) (f j) using ⟨i < j⟩ by (auto simp: good_def)
    with mon have ¬ Q (h (f i)) (h (f j))
    using * by (auto simp: bij_betw_def inj_on_def) }
  then have bad Q (h ◦ f) by (auto simp: good_def)
  with af and ** show False by (auto simp: good_def almost-full-on_def)
qed
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

26
proof (rule ccontr)
assume ¬ almost-full-on \( P \equiv A \)
then obtain \( f :: \text{nat} \rightarrow \mathcal{P}(\text{nat}) \)
and \( \forall i\ j. \ i < j \rightarrow f(i) 
\neq f(j) \)
unfolding almost-full-on-def by (auto dest: badE)
with \( \text{total-on} \ A \)
have \( \forall i\ j. \ i < j \rightarrow P(f(i)) \land P(f(j)) \)
unfolding total-on-def by blast
then have \( \forall i\ j. \ i < j \tablecolumn{0}{-} P(f(j)) \land P(f(i)) \)
unfolding total-on-def by blast
have False
unfolding wfp-on-def by blast
qed

4.2 Adding a Bottom Element to a Set

definition with-bot :: \('a set \Rightarrow 'a option set \)
where
\( A_{\bot} = \{\text{None}\} \cup \text{Some ' A} \)

lemma with-bot-iff [iff]:
\( \text{Some x} \in A_{\bot} \tablecolumn{0}{-} x \in A \)
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]:
\( x \neq \text{None} \rightarrow (\forall x. \ x \in A \rightarrow x \in A_{\bot}) \)
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]:
\( A_{\bot} = \{\text{None}\} \rightarrow A = \{\} \)
by (auto elim: with-bot-cases)

lemma with-bot-UNIV [simp]:
\( UNIV_{\bot} = UNIV \)
proof (rule set-eqI)
fix \( x :: 'a option \)
show \( x \in UNIV_{\bot} \rightarrow x \in UNIV \)
by (cases x) auto
qed

4.3 Adding a Bottom Element to an Almost-Full Set

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

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

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

4.4 Disjoint Union of Almost-Full Sets

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

lemma not-sum-le-cases:
  assumes ¬ sum-le P Q a b
  and ∨ x y. [a = Inl x; b = Inl y; ¬ P x y] ⇒ thesis
  and ∨ x y. [a = Inr x; b = Inr y; ¬ Q x y] ⇒ thesis
  and ∨ x y. [a = Inl x; b = Inr y] ⇒ thesis
  and ∨ x y. [a = Inr x; b = Inl y] ⇒ thesis
  shows thesis
  using assms by (cases a b rule: sum.exhaust [case-product sum.exhaust]) auto

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

lemma almost-full-on-Plus:
  assumes almost-full-on P A and almost-full-on Q B
shows almost-full-on (sum-le P Q) (A <+> B) (is almost-full-on ?P ?A)
proof
  fix f :: nat ⇒ ('a + 'b)
  let ?I = f - 'Inl ' A
  let ?J = f - 'Inr ' B
  assume ∀ i. f i ∈ ?A
  then have "?J = (UNIV::nat set) − ?I" by (fastforce)
  show good ?P f
  proof (rule ccontr)
    assume bad: bad ?P f
    show False
    proof (cases finite ?I)
      assume finite ?I
      then have infinite ?J by (auto simp: *)
      then interpret infinitely-many1 λ i. f i ∈ Inr ' B
        by (unfold-locales) (simp add: infinite-nat-iff-unbounded)
      have [dest]: ∀ i x. f (enum i) = Inl x ⇒ False
        using enum-P by (auto simp: image-iff) (metis Inr-Inl-False)
      let ?f = λ i. projr (f (enum i))
      have B: ∀ i. ?f i ∈ 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 ¬ ?P (f (enum i)) (f (enum j)) by (auto simp: good-def)
        then have ¬ Q (?i) (?f) 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 λ i. f i ∈ Inl ' A
        by (unfold-locales) (simp add: infinite-nat-iff-unbounded)
      have [dest]: ∀ i x. f (enum i) = Inr x ⇒ False
        using enum-P by (auto simp: image-iff) (metis Inr-Inl-False)
      let ?f = λ i. projl (f (enum i))
      have A: ∀ i. ?f i ∈ A using enum-P by (auto simp: image-iff) (metis sum.sel(1))
      { fix i j :: nat
        assume i < j
        then have enum i < enum j using enum-less by auto
        with bad have ¬ ?P (?i) (?f) by (auto simp: good-def)
        then have ¬ P (?i) (?f) by (auto elim: not-sum-le-cases) }
      then have bad P ?f by (auto simp: good-def)
      moreover from almost-full-on P A and A
      have good P ?f by (auto simp: good-def almost-full-on-def)
      ultimately show False by blast
      qed
4.5 Dickson’s Lemma for Almost-Full Relations

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

**definition**

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

**where**

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

**lemma** prod-le-True [simp]:

\[ prod-le P (\lambda - - True) a b = P (fst a) (fst b) \]

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 \( \neg \) almost-full-on ?P ?A

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

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

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

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

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

by (metis SigmaE fst-conv, metis SigmaE snd-conv)

from almost-full-on-imp-homogeneous-subseq [OF assms (1) fst]

obtain \( \varphi :: nat \Rightarrow nat \) where mono: \( \forall i. j. i < j \Rightarrow \varphi i < \varphi j \)

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

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

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

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

then obtain i j :: nat

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

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

with mono [OF i < j] and bad show False by auto

qed

4.6 Higman’s Lemma for Almost-Full Relations

**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) (f i) (f (Suc i))

unfolding assms by auto

ultimately show False by auto

qed
unfolding good-def by auto

qed

lemma ne-lists:
  assumes xs ≠ [] and xs ∈ lists A
  shows hd xs ∈ A and tl xs ∈ lists A
  using assms by (case-tac [] xs) simp-all

lemma almost-full-on-lists:
  assumes almost-full-on P A
  shows almost-full-on (list-emb P) (lists A) (is almost-full-on ?P ?A)
  proof (rule contr)
    interpret mbs ?A .
    assume ¬ ?thesis
    from mbs ′ [OF this] obtain m
      where bad: m ∈ BAD ?P
      and min: ∀ g. (m, g) ∈ gseq ⇒ good ?P g ...
    then have lists: ∃ i. m i ∈ lists A
      and ne: ∃ i. m i ≠ [] by auto
    def h ≡ λ i. hd (m i)
    def t ≡ λ i. tl (m i)
    have m: ∃ i. m i = h i # t i using ne by (simp add: h-def t-def)
    have ∀ i. h i ∈ A using ne-lists [OF ne] and lists by (auto simp add: h-def)
    from almost-full-on-imp-homogeneous-subseq [OF assms this] obtain ϕ :: nat ⇒ nat
      where less: ∃ i j. i < j ⇒ ϕ i < ϕ j
      and P: ∀ i j. i < j ⇒ P (h (ϕ i)) (h (ϕ j)) by blast
    have bad-t: bad ?P (t ∘ ϕ)
      proof
        assume good ?P (t ∘ ϕ)
        then obtain i j where i < j and ?P (t (ϕ i)) (t (ϕ j)) by auto
        moreover with P have P (h (ϕ i)) (h (ϕ j)) by blast
        ultimately have ?P (m (ϕ i)) (m (ϕ j))
          by (subst (1 2) m) (rule list-emb-Cons2, auto)
        with less and (i < j) have good ?P m by (auto simp: good-def)
        with bad show False by blast
      qed
    def m′ ≡ λ i. if i < ϕ 0 then m i else t (ϕ (i − ϕ 0))
    have m′-less: ∃ i. i < ϕ 0 ⇒ m′ i = m i by (simp add: m′-def)
    have m′-geq: ∃ i. i ≥ ϕ 0 ⇒ m′ i = m (ϕ (i − ϕ 0)) by (simp add: m′-def)
    have ∀ i. m′ i ∈ lists A using ne-lists [OF ne] and lists by (auto simp: m′-def t-def)
moreover have \( \text{length} (m' (\varphi 0)) < \text{length} (m (\varphi 0)) \) using ne by (simp add: t-def m'-geq)
moreover have \( \forall j < \varphi 0. \ m' j = m j \) by (auto simp: m'-less)
ultimately have \( (m, m') \in \text{gseq} \) using lists by (auto simp: gseq-def)
moreover have bad ?P m'
proof
  assume good ?P m'
  then obtain \( i j \) where \( i < j \) and emb: \( ?P (m' i) (m' j) \) by (auto simp: good-def)
  { assume \( j < \varphi 0 \)
    with \( i < j \) and emb have \( ?P (m i) (m j) \) by (auto simp: m'-less)
    moreover
    { assume \( \varphi 0 \leq i \)
      with \( i < j \) and emb have \( t (\varphi (i - \varphi 0)) \) \( t (\varphi (j - \varphi 0)) \)
        and \( i - \varphi 0 < j - \varphi 0 \) by (auto simp: m'-geq)
      with bad-t have False by auto }
    moreover
    { assume \( i < \varphi 0 \) and \( \varphi 0 \leq j \)
      with \( i < j \) and bad have False by auto }
    from list-emb-Cons [OF this, of \( (\varphi (j - \varphi 0)) \)]
      have \( ?P (m i) (m (\varphi (j - \varphi 0))) \) using ne by (simp add: h-def t-def)
    moreover have \( i < \varphi 0 \)
      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 have False using min by blast
qed
ultimately show False using min by blast
qed

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

lemma list-emb-eq-length-P:
assumes \( \text{length} \ xs = \text{length} \ ys \)
and \( \text{list-emb} P \ xs \ys \)
shows \( \forall i<\text{length} \ xs. P (xs ! i) (ys ! i) \)
using \( \text{assms} \)
proof (induct rule: list-emb-eq-length-induct)
case \( (\text{Cons} \ x \ y \ xs \ys) \)
show ?case
proof (intro allI impI)
4.7 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 :: nat ⇒ 'a
- assume *: ∀ i. f i ∈ A
- let ?I = UNIV :: nat set
- have f ' ?I ⊆ 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 1]
- 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 < l⟩ show good P f by (auto simp: good-def)

**qed**

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

4.8 Natural Numbers

**lemma** almost-full-on-UNIV-nat:
- almost-full-on (op ≤) (UNIV :: nat set)

**proof**
- let ?P = sublisteq :: bool list ⇒ bool list ⇒ bool
- have *: length ' (UNIV :: bool list set) = (UNIV :: nat set)
  by (metis Ex-list-of-length surj-def)
- have almost-full-on (op ≤) (length ' (UNIV :: bool list set))
  by (rule almost-full-on-hom)
- fix xs ys :: bool list
- assume ?P xs ys
- then show length xs ≤ length ys
  by (metis list-emb-length)

**next**
have finite (UNIV :: bool set) by auto
from almost-full-on-lists [OF eq-almost-full-on-finite-set [OF this]]
  show almost-full-on ?P UNIV unfolding lists-UNIV .
qed
then show ?thesis unfolding * .
qed

4.9 Further Results

lemma af-trans-imp-wf:
  assumes af: almost-full-on P A
  and trans: transp-on P A
  shows wfp-on (strict P) A
proof –
  show wfp-on (strict P) A
proof (unfold wfp-on-def, rule notI)
    assume ∃f. ∀i. f i ∈ A ∧ strict P (f (Suc i)) (f i)
    then obtain f where *: chain-on ((strict P)⁻¹⁻¹) f A by blast
  from chain-on-transp-on-less [OF this]
  and transp-on-strict [THEN transp-on-converse, OF trans]
  have ∀i j. i < j −→ ¬ P (f i) (f j) by blast
  with af show False
  using * by (auto simp: almost-full-on-def good-def)
qed

lemma wf-and-no-antichain-imp-qo-extension-wf:
  assumes wf: wfp-on (strict P) A
  and anti: ¬ (∃f. antichain-on P f A)
  and subrel: ∀x ∈ A. ∀y ∈ A. P x y −→ Q x y
  and qo: qo-on Q A
  shows wfp-on (strict Q) A
proof (rule ccontr)
  have transp-on (strict Q) A
  using qo unfolding go-on-def transp-on-def by blast
then have *: transp-on ((strict Q)⁻¹⁻¹) A by (rule transp-on-converse)
  assume ¬ wfp-on (strict Q) A
then obtain f :: nat ⇒ 'a where A: ∀i. f i ∈ A
  and ∀i. strict Q (f (Suc i)) (f i) unfolding wfp-on-def by blast+
then have chain-on ((strict Q)⁻¹⁻¹) f A by auto
from chain-on-transp-on-less [OF this *]
  have *: ∀i j. i < j −→ P (f i) (f j)
  using subrel and A by blast
show False
proof (cases)
  assume ∃k. ∀i > k. ∃j > i. P (f j) (f i)
then obtain & where ∀i > k. ∃j > i. P (f j) (f i) by auto
from subchain [of k - f, OF this] obtain g
  where ∀i j. i < j −→ g i < g j
and \(\forall i. P (f (g (\text{Suc } i)))) (f (g i))\) by auto

with \(\star\) have \(\forall i. \text{strict } P (f (g (\text{Suc } i)))) (f (g i))\) by blast

with \(\text{wf } [\text{unfolded wfp-on-def not-ex}, \text{THEN spec, of } \lambda i. f (g i)]\) 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 \(\forall k. h \cdot k > k\)

and \(\star\star\) \(\forall k. (\forall j > h \cdot k. \neg P (f j) (f (h k)))\) by auto

def [simp]: \(\varphi \equiv \text{li} (h ^{\text{Suc } i} 0)\)

have \(\forall i. \varphi i < \varphi (\text{Suc } i)\)

using \(\forall k. h \cdot k > k\) by (induct-tac \(i\)) auto

then have mono: \(\forall i j. i < j \implies \varphi i < \varphi j\) by (metis lift-Suc mono-less)

then have \(\forall i j. i < j \implies \neg P (f (\varphi j)) (f (\varphi i))\)

using \(\star\star\) by auto

with mono [THEN \(\star\)]

have \(\forall i j. i < j \implies \text{incomparable } P (f (\varphi j)) (f (\varphi i))\) by blast

moreover have \(\exists i j. i < j \land \neg \text{incomparable } P (f (\varphi i)) (f (\varphi j))\)

using anti [unfolded not-ex, THEN spec, of \(\lambda i. f (\varphi i)\)] and \(A\) by blast

ultimately show False by blast

qed

qed

lemma every-go-extension-wf-imp-af:

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

and qo-on Q P A

shows almost-full-on P A

proof

from qo-on P A:

have refl: reflp-on P A

and trans: transp-on P A

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

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

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

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

show good P \(f\)

proof (rule excontr)

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

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

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

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

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

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

have \(\star\star:: \forall i j. (D OO P'\star\star)\star\star (f i) (f j) \implies i > j\)
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!: simp: \(P'^{**} def\) refl-on-restrict-to-rtranclp [OF refl trans])
    apply (metis * dual-order.strict-trans1 less-Suc-eq-le refl reflp-on-def)
  by (metis le-imp-less-Suc less-trans)
qed

have \(\forall x \in A. \forall y \in A. P \ x \ y \rightarrow Q x y\) by (auto simp: \(P'^{**}\) def)
moreover have \(qo-on Q A\) by (auto simp: qo-on-def reflp-on-def transp-on-def)
ultimately have \(wfp-on (strict Q) A\)
using ext [THEN spec, of Q] by blast
moreover have \(\forall i. f i \in A \land strict Q (f (Suc i)) (f i)\)
proof
  fix i
  have \(\neg Q (f i) (f (Suc i))\)
  proof
    assume \(Q (f i) (f (Suc i))\)
    then have \((sup P' D)^{**} (f i) (f (Suc i))\) by auto
  moreover have \((sup P' D)^{**} = sup (P'^{**}) \((P'^{**} OO (D OO P'^{**}))^{++}\)\)
  proof
    have \(\bigwedge A B. (A \cup B)^+ = A^+ \cup A^+ O (B O A^+)\) by regexp
    from this [to-pred] show \(?thesis\) by blast
  qed
  ultimately have \(sup (P'^{**}) \((P'^{**} OO (D OO P'^{**}))^{++}\) (f i) (f (Suc i))\)
  by simp
  then have \((P'^{**} OO (D OO P'^{**}))^{++} (f i) (f (Suc i))\) by auto
  then have \(Suc i < i\)
  using ** apply auto
  by (metis (lifting, mono-tags) less-le relcomp.pre relcomp.pre tranclp-into-tranclp2)
  then show \(False\) by auto
qed
  with \(A \ [of i]\) show \(f i \in A \land strict Q (f (Suc i)) (f i)\) by auto
  qed
  ultimately show \(False\) unfolding wfp-on-def by blast
qed

end

5 Well-Quasi-Orders

theory Well-Quasi-Orders
imports Almost-Full-Relations
begin
5.1 Basic Definitions

**definition** `wqo-on :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool` where

\[ \text{wqo-on } P A \iff \text{transp-on } P A \land \text{almost-full-on } P A \]

**lemma** `wqo-on-UNIV`:

\[ \text{wqo-on } (\lambda x. \text{True}) \text{ UNIV} \]

**using** `almost-full-on-UNIV` by (auto simp: `wqo-on-def` `transp-on-def`)

**lemma** `wqo-onI [Pure.intro]`:

\[ \text{transp-on } P A ; \text{almost-full-on } P A \implies \text{wqo-on } P A \]

**unfolding** `wqo-on-def` `almost-full-on-def` by blast

**lemma** `wqo-on-imp-reflp-on`:

\[ \text{wqo-on } P A \implies \text{reflp-on } P A \]

**using** `almost-full-on-imp-reflp-on` by (auto simp: `wqo-on-def`)

**lemma** `wqo-on-imp-transp-on`:

\[ \text{wqo-on } P A \implies \text{transp-on } P A \]

by (auto simp: `wqo-on-def`)

**lemma** `wqo-on-imp-almost-full-on`:

\[ \text{wqo-on } P A \implies \text{almost-full-on } P A \]

by (auto simp: `wqo-on-def`)

**lemma** `wqo-on-imp-qo-on`:

\[ \text{wqo-on } P A \implies \text{qo-on } P A \]

by (metis `qo-on-def` `wqo-on-imp-reflp-on` `wqo-on-imp-transp-on`)

**lemma** `wqo-on-imp-good`:

\[ \forall i. f i \in A \implies \text{good } P f \]

by (auto simp: `wqo-on-def` `almost-full-on-def`)

**lemma** `wqo-on-subset`:

\[ A \subseteq B \implies \text{wqo-on } P B \implies \text{wqo-on } P A \]

**using** `almost-full-on-subset [of A B P]`

**and** `transp-on-subset [of A B P]`

**unfolding** `wqo-on-def` by blast

5.2 Equivalent Definitions

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

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

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

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

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

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

The monomorphic preimage of a wqo set is wqo.

lemma $wqo-on-mon$:
assumes $*: \forall x \in A. \forall y \in A. \ P \ x \ y \iff Q (h \ x) (h \ y)$
and $bij$: $bij-betw \ h \ A \ B$
and $wqo$: $wqo-on \ Q \ B$
shows $wqo-on \ P \ A$
proof –
have $transp-on \ P \ A$
proof
fix $x \ y \ z$ assume [intro]: $x \in A$ $y \in A$ $z \in A$
and $P \ x \ y$ and $P \ y \ z$

38
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

5.3 A Type Class for Well-Quasi-Orders

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

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

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

lemma wqo-on-UNIV-class-wqo [intro!]:
wqo-on P UNIV ⇒ class.wqo P (strict P)
by (unfold-locales) (auto simp: wqo-on-def almost-full-on-def, unfold transp-on-def, blast)

The following lemma converts between wqo-on (for the special case that the
domain is the universe of a type) and the class predicate class.wqo.

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

The strict part of a wqo is well-founded.

lemma (in wqo) wfP (op <)
proof −
  have class.wqo (op ≤) (op <) ..
  hence wqo-on (op ≤) 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 \( wqo\text{-}on\ P\ A \)

shows \( wqo\text{-}on\ (option\text{-}le\ P)\ A \perp\ (is\ wqo\text{-}on\ ?P\ ?A) \)

proof –

\[
\begin{array}{l}
\{
\text{from assms have trans [unfolded transp-on-def]: transp-on P A}
\end{array}
\]

\[
\begin{array}{l}
\text{by (auto simp: wqo-on-def)}
\end{array}
\]

\[
\begin{array}{l}
\text{have transp-on ?P ?A}
\end{array}
\]

\[
\begin{array}{l}
\text{by (auto simp: transp-on-def elim!: with-bot-cases, insert trans) blast }
\end{array}
\]

moreover

\[
\begin{array}{l}
\text{from assms and almost-full-on-with-bot}
\end{array}
\]

\[
\begin{array}{l}
\text{have almost-full-on ?P ?A by (auto simp: wqo-on-def)}
\end{array}
\]

ultimately

show \( \text{thesis by (auto simp: wqo-on-def)} \)

qed

lemma \( wqo\text{-}on\text{-}option\text{-}UNIV \) [intro]:

\( wqo\text{-}on\ P\ \text{UNIV} \implies wqo\text{-}on\ (option\text{-}le\ P)\ \text{UNIV} \)

using \( wqo\text{-}on\text{-}with\text{-}bot \) [of P \( \text{UNIV} \)] by simp

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

lemma \( wqo\text{-}on\text{-}Plus: \)

assumes \( wqo\text{-}on\ P\ A \) and \( wqo\text{-}on\ Q\ B \)

shows \( wqo\text{-}on\ (sum\text{-}le\ P\ Q)\ (A <+> B)\ (is\ wqo\text{-}on\ ?P\ ?A) \)

proof –

\[
\begin{array}{l}
\{
\text{from assms have trans [unfolded transp-on-def]: transp-on P A transp-on Q B}
\end{array}
\]

\[
\begin{array}{l}
\text{by (auto simp: wqo-on-def)}
\end{array}
\]

\[
\begin{array}{l}
\text{have transp-on ?P ?A}
\end{array}
\]

\[
\begin{array}{l}
\text{unfolding transp-on-def by (auto, insert trans) (blast+ )}
\end{array}
\]

moreover

\[
\begin{array}{l}
\text{from assms and almost-full-on-Plus have almost-full-on ?P ?A by (auto simp: wqo-on-def)}
\end{array}
\]

ultimately

show \( \text{thesis by (auto simp: wqo-on-def)} \)

qed

lemma \( wqo\text{-}on\text{-}sum\text{-}UNIV \) [intro]:

\( wqo\text{-}on\ P\ \text{UNIV} \implies wqo\text{-}on\ Q\ \text{UNIV} \implies wqo\text{-}on\ (sum\text{-}le\ P\ Q)\ \text{UNIV} \)

using \( wqo\text{-}on\text{-}Plus \) [of P \( \text{UNIV} \) Q \( \text{UNIV} \)] by simp

5.4 Dickson’s Lemma

lemma \( wqo\text{-}on\text{-}Sigma: \)

fixes \( A1 :: 'a\ \text{set} \) and \( A2 :: 'b\ \text{set} \)

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

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

proof –

\[
\begin{array}{l}
\{
\text{from assms have transp-on P1 A1 and transp-on P2 A2 by (auto simp: wqo-on-def)}
\end{array}
\]

\[
\begin{array}{l}
\text{hence transp-on ?P ?A unfolding transp-on-def prod-le-def by blast }
\end{array}
\]

40
moreover
  { from assms and almost-full-on-Sigma [of P1 A1 P2 A2]
    have almost-full-on ?P ?A by (auto simp: wqo-on-def) }
ultimately
show ?thesis by (auto simp: wqo-on-def)
qed

lemmas dickson = wqo-on-Sigma

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

5.5 Higman’s Lemma

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

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

lemmas higman = wqo-on-lists

lemma wqo-on-list-UNIV [intro]:
  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.

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

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

lemma wqo-on-lists-over-finite-sets:
  wqo-on (list-emb (op =)) (UNIV::(a::finite) list set)
  using wqo-on-lists [OF finite-eq-wqo-on [OF finite [of UNIV::(a::finite) set]]]
  by simp
lemma wqo-on-map:
fixes P and Q and h
defines \( P' \equiv \lambda x y. P x y \land Q (h x) (h y) \)
assumes wqo-on P A
and wqo-on Q B
and \( \text{subset}: h \upharpoonright A \subseteq B \)
shows wqo-on P' A

proof
let \( ?Q = \lambda x y. Q (h x) (h y) \)
from \( \langle wqo-on P A \rangle \) have transp-on P A
by (rule wqo-on-imp-transp-on)
thен show transp-on P' A
using \( \langle wqo-on Q B \rangle \) and \( \text{subset} \)
unfolding wqo-on-def transp-on-def P'-def by blast

from \( \langle wqo-on P A \rangle \) have almost-full-on P A
by (rule wqo-on-imp-almost-full-on)
from \( \langle wqo-on Q B \rangle \) have almost-full-on Q B
by (rule wqo-on-imp-almost-full-on)

show almost-full-on P' A

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

qed

lemma wqo-on-UNIV-nat:
wqo-on (op \leq) (UNIV :: \text{nat set})
unfolding wqo-on-def transp-on-def
using almost-full-on-UNIV-nat by simp

end
6 Kruskal’s Tree Theorem

theory Kruskal
imports Well-Quasi-Orders
begin

locale kruskal-tree =  
  fixes \textit{F} :: \textit{('b × nat) set}  
  \textit{mk} :: \textit{'b} ⇒ \textit{'}a list ⇒ (\textit{'a::size})  
  \textit{root} :: \textit{'a} ⇒ \textit{'}b × nat  
  \textit{args} :: \textit{'a} ⇒ \textit{'}a list  
  \textit{trees} :: \textit{'a set} 
  assumes \textit{size-arg} : \textit{t ∈ trees} ⇒ \textit{s ∈ set (args t)} ⇒ \textit{size s < size t}  
  \textit{root-mk} : (\textit{f, length ts} ∈ \textit{F} ⇒ \textit{root (mk ts)} = (\textit{f, length ts})  
  \textit{args-mk} : (\textit{f, length ts} ∈ \textit{F} ⇒ \textit{args (mk ts)} = \textit{ts}  
  \textit{trees-root} : \textit{t ∈ trees} ⇒ \textit{root t ∈ F}  
  \textit{trees-arity} : \textit{t ∈ trees} ⇒ \textit{length (args t)} = \textit{snd (root t)}  
  \textit{trees-args} : \textit{⋀ s. t ∈ trees} ⇒ \textit{s ∈ set (args t)} ⇒ \textit{s ∈ trees} 

begin

lemma \textit{mk-inject} [\textit{iff}]:  
  \textit{assumes (f, length ss) ∈ F and (g, length ts) ∈ F}  
  \textit{shows mk f ss = mk g ts}  
  \textit{proof} −  
  \{ \textit{assume mk f ss = mk g ts}  
  \textit{then have root (mk f ss) = root (mk g ts)}  
  \textit{and args (mk f ss) = args (mk g ts) by auto} \}  
  \textit{show \$\textit{thesis}}  
  \textit{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} 
  \textit{qed} 

inductive \textit{emb} for \textit{P}  
where  
  \textit{arg}: [(\textit{f, m}) ∈ \textit{F}; length ts = m; \forall t ∈ set ts. t ∈ trees; \textit{t ∈ set ts; emb P s t}] ⇒ \textit{emb P s (mk ts)} |  
  \textit{list-emb}: [(\textit{f, m}) ∈ \textit{F}; (g, n) ∈ \textit{F}; length ss = m; length ts = n; \forall s ∈ set ss, s ∈ trees; \forall t ∈ set ts. t ∈ trees; \textit{P (f, m) (g, n); list-emb (emb P) ss ts}] ⇒ \textit{emb P (mk f ss) (mk g ts)} 
  \textit{monos} list-emb-mono 

lemma \textit{almost-full-on-trees}:  
  \textit{assumes almost-full-on P F}  
  \textit{shows almost-full-on (emb P) trees (is almost-full-on ?P ?A)}  
  \textit{proof (rule ccontr)}  
  \textit{interpret mbs ?A} .  
  \textit{assume ¬ \$\textit{thesis}}  
  \textit{from mbs [OF this] obtain m} 

43
where bad: \( m \in \text{BAD} \ ?P \)
and \( \text{min: } \forall g. \ (m, g) \in \text{gseq} \longrightarrow \text{good} \ ?P \ g \ .. \)
then have trees: \( \lambda i. \ m \ i \in \text{trees} \) by auto

\[
\begin{align*}
def r & \equiv \lambda i. \ \text{root} \ (m \ i) \\
def a & \equiv \lambda i. \ \text{args} \ (m \ i) \\
def S & \equiv \bigcup \{ \text{set} \ (a \ i) \mid i. \ \text{True} \}
\end{align*}
\]

have \( m: \ \lambda i. \ m \ i = \text{mk} \ (\text{fst} \ (r \ i)) \) \((a \ i)\)
by (simp add: r-def a-def mk-root-args \([\text{OF \ trees}]\))

have lists: \( \forall i. \ a \ i \in \text{lists} \ S \) by (auto simp: a-def S-def)

have arity: \( \lambda i. \ \text{length} \ (a \ i) = \text{snd} \ (r \ i) \)
using trees-arity \([\text{OF \ trees}]\) by (auto simp: r-def a-def)

then have sig: \( \lambda i. \ (\text{fst} \ (r \ i), \ \text{length} \ (a \ i)) \in F \)
using trees-root \([\text{OF \ trees}]\) by (auto simp: a-def r-def)

have a-trees: \( \lambda i. \ \forall t \in \text{set} \ (a \ i). \ t \in \text{trees} \) by (auto simp: a-def trees-args \([\text{OF \ trees}]\))

have almost-full-on \( ?P \ S \)

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

\[
\begin{align*}
def n & \equiv \text{LEAST} \ n. \ \exists k. \ s \ k \in \text{set} \ (a \ n) \\
\text{have } \exists n. \ \exists k. \ s \ k \in \text{set} \ (a \ n) & \text{ using } S \text{ by (force simp: S-def)} \\
\text{from LeastI-ex } [\text{OF \ this}] & \text{ obtain } k \\
\text{where } \text{sk: } s \ k \in \text{set} \ (a \ n) & \text{ by (auto simp: n-def)} \\
\text{have args: } \lambda k. \ \exists m \geq n. \ s \ k \in \text{set} \ (a \ m) \\
\text{using } S & \text{ by (auto simp: S-def)} \text{ (metis Least-le n-def)}
\end{align*}
\]

\[
\begin{align*}
def m' & \equiv \lambda i. \ \text{if } i < n \ \text{then } m \ i \ \text{else } (k + (i - n)) \\
\text{have } m'\text{-less: } \lambda i. \ i < n \Longrightarrow m' \ i = m \ i & \text{ by (simp add: m'-def)} \\
\text{have } m'\text{-geq: } \lambda i. \ i \geq n \Longrightarrow m' \ i = s \ (k + (i - n)) & \text{ by (simp add: m'-def)}
\end{align*}
\]

have bad \( ?P \ m' \)

proof
  assume 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: m'-less)
  with \( i < j \) and \( \text{bad have 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: m'-geq)
  with bad-s have False by auto \}
  moreover

\[
\begin{align*}
\end{align*}
\]
\{
    \text{assume } i < n \text{ and } n \leq j \\
    \text{with } (i < j) \text{ and } \text{emb have } \ast: \iff \text{P } (m \ i) \ (s \ (k \ (j - n))) \text{ by } \text{(auto simp: m'-less m'-geq)} \\
    \text{with args obtain } l \text{ where } l \geq n \text{ and } \ast\ast: \iff s \ (k \ (j - n)) \in \text{set } (a \ l) \text{ by } \text{blast} \\
    \text{from emb.args } [\text{OF sig } (a \ l) \ - \ a\text{-trees } (a \ l) \ \ast\ast] \\
    \text{have } \iff \text{P } (m \ i) \ (m \ l) \text{ by } \text{(simp add: m)} \\
    \text{moreover have } i < l \text{ using } (i < n) \text{ and } (n \leq l) \text{ by } \text{auto} \\
    \text{ultimately have } \text{False using bad by blast } \} \\
    \text{ultimately show } \text{False using } (i < j) \text{ by } \text{arith} \\
    \text{qed} \\
    \text{moreover have } (m, m') \in \text{gseq} \\
    \text{proof} - \\
    \text{have } m \in \text{SEQ } ?A \text{ using } \text{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 } \text{(auto simp: m'-def a-def S-def)} \\
    \text{moreover have } \forall i < n. \ m \ i = m' \ i \text{ by } \text{(auto simp: m'-less)} \\
    \text{moreover have } \text{size } (m' \ n) < \text{size } (m \ n) \\
    \text{by } \text{(auto simp: m'-geq root-mk } [\text{OF sig}] \text{ args-mk } [\text{OF sig}]) \\
    \text{ultimately show } \text{?thesis by } \text{(auto simp: gseq-def)} \\
    \text{qed} \\
    \text{ultimately show } \text{False using } \text{min by blast} \\
    \text{qed} \\
    \text{from } \text{almost-full-on-lists } [\text{OF this, THEN } \text{almost-full-on-imp-homogeneous-subseq,} \\
    \text{OF lists}] \\
    \text{obtain } \varphi :: \text{nat} \Rightarrow \text{nat} \\
    \text{where } \text{less: } \forall i \ j. \ i < j \Rightarrow \varphi \ i < \varphi \ j \\
    \text{and } \text{emb: } \forall i \ j. \ i < j \Rightarrow \text{list-emb } \iff \text{P } (a \ (\varphi \ i)) \ (a \ (\varphi \ j)) \text{ by } \text{blast} \\
    \text{have roots: } \exists i. \ r \ (\varphi \ i) \in F \text{ using } \text{trees } [\text{THEN trees-root}] \text{ by } \text{(auto simp: r-def)} \\
    \text{then have } r \circ \varphi \in \text{SEQ } F \text{ by } \text{auto} \\
    \text{with } \text{assms have } \text{good } P \ (r \circ \varphi) \text{ by } \text{(auto simp: almost-full-on-def)} \\
    \text{then obtain } i \ j \\
    \text{where } i < j \text{ and } P \ (r \ (\varphi \ i)) \ (r \ (\varphi \ j)) \text{ by } \text{auto} \\
    \text{with } \text{emb } [\text{OF } (i < j)] \text{ have } \iff \text{P } (m \ (\varphi \ i)) \ (m \ (\varphi \ j)) \\
    \text{using sig and arity and } \text{a-trees by } \text{(auto simp: m intro!: emb.list-emb)} \\
    \text{with } \text{less } [\text{OF } (i < j)] \text{ and } \text{bad show } \text{False by blast} \\
    \text{qed} \\
\}

\text{inductive-cases} \\
\text{emb-mk2 } [\text{consumes 1, case-names arg list-emb}]: \text{emb } P \ s \ (mk \ g \ ts) \\

\text{inductive-cases} \\
\text{list-emb-Nil2-cases: list-emb } P \ x s \ [] \text{ and} \\
\text{list-emb-Cons-cases: list-emb } P \ x s \ (y#ys) \\

\text{lemma list-emb-trans-right:} \\
\text{assumes list-emb } P \ x s \ ys \text{ and list-emb } (\lambda y \ z. \ P \ y \ z \wedge (\forall x. \ P \ x \ y \rightarrow P \ x \ z))
ys zs

shows list-emb P xs zs using assms(2, 1) by (induct arbitrary: xs) (auto elim!: list-emb-Nil2-cases list-emb-Cons-cases)

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

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

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

end

end

theory Kruskal-Examples
imports Kruskal
begin

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

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

fun succs
where
  succs (Node f ts) = ts

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

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

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

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

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

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

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

instantiation tree :: (wqo) wqo
begin
  definition s ≤ t ←→ tree-emb UNIV (op ≤) s t
  definition (s :: 'a tree) < t ←→ s ≤ t ∧ ¬ (t ≤ s)

instance
  by (rule class.wqo.of-class.intro)
    (auto simp: less-eq-tree-def [abs-def] less-tree-def [abs-def]
      intro: wqo-on-trees [of - UNIV, simplified])
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 \textit{rt} where
\begin{align*}
\textit{rt} (V x) &= (v x, 0::\textit{nat}) \\
\textit{rt} (C n) &= (c n, 0) \\
\textit{rt} (\text{Plus} \ a \ b) &= (p, 2)
\end{align*}

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

inductive-set \textit{exp}s where
\begin{align*}
V x &\in \textit{exp}s \\
C n &\in \textit{exp}s \\
\text{a} &\in \textit{exp}s \implies \text{b} &\in \textit{exp}s \implies \text{Plus} \ a \ b &\in \textit{exp}s
\end{align*}

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

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

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

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

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

end

7 Instances of Well-Quasi-Orders

theory \textit{Wqo-Instances}
imports \textit{Kruskal}
begin
7.1 The Option Type is Well-Quasi-Ordered

**instantiation** option :: (wqo) wqo

begin
  **definition** \( x \leq y \leftrightarrow \text{option-le} \) \( \leq \) \( x \) \( y \)
  **definition** \( (x :: 'a \text{ option}) < y \leftrightarrow x \leq y \land \neg (y \leq x) \)

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

end

7.2 The Sum Type is Well-Quasi-Ordered

**instantiation** sum :: (wqo, wqo) wqo

begin
  **definition** \( x \leq y \leftrightarrow \text{sum-le} \) \( \leq \) \( x \) \( y \)
  **definition** \( (x :: 'a + 'b) < y \leftrightarrow x \leq y \land \neg (y \leq x) \)

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

end

7.3 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** \( p \leq q \leftrightarrow \text{prod-le} \) \( \leq \) \( p \) \( q \)
  **definition** \( (p :: 'a \times 'b) < q \leftrightarrow p \leq q \land \neg (q \leq p) \)

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

end

7.4 Lists are Well-Quasi-Ordered

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

**instantiation** list :: (wqo) wqo

begin
  **definition** \( xs \leq ys \leftrightarrow \text{list-emb} \) \( \leq \) \( xs \) \( ys \)
  **definition** \( (xs :: 'a \text{ list}) < ys \leftrightarrow xs \leq ys \land \neg (ys \leq xs) \)

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

theory Multiset-Extension
imports
  Restricted-Predicates
  ~~/src/HOL/Library/Multiset
begin

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

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

lemma multisets-union [simp]:
M ∈ multisets A ⇒ N ∈ multisets A ⇒ M + N ∈ multisets A
by (auto simp: multisets-def)

definition mulex1 :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool where
mulex1 P = (λM N. (M, N) ∈ mult1 {(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 M0 a {(x, y). P x y}]
by (auto simp: mulex1-def)

lemma mulex1-self-add-right [simp]:
mulex1 P A (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

51
ultimately have \((A, A + \{\#a\#\}) \in \text{mult1} \ ? R\)
unfolding \text{mult1-def} by blast
then show \(\text{thesis} \) by \((\text{simp add: mulex1-def})\)

qed

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

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

definition \text{mulex-on} :: \((\prime x \Rightarrow \prime y \Rightarrow \text{bool}) \Rightarrow \prime x \text{ multiset} \Rightarrow \prime x \text{ multiset} \Rightarrow \text{bool}\)
where
\text{mulex-on} \ P \ A = \text{restrict-to} (\text{mulex1} \ P) (\text{multisets} A)

abbreviation \text{mulex} :: \((\prime x \Rightarrow \prime y \Rightarrow \text{bool}) \Rightarrow \prime x \text{ multiset} \Rightarrow \prime x \text{ multiset} \Rightarrow \text{bool}\)
where
\text{mulex} \ P \equiv \text{mulex-on} \ P \ \text{UNIV}

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

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

lemma \text{singleton-multisets [iff]}:
\(\{\#x\#\} \in \text{multisets} A \iff x \in A\)
by \((\text{auto simp: multisets-def})\)

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

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

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

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 (auto simp: mulex-on-def)

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

lemma mulex-on-add-right [simp]:
assumes mulex-on $P \; A \; M \; N$ and $a \in A$
shows mulex-on $P \; A \; M \; (N + \{#\,a\#\})$
proof –
from assms have $a \in A$ and $N \in \text{multisets } A$ by auto
then have mulex-on $P \; A \; N \; (N + \{#\,a\#\})$ by simp
with $\langle\text{mulex-on } P \; A \; M \; N\rangle$ show $?\text{thesis}$ by (rule mulex-on-trans)
qed

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

lemma mulex-on-self-add-right [simp]:
assumes $M \in \text{multisets } A$ and $K \in \text{multisets } A$ and $K \neq \{\#\}$
shows mulex-on $P \; A \; M \; (M + K)$
using assms

proof (induct K)
  case empty
  then show ?case by (cases K = {#}) auto
next
  case (add M a)
  show ?case
  proof (cases M)
  assume M = {#}
  with add show ?thesis by auto
  next
  assume M ≠ {#}
  with add show ?thesis
      by (auto dest: mulex-on-add-right simp add: ac-simps)
  qed
qed

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

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

lemma singleton-mulex-onI:
P x y x∈A y∈A mulex-on P A \{#x#\} \{#y#\}
by (auto simp: mulex-on-def)

lemma reflclp-mulex-on-add-right [simp]:
  assumes (mulex-on P A)≈= M N and M ∈ multisets A and a ∈ A
  shows mulex-on P A M (N + (#a#))
using assms by (cases M = N) simp-all

lemma reflclp-mulex-on-add-right' [simp]:
  assumes (mulex-on P A)≈= M N and M ∈ multisets A and a ∈ A
  shows 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 mulex-on P F A B and K ∈ multisets F
  shows mulex-on P F A (K + B)
using assms
proof (induct K)
case (add K a)
 then have a ∈ F and mulex-on P F A (B + K)
      by (auto simp: multisets-def ac-simps)
 then have 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 mulex-on P F A B and K ∈ multisets F
  shows mulex-on P F A (B + K)
using mulex-on-union-right [OF assms] by (simp add: ac-simps)

Adapted from wf ?r ⇒ ∀ M. M ∈ Wellfounded.acc (mult1 ?r) in Multiset.

lemma accessible-on-mulex1-multisets:
  assumes wf: wfP-on P A
  shows ∀ M ∈ multisets A. accessible-on (mulex1 P) (multisets A) M
proof
  let ?P = mulex1 P
  let ?A = multisets A
  let ?acc = accessible-on ?P ?A
  { fix M M0 a
    assume M0: ?acc M0
    and a ∈ A
    and M0 ∈ ?A
    and wf-hyp: ∀ b. [b ∈ A; P b a] ⇒ (∀ M. ?acc (M) → ?acc (M + {#b#}))
    and acc-hyp: ∀ M. M ∈ ?A ∧ ?P M M0 ⇒ ?acc (M + {#a#})
    then have M0 + {#a#} ∈ ?A by (auto simp: multisets-def)
    then have ?acc (M0 + {#a#})
    proof (rule accessible-onI [of M0 + {#a#}])
      fix N
      assume N ∈ ?A
      and ?P N (M0 + {#a#})
      then have (∀ M. M ∈ ?A ∧ ?P M M0 ∧ N = M + {#a#} ∧ (∃ K. ∀ b ∈ #. (∀ 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) by (simp only: 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 $\forall b, b \in \# K \rightarrow P b a$
moreover from $N$ and $\forall N \in \# A$ have $K \in \# A$ by (auto simp: multisets-def)
ultimately have $\forall N \in \# A$ have $K \in \# A$ by (auto simp: multisets-def)
proof (induct $K$)
  case empty
  from $M0$ show $\forall \{\#\}$ by simp
next
  case (add $K$ $x$)
  from add.prems have $x \in A$ and $P x a$ by (auto simp: multisets-def)
with wf-hyp have $\forall M. \forall \{\#\} \rightarrow \forall \{\#\}$ by blast
moreover from add have $\forall \{\#\}$ by (auto simp: multisets-def)
ultimately have $\forall \{\#\}$ by auto
then show $\forall \{\#\}$ by (simp only: add.assoc)
qed
then show $\forall \{\#\}$ by (simp only: N)
qed

\{ note tedious-reasoning = this \}

fix $M$
assume $M \in \# A$
then show $\forall \#$ by (auto simp: multisets-def)
proof (induct $M$)
  case empty
  from $M0$ have $\forall \#$ by simp
next
  case (add $M$ $a$)
  from add have $a \in A$ by (auto simp: multisets-def)
with wf have $\forall M. \forall \# \rightarrow \forall \#$ by blast
proof (induct)
  case (less $a$
  then have $r: \forall \# \rightarrow \forall \#$ by (auto simp: multisets-def)
by auto
  from $M0$ have $\forall \#$ by simp
proof (intro allI impI)
  fix $M'$
    assume $\forall \#$
moreover then have $\forall \#$ by (blast dest: accessible-on-imp-mem)
ultimately show $\forall \#$ by (intro tedious-reasoning [OF $a \in A - r$, auto])
qed
qed
with (?acc (M) \show ?acc (M + \{#a##\}) 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 \implies mulex1 P (K + M) (K + N)
  by (auto simp: mulex1-def mult1-union)

lemma mulex-on-union:
  assumes mulex-on P A M N and K \in 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) \in multisets A
      and (K + N) \in 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) \in multisets A and (K + N) \in 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 \in multisets A

57
shows \( mulex-on \ P \ A \ (M + K) \ (N + K) \)
using \( mulex-on-union \ [OF \ \text{assms}] \) by \( \text{simp add: ac-simps} \)

lemma \( \text{union-mulex-on-mono} \):
\[
mulex-on \ P \ F \ A \ C \ \Rightarrow \ mulex-on \ P \ F \ B \ D \ \Rightarrow \ mulex-on \ P \ F \ (A + B) \ (C + D)
\]
by \( \text{metis mulex-on-multisetsD mulex-on-trans mulex-on-union mulex-on-union}' \)

lemma \( \text{union-mulex-on-mono1} \):
\[
A \in \multisets \ F \ \Rightarrow \ (mulex-on \ P \ F) = A \ C \ \Rightarrow \ mulex-on \ P \ F \ B \ D \ \Rightarrow \ mulex-on \ P \ F \ (A + B) \ (C + D)
\]
by \( \text{auto intro: union-mulex-on-mono mulex-on-union} \)

lemma \( \text{union-mulex-on-mono2} \):
\[
B \in \multisets \ F \ \Rightarrow \ mulex-on \ P \ F \ A \ C \ \Rightarrow \ (mulex-on \ P \ F) = B \ D \ \Rightarrow \ mulex-on \ P \ F \ (A + B) \ (C + D)
\]
by \( \text{auto intro: union-mulex-on-mono mulex-on-union} \)

lemma \( \text{mult1-mono} \):
\[
\text{assumes} \ \forall \ x \ y. \ [(x \in A; \ y \in A; (x, y) \in R)] \ \Rightarrow \ (x, y) \in S
\]
\[
\text{and} \quad M \in \multisets \ A
\]
\[
\text{and} \quad N \in \multisets \ A
\]
\[
\text{and} \quad \multiset1 \ R
\]
shows \( (M, N) \in \multiset1 \ S \)
using \( \text{assms unfolding mult1-def multisets-def} \)
by \( \text{auto} \) \( \text{(metis (full-types) mem-set-mset-iff set-mp)} \)

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

lemma \( \text{mulex-on-mono} \):
\[
\text{assumes} \ \ast: \ \forall \ x \ y. \ [(x \in A; \ y \in A; P x y)] \ \Rightarrow \ Q x y
\]
\[
\text{and} \quad \text{mulex-on} \ P \ A \ M \ N
\]
shows \( \text{mulex-on} \ Q \ A \ M \ N \)

proof --
let \( \text{rel} = \lambda P. \ (\text{restrict-to} \ (\text{mulex1} \ P) \ (\text{multisets} \ A)) \)
from \( \text{mulex-on} \ P \ A \ M \ N \) have \( (?\text{rel} \ P)^++ \ M \ N \) by \( \text{simp add: mulex-on-def} \)
then have \( (?\text{rel} \ Q)^++ \ M \ N \)
proof \( \text{(induct rule: tranclp.induct)} \)
\[
\text{case} \ (\text{r-into-trancl} \ M \ N)
\]
then have \( M \in \multisets \ A \) and \( N \in \multisets \ A \) by \( \text{auto} \)
from \( \text{mulex1-mono} \ [OF \ \ast \ \text{this}] \) and \( \text{r-into-trancl} \)
show \( ?\text{case by auto} \)

58
next
  case (trancl-into-trancl L M N)
    then have \( M \in \text{multisets} A \) and \( N \in \text{multisets} A \) by auto
    from mulex1-mono [OF * this] and trancl-into-trancl
    have \( \rel Q M N \) by auto
    with \( (\rel Q)^+ L M \) show ?case by (rule tranclp.trancl-into-trancl)
  qed
then show ?thesis by (simp add: mulex-on-def)
qed

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

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

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

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

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

lemma multisets-UNIV [simp]: \( \text{multisets} \text{UNIV} = \text{UNIV} \)
lemma non-empty-multiset-induct [consumes 1, case-names singleton add]:
assumes \( M \neq \{\#\} \)
and \( \forall x. P \{\#x\#\} \)
and \( \forall M x. P M \implies P (M + \{\#x\#\}) \)
shows \( P M \)
using assms by (induct \( M \)) (auto, metis union-is-single)

lemma mulex-on-all-strict:
assumes \( X \neq \{\#\} \)
assumes \( X \in \text{multisets } A \) and \( Y \in \text{multisets } A \)
and \( \exists y. y \in \# Y \implies (\exists x. x \in \# X \land P y x) \)
shows \( \text{mulex-on } P A Y X \)
using assms
proof (induction \( X \) arbitrary: \( Y \) rule: non-empty-multiset-induct)
case (singleton \( x \))
then have \( \text{mulex1 } P Y \{\#x\#\} \)
unfolding mulex1-def mult1-def
by auto (metis count-single empty-neutral (1) less-nat-zero-code singleton.prems(3))
with singleton show ?case by (auto simp: mulex-on-def)
next
case (add \( M x \))
let \( ?Y = \{ y \in \# Y. \exists x. x \in \# M \land P y x \} \)
let \( ?Z = Y - ?Y \)
have \( Y = ?Z + ?Y \) by (subt multiset-eq-iff) auto
from \( \exists Y \in \text{multisets } A \) have \( ?Y \in \text{multisets } A \) by (metis multiset-partition union-multisets-iff)
moreover have \( \forall y. y \in \# ?Y \implies (\exists x. x \in \# M \land P y x) \) by auto
moreover have \( M \in \text{multisets } A \) using add by auto
ultimately have \( \text{mulex-on } P A ?Y M \) using add by blast
moreover have \( \text{mulex-on } P A ?Z \{\#x\#\} \)
proof –
have \( \{\#x\#\} = \{\#\} + \{\#x\#\} \) by simp
moreover have \( ?Z = \{\#\} + ?Z \) by simp
moreover have \( \forall y. y \in \# ?Z \implies P y x \)
using add.prems by (auto, metis (full-types) less-not-refl3)
ultimately have \( \text{mulex1 } P \{\#x\#\} \) unfolding mulex1-def mult1-def by blast
moreover have \( \{\#x\#\} \in \text{multisets } A \) using add.prems by auto
moreover have \( ?Z \in \text{multisets } A \)
using \( Y \in \text{multisets } A \) by (metis diff-union-cancelL multiset-partition union-multisetsD)
ultimately show \( \text{thesis} \) by (auto simp: mulex-on-def)
qed
ultimately have \( \text{mulex-on } P A (?Y + ?Z) (M + \{\#x\#\}) \) by (rule union-mulex-on-mono)
then show ?case using \( Y \) by (simp add: ac-simps)
qed

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

**lemma** `diff-set-Ex-iff`:
\[ X \neq \{\#\} \land X \leq \# M \land N = (M - X) + Y \iff X \neq \{\#\} \land (\exists Z. M = Z + X \land N = Z + Y) \]
by (auto) (`metis add-diff-cancel-left` multiset-diff-union-assoc union-commute)

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

**lemma** `mulex-on-alt-def`:
assumes `trans`: `transp-on P A`
shows `mulex-on P A M N \iff M \in \text{multisets } A \land N \in \text{multisets } A \land (\exists X Y Z.
\begin{align*}
X &\neq \{\#\} \land N = Z + X \land M = Z + Y \land (\forall y. y \in \# Y \longrightarrow (\exists x. x \in \# X \land P y x))
\end{align*}
) (is `?P M N \iff ?Q M N`)
proof
assume `?P M N` then show `?Q M N`
proof (induct `M N`)
case `base M N`
then obtain `a M0 K` where `N = M0 + \{\#a\#\}`
and `M = M0 + K`
and `*: \forall b. b \in \# K \longrightarrow P b a`
and `M \in \text{multisets } A` and `N \in \text{multisets } A` by (auto simp: mulex1-def `mult1-def`)
moreover then have `\{\#a\#\} \in \text{multisets } A` and `K \in \text{multisets } A` by auto
moreover have `\{\#a\#\} \neq \{\#\}` by auto
moreover have `N = M0 + \{\#a\#\}` by fact
moreover have `M = M0 + K` by fact
moreover have `\forall y. y \in \# K \longrightarrow (\exists x. x \in \# (\{\#a\#\} \land P y x))` using `*` by auto
ultimately show `?case` by blast
next
case `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 = Z + X`
and `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 `mulex1 P M N`
by `blast`
from `mulex1 P M N` obtain `a M0 K`
where `N: N = M0 + \{\#a\#\}` and `M': M = M0 + K`
and `*: \forall b. b \in \# K \longrightarrow P b a` unfolding `mulex1-def` `mult1-def` by `blast`
have `L': L = (M - X) + Y` by (simp `add: L M`)
have `K: \forall y. y \in \# K \longrightarrow (\exists x. x \in \# (\{\#a\#\} \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 $\exists X = \{#a\} + (X - K)$
let $\exists Y = (K - X) + Y$

have $L \in multisets A$ and $N \in multisets A$ by fact+
moreover have $\exists X \neq \{\#\} \land (\exists Z. N = Z + \exists X \land L = Z + \exists Y)$
proof
  have $\exists X \neq \{\#\}$ by auto
moreover have $\exists X \leq \# N$
  using $M N M'$ by (simp add: add.commute [of $\{#a\}$])
  (metis Multiset.diff-le_self add.commute add-diff-cancel-right)
moreover have $L = (N - \exists X) + \exists Y$
proof (rule multiset-eqI)
  fix $x :: 'a$
  let $?c = \lambda M. count M x$
  let $?ic = \lambda x. \text{int}(?c x)$
  from ($\exists X \leq \# N$) have $*: \exists c (\{#a\} + ?c (X - K) \leq ?c N$
  by (simp add: subseteq-mset-def)
  from $*$ have $**: \exists c (X - K) \leq ?c M0$ unfolding $N$ by simp
  have $?ic (N - \exists X + \exists Y) = \text{int}(\exists c N - \exists c \exists X) + \exists Y$ by simp
also have $\ldots = \text{int}(\exists c N - (\exists c \{#a\} + \exists c (X - K))) + \exists c (K - X) + \exists Y$
also have $\ldots = \exists c N - (\exists c \{#a\} + \exists c (X - K)) + \exists c (K - X) + \exists Y$
  using zdiff-int [OF $*$] by simp
also have $\ldots = (\exists c N - \exists c \{#a\}) - \exists c (X - K) + \exists c (K - X) + \exists Y$
also have $\ldots = (\exists c N - \exists c \{#a\}) + (\exists c (K - X) - \exists c (X - K)) + \exists Y$
also have $\ldots = \exists c L$
  unfolding $L'M'N$
  using zdiff-int [OF $**$]
  by simp
finally show $?c L = ?c (N - \exists X + \exists Y)$ by simp
qed
ultimately show $?c (N - \exists X + \exists Y)$ by (metis diff-set-Ex-iff)
qed
moreover have $\forall y. y \in \# \exists Y \longrightarrow (\exists x. x \in \# \exists X \land P y x)$
proof (intro allIImpl)
fix $y$ assume $y \in \# \exists Y$
then have $y \in \# K - X \lor y \in \# Y$ by auto
then show $\exists x. x \in \# \exists X \land P y x$ by auto
proof
  assume $y \in \# K - X$
  with $K$ show $?thesis$ by force
next
  assume $y \in \# Y$
  with $Y$ obtain $x$ where $x \in \# X$ and $P y x$ by blast

62
9  Multiset Extension Preserves Well-Quasi-Orders
case (list-emb-Cons2 x y xs ys)
then show ?case
  by (auto intro: union-mulex-on-mono mulex-on-union'
      intro!: singleton-mulex-onI mulex-on-union
      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-refclp-mulex-on, simplified]
  show ?thesis using assms by blast
qed

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

end

References

doi:10.1017/S0305004100003844.