

Partial Order Reduction

Julian Brunner

March 17, 2025

Abstract

This entry provides a formalization of the abstract theory of ample set partial order reduction as presented in [2, 1]. The formalization includes transition systems with actions, trace theory, as well as basics on finite, infinite, and lazy sequences. We also provide a basic framework for static analysis on concurrent systems with respect to the ample set condition.

Contents

| | | |
|-----------|--|-----------|
| 1 | List Prefixes | 2 |
| 2 | Lists | 3 |
| 3 | Finite Prefixes of Infinite Sequences | 5 |
| 4 | Sets | 11 |
| 5 | Basics | 18 |
| 5.1 | Types | 18 |
| 5.2 | Rules | 18 |
| 5.3 | Constants | 19 |
| 5.4 | Theorems for @termcurry and @termsplit | 21 |
| 6 | Relations | 23 |
| 7 | Transition Systems | 25 |
| 8 | Trace Theory | 30 |
| 9 | Transition Systems and Trace Theory | 50 |
| 10 | Functions | 53 |
| 11 | Extended Natural Numbers | 55 |

| | |
|--|------------|
| 12 Chain-Complete Partial Orders | 55 |
| 13 Sets and Extended Natural Numbers | 59 |
| 14 Coinductive Lists | 70 |
| 14.1 Index Sets | 77 |
| 14.2 Selections | 79 |
| 15 Prefixes on Coinductive Lists | 87 |
| 16 Stuttering | 89 |
| 17 Interpreted Transition Systems and Traces | 94 |
| 18 Abstract Theory of Ample Set Partial Order Reduction | 100 |
| 19 LTL Formulae | 115 |
| 20 Correctness Theorem of Partial Order Reduction | 115 |
| 21 Static Analysis for Partial Order Reduction | 116 |

1 List Prefixes

```

theory List-Prefixes
imports HOL-Library.Prefix-Order
begin

lemmas [intro] = prefixI strict-prefixI[folded less-eq-list-def]
lemmas [elim] = prefixE strict-prefixE[folded less-eq-list-def]

lemmas [intro?] = take-is-prefix[folded less-eq-list-def]

hide-const (open) Sublist.prefix Sublist.suffix

lemma prefix-finI-item[intro!]:
  assumes  $a = b$   $u \leq v$ 
  shows  $a \# u \leq b \# v$ 
  using assms by force
lemma prefix-finE-item[elim!]:
  assumes  $a \# u \leq b \# v$ 
  obtains  $a = b$   $u \leq v$ 
  using assms by force

lemma prefix-fin-append[intro]:  $u \leq u @ v$  by auto
lemma pprefix-fin-length[dest]:
  assumes  $u < v$ 
  shows  $length\ u < length\ v$ 

```

```

proof –
  obtain  $a\ w$  where  $1: v = u @ a \# w$  using assms by rule
  show ?thesis unfolding  $1$  by simp
qed

```

end

2 Lists

```

theory List-Extensions
imports HOL-Library.Sublist
begin

```

```

declare remove1-idem[simp]

```

```

lemma nth-append-simps[simp]:
   $i < \text{length } xs \implies (xs @ ys) ! i = xs ! i$ 
   $i \geq \text{length } xs \implies (xs @ ys) ! i = ys ! (i - \text{length } xs)$ 
unfolding nth-append by simp+

```

```

notation zip (infixr  $\langle || \rangle$  51)

```

```

abbreviation project  $A \equiv \text{filter } (\lambda a. a \in A)$ 

```

```

abbreviation select  $s\ w \equiv \text{nths } w\ s$ 

```

```

lemma map-plus[simp]:  $\text{map } (\text{plus } n) [i ..< j] = [i + n ..< j + n]$ 

```

```

proof (induct  $n$ )

```

```

  case  $0$ 

```

```

    show ?case by simp

```

```

next

```

```

  case (Suc  $n$ )

```

```

    have  $\text{map } (\text{plus } (\text{Suc } n)) [i ..< j] = \text{map } (\text{Suc } \circ \text{plus } n) [i ..< j]$  by simp

```

```

    also have  $\dots = (\text{map } \text{Suc} \circ \text{map } (\text{plus } n)) [i ..< j]$  by simp

```

```

    also have  $\dots = \text{map } \text{Suc} (\text{map } (\text{plus } n) [i ..< j])$  by simp

```

```

    also have  $\dots = \text{map } \text{Suc} [i + n ..< j + n]$  unfolding Suc by simp

```

```

    also have  $\dots = [\text{Suc } (i + n) ..< \text{Suc } (j + n)]$  unfolding map-Suc-upt by simp

```

```

    also have  $\dots = [i + \text{Suc } n ..< j + \text{Suc } n]$  by simp

```

```

    finally show ?case by this

```

```

qed

```

```

lemma singleton-list-lengthE[elim]:

```

```

  assumes  $\text{length } xs = 1$ 

```

```

  obtains  $x$ 

```

```

  where  $xs = [x]$ 

```

```

proof –

```

```

  have  $0: \text{length } xs = \text{Suc } 0$  using assms by simp

```

```

  obtain  $y\ ys$  where  $1: xs = y \# ys$   $\text{length } ys = 0$  using  $0$  Suc-length-conv by

```

```

metis

```

```

  show ?thesis using that  $1$  by blast

```

qed

lemma *singleton-hd-last*: $length\ xs = 1 \implies hd\ xs = last\ xs$ by *fastforce*

lemma *set-subsetI*[*intro*]:

assumes $\bigwedge i. i < length\ xs \implies xs\ !\ i \in S$
shows $set\ xs \subseteq S$

proof

fix x

assume $0: x \in set\ xs$

obtain i where $1: i < length\ xs\ x = xs\ !\ i$ using 0 unfolding *in-set-conv-nth*

by *blast*

show $x \in S$ using *assms*(1) 1 by *auto*

qed

lemma *hd-take*[*simp*]:

assumes $n \neq 0\ xs \neq []$
shows $hd\ (take\ n\ xs) = hd\ xs$

proof -

have $1: take\ n\ xs \neq []$ using *assms* by *simp*

have $2: 0 < n$ using *assms* by *simp*

have $hd\ (take\ n\ xs) = take\ n\ xs\ !\ 0$ using *hd-conv-nth*[*OF* 1] by *this*

also have $\dots = xs\ !\ 0$ using *nth-take*[*OF* 2] by *this*

also have $\dots = hd\ xs$ using *hd-conv-nth*[*OF* *assms*(2)] by *simp*

finally show *?thesis* by *this*

qed

lemma *hd-drop*[*simp*]:

assumes $n < length\ xs$
shows $hd\ (drop\ n\ xs) = xs\ !\ n$
using *hd-drop-conv-nth* *assms* by *this*

lemma *last-take*[*simp*]:

assumes $n < length\ xs$
shows $last\ (take\ (Suc\ n)\ xs) = xs\ !\ n$

using *assms*

proof (induct *xs* arbitrary: n)

case (*Nil*)

show *?case* using *Nil* by *simp*

next

case (*Cons* $x\ xs$)

show *?case* using *Cons* by (*auto*) (*metis* *Suc-less-eq* *Suc-pred*)

qed

lemma *split-list-first-unique*:

assumes $u_1\ @\ [a]\ @\ u_2 = v_1\ @\ [a]\ @\ v_2\ a \notin set\ u_1\ a \notin set\ v_1$
shows $u_1 = v_1$

proof -

obtain w where $u_1 = v_1\ @\ w \wedge w\ @\ [a]\ @\ u_2 = [a]\ @\ v_2 \vee$

$u_1\ @\ w = v_1\ @\ [a]\ @\ u_2 = w\ @\ [a]\ @\ v_2$ using *assms*(1) *append-eq-append-conv2*

by *blast*

thus *?thesis* **using** *assms(2, 3)* **by** (*auto*) (*metis hd-append2 list.sel(1) list.set-sel(1)*)+
qed

end

3 Finite Prefixes of Infinite Sequences

theory *Word-Prefixes*

imports

List-Prefixes

../Extensions/List-Extensions

Transition-Systems-and-Automata.Sequence

begin

definition *prefix-fininf* :: '*a list* \Rightarrow '*a stream* \Rightarrow *bool* (**infix** $\langle \leq_{FI} \rangle$ 50)
where $u \leq_{FI} v \equiv \exists w. u @- w = v$

lemma *prefix-fininfI[intro]*:
assumes $u @- w = v$
shows $u \leq_{FI} v$
using *assms* **unfolding** *prefix-fininf-def* **by** *auto*

lemma *prefix-fininfE[elim]*:
assumes $u \leq_{FI} v$
obtains w
where $v = u @- w$
using *assms* **unfolding** *prefix-fininf-def* **by** *auto*

lemma *prefix-fininfI-empty[intro!]*: $[] \leq_{FI} w$ **by** *force*

lemma *prefix-fininfI-item[intro!]*:
assumes $a = b$ $u \leq_{FI} v$
shows $a \# u \leq_{FI} b \#\# v$
using *assms* **by** *force*

lemma *prefix-fininfE-item[elim!]*:
assumes $a \# u \leq_{FI} b \#\# v$
obtains $a = b$ $u \leq_{FI} v$
using *assms* **by** *force*

lemma *prefix-fininf-item[simp]*: $a \# u \leq_{FI} a \#\# v \longleftrightarrow u \leq_{FI} v$ **by** *force*

lemma *prefix-fininf-list[simp]*: $w @ u \leq_{FI} w @- v \longleftrightarrow u \leq_{FI} v$ **by** (*induct w, auto*)

lemma *prefix-fininf-conc[intro]*: $u \leq_{FI} u @- v$ **by** *auto*

lemma *prefix-fininf-prefix[intro]*: *stake k w* \leq_{FI} *w* **using** *stake-sdrop* **by** *blast*

lemma *prefix-fininf-set-range[dest]*: $u \leq_{FI} v \Longrightarrow \text{set } u \subseteq \text{sset } v$ **by** *auto*

lemma *prefix-fininf-absorb*:
assumes $u \leq_{FI} v @- w$ $\text{length } u \leq \text{length } v$
shows $u \leq v$

proof $-$

obtain x **where** $1: u @- x = v @- w$ **using** *assms(1)* **by** *auto*

have $u \leq u @ \text{stake} (\text{length } v - \text{length } u) x$ **by rule**
also have $\dots = \text{stake} (\text{length } v) (u @- x)$ **using** *assms(2)* **by** (*simp add: stake-shift*)
also have $\dots = \text{stake} (\text{length } v) (v @- w)$ **unfolding 1 by rule**
also have $\dots = v$ **using** *eq-shift* **by blast**
finally show *?thesis* **by this**
qed
lemma *prefix-fininf-extend*:
assumes $u \leq_{FI} v @- w$ $\text{length } v \leq \text{length } u$
shows $v \leq u$
proof –
obtain x **where** $1: u @- x = v @- w$ **using** *assms(1)* **by auto**
have $v \leq v @ \text{stake} (\text{length } u - \text{length } v) w$ **by rule**
also have $\dots = \text{stake} (\text{length } u) (v @- w)$ **using** *assms(2)* **by** (*simp add: stake-shift*)
also have $\dots = \text{stake} (\text{length } u) (u @- x)$ **unfolding 1 by rule**
also have $\dots = u$ **using** *eq-shift* **by blast**
finally show *?thesis* **by this**
qed
lemma *prefix-fininf-length*:
assumes $u \leq_{FI} w$ $v \leq_{FI} w$ $\text{length } u \leq \text{length } v$
shows $u \leq v$
proof –
obtain $u' v'$ **where** $1: w = u @- u'$ $w = v @- v'$ **using** *assms(1, 2)* **by**
blast+
have $u = \text{stake} (\text{length } u) (u @- u')$ **using** *shift-eq* **by blast**
also have $\dots = \text{stake} (\text{length } u) w$ **unfolding 1(1) by rule**
also have $\dots = \text{stake} (\text{length } u) (v @- v')$ **unfolding 1(2) by rule**
also have $\dots = \text{take} (\text{length } u) v$ **using** *assms* **by** (*simp add: min.absorb2 stake-append*)
also have $\dots \leq v$ **by rule**
finally show *?thesis* **by this**
qed

lemma *prefix-fininf-append*:
assumes $u \leq_{FI} v @- w$
obtains (*absorb*) $u \leq v$ | (*extend*) z **where** $u = v @ z$ $z \leq_{FI} w$
proof (*cases length u length v rule: le-cases*)
case *le*
obtain x **where** $1: u @- x = v @- w$ **using** *assms(1)* **by auto**
show *?thesis*
proof (*rule absorb*)
have $u \leq u @ \text{stake} (\text{length } v - \text{length } u) x$ **by rule**
also have $\dots = \text{stake} (\text{length } v) (u @- x)$ **using** *le* **by** (*simp add: stake-shift*)
also have $\dots = \text{stake} (\text{length } v) (v @- w)$ **unfolding 1 by rule**
also have $\dots = v$ **using** *eq-shift* **by blast**
finally show $u \leq v$ **by this**
qed
next

```

case ge
obtain x where 1: u @- x = v @- w using assms(1) by auto
show ?thesis
proof (rule extend)
  have u = stake (length u) (u @- x) using shift-eq by auto
  also have ... = stake (length u) (v @- w) unfolding 1 by rule
  also have ... = v @ stake (length u - length v) w using ge by (simp add:
stake-shift)
  finally show u = v @ stake (length u - length v) w by this
  show stake (length u - length v) w ≤FI w by rule
qed
qed

```

```

lemma prefix-fin-prefix-fininf-trans[trans, intro]: u ≤ v ⇒ v ≤FI w ⇒ u ≤FI
w
by (metis Prefix-Order.prefixE prefix-fininf-def shift-append)

```

```

lemma prefix-finE-nth:
  assumes u ≤ v i < length u
  shows u ! i = v ! i

```

```

proof -
  obtain w where 1: v = u @ w using assms(1) by auto
  show ?thesis unfolding 1 using assms(2) by (simp add: nth-append)
qed

```

```

lemma prefix-fininfI-nth:
  assumes ∧ i. i < length u ⇒ u ! i = w !! i
  shows u ≤FI w

```

```

proof (rule prefix-fininfI)
  show u @- sdrop (length u) w = w by (simp add: assms list-eq-iff-nth-eq
shift-eq)
qed

```

```

definition chain :: (nat ⇒ 'a list) ⇒ bool
  where chain w ≡ mono w ∧ (∀ k. ∃ l. k < length (w l))

```

```

definition limit :: (nat ⇒ 'a list) ⇒ 'a stream
  where limit w ≡ smap (λ k. w (SOME l. k < length (w l)) ! k) nats

```

```

lemma chainI[intro?]:
  assumes mono w
  assumes ∧ k. ∃ l. k < length (w l)
  shows chain w
  using assms unfolding chain-def by auto

```

```

lemma chainD-mono[dest?]:
  assumes chain w
  shows mono w
  using assms unfolding chain-def by auto

```

```

lemma chainE-length[elim?]:
  assumes chain w
  obtains l

```

where $k < \text{length } (w \ l)$
using *assms unfolding chain-def* **by** *auto*

lemma *chain-prefix-limit:*

assumes *chain w*

shows $w \ k \leq_{FI} \text{limit } w$

proof (*rule prefix-fininfI-nth*)

fix *i*

assume $1: i < \text{length } (w \ k)$

have $2: \text{mono } w \ \wedge \ k. \ \exists \ l. \ k < \text{length } (w \ l)$ **using** *chainD-mono chainE-length*
assms **by** *blast+*

have $3: i < \text{length } (w \ (\text{SOME } l. \ i < \text{length } (w \ l)))$ **using** *someI-ex 2(2)* **by**
this

show $w \ k \ ! \ i = \text{limit } w \ !! \ i$

proof (*cases k SOME l. i < length (w l) rule: le-cases*)

case (*le*)

have $4: w \ k \leq w \ (\text{SOME } l. \ i < \text{length } (w \ l))$ **using** *monoD 2(1) le* **by** *this*

show *?thesis* **unfolding** *limit-def* **using** *prefix-finE-nth 4 1* **by** *auto*

next

case (*ge*)

have $4: w \ (\text{SOME } l. \ i < \text{length } (w \ l)) \leq w \ k$ **using** *monoD 2(1) ge* **by** *this*

show *?thesis* **unfolding** *limit-def* **using** *prefix-finE-nth 4 3* **by** *auto*

qed

qed

lemma *chain-construct-1:*

assumes $P \ 0 \ x_0 \ \wedge \ k \ x. \ P \ k \ x \implies \exists \ x'. \ P \ (\text{Suc } k) \ x' \ \wedge \ f \ x \leq f \ x'$

assumes $\bigwedge \ k \ x. \ P \ k \ x \implies k \leq \text{length } (f \ x)$

obtains *Q*

where $\bigwedge \ k. \ P \ k \ (Q \ k)$ *chain* $(f \circ Q)$

proof $-$

obtain *x'* **where** $1:$

$P \ 0 \ x_0 \ \wedge \ k \ x. \ P \ k \ x \implies P \ (\text{Suc } k) \ (x' \ k \ x) \ \wedge \ f \ x \leq f \ (x' \ k \ x)$

using *assms(1, 2)* **by** *metis*

define *Q* **where** $Q \equiv \text{rec-nat } x_0 \ x'$

have [*simp*]: $Q \ 0 = x_0 \ \wedge \ k. \ Q \ (\text{Suc } k) = x' \ k \ (Q \ k)$ **unfolding** *Q-def* **by**
simp+

have $2: \bigwedge \ k. \ P \ k \ (Q \ k)$

proof $-$

fix *k*

show $P \ k \ (Q \ k)$ **using** 1 **by** (*induct k, auto*)

qed

show *?thesis*

proof (*intro that chainI monoI, unfold comp-apply*)

fix *k*

show $P \ k \ (Q \ k)$ **using** 2 **by** *this*

next

fix $x \ y :: \text{nat}$

assume $x \leq y$


```

thus  $f(Q x) \leq f(Q y)$ 
proof (induct y - x arbitrary: x y)
  case 0
  show ?case using 0 by simp
next
  case (Suc k)
  have  $f(Q x) \leq f(Q (Suc x))$  using 1(2) 2 by auto
  also have  $\dots \leq f(Q y)$  using Suc(2) by (intro Suc(1), auto)
  finally show ?case by this
qed
next
fix k
  have 3:  $P(Suc k)(Q(Suc k))$  using 2 by this
  have 4:  $Suc k \leq \text{length}(f(Q(Suc k)))$  using assms(3) 3 by this
  have 5:  $k < \text{length}(f(Q(Suc k)))$  using 4 by auto
  show  $\exists l. k < \text{length}(f(Q l))$  using 5 by blast
qed
qed
lemma chain-construct-2:
  assumes  $P 0 x_0 \wedge k x. P k x \implies \exists x'. P(Suc k) x' \wedge f x \leq f x' \wedge g x \leq g x'$ 
  assumes  $\wedge k x. P k x \implies k \leq \text{length}(f x) \wedge k x. P k x \implies k \leq \text{length}(g x)$ 
  obtains Q
  where  $\wedge k. P k(Q k)$  chain (f o Q) chain (g o Q)
proof -
  obtain x' where 1:
     $P 0 x_0 \wedge k x. P k x \implies P(Suc k)(x' k x) \wedge f x \leq f(x' k x) \wedge g x \leq g(x'$ 
     $k x)$ 
    using assms(1, 2) by metis
  define Q where  $Q \equiv \text{rec-nat } x_0 x'$ 
  have [simp]:  $Q 0 = x_0 \wedge k. Q(Suc k) = x' k(Q k)$  unfolding Q-def by
simp+
  have 2:  $\wedge k. P k(Q k)$ 
  proof -
    fix k
    show  $P k(Q k)$  using 1 by (induct k, auto)
  qed
  show ?thesis
proof (intro that chainI monoI, unfold comp-apply)
  fix k
  show  $P k(Q k)$  using 2 by this
next
fix x y :: nat
assume  $x \leq y$ 
thus  $f(Q x) \leq f(Q y)$ 
proof (induct y - x arbitrary: x y)
  case 0
  show ?case using 0 by simp
next
  case (Suc k)

```

```

    have  $f(Q x) \leq f(Q (Suc x))$  using 1(2) 2 by auto
    also have  $\dots \leq f(Q y)$  using Suc(2) by (intro Suc(1), auto)
    finally show ?case by this
  qed
next
  fix  $k$ 
  have 3:  $P(Suc k)(Q(Suc k))$  using 2 by this
  have 4:  $Suc k \leq \text{length}(f(Q(Suc k)))$  using assms(3) 3 by this
  have 5:  $k < \text{length}(f(Q(Suc k)))$  using 4 by auto
  show  $\exists l. k < \text{length}(f(Q l))$  using 5 by blast
next
  fix  $x y :: nat$ 
  assume  $x \leq y$ 
  thus  $g(Q x) \leq g(Q y)$ 
  proof (induct  $y - x$  arbitrary:  $x y$ )
    case 0
    show ?case using 0 by simp
  next
    case (Suc  $k$ )
    have  $g(Q x) \leq g(Q(Suc x))$  using 1(2) 2 by auto
    also have  $\dots \leq g(Q y)$  using Suc(2) by (intro Suc(1), auto)
    finally show ?case by this
  qed
next
  fix  $k$ 
  have 3:  $P(Suc k)(Q(Suc k))$  using 2 by this
  have 4:  $Suc k \leq \text{length}(g(Q(Suc k)))$  using assms(4) 3 by this
  have 5:  $k < \text{length}(g(Q(Suc k)))$  using 4 by auto
  show  $\exists l. k < \text{length}(g(Q l))$  using 5 by blast
qed
qed
lemma chain-construct-2':
  assumes  $P 0 u_0 v_0 \wedge k u v. P k u v \implies \exists u' v'. P(Suc k) u' v' \wedge u \leq u' \wedge v \leq v'$ 
  assumes  $\wedge k u v. P k u v \implies k \leq \text{length } u \wedge k u v. P k u v \implies k \leq \text{length } v$ 
  obtains  $u v$ 
  where  $\wedge k. P k (u k) (v k)$  chain  $u$  chain  $v$ 
proof -
  obtain  $Q$  where 1:  $\wedge k. (\text{case-prod} \circ P) k (Q k)$  chain  $(fst \circ Q)$  chain  $(snd \circ Q)$ 
  proof (rule chain-construct-2)
    show  $\exists x'. (\text{case-prod} \circ P) (Suc k) x' \wedge \text{fst } x \leq \text{fst } x' \wedge \text{snd } x \leq \text{snd } x'$ 
    if  $(\text{case-prod} \circ P) k x$  for  $k x$  using assms that by auto
    show  $(\text{case-prod} \circ P) 0 (u_0, v_0)$  using assms by auto
    show  $k \leq \text{length}(fst x)$  if  $(\text{case-prod} \circ P) k x$  for  $k x$  using assms that by auto
  auto
  show  $k \leq \text{length}(snd x)$  if  $(\text{case-prod} \circ P) k x$  for  $k x$  using assms that by auto
  auto
qed rule

```

```

    show ?thesis
  proof
    show  $P\ k\ ((fst \circ Q)\ k)\ ((snd \circ Q)\ k)$  for  $k$  using 1(1) by (auto simp:
prod.case-eq-if)
    show chain (fst  $\circ$  Q) chain (snd  $\circ$  Q) using 1(2, 3) by this
  qed
qed
end

```

4 Sets

theory Set-Extensions

imports

HOL-Library.Infinite-Set

begin

declare finite-subset[intro]

lemma set-not-emptyI[intro 0]: $x \in S \implies S \neq \{\}$ by auto

lemma sets-empty-iffI[intro 0]:

assumes $\bigwedge a. a \in A \implies \exists b. b \in B$

assumes $\bigwedge b. b \in B \implies \exists a. a \in A$

shows $A = \{\} \longleftrightarrow B = \{\}$

using assms by auto

lemma disjointI[intro 0]:

assumes $\bigwedge x. x \in A \implies x \in B \implies False$

shows $A \cap B = \{\}$

using assms by auto

lemma range-subsetI[intro 0]:

assumes $\bigwedge x. f\ x \in S$

shows $range\ f \subseteq S$

using assms by blast

lemma finite-imageI-range:

assumes finite (range f)

shows finite (f ' A)

using finite-subset image-mono subset-UNIV assms by metis

lemma inf-img-fin-domE':

assumes infinite A

assumes finite (f ' A)

obtains y

where $y \in f\ ' A$ infinite (A \cap f - ' {y})

proof (rule ccontr)

assume 1: $\neg thesis$

have 2: finite ($\bigcup y \in f\ ' A. A \cap f\ -\ ' \{y\}$)

proof (rule finite-UN-I)

show finite (f ' A) using assms(2) by this

show $\bigwedge y. y \in f \text{ ` } A \implies \text{finite } (A \cap f \text{ - ` } \{y\})$ **using** *that 1* **by** *blast*
qed
have $\exists: A \subseteq (\bigcup y \in f \text{ ` } A. A \cap f \text{ - ` } \{y\})$ **by** *blast*
show *False* **using** *assms(1) 2 3* **by** *blast*
qed

lemma *vimage-singleton[simp]:* $f \text{ - ` } \{y\} = \{x. f \ x = y\}$ **unfolding** *vimage-def*
by *simp*

lemma *these-alt-def:* $\text{Option.these } S = \text{Some - ` } S$ **unfolding** *Option.these-def*
by *force*

lemma *the-vimage-subset:* $\text{the - ` } \{a\} \subseteq \{\text{None}, \text{Some } a\}$ **by** *auto*

lemma *finite-induct-reverse[consumes 1, case-names remove]:*

assumes *finite S*

assumes $\bigwedge S. \text{finite } S \implies (\bigwedge x. x \in S \implies P (S - \{x\})) \implies P \ S$

shows $P \ S$

using *assms(1)*

proof (*induct rule: finite-psubset-induct*)

case (*psubset S*)

show *?case*

proof (*rule assms(2)*)

show *finite S* **using** *psubset(1)* **by** *this*

next

fix *x*

assume $0: x \in S$

show $P (S - \{x\})$

proof (*rule psubset(2)*)

show $S - \{x\} \subset S$ **using** 0 **by** *auto*

qed

qed

qed

lemma *zero-not-in-Suc-image[simp]:* $0 \notin \text{Suc ` } A$ **by** *auto*

lemma *Collect-split-Suc:*

$\neg P \ 0 \implies \{i. P \ i\} = \text{Suc ` } \{i. P \ (\text{Suc } i)\}$

$P \ 0 \implies \{i. P \ i\} = \{0\} \cup \text{Suc ` } \{i. P \ (\text{Suc } i)\}$

proof $-$

assume $\neg P \ 0$

thus $\{i. P \ i\} = \text{Suc ` } \{i. P \ (\text{Suc } i)\}$

by (*auto,metis image-eqI mem-Collect-eq nat.exhaust*)

next

assume $P \ 0$

thus $\{i. P \ i\} = \{0\} \cup \text{Suc ` } \{i. P \ (\text{Suc } i)\}$

by (*auto,metis imageI mem-Collect-eq not0-implies-Suc*)

qed

lemma *Collect-subsume[simp]:*

assumes $\bigwedge x. x \in A \implies P x$
shows $\{x \in A. P x\} = A$
using *assms unfolding simp-implies-def* **by** *auto*

lemma *Max-ge'*:
assumes *finite A A $\neq \{\}$*
assumes *b $\in A$ a \leq b*
shows *a \leq Max A*
using *assms Max-ge-iff* **by** *auto*

abbreviation *least A \equiv LEAST k. k $\in A$*

lemma *least-contains[intro?, simp]*:
fixes *A :: 'a :: wellorder set*
assumes *k $\in A$*
shows *least A $\in A$*
using *assms by (metis LeastI)*

lemma *least-contains'[intro?, simp]*:
fixes *A :: 'a :: wellorder set*
assumes *A $\neq \{\}$*
shows *least A $\in A$*
using *assms by (metis LeastI equalsOI)*

lemma *least-least[intro?, simp]*:
fixes *A :: 'a :: wellorder set*
assumes *k $\in A$*
shows *least A \leq k*
using *assms by (metis Least-le)*

lemma *least-unique*:
fixes *A :: 'a :: wellorder set*
assumes *k $\in A$ k \leq least A*
shows *k = least A*
using *assms by (metis Least-le antisym)*

lemma *least-not-less*:
fixes *A :: 'a :: wellorder set*
assumes *k < least A*
shows *k $\notin A$*
using *assms by (metis not-less-Least)*

lemma *leastI2-order[simp]*:
fixes *A :: 'a :: wellorder set*
assumes *A $\neq \{\}$ $\bigwedge k. k \in A \implies (\bigwedge l. l \in A \implies k \leq l) \implies P k$*
shows *P (least A)*

proof (*rule LeastI2-order*)
show *least A $\in A$* **using** *assms(1)* **by** *rule*

next
fix *k*
assume *1: k $\in A$*
show *least A \leq k* **using** *1* **by** *rule*
next
fix *k*

assume 1: $k \in A \forall l. l \in A \longrightarrow k \leq l$
show $P k$ **using** $assms(2)$ 1 **by** *auto*
qed

lemma *least-singleton[simp]*:
fixes $a :: 'a :: wellorder$
shows $least \{a\} = a$
by (*metis insert-not-empty least-contains' singletonD*)

lemma *least-image[simp]*:
fixes $f :: 'a :: wellorder \Rightarrow 'b :: wellorder$
assumes $A \neq \{\}$ $\wedge k l. k \in A \Longrightarrow l \in A \Longrightarrow k \leq l \Longrightarrow f k \leq f l$
shows $least (f ' A) = f (least A)$
proof (*rule leastI2-order*)
show $A \neq \{\}$ **using** $assms(1)$ **by** *this*
next
fix k
assume 1: $k \in A \wedge i. i \in A \Longrightarrow k \leq i$
show $least (f ' A) = f k$
proof (*rule leastI2-order*)
show $f ' A \neq \{\}$ **using** $assms(1)$ **by** *simp*
next
fix l
assume 2: $l \in f ' A \wedge i. i \in f ' A \Longrightarrow l \leq i$
show $l = f k$ **using** $assms(2)$ 1 2 **by** *force*
qed
qed

lemma *least-le*:
fixes $A B :: 'a :: wellorder set$
assumes $B \neq \{\}$
assumes $\wedge i. i \leq least A \Longrightarrow i \leq least B \Longrightarrow i \in B \Longrightarrow i \in A$
shows $least A \leq least B$
proof (*rule ccontr*)
assume 1: $\neg least A \leq least B$
have 2: $least B \in A$ **using** $assms(1, 2)$ 1 **by** *simp*
have 3: $least A \leq least B$ **using** 2 **by** *rule*
show *False* **using** 1 3 **by** *rule*
qed

lemma *least-eq*:
fixes $A B :: 'a :: wellorder set$
assumes $A \neq \{\}$ $B \neq \{\}$
assumes $\wedge i. i \leq least A \Longrightarrow i \leq least B \Longrightarrow i \in A \longleftrightarrow i \in B$
shows $least A = least B$
using $assms$ **by** (*auto intro: antisym least-le*)

lemma *least-Suc[simp]*:
assumes $A \neq \{\}$
shows $least (Suc ' A) = Suc (least A)$

proof (*rule antisym*)
obtain k **where** $10: k \in A$ **using** *assms* **by** *blast*
have $11: \text{Suc } k \in \text{Suc } 'A$ **using** 10 **by** *auto*
have $20: \text{least } A \in A$ **using** 10 *LeastI* **by** *metis*
have $21: \text{least } (\text{Suc } 'A) \in \text{Suc } 'A$ **using** 11 *LeastI* **by** *metis*
have $30: \bigwedge l. l \in A \implies \text{least } A \leq l$ **using** 10 *Least-le* **by** *metis*
have $31: \bigwedge l. l \in \text{Suc } 'A \implies \text{least } (\text{Suc } 'A) \leq l$ **using** 11 *Least-le* **by** *metis*
show $\text{least } (\text{Suc } 'A) \leq \text{Suc } (\text{least } A)$ **using** 20 31 **by** *auto*
show $\text{Suc } (\text{least } A) \leq \text{least } (\text{Suc } 'A)$ **using** 21 30 **by** *auto*
qed

lemma *least-Suc-diff[simp]*: $\text{Suc } 'A - \{\text{least } (\text{Suc } 'A)\} = \text{Suc } '(A - \{\text{least } A\})$

proof (*cases* $A = \{\}$)
case *True*
show *?thesis* **unfolding** *True* **by** *simp*
next
case *False*
have $\text{Suc } 'A - \{\text{least } (\text{Suc } 'A)\} = \text{Suc } 'A - \{\text{Suc } (\text{least } A)\}$ **using** *False* **by**
simp
also **have** $\dots = \text{Suc } 'A - \text{Suc } '\{\text{least } A\}$ **by** *simp*
also **have** $\dots = \text{Suc } '(A - \{\text{least } A\})$ **by** *blast*
finally **show** *?thesis* **by** *this*
qed

lemma *Max-diff-least[simp]*:

fixes $A :: 'a :: \text{wellorder set}$
assumes *finite* $A - \{\text{least } A\} \neq \{\}$
shows $\text{Max } (A - \{\text{least } A\}) = \text{Max } A$

proof –
have $1: \text{least } A \in A$ **using** *assms(2)* **by** *auto*
obtain a **where** $2: a \in A - \{\text{least } A\}$ **using** *assms(2)* **by** *blast*
have $\text{Max } A = \text{Max } (\text{insert } (\text{least } A) (A - \{\text{least } A\}))$ **using** *insert-absorb 1*
by *force*
also **have** $\dots = \text{max } (\text{least } A) (\text{Max } (A - \{\text{least } A\}))$
proof (*rule* *Max-insert*)
show *finite* $(A - \{\text{least } A\})$ **using** *assms(1)* **by** *auto*
show $A - \{\text{least } A\} \neq \{\}$ **using** *assms(2)* **by** *this*
qed
also **have** $\dots = \text{Max } (A - \{\text{least } A\})$
proof (*rule* *max-absorb2*, *rule* *Max-ge'*)
show *finite* $(A - \{\text{least } A\})$ **using** *assms(1)* **by** *auto*
show $A - \{\text{least } A\} \neq \{\}$ **using** *assms(2)* **by** *this*
show $a \in A - \{\text{least } A\}$ **using** 2 **by** *this*
show $\text{least } A \leq a$ **using** 2 **by** *simp*
qed
finally **show** *?thesis* **by** *rule*
qed

lemma *nat-set-card-equality-less*:

```

fixes A :: nat set
assumes x ∈ A y ∈ A card {z ∈ A. z < x} = card {z ∈ A. z < y}
shows x = y
proof (cases x y rule: linorder-cases)
  case less
    have 0: finite {z ∈ A. z < y} by simp
    have 1: {z ∈ A. z < x} ⊂ {z ∈ A. z < y} using assms(1, 2) less by force
    have 2: card {z ∈ A. z < x} < card {z ∈ A. z < y} using psubset-card-mono
  0 1 by this
  show ?thesis using assms(3) 2 by simp
next
  case equal
    show ?thesis using equal by this
next
  case greater
    have 0: finite {z ∈ A. z < x} by simp
    have 1: {z ∈ A. z < y} ⊂ {z ∈ A. z < x} using assms(1, 2) greater by force
    have 2: card {z ∈ A. z < y} < card {z ∈ A. z < x} using psubset-card-mono
  0 1 by this
  show ?thesis using assms(3) 2 by simp
qed

```

```

lemma nat-set-card-equality-le:
  fixes A :: nat set
  assumes x ∈ A y ∈ A card {z ∈ A. z ≤ x} = card {z ∈ A. z ≤ y}
  shows x = y
  proof (cases x y rule: linorder-cases)
    case less
      have 0: finite {z ∈ A. z ≤ y} by simp
      have 1: {z ∈ A. z ≤ x} ⊂ {z ∈ A. z ≤ y} using assms(1, 2) less by force
      have 2: card {z ∈ A. z ≤ x} < card {z ∈ A. z ≤ y} using psubset-card-mono
    0 1 by this
    show ?thesis using assms(3) 2 by simp
  next
  case equal
    show ?thesis using equal by this
  next
  case greater
    have 0: finite {z ∈ A. z ≤ x} by simp
    have 1: {z ∈ A. z ≤ y} ⊂ {z ∈ A. z ≤ x} using assms(1, 2) greater by force
    have 2: card {z ∈ A. z ≤ y} < card {z ∈ A. z ≤ x} using psubset-card-mono
  0 1 by this
  show ?thesis using assms(3) 2 by simp
qed

```

```

lemma nat-set-card-mono[simp]:
  fixes A :: nat set
  assumes x ∈ A
  shows card {z ∈ A. z < x} < card {z ∈ A. z < y} ↔ x < y

```



```

proof
  assume 1:  $\text{card } \{z \in A. z < x\} < \text{card } \{z \in A. z < y\}$ 
  show  $x < y$ 
  proof (rule ccontr)
    assume 2:  $\neg x < y$ 
    have 3:  $\text{card } \{z \in A. z < y\} \leq \text{card } \{z \in A. z < x\}$  using 2 by (auto intro: card-mono)
    show False using 1 3 by simp
  qed
next
  assume 1:  $x < y$ 
  show  $\text{card } \{z \in A. z < x\} < \text{card } \{z \in A. z < y\}$ 
  proof (intro psubset-card-mono psubsetI)
    show finite  $\{z \in A. z < y\}$  by simp
    show  $\{z \in A. z < x\} \subseteq \{z \in A. z < y\}$  using 1 by auto
    show  $\{z \in A. z < x\} \neq \{z \in A. z < y\}$  using assms 1 by blast
  qed
qed

lemma card-one[elim]:
  assumes  $\text{card } A = 1$ 
  obtains  $a$ 
  where  $A = \{a\}$ 
  using assms by (metis One-nat-def card-Suc-eq)

lemma image-alt-def:  $f \text{ ` } A = \{f x \mid x. x \in A\}$  by auto

lemma supset-mono-inductive[mono]:
  assumes  $\bigwedge x. x \in B \longrightarrow x \in C$ 
  shows  $A \subseteq B \longrightarrow A \subseteq C$ 
  using assms by auto
lemma Collect-mono-inductive[mono]:
  assumes  $\bigwedge x. P x \longrightarrow Q x$ 
  shows  $x \in \{x. P x\} \longrightarrow x \in \{x. Q x\}$ 
  using assms by auto

lemma image-union-split:
  assumes  $f \text{ ` } (A \cup B) = g \text{ ` } C$ 
  obtains  $D E$ 
  where  $f \text{ ` } A = g \text{ ` } D$   $f \text{ ` } B = g \text{ ` } E$   $D \subseteq C$   $E \subseteq C$ 
  using assms unfolding image-Un
  by (metis (erased, lifting) inf-sup-ord(3) inf-sup-ord(4) subset-imageE)
lemma image-insert-split:
  assumes inj  $g f \text{ ` } \text{insert } a B = g \text{ ` } C$ 
  obtains  $d E$ 
  where  $f a = g d$   $f \text{ ` } B = g \text{ ` } E$   $d \in C$   $E \subseteq C$ 
proof –
  have 1:  $f \text{ ` } (\{a\} \cup B) = g \text{ ` } C$  using assms(2) by simp
  obtain  $D E$  where 2:  $f \text{ ` } \{a\} = g \text{ ` } D$   $f \text{ ` } B = g \text{ ` } E$   $D \subseteq C$   $E \subseteq C$ 

```

```

    using image-union-split 1 by this
    obtain d where  $\exists: D = \{d\}$  using assms(1) 2(1) by (auto, metis (erased,
opaque-lifting) imageE
      image-empty image-insert inj-image-eq-iff singletonI)
    show ?thesis using that 2 unfolding  $\exists$  by simp
  qed

end

```

5 Basics

```

theory Basic-Extensions
imports HOL-Library.Infinite-Set
begin

```

5.1 Types

```

type-synonym 'a step = 'a  $\Rightarrow$  'a

```

5.2 Rules

```

declare less-imp-le[dest, simp]

```

```

declare le-funI[intro]
declare le-funE[elim]
declare le-funD[dest]

```

```

lemma IdI'[intro]:
  assumes  $x = y$ 
  shows  $(x, y) \in Id$ 
  using assms by auto

```

```

lemma (in order) order-le-cases:
  assumes  $x \leq y$ 
  obtains  $(eq) x = y \mid (lt) x < y$ 
  using assms le-less by auto

```

```

lemma (in linorder) linorder-cases':
  obtains  $(le) x \leq y \mid (gt) x > y$ 
  by force

```

```

lemma monoI-comp[intro]:
  assumes mono f mono g
  shows mono (f  $\circ$  g)
  using assms by (intro monoI, auto dest: monoD)

```

```

lemma strict-monoI-comp[intro]:
  assumes strict-mono f strict-mono g
  shows strict-mono (f  $\circ$  g)
  using assms by (intro strict-monoI, auto dest: strict-monoD)

```

```

lemma eq-le-absorb[simp]:
  fixes  $x\ y :: 'a :: \text{order}$ 
  shows  $x = y \wedge x \leq y \longleftrightarrow x = y\ x \leq y \wedge x = y \longleftrightarrow x = y$ 
  by auto

lemma INFM-Suc[simp]:  $(\exists_{\infty} i. P (Suc\ i)) \longleftrightarrow (\exists_{\infty} i. P\ i)$ 
  unfolding INFM-nat using Suc-lessE less-Suc-eq by metis
lemma INFM-plus[simp]:  $(\exists_{\infty} i. P (i + n :: \text{nat})) \longleftrightarrow (\exists_{\infty} i. P\ i)$ 
proof (induct  $n$ )
  case  $0$ 
  show ?case by simp
next
  case (Suc  $n$ )
  have  $(\exists_{\infty} i. P (i + Suc\ n)) \longleftrightarrow (\exists_{\infty} i. P (Suc\ i + n))$  by simp
  also have  $\dots \longleftrightarrow (\exists_{\infty} i. P (i + n))$  using INFM-Suc by this
  also have  $\dots \longleftrightarrow (\exists_{\infty} i. P\ i)$  using Suc by this
  finally show ?case by this
qed
lemma INFM-minus[simp]:  $(\exists_{\infty} i. P (i - n :: \text{nat})) \longleftrightarrow (\exists_{\infty} i. P\ i)$ 
proof (induct  $n$ )
  case  $0$ 
  show ?case by simp
next
  case (Suc  $n$ )
  have  $(\exists_{\infty} i. P (i - Suc\ n)) \longleftrightarrow (\exists_{\infty} i. P (Suc\ i - Suc\ n))$  using INFM-Suc
by meson
  also have  $\dots \longleftrightarrow (\exists_{\infty} i. P (i - n))$  by simp
  also have  $\dots \longleftrightarrow (\exists_{\infty} i. P\ i)$  using Suc by this
  finally show ?case by this
qed

```

5.3 Constants

```

definition const ::  $'a \Rightarrow 'b \Rightarrow 'a$ 
  where const  $x \equiv \lambda -. x$ 
definition const2 ::  $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a$ 
  where const2  $x \equiv \lambda - . x$ 
definition const3 ::  $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'a$ 
  where const3  $x \equiv \lambda - - . x$ 
definition const4 ::  $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'a$ 
  where const4  $x \equiv \lambda - - - . x$ 
definition const5 ::  $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow 'a$ 
  where const5  $x \equiv \lambda - - - - . x$ 

```

```

lemma const-apply[simp]:  $\text{const}\ x\ y = x$  unfolding const-def by rule
lemma const2-apply[simp]:  $\text{const2}\ x\ y\ z = x$  unfolding const2-def by rule
lemma const3-apply[simp]:  $\text{const3}\ x\ y\ z\ u = x$  unfolding const3-def by rule
lemma const4-apply[simp]:  $\text{const4}\ x\ y\ z\ u\ v = x$  unfolding const4-def by rule

```

lemma *const5-apply[simp]*: $\text{const5 } x y z u v w = x$ **unfolding** *const5-def* **by** *rule*

definition *zip-fun* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infixr** $\langle || \rangle$ 51)
where $f || g \equiv \lambda x. (f x, g x)$

lemma *zip-fun-simps[simp]*:
 $(f || g) x = (f x, g x)$
 $\text{fst} \circ (f || g) = f$
 $\text{snd} \circ (f || g) = g$
 $\text{fst} \circ h || \text{snd} \circ h = h$
 $\text{fst } ' \text{range } (f || g) = \text{range } f$
 $\text{snd } ' \text{range } (f || g) = \text{range } g$
unfolding *zip-fun-def* **by** *force+*

lemma *zip-fun-eq[dest]*:
assumes $f || g = h || i$
shows $f = h$ $g = i$
using *assms* **unfolding** *zip-fun-def* **by** (*auto dest: fun-cong*)

lemma *zip-fun-range-subset[intro, simp]*: $\text{range } (f || g) \subseteq \text{range } f \times \text{range } g$
unfolding *zip-fun-def* **by** *blast*

lemma *zip-fun-range-finite[elim]*:
assumes *finite* $(\text{range } (f || g))$
obtains *finite* $(\text{range } f)$ *finite* $(\text{range } g)$

proof
show *finite* $(\text{range } f)$ **using** *finite-imageI* [*OF* *assms*(1), *of fst*]
by (*simp add: image-image*)
show *finite* $(\text{range } g)$ **using** *finite-imageI* [*OF* *assms*(1), *of snd*]
by (*simp add: image-image*)
qed

lemma *zip-fun-split*:
obtains $f g$
where $h = f || g$

proof
show $h = \text{fst} \circ h || \text{snd} \circ h$ **by** *simp*
qed

abbreviation *None-None* $\equiv (None, None)$
abbreviation *None-Some* $\equiv \lambda (y). (None, Some y)$
abbreviation *Some-None* $\equiv \lambda (x). (Some x, None)$
abbreviation *Some-Some* $\equiv \lambda (x, y). (Some x, Some y)$

abbreviation *None-None-None* $\equiv (None, None, None)$
abbreviation *None-None-Some* $\equiv \lambda (z). (None, None, Some z)$
abbreviation *None-Some-None* $\equiv \lambda (y). (None, Some y, None)$
abbreviation *None-Some-Some* $\equiv \lambda (y, z). (None, Some y, Some z)$
abbreviation *Some-None-None* $\equiv \lambda (x). (Some x, None, None)$

abbreviation *Some-None-Some* $\equiv \lambda (x, z). (\text{Some } x, \text{None}, \text{Some } z)$
abbreviation *Some-Some-None* $\equiv \lambda (x, y). (\text{Some } x, \text{Some } y, \text{None})$
abbreviation *Some-Some-Some* $\equiv \lambda (x, y, z). (\text{Some } x, \text{Some } y, \text{Some } z)$

lemma *inj-Some2*[*simp*, *intro*]:

inj None-Some
inj Some-None
inj Some-Some
by (*rule injI*, *force*)+

lemma *inj-Some3*[*simp*, *intro*]:

inj None-None-Some
inj None-Some-None
inj None-Some-Some
inj Some-None-None
inj Some-None-Some
inj Some-Some-None
inj Some-Some-Some
by (*rule injI*, *force*)+

definition *swap* :: $'a \times 'b \Rightarrow 'b \times 'a$

where *swap* $x \equiv (\text{snd } x, \text{fst } x)$

lemma *swap-simps*[*simp*]: *swap* (a, b) = (b, a) **unfolding** *swap-def* **by** *simp*

lemma *swap-inj*[*intro*, *simp*]: *inj swap* **by** (*rule injI*, *auto*)

lemma *swap-surj*[*intro*, *simp*]: *surj swap* **by** (*rule surjI*[**where** $?f = \text{swap}$], *auto*)

lemma *swap-bij*[*intro*, *simp*]: *bij swap* **by** (*rule bijI*, *auto*)

definition *push* :: $('a \times 'b) \times 'c \Rightarrow 'a \times 'b \times 'c$

where *push* $x \equiv (\text{fst } (\text{fst } x), \text{snd } (\text{fst } x), \text{snd } x)$

definition *pull* :: $'a \times 'b \times 'c \Rightarrow ('a \times 'b) \times 'c$

where *pull* $x \equiv ((\text{fst } x, \text{fst } (\text{snd } x)), \text{snd } (\text{snd } x))$

lemma *push-simps*[*simp*]: *push* ($(x, y), z$) = (x, y, z) **unfolding** *push-def* **by** *simp*

lemma *pull-simps*[*simp*]: *pull* (x, y, z) = ($(x, y), z$) **unfolding** *pull-def* **by** *simp*

definition *label* :: $'vertex \times 'label \times 'vertex \Rightarrow 'label$

where *label* $\equiv \text{fst} \circ \text{snd}$

lemma *label-select*[*simp*]: *label* (p, a, q) = a **unfolding** *label-def* **by** *simp*

5.4 Theorems for @termcurry and @termsplit

lemma *curry-split*[*simp*]: *curry* \circ *case-prod* = *id* **by** *auto*

lemma *split-curry*[*simp*]: *case-prod* \circ *curry* = *id* **by** *auto*

lemma *curry-le*[simp]: $\text{curry } f \leq \text{curry } g \iff f \leq g$ **unfolding** *le-fun-def* **by** *force*

lemma *split-le*[simp]: $\text{case-prod } f \leq \text{case-prod } g \iff f \leq g$ **unfolding** *le-fun-def* **by** *force*

lemma *mono-curry-left*[simp]: $\text{mono } (\text{curry } \circ h) \iff \text{mono } h$
unfolding *mono-def* **by** *fastforce*

lemma *mono-split-left*[simp]: $\text{mono } (\text{case-prod } \circ h) \iff \text{mono } h$
unfolding *mono-def* **by** *fastforce*

lemma *mono-curry-right*[simp]: $\text{mono } (h \circ \text{curry}) \iff \text{mono } h$
unfolding *mono-def* *split-le*[*symmetric*] **by** *bestsimp*

lemma *mono-split-right*[simp]: $\text{mono } (h \circ \text{case-prod}) \iff \text{mono } h$
unfolding *mono-def* *curry-le*[*symmetric*] **by** *bestsimp*

lemma *Collect-curry*[simp]: $\{x. P (\text{curry } x)\} = \text{case-prod } \{x. P x\}$ **using** *image-Collect* **by** *fastforce*

lemma *Collect-split*[simp]: $\{x. P (\text{case-prod } x)\} = \text{curry } \{x. P x\}$ **using** *image-Collect* **by** *force*

lemma *gfp-split-curry*[simp]: $\text{gfp } (\text{case-prod } \circ f \circ \text{curry}) = \text{case-prod } (\text{gfp } f)$
proof –

have $\text{gfp } (\text{case-prod } \circ f \circ \text{curry}) = \text{Sup } \{u. u \leq \text{case-prod } (f (\text{curry } u))\}$
unfolding *gfp-def* **by** *simp*

also have $\dots = \text{Sup } \{u. \text{curry } u \leq \text{curry } (\text{case-prod } (f (\text{curry } u)))\}$ **unfolding** *curry-le* **by** *simp*

also have $\dots = \text{Sup } \{u. \text{curry } u \leq f (\text{curry } u)\}$ **by** *simp*

also have $\dots = \text{Sup } (\text{case-prod } \{u. u \leq f u\})$ **unfolding** *Collect-curry*[*of* $\lambda u. u \leq f u$] **by** *simp*

also have $\dots = \text{case-prod } (\text{Sup } \{u. u \leq f u\})$ **by** (*force simp add: image-comp*)

also have $\dots = \text{case-prod } (\text{gfp } f)$ **unfolding** *gfp-def* **by** *simp*

finally show *?thesis* **by** *this*

qed

lemma *gfp-curry-split*[simp]: $\text{gfp } (\text{curry } \circ f \circ \text{case-prod}) = \text{curry } (\text{gfp } f)$

proof –

have $\text{gfp } (\text{curry } \circ f \circ \text{case-prod}) = \text{Sup } \{u. u \leq \text{curry } (f (\text{case-prod } u))\}$
unfolding *gfp-def* **by** *simp*

also have $\dots = \text{Sup } \{u. \text{case-prod } u \leq \text{case-prod } (\text{curry } (f (\text{case-prod } u)))\}$
unfolding *split-le* **by** *simp*

also have $\dots = \text{Sup } \{u. \text{case-prod } u \leq f (\text{case-prod } u)\}$ **by** *simp*

also have $\dots = \text{Sup } (\text{curry } \{u. u \leq f u\})$ **unfolding** *Collect-split*[*of* $\lambda u. u \leq f u$] **by** *simp*

also have $\dots = \text{curry } (\text{Sup } \{u. u \leq f u\})$ **by** (*force simp add: image-comp*)

also have $\dots = \text{curry } (\text{gfp } f)$ **unfolding** *gfp-def* **by** *simp*

finally show *?thesis* **by** *this*

qed

lemma *not-someI*:

assumes $\bigwedge x. P x \implies \text{False}$

shows $\neg P (\text{SOME } x. P x)$

```

    using assms by blast
  lemma some-ccontr:
    assumes  $(\bigwedge x. \neg P x) \implies \text{False}$ 
    shows  $P$  (SOME  $x. P x$ )
    using assms someI-ex ccontr by metis

end

```

6 Relations

```

theory Relation-Extensions
imports
  Basic-Extensions
begin

```

```

  abbreviation rev-lex-prod (infixr  $\langle \langle *rlex* \rangle \rangle$  80)
    where  $r_1 \langle *rlex* \rangle r_2 \equiv \text{inv-image } (r_2 \langle *lex* \rangle r_1) \text{ swap}$ 

```

```

  lemmas sym-rtranclp[intro] = sym-rtrancl[to-pred]

```

```

  definition liftablep ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow \text{bool}$ 
    where  $\text{liftablep } r f \equiv \forall x y. r x y \longrightarrow r (f x) (f y)$ 

```

```

  lemma liftablepI[intro]:
    assumes  $\bigwedge x y. r x y \implies r (f x) (f y)$ 
    shows liftablep  $r f$ 
    using assms
    unfolding liftablep-def
    by simp

```

```

  lemma liftablepE[elim]:
    assumes liftablep  $r f$ 
    assumes  $r x y$ 
    obtains  $r (f x) (f y)$ 
    using assms
    unfolding liftablep-def
    by simp

```

```

  lemma liftablep-rtranclp:
    assumes liftablep  $r f$ 
    shows liftablep  $r^{**} f$ 

```

```

  proof
    fix  $x y$ 
    assume  $r^{**} x y$ 
    thus  $r^{**} (f x) (f y)$ 
      using assms
      by (induct rule: rtranclp-induct, force+)

```

```

  qed

```

```

  definition confluentp ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ 

```

where $\text{confluentp } r \equiv \forall x y1 y2. r^{**} x y1 \longrightarrow r^{**} x y2 \longrightarrow (\exists z. r^{**} y1 z \wedge r^{**} y2 z)$

lemma $\text{confluentpI}[\text{intro}]$:

assumes $\bigwedge x y1 y2. r^{**} x y1 \implies r^{**} x y2 \implies \exists z. r^{**} y1 z \wedge r^{**} y2 z$
shows $\text{confluentp } r$
using assms
unfolding confluentp-def
by simp

lemma $\text{confluentpE}[\text{elim}]$:

assumes $\text{confluentp } r$
assumes $r^{**} x y1 r^{**} x y2$
obtains z
where $r^{**} y1 z r^{**} y2 z$
using assms
unfolding confluentp-def
by blast

lemma $\text{confluentpI}'[\text{intro}]$:

assumes $\bigwedge x y1 y2. r^{**} x y1 \implies r x y2 \implies \exists z. r^{**} y1 z \wedge r^{**} y2 z$
shows $\text{confluentp } r$

proof

fix $x y1 y2$

assume $r^{**} x y1 r^{**} x y2$

thus $\exists z. r^{**} y1 z \wedge r^{**} y2 z$ **using** assms **by** ($\text{induct rule: rtranclp-induct, force+}$)

qed

lemma $\text{transclp-eq-implies-confluent-imp}$:

assumes $r1^{**} = r2^{**}$
assumes $\text{confluentp } r1$
shows $\text{confluentp } r2$
using assms
by force

lemma $\text{transclp-eq-implies-confluent-eq}$:

assumes $r1^{**} = r2^{**}$
shows $\text{confluentp } r1 \longleftrightarrow \text{confluentp } r2$
using $\text{assms transclp-eq-implies-confluent-imp}$
by metis

definition $\text{diamondp} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where $\text{diamondp } r \equiv \forall x y1 y2. r x y1 \longrightarrow r x y2 \longrightarrow (\exists z. r y1 z \wedge r y2 z)$

lemma $\text{diamondpI}[\text{intro}]$:

assumes $\bigwedge x y1 y2. r x y1 \implies r x y2 \implies \exists z. r y1 z \wedge r y2 z$
shows $\text{diamondp } r$
using assms

unfolding *diamondp-def*
by *simp*

lemma *diamondpE[elim]*:
assumes *diamondp r*
assumes *r x y1 r x y2*
obtains *z*
where *r y1 z r y2 z*
using *assms*
unfolding *diamondp-def*
by *blast*

lemma *diamondp-implies-confluentp*:
assumes *diamondp r*
shows *confluentp r*
proof (*rule confluentpI'*)
fix *x y1 y2*
assume *r** x y1 r x y2*
hence $\exists z. r y1 z \wedge r** y2 z$ **using** *assms* **by** (*induct rule: rtranclp-induct,*
force+)
thus $\exists z. r** y1 z \wedge r** y2 z$ **by** *blast*
qed

locale *wellfounded-relation* =
fixes *R :: 'a \Rightarrow 'a \Rightarrow bool*
assumes *wellfounded: wfP R*

end

7 Transition Systems

theory *Transition-System-Extensions*

imports

Basics/Word-Prefixes

Extensions/Set-Extensions

Extensions/Relation-Extensions

Transition-Systems-and-Automata.Transition-System

Transition-Systems-and-Automata.Transition-System-Extra

Transition-Systems-and-Automata.Transition-System-Construction

begin

context *transition-system-initial*

begin

definition *cycles* :: *'state \Rightarrow 'transition list set*
where *cycles p \equiv {w. path w p \wedge target w p = p}*

lemma *cyclesI[intro!]*:
assumes *path w p target w p = p*

```

    shows  $w \in \text{cycles } p$ 
    using assms unfolding cycles-def by auto
lemma cyclesE[elim!]:
    assumes  $w \in \text{cycles } p$ 
    obtains path  $w \ p \ \text{target } w \ p = p$ 
    using assms unfolding cycles-def by auto

inductive-set executable :: 'transition set
  where executable:  $p \in \text{nodes} \implies \text{enabled } a \ p \implies a \in \text{executable}$ 

lemma executableI-step[intro!]:
    assumes  $p \in \text{nodes} \ \text{enabled } a \ p$ 
    shows  $a \in \text{executable}$ 
    using executable assms by this
lemma executableI-words-fin[intro!]:
    assumes  $p \in \text{nodes} \ \text{path } w \ p$ 
    shows  $\text{set } w \subseteq \text{executable}$ 
    using assms by (induct w arbitrary: p, auto del: subsetI)
lemma executableE[elim?]:
    assumes  $a \in \text{executable}$ 
    obtains p
    where  $p \in \text{nodes} \ \text{enabled } a \ p$ 
    using assms by induct auto

end

locale transition-system-interpreted =
  transition-system ex en
  for ex :: 'action  $\Rightarrow$  'state  $\Rightarrow$  'state
  and en :: 'action  $\Rightarrow$  'state  $\Rightarrow$  bool
  and int :: 'state  $\Rightarrow$  'interpretation
begin

definition visible :: 'action set
  where visible  $\equiv \{a. \exists q. \text{en } a \ q \wedge \text{int } q \neq \text{int } (ex \ a \ q)\}$ 

lemma visibleI[intro]:
    assumes  $\text{en } a \ q \ \text{int } q \neq \text{int } (ex \ a \ q)$ 
    shows  $a \in \text{visible}$ 
    using assms unfolding visible-def by auto
lemma visibleE[elim]:
    assumes  $a \in \text{visible}$ 
    obtains q
    where  $\text{en } a \ q \ \text{int } q \neq \text{int } (ex \ a \ q)$ 
    using assms unfolding visible-def by auto

abbreviation invisible  $\equiv - \ \text{visible}$ 

lemma execute-fin-word-invisible:

```

```

    assumes path w p set w  $\subseteq$  invisible
    shows int (target w p) = int p
    using assms by (induct w arbitrary: p rule: list.induct, auto)
lemma execute-inf-word-invisible:
    assumes run w p k  $\leq$  l  $\wedge$  i. k  $\leq$  i  $\implies$  i < l  $\implies$  w !! i  $\notin$  visible
    shows int ((p ## trace w p) !! k) = int ((p ## trace w p) !! l)
proof -
    have (p ## trace w p) !! l = target (stake l w) p by simp
    also have stake l w = stake k w @ stake (l - k) (sdrop k w) using assms(2)
by simp
    also have target ... p = target (stake (l - k) (sdrop k w)) (target (stake k
w) p)
    unfolding fold-append comp-apply by rule
    also have int ... = int (target (stake k w) p)
    proof (rule execute-fin-word-invisible)
        have w = stake l w @- sdrop l w by simp
        also have stake l w = stake k w @ stake (l - k) (sdrop k w) using assms(2)
by simp
    finally have 1: run (stake k w @- stake (l - k) (sdrop k w) @- sdrop l w)
p
        unfolding shift-append using assms(1) by simp
    show path (stake (l - k) (sdrop k w)) (target (stake k w) p) using 1 by
auto
    show set (stake (l - k) (sdrop k w))  $\subseteq$  invisible using assms(3) by (auto
simp: set-stake-snth)
    qed
    also have ... = int ((p ## trace w p) !! k) by simp
    finally show ?thesis by rule
    qed
end

locale transition-system-complete =
    transition-system-initial ex en init +
    transition-system-interpreted ex en int
    for ex :: 'action  $\Rightarrow$  'state  $\Rightarrow$  'state
    and en :: 'action  $\Rightarrow$  'state  $\Rightarrow$  bool
    and init :: 'state  $\Rightarrow$  bool
    and int :: 'state  $\Rightarrow$  'interpretation
begin

definition language :: 'interpretation stream set
    where language  $\equiv$  {smap int (p ## trace w p) | p w. init p  $\wedge$  run w p}

lemma languageI[intro!]:
    assumes w = smap int (p ## trace v p) init p run v p
    shows w  $\in$  language
    using assms unfolding language-def by auto
lemma languageE[elim!]:

```

```

    assumes  $w \in \text{language}$ 
    obtains  $p v$ 
    where  $w = \text{smap int } (p \text{ \#\# trace } v p) \text{ init } p \text{ run } v p$ 
    using assms unfolding language-def by auto

end

locale transition-system-finite-nodes =
  transition-system-initial ex en init
  for  $ex :: 'action \Rightarrow 'state \Rightarrow 'state$ 
  and  $en :: 'action \Rightarrow 'state \Rightarrow \text{bool}$ 
  and  $init :: 'state \Rightarrow \text{bool}$ 
  +
  assumes reachable-finite: finite nodes

locale transition-system-cut =
  transition-system-finite-nodes ex en init
  for  $ex :: 'action \Rightarrow 'state \Rightarrow 'state$ 
  and  $en :: 'action \Rightarrow 'state \Rightarrow \text{bool}$ 
  and  $init :: 'state \Rightarrow \text{bool}$ 
  +
  fixes  $cuts :: 'action \text{ set}$ 
  assumes cycles-cut:  $p \in \text{nodes} \Longrightarrow w \in \text{cycles } p \Longrightarrow w \neq [] \Longrightarrow \text{set } w \cap \text{cuts}$ 
   $\neq \{\}$ 
begin

  inductive  $scut :: 'state \Rightarrow 'state \Rightarrow \text{bool}$ 
    where  $scut: p \in \text{nodes} \Longrightarrow en \ a \ p \Longrightarrow a \notin \text{cuts} \Longrightarrow scut \ p \ (ex \ a \ p)$ 

  declare  $scut.intros[intro!]$ 
  declare  $scut.cases[elim!]$ 

  lemma scut-reachable:
    assumes  $scut \ p \ q$ 
    shows  $p \in \text{nodes} \ q \in \text{nodes}$ 
    using assms by auto
  lemma scut-transl:
    assumes  $scut^{++} \ p \ q$ 
    obtains  $w$ 
    where  $path \ w \ p \ \text{target } w \ p = q \ \text{set } w \cap \text{cuts} = \{\} \ w \neq []$ 
  using assms
  proof (induct arbitrary: thesis)
    case (base q)
    show ?case using base by force
  next
  case (step q r)
  obtain  $w$  where 1:  $path \ w \ p \ \text{target } w \ p = q \ \text{set } w \cap \text{cuts} = \{\} \ w \neq []$ 
    using step(3) by this
  obtain  $a$  where 2:  $en \ a \ q \ a \notin \text{cuts} \ ex \ a \ q = r$  using step(2) by auto

```

```

show ?case
proof (rule step(4))
  show path (w @ [a]) p using 1 2 by auto
  show target (w @ [a]) p = r using 1 2 by auto
  show set (w @ [a]) ∩ cuts = {} using 1 2 by auto
  show w @ [a] ≠ [] by auto
qed
qed

sublocale wellfounded-relation scut-1-1
proof (unfold-locales, intro finite-acyclic-wf-converse[to-pred] acyclicI[to-pred],
safe)
  have 1: {(p, q). scut p q} ⊆ nodes × nodes using scut-reachable by blast
  have 2: finite (nodes × nodes)
    using finite-cartesian-product reachable-finite by blast
  show finite {(p, q). scut p q} using 1 2 by blast
next
  fix p
  assume 1: scut++ p p
  have 2: p ∈ nodes using 1 tranclE[to-pred] scut-reachable by metis
  obtain w where 3: path w p target w p = p set w ∩ cuts = {} w ≠ []
    using scut-trancl 1 by this
  have 4: w ∈ cycles p using 3(1, 2) by auto
  have 5: set w ∩ cuts ≠ {} using cycles-cut 2 4 3(4) by this
  show False using 3(3) 5 by simp
qed

lemma no-cut-scut:
  assumes p ∈ nodes en a p a ∉ cuts
  shows scut-1-1 (ex a p) p
  using assms by auto

end

locale transition-system-sticky =
  transition-system-complete ex en init int +
  transition-system-cut ex en init sticky
  for ex :: 'action ⇒ 'state ⇒ 'state
  and en :: 'action ⇒ 'state ⇒ bool
  and init :: 'state ⇒ bool
  and int :: 'state ⇒ 'interpretation
  and sticky :: 'action set
  +
  assumes executable-visible-sticky: executable ∩ visible ⊆ sticky

end

```

8 Trace Theory

```

theory Traces
imports Basics/Word-Prefixes
begin

  locale traces =
    fixes ind :: 'item  $\Rightarrow$  'item  $\Rightarrow$  bool
    assumes independence-symmetric[sym]: ind a b  $\Longrightarrow$  ind b a
  begin

    abbreviation Ind :: 'item set  $\Rightarrow$  'item set  $\Rightarrow$  bool
      where Ind A B  $\equiv$   $\forall$  a  $\in$  A.  $\forall$  b  $\in$  B. ind a b

    inductive eq-swap :: 'item list  $\Rightarrow$  'item list  $\Rightarrow$  bool (infix  $\langle =_S \rangle$  50)
      where swap: ind a b  $\Longrightarrow$  u @ [a] @ [b] @ v =S u @ [b] @ [a] @ v

    declare eq-swap.intros[intro]
    declare eq-swap.cases[elim]

    lemma eq-swap-sym[sym]: v =S w  $\Longrightarrow$  w =S v using independence-symmetric
  by auto

    lemma eq-swap-length[dest]: w1 =S w2  $\Longrightarrow$  length w1 = length w2 by force
    lemma eq-swap-range[dest]: w1 =S w2  $\Longrightarrow$  set w1 = set w2 by force

    lemma eq-swap-extend:
      assumes w1 =S w2
      shows u @ w1 @ v =S u @ w2 @ v
    using assms
    proof induct
      case (swap a b u' v')
      have u @ (u' @ [a] @ [b] @ v') @ v = (u @ u') @ [a] @ [b] @ (v' @ v) by
simp
      also have ... =S (u @ u') @ [b] @ [a] @ (v' @ v) using swap by blast
      also have ... = u @ (u' @ [b] @ [a] @ v') @ v by simp
      finally show ?case by this
    qed

    lemma eq-swap-remove1:
      assumes w1 =S w2
      obtains (equal) remove1 c w1 = remove1 c w2 | (swap) remove1 c w1 =S
remove1 c w2
      using assms
      proof induct
        case (swap a b u v)
        have c  $\notin$  set (u @ [a] @ [b] @ v)  $\vee$ 
          c  $\in$  set u  $\vee$ 
          c  $\notin$  set u  $\wedge$  c = a  $\vee$ 

```

```

c ∉ set u ∧ c ≠ a ∧ c = b ∨
c ∉ set u ∧ c ≠ a ∧ c ≠ b ∧ c ∈ set v
by auto
thus ?case
proof (elim disjE)
  assume 0: c ∉ set (u @ [a] @ [b] @ v)
  have 1: c ∉ set (u @ [b] @ [a] @ v) using 0 by auto
  have 2: remove1 c (u @ [a] @ [b] @ v) = u @ [a] @ [b] @ v using
remove1-idem 0 by this
  have 3: remove1 c (u @ [b] @ [a] @ v) = u @ [b] @ [a] @ v using
remove1-idem 1 by this
  have 4: remove1 c (u @ [a] @ [b] @ v) =S remove1 c (u @ [b] @ [a] @ v)
  unfolding 2 3 using eq-swap.intros swap(1) by this
  show thesis using swap(3) 4 by this
next
  assume 0: c ∈ set u
  have 2: remove1 c (u @ [a] @ [b] @ v) = remove1 c u @ [a] @ [b] @ v
  unfolding remove1-append using 0 by simp
  have 3: remove1 c (u @ [b] @ [a] @ v) = remove1 c u @ [b] @ [a] @ v
  unfolding remove1-append using 0 by simp
  have 4: remove1 c (u @ [a] @ [b] @ v) =S remove1 c (u @ [b] @ [a] @ v)
  unfolding 2 3 using eq-swap.intros swap(1) by this
  show thesis using swap(3) 4 by this
next
  assume 0: c ∉ set u ∧ c = a
  have 2: remove1 c (u @ [a] @ [b] @ v) = u @ [b] @ v
  unfolding remove1-append using remove1-idem 0 by auto
  have 3: remove1 c (u @ [b] @ [a] @ v) = u @ [b] @ v
  unfolding remove1-append using remove1-idem 0 by auto
  have 4: remove1 c (u @ [a] @ [b] @ v) = remove1 c (u @ [b] @ [a] @ v)
  unfolding 2 3 by rule
  show thesis using swap(2) 4 by this
next
  assume 0: c ∉ set u ∧ c ≠ a ∧ c = b
  have 2: remove1 c (u @ [a] @ [b] @ v) = u @ [a] @ v
  unfolding remove1-append using remove1-idem 0 by auto
  have 3: remove1 c (u @ [b] @ [a] @ v) = u @ [a] @ v
  unfolding remove1-append using remove1-idem 0 by auto
  have 4: remove1 c (u @ [a] @ [b] @ v) = remove1 c (u @ [b] @ [a] @ v)
  unfolding 2 3 by rule
  show thesis using swap(2) 4 by this
next
  assume 0: c ∉ set u ∧ c ≠ a ∧ c ≠ b ∧ c ∈ set v
  have 2: remove1 c (u @ [a] @ [b] @ v) = u @ [a] @ [b] @ remove1 c v
  unfolding remove1-append using 0 by simp
  have 3: remove1 c (u @ [b] @ [a] @ v) = u @ [b] @ [a] @ remove1 c v
  unfolding remove1-append using 0 by simp
  have 4: remove1 c (u @ [a] @ [b] @ v) =S remove1 c (u @ [b] @ [a] @ v)
  unfolding 2 3 using eq-swap.intros swap(1) by this

```

```

    show ?thesis using swap(3) 4 by this
  qed
qed

lemma eq-swap-rev:
  assumes  $w_1 =_S w_2$ 
  shows  $rev\ w_1 =_S rev\ w_2$ 
using assms
proof induct
  case (swap a b u v)
  have 1:  $rev\ v @ [a] @ [b] @ rev\ u =_S rev\ v @ [b] @ [a] @ rev\ u$  using swap
by blast
  have 2:  $rev\ v @ [b] @ [a] @ rev\ u =_S rev\ v @ [a] @ [b] @ rev\ u$  using 1
eq-swap-sym by blast
  show ?case using 2 by simp
qed

abbreviation eq-fin :: 'item list  $\Rightarrow$  'item list  $\Rightarrow$  bool (infix <=F> 50)
  where eq-fin  $\equiv$  eq-swap**

lemma eq-fin-symp[intro, sym]:  $u =_F v \Longrightarrow v =_F u$ 
  using eq-swap-sym sym-rtrancl[to-pred] unfolding symp-def by metis

lemma eq-fin-length[dest]:  $w_1 =_F w_2 \Longrightarrow length\ w_1 = length\ w_2$ 
  by (induct rule: rtrancl.induct, auto)
lemma eq-fin-range[dest]:  $w_1 =_F w_2 \Longrightarrow set\ w_1 = set\ w_2$ 
  by (induct rule: rtrancl.induct, auto)

lemma eq-fin-remove1:
  assumes  $w_1 =_F w_2$ 
  shows  $remove1\ c\ w_1 =_F remove1\ c\ w_2$ 
using assms
proof induct
  case (base)
  show ?case by simp
next
  case (step w2 w3)
  show ?case
  using step(2)
  proof (cases rule: eq-swap-remove1[where ?c = c])
    case equal
    show ?thesis using step equal by simp
  next
    case swap
    show ?thesis using step swap by auto
  qed
qed
qed

lemma eq-fin-rev:

```


assumes $w_1 =_F w_2$
shows $\text{rev } w_1 =_F \text{rev } w_2$
using *assms* **by** (*induct*, *auto dest: eq-swap-rev*)

lemma *eq-fin-concat-eq-fin-start*:

assumes $u @ v_1 =_F u @ v_2$
shows $v_1 =_F v_2$

using *assms*

proof (*induct u arbitrary: v1 v2 rule: rev-induct*)

case (*Nil*)

show *?case* **using** *Nil* **by** *simp*

next

case (*snoc a u*)

have 1: $u @ [a] @ v_1 =_F u @ [a] @ v_2$ **using** *snoc(2)* **by** *simp*

have 2: $[a] @ v_1 =_F [a] @ v_2$ **using** *snoc(1) 1* **by** *this*

show *?case* **using** *eq-fin-remove1[OF 2, of a]* **by** *simp*

qed

lemma *eq-fin-concat*: $u @ w_1 @ v =_F u @ w_2 @ v \longleftrightarrow w_1 =_F w_2$

proof

assume 0: $u @ w_1 @ v =_F u @ w_2 @ v$

have 1: $w_1 @ v =_F w_2 @ v$ **using** *eq-fin-concat-eq-fin-start 0* **by** *this*

have 2: $\text{rev } (w_1 @ v) =_F \text{rev } (w_2 @ v)$ **using** 1 **by** (*blast dest: eq-fin-rev*)

have 3: $\text{rev } v @ \text{rev } w_1 =_F \text{rev } v @ \text{rev } w_2$ **using** 2 **by** *simp*

have 4: $\text{rev } w_1 =_F \text{rev } w_2$ **using** *eq-fin-concat-eq-fin-start 3* **by** *this*

have 5: $\text{rev } (\text{rev } w_1) =_F \text{rev } (\text{rev } w_2)$ **using** 4 **by** (*blast dest: eq-fin-rev*)

show $w_1 =_F w_2$ **using** 5 **by** *simp*

next

show $u @ w_1 @ v =_F u @ w_2 @ v$ **if** $w_1 =_F w_2$

using *that* **by** (*induct*, *auto dest: eq-swap-extend[of - - u v]*)

qed

lemma *eq-fin-concat-start[iff]*: $w @ w_1 =_F w @ w_2 \longleftrightarrow w_1 =_F w_2$

using *eq-fin-concat[of w - []]* **by** *simp*

lemma *eq-fin-concat-end[iff]*: $w_1 @ w =_F w_2 @ w \longleftrightarrow w_1 =_F w_2$

using *eq-fin-concat[of [] - w]* **by** *simp*

lemma *ind-eq-fin'*:

assumes *Ind {a} (set v)*

shows $[a] @ v =_F v @ [a]$

using *assms*

proof (*induct v*)

case (*Nil*)

show *?case* **by** *simp*

next

case (*Cons b v*)

have 1: *Ind {a} (set v)* **using** *Cons(2)* **by** *auto*

have 2: *ind a b* **using** *Cons(2)* **by** *auto*

have $[a] @ b \# v = [a] @ [b] @ v$ **by** *simp*

also have $\dots =_S [b] @ [a] @ v$ **using** *eq-swap.intros[OF 2, of []]* **by** *auto*

also have $\dots =_F [b] @ v @ [a]$ **using** *Cons(1) 1* **by** *blast*
also have $\dots = (b \# v) @ [a]$ **by** *simp*
finally show *?case* **by** *this*
qed

lemma *ind-eq-fin[intro]*:
assumes *Ind (set u) (set v)*
shows $u @ v =_F v @ u$
using *assms*
proof (*induct u*)
case (*Nil*)
show *?case* **by** *simp*
next
case (*Cons a u*)
have *1: Ind (set u) (set v)* **using** *Cons(2)* **by** *auto*
have *2: Ind {a} (set v)* **using** *Cons(2)* **by** *auto*
have $(a \# u) @ v = [a] @ u @ v$ **by** *simp*
also have $\dots =_F [a] @ v @ u$ **using** *Cons(1) 1* **by** *blast*
also have $\dots = ([a] @ v) @ u$ **by** *simp*
also have $\dots =_F (v @ [a]) @ u$ **using** *ind-eq-fin' 2* **by** *blast*
also have $\dots = v @ (a \# u)$ **by** *simp*
finally show *?case* **by** *this*
qed

definition *le-fin* :: *'item list* \Rightarrow *'item list* \Rightarrow *bool* (**infix** \preceq_F 50)
where $w_1 \preceq_F w_2 \equiv \exists v_1. w_1 @ v_1 =_F w_2$

lemma *le-finI[intro 0]*:
assumes $w_1 @ v_1 =_F w_2$
shows $w_1 \preceq_F w_2$
using *assms* **unfolding** *le-fin-def* **by** *auto*

lemma *le-finE[elim 0]*:
assumes $w_1 \preceq_F w_2$
obtains v_1
where $w_1 @ v_1 =_F w_2$
using *assms* **unfolding** *le-fin-def* **by** *auto*

lemma *le-fin-empty[simp]*: $[] \preceq_F w$ **by** *force*
lemma *le-fin-trivial[intro]*: $w_1 =_F w_2 \Longrightarrow w_1 \preceq_F w_2$
proof
assume *1: w₁ =_F w₂*
show $w_1 @ [] =_F w_2$ **using** *1* **by** *simp*
qed

lemma *le-fin-length[dest]*: $w_1 \preceq_F w_2 \Longrightarrow \text{length } w_1 \leq \text{length } w_2$ **by** *force*
lemma *le-fin-range[dest]*: $w_1 \preceq_F w_2 \Longrightarrow \text{set } w_1 \subseteq \text{set } w_2$ **by** *force*

lemma *eq-fin-alt-def*: $w_1 =_F w_2 \longleftrightarrow w_1 \preceq_F w_2 \wedge w_2 \preceq_F w_1$
proof

show $w_1 \preceq_F w_2 \wedge w_2 \preceq_F w_1$ **if** $w_1 =_F w_2$ **using** *that* **by** *blast*
next
assume $0: w_1 \preceq_F w_2 \wedge w_2 \preceq_F w_1$
have $1: w_1 \preceq_F w_2$ $w_2 \preceq_F w_1$ **using** 0 **by** *auto*
have $10: \text{length } w_1 = \text{length } w_2$ **using** 1 **by** *force*
obtain $v_1 v_2$ **where** $2: w_1 @ v_1 =_F w_2$ $w_2 @ v_2 =_F w_1$ **using** 1 **by** (*elim*
le-finE)
have $3: \text{length } w_1 = \text{length } (w_1 @ v_1)$ **using** 2 10 **by** *force*
have $4: w_1 = w_1 @ v_1$ **using** 3 **by** *auto*
have $5: \text{length } w_2 = \text{length } (w_2 @ v_2)$ **using** 2 10 **by** *force*
have $6: w_2 = w_2 @ v_2$ **using** 5 **by** *auto*
show $w_1 =_F w_2$ **using** 4 6 2 **by** *simp*
qed

lemma *le-fin-reflp*[*simp, intro*]: $w \preceq_F w$ **by** *auto*
lemma *le-fin-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
proof –
obtain v_1 **where** $1: w_1 @ v_1 =_F w_2$ **using** *assms(1)* **by** *rule*
obtain v_2 **where** $2: w_2 @ v_2 =_F w_3$ **using** *assms(2)* **by** *rule*
show *?thesis*
proof
have $w_1 @ v_1 @ v_2 = (w_1 @ v_1) @ v_2$ **by** *simp*
also have $\dots =_F w_2 @ v_2$ **using** 1 **by** *blast*
also have $\dots =_F w_3$ **using** 2 **by** *blast*
finally show $w_1 @ v_1 @ v_2 =_F w_3$ **by** *this*
qed

qed
lemma *eq-fin-le-fin-transp*[*intro, trans*]:
assumes $w_1 =_F w_2$ $w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
using *assms* **by** *auto*
lemma *le-fin-eq-fin-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 =_F w_3$
shows $w_1 \preceq_F w_3$
using *assms* **by** *auto*
lemma *prefix-le-fin-transp*[*intro, trans*]:
assumes $w_1 \leq w_2$ $w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
proof –
obtain v_1 **where** $1: w_2 = w_1 @ v_1$ **using** *assms(1)* **by** *rule*
obtain v_2 **where** $2: w_2 @ v_2 =_F w_3$ **using** *assms(2)* **by** *rule*
show *?thesis*
proof
show $w_1 @ v_1 @ v_2 =_F w_3$ **using** 1 2 **by** *simp*
qed

qed
lemma *le-fin-prefix-transp*[*intro, trans*]:

```

assumes  $w_1 \preceq_F w_2$   $w_2 \leq w_3$ 
shows  $w_1 \preceq_F w_3$ 
proof –
  obtain  $v_1$  where  $1: w_1 @ v_1 =_F w_2$  using assms(1) by rule
  obtain  $v_2$  where  $2: w_3 = w_2 @ v_2$  using assms(2) by rule
  show ?thesis
proof
  have  $w_1 @ v_1 @ v_2 = (w_1 @ v_1) @ v_2$  by simp
  also have  $\dots =_F w_2 @ v_2$  using 1 by blast
  also have  $\dots = w_3$  using 2 by simp
  finally show  $w_1 @ v_1 @ v_2 =_F w_3$  by this
qed
qed
lemma prefix-eq-fin-transp[intro, trans]:
  assumes  $w_1 \leq w_2$   $w_2 =_F w_3$ 
  shows  $w_1 \preceq_F w_3$ 
  using assms by auto

lemma le-fin-concat-start[iff]:  $w @ w_1 \preceq_F w @ w_2 \longleftrightarrow w_1 \preceq_F w_2$ 
proof
  assume  $0: w @ w_1 \preceq_F w @ w_2$ 
  obtain  $v_1$  where  $1: w @ w_1 @ v_1 =_F w @ w_2$  using 0 by auto
  show  $w_1 \preceq_F w_2$  using 1 by auto
next
  assume  $0: w_1 \preceq_F w_2$ 
  obtain  $v_1$  where  $1: w_1 @ v_1 =_F w_2$  using 0 by auto
  have  $2: (w @ w_1) @ v_1 =_F w @ w_2$  using 1 by auto
  show  $w @ w_1 \preceq_F w @ w_2$  using 2 by blast
qed
lemma le-fin-concat-end[dest]:
  assumes  $w_1 \preceq_F w_2$ 
  shows  $w_1 \preceq_F w_2 @ w$ 
proof –
  obtain  $v_1$  where  $1: w_1 @ v_1 =_F w_2$  using assms by rule
  show ?thesis
proof
  have  $w_1 @ v_1 @ w = (w_1 @ v_1) @ w$  by simp
  also have  $\dots =_F w_2 @ w$  using 1 by blast
  finally show  $w_1 @ v_1 @ w =_F w_2 @ w$  by this
qed
qed

definition le-fininf :: 'item list  $\Rightarrow$  'item stream  $\Rightarrow$  bool (infix  $\langle \preceq_{FI} \rangle$  50)
  where  $w_1 \preceq_{FI} w_2 \equiv \exists v_2. v_2 \leq_{FI} w_2 \wedge w_1 \preceq_F v_2$ 

lemma le-fininfI[intro 0]:
  assumes  $v_2 \leq_{FI} w_2$   $w_1 \preceq_F v_2$ 
  shows  $w_1 \preceq_{FI} w_2$ 
  using assms unfolding le-fininf-def by auto

```

lemma *le-fininfE*[*elim 0*]:
assumes $w_1 \preceq_{FI} w_2$
obtains v_2
where $v_2 \leq_{FI} w_2$ $w_1 \preceq_F v_2$
using *assms unfolding le-fininf-def by auto*

lemma *le-fininf-empty*[*simp*]: $[] \preceq_{FI} w$ **by force**

lemma *le-fininf-range*[*dest*]: $w_1 \preceq_{FI} w_2 \implies \text{set } w_1 \subseteq \text{sset } w_2$ **by force**

lemma *eq-fin-le-fininf-transp*[*intro, trans*]:
assumes $w_1 =_F w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by blast*

lemma *le-fin-le-fininf-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by blast*

lemma *prefix-le-fininf-transp*[*intro, trans*]:
assumes $w_1 \leq w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by auto*

lemma *le-fin-prefix-fininf-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by auto*

lemma *eq-fin-prefix-fininf-transp*[*intro, trans*]:
assumes $w_1 =_F w_2$ $w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by auto*

lemma *le-fininf-concat-start*[*iff*]: $w @ w_1 \preceq_{FI} w @- w_2 \longleftrightarrow w_1 \preceq_{FI} w_2$
proof
assume 0 : $w @ w_1 \preceq_{FI} w @- w_2$
obtain v_2 **where** 1 : $v_2 \leq_{FI} w @- w_2$ $w @ w_1 \preceq_F v_2$ **using** 0 **by rule**
have 2 : $\text{length } w \leq \text{length } v_2$ **using** $1(2)$ **by force**
have 4 : $w \leq v_2$ **using** *prefix-fininf-extend[OF 1(1) 2]* **by this**
obtain v_1 **where** 5 : $v_2 = w @ v_1$ **using** 4 **by rule**
show $w_1 \preceq_{FI} w_2$
proof
show $v_1 \leq_{FI} w_2$ **using** $1(1)$ **unfolding** 5 **by auto**
show $w_1 \preceq_F v_1$ **using** $1(2)$ **unfolding** 5 **by simp**
qed

next
assume 0 : $w_1 \preceq_{FI} w_2$
obtain v_2 **where** 1 : $v_2 \leq_{FI} w_2$ $w_1 \preceq_F v_2$ **using** 0 **by rule**
show $w @ w_1 \preceq_{FI} w @- w_2$
proof
show $w @ v_2 \leq_{FI} (w @- w_2)$ **using** $1(1)$ **by auto**

show $w @ w_1 \preceq_F w @ v_2$ **using** 1(2) **by** *auto*
qed
qed

lemma *le-fininf-singleton*[*intro, simp*]: $[shd\ v] \preceq_{FI} v$
proof –
have $[shd\ v] \preceq_{FI} shd\ v \#\# sdrop\ 1\ v$ **by** *blast*
also have $\dots = v$ **by** *simp*
finally show *?thesis* **by** *this*
qed

definition *le-inf* :: $'item\ stream \Rightarrow 'item\ stream \Rightarrow bool$ (**infix** $\langle \preceq_I \rangle$ 50)
where $w_1 \preceq_I w_2 \equiv \forall v_1. v_1 \leq_{FI} w_1 \longrightarrow v_1 \preceq_{FI} w_2$

lemma *le-infI*[*intro 0*]:
assumes $\bigwedge v_1. v_1 \leq_{FI} w_1 \Longrightarrow v_1 \preceq_{FI} w_2$
shows $w_1 \preceq_I w_2$
using *assms unfolding le-inf-def* **by** *auto*
lemma *le-infE*[*elim 0*]:
assumes $w_1 \preceq_I w_2\ v_1 \leq_{FI} w_1$
obtains $v_1 \preceq_{FI} w_2$
using *assms unfolding le-inf-def* **by** *auto*

lemma *le-inf-range*[*dest*]:
assumes $w_1 \preceq_I w_2$
shows $sset\ w_1 \subseteq sset\ w_2$
proof
fix *a*
assume 1: $a \in sset\ w_1$
obtain *i* **where** 2: $a = w_1 !! i$ **using** 1 **by** (*metis imageE sset-range*)
have 3: $stake\ (Suc\ i)\ w_1 \leq_{FI} w_1$ **by** *rule*
have 4: $stake\ (Suc\ i)\ w_1 \preceq_{FI} w_2$ **using** *assms 3* **by** *rule*
have 5: $w_1 !! i \in set\ (stake\ (Suc\ i)\ w_1)$ **by** (*meson lessI set-stake-snth*)
show $a \in sset\ w_2$ **unfolding** 2 **using** 5 4 **by** *fastforce*
qed

lemma *le-inf-reflp*[*simp, intro*]: $w \preceq_I w$ **by** *auto*
lemma *prefix-fininf-le-inf-transp*[*intro, trans*]:
assumes $w_1 \leq_{FI} w_2\ w_2 \preceq_I w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by blast*
lemma *le-fininf-le-inf-transp*[*intro, trans*]:
assumes $w_1 \preceq_{FI} w_2\ w_2 \preceq_I w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms by blast*
lemma *le-inf-transp*[*intro, trans*]:
assumes $w_1 \preceq_I w_2\ w_2 \preceq_I w_3$
shows $w_1 \preceq_I w_3$
using *assms by blast*

lemma *le-infI'*:
assumes $\bigwedge k. \exists v. v \leq_{FI} w_1 \wedge k < \text{length } v \wedge v \preceq_{FI} w_2$
shows $w_1 \preceq_I w_2$
proof
fix u
assume $1: u \leq_{FI} w_1$
obtain v **where** $2: v \leq_{FI} w_1$ $\text{length } u < \text{length } v$ $v \preceq_{FI} w_2$ **using** *assms* **by**
auto
have $3: \text{length } u \leq \text{length } v$ **using** $2(2)$ **by** *auto*
have $4: u \leq v$ **using** *prefix-fininf-length* 1 $2(1)$ 3 **by** *this*
show $u \preceq_{FI} w_2$ **using** 4 $2(3)$ **by** *rule*
qed

lemma *le-infI-chain-left*:
assumes $\text{chain } w \wedge k. w k \preceq_{FI} v$
shows $\text{limit } w \preceq_I v$
proof (*rule le-infI'*)
fix k
obtain l **where** $1: k < \text{length } (w l)$ **using** *assms(1)* **by** *rule*
show $\exists va. va \leq_{FI} \text{limit } w \wedge k < \text{length } va \wedge va \preceq_{FI} v$
proof (*intro exI conjI*)
show $w l \leq_{FI} \text{limit } w$ **using** *chain-prefix-limit* *assms(1)* **by** *this*
show $k < \text{length } (w l)$ **using** 1 **by** *this*
show $w l \preceq_{FI} v$ **using** *assms(2)* **by** *this*
qed

lemma *le-infI-chain-right*:
assumes $\text{chain } w \wedge u. u \leq_{FI} v \implies u \preceq_F w (l u)$
shows $v \preceq_I \text{limit } w$
proof
fix u
assume $1: u \leq_{FI} v$
show $u \preceq_{FI} \text{limit } w$
proof
show $w (l u) \leq_{FI} \text{limit } w$ **using** *chain-prefix-limit* *assms(1)* **by** *this*
show $u \preceq_F w (l u)$ **using** *assms(2)* 1 **by** *this*
qed

lemma *le-infI-chain-right'*:
assumes $\text{chain } w \wedge k. \text{stake } k v \preceq_F w (l k)$
shows $v \preceq_I \text{limit } w$
proof (*rule le-infI-chain-right*)
show $\text{chain } w$ **using** *assms(1)* **by** *this*
next
fix u
assume $1: u \leq_{FI} v$
have $2: \text{stake } (\text{length } u) v = u$ **using** 1 **by** (*simp add: prefix-fininf-def shift-eq*)
have $3: \text{stake } (\text{length } u) v \preceq_F w (l (\text{length } u))$ **using** *assms(2)* **by** *this*

show $u \preceq_F w$ (l ($length\ u$)) **using** 3 **unfolding** 2 **by this**
qed

definition $eq\text{-}inf :: 'item\ stream \Rightarrow 'item\ stream \Rightarrow bool$ (**infix** $\langle =_I \rangle$ 50)
where $w_1 =_I w_2 \equiv w_1 \preceq_I w_2 \wedge w_2 \preceq_I w_1$

lemma $eq\text{-}infI$ [*intro 0*]:
assumes $w_1 \preceq_I w_2$ $w_2 \preceq_I w_1$
shows $w_1 =_I w_2$
using *assms* **unfolding** $eq\text{-}inf\text{-}def$ **by auto**

lemma $eq\text{-}infE$ [*elim 0*]:
assumes $w_1 =_I w_2$
obtains $w_1 \preceq_I w_2$ $w_2 \preceq_I w_1$
using *assms* **unfolding** $eq\text{-}inf\text{-}def$ **by auto**

lemma $eq\text{-}inf\text{-}range$ [*dest*]: $w_1 =_I w_2 \implies sset\ w_1 = sset\ w_2$ **by force**

lemma $eq\text{-}inf\text{-}reflp$ [*simp, intro*]: $w =_I w$ **by auto**

lemma $eq\text{-}inf\text{-}symp$ [*intro*]: $w_1 =_I w_2 \implies w_2 =_I w_1$ **by auto**

lemma $eq\text{-}inf\text{-}transp$ [*intro, trans*]:

assumes $w_1 =_I w_2$ $w_2 =_I w_3$
shows $w_1 =_I w_3$
using *assms* **by blast**

lemma $le\text{-}fininf\text{-}eq\text{-}inf\text{-}transp$ [*intro, trans*]:

assumes $w_1 \preceq_{FI} w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms* **by blast**

lemma $le\text{-}inf\text{-}eq\text{-}inf\text{-}transp$ [*intro, trans*]:

assumes $w_1 \preceq_I w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_I w_3$
using *assms* **by blast**

lemma $eq\text{-}inf\text{-}le\text{-}inf\text{-}transp$ [*intro, trans*]:

assumes $w_1 =_I w_2$ $w_2 \preceq_I w_3$
shows $w_1 \preceq_I w_3$
using *assms* **by blast**

lemma $prefix\text{-}fininf\text{-}eq\text{-}inf\text{-}transp$ [*intro, trans*]:

assumes $w_1 \leq_{FI} w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_{FI} w_3$
using *assms* **by blast**

lemma $le\text{-}inf\text{-}concat\text{-}start$ [*iff*]: $w @- w_1 \preceq_I w @- w_2 \longleftrightarrow w_1 \preceq_I w_2$

proof

assume 1: $w @- w_1 \preceq_I w @- w_2$

show $w_1 \preceq_I w_2$

proof

fix v_1

assume 2: $v_1 \leq_{FI} w_1$

have $w @ v_1 \leq_{FI} w @- w_1$ **using** 2 **by auto**

also have $\dots \preceq_I w @- w_2$ **using** 1 **by this**


```

    finally show  $v_1 \preceq_{FI} w_2$  by rule
  qed
next
assume 1:  $w_1 \preceq_I w_2$ 
show  $w @- w_1 \preceq_I w @- w_2$ 
proof
  fix  $v_1$ 
  assume 2:  $v_1 \leq_{FI} w @- w_1$ 
  then show  $v_1 \preceq_{FI} w @- w_2$ 
  proof (cases rule: prefix-fininf-append)
    case (absorb)
    show ?thesis using absorb by auto
  next
    case (extend z)
    show ?thesis using 1 extend by auto
  qed
qed
qed
lemma eq-fin-le-inf-concat-end[dest]:  $w_1 =_F w_2 \implies w_1 @- w \preceq_I w_2 @- w$ 
proof
  fix  $v_1$ 
  assume 1:  $w_1 =_F w_2$   $v_1 \leq_{FI} w_1 @- w$ 
  show  $v_1 \preceq_{FI} w_2 @- w$ 
  using 1(2)
  proof (cases rule: prefix-fininf-append)
    case (absorb)
    show ?thesis
  proof
    show  $w_2 \leq_{FI} (w_2 @- w)$  by auto
    show  $v_1 \preceq_F w_2$  using absorb 1(1) by auto
  qed
  next
  case (extend w')
  show ?thesis
  proof
    show  $w_2 @ w' \leq_{FI} (w_2 @- w)$  using extend(2) by auto
    show  $v_1 \preceq_F w_2 @ w'$  unfolding extend(1) using 1(1) by auto
  qed
qed
qed
lemma eq-inf-concat-start[iff]:  $w @- w_1 =_I w @- w_2 \iff w_1 =_I w_2$  by blast
lemma eq-inf-concat-end[dest]:  $w_1 =_F w_2 \implies w_1 @- w =_I w_2 @- w$ 
proof -
  assume 0:  $w_1 =_F w_2$ 
  have 1:  $w_2 =_F w_1$  using 0 by auto
  show  $w_1 @- w =_I w_2 @- w$ 
  using eq-fin-le-inf-concat-end[OF 0] eq-fin-le-inf-concat-end[OF 1] by auto
qed

```

```

lemma le-fininf-suffixI[intro]:
  assumes  $w =_I w_1 @- w_2$ 
  shows  $w_1 \preceq_{FI} w$ 
  using assms by blast
lemma le-fininf-suffixE[elim]:
  assumes  $w_1 \preceq_{FI} w$ 
  obtains  $w_2$ 
  where  $w =_I w_1 @- w_2$ 
proof -
  obtain  $v_2$  where  $1: v_2 \leq_{FI} w$   $w_1 \preceq_F v_2$  using assms(1) by rule
  obtain  $u_1$  where  $2: w_1 @ u_1 =_F v_2$  using  $1(2)$  by rule
  obtain  $v_2'$  where  $3: w = v_2 @- v_2'$  using  $1(1)$  by rule
  show ?thesis
proof
  show  $w =_I w_1 @- u_1 @- v_2'$  unfolding  $3$  using  $2$  by fastforce
qed
qed

lemma subsume-fin:
  assumes  $u_1 \preceq_{FI} w$   $v_1 \preceq_{FI} w$ 
  obtains  $w_1$ 
  where  $u_1 \preceq_F w_1$   $v_1 \preceq_F w_1$ 
proof -
  obtain  $u_2$  where  $2: u_2 \leq_{FI} w$   $u_1 \preceq_F u_2$  using assms(1) by rule
  obtain  $v_2$  where  $3: v_2 \leq_{FI} w$   $v_1 \preceq_F v_2$  using assms(2) by rule
  show ?thesis
proof (cases length u_2 length v_2 rule: le-cases)
  case le
  show ?thesis
proof
  show  $u_1 \preceq_F v_2$  using  $2(2)$  prefix-fininf-length[OF 2(1) 3(1) le] by auto
  show  $v_1 \preceq_F v_2$  using  $3(2)$  by this
qed
next
  case ge
  show ?thesis
proof
  show  $u_1 \preceq_F u_2$  using  $2(2)$  by this
  show  $v_1 \preceq_F u_2$  using  $3(2)$  prefix-fininf-length[OF 3(1) 2(1) ge] by auto
qed
qed
qed

lemma eq-fin-end:
  assumes  $u_1 =_F u_2$   $u_1 @ v_1 =_F u_2 @ v_2$ 
  shows  $v_1 =_F v_2$ 
proof -
  have  $u_1 @ v_2 =_F u_2 @ v_2$  using assms(1) by blast

```

also have $\dots =_F u_1 @ v_1$ **using** *assms(2)* **by** *blast*
finally show *?thesis* **by** *blast*
qed

definition *indoc* :: 'item \Rightarrow 'item list \Rightarrow bool
where *indoc a u* $\equiv \exists u_1 u_2. u = u_1 @ [a] @ u_2 \wedge a \notin \text{set } u_1 \wedge \text{Ind } \{a\} (\text{set } u_1)$

lemma *indoc-set*: *indoc a u* $\implies a \in \text{set } u$ **unfolding** *indoc-def* **by** *auto*

lemma *indoc-appendI1*[*intro*]:
assumes *indoc a u*
shows *indoc a (u @ v)*
using *assms* **unfolding** *indoc-def* **by** *force*

lemma *indoc-appendI2*[*intro*]:
assumes $a \notin \text{set } u$ *Ind {a} (set u)* *indoc a v*
shows *indoc a (u @ v)*

proof –
obtain $v_1 v_2$ **where** $1: v = v_1 @ [a] @ v_2$ $a \notin \text{set } v_1$ *Ind {a} (set v_1)*
using *assms(3)* **unfolding** *indoc-def* **by** *blast*
show *?thesis*
proof (*unfold indoc-def, intro exI conjI*)
show $u @ v = (u @ v_1) @ [a] @ v_2$ **unfolding** $1(1)$ **by** *simp*
show $a \notin \text{set } (u @ v_1)$ **using** *assms(1)* $1(2)$ **by** *auto*
show *Ind {a} (set (u @ v_1))* **using** *assms(2)* $1(3)$ **by** *auto*
qed

qed

lemma *indoc-appendE*[*elim!*]:
assumes *indoc a (u @ v)*
obtains (*first*) $a \in \text{set } u$ *indoc a u* | (*second*) $a \notin \text{set } u$ *Ind {a} (set u)* *indoc a v*

proof –
obtain $w_1 w_2$ **where** $1: u @ v = w_1 @ [a] @ w_2$ $a \notin \text{set } w_1$ *Ind {a} (set w_1)*
using *assms* **unfolding** *indoc-def* **by** *blast*
show *?thesis*
proof (*cases a ∈ set u*)
case *True*
obtain $u_1 u_2$ **where** $2: u = u_1 @ [a] @ u_2$ $a \notin \text{set } u_1$
using *split-list-first[OF True]* **by** *auto*
have $3: w_1 = u_1$
proof (*rule split-list-first-unique*)
show $w_1 @ [a] @ w_2 = u_1 @ [a] @ u_2 @ v$ **using** $1(1)$ **unfolding** $2(1)$
by *simp*
show $a \notin \text{set } w_1$ **using** $1(2)$ **by** *auto*
show $a \notin \text{set } u_1$ **using** $2(2)$ **by** *this*
qed
show *?thesis*
proof (*rule first*)
show $a \in \text{set } u$ **using** *True* **by** *this*

```

    show indoc a u
    proof (unfold indoc-def, intro exI conjI)
      show  $u = u_1 @ [a] @ u_2$  using 2(1) by this
      show  $a \notin \text{set } u_1$  using 1(2) unfolding 3 by this
      show  $\text{Ind } \{a\} (\text{set } u_1)$  using 1(3) unfolding 3 by this
    qed
  qed
next
case False
have 2:  $a \in \text{set } v$  using indoc-set assms False by fastforce
obtain  $v_1 v_2$  where 3:  $v = v_1 @ [a] @ v_2$   $a \notin \text{set } v_1$ 
  using split-list-first[OF 2] by auto
have 4:  $w_1 = u @ v_1$ 
proof (rule split-list-first-unique)
  show  $w_1 @ [a] @ w_2 = (u @ v_1) @ [a] @ v_2$  using 1(1) unfolding 3(1)
by simp
  show  $a \notin \text{set } w_1$  using 1(2) by auto
  show  $a \notin \text{set } (u @ v_1)$  using False 3(2) by auto
qed
show ?thesis
proof (rule second)
  show  $a \notin \text{set } u$  using False by this
  show  $\text{Ind } \{a\} (\text{set } u)$  using 1(3) 4 by auto
  show indoc a v
  proof (unfold indoc-def, intro exI conjI)
    show  $v = v_1 @ [a] @ v_2$  using 3(1) by this
    show  $a \notin \text{set } v_1$  using 1(2) unfolding 4 by auto
    show  $\text{Ind } \{a\} (\text{set } v_1)$  using 1(3) unfolding 4 by auto
  qed
qed
qed
qed
qed
lemma indoc-single:  $\text{indoc } a [b] \longleftrightarrow a = b$ 
proof
  assume 1: indoc a [b]
  obtain  $u_1 u_2$  where 2:  $[b] = u_1 @ [a] @ u_2$   $\text{Ind } \{a\} (\text{set } u_1)$ 
    using 1 unfolding indoc-def by auto
  show  $a = b$  using 2(1)
  by (metis append-eq-Cons-conv append-is-Nil-conv list.distinct(2) list.inject)
next
  assume 1:  $a = b$ 
  show indoc a [b]
  unfolding indoc-def 1
  proof (intro exI conjI)
    show  $[b] = [] @ [b] @ []$  by simp
    show  $b \notin \text{set } []$  by simp
    show  $\text{Ind } \{b\} (\text{set } [])$  by simp
  qed

```

qed

lemma *indoc-append[simp]*: $\text{indoc } a (u @ v) \longleftrightarrow$

$\text{indoc } a u \vee a \notin \text{set } u \wedge \text{Ind } \{a\} (\text{set } u) \wedge \text{indoc } a v$ **by** *blast*

lemma *indoc-Nil[simp]*: $\text{indoc } a [] \longleftrightarrow \text{False}$ **unfolding** *indoc-def* **by** *auto*

lemma *indoc-Cons[simp]*: $\text{indoc } a (b \# v) \longleftrightarrow a = b \vee a \neq b \wedge \text{ind } a b \wedge$
indoc } a v

proof –

have $\text{indoc } a (b \# v) \longleftrightarrow \text{indoc } a ([b] @ v)$ **by** *simp*

also have $\dots \longleftrightarrow \text{indoc } a [b] \vee a \notin \text{set } [b] \wedge \text{Ind } \{a\} (\text{set } [b]) \wedge \text{indoc } a v$
unfolding *indoc-append* **by** *rule*

also have $\dots \longleftrightarrow a = b \vee a \neq b \wedge \text{ind } a b \wedge \text{indoc } a v$ **unfolding** *indoc-single*
by *simp*

finally show *?thesis* **by** *this*

qed

lemma *eq-swap-indoc*: $u =_S v \implies \text{indoc } c u \implies \text{indoc } c v$ **by** *auto*

lemma *eq-fin-indoc*: $u =_F v \implies \text{indoc } c u \implies \text{indoc } c v$ **by** (*induct rule*:
rtranclp.induct, auto)

lemma *eq-fin-ind'*:

assumes $[a] @ u =_F u_1 @ [a] @ u_2$ $a \notin \text{set } u_1$

shows $\text{Ind } \{a\} (\text{set } u_1)$

proof –

have $1: \text{indoc } a ([a] @ u)$ **by** *simp*

have $2: \text{indoc } a (u_1 @ [a] @ u_2)$ **using** *eq-fin-indoc assms(1) 1* **by** *this*

show *?thesis* **using** *assms(2) 2* **by** *blast*

qed

lemma *eq-fin-ind*:

assumes $u @ v =_F v @ u$ $\text{set } u \cap \text{set } v = \{\}$

shows $\text{Ind } (\text{set } u) (\text{set } v)$

using *assms*

proof (*induct u*)

case *Nil*

show *?case* **by** *simp*

next

case (*Cons a u*)

have $1: \text{Ind } \{a\} (\text{set } v)$

proof (*rule eq-fin-ind'*)

show $[a] @ u @ v =_F v @ [a] @ u$ **using** *Cons(2)* **by** *simp*

show $a \notin \text{set } v$ **using** *Cons(3)* **by** *simp*

qed

have $2: \text{Ind } (\text{set } [a]) (\text{set } v)$ **using** 1 **by** *simp*

have $4: \text{Ind } (\text{set } u) (\text{set } v)$

proof (*rule Cons(1)*)

have $[a] @ u @ v = (a \# u) @ v$ **by** *simp*

also have $\dots =_F v @ a \# u$ **using** *Cons(2)* **by** *this*

also have $\dots = (v @ [a]) @ u$ **by** *simp*

also have $\dots =_F ([a] @ v) @ u$ **using** 2 **by** *blast*

also have $\dots = [a] @ v @ u$ by *simp*
 finally show $u @ v =_F v @ u$ by *blast*
 show $set\ u \cap set\ v = \{\}$ using *Cons(3)* by *auto*
 qed
 show *?case* using 1 4 by *auto*
 qed

lemma *le-fin-member'*:

assumes $[a] \preceq_F u @ v$ $a \in set\ u$

shows $[a] \preceq_F u$

proof –

obtain w where 1: $[a] @ w =_F u @ v$ using *assms(1)* by *rule*

obtain $u_1\ u_2$ where 2: $u = u_1 @ [a] @ u_2$ $a \notin set\ u_1$

using *split-list-first[OF assms(2)]* by *auto*

have 3: *Ind* $\{a\}$ (*set* u_1)

proof (*rule eq-fin-ind'*)

show $[a] @ w =_F u_1 @ [a] @ u_2 @ v$ using 1 unfolding 2(1) by *simp*

show $a \notin set\ u_1$ using 2(2) by *this*

qed

have 4: *Ind* (*set* $[a]$) (*set* u_1) using 3 by *simp*

have $[a] \leq [a] @ u_1 @ u_2$ by *auto*

also have $\dots = ([a] @ u_1) @ u_2$ by *simp*

also have $\dots =_F (u_1 @ [a]) @ u_2$ using 4 by *blast*

also have $\dots = u$ unfolding 2(1) by *simp*

finally show *?thesis* by *this*

qed

lemma *le-fin-not-member'*:

assumes $[a] \preceq_F u @ v$ $a \notin set\ u$

shows $[a] \preceq_F v$

proof –

obtain w where 1: $[a] @ w =_F u @ v$ using *assms(1)* by *rule*

have 3: $a \in set\ v$ using *assms* by *auto*

obtain $v_1\ v_2$ where 4: $v = v_1 @ [a] @ v_2$ $a \notin set\ v_1$ using *split-list-first[OF 3]* by *auto*

have 5: $[a] @ w =_F u @ v_1 @ [a] @ v_2$ using 1 unfolding 4(1) by *this*

have 6: *Ind* $\{a\}$ (*set* ($u @ v_1$))

proof (*rule eq-fin-ind'*)

show $[a] @ w =_F (u @ v_1) @ [a] @ v_2$ using 5 by *simp*

show $a \notin set\ (u @ v_1)$ using *assms(2)* 4(2) by *auto*

qed

have 9: *Ind* (*set* $[a]$) (*set* v_1) using 6 by *auto*

have $[a] \leq [a] @ v_1 @ v_2$ by *auto*

also have $\dots = ([a] @ v_1) @ v_2$ by *simp*

also have $\dots =_F (v_1 @ [a]) @ v_2$ using 9 by *blast*

also have $\dots = v_1 @ [a] @ v_2$ by *simp*

also have $\dots = v$ unfolding 4(1) by *rule*

finally show *?thesis* by *this*

qed

lemma *le-fininf-not-member'*:

assumes $[a] \preceq_{FI} u @- v a \notin \text{set } u$
shows $[a] \preceq_{FI} v$
proof –
obtain v_2 **where** $1: v_2 \leq_{FI} u @- v [a] \preceq_F v_2$ **using** *le-fininfE* *assms(1)* **by**
this
show *?thesis*
using $1(1)$
proof (*cases rule: prefix-fininf-append*)
case *absorb*
have $[a] \preceq_F v_2$ **using** $1(2)$ **by** *this*
also have $\dots \leq u$ **using** *absorb* **by** *this*
finally have $2: a \in \text{set } u$ **by** *force*
show *?thesis* **using** *assms(2)* 2 **by** *simp*
next
case (*extend z*)
have $[a] \preceq_F v_2$ **using** $1(2)$ **by** *this*
also have $\dots = u @ z$ **using** *extend(1)* **by** *this*
finally have $2: [a] \preceq_F u @ z$ **by** *this*
have $[a] \preceq_F z$ **using** *le-fin-not-member'* 2 *assms(2)* **by** *this*
also have $\dots \leq_{FI} v$ **using** *extend(2)* **by** *this*
finally show *?thesis* **by** *this*
qed
qed

lemma *le-fin-ind''*:
assumes $[a] \preceq_F w [b] \preceq_F w a \neq b$
shows *ind a b*
proof –
obtain u **where** $1: [a] @ u =_F w$ **using** *assms(1)* **by** *rule*
obtain v **where** $2: [b] @ v =_F w$ **using** *assms(2)* **by** *rule*
have $3: [a] @ u =_F [b] @ v$ **using** 1 2 [*symmetric*] **by** *auto*
have $4: a \in \text{set } v$ **using** 3 *assms(3)*
by (*metis append-Cons append-Nil eq-fin-range list.set-intros(1) set-ConsD*)
obtain $v_1 v_2$ **where** $5: v = v_1 @ [a] @ v_2 a \notin \text{set } v_1$ **using** *split-list-first[OF*
 $4]$ **by** *auto*
have $7: \text{Ind } \{a\} (\text{set } ([b] @ v_1))$
proof (*rule eq-fin-ind'*)
show $[a] @ u =_F ([b] @ v_1) @ [a] @ v_2$ **using** 3 *unfolding* $5(1)$ **by** *simp*
show $a \notin \text{set } ([b] @ v_1)$ **using** *assms(3)* $5(2)$ **by** *auto*
qed
show *?thesis* **using** 7 **by** *auto*
qed

lemma *le-fin-ind'*:
assumes $[a] \preceq_F w v \preceq_F w a \notin \text{set } v$
shows *Ind {a} (set v)*
using *assms*
proof (*induct v arbitrary: w*)
case *Nil*
show *?case* **by** *simp*

```

next
  case (Cons b v)
  have 1: ind a b
  proof (rule le-fin-ind'')
    show [a]  $\preceq_F$  w using Cons(2) by this
    show [b]  $\preceq_F$  w using Cons(3) by auto
    show a  $\neq$  b using Cons(4) by auto
  qed
  obtain w' where 2: [b] @ w' =F w using Cons(3) by auto
  have 3: Ind {a} (set v)
  proof (rule Cons(1))
    show [a]  $\preceq_F$  w'
    proof (rule le-fin-not-member')
      show [a]  $\preceq_F$  [b] @ w' using Cons(2) 2 by auto
      show a  $\notin$  set [b] using Cons(4) by auto
    qed
    have [b] @ v = b # v by simp
    also have ...  $\preceq_F$  w using Cons(3) by this
    also have ... =F [b] @ w' using 2 by auto
    finally show v  $\preceq_F$  w' by blast
    show a  $\notin$  set v using Cons(4) by auto
  qed
  show ?case using 1 3 by auto
qed
lemma le-fininf-ind'':
  assumes [a]  $\preceq_{FI}$  w [b]  $\preceq_{FI}$  w a  $\neq$  b
  shows ind a b
  using subsume-fin le-fin-ind'' assms by metis
lemma le-fininf-ind':
  assumes [a]  $\preceq_{FI}$  w v v  $\preceq_{FI}$  w a  $\notin$  set v
  shows Ind {a} (set v)
  using subsume-fin le-fin-ind' assms by metis

lemma indoc-alt-def: indoc a v  $\longleftrightarrow$  v =F [a] @ remove1 a v
proof
  assume 0: indoc a v
  obtain v1 v2 where 1: v = v1 @ [a] @ v2 a  $\notin$  set v1 Ind {a} (set v1)
  using 0 unfolding indoc-def by blast
  have 2: Ind (set [a]) (set v1) using 1(3) by simp
  have v = v1 @ [a] @ v2 using 1(1) by this
  also have ... = (v1 @ [a]) @ v2 by simp
  also have ... =F ([a] @ v1) @ v2 using 2 by blast
  also have ... = [a] @ v1 @ v2 by simp
  also have ... = [a] @ remove1 a v unfolding 1(1) remove1-append using
1(2) by auto
  finally show v =F [a] @ remove1 a v by this
next
  assume 0: v =F [a] @ remove1 a v
  have 1: indoc a ([a] @ remove1 a v) by simp

```


show *indoc a v* using *eq-fin-indoc 0 1* by *blast*
qed

lemma *levi-lemma*:

assumes $t @ u =_F v @ w$

obtains $p r s q$

where $t =_F p @ r$ $u =_F s @ q$ $v =_F p @ s$ $w =_F r @ q$ *Ind (set r) (set s)*

using *assms*

proof (*induct t arbitrary: thesis v w*)

case *Nil*

show *?case*

proof (*rule Nil(1)*)

show $[] =_F [] @ []$ by *simp*

show $v =_F [] @ v$ by *simp*

show $u =_F v @ w$ using *Nil(2)* by *simp*

show $w =_F [] @ w$ by *simp*

show *Ind (set []) (set v)* by *simp*

qed

next

case (*Cons a t'*)

have 1: $[a] \preceq_F v @ w$ using *Cons(3)* by *blast*

show *?case*

proof (*cases a ∈ set v*)

case *False*

have 2: $[a] \preceq_F w$ using *le-fin-not-member' 1 False* by *this*

obtain w' where 3: $w =_F [a] @ w'$ using 2 by *blast*

have 4: $v \preceq_F v @ w$ by *auto*

have 5: *Ind (set [a]) (set v)* using *le-fin-ind'[OF 1 4] False* by *simp*

have $[a] @ t' @ u = (a \# t') @ u$ by *simp*

also have $\dots =_F v @ w$ using *Cons(3)* by *this*

also have $\dots =_F v @ [a] @ w'$ using 3 by *blast*

also have $\dots = (v @ [a]) @ w'$ by *simp*

also have $\dots =_F ([a] @ v) @ w'$ using 5 by *blast*

also have $\dots = [a] @ v @ w'$ by *simp*

finally have 6: $t' @ u =_F v @ w'$ by *blast*

obtain $p r' s q$ where 7: $t' =_F p @ r'$ $u =_F s @ q$ $v =_F p @ s$ $w' =_F r'$

@ q

Ind (set r') (set s) using *Cons(1)[OF - 6]* by *this*

have 8: $set v = set p \cup set s$ using *eq-fin-range 7(3)* by *auto*

have 9: *Ind (set [a]) (set p)* using 5 8 by *auto*

have 10: *Ind (set [a]) (set s)* using 5 8 by *auto*

show *?thesis*

proof (*rule Cons(2)*)

have $a \# t' = [a] @ t'$ by *simp*

also have $\dots =_F [a] @ p @ r'$ using 7(1) by *blast*

also have $\dots = ([a] @ p) @ r'$ by *simp*

also have $\dots =_F (p @ [a]) @ r'$ using 9 by *blast*

also have $\dots = p @ [a] @ r'$ by *simp*

finally show $a \# t' =_F p @ [a] @ r'$ by *this*

```

    show  $u =_F s @ q$  using  $\gamma(2)$  by this
    show  $v =_F p @ s$  using  $\gamma(3)$  by this
    have  $w =_F [a] @ w'$  using  $\beta$  by this
    also have  $\dots =_F [a] @ r' @ q$  using  $\gamma(4)$  by blast
    also have  $\dots = ([a] @ r') @ q$  by simp
    finally show  $w =_F ([a] @ r') @ q$  by this
    show  $Ind (set ([a] @ r')) (set s)$  using  $\gamma(5)$  10 by auto
qed
next
case True
have  $2: [a] \preceq_F v$  using le-fin-member' 1 True by this
obtain  $v'$  where  $3: v =_F [a] @ v'$  using 2 by blast
have  $[a] @ t' @ u = (a \# t') @ u$  by simp
also have  $\dots =_F v @ w$  using Cons(3) by this
also have  $\dots =_F ([a] @ v') @ w$  using 3 by blast
also have  $\dots = [a] @ v' @ w$  by simp
finally have  $4: t' @ u =_F v' @ w$  by blast
obtain  $p' r s q$  where  $7: t' =_F p' @ r$   $u =_F s @ q$   $v' =_F p' @ s$   $w =_F r$ 
@ q
     $Ind (set r) (set s)$  using Cons(1)[OF - 4] by this
show ?thesis
proof (rule Cons(2))
    have  $a \# t' = [a] @ t'$  by simp
    also have  $\dots =_F [a] @ p' @ r$  using  $\gamma(1)$  by blast
    also have  $\dots = ([a] @ p') @ r$  by simp
    finally show  $a \# t' =_F ([a] @ p') @ r$  by this
    show  $u =_F s @ q$  using  $\gamma(2)$  by this
    have  $v =_F [a] @ v'$  using  $\beta$  by this
    also have  $\dots =_F [a] @ p' @ s$  using  $\gamma(3)$  by blast
    also have  $\dots = ([a] @ p') @ s$  by simp
    finally show  $v =_F ([a] @ p') @ s$  by this
    show  $w =_F r @ q$  using  $\gamma(4)$  by this
    show  $Ind (set r) (set s)$  using  $\gamma(5)$  by this
qed
qed
qed

end

end

```

9 Transition Systems and Trace Theory

```

theory Transition-System-Traces
imports
  Transition-System-Extensions
  Traces
begin

```

lemma (in *transition-system*) *words-infI-construct*[*rule-format, intro?*]:
assumes $\forall v. v \leq_{FI} w \longrightarrow \text{path } v \text{ } p$
shows *run w p*
using *assms* **by** *coinduct auto*

lemma (in *transition-system*) *words-infI-construct'*:
assumes $\bigwedge k. \exists v. v \leq_{FI} w \wedge k < \text{length } v \wedge \text{path } v \text{ } p$
shows *run w p*
proof
fix *u*
assume *1: u ≤_{FI} w*
obtain *v* **where** *2: v ≤_{FI} w length u < length v path v p* **using** *assms(1)* **by**
auto
have *3: length u ≤ length v* **using** *2(2)* **by** *simp*
have *4: u ≤ v* **using** *prefix-fininf-length 1 2(1) 3* **by** *this*
show *path u p* **using** *4 2(3)* **by** *auto*
qed

lemma (in *transition-system*) *words-infI-construct-chain*[*intro*]:
assumes *chain w* $\bigwedge k. \text{path } (w \ k) \text{ } p$
shows *run (limit w) p*
proof (*rule words-infI-construct'*)
fix *k*
obtain *l* **where** *1: k < length (w l)* **using** *assms(1)* **by** *rule*
show $\exists v. v \leq_{FI} \text{limit } w \wedge k < \text{length } v \wedge \text{path } v \text{ } p$
proof (*intro exI conjI*)
show $w \ l \leq_{FI} \text{limit } w$ **using** *chain-prefix-limit assms(1)* **by** *this*
show $k < \text{length } (w \ l)$ **using** *1* **by** *this*
show *path (w l) p* **using** *assms(2)* **by** *this*
qed
qed

lemma (in *transition-system*) *words-fin-blocked*:
assumes $\bigwedge w. \text{path } w \text{ } p \implies A \cap \text{set } w = \{\} \implies A \cap \{a. \text{enabled } a \text{ (target } w \text{ } p)\} \subseteq A \cap \{a. \text{enabled } a \text{ } p\}$
assumes *path w p* $A \cap \{a. \text{enabled } a \text{ } p\} \cap \text{set } w = \{\}$
shows $A \cap \text{set } w = \{\}$
using *assms* **by** (*induct w rule: rev-induct, auto*)

locale *transition-system-traces* =
transition-system ex en +
traces ind
for *ex* :: *'action* \Rightarrow *'state* \Rightarrow *'state*
and *en* :: *'action* \Rightarrow *'state* \Rightarrow *bool*
and *ind* :: *'action* \Rightarrow *'action* \Rightarrow *bool*
+
assumes *en: ind a b* \implies *en a p* \implies *en b p* \iff *en b (ex a p)*
assumes *ex: ind a b* \implies *en a p* \implies *en b p* \implies *ex b (ex a p) = ex a (ex b p)*
begin

lemma *diamond-bottom*:
assumes *ind a b*
assumes *en a p en b p*
shows *en a (ex b p) en b (ex a p) ex b (ex a p) = ex a (ex b p)*
using *assms independence-symmetric en ex* **by** *metis+*

lemma *diamond-right*:
assumes *ind a b*
assumes *en a p en b (ex a p)*
shows *en a (ex b p) en b p ex b (ex a p) = ex a (ex b p)*
using *assms independence-symmetric en ex* **by** *metis+*

lemma *diamond-left*:
assumes *ind a b*
assumes *en a (ex b p) en b p*
shows *en a p en b (ex a p) ex b (ex a p) = ex a (ex b p)*
using *assms independence-symmetric en ex* **by** *metis+*

lemma *eq-swap-word*:
assumes *w₁ =_S w₂ path w₁ p*
shows *path w₂ p*
using *assms diamond-right* **by** *(induct, auto)*

lemma *eq-fin-word*:
assumes *w₁ =_F w₂ path w₁ p*
shows *path w₂ p*
using *assms eq-swap-word* **by** *(induct, auto)*

lemma *le-fin-word*:
assumes *w₁ ≼_F w₂ path w₂ p*
shows *path w₁ p*
using *assms eq-fin-word* **by** *blast*

lemma *le-fininf-word*:
assumes *w₁ ≼_{FI} w₂ run w₂ p*
shows *path w₁ p*
using *assms le-fin-word* **by** *blast*

lemma *le-inf-word*:
assumes *w₂ ≼_I w₁ run w₁ p*
shows *run w₂ p*
using *assms le-fininf-word* **by** *(blast intro: words-infI-construct)*

lemma *eq-inf-word*:
assumes *w₁ =_I w₂ run w₁ p*
shows *run w₂ p*
using *assms le-inf-word* **by** *auto*

lemma *eq-swap-execute*:
assumes *path w₁ p w₁ =_S w₂*
shows *fold ex w₁ p = fold ex w₂ p*
using *assms(2, 1) diamond-right* **by** *(induct, auto)*

lemma *eq-fin-execute*:
assumes *path w₁ p w₁ =_F w₂*
shows *fold ex w₁ p = fold ex w₂ p*

using *assms*(2, 1) *eq-fin-word eq-swap-execute* **by** (*induct*, *auto*)

lemma *diamond-fin-word-step*:

assumes *Ind* {*a*} (*set v*) *en a p path v p*

shows *path v (ex a p)*

using *diamond-bottom assms* **by** (*induct v arbitrary: p, auto, metis*)

lemma *diamond-inf-word-step*:

assumes *Ind* {*a*} (*sset w*) *en a p run w p*

shows *run w (ex a p)*

using *diamond-fin-word-step assms* **by** (*fast intro: words-infI-construct*)

lemma *diamond-fin-word-inf-word*:

assumes *Ind* (*set v*) (*sset w*) *path v p run w p*

shows *run w (fold ex v p)*

using *diamond-inf-word-step assms* **by** (*induct v arbitrary: p, auto*)

lemma *diamond-fin-word-inf-word'*:

assumes *Ind* (*set v*) (*sset w*) *path (u @ v) p run (u @- w) p*

shows *run (u @- v @- w) p*

using *assms diamond-fin-word-inf-word* **by** *auto*

end

end

10 Functions

theory *Functions*

imports *../Extensions/Set-Extensions*

begin

locale *bounded-function* =

fixes *A* :: 'a *set*

fixes *B* :: 'b *set*

fixes *f* :: 'a \Rightarrow 'b

assumes *wellformed*[*intro?*, *simp*]: $x \in A \Longrightarrow f x \in B$

locale *bounded-function-pair* =

f: *bounded-function* *A B f* +

g: *bounded-function* *B A g*

for *A* :: 'a *set*

and *B* :: 'b *set*

and *f* :: 'a \Rightarrow 'b

and *g* :: 'b \Rightarrow 'a

locale *injection* = *bounded-function-pair* +

assumes *left-inverse*[*simp*]: $x \in A \Longrightarrow g (f x) = x$

begin

lemma *inj-on*[*intro*]: *inj-on f A* **using** *inj-onI left-inverse* **by** *metis*

```

lemma injective-on:
  assumes  $x \in A \ y \in A \ f \ x = f \ y$ 
  shows  $x = y$ 
  using assms left-inverse by metis

end

locale injective = bounded-function +
  assumes injection:  $\exists g. \text{injection } A \ B \ f \ g$ 
begin

  definition  $g \equiv \text{SOME } g. \text{injection } A \ B \ f \ g$ 

  sublocale injection  $A \ B \ f \ g$  unfolding g-def using someI-ex[OF injection] by
  this

end

locale surjection = bounded-function-pair +
  assumes right-inverse[simp]:  $y \in B \implies f (g \ y) = y$ 
begin

  lemma image-superset[intro]:  $f \ ' \ A \supseteq B$ 
  using g.wellformed image-iff right-inverse subsetI by metis

  lemma image-eq[simp]:  $f \ ' \ A = B$  using image-superset by auto

end

locale surjective = bounded-function +
  assumes surjection:  $\exists g. \text{surjection } A \ B \ f \ g$ 
begin

  definition  $g \equiv \text{SOME } g. \text{surjection } A \ B \ f \ g$ 

  sublocale surjection  $A \ B \ f \ g$  unfolding g-def using someI-ex[OF surjection]
  by this

end

locale bijection = injection + surjection

lemma inj-on-bijection:
  assumes inj-on  $f \ A$ 
  shows bijection  $A \ (f \ ' \ A) \ f \ (\text{inv-into } A \ f)$ 
proof
  show  $\bigwedge x. x \in A \implies f \ x \in f \ ' \ A$  using imageI by this
  show  $\bigwedge y. y \in f \ ' \ A \implies \text{inv-into } A \ f \ y \in A$  using inv-into-into by this
  show  $\bigwedge x. x \in A \implies \text{inv-into } A \ f \ (f \ x) = x$  using inv-into-f-f assms by this

```

show $\bigwedge y. y \in f^{-1} A \implies f(\text{inv-into } A f y) = y$ **using** *f-inv-into-f* **by this**
qed

end

11 Extended Natural Numbers

theory *ENat-Extensions*

imports

Coinductive.Coinductive-Nat

begin

declare *eSuc-enat[simp]*

declare *iadd-Suc[simp] iadd-Suc-right[simp]*

declare *enat-0[simp] enat-1[simp] one-eSuc[simp]*

declare *enat-0-iff[iff] enat-1-iff[iff]*

declare *Suc-ile-eq[iff]*

lemma *enat-Suc0[simp]*: *enat (Suc 0) = eSuc 0* **by** (*metis One-nat-def one-eSuc one-enat-def*)

lemma *le-epred[iff]*: $l < \text{epred } k \iff \text{eSuc } l < k$

by (*metis eSuc-le-iff epred-eSuc epred-le-epredI less-le-not-le not-le*)

lemma *eq-infI[intro]*:

assumes $\bigwedge n. \text{enat } n \leq m$

shows $m = \infty$

using *assms* **by** (*metis enat-less-imp-le enat-ord-simps(5) less-le-not-le*)

end

12 Chain-Complete Partial Orders

theory *CCPO-Extensions*

imports

HOL-Library.Complete-Partial-Order2

ENat-Extensions

Set-Extensions

begin

lemma *chain-split[dest]*:

assumes *Complete-Partial-Order.chain ord C x ∈ C*

shows $C = \{y \in C. \text{ord } x y\} \cup \{y \in C. \text{ord } y x\}$

proof –

have *1*: $\bigwedge y. y \in C \implies \text{ord } x y \vee \text{ord } y x$ **using** *chainD assms* **by this**

show *?thesis* **using** *1* **by blast**

qed

lemma *infinite-chain-below*[*dest*]:
assumes *Complete-Partial-Order.chain ord C infinite C x ∈ C*
assumes *finite {y ∈ C. ord x y}*
shows *infinite {y ∈ C. ord y x}*
proof –
have $1: C = \{y \in C. \text{ord } x \ y\} \cup \{y \in C. \text{ord } y \ x\}$ **using** *assms(1, 3)* **by rule**
show *?thesis* **using** *finite-Un assms(2, 4) 1* **by** (*metis (poly-guards-query)*)
qed

lemma *infinite-chain-above*[*dest*]:
assumes *Complete-Partial-Order.chain ord C infinite C x ∈ C*
assumes *finite {y ∈ C. ord y x}*
shows *infinite {y ∈ C. ord x y}*
proof –
have $1: C = \{y \in C. \text{ord } x \ y\} \cup \{y \in C. \text{ord } y \ x\}$ **using** *assms(1, 3)* **by rule**
show *?thesis* **using** *finite-Un assms(2, 4) 1* **by** (*metis (poly-guards-query)*)
qed

lemma (in *ccpo*) *ccpo-Sup-upper-inv*:
assumes *Complete-Partial-Order.chain less-eq C x > ⋓ C*
shows $x \notin C$
using *assms ccpo-Sup-upper* **by** *fastforce*

lemma (in *ccpo*) *ccpo-Sup-least-inv*:
assumes *Complete-Partial-Order.chain less-eq C ⋓ C > x*
obtains y
where $y \in C \wedge y \leq x$
using *assms ccpo-Sup-least that* **by** *fastforce*

lemma *ccpo-Sup-least-inv'*:
fixes $C :: 'a :: \{\text{ccpo}, \text{linorder}\} \text{ set}$
assumes *Complete-Partial-Order.chain less-eq C ⋓ C > x*
obtains y
where $y \in C \wedge y > x$
proof –
obtain y **where** $1: y \in C \wedge y \leq x$ **using** *ccpo-Sup-least-inv assms* **by** *this*
show *?thesis* **using** *that 1* **by** *simp*
qed

lemma *mcont2mcont-lessThan*[*THEN lfp.mcont2mcont, simp, cont-intro*]:
shows *mcont-lessThan: mcont Sup less-eq Sup less-eq*
(lessThan :: 'a :: {\ccpo, linorder} ⇒ 'a set)
proof
show *monotone less-eq less-eq (lessThan :: 'a ⇒ 'a set)* **by** (*rule, auto*)
show *cont Sup less-eq Sup less-eq (lessThan :: 'a ⇒ 'a set)*
proof
fix $C :: 'a \text{ set}$
assume $1: \text{Complete-Partial-Order.chain less-eq } C$
show $\{.. < \bigsqcup C\} = \bigcup (\text{lessThan } ' C)$
proof (*intro equalityI subsetI*)
fix A


```

    assume 2:  $A \in \{..< \sqcup C\}$ 
    obtain B where 3:  $B \in C B > A$  using ccpo-Sup-least-inv' 1 2 by blast
    show  $A \in \cup (lessThan ' C)$  using 3 by auto
  next
    fix A
    assume 2:  $A \in \cup (lessThan ' C)$ 
    show  $A \in \{..< \sqcup C\}$  using ccpo-Sup-upper 2 by force
  qed
qed
qed

class esize =
  fixes esize :: 'a  $\Rightarrow$  enat

class esize-order = esize + order +
  assumes esize-finite[dest]:  $esize\ x \neq \infty \implies finite\ \{y.\ y \leq x\}$ 
  assumes esize-mono[intro]:  $x \leq y \implies esize\ x \leq esize\ y$ 
  assumes esize-strict-mono[intro]:  $esize\ x \neq \infty \implies x < y \implies esize\ x < esize\ y$ 
begin

lemma infinite-chain-eSuc-esize[dest]:
  assumes Complete-Partial-Order.chain less-eq C infinite C  $x \in C$ 
  obtains y
  where  $y \in C esize\ y \geq eSuc\ (esize\ x)$ 
proof (cases esize x)
  case (enat k)
  have 1:  $finite\ \{y \in C.\ y \leq x\}$  using esize-finite enat by simp
  have 2:  $infinite\ \{y \in C.\ y \geq x\}$  using assms 1 by rule
  have 3:  $\{y \in C.\ y > x\} = \{y \in C.\ y \geq x\} - \{x\}$  by auto
  have 4:  $infinite\ \{y \in C.\ y > x\}$  using 2 unfolding 3 by simp
  obtain y where 5:  $y \in C\ y > x$  using 4 by auto
  have 6:  $esize\ y > esize\ x$  using esize-strict-mono enat 5(2) by blast
  show ?thesis using that 5(1) 6 ileI1 by simp
next
  case (infinity)
  show ?thesis using that infinity assms(3) by simp
qed

lemma infinite-chain-arbitrary-esize[dest]:
  assumes Complete-Partial-Order.chain less-eq C infinite C
  obtains x
  where  $x \in C esize\ x \geq enat\ n$ 
proof (induct n arbitrary: thesis)
  case 0
  show ?case using assms(2) 0 by force
next
  case (Suc n)
  obtain x where 1:  $x \in C esize\ x \geq enat\ n$  using Suc(1) by blast
  obtain y where 2:  $y \in C esize\ y \geq eSuc\ (esize\ x)$  using assms 1(1) by rule

```

```

    show ?case using gfp.leq-trans Suc(2) 1(2) 2 by fastforce
  qed

end

class esize-ccpo = esize-order + ccpo
begin

lemma esize-cont[dest]:
  assumes Complete-Partial-Order.chain less-eq C C ≠ {}
  shows esize (⊔ C) = ⊔ (esize ` C)
proof (cases finite C)
  case False
  have 1: esize (⊔ C) = ∞
  proof
    fix n
    obtain A where 1: A ∈ C esize A ≥ enat n using assms(1) False by rule
    have 2: A ≤ ⊔ C using ccpo-Sup-upper assms(1) 1(1) by this
    have enat n ≤ esize A using 1(2) by this
    also have ... ≤ esize (⊔ C) using 2 by rule
    finally show enat n ≤ esize (⊔ C) by this
  qed
  have 2: (⊔ A ∈ C. esize A) = ∞
  proof
    fix n
    obtain A where 1: A ∈ C esize A ≥ enat n using assms(1) False by rule
    show enat n ≤ (⊔ A ∈ C. esize A) using SUP-upper2 1 by this
  qed
  show ?thesis using 1 2 by simp
next
  case True
  have 1: esize (⊔ C) = (⊔ x ∈ C. esize x)
  proof (intro order-class.order.antisym SUP-upper SUP-least esize-mono)
    show ⊔ C ∈ C using in-chain-finite assms(1) True assms(2) by this
    show ∧ x. x ∈ C ⇒ x ≤ ⊔ C using ccpo-Sup-upper assms(1) by this
  qed
  show ?thesis using 1 by simp
qed

lemma esize-mcont: mcont Sup less-eq Sup less-eq esize
  by (blast intro: mcontI monotoneI contI)

lemmas mcont2mcont-esome = esize-mcont[THEN lfp.mcont2mcont, simp, cont-intro]

end

end

```

13 Sets and Extended Natural Numbers

theory *ESet-Extensions*

imports

../Basics/Functions

Basic-Extensions

CCPO-Extensions

begin

lemma *card-lessThan-enat[simp]*: $\text{card } \{.. \lt enat $k\} = \text{card } \{.. \lt $k\}$$$

proof –

have 1: $\{.. \lt enat $k\} = \text{enat } \{.. \lt $k\}$$$

unfolding *lessThan-def image-Collect* **using** *enat-iless* **by force**

have $\text{card } \{.. \lt enat $k\} = \text{card } (\text{enat } \{.. \lt $k\})$ **unfolding** 1 **by rule**$$

also have $\dots = \text{card } \{.. \lt $k\}$ **using** *card-image inj-enat* **by metis**$

finally show *?thesis* **by this**

qed

lemma *card-atMost-enat[simp]*: $\text{card } \{.. \leq enat $k\} = \text{card } \{.. \leq $k\}$$$

proof –

have 1: $\{.. \leq enat $k\} = \text{enat } \{.. \leq $k\}$$$

unfolding *atMost-def image-Collect* **using** *enat-ile* **by force**

have $\text{card } \{.. \leq enat $k\} = \text{card } (\text{enat } \{.. \leq $k\})$ **unfolding** 1 **by rule**$$

also have $\dots = \text{card } \{.. \leq $k\}$ **using** *card-image inj-enat* **by metis**$

finally show *?thesis* **by this**

qed

lemma *enat-Collect*:

assumes $\infty \notin A$

shows $\{i. \text{enat } i \in A\} = \text{the-enat } A$

using *assms* **by** (*safe, force*) (*metis enat-the-enat*)

lemma *Collect-lessThan*: $\{i. \text{enat } i < n\} = \text{the-enat } \{.. \lt $n\}$$

proof –

have 1: $\infty \notin \{.. \lt $n\}$ **by simp**$

have $\{i. \text{enat } i < n\} = \{i. \text{enat } i \in \{.. \lt $n\}\}$ **by simp**$

also have $\dots = \text{the-enat } \{.. \lt $n\}$ **using** *enat-Collect 1* **by this**$

finally show *?thesis* **by this**

qed

instantiation *set* :: (*type*) *esize-ccpo*

begin

function *esize-set* **where** *finite* $A \implies \text{esize } A = \text{enat } (\text{card } A) \mid \text{infinite } A \implies \text{esize } A = \infty$

by auto termination by lexicographic-order

lemma *esize-iff-empty[iff]*: $\text{esize } A = 0 \iff A = \{\}$ **by** (*cases finite A, auto*)

lemma *esize-iff-infinite[iff]*: $\text{esize } A = \infty \iff \text{infinite } A$ **by force**

lemma *esize-singleton[simp]*: $\text{esize } \{a\} = \text{eSuc } 0$ **by simp**

lemma *esize-infinite-enat*[*dest, simp*]: $\text{infinite } A \implies \text{enat } k < \text{esize } A$ **by force**

instance

proof

fix $A :: 'a \text{ set}$

assume $1: \text{esize } A \neq \infty$

show *finite* $\{B. B \subseteq A\}$ **using 1 by simp**

next

fix $A B :: 'a \text{ set}$

assume $1: A \subseteq B$

show $\text{esize } A \leq \text{esize } B$

proof (*cases finite B*)

case *False*

show *?thesis* **using False by auto**

next

case *True*

have $2: \text{finite } A$ **using True 1 by auto**

show *?thesis* **using card-mono True 1 2 by auto**

qed

next

fix $A B :: 'a \text{ set}$

assume $1: \text{esize } A \neq \infty \ A \subset B$

show $\text{esize } A < \text{esize } B$ **using psubset-card-mono 1 by (cases finite B, auto)**

qed

end

lemma *esize-image*[*simp, intro*]:

assumes *inj-on f A*

shows $\text{esize } (f \text{ ` } A) = \text{esize } A$

using card-image finite-imageD assms by (cases finite A, auto)

lemma *esize-insert1*[*simp*]: $a \notin A \implies \text{esize } (\text{insert } a \ A) = \text{eSuc } (\text{esize } A)$

by (cases finite A, force+)

lemma *esize-insert2*[*simp*]: $a \in A \implies \text{esize } (\text{insert } a \ A) = \text{esize } A$

using insert-absorb by metis

lemma *esize-remove1*[*simp*]: $a \notin A \implies \text{esize } (A - \{a\}) = \text{esize } A$

by (cases finite A, force+)

lemma *esize-remove2*[*simp*]: $a \in A \implies \text{esize } (A - \{a\}) = \text{epred } (\text{esize } A)$

by (cases finite A, force+)

lemma *esize-union-disjoint*[*simp*]:

assumes $A \cap B = \{\}$

shows $\text{esize } (A \cup B) = \text{esize } A + \text{esize } B$

proof (*cases finite (A ∪ B)*)

case *True*

show *?thesis* **using card-Un-disjoint assms True by auto**

next

case *False*

show *?thesis* **using False by (cases finite A, auto)**

qed

```

lemma esize-lessThan[simp]: esize {..n} = n
proof (cases n)
  case (enat k)
    have 1: finite {..n} unfolding enat by (metis finite-lessThan-enat-iff
not-enat-eq)
    show ?thesis using 1 unfolding enat by simp
  next
  case (infinity)
    have 1: infinite {..n} unfolding infinity using infinite-lessThan-infty by
simp
    show ?thesis using 1 unfolding infinity by simp
  qed
lemma esize-atMost[simp]: esize {..n} = eSuc n
proof (cases n)
  case (enat k)
    have 1: finite {..n} unfolding enat by (metis atMost-iff finite-enat-bounded)
    show ?thesis using 1 unfolding enat by simp
  next
  case (infinity)
    have 1: infinite {..n}
    unfolding infinity
    by (metis atMost-iff enat-ord-code(3) infinite-lessThan-infty infinite-super
subsetI)
    show ?thesis using 1 unfolding infinity by simp
  qed

lemma least-eSuc[simp]:
  assumes A ≠ {}
  shows least (eSuc ' A) = eSuc (least A)
proof (rule antisym)
  obtain k where 10: k ∈ A using assms by blast
  have 11: eSuc k ∈ eSuc ' A using 10 by auto
  have 20: least A ∈ A using 10 LeastI by metis
  have 21: least (eSuc ' A) ∈ eSuc ' A using 11 LeastI by metis
  have 30:  $\bigwedge l. l \in A \implies \text{least } A \leq l$  using 10 Least-le by metis
  have 31:  $\bigwedge l. l \in \text{eSuc } ' A \implies \text{least } (\text{eSuc } ' A) \leq l$  using 11 Least-le by metis
  show least (eSuc ' A) ≤ eSuc (least A) using 20 31 by auto
  show eSuc (least A) ≤ least (eSuc ' A) using 21 30 by auto
qed

lemma Inf-enat-eSuc[simp]:  $\sqcap$  (eSuc ' A) = eSuc ( $\sqcap$  A) unfolding Inf-enat-def
by simp

definition lift :: nat set ⇒ nat set
  where lift A ≡ insert 0 (Suc ' A)

lemma liftI-0[intro, simp]: 0 ∈ lift A unfolding lift-def by auto
lemma liftI-Suc[intro]: a ∈ A ⇒ Suc a ∈ lift A unfolding lift-def by auto
lemma liftE[elim]:

```

assumes $b \in \text{lift } A$
obtains $(0) b = 0 \mid (\text{Suc}) a$ **where** $b = \text{Suc } a \ a \in A$
using *assms* **unfolding** *lift-def* **by** *auto*

lemma *lift-esize[simp]*: $\text{esize } (\text{lift } A) = \text{eSuc } (\text{esize } A)$ **unfolding** *lift-def* **by** *auto*

lemma *lift-least[simp]*: $\text{least } (\text{lift } A) = 0$ **unfolding** *lift-def* **by** *auto*

primrec *nth-least* :: $'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a :: \text{wellorder}$
where $\text{nth-least } A \ 0 = \text{least } A \mid \text{nth-least } A \ (\text{Suc } n) = \text{nth-least } (A - \{\text{least } A\}) \ n$

lemma *nth-least-wellformed[intro?, simp]*:

assumes $\text{enat } n < \text{esize } A$
shows $\text{nth-least } A \ n \in A$

using *assms*

proof (*induct n arbitrary: A*)

case *0*

show *?case* **using** *0* **by** *simp*

next

case $(\text{Suc } n)$

have $1: A \neq \{\}$ **using** *Suc(2)* **by** *auto*

have $2: \text{enat } n < \text{esize } (A - \{\text{least } A\})$ **using** *Suc(2)* 1 **by** *simp*

have $3: \text{nth-least } (A - \{\text{least } A\}) \ n \in A - \{\text{least } A\}$ **using** *Suc(1)* 2 **by** *this*

show *?case* **using** 3 **by** *simp*

qed

lemma *card-wellformed[intro?, simp]*:

fixes $k :: 'a :: \text{wellorder}$

assumes $k \in A$

shows $\text{enat } (\text{card } \{i \in A. i < k\}) < \text{esize } A$

proof (*cases finite A*)

case *False*

show *?thesis* **using** *False* **by** *simp*

next

case *True*

have $1: \text{esize } \{i \in A. i < k\} < \text{esize } A$ **using** *True* *assms* **by** *fastforce*

show *?thesis* **using** 1 **by** *simp*

qed

lemma *nth-least-strict-mono*:

assumes $\text{enat } l < \text{esize } A \ k < l$

shows $\text{nth-least } A \ k < \text{nth-least } A \ l$

using *assms*

proof (*induct k arbitrary: A l*)

case *0*

obtain l' **where** $1: l = \text{Suc } l'$ **using** $0(2)$ **by** (*metis gr0-conv-Suc*)

have $2: A \neq \{\}$ **using** $0(1)$ **by** *auto*

have $3: \text{enat } l' < \text{esize } (A - \{\text{least } A\})$ **using** $0(1)$ 2 **unfolding** 1 **by** *simp*

```

    have 4: nth-least (A - {least A}) l' ∈ A - {least A} using 3 by rule
    show ?case using 1 4 by (auto intro: le-neg-trans)
next
  case (Suc k)
  obtain l' where 1: l = Suc l' using Suc(3) by (metis Suc-lessE)
  have 2: A ≠ {} using Suc(2) by auto
  show ?case using Suc 2 unfolding 1 by simp
qed

lemma nth-least-mono[intro, simp]:
  assumes enat l < esize A k ≤ l
  shows nth-least A k ≤ nth-least A l
  using nth-least-strict-mono le-less assms by metis

lemma card-nth-least[simp]:
  assumes enat n < esize A
  shows card {k ∈ A. k < nth-least A n} = n
  using assms
  proof (induct n arbitrary: A)
    case 0
    have 1: {k ∈ A. k < least A} = {} using least-not-less by auto
    show ?case using nth-least.simps(1) card.empty 1 by metis
  next
    case (Suc n)
    have 1: A ≠ {} using Suc(2) by auto
    have 2: enat n < esize (A - {least A}) using Suc(2) 1 by simp
    have 3: nth-least A 0 < nth-least A (Suc n) using nth-least-strict-mono Suc(2)
  by blast
    have 4: {k ∈ A. k < nth-least A (Suc n)} =
      {least A} ∪ {k ∈ A - {least A}. k < nth-least (A - {least A}) n} using 1 3
  by auto
    have 5: card {k ∈ A - {least A}. k < nth-least (A - {least A}) n} = n using
  Suc(1) 2 by this
    have 6: finite {k ∈ A - {least A}. k < nth-least (A - {least A}) n}
      using 5 Collect-empty-eq card.infinite infinite-imp-nonempty least-not-less
  nth-least.simps(1)
    by (metis (no-types, lifting))
    have card {k ∈ A. k < nth-least A (Suc n)} =
      card ({least A} ∪ {k ∈ A - {least A}. k < nth-least (A - {least A}) n})
  using 4 by simp
    also have ... = card {least A} + card {k ∈ A - {least A}. k < nth-least (A
  - {least A}) n}
      using 6 by simp
    also have ... = Suc n using 5 by simp
    finally show ?case by this
  qed

lemma card-nth-least-le[simp]:
  assumes enat n < esize A

```

shows $\text{card } \{k \in A. k \leq \text{nth-least } A \ n\} = \text{Suc } n$
proof –
have 1: $\{k \in A. k \leq \text{nth-least } A \ n\} = \{\text{nth-least } A \ n\} \cup \{k \in A. k < \text{nth-least } A \ n\}$
using *assms by auto*
have 2: $\text{card } \{k \in A. k < \text{nth-least } A \ n\} = n$ **using** *assms by simp*
have 3: *finite* $\{k \in A. k < \text{nth-least } A \ n\}$
using 2 *Collect-empty-eq card.infinite infinite-imp-nonempty least-not-less nth-least.simps(1)*
by (*metis (no-types, lifting)*)
have $\text{card } \{k \in A. k \leq \text{nth-least } A \ n\} = \text{card } (\{\text{nth-least } A \ n\} \cup \{k \in A. k < \text{nth-least } A \ n\})$
unfolding 1 **by** *rule*
also $\text{have } \dots = \text{card } \{\text{nth-least } A \ n\} + \text{card } \{k \in A. k < \text{nth-least } A \ n\}$ **using** 3 **by** *simp*
also $\text{have } \dots = \text{Suc } n$ **using** *assms by simp*
finally **show** *?thesis by this*
qed

lemma *nth-least-card*:

fixes $k :: \text{nat}$
assumes $k \in A$
shows $\text{nth-least } A \ (\text{card } \{i \in A. i < k\}) = k$
proof (*rule nat-set-card-equality-less*)
have 1: $\text{enat } (\text{card } \{l \in A. l < k\}) < \text{esize } A$
proof (*cases finite A*)
case *False*
show *?thesis using False by simp*
next
case *True*
have 1: $\{l \in A. l < k\} \subset A$ **using** *assms by blast*
have 2: $\text{card } \{l \in A. l < k\} < \text{card } A$ **using** *psubset-card-mono True 1 by this*
show *?thesis using True 2 by simp*
qed
show $\text{nth-least } A \ (\text{card } \{l \in A. l < k\}) \in A$ **using** 1 **by** *rule*
show $k \in A$ **using** *assms by this*
show $\text{card } \{z \in A. z < \text{nth-least } A \ (\text{card } \{i \in A. i < k\})\} = \text{card } \{z \in A. z < k\}$ **using** 1 **by** *simp*
qed

interpretation *nth-least*:

bounded-function-pair $\{i. \text{enat } i < \text{esize } A\} A \ \text{nth-least } A \ \lambda k. \text{card } \{i \in A. i < k\}$
using *nth-least-wellformed card-wellformed by (unfold-locales, blast+)*

interpretation *nth-least*:

injection $\{i. \text{enat } i < \text{esize } A\} A \ \text{nth-least } A \ \lambda k. \text{card } \{i \in A. i < k\}$
using *card-nth-least by (unfold-locales, blast)*

interpretation *nth-least*:
surjection $\{i. \text{enat } i < \text{esize } A\} A \text{ nth-least } A \lambda k. \text{card } \{i \in A. i < k\}$
for $A :: \text{nat set}$
using *nth-least-card* **by** (*unfold-locales*, *blast*)

interpretation *nth-least*:
bijection $\{i. \text{enat } i < \text{esize } A\} A \text{ nth-least } A \lambda k. \text{card } \{i \in A. i < k\}$
for $A :: \text{nat set}$
by *unfold-locales*

lemma *nth-least-strict-mono-inverse*:
fixes $A :: \text{nat set}$
assumes $\text{enat } k < \text{esize } A \text{ enat } l < \text{esize } A \text{ nth-least } A k < \text{nth-least } A l$
shows $k < l$
using *assms* **by** (*metis not-less-iff-gr-or-eq nth-least-strict-mono*)

lemma *nth-least-less-card-less*:
fixes $k :: \text{nat}$
shows $\text{enat } n < \text{esize } A \wedge \text{nth-least } A n < k \longleftrightarrow n < \text{card } \{i \in A. i < k\}$

proof *safe*

assume $1: \text{enat } n < \text{esize } A \text{ nth-least } A n < k$
have $2: \text{nth-least } A n \in A$ **using** $1(1)$ **by** *rule*
have $n = \text{card } \{i \in A. i < \text{nth-least } A n\}$ **using** 1 **by** *simp*
also have $\dots < \text{card } \{i \in A. i < k\}$ **using** $1(2)$ 2 **by** *simp*
finally show $n < \text{card } \{i \in A. i < k\}$ **by** *this*

next

assume $1: n < \text{card } \{i \in A. i < k\}$
have $\text{enat } n < \text{enat } (\text{card } \{i \in A. i < k\})$ **using** 1 **by** *simp*
also have $\dots = \text{esize } \{i \in A. i < k\}$ **by** *simp*
also have $\dots \leq \text{esize } A$ **by** *blast*
finally show $2: \text{enat } n < \text{esize } A$ **by** *this*
have $3: n = \text{card } \{i \in A. i < \text{nth-least } A n\}$ **using** 2 **by** *simp*
have $4: \text{card } \{i \in A. i < \text{nth-least } A n\} < \text{card } \{i \in A. i < k\}$ **using** 1 2 **by**

simp

have $5: \text{nth-least } A n \in A$ **using** 2 **by** *rule*
show $\text{nth-least } A n < k$ **using** 4 5 **by** *simp*

qed

lemma *nth-least-less-esize-less*:
 $\text{enat } n < \text{esize } A \wedge \text{enat } (\text{nth-least } A n) < k \longleftrightarrow \text{enat } n < \text{esize } \{i \in A. \text{enat } i < k\}$
using *nth-least-less-card-less* **by** (*cases k*, *simp+*)

lemma *nth-least-le*:
assumes $\text{enat } n < \text{esize } A$
shows $n \leq \text{nth-least } A n$
using *assms*
proof (*induct n*)

```

    case 0
    show ?case using 0 by simp
next
  case (Suc n)
  have n ≤ nth-least A n using Suc by (metis Suc-ile-eq less-imp-le)
  also have ... < nth-least A (Suc n) using nth-least-strict-mono Suc(2) by
blast
  finally show ?case by simp
qed

lemma nth-least-eq:
  assumes enat n < esize A enat n < esize B
  assumes  $\bigwedge i. i \leq \text{nth-least } A \ n \implies i \leq \text{nth-least } B \ n \implies i \in A \longleftrightarrow i \in B$ 
  shows nth-least A n = nth-least B n
using assms
proof (induct n arbitrary: A B)
  case 0
  have 1: least A = least B
  proof (rule least-eq)
    show A ≠ {} using 0(1) by simp
    show B ≠ {} using 0(2) by simp
  next
  fix i
  assume 2: i ≤ least A i ≤ least B
  show i ∈ A ↔ i ∈ B using 0(3) 2 unfolding nth-least.simps by this
  qed
  show ?case using 1 by simp
next
  case (Suc n)
  have 1: A ≠ {} B ≠ {} using Suc(2, 3) by auto
  have 2: least A = least B
  proof (rule least-eq)
    show A ≠ {} using 1(1) by this
    show B ≠ {} using 1(2) by this
  next
  fix i
  assume 3: i ≤ least A i ≤ least B
  have 4: nth-least A 0 ≤ nth-least A (Suc n) using Suc(2) by blast
  have 5: nth-least B 0 ≤ nth-least B (Suc n) using Suc(3) by blast
  have 6: i ≤ nth-least A (Suc n) i ≤ nth-least B (Suc n) using 3 4 5 by auto
  show i ∈ A ↔ i ∈ B using Suc(4) 6 by this
  qed
  have 3: nth-least (A - {least A}) n = nth-least (B - {least B}) n
  proof (rule Suc(1))
    show enat n < esize (A - {least A}) using Suc(2) 1(1) by simp
    show enat n < esize (B - {least B}) using Suc(3) 1(2) by simp
  next
  fix i
  assume 3: i ≤ nth-least (A - {least A}) n i ≤ nth-least (B - {least B}) n

```

```

    have 4:  $i \leq \text{nth-least } A \text{ (Suc } n) \leq \text{nth-least } B \text{ (Suc } n)$  using 3 by simp+
    have 5:  $i \in A \longleftrightarrow i \in B$  using Suc(4) 4 by this
    show  $i \in A - \{\text{least } A\} \longleftrightarrow i \in B - \{\text{least } B\}$  using 2 5 by auto
  qed
  show ?case using 3 by simp
qed

lemma nth-least-restrict[simp]:
  assumes  $\text{enat } i < \text{esize } \{i \in s. \text{enat } i < k\}$ 
  shows  $\text{nth-least } \{i \in s. \text{enat } i < k\} i = \text{nth-least } s i$ 
proof (rule nth-least-eq)
  show  $\text{enat } i < \text{esize } \{i \in s. \text{enat } i < k\}$  using assms by this
  show  $\text{enat } i < \text{esize } s$  using nth-least-less-esize-less assms by auto
next
  fix l
  assume 1:  $l \leq \text{nth-least } \{i \in s. \text{enat } i < k\} i$ 
  have 2:  $\text{nth-least } \{i \in s. \text{enat } i < k\} i \in \{i \in s. \text{enat } i < k\}$  using assms by
rule
  have  $\text{enat } l \leq \text{enat } (\text{nth-least } \{i \in s. \text{enat } i < k\} i)$  using 1 by simp
  also have  $\dots < k$  using 2 by simp
  finally show  $l \in \{i \in s. \text{enat } i < k\} \longleftrightarrow l \in s$  by auto
qed

lemma least-nth-least[simp]:
  assumes  $A \neq \{\} \wedge i. i \in A \implies \text{enat } i < \text{esize } B$ 
  shows  $\text{least } (\text{nth-least } B \text{ ' } A) = \text{nth-least } B \text{ (least } A)$ 
  using assms by simp

lemma nth-least-nth-least[simp]:
  assumes  $\text{enat } n < \text{esize } A \wedge i. i \in A \implies \text{enat } i < \text{esize } B$ 
  shows  $\text{nth-least } B \text{ (nth-least } A \ n) = \text{nth-least } (\text{nth-least } B \text{ ' } A) \ n$ 
  using assms
proof (induct n arbitrary: A)
  case 0
  show ?case using 0 by simp
next
  case (Suc n)
  have 1:  $A \neq \{\}$  using Suc(2) by auto
  have 2:  $\text{nth-least } B \text{ ' } (A - \{\text{least } A\}) = \text{nth-least } B \text{ ' } A - \text{nth-least } B \text{ ' } \{\text{least } A\}$ 
  proof (rule inj-on-image-set-diff)
    show  $\text{inj-on } (\text{nth-least } B) \{i. \text{enat } i < \text{esize } B\}$  using nth-least.inj-on by this
    show  $A - \{\text{least } A\} \subseteq \{i. \text{enat } i < \text{esize } B\}$  using Suc(3) by blast
    show  $\{\text{least } A\} \subseteq \{i. \text{enat } i < \text{esize } B\}$  using Suc(3) 1 by force
  qed
  have  $\text{nth-least } B \text{ (nth-least } A \text{ (Suc } n)) = \text{nth-least } B \text{ (nth-least } (A - \{\text{least } A\}) \ n)$  by simp
  also have  $\dots = \text{nth-least } (\text{nth-least } B \text{ ' } (A - \{\text{least } A\})) \ n$  using Suc 1 by
force

```

also have $\dots = \text{nth-least } (\text{nth-least } B \text{ ' } A - \text{nth-least } B \text{ ' } \{ \text{least } A \}) n$ **unfolding**
2 by rule
also have $\dots = \text{nth-least } (\text{nth-least } B \text{ ' } A - \{ \text{nth-least } B \text{ (least } A \}) \} n$ **by simp**
also have $\dots = \text{nth-least } (\text{nth-least } B \text{ ' } A - \{ \text{least } (\text{nth-least } B \text{ ' } A) \}) n$ **using**
Suc(3) 1 by auto
also have $\dots = \text{nth-least } (\text{nth-least } B \text{ ' } A) (\text{Suc } n)$ **by simp**
finally show *?case* **by this**
qed

lemma *nth-least-Max[simp]*:

assumes *finite A A ≠ {}*

shows $\text{nth-least } A \text{ (card } A - 1) = \text{Max } A$

using *assms*

proof (*induct card A - 1 arbitrary: A*)

case *0*

have *1: card A = 1* **using** *0* **by** (*metis One-nat-def Suc-diff-1 card-gt-0-iff*)

obtain *a* **where** *2: A = {a}* **using** *1* **by rule**

show *?case* **unfolding** *2* **by** (*simp del: insert-iff*)

next

case (*Suc n*)

have *1: least A ∈ A* **using** *Suc(4)* **by rule**

have *2: card (A - {least A}) = Suc n* **using** *Suc(2, 3) 1* **by simp**

have *3: A - {least A} ≠ {}* **using** *2 Suc(3)* **by fastforce**

have $\text{nth-least } A \text{ (card } A - 1) = \text{nth-least } A \text{ (Suc } n)$ **unfolding** *Suc(2)* **by**

rule

also have $\dots = \text{nth-least } (A - \{ \text{least } A \}) n$ **by simp**

also have $\dots = \text{nth-least } (A - \{ \text{least } A \}) \text{ (card } (A - \{ \text{least } A \}) - 1)$ **unfolding**

2 by simp

also have $\dots = \text{Max } (A - \{ \text{least } A \})$

proof (*rule Suc(1)*)

show $n = \text{card } (A - \{ \text{least } A \}) - 1$ **unfolding** *2* **by simp**

show *finite (A - {least A})* **using** *Suc(3)* **by simp**

show $A - \{ \text{least } A \} \neq \{ \}$ **using** *3* **by this**

qed

also have $\dots = \text{Max } A$ **using** *Suc(3) 3* **by simp**

finally show *?case* **by this**

qed

lemma *nth-least-le-Max*:

assumes *finite A A ≠ {} enat n < esize A*

shows $\text{nth-least } A \text{ } n \leq \text{Max } A$

proof -

have $\text{nth-least } A \text{ } n \leq \text{nth-least } A \text{ (card } A - 1)$

proof (*rule nth-least-mono*)

show $\text{enat } (\text{card } A - 1) < \text{esize } A$ **by** (*metis Suc-diff-1 Suc-ile-eq assms(1)*)

assms(2)

card-eq-0-iff esize-set.simps(1) not-gr0 order-refl)

show $n \leq \text{card } A - 1$ **by** (*metis Suc-diff-1 Suc-leI antisym-conv assms(1)*)

assms(3)

```

    enat-ord-simps(2) esize-set.simps(1) le-less neq-iff not-gr0)
  qed
  also have ... = Max A using nth-least-Max assms(1, 2) by this
  finally show ?thesis by this
  qed

lemma nth-least-not-contains:
  fixes k :: nat
  assumes enat (Suc n) < esize A nth-least A n < k k < nth-least A (Suc n)
  shows k ∉ A
  proof
    assume 1: k ∈ A
    have 2: nth-least A (card {i ∈ A. i < k}) = k using nth-least.right-inverse 1
  by this
    have 3: n < card {i ∈ A. i < k}
    proof (rule nth-least-strict-mono-inverse)
      show enat n < esize A using assms(1) by auto
      show enat (card {i ∈ A. i < k}) < esize A using nth-least.g.wellformed 1
    by simp
      show nth-least A n < nth-least A (card {i ∈ A. i < k}) using assms(2) 2
    by simp
    qed
    have 4: card {i ∈ A. i < k} < Suc n
    proof (rule nth-least-strict-mono-inverse)
      show enat (card {i ∈ A. i < k}) < esize A using nth-least.g.wellformed 1
    by simp
      show enat (Suc n) < esize A using assms(1) by this
      show nth-least A (card {i ∈ A. i < k}) < nth-least A (Suc n) using assms(3)
    2 by simp
    qed
    show False using 3 4 by auto
  qed

lemma nth-least-Suc[simp]:
  assumes enat n < esize A
  shows nth-least (Suc ' A) n = Suc (nth-least A n)
  using assms
  proof (induct n arbitrary: A)
    case (0)
    have 1: A ≠ {} using 0 by auto
    show ?case using 1 by simp
  next
    case (Suc n)
    have 1: enat n < esize (A - {least A})
    proof -
      have 2: A ≠ {} using Suc(2) by auto
      have 3: least A ∈ A using LeastI 2 by fast
      have 4: A = insert (least A) A using 3 by auto
      have eSuc (enat n) = enat (Suc n) by simp
    qed
  qed

```

```

    also have ... < esize A using Suc(2) by this
    also have ... = esize (insert (least A) A) using 4 by simp
    also have ... = eSuc (esize (A - {least A})) using 3 2 by simp
    finally show ?thesis using Extended-Nat.eSuc-mono by metis
qed
have nth-least (Suc ' A) (Suc n) = nth-least (Suc ' A - {least (Suc ' A)}) n
by simp
also have ... = nth-least (Suc ' (A - {least A})) n by simp
also have ... = Suc (nth-least (A - {least A}) n) using Suc(1) 1 by this
also have ... = Suc (nth-least A (Suc n)) by simp
finally show ?case by this
qed

```

```

lemma nth-least-lift[simp]:
  nth-least (lift A) 0 = 0
  enat n < esize A  $\implies$  nth-least (lift A) (Suc n) = Suc (nth-least A n)
  unfolding lift-def by simp+

```

```

lemma nth-least-list-card[simp]:
  assumes enat n  $\leq$  esize A
  shows card {k  $\in$  A. k < nth-least (lift A) n} = n
  using less-Suc-eq-le assms by (cases n, auto simp del: nth-least.simps)

```

end

14 Coinductive Lists

```

theory Coinductive-List-Extensions
imports

```

```

  Coinductive.Coinductive-List
  Coinductive.Coinductive-List-Prefix
  Coinductive.Coinductive-Stream
  ../Extensions/List-Extensions
  ../Extensions/ESet-Extensions

```

```

begin

```

```

hide-const (open) Sublist.prefix
hide-const (open) Sublist.suffix

```

```

declare list-of-lappend[simp]
declare lnth-lappend1 [simp]
declare lnth-lappend2 [simp]
declare lprefix-llength-le[dest]
declare Sup-llist-def[simp]
declare length-list-of[simp]
declare llast-linfinite[simp]
declare lnth-ltake[simp]
declare lappend-assoc[simp]
declare lprefix-lappend[simp]

```

lemma *lprefix-lSup-revert*: $lSup = Sup\ lprefix = less-eq$ **by** *auto*
lemma *admissible-lprefixI*[*cont-intro*]:
assumes *mcont lub ord lSup lprefix f*
assumes *mcont lub ord lSup lprefix g*
shows *ccpo.admissible lub ord* $(\lambda x. lprefix\ (f\ x)\ (g\ x))$
using *ccpo-class.admissible-leI assms* **unfolding** *lprefix-lSup-revert* **by** *this*
lemma *lList-lift-admissible*:
assumes *ccpo.admissible lSup lprefix P*
assumes $\bigwedge u. u \leq v \implies lfinite\ u \implies P\ u$
shows $P\ v$
using *assms* **by** $(metis\ LNil-lprefix\ le-lList-conv-lprefix\ lfinite.simps\ lList-gen-induct)$

abbreviation $lfinite\ w \equiv \neg\ lfinite\ w$

notation *LNil* $\langle \langle \rangle \rangle$
notation *LCons* (**infixr** $\langle \% \rangle$ 65)
notation *lzip* (**infixr** $\langle || \rangle$ 51)
notation *lappend* (**infixr** $\langle \$ \rangle$ 65)
notation *lnth* (**infixl** $\langle ?! \rangle$ 100)

syntax *-lList* :: *args* \Rightarrow 'a *lList* $\langle \langle - \rangle \rangle$
syntax-consts *-lList* \Leftarrow *LCons*

translations

$\langle a, x \rangle \Leftarrow a\ \% \langle x \rangle$
 $\langle a \rangle \Leftarrow a\ \% \langle \rangle$

lemma *eq-LNil-conv-lnull*[*simp*]: $w = \langle \rangle \longleftrightarrow lnull\ w$ **by** *auto*
lemma *Collect-lnull*[*simp*]: $\{w. lnull\ w\} = \{\langle \rangle\}$ **by** *auto*

lemma *inj-on-ltake*: *inj-on* $(\lambda k. ltake\ k\ w)$ $\{.. llength\ w\}$
by $(rule\ inj-onI, auto, metis\ llength-ltake\ min-def)$

lemma *lnth-inf-lList'*[*simp*]: $lnth\ (inf-lList\ f) = f$ **by** *auto*

lemma *not-lnull-lappend-startE*[*elim*]:
assumes $\neg\ lnull\ w$
obtains *a v*
where $w = \langle a \rangle \$ v$
using *not-lnull-conv assms* **by** $(simp, metis)$

lemma *not-lnull-lappend-endE*[*elim*]:

assumes $\neg\ lnull\ w$
obtains *a v*
where $w = v \$ \langle a \rangle$
proof $(cases\ lfinite\ w)$
case *False*
show *?thesis*
proof
show $w = w \$ \langle a \rangle$ **using** *lappend-inf False* **by** *force*

```

qed
next
case True
show ?thesis
using True assms that
proof (induct arbitrary: thesis)
  case (lfinite-LNil)
  show ?case using lfinite-LNil by auto
next
case (lfinite-LConsI w a)
show ?case
proof (cases lnull w)
  case False
  obtain b v where 1: w = v $ <b> using lfinite-LConsI(2) False by this
  show ?thesis
  proof (rule lfinite-LConsI(4))
    show a % w = (a % v) $ <b> unfolding 1 by simp
  qed
next
case True
show ?thesis
proof (rule lfinite-LConsI(4))
  show a % w = <> $ <a> using True by simp
qed
qed
qed
qed

```

```

lemma llength-lappend-startE[elim]:
  assumes llength w ≥ eSuc n
  obtains a v
  where w = <a> $ v llength v ≥ n
proof -
  have 1: ¬ lnull w using assms by auto
  show ?thesis using assms 1 that by auto
qed

```

```

lemma llength-lappend-endE[elim]:
  assumes llength w ≥ eSuc n
  obtains a v
  where w = v $ <a> llength v ≥ n
proof -
  have 1: ¬ lnull w using assms by auto
  show ?thesis using assms 1 that by auto
qed

```

```

lemma llength-lappend-start'E[elim]:
  assumes llength w = enat (Suc n)
  obtains a v
  where w = <a> $ v llength v = enat n

```


proof –
have 1: $\text{length } w \geq \text{eSuc } (\text{enat } n)$ **using** *assms* **by** *simp*
obtain $a \ v$ **where** 2: $w = \langle a \rangle \$ v$ **using** 1 **by** *blast*
show *?thesis*
proof
show $w = \langle a \rangle \$ v$ **using** 2(1) **by** *this*
show $\text{length } v = \text{enat } n$ **using** *assms* **unfolding** 2(1) **by** (*simp*, *metis*
eSuc-enat eSuc-inject)
qed
qed
lemma *length-lappend-end'E[elim]*:
assumes $\text{length } w = \text{enat } (\text{Suc } n)$
obtains $a \ v$
where $w = v \$ \langle a \rangle$ $\text{length } v = \text{enat } n$
proof –
have 1: $\text{length } w \geq \text{eSuc } (\text{enat } n)$ **using** *assms* **by** *simp*
obtain $a \ v$ **where** 2: $w = v \$ \langle a \rangle$ **using** 1 **by** *blast*
show *?thesis*
proof
show $w = v \$ \langle a \rangle$ **using** 2(1) **by** *this*
show $\text{length } v = \text{enat } n$ **using** *assms* **unfolding** 2(1) **by** (*simp*, *metis*
eSuc-enat eSuc-inject)
qed
qed

lemma *ltake-llast[simp]*:
assumes $\text{enat } k < \text{length } w$
shows $\text{llast } (\text{ltake } (\text{enat } (\text{Suc } k)) \ w) = w \ ?! \ k$
proof –
have 1: $\text{length } (\text{ltake } (\text{enat } (\text{Suc } k)) \ w) = \text{eSuc } (\text{enat } k)$ **using** *min.absorb-iff1*
assms **by** *auto*
have $\text{llast } (\text{ltake } (\text{enat } (\text{Suc } k)) \ w) = \text{ltake } (\text{enat } (\text{Suc } k)) \ w \ ?! \ k$
using *llast-conv-lnth 1* **by** *this*
also have $\dots = w \ ?! \ k$ **by** (*rule lnth-ltake*, *simp*)
finally show *?thesis* **by** *this*
qed

lemma *linfinite-length[dest, simp]*:
assumes *linfinite* w
shows $\text{enat } k < \text{length } w$
using *assms* *not-lfinite-length* **by** *force*

lemma *llist-nth-eqI[intro]*:
assumes $\text{length } u = \text{length } v$
assumes $\bigwedge i. \text{enat } i < \text{length } u \implies \text{enat } i < \text{length } v \implies u \ ?! \ i = v \ ?! \ i$
shows $u = v$
using *assms*
proof (*coinduction arbitrary: u v*)
case *Eq-llist*

```

have 10: llength u = llength v using Eq-llist by auto
have 11:  $\bigwedge i. \text{enat } i < \text{llength } u \implies \text{enat } i < \text{llength } v \implies u \text{ ?! } i = v \text{ ?! } i$ 
  using Eq-llist by auto
show ?case
proof (intro conjI impI exI allI)
  show lnull u  $\longleftrightarrow$  lnull v using 10 by auto
next
  assume 20:  $\neg \text{lnull } u \neg \text{lnull } v$ 
  show lhd u = lhd v using lhd-conv-lnth enat-0 11 20 by force
next
  show ltl u = ltl u by rule
next
  show ltl v = ltl v by rule
next
  assume 30:  $\neg \text{lnull } u \neg \text{lnull } v$ 
  show llength (ltl u) = llength (ltl v) using 10 30 by force
next
  fix i
  assume 40:  $\neg \text{lnull } u \neg \text{lnull } v \text{enat } i < \text{llength } (\text{ltl } u) \text{enat } i < \text{llength } (\text{ltl } v)$ 
  have 41:  $u \text{ ?! } \text{Suc } i = v \text{ ?! } \text{Suc } i$ 
  proof (rule 11)
    show enat (Suc i) < llength u using Suc-ile-eq 40(1) 40(3) by auto
    show enat (Suc i) < llength v using Suc-ile-eq 40(2) 40(4) by auto
  qed
  show ltl u ?! i = ltl v ?! i using lnth-ltl 40(1-2) 41 by metis
qed
qed

primcorec lscan :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'a llist  $\Rightarrow$  'b  $\Rightarrow$  'b llist
  where lscan f w a = (case w of <>  $\Rightarrow$  <a> | x % xs  $\Rightarrow$  a % lscan f xs (f x a))

lemma lscan-simps[simp]:
  lscan f <> a = <a>
  lscan f (x % xs) a = a % lscan f xs (f x a)
  by (metis llist.simps(4) lscan.code, metis llist.simps(5) lscan.code)

lemma lscan-lfinite[iff]: lfinite (lscan f w a)  $\longleftrightarrow$  lfinite w
proof
  assume lfinite (lscan f w a)
  thus lfinite w
proof (induct lscan f w a arbitrary: w a rule: lfinite-induct)
  case LNil
  show ?case using LNil by simp
next
  case LCons
  show ?case by (cases w, simp, simp add: LCons(3))
qed
next
  assume lfinite w

```

```

    thus lfinite (lscan f w a) by (induct arbitrary: a, auto)
  qed
lemma lscan-llength[simp]: llength (lscan f w a) = eSuc (llength w)
proof (cases lfinite w)
  case False
    have 1: llength (lscan f w a) = ∞ using not-lfinite-llength False by auto
    have 2: llength w = ∞ using not-lfinite-llength False by auto
    show ?thesis using 1 2 by simp
  next
    case True
    show ?thesis using True by (induct arbitrary: a, auto)
  qed

function lfold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a llist ⇒ 'b ⇒ 'b
  where lfinite w ⇒ lfold f w = fold f (list-of w) | linfinite w ⇒ lfold f w = id
  by (auto, metis) termination by lexicographic-order

lemma lfold-llist-of[simp]: lfold f (llist-of xs) = fold f xs by simp

lemma finite-UNIV-llength-eq:
  assumes finite (UNIV :: 'a set)
  shows finite {w :: 'a llist. llength w = enat n}
proof (induct n)
  case (0)
  show ?case by simp
  next
    case (Suc n)
    have 1: finite ({v. llength v = enat n} × UNIV :: ('a llist × 'a) set)
      using Suc assms by simp
    have 2: finite ((λ (v, a). v $ <a> :: 'a llist) ' ({v. llength v = enat n} × UNIV))
      using 1 by auto
    have 3: finite {v $ <a> :: 'a llist | v a. llength v = enat n}
    proof -
      have 0: {v $ <a> :: 'a llist | v a. llength v = enat n} =
        (λ (v, a). v $ <a> :: 'a llist) ' ({v. llength v = enat n} × UNIV) by auto
      show ?thesis using 2 unfolding 0 by this
    qed
    have 4: finite {w :: 'a llist . llength w = enat (Suc n)}
    proof -
      have 0: {w :: 'a llist . llength w = enat (Suc n)} =
        {v $ <a> :: 'a llist | v a. llength v = enat n} by force
      show ?thesis using 3 unfolding 0 by this
    qed
    show ?case using 4 by this
  qed
qed

lemma finite-UNIV-llength-le:
  assumes finite (UNIV :: 'a set)
  shows finite {w :: 'a llist. llength w ≤ enat n}

```

proof –
have $1: \{w. \text{length } w \leq \text{enat } n\} = (\bigcup k \leq n. \{w. \text{length } w = \text{enat } k\})$
by (*auto*, *metis atMost-iff enat-ile enat-ord-simps(1)*)
show *?thesis unfolding 1 using finite-UNIV-length-eq assms by auto*
qed

lemma *lprefix-ltake[dest]: $u \leq v \implies u = \text{ltake } (\text{length } u) v$*
by (*metis le-llist-conv-lprefix lprefix-conv-lappend ltake-all ltake-lappend1 order-refl*)

lemma *prefixes-set: $\{v. v \leq w\} = \{\text{ltake } k w \mid k. k \leq \text{length } w\}$ **by** *fastforce**
lemma *esize-prefixes[simp]: $\text{esize } \{v. v \leq w\} = \text{eSuc } (\text{length } w)$*

proof –
have $\text{esize } \{v. v \leq w\} = \text{esize } \{\text{ltake } k w \mid k. k \leq \text{length } w\}$ **unfolding**
prefixes-set by rule
also have $\dots = \text{esize } ((\lambda k. \text{ltake } k w) \text{ ` } \{.. \text{length } w\})$
unfolding *atMost-def image-Collect by rule*
also have $\dots = \text{esize } \{.. \text{length } w\}$ **using** *inj-on-ltake esize-image by blast*
also have $\dots = \text{eSuc } (\text{length } w)$ **by** *simp*
finally show *?thesis by this*

qed
lemma *prefix-subsume: $v \leq w \implies u \leq w \implies \text{length } v \leq \text{length } u \implies v \leq u$*
by (*metis le-llist-conv-lprefix lprefix-conv-lappend lprefix-ltake ltake-is-lprefix ltake-lappend1*)

lemma *ltake-infinite[simp]: $\text{ltake } \infty w = w$ **by** (*metis enat-ord-code(3) ltake-all*)*

lemma *lprefix-infinite:*
assumes $u \leq v$ *linfinite* u
shows $u = v$

proof –
have $1: \text{length } u = \infty$ **using** *not-lfinite-length assms(2) by this*
have $u = \text{ltake } (\text{length } u) v$ **using** *lprefix-ltake assms(1) by this*
also have $\dots = v$ **using** 1 **by** *simp*
finally show *?thesis by this*

qed

instantiation *llist :: (type) esize-order*
begin

definition *[simp]: $\text{esize} \equiv \text{length}$*

instance

proof

fix $w :: 'a \text{ llist}$

assume $1: \text{esize } w \neq \infty$

show *finite* $\{v. v \leq w\}$

using *esize-prefixes 1 by (metis eSuc-eq-infinity-iff esize-set.simps(2) esize-llist-def)*

next

```

    fix u v :: 'a llist
    assume 1: u ≤ v
    show esize u ≤ esize v using lprefix-llength-le 1 by auto
next
    fix u v :: 'a llist
    assume 1: u < v
    show esize u < esize v using lstrict-prefix-llength-less 1 by auto
qed

end

```

14.1 Index Sets

definition *liset* :: 'a set ⇒ 'a llist ⇒ nat set
where *liset* A w ≡ {i. enat i < llength w ∧ w ?! i ∈ A}

lemma *lisetI[intro]*:
assumes enat i < llength w w ?! i ∈ A
shows i ∈ *liset* A w
using *assms* **unfolding** *liset-def* **by** auto

lemma *lisetD[dest]*:
assumes i ∈ *liset* A w
shows enat i < llength w w ?! i ∈ A
using *assms* **unfolding** *liset-def* **by** auto

lemma *liset-finite*:
assumes lfinite w
shows finite (*liset* A w)

proof –
show *liset* A w ⊆ {i. enat i < llength w} **by** auto
show finite {i. enat i < llength w} **using** lfinite-finite-index *assms* **by** this
qed

lemma *liset-nil[simp]*: *liset* A <> = {} **by** auto

lemma *liset-cons-not-member[simp]*:
assumes a ∉ A
shows *liset* A (a % w) = Suc ‘ *liset* A w

proof –
have *liset* A (a % w) = {i. enat i < llength (a % w) ∧ (a % w) ?! i ∈ A} **by**
auto
also have ... = Suc ‘ {i. enat (Suc i) < llength (a % w) ∧ (a % w) ?! Suc i
∈ A}
using *Collect-split-Suc(1)* *assms* **by** *simp*
also have ... = Suc ‘ {i. enat i < llength w ∧ w ?! i ∈ A} **using** *Suc-ile-eq*
by *simp*
also have ... = Suc ‘ *liset* A w **by** auto
finally show ?thesis **by** this

qed
lemma *liset-cons-member[simp]*:

```

assumes  $a \in A$ 
shows  $\text{liset } A (a \% w) = \{0\} \cup \text{Suc } \text{' liset } A w$ 
proof –
  have  $\text{liset } A (a \% w) = \{i. \text{enat } i < \text{llength } (a \% w) \wedge (a \% w) \text{ ?! } i \in A\}$  by
  auto
  also have  $\dots = \{0\} \cup \text{Suc } \text{' } \{i. \text{enat } (\text{Suc } i) < \text{llength } (a \% w) \wedge (a \% w) \text{ ?! } i \in A\}$ 
  ?! Suc i ∈ A
    using Collect-split-Suc(2) assms by simp
  also have  $\dots = \{0\} \cup \text{Suc } \text{' } \{i. \text{enat } i < \text{llength } w \wedge w \text{ ?! } i \in A\}$  using
  Suc-ile-eq by simp
  also have  $\dots = \{0\} \cup \text{Suc } \text{' liset } A w$  by auto
  finally show ?thesis by this
qed

```

lemma *liset-prefix*:

```

assumes  $i \in \text{liset } A v \ u \leq v \ \text{enat } i < \text{llength } u$ 
shows  $i \in \text{liset } A u$ 
unfolding liset-def
proof (intro CollectI conjI)
  have  $1: v \text{ ?! } i \in A$  using assms(1) by auto
  show  $\text{enat } i < \text{llength } u$  using assms(3) by this
  show  $u \text{ ?! } i \in A$  using lprefix-lnthD assms(2, 3) 1 by force
qed

```

lemma *liset-suffix*:

```

assumes  $i \in \text{liset } A u \ u \leq v$ 
shows  $i \in \text{liset } A v$ 
unfolding liset-def
proof (intro CollectI conjI)
  have  $1: \text{enat } i < \text{llength } u \ u \text{ ?! } i \in A$  using assms(1) by auto
  show  $\text{enat } i < \text{llength } v$  using lprefix-llength-le 1(1) assms(2) by fastforce
  show  $v \text{ ?! } i \in A$  using lprefix-lnthD assms(2) 1 by force
qed

```

lemma *liset-ltake[simp]*: $\text{liset } A (\text{ltake } (\text{enat } k) w) = \text{liset } A w \cap \{.. < k\}$

```

proof (intro equalityI subsetI)
  fix  $i$ 
  assume  $1: i \in \text{liset } A (\text{ltake } (\text{enat } k) w)$ 
  have  $2: \text{enat } i < \text{enat } k$  using  $1$  by auto
  have  $3: \text{ltake } (\text{enat } k) w \text{ ?! } i = w \text{ ?! } i$  using lnth-ltake 2 by this
  show  $i \in \text{liset } A w \cap \{.. < k\}$  using  $1\ 3$  by fastforce
next
  fix  $i$ 
  assume  $1: i \in \text{liset } A w \cap \{.. < k\}$ 
  have  $2: \text{enat } i < \text{enat } k$  using  $1$  by auto
  have  $3: \text{ltake } (\text{enat } k) w \text{ ?! } i = w \text{ ?! } i$  using lnth-ltake 2 by this
  show  $i \in \text{liset } A (\text{ltake } (\text{enat } k) w)$  using  $1\ 3$  by fastforce
qed

```

lemma *liset-mono[dest]*: $u \leq v \implies \text{liset } A u \subseteq \text{liset } A v$

```

    unfolding liset-def using lprefix-lnthD by fastforce
lemma liset-cont[dest]:
  assumes Complete-Partial-Order.chain less-eq C C ≠ {}
  shows liset A (⊔ C) = (⋃ w ∈ C. liset A w)
proof safe
  fix i
  assume 1: i ∈ liset A (⊔ C)
  show i ∈ (⋃ w ∈ C. liset A w)
  proof (cases finite C)
    case False
    obtain w where 2: w ∈ C enat i < llength w
    using esize-llist-def infinite-chain-arbitrary-esize assms(1) False Suc-ile-eq
  by metis
    have 3: w ≤ ⊔ C using chain-lprefix-lSup assms(1) 2(1) by simp
    have 4: i ∈ liset A w using liset-prefix 1 3 2(2) by this
    show ?thesis using 2(1) 4 by auto
  next
    case True
    have 2: ⊔ C ∈ C using in-chain-finite assms(1) True assms(2) by this
    show ?thesis using 1 2 by auto
  qed
next
  fix w i
  assume 1: w ∈ C i ∈ liset A w
  have 2: w ≤ ⊔ C using chain-lprefix-lSup assms(1) 1(1) by simp
  show i ∈ liset A (⊔ C) using liset-suffix 1(2) 2 by this
qed

lemma liset-mcont: Complete-Partial-Order2.mcont lSup lprefix Sup less-eq
(liset A)
  unfolding lprefix-lSup-revert by (blast intro: mcontI monotoneI contI)

lemmas mcont2mcont-liset = liset-mcont[THEN lfp.mcont2mcont, simp, cont-intro]

```

14.2 Selections

abbreviation $lproject\ A \equiv lfilter\ (\lambda\ a.\ a \in A)$
abbreviation $lselect\ s\ w \equiv lnths\ w\ s$

lemma $lselect\ to\ lproject$: $lselect\ s\ w = lmap\ fst\ (lproject\ (UNIV \times s)\ (w \parallel iterates\ Suc\ 0))$

proof –
 have 1: $\{(x, y). y \in s\} = UNIV \times s$ by auto
 have $lselect\ s\ w = lmap\ fst\ (lproject\ \{(x, y). y \in s\}\ (w \parallel iterates\ Suc\ 0))$
 unfolding $lnths\ def$ by simp
 also have $\dots = lmap\ fst\ (lproject\ (UNIV \times s)\ (w \parallel iterates\ Suc\ 0))$ unfolding
 1 by rule
 finally show ?thesis by this
 qed

lemma *lproject-to-lselect*: $lproject\ A\ w = lselect\ (lset\ A\ w)\ w$
unfolding *lfilter-conv-lnth* *lset-def* **by** *rule*

lemma *lproject-llength[simp]*: $llength\ (lproject\ A\ w) = esize\ (lset\ A\ w)$
by (*induct rule: llist-induct*) (*auto*)

lemma *lproject-lfinite[simp]*: $lfinite\ (lproject\ A\ w) \longleftrightarrow finite\ (lset\ A\ w)$
using *lproject-llength esize-iff-infinite llength-eq-infty-conv-lfinite* **by** *metis*

lemma *lselect-restrict-indices[simp]*: $lselect\ \{i \in s.\ enat\ i < llength\ w\}\ w = lselect\ s\ w$
proof (*rule lnths-cong*)
show $w = w$ **by** *rule*
next
fix n
assume $1: enat\ n < llength\ w$
show $n \in \{i \in s.\ enat\ i < llength\ w\} \longleftrightarrow n \in s$ **using** 1 **by** *blast*
qed

lemma *lselect-llength*: $llength\ (lselect\ s\ w) = esize\ \{i \in s.\ enat\ i < llength\ w\}$
proof –
have $1: \bigwedge i.\ enat\ i < llength\ w \implies (w \parallel iterates\ Suc\ 0) \text{ ?! } i = (w \text{ ?! } i, i)$
by (*metis Suc-funpow enat.distinct(1) enat-ord-simps(4) llength-iterates lnth-iterates lnth-lzip monoid-add-class.add.right-neutral*)
have $2: \{i.\ enat\ i < llength\ w \wedge (w \parallel iterates\ Suc\ 0) \text{ ?! } i \in UNIV \times s\} = \{i \in s.\ enat\ i < llength\ w\}$ **using** 1 **by** *auto*
have $llength\ (lselect\ s\ w) = esize\ (lset\ (UNIV \times s)\ (w \parallel iterates\ Suc\ 0))$
unfolding *lselect-to-lproject* **by** *simp*
also have $\dots = esize\ \{i.\ enat\ i < llength\ w \wedge (w \parallel iterates\ Suc\ 0) \text{ ?! } i \in UNIV \times s\}$
unfolding *lset-def* **by** *simp*
also have $\dots = esize\ \{i \in s.\ enat\ i < llength\ w\}$ **unfolding** 2 **by** *rule*
finally show *?thesis* **by** *this*
qed

lemma *lselect-llength-le[simp]*: $llength\ (lselect\ s\ w) \leq esize\ s$
proof –
have $llength\ (lselect\ s\ w) = esize\ \{i \in s.\ enat\ i < llength\ w\}$
unfolding *lselect-llength* **by** *rule*
also have $\dots = esize\ (s \cap \{i.\ enat\ i < llength\ w\})$ **unfolding** *Collect-conj-eq*
by *simp*
also have $\dots \leq esize\ s$ **by** *blast*
finally show *?thesis* **by** *this*
qed

lemma *least-lselect-llength*:
assumes $\neg\ lnull\ (lselect\ s\ w)$
shows $enat\ (least\ s) < llength\ w$
proof –
have $0: llength\ (lselect\ s\ w) > 0$ **using** *assms* **by** *auto*
have $1: \bigwedge i.\ i \in s \implies least\ s \leq i$ **using** *Least-le 0* **by** *fast*

obtain i where $2: i \in s$ enat $i < \text{length } w$ using 0 unfolding lselect-llength
by auto
have enat $(\text{least } s) \leq \text{enat } i$ using 1 $2(1)$ by auto
also have $\dots < \text{length } w$ using $2(2)$ by this
finally show enat $(\text{least } s) < \text{length } w$ by this
qed
lemma $\text{lselect-lnull}: \text{lnull } (\text{lselect } s \ w) \longleftrightarrow (\forall i \in s. \text{enat } i \geq \text{length } w)$
unfolding $\text{llength-eq-0}[\text{symmetric}]$ lselect-llength by auto

lemma $\text{lselect-discard-start}$:
assumes $\bigwedge i. i \in s \implies k \leq i$
shows $\text{lselect } \{i. k + i \in s\} (\text{ldropn } k \ w) = \text{lselect } s \ w$
proof –
have $1: \text{lselect } s (\text{ltake } (\text{enat } k) \ w) = \langle \rangle$
using assms by $(\text{fastforce simp add: lselect-lnull min-le-iff-disj})$
have $\text{lselect } \{m. k + m \in s\} (\text{ldropn } k \ w) =$
 $\text{lselect } s (\text{ltake } (\text{enat } k) \ w) \ \$ \ \text{lselect } \{m. k + m \in s\} (\text{ldropn } k \ w)$ unfolding
 1 by simp
also have $\dots = \text{lselect } s \ w$ using lnths-split by rule
finally show $?thesis$ by this
qed

lemma $\text{lselect-discard-end}$:
assumes $\bigwedge i. i \in s \implies i < k$
shows $\text{lselect } s (\text{ltake } (\text{enat } k) \ w) = \text{lselect } s \ w$
proof –
have $1: \text{lselect } \{m. k + m \in s\} (\text{ldropn } k \ w) = \langle \rangle$
using assms by $(\text{fastforce simp add: lselect-lnull min-le-iff-disj})$
have $\text{lselect } s (\text{ltake } (\text{enat } k) \ w) =$
 $\text{lselect } s (\text{ltake } (\text{enat } k) \ w) \ \$ \ \text{lselect } \{m. k + m \in s\} (\text{ldropn } k \ w)$ unfolding
 1 by simp
also have $\dots = \text{lselect } s \ w$ using lnths-split by rule
finally show $?thesis$ by this
qed

lemma lselect-least :
assumes $\neg \text{lnull } (\text{lselect } s \ w)$
shows $\text{lselect } s \ w = w \ ?! \ \text{least } s \ \% \ \text{lselect } (s - \{\text{least } s\}) \ w$
proof –
have $0: s \neq \{\}$ using assms by auto
have $1: \text{least } s \in s$ using $\text{LeastI } 0$ by fast
have $2: \bigwedge i. i \in s \implies \text{least } s \leq i$ using $\text{Least-le } 0$ by fast
have $3: \bigwedge i. i \in s - \{\text{least } s\} \implies \text{Suc } (\text{least } s) \leq i$ using $\text{least-unique } 2$ by
 force
have $4: \text{insert } (\text{least } s) (s - \{\text{least } s\}) = s$ using 1 by auto
have $5: \text{enat } (\text{least } s) < \text{length } w$ using $\text{least-lselect-llength assms}$ by this
have $6: \text{lselect } (s - \{\text{least } s\}) (\text{ltake } (\text{enat } (\text{least } s)) \ w) = \langle \rangle$
by $(\text{rule, auto simp: lselect-llength dest: least-not-less})$
have $7: \text{lselect } \{i. \text{Suc } (\text{least } s) + i \in s - \{\text{least } s\}\} (\text{ldropn } (\text{Suc } (\text{least } s))$
 $w) =$

$lselect (s - \{least\ s\})\ w$ **using** *lselect-discard-start 3* **by this**
have $lselect\ s\ w = lselect (insert (least\ s) (s - \{least\ s\}))\ w$ **unfolding 4** **by**
simp
also have $\dots = lselect (s - \{least\ s\}) (ltake (enat (least\ s))\ w)$ $\$ <w\ ?!\ least\ s >\ \$$
 $lselect \{m.\ Suc (least\ s) + m \in s - \{least\ s\}\} (ldropn (Suc (least\ s))\ w)$
unfolding *lnths-insert[OF 5]* **by** *simp*
also have $\dots = <w\ ?!\ least\ s >\ \$$
 $lselect \{m.\ Suc (least\ s) + m \in s - \{least\ s\}\} (ldropn (Suc (least\ s))\ w)$
unfolding 6 **by** *simp*
also have $\dots = w\ ?!\ (least\ s)\ \% lselect (s - \{least\ s\})\ w$ **unfolding 7** **by**
simp
finally show *?thesis* **by this**
qed

lemma *lselect-lnth[simp]*:
assumes $enat\ i < llength (lselect\ s\ w)$
shows $lselect\ s\ w\ ?!\ i = w\ ?!\ nth\ least\ s\ i$
using *assms*
proof (*induct i arbitrary: s*)
case 0
have $1: \neg\ lnull (lselect\ s\ w)$ **using 0** **by** *auto*
show *?case* **using** *lselect-least 1* **by** *force*
next
case (*Suc i*)
have $1: \neg\ lnull (lselect\ s\ w)$ **using** *Suc(2)* **by** *auto*
have $2: lselect\ s\ w = w\ ?!\ least\ s\ \% lselect (s - \{least\ s\})\ w$ **using** *lselect-least 1* **by this**
have $3: llength (lselect\ s\ w) = eSuc (llength (lselect (s - \{least\ s\})\ w))$ **using 2** **by** *simp*
have $4: enat\ i < llength (lselect (s - \{least\ s\})\ w)$ **using 3** *Suc(2)* **by** *simp*
have $lselect\ s\ w\ ?!\ Suc\ i = (w\ ?!\ least\ s\ \% lselect (s - \{least\ s\})\ w)\ ?!\ Suc\ i$
using 2 **by** *simp*
also have $\dots = lselect (s - \{least\ s\})\ w\ ?!\ i$ **by** *simp*
also have $\dots = w\ ?!\ nth\ least\ (s - \{least\ s\})\ i$ **using** *Suc(1)* **4** **by** *simp*
also have $\dots = w\ ?!\ nth\ least\ s (Suc\ i)$ **by** *simp*
finally show *?case* **by this**
qed

lemma *lproject-lnth[simp]*:
assumes $enat\ i < llength (lproject\ A\ w)$
shows $lproject\ A\ w\ ?!\ i = w\ ?!\ nth\ least (lset\ A\ w)\ i$
using *assms* **unfolding** *lproject-to-lselect* **by** *simp*

lemma *lproject-ltake[simp]*:
assumes $enat\ k \leq llength (lproject\ A\ w)$
shows $lproject\ A (ltake (enat (nth\ least (lift (lset\ A\ w))\ k))\ w) =$
 $ltake (enat\ k) (lproject\ A\ w)$
proof
have $llength (lproject\ A (ltake (enat (nth\ least (lift (lset\ A\ w))\ k))\ w)) =$

$enat (card (liset A w \cap \{.. < nth-least (lift (liset A w)) k\}))$ **by simp**
also have $\dots = enat (card \{i \in liset A w. i < nth-least (lift (liset A w)) k\})$
unfolding lessThan-def Collect-conj-eq by simp
also have $\dots = enat k$ **using** *assms* **by simp**
also have $\dots = llength (ltake (enat k) (lproject A w))$ **using** *min-absorb1*
assms **by force**
finally show $llength (lproject A (ltake (enat (nth-least (lift (liset A w)) k)) w)) =$
 $llength (ltake (enat k) (lproject A w))$ **by this**
next
fix i
assume 1: $enat i < llength (lproject A (ltake (enat (nth-least (lift (liset A w)) k)) w))$
assume 2: $enat i < llength (ltake (enat k) (lproject A w))$
obtain k' **where** 3: $k = Suc k'$ **using** 2 *nat.exhaust* **by auto**
have 4: $enat k' < llength (lproject A w)$ **using** *assms* 3 **by simp**
have 5: $i \leq k'$ **using** 2 3 **by simp**
have 6: $nth-least (lift (liset A w)) k = Suc (nth-least (liset A w) k')$
using 3 4 **by** (*simp del: nth-least.simps*)
have 7: $nth-least (liset A w) i < Suc (nth-least (liset A w) k')$
proof –
have $nth-least (liset A w) i \leq nth-least (liset A w) k'$ **using** 4 5 **by simp**
also have $\dots < Suc (nth-least (liset A w) k')$ **by simp**
finally show *?thesis* **by this**
qed
have 8: $nth-least (liset A w \cap \{.. < Suc (nth-least (liset A w) k')\}) i =$
 $nth-least (liset A w) i$
proof (*rule nth-least-eq*)
show $enat i < esize (liset A w \cap \{.. < Suc (nth-least (liset A w) k')\})$ **using**
1 6 **by simp**
have $enat i \leq enat k'$ **using** 5 **by simp**
also have $enat k' < esize (liset A w)$ **using** 4 **by simp**
finally show $enat i < esize (liset A w)$ **by this**
next
fix j
assume 1: $j \leq nth-least (liset A w) i$
show $j \in liset A w \cap \{.. < Suc (nth-least (liset A w) k')\} \longleftrightarrow j \in liset A w$
using 1 7 **by simp**
qed
have $lproject A (ltake (enat (nth-least (lift (liset A w)) k)) w) \text{?! } i =$
 $ltake (enat (Suc (nth-least (liset A w) k'))) w \text{?!}$
 $nth-least (liset A w \cap \{.. < Suc (nth-least (liset A w) k')\}) i$
using 1 6 **by simp**
also have $\dots = ltake (enat (Suc (nth-least (liset A w) k'))) w \text{?! } nth-least$
(*liset A w*) i
using 8 **by simp**
also have $\dots = w \text{?! } nth-least (liset A w) i$ **using** 7 **by simp**
also have $\dots = lproject A w \text{?! } i$ **using** 2 **by simp**
also have $\dots = ltake (enat k) (lproject A w) \text{?! } i$ **using** 2 **by simp**

finally show $lproject\ A\ (ltake\ (enat\ (nth\ least\ (lift\ (lset\ A\ w))\ k))\ w)\ ?!\ i =$
 $ltake\ (enat\ k)\ (lproject\ A\ w)\ ?!\ i$ **by this**

qed

lemma *llength-less-llength-lselect-less*:

$enat\ i < esize\ s \wedge enat\ (nth\ least\ s\ i) < llength\ w \iff enat\ i < llength\ (lselect\ s\ w)$

using *nth-least-less-esize-less unfolding lselect-llength* **by this**

lemma *lselect-lselect''*:

assumes $\bigwedge i. i \in s \implies enat\ i < llength\ w$

assumes $\bigwedge i. i \in t \implies enat\ i < llength\ (lselect\ s\ w)$

shows $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\ least\ s\ 't)\ w$

proof

note *lselect-llength[simp]*

have 1: $\bigwedge i. i \in nth\ least\ s\ 't \implies enat\ i < llength\ w$ **using** *assms* **by auto**

have 2: $t \subseteq \{i. enat\ i < esize\ s\}$

using *assms(2) lselect-llength-le less-le-trans* **by blast**

have 3: *inj-on* $(nth\ least\ s)\ t$ **using** *subset-inj-on nth-least.inj-on 2* **by this**

have $llength\ (lselect\ t\ (lselect\ s\ w)) = esize\ t$ **using** *assms(2)* **by simp**

also have $\dots = esize\ (nth\ least\ s\ 't)$ **using** 3 **by auto**

also have $\dots = llength\ (lselect\ (nth\ least\ s\ 't)\ w)$ **using** 1 **by simp**

finally show $llength\ (lselect\ t\ (lselect\ s\ w)) = llength\ (lselect\ (nth\ least\ s\ 't)\ w)$

w)

by this

next

fix *i*

assume 1: $enat\ i < llength\ (lselect\ t\ (lselect\ s\ w))$

assume 2: $enat\ i < llength\ (lselect\ (nth\ least\ s\ 't)\ w)$

have 3: $enat\ i < esize\ t$ **using** *less-le-trans 1 lselect-llength-le* **by this**

have 4: $\bigwedge i. i \in t \implies enat\ i < esize\ s$

using *assms(2) lselect-llength-le less-le-trans* **by blast**

have $lselect\ t\ (lselect\ s\ w)\ ?!\ i = lselect\ s\ w\ ?!\ nth\ least\ t\ i$ **using** 1 **by simp**

also have $\dots = w\ ?!\ nth\ least\ s\ (nth\ least\ t\ i)$ **using** *assms(2) 3* **by simp**

also have $\dots = w\ ?!\ nth\ least\ (nth\ least\ s\ 't)\ i$ **using** 3 4 **by simp**

also have $\dots = lselect\ (nth\ least\ s\ 't)\ w\ ?!\ i$ **using** 2 **by simp**

finally show $lselect\ t\ (lselect\ s\ w)\ ?!\ i = lselect\ (nth\ least\ s\ 't)\ w\ ?!\ i$ **by this**

qed

lemma *lselect-lselect'[simp]*:

assumes $\bigwedge i. i \in t \implies enat\ i < esize\ s$

shows $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\ least\ s\ 't)\ w$

proof –

have 1: $nth\ least\ \{i \in s. enat\ i < llength\ w\}\ ' \{i \in t. enat\ i < llength\ (lselect\ s\ w)\} =$

$\{i \in nth\ least\ s\ 't. enat\ i < llength\ w\}$

unfolding *Compr-image-eq*

proof (*rule image-cong*)

show $\{i \in t. enat\ i < llength\ (lselect\ s\ w)\} = \{i \in t. enat\ (nth\ least\ s\ i) <$

```

length w}
  using length-less-length-lselect-less assms by blast
next
fix i
assume 1: i ∈ {i ∈ t. enat (nth-least s i) < length w}
have 2: enat i < esize {i ∈ s. enat i < length w}
  using nth-least-less-esize-less assms 1 by blast
show nth-least {i ∈ s. enat i < length w} i = nth-least s i using 2 by
simp
qed
have lselect t (lselect s w) =
  lselect {i ∈ t. enat i < length (lselect s w)} (lselect {i ∈ s. enat i < length
w} w)
  by simp
also have ... = lselect (nth-least {i ∈ s. enat i < length w} ‘
  {i ∈ t. enat i < length (lselect s w)}’) w
  by (rule lselect-lselect'', auto simp: lselect-length)
also have ... = lselect {i ∈ nth-least s ‘ t. enat i < length w} w unfolding
1 by rule
also have ... = lselect (nth-least s ‘ t) w by simp
finally show ?thesis by this
qed

lemma lselect-lselect:
  lselect t (lselect s w) = lselect (nth-least s ‘ {i ∈ t. enat i < esize s}) w
proof -
  have lselect t (lselect s w) = lselect {i ∈ t. enat i < length (lselect s w)}
(lselect s w)
  by simp
  also have ... = lselect (nth-least s ‘ {i ∈ t. enat i < length (lselect s w)}’) w
  using lselect-length-le less-le-trans by (blast intro: lselect-lselect')
  also have ... = lselect (nth-least s ‘ {i ∈ t. enat i < esize s}) w
  using length-less-length-lselect-less by (auto intro!: lnths-cong)
  finally show ?thesis by this
qed

lemma lselect-lproject':
  assumes  $\bigwedge i. i \in s \implies \text{enat } i < \text{length } w$ 
  shows lproject A (lselect s w) = lselect (s  $\cap$  lset A w) w
proof -
  have 1:  $\bigwedge i. i \in \text{lset } A \text{ (lselect s w)} \implies \text{enat } i < \text{esize } s$  using less-le-trans
by force
  have 2: {i ∈ lset A (lselect s w). enat i < esize s} = lset A (lselect s w)
  using 1 by auto
  have 3: nth-least s ‘ lset A (lselect s w) = s  $\cap$  lset A w
proof safe
  fix k
  assume 4: k ∈ lset A (lselect s w)
  show nth-least s k ∈ s using 1 4 by simp

```

show $\text{nth-least } s \ k \in \text{liset } A \ w$
using $\text{llength-less-llength-lselect-less } 4$ **unfolding** liset-def **by** auto
next
fix k
assume $1: k \in s \ k \in \text{liset } A \ w$
have $2: \text{nth-least } s \ (\text{card } \{i \in s. i < k\}) = k$ **using** $\text{nth-least-card } 1(1)$ **by**
this
have $3: \text{enat } (\text{card } \{i \in s. i < k\}) < \text{llength } (\text{lselect } s \ w)$
unfolding lselect-llength **using** $\text{assms } 1(1)$ **by** simp
show $k \in \text{nth-least } s \ ' \ \text{liset } A \ (\text{lselect } s \ w)$
proof
show $k = \text{nth-least } s \ (\text{card } \{i \in s. i < k\})$ **using** 2 **by** simp
show $\text{card } \{i \in s. i < k\} \in \text{liset } A \ (\text{lselect } s \ w)$ **using** $1(2) \ 2 \ 3$ **by** fastforce
qed
qed
have $\text{lproject } A \ (\text{lselect } s \ w) = \text{lselect } (\text{liset } A \ (\text{lselect } s \ w)) \ (\text{lselect } s \ w)$
unfolding $\text{lproject-to-lselect}$ **by** rule
also have $\dots = \text{lselect } (\text{nth-least } s \ ' \ \{i \in \text{liset } A \ (\text{lselect } s \ w). \text{enat } i < \text{esize}$
 $s\}) \ w$
unfolding lselect-lselect **by** rule
also have $\dots = \text{lselect } (\text{nth-least } s \ ' \ \text{liset } A \ (\text{lselect } s \ w)) \ w$ **unfolding** 2 **by**
rule
also have $\dots = \text{lselect } (s \cap \text{liset } A \ w) \ w$ **unfolding** 3 **by** rule
finally show *?thesis* **by** *this*
qed

lemma $\text{lselect-lproject}[simp]: \text{lproject } A \ (\text{lselect } s \ w) = \text{lselect } (s \cap \text{liset } A \ w) \ w$
proof –
have $1: \{i \in s. \text{enat } i < \text{llength } w\} \cap \text{liset } A \ w = s \cap \text{liset } A \ w$ **by** auto
have $\text{lproject } A \ (\text{lselect } s \ w) = \text{lproject } A \ (\text{lselect } \{i \in s. \text{enat } i < \text{llength } w\}$
 $w)$ **by** simp
also have $\dots = \text{lselect } (\{i \in s. \text{enat } i < \text{llength } w\} \cap \text{liset } A \ w) \ w$
by $(\text{rule } \text{lselect-lproject}', \text{simp})$
also have $\dots = \text{lselect } (s \cap \text{liset } A \ w) \ w$ **unfolding** 1 **by** rule
finally show *?thesis* **by** *this*
qed

lemma $\text{lproject-lselect-subset}[simp]:$
assumes $\text{liset } A \ w \subseteq s$
shows $\text{lproject } A \ (\text{lselect } s \ w) = \text{lproject } A \ w$
proof –
have $1: s \cap \text{liset } A \ w = \text{liset } A \ w$ **using** assms **by** auto
have $\text{lproject } A \ (\text{lselect } s \ w) = \text{lselect } (s \cap \text{liset } A \ w) \ w$ **by** simp
also have $\dots = \text{lselect } (\text{liset } A \ w) \ w$ **unfolding** 1 **by** rule
also have $\dots = \text{lproject } A \ w$ **unfolding** $\text{lproject-to-lselect}$ **by** rule
finally show *?thesis* **by** *this*
qed

lemma $\text{lselect-prefix}[intro]:$

```

    assumes  $u \leq v$ 
    shows  $\text{lselect } s \ u \leq \text{lselect } s \ v$ 
  proof (cases  $\text{lfinite } u$ )
    case False
    show ?thesis using lprefix-infinite assms False by auto
  next
    case True
    obtain  $k$  where  $1: \text{llength } u = \text{enat } k$  using True length-list-of by metis
    obtain  $w$  where  $2: v = u \ \$ \ w$  using lprefix-conv-lappend assms by auto
    have  $\text{lselect } s \ u \leq \text{lselect } s \ u \ \$ \ \text{lselect } \{n. n + k \in s\} \ w$  by simp
    also have  $\dots = \text{lselect } s \ (u \ \$ \ w)$  using lnths-lappend-lfinite[symmetric]  $1$  by
  this
    also have  $\dots = \text{lselect } s \ v$  unfolding  $2$  by rule
    finally show ?thesis by this
  qed
lemma lproject-prefix[intro]:
  assumes  $u \leq v$ 
  shows  $\text{lproject } A \ u \leq \text{lproject } A \ v$ 
  using lprefix-lfilterI assms by auto

lemma lproject-prefix-limit[intro?]:
  assumes  $\bigwedge v. v \leq w \implies \text{lfinite } v \implies \text{lproject } A \ v \leq x$ 
  shows  $\text{lproject } A \ w \leq x$ 
  proof -
    have  $1: \text{ccpo.admissible } \text{lSup } \text{lprefix } (\lambda v. \text{lproject } A \ v \leq x)$  by simp
    show ?thesis using llist-lift-admissible 1 assms(1) by this
  qed
lemma lproject-prefix-limit':
  assumes  $\bigwedge k. \exists v. v \leq w \wedge \text{enat } k < \text{llength } v \wedge \text{lproject } A \ v \leq x$ 
  shows  $\text{lproject } A \ w \leq x$ 
  proof (rule lproject-prefix-limit)
    fix  $u$ 
    assume  $1: u \leq w \ \text{lfinite } u$ 
    obtain  $k$  where  $2: \text{llength } u = \text{enat } k$  using  $1(2)$  by (metis length-list-of)
    obtain  $v$  where  $3: v \leq w \ \text{llength } u < \text{llength } v \ \text{lproject } A \ v \leq x$ 
      unfolding  $2$  using assms(1) by auto
    have  $4: \text{llength } u \leq \text{llength } v$  using  $3(2)$  by simp
    have  $5: u \leq v$  using prefix-subsume 1(1) 3(1) 4 by this
    have  $\text{lproject } A \ u \leq \text{lproject } A \ v$  using  $5$  by rule
    also have  $\dots \leq x$  using  $3(3)$  by this
    finally show  $\text{lproject } A \ u \leq x$  by this
  qed
end

```

15 Prefixes on Coinductive Lists

```

theory LList-Prefixes
imports

```

Word-Prefixes
../Extensions/Coinductive-List-Extensions
begin

lemma *unfold-stream-siterate-smap*: $\text{unfold-stream } f \ g = \text{smap } f \circ \text{siterate } g$
by (*rule*, *coinduction*, *auto*) (*metis unfold-stream-eq-SCons*)⁺

lemma *lappend-stream-of-llist*:
assumes *lfinite* *u*
shows $\text{stream-of-llist } (u \ \$ \ v) = \text{list-of } u \ @- \ \text{stream-of-llist } v$
using *assms* **unfolding** *stream-of-llist-def* **by** *induct* *auto*

lemma *llist-of-inf-llist-prefix*[*intro*]: $u \leq_{FI} v \implies \text{llist-of } u \leq \text{llist-of-stream } v$
by (*metis lappend-llist-of-stream-conv-shift le-llist-conv-lprefix lprefix-lappend prefix-fininfE*)

lemma *prefix-llist-of-inf-llist*[*intro*]: $\text{lfinite } u \implies u \leq v \implies \text{list-of } u \leq_{FI} \text{stream-of-llist } v$
by (*metis lappend-stream-of-llist le-llist-conv-lprefix lprefix-conv-lappend prefix-fininfI*)

lemma *lproject-prefix-limit-chain*:
assumes $\text{chain } w \ \wedge \ k. \ \text{lproject } A \ (\text{llist-of } (w \ k)) \leq x$
shows $\text{lproject } A \ (\text{llist-of-stream } (\text{limit } w)) \leq x$
proof (*rule lproject-prefix-limit'*)
fix *k*
obtain *l* **where** $1: k < \text{length } (w \ l)$ **using** *assms*(1) **by** *rule*
show $\exists v \leq \text{llist-of-stream } (\text{limit } w). \ \text{enat } k < \text{llength } v \ \wedge \ \text{lproject } A \ v \leq x$
proof (*intro exI conjI*)
show $\text{llist-of } (w \ l) \leq \text{llist-of-stream } (\text{limit } w)$
using *llist-of-inf-llist-prefix chain-prefix-limit assms*(1) **by** *this*
show $\text{enat } k < \text{llength } (\text{llist-of } (w \ l))$ **using** 1 **by** *simp*
show $\text{lproject } A \ (\text{llist-of } (w \ l)) \leq x$ **using** *assms*(2) **by** *this*
qed

qed

lemma *lproject-eq-limit-chain*:
assumes $\text{chain } u \ \text{chain } v \ \wedge \ k. \ \text{project } A \ (u \ k) = \text{project } A \ (v \ k)$
shows $\text{lproject } A \ (\text{llist-of-stream } (\text{limit } u)) = \text{lproject } A \ (\text{llist-of-stream } (\text{limit } v))$
proof (*rule antisym*)
show $\text{lproject } A \ (\text{llist-of-stream } (\text{limit } u)) \leq \text{lproject } A \ (\text{llist-of-stream } (\text{limit } v))$
proof (*rule lproject-prefix-limit-chain*)
show $\text{chain } u$ **using** *assms*(1) **by** *this*
next
fix *k*
have $\text{lproject } A \ (\text{llist-of } (u \ k)) = \text{lproject } A \ (\text{llist-of } (v \ k))$ **using** *assms*(3)
by *simp*
also **have** $\dots \leq \text{lproject } A \ (\text{llist-of-stream } (\text{limit } v))$ **using** *chain-prefix-limit assms*(2) **by** *blast*


```

    finally show lproject A (llist-of (u k)) ≤ lproject A (llist-of-stream (limit v))
  by this
  qed
  show lproject A (llist-of-stream (limit v)) ≤ lproject A (llist-of-stream (limit
u))
  proof (rule lproject-prefix-limit-chain)
    show chain v using assms(2) by this
  next
    fix k
    have lproject A (llist-of (v k)) = lproject A (llist-of (u k)) using assms(3)
  by simp
    also have ... ≤ lproject A (llist-of-stream (limit u)) using chain-prefix-limit
assms(1) by blast
    finally show lproject A (llist-of (v k)) ≤ lproject A (llist-of-stream (limit u))
  by this
  qed
  qed
end

```

16 Stuttering

theory *Stuttering*

imports

Stuttering-Equivalence.StutterEquivalence

LList-Prefixes

begin

function *nth-least-ext* :: *nat set* ⇒ *nat* ⇒ *nat*

where

enat k < esize A ⇒ *nth-least-ext A k* = *nth-least A k* |

enat k ≥ esize A ⇒ *nth-least-ext A k* = *Suc (Max A + (k - card A))*

by *force+* **termination** **by** *lexicographic-order*

lemma *nth-least-ext-strict-mono*:

assumes *k < l*

shows *nth-least-ext s k < nth-least-ext s l*

proof (*cases enat l < esize s*)

case *True*

have *1: enat k < esize s* **using** *assms True* **by** (*metis enat-ord-simps(2)*)

less-trans)

show *?thesis* **using** *nth-least-strict-mono assms True 1* **by** *simp*

next

case *False*

have *1: finite s* **using** *False esize-infinite-enat* **by** *auto*

have *2: enat l ≥ esize s* **using** *False* **by** *simp*

have *3: l ≥ card s* **using** *1 2* **by** *simp*

show *?thesis*

proof (*cases enat k < esize s*)

```

case True
have 4: s ≠ {} using True by auto
have nth-least-ext s k = nth-least s k using True by simp
also have ... ≤ Max s using nth-least-le-Max 1 4 True by this
also have ... < Suc (Max s) by auto
also have ... ≤ Suc (Max s + (l - card s)) by auto
also have Suc (Max s + (l - card s)) = nth-least-ext s l using 2 by simp
finally show ?thesis by this
next
case False
have 4: enat k ≥ esize s using False by simp
have 5: k ≥ card s using 1 4 by simp
have nth-least-ext s k = Suc (Max s + (k - card s)) using 4 by simp
also have ... < Suc (Max s + (l - card s)) using assms 5 by simp
also have ... = nth-least-ext s l using 2 by simp
finally show ?thesis by this
qed
qed

```

definition *stutter-selection* :: nat set ⇒ 'a llist ⇒ bool

```

where stutter-selection s w ≡ 0 ∈ s ∧
  (∀ k i. enat i < llength w → enat (Suc k) < esize s →
    nth-least s k < i → i < nth-least s (Suc k) → w ?! i = w ?! nth-least s k) ∧
  (∀ i. enat i < llength w → finite s → Max s < i → w ?! i = w ?! Max s)

```

lemma *stutter-selectionI*[intro]:

```

assumes 0 ∈ s
assumes ∧ k i. enat i < llength w ⇒ enat (Suc k) < esize s ⇒
  nth-least s k < i ⇒ i < nth-least s (Suc k) ⇒ w ?! i = w ?! nth-least s k
assumes ∧ i. enat i < llength w ⇒ finite s ⇒ Max s < i ⇒ w ?! i = w

```

?! Max s

```

shows stutter-selection s w
using assms unfolding stutter-selection-def by auto

```

lemma *stutter-selectionD-0*[dest]:

```

assumes stutter-selection s w
shows 0 ∈ s
using assms unfolding stutter-selection-def by auto

```

lemma *stutter-selectionD-inside*[dest]:

```

assumes stutter-selection s w
assumes enat i < llength w enat (Suc k) < esize s
assumes nth-least s k < i i < nth-least s (Suc k)
shows w ?! i = w ?! nth-least s k
using assms unfolding stutter-selection-def by auto

```

lemma *stutter-selectionD-infinite*[dest]:

```

assumes stutter-selection s w
assumes enat i < llength w finite s Max s < i
shows w ?! i = w ?! Max s
using assms unfolding stutter-selection-def by auto

```

```

lemma stutter-selection-stutter-sampler[intro]:
  assumes linfinite w stutter-selection s w
  shows stutter-sampler (nth-least-ext s) (lnth w)
unfolding stutter-sampler-def
proof safe
  show nth-least-ext s 0 = 0 using assms(2) by (cases enat 0 < esize s, auto)
  show strict-mono (nth-least-ext s) using strict-monoI nth-least-ext-strict-mono
by blast
next
fix k i
assume 1: nth-least-ext s k < i i < nth-least-ext s (Suc k)
show w ?! i = w ?! nth-least-ext s k
proof (cases enat (Suc k) esize s rule: linorder-cases)
  case less
  have w ?! i = w ?! nth-least s k
  proof (rule stutter-selectionD-inside)
    show stutter-selection s w using assms(2) by this
    show enat i < llength w using assms(1) by auto
    show enat (Suc k) < esize s using less by this
    show nth-least s k < i using 1(1) less by auto
    show i < nth-least s (Suc k) using 1(2) less by simp
  qed
  also have w ?! nth-least s k = w ?! nth-least-ext s k using less by auto
  finally show ?thesis by this
next
case equal
  have 2: enat k < esize s using equal by (metis enat-ord-simps(2) lessI)
  have 3: finite s using equal by (metis esize-infinite-enat less-irrefl)
  have 4:  $\bigwedge i. i > \text{Max } s \implies w ?! i = w ?! \text{Max } s$  using assms 3 by auto
  have 5:  $k = \text{card } s - 1$  using equal 3 by (metis diff-Suc-1 enat.inject
  esize-set.simps(1))
  have Max s = nth-least s (card s - 1) using nth-least-Max 3 assms(2) by
  force
  also have ... = nth-least s k unfolding 5 by rule
  also have ... = nth-least-ext s k using 2 by simp
  finally have 6: Max s = nth-least-ext s k by this
  have w ?! i = w ?! Max s using 1(1) 4 6 by auto
  also have ... = w ?! nth-least-ext s k unfolding 6 by rule
  finally show ?thesis by this
next
case greater
  have 2: enat k  $\geq$  esize s using greater by (metis Suc-ile-eq not-le)
  have 3: finite s using greater by (metis esize-infinite-enat less-asymp)
  have 4:  $\bigwedge i. i > \text{Max } s \implies w ?! i = w ?! \text{Max } s$  using assms 3 by auto
  have w ?! i = w ?! Max s using 1(1) 2 4 by auto
  also have ... = w ?! Suc (Max s + (k - card s)) using 4 by simp
  also have ... = w ?! nth-least-ext s k using 2 by simp
  finally show ?thesis by this
qed

```

qed

```
lemma stutter-equivI-selection[intro]:
  assumes linfinite u linfinite v
  assumes stutter-selection s u stutter-selection t v
  assumes lselect s u = lselect t v
  shows lnth u ≈ lnth v
proof (rule stutter-equivI)
  have 1: llength (lselect s u) = llength (lselect t v) unfolding assms(5) by rule
  have 2: esize s = esize t using 1 assms(1, 2) unfolding lselect-llength by
simp
  show stutter-sampler (nth-least-ext s) (lnth u) using assms(1, 3) by rule
  show stutter-sampler (nth-least-ext t) (lnth v) using assms(2, 4) by rule
  show lnth u ∘ nth-least-ext s = lnth v ∘ nth-least-ext t
proof (rule ext, unfold comp-apply)
  fix i
  show u ?! nth-least-ext s i = v ?! nth-least-ext t i
proof (cases enat i < esize s)
  case True
  have 3: enat i < llength (lselect s u) enat i < llength (lselect t v)
    using assms(1, 2) 2 True unfolding lselect-llength by auto
  have u ?! nth-least-ext s i = u ?! nth-least s i using True by simp
  also have ... = lselect s u ?! i using 3(1) by simp
  also have ... = lselect t v ?! i unfolding assms(5) by rule
  also have ... = v ?! nth-least t i using 3(2) by simp
  also have ... = v ?! nth-least-ext t i using True unfolding 2 by simp
  finally show u ?! nth-least-ext s i = v ?! nth-least-ext t i by this
next
  case False
  have 3: s ≠ {} t ≠ {} using assms(3, 4) by auto
  have 4: finite s finite t using esize-infinite-enat 2 False bymetis+
  have 5: ∧ i. i > Max s ⇒ u ?! i = u ?! Max s using assms(1, 3) 4(1)
by auto
  have 6: ∧ i. i > Max t ⇒ v ?! i = v ?! Max t using assms(2, 4) 4(2)
by auto
  have 7: esize s = enat (card s) esize t = enat (card t) using 4 by auto
  have 8: card s ≠ 0 card t ≠ 0 using 3 4 by auto
  have 9: enat (card s - 1) < llength (lselect s u)
    using assms(1) 7(1) 8(1) unfolding lselect-llength by simp
  have 10: enat (card t - 1) < llength (lselect t v)
    using assms(2) 7(2) 8(2) unfolding lselect-llength by simp
  have u ?! nth-least-ext s i = u ?! Suc (Max s + (i - card s)) using False
by simp
  also have ... = u ?! Max s using 5 by simp
  also have ... = u ?! nth-least s (card s - 1) using nth-least-Max 4(1) 3(1)
by force
  also have ... = lselect s u ?! (card s - 1) using lselect-lnth 9 by simp
  also have ... = lselect s u ?! (card t - 1) using 2 4 by simp
  also have ... = lselect t v ?! (card t - 1) unfolding assms(5) by rule
```

also have $\dots = v$?! *nth-least* t (*card* $t - 1$) **using** *lselect-lnth* 10 **by** *simp*
also have $\dots = v$?! *Max* t **using** *nth-least-Max* 4(2) 3(2) **by** *force*
also have $\dots = v$?! *Suc* (*Max* $t + (i - \text{card } t)$) **using** 6 **by** *simp*
also have $\dots = v$?! *nth-least-ext* t i **using** 2 *False* **by** *simp*
finally show ?thesis **by** *this*

qed

qed

qed

definition *stuttering-invariant* :: 'a word set \Rightarrow bool
where *stuttering-invariant* $A \equiv \forall u v. u \approx v \longrightarrow u \in A \longleftrightarrow v \in A$

lemma *stuttering-invariant-complement*[intro]:
assumes *stuttering-invariant* A
shows *stuttering-invariant* ($- A$)
using *assms* **unfolding** *stuttering-invariant-def* **by** *simp*

lemma *stutter-equiv-forw-subst*[trans]: $w_1 = w_2 \Longrightarrow w_2 \approx w_3 \Longrightarrow w_1 \approx w_3$ **by**
auto

lemma *stutter-sampler-build*:
assumes *stutter-sampler* f w
shows *stutter-sampler* ($0 \## (Suc \circ f)$) ($a \## w$)
unfolding *stutter-sampler-def*
proof *safe*
have 0: $f\ 0 = 0$ **using** *assms* **unfolding** *stutter-sampler-def* **by** *auto*
have 1: $f\ x < f\ y$ **if** $x < y$ **for** $x\ y$
using *assms* **that** **unfolding** *stutter-sampler-def* *strict-mono-def* **by** *auto*
have 2: $(0 \## (Suc \circ f))\ x < (0 \## (Suc \circ f))\ y$ **if** $x < y$ **for** $x\ y$
using 1 **that** **by** (*cases* x ; *cases* y) (*auto*)
have 3: $w\ n = w\ (f\ k)$ **if** $f\ k < n < f\ (Suc\ k)$ **for** $k\ n$
using *assms* **that** **unfolding** *stutter-sampler-def* **by** *auto*
show $(0 \## (Suc \circ f))\ 0 = 0$ **by** *simp*
show *strict-mono* $(0 \## (Suc \circ f))$ **using** 2 **by** *rule*
show $(a \## w)\ n = (a \## w)\ ((0 \## (Suc \circ f))\ k)$
if $(0 \## (Suc \circ f))\ k < n < (0 \## (Suc \circ f))\ (Suc\ k)$ **for** $k\ n$
using 0 3 **that** **by** (*cases* k ; *cases* n) (*force*)+

qed

lemma *stutter-extend-build*:

assumes $u \approx v$
shows $a \## u \approx a \## v$

proof $-$

obtain $f\ g$ **where** 1: *stutter-sampler* $f\ u$ *stutter-sampler* $g\ v$ $u \circ f = v \circ g$
using *stutter-equivE* *assms* **by** *this*

show ?thesis

proof (*intro* *stutter-equivI* *ext*)

show *stutter-sampler* $(0 \## (Suc \circ f))$ ($a \## u$) **using** *stutter-sampler-build*

1(1) **by** *this*

show *stutter-sampler* $(0 \## (Suc \circ g))$ ($a \## v$) **using** *stutter-sampler-build*

```

1(2) by this
  show (a ## u ◦ 0 ## (Suc ◦ f)) i = (a ## v ◦ 0 ## (Suc ◦ g)) i for i
  using fun-cong[OF 1(3)] by (cases i) (auto)
qed
qed
lemma stutter-extend-concat:
  assumes u ≈ v
  shows w ◦ u ≈ w ◦ v
  using stutter-extend-build assms by (induct w, force+)
lemma build-stutter: w 0 ## w ≈ w
proof (rule stutter-equivI)
  show stutter-sampler (Suc (0 := 0)) (w 0 ## w)
  unfolding stutter-sampler-def
  proof safe
    show (Suc (0 := 0)) 0 = 0 by simp
    show strict-mono (Suc (0 := 0)) by (rule strict-monoI, simp)
  next
  fix k n
  assume 1: (Suc (0 := 0)) k < n n < (Suc (0 := 0)) (Suc k)
  show (w 0 ## w) n = (w 0 ## w) ((Suc (0 := 0)) k) using 1 by (cases
n, auto)
  qed
  show stutter-sampler id w by rule
  show w 0 ## w ◦ (Suc (0 := 0)) = w ◦ id by auto
qed
lemma replicate-stutter: replicate n (v 0) ◦ v ≈ v
proof (induct n)
  case 0
  show ?case using stutter-equiv-refl by simp
next
  case (Suc n)
  have replicate (Suc n) (v 0) ◦ v = v 0 ## replicate n (v 0) ◦ v by simp
  also have ... = (replicate n (v 0) ◦ v) 0 ## replicate n (v 0) ◦ v by (cases
n, auto)
  also have ... ≈ replicate n (v 0) ◦ v using build-stutter by this
  also have ... ≈ v using Suc by this
  finally show ?case by this
qed

lemma replicate-stutter': u ◦ replicate n (v 0) ◦ v ≈ u ◦ v
  using stutter-extend-concat replicate-stutter by this

end

```

17 Interpreted Transition Systems and Traces

```

theory Transition-System-Interpreted-Traces
imports
  Transition-System-Traces

```

Basics/Stuttering

begin

locale *transition-system-interpreted-traces* =
 transition-system-interpreted *ex en int* +
 transition-system-traces *ex en ind*
 for *ex* :: 'action \Rightarrow 'state \Rightarrow 'state
 and *en* :: 'action \Rightarrow 'state \Rightarrow bool
 and *int* :: 'state \Rightarrow 'interpretation
 and *ind* :: 'action \Rightarrow 'action \Rightarrow bool
 +
 assumes *independence-invisible*: $a \in \text{visible} \Longrightarrow b \in \text{visible} \Longrightarrow \neg \text{ind } a \ b$

begin

lemma *eq-swap-lproject-visible*:

assumes $u =_S v$
 shows *lproject visible* (*l*list-of u) = *lproject visible* (*l*list-of v)
 using *assms independence-invisible* **by** (*induct*, *auto*)

lemma *eq-fin-lproject-visible*:

assumes $u =_F v$
 shows *lproject visible* (*l*list-of u) = *lproject visible* (*l*list-of v)
 using *assms eq-swap-lproject-visible* **by** (*induct*, *auto*)

lemma *le-fin-lproject-visible*:

assumes $u \preceq_F v$
 shows *lproject visible* (*l*list-of u) \leq *lproject visible* (*l*list-of v)

proof –

obtain w **where** $1: u @ w =_F v$ **using** *assms* **by** *rule*

have *lproject visible* (*l*list-of u) \leq

lproject visible (*l*list-of u) \$ *lproject visible* (*l*list-of w) **by** *auto*

also have $\dots = \text{lproject visible (l}u @ w\text{)}$ **using** *lappend-l*list-of-l

by *auto*

also have $\dots = \text{lproject visible (l}v\text{)}$ **using** *eq-fin-lproject-visible 1* **by**

this

finally show *?thesis* **by** *this*

qed

lemma *le-fininf-lproject-visible*:

assumes $u \preceq_{FI} v$
 shows *lproject visible* (*l*list-of u) \leq *lproject visible* (*l*list-of-stream v)

proof –

obtain w **where** $1: w \preceq_{FI} v \ u \preceq_F w$ **using** *assms* **by** *rule*

have *lproject visible* (*l*list-of u) \leq *lproject visible* (*l*list-of w)

using *le-fin-lproject-visible 1(2)* **by** *this*

also have $\dots \leq \text{lproject visible (l}v\text{)}$ **using** *1(1)* **by** *blast*

finally show *?thesis* **by** *this*

qed

lemma *le-inf-lproject-visible*:

assumes $u \preceq_I v$
 shows *lproject visible* (*l*list-of-stream u) \leq *lproject visible* (*l*list-of-stream v)

proof (*rule lproject-prefix-limit*)

```

fix w
assume 1:  $w \leq \text{lstream-of-list } u \text{ lfinite } w$ 
have 2:  $\text{list-of } w \leq_{FI} \text{stream-of-list } (\text{lstream-of-list } u)$  using 1 by blast
have 3:  $\text{list-of } w \preceq_{FI} v$  using assms 2 by auto
have  $\text{lproject visible } w = \text{lproject visible } (\text{lstream-of-list } (\text{list-of } w))$  using 1(2) by
simp
also have  $\dots \leq \text{lproject visible } (\text{lstream-of-list } v)$  using le-fininf-lproject-visible
3 by this
finally show  $\text{lproject visible } w \leq \text{lproject visible } (\text{lstream-of-list } v)$  by this
qed
lemma eq-inf-lproject-visible:
assumes  $u =_I v$ 
shows  $\text{lproject visible } (\text{lstream-of-list } u) = \text{lproject visible } (\text{lstream-of-list } v)$ 
using le-inf-lproject-visible assms by (metis antisym eq-infE)

lemma stutter-selection-lproject-visible:
assumes  $\text{run } u \text{ } p$ 
shows  $\text{stutter-selection } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u)))$ 
 $(\text{lstream-of-list } (\text{smap int } (p \#\# \text{trace } u \text{ } p)))$ 
proof
show  $0 \in \text{lift } (\text{liet visible } (\text{lstream-of-list } u))$  by auto
next
fix k i
assume 3:  $\text{enat } (\text{Suc } k) < \text{esize } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u)))$ 
assume 4:  $\text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) \text{ } k < i$ 
assume 5:  $i < \text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) (\text{Suc } k)$ 
have 6:  $\text{int } ((p \#\# \text{trace } u \text{ } p) \#\# \text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) k) =$ 
 $\text{int } ((p \#\# \text{trace } u \text{ } p) \#\# i)$ 
proof (rule execute-inf-word-invisible)
show  $\text{run } u \text{ } p$  using assms by this
show  $\text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) \text{ } k \leq i$  using 4 by auto
next
fix j
assume 6:  $\text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) \text{ } k \leq j$ 
assume 7:  $j < i$ 
have 8:  $\text{Suc } j \notin \text{lift } (\text{liet visible } (\text{lstream-of-list } u))$ 
proof (rule nth-least-not-contains)
show  $\text{enat } (\text{Suc } k) < \text{esize } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u)))$  using 3
by this
show  $\text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) \text{ } k < \text{Suc } j$  using 6
by auto
show  $\text{Suc } j < \text{nth-least } (\text{lift } (\text{liet visible } (\text{lstream-of-list } u))) (\text{Suc } k)$  using
5 7 by simp
qed
have 9:  $j \notin \text{liet visible } (\text{lstream-of-list } u)$  using 8 by auto
show  $u \#\# j \notin \text{visible}$  using 9 by auto
qed
show  $\text{lstream-of-list } (\text{smap int } (p \#\# \text{trace } u \text{ } p)) \text{ } ?! i = \text{lstream-of-list } (\text{smap}$ 

```



```

int (p ## trace u p) ?!
  nth-least (lift (lset visible (lstream u))) k
  using 6 by (metis lnth-list-of-stream snth-smap)
next
fix i
assume 1: finite (lift (lset visible (lstream u)))
assume 3: Max (lift (lset visible (lstream u))) < i
have 4: int ((p ## trace u p) !! Max (lift (lset visible (lstream u))))
=
  int ((p ## trace u p) !! i)
proof (rule execute-inf-word-invisible)
  show run u p using assms by this
  show Max (lift (lset visible (lstream u))) ≤ i using 3 by auto
next
fix j
assume 6: Max (lift (lset visible (lstream u))) ≤ j
assume 7: j < i
have 8: Suc j ∉ lift (lset visible (lstream u))
proof (rule ccontr)
  assume 9: ¬ Suc j ∉ lift (lset visible (lstream u))
  have 10: Suc j ∈ lift (lset visible (lstream u)) using 9 by simp
  have 11: Suc j ≤ Max (lift (lset visible (lstream u))) using Max-ge
1 10 by this
  show False using 6 11 by auto
qed
have 9: j ∉ lset visible (lstream u) using 8 by auto
show u !! j ∉ visible using 9 by auto
qed
show lstream (smap int (p ## trace u p)) ?! i = lstream (smap
int (p ## trace u p)) ?!
  Max (lift (lset visible (lstream u))) using 4 by (metis lnth-list-of-stream
snth-smap)
qed

```

lemma *execute-fin-visible*:

```

assumes path u q path v q u ≼FI w v ≼FI w
assumes project visible u = project visible v
shows int (target u q) = int (target v q)
proof -
  obtain w' where 1: u ≼F w' v ≼F w' using subsume-fin assms(3, 4) by
this
  obtain u' v' where 2: u @ u' =F w' v @ v' =F w' using 1 by blast
  have u @ u' =F w' using 2(1) by this
  also have ... =F v @ v' using 2(2) by blast
  finally have 3: u @ u' =F v @ v' by this
  obtain s1 s2 s3 where 4: u =F s1 @ s2 v =F s1 @ s3 Ind (set s2) (set s3)
  using levi-lemma 3 by this
  have 5: project visible (s1 @ s2) = project visible (s1 @ s3)
  using eq-fin-lproject-visible assms(5) 4(1, 2) by auto

```

have 6: *project visible* $s_2 = \text{project visible } s_3$ **using 5 by simp**
have 7: *set (project visible* $s_2) = \text{set (project visible } s_3)$ **using 6 by simp**
have 8: *set* $s_2 \cap \text{visible} = \text{set } s_3 \cap \text{visible}$ **using 7 by auto**
have 9: *set* $s_2 \subseteq \text{invisible} \vee \text{set } s_3 \subseteq \text{invisible}$ **using independence-invisible**
4(3) by auto
have 10: *set* $s_2 \subseteq \text{invisible}$ *set* $s_3 \subseteq \text{invisible}$ **using 8 9 by auto**
have 11: *path* s_2 (*target* s_1 q) **using eq-fin-word 4(1) assms(1) by auto**
have 12: *path* s_3 (*target* s_1 q) **using eq-fin-word 4(2) assms(2) by auto**
have int (fold ex u q) = int (fold ex (s1 @ s2) q) using eq-fin-execute assms(1)
4(1) by simp
also have ... = int (fold ex s1 q) using execute-fin-word-invisible 11 10(1)
by simp
also have ... = int (fold ex (s1 @ s3) q) using execute-fin-word-invisible 12
10(2) by simp
also have ... = int (fold ex v q) using eq-fin-execute assms(2) 4(2) by simp
finally show ?thesis by this
qed
lemma execute-inf-visible:
assumes *run* u q *run* v q $u \preceq_I w$ $v \preceq_I w$
assumes *lproject visible* (*llist-of-stream* u) = *lproject visible* (*llist-of-stream* v)
shows *snth* (*smap int* ($q \#\# \text{trace } u$ q)) \approx *snth* (*smap int* ($q \#\# \text{trace } v$ q))
proof -
have 1: *lnth* (*llist-of-stream* (*smap int* ($q \#\# \text{trace } u$ q))) \approx
lnth (*llist-of-stream* (*smap int* ($q \#\# \text{trace } v$ q)))
proof
show *linfinite* (*llist-of-stream* (*smap int* ($q \#\# \text{trace } u$ q))) **by simp**
show *linfinite* (*llist-of-stream* (*smap int* ($q \#\# \text{trace } v$ q))) **by simp**
show *stutter-selection* (*lift* (*lset visible* (*llist-of-stream* u))) (*llist-of-stream*
(*smap int* ($q \#\# \text{trace } u$ q)))
using *stutter-selection-lproject-visible* *assms(1)* **by this**
show *stutter-selection* (*lift* (*lset visible* (*llist-of-stream* v))) (*llist-of-stream*
(*smap int* ($q \#\# \text{trace } v$ q)))
using *stutter-selection-lproject-visible* *assms(2)* **by this**
show *lselect* (*lift* (*lset visible* (*llist-of-stream* u))) (*llist-of-stream* (*smap int*
($q \#\# \text{trace } u$ q))) =
lselect (*lift* (*lset visible* (*llist-of-stream* v))) (*llist-of-stream* (*smap int* (q
 $\#\# \text{trace } v$ q)))
proof
have *llength* (*lselect* (*lift* (*lset visible* (*llist-of-stream* u))))
(*llist-of-stream* (*smap int* ($q \#\# \text{trace } u$ q))) = *eSuc* (*llength* (*lproject*
visible (*llist-of-stream* u)))
by (*simp add: lselect-llength*)
also have ... = *eSuc* (*llength* (*lproject visible* (*llist-of-stream* v)))
unfolding *assms(5)* **by rule**
also have ... = *llength* (*lselect* (*lift* (*lset visible* (*llist-of-stream* v)))
(*llist-of-stream* (*smap int* ($q \#\# \text{trace } v$ q)))) **by** (*simp add: lselect-llength*)
finally show *llength* (*lselect* (*lift* (*lset visible* (*llist-of-stream* u)))
(*llist-of-stream* (*smap int* ($q \#\# \text{trace } u$ q)))) = *llength* (*lselect* (*lift* (*lset*
visible (*llist-of-stream* v))))

```

      (llist-of-stream (smap int (q ## trace v q))) by this
next
  fix i
  assume 1:
    enat i < llength (lselect (lift (lset visible (llist-of-stream u)))
      (llist-of-stream (smap int (q ## trace u q))))
    enat i < llength (lselect (lift (lset visible (llist-of-stream v)))
      (llist-of-stream (smap int (q ## trace v q))))
  have 2:
    enat i ≤ llength (lproject visible (llist-of-stream u))
    enat i ≤ llength (lproject visible (llist-of-stream v))
    using 1 by (simp add: lselect-llength)+
  define k where k ≡ nth-least (lift (lset visible (llist-of-stream u))) i
  define l where l ≡ nth-least (lift (lset visible (llist-of-stream v))) i
  have lselect (lift (lset visible (llist-of-stream u))) (llist-of-stream (smap
int (q ## trace u q))) ?! i =
    int ((q ## trace u q) !! nth-least (lift (lset visible (llist-of-stream u))) i)
  by (metis 1(1) lnth-list-of-stream lselect-lnth snth-smap)
  also have ... = int ((q ## trace u q) !! k) unfolding k-def by rule
  also have ... = int ((q ## trace v q) !! l)
  unfolding sscan-scons-snth
  proof (rule execute-fin-visible)
    show path (stake k u) q using assms(1) by (metis run-shift-elim
stake-sdrop)
    show path (stake l v) q using assms(2) by (metis run-shift-elim
stake-sdrop)
    show stake k u ≤FI w stake l v ≤FI w using assms(3, 4) by auto
    have project visible (stake k u) =
      list-of (lproject visible (llist-of (stake k u))) by simp
    also have ... = list-of (lproject visible (ltake (enat k) (llist-of-stream
u)))
    by (metis length-stake llength-llist-of llist-of-inf-llist-prefix lprefix-ltake
prefix-fininf-prefix)
    also have ... = list-of (ltake (enat i) (lproject visible (llist-of-stream u)))
    unfolding k-def lproject-ltake[OF 2(1)] by rule
    also have ... = list-of (ltake (enat i) (lproject visible (llist-of-stream v)))
    unfolding assms(5) by rule
    also have ... = list-of (lproject visible (ltake (enat l) (llist-of-stream v)))
    unfolding l-def lproject-ltake[OF 2(2)] by rule
    also have ... = project visible (stake l v)
    by (metis length-stake lfilter-llist-of list-of-llist-of llength-llist-of
llist-of-inf-llist-prefix lprefix-ltake prefix-fininf-prefix)
    finally show project visible (stake k u) = project visible (stake l v) by
this
  qed
  also have ... = int ((q ## trace v q) !! nth-least (lift (lset visible
(llist-of-stream v))) i)
  unfolding l-def by simp
  also have ... = lselect (lift (lset visible (llist-of-stream v)))

```

```

      (lstream (smap int (q ## trace v q))) ?! i
      using 1 by (metis lnth-list-of-stream lselect-lnth snth-smap)
      finally show lselect (lift (lset visible (lstream u)))
        (lstream (smap int (q ## trace u q))) ?! i = lselect (lift (lset
visible (lstream v)))
        (lstream (smap int (q ## trace v q))) ?! i by this
      qed
      qed
      show ?thesis using 1 by simp
      qed
    end
  end
end

```

18 Abstract Theory of Ample Set Partial Order Reduction

theory *Ample-Abstract*

imports

Transition-System-Interpreted-Traces

Extensions/Relation-Extensions

begin

locale *ample-base* =

transition-system-interpreted-traces ex en int ind +

wellfounded-relation src

for *ex* :: 'action \Rightarrow 'state \Rightarrow 'state

and *en* :: 'action \Rightarrow 'state \Rightarrow bool

and *int* :: 'state \Rightarrow 'interpretation

and *ind* :: 'action \Rightarrow 'action \Rightarrow bool

and *src* :: 'state \Rightarrow 'state \Rightarrow bool

begin

definition *ample-set* :: 'state \Rightarrow 'action set \Rightarrow bool

where *ample-set* *q* *A* \equiv

$A \subseteq \{a. \text{en } a \text{ } q\} \wedge$

$(A \subset \{a. \text{en } a \text{ } q\} \longrightarrow A \neq \{\}) \wedge$

$(\forall a. A \subset \{a. \text{en } a \text{ } q\} \longrightarrow a \in A \longrightarrow \text{src } (ex \ a \ q) \ q) \wedge$

$(A \subset \{a. \text{en } a \text{ } q\} \longrightarrow A \subseteq \text{invisible}) \wedge$

$(\forall w. A \subset \{a. \text{en } a \text{ } q\} \longrightarrow \text{path } w \ q \longrightarrow A \cap \text{set } w = \{\} \longrightarrow \text{Ind } A \ (\text{set } w))$

lemma *ample-subset*:

assumes *ample-set* *q* *A*

shows $A \subseteq \{a. \text{en } a \text{ } q\}$

```

using assms unfolding ample-set-def by auto

lemma ample-nonempty:
  assumes ample-set q A  $A \subset \{a. \text{en } a \text{ } q\}$ 
  shows  $A \neq \{\}$ 
  using assms unfolding ample-set-def by auto

lemma ample-wellfounded:
  assumes ample-set q A  $A \subset \{a. \text{en } a \text{ } q\}$   $a \in A$ 
  shows src (ex a q) q
  using assms unfolding ample-set-def by auto

lemma ample-invisible:
  assumes ample-set q A  $A \subset \{a. \text{en } a \text{ } q\}$ 
  shows  $A \subseteq \textit{invisible}$ 
  using assms unfolding ample-set-def by auto

lemma ample-independent:
  assumes ample-set q A  $A \subset \{a. \text{en } a \text{ } q\}$  path w q  $A \cap \textit{set } w = \{\}$ 
  shows Ind A (set w)
  using assms unfolding ample-set-def by auto

lemma ample-en[intro]: ample-set q  $\{a. \text{en } a \text{ } q\}$  unfolding ample-set-def by
blast

end

locale ample-abstract =
  S?: transition-system-complete ex en init int +
  R: transition-system-complete ex ren init int +
  ample-base ex en int ind src
  for ex :: 'action  $\Rightarrow$  'state  $\Rightarrow$  'state
  and en :: 'action  $\Rightarrow$  'state  $\Rightarrow$  bool
  and init :: 'state  $\Rightarrow$  bool
  and int :: 'state  $\Rightarrow$  'interpretation
  and ind :: 'action  $\Rightarrow$  'action  $\Rightarrow$  bool
  and src :: 'state  $\Rightarrow$  'state  $\Rightarrow$  bool
  and ren :: 'action  $\Rightarrow$  'state  $\Rightarrow$  bool
  +
  assumes reduction-ample:  $q \in \textit{nodes} \Longrightarrow \textit{ample-set } q \{a. \text{ren } a \text{ } q\}$ 
begin

lemma reduction-words-fin:
  assumes  $q \in \textit{nodes}$  R.path w q
  shows S.path w q
  using assms(2, 1) ample-subset reduction-ample by induct auto
lemma reduction-words-inf:
  assumes  $q \in \textit{nodes}$  R.run w q
  shows S.run w q

```

using *reduction-words-fin* *assms* by (auto intro: *words-infI-construct*)

lemma *reduction-step*:

assumes $q \in \text{nodes}$ *run* w q

obtains

(deferred) a **where** $\text{ren } a \ q \ [a] \preceq_{FI} w \mid$

(omitted) $\{a. \text{ren } a \ q\} \subseteq \text{invisible } \text{Ind } \{a. \text{ren } a \ q\} \ (\text{sset } w)$

proof (cases $\{a. \text{en } a \ q\} = \{a. \text{ren } a \ q\}$)

case *True*

have 1: *run* (*shd* w *##* *sdrop* 1 w) q **using** *assms*(2) **by** *simp*

show *?thesis*

proof (rule *deferred*)

show *ren* (*shd* w) q **using** *True* 1 **by** *blast*

show $[\text{shd } w] \preceq_{FI} w$ **by** *simp*

qed

next

case *False*

have 1: $\{a. \text{ren } a \ q\} \subset \{a. \text{en } a \ q\}$ **using** *ample-subset* *reduction-ample* *assms*(1) *False* **by** *auto*

show *?thesis*

proof (cases $\{a. \text{ren } a \ q\} \cap \text{sset } w = \{\}$)

case *True*

show *?thesis*

proof (rule *omitted*)

assms(1) 1 **by** *auto*

show *Ind* $\{a. \text{ren } a \ q\} \ (\text{sset } w)$

proof *safe*

fix $a \ b$

assume 2: $b \in \text{sset } w$ *ren* $a \ q$

obtain $u \ v$ **where** 3: $w = u \ @- \ b \ ## \ v$ **using** *split-stream-first'* 2(1)

by *this*

have 4: *Ind* $\{a. \text{ren } a \ q\} \ (\text{set } (u \ @ \ [b]))$

proof (rule *ample-independent*)

show *ample-set* $q \ \{a. \text{ren } a \ q\}$ **using** *reduction-ample* *assms*(1) **by** *this*

show $\{a. \text{ren } a \ q\} \subset \{a. \text{en } a \ q\}$ **using** 1 **by** *this*

show *path* $(u \ @ \ [b]) \ q$ **using** *assms*(2) 3 **by** *blast*

show $\{a. \text{ren } a \ q\} \cap \text{set } (u \ @ \ [b]) = \{\}$ **using** *True* 3 **by** *auto*

qed

show *ind* $a \ b$ **using** 2 3 4 **by** *auto*

qed

qed

next

case *False*

obtain $u \ a \ v$ **where** 2: $w = u \ @- \ a \ ## \ v$ $\{a. \text{ren } a \ q\} \cap \text{set } u = \{\}$ *ren* a

q

using *split-stream-first*[*OF* *False*] **by** *auto*

have 3: *path* $u \ q$ **using** *assms*(2) **unfolding** 2(1) **by** *auto*

have 4: $Ind \{a. ren a q\}$ (set u)
using *ample-independent reduction-ample assms(1) 1 3 2(2)* **by this**
have 5: Ind (set $[a]$) (set u) **using** 4 2(3) **by simp**
have 6: $[a] @ u =_F u @ [a]$ **using** 5 **by blast**
show ?thesis
proof (rule deferred)
show $ren a q$ **using** 2(3) **by this**
have $[a] \preceq_{FI} [a] @- u @- v$ **by blast**
also have $[a] @- u @- v = ([a] @ u) @- v$ **by simp**
also have $([a] @ u) @- v =_I (u @ [a]) @- v$ **using** 6 **by blast**
also have $(u @ [a]) @- v = u @- [a] @- v$ **by simp**
also have $\dots = w$ **unfolding** 2(1) **by simp**
finally show $[a] \preceq_{FI} w$ **by this**
qed
qed
qed

lemma *reduction-chunk*:

assumes $q \in nodes$ run ($[a] @- v$) q
obtains $b b_1 b_2 u$
where
 $R.path$ ($b @ [a]$) q
 $Ind \{a\}$ (set b) set $b \subseteq invisible$
 $b =_F b_1 @ b_2$ $b_1 @- u =_I v$ Ind (set b_2) (sset u)
using *wellfounded assms*
proof (induct q arbitrary: thesis v rule: *wfp-induct-rule*)
case (*less* q)
show ?case
proof (cases $ren a q$)
case (*True*)
show ?thesis
proof (rule *less(2)*)
show $R.path$ ($[] @ [a]$) q **using** *True* **by auto**
show $Ind \{a\}$ (set $[]$) **by auto**
show set $[] \subseteq invisible$ **by auto**
show $[] =_F [] @ []$ **by auto**
show $[] @- v =_I v$ **by auto**
show Ind (set $[]$) (sset v) **by auto**
qed
next
case (*False*)
have 0: $\{a. en a q\} \neq \{a. ren a q\}$ **using** *False less(4)* **by auto**
show ?thesis
using *less(3, 4)*
proof (cases rule: *reduction-step*)
case (*deferred* c)
have 1: $ren c q$ **using** *deferred(1)* **by simp**
have 2: $ind a c$

proof (rule *le-fininf-ind''*)
show $[a] \preceq_{FI} [a] @- v$ **by** *blast*
show $[c] \preceq_{FI} [a] @- v$ **using** *deferred(2)* **by** *this*
show $a \neq c$ **using** *False 1* **by** *auto*
qed
obtain v' **where** $\exists: [a] @- v =_I [c] @- [a] @- v'$
proof –
have $10: [c] \preceq_{FI} v$
proof (rule *le-fininf-not-member'*)
show $[c] \preceq_{FI} [a] @- v$ **using** *deferred(2)* **by** *this*
show $c \notin \text{set } [a]$ **using** *False 1* **by** *auto*
qed
obtain v' **where** $11: v =_I [c] @- v'$ **using** 10 **by** *blast*
have $12: \text{Ind } (\text{set } [a]) (\text{set } [c])$ **using** 2 **by** *auto*
have $13: [a] @ [c] =_F [c] @ [a]$ **using** 12 **by** *blast*
have $[a] @- v =_I [a] @- [c] @- v'$ **using** 11 **by** *blast*
also have $\dots = ([a] @ [c]) @- v'$ **by** *simp*
also have $\dots =_I ([c] @ [a]) @- v'$ **using** 13 **by** *blast*
also have $\dots = [c] @- [a] @- v'$ **by** *simp*
finally show *?thesis* **using** *that* **by** *auto*
qed
have $4: \text{run } ([c] @- [a] @- v') q$ **using** *eq-inf-word 3 less(4)* **by** *this*
show *?thesis*
proof (rule *less(1)*)
show *src* $(ex\ c\ q)\ q$
using *ample-wellfounded ample-subset reduction-ample less(3) 0 1* **by**
blast
have $100: ex\ c\ q$ **using** *less(4) deferred(2) le-fininf-word* **by** *auto*
show $ex\ c\ q \in \text{nodes}$ **using** *less(3) 100* **by** *auto*
show $\text{run } ([a] @- v') (ex\ c\ q)$ **using** 4 **by** *auto*
next
fix $b\ b_1\ b_2\ u$
assume $5: R.\text{path } (b @ [a]) (ex\ c\ q)$
assume $6: \text{Ind } \{a\} (\text{set } b)$
assume $7: \text{set } b \subseteq \text{invisible}$
assume $8: b =_F b_1 @ b_2$
assume $9: b_1 @- u =_I v'$
assume $10: \text{Ind } (\text{set } b_2) (\text{sset } u)$
show *thesis*
proof (rule *less(2)*)
show $R.\text{path } (([c] @ b) @ [a]) q$ **using** $1\ 5$ **by** *auto*
show $\text{Ind } \{a\} (\text{set } ([c] @ b))$ **using** $6\ 2$ **by** *auto*
have $11: c \in \text{invisible}$
using *ample-invisible ample-subset reduction-ample less(3) 0 1* **by**
blast
show $\text{set } ([c] @ b) \subseteq \text{invisible}$ **using** $7\ 11$ **by** *auto*
have $[c] @ b =_F [c] @ b_1 @ b_2$ **using** 8 **by** *blast*
also have $[c] @ b_1 @ b_2 = ([c] @ b_1) @ b_2$ **by** *simp*
finally show $[c] @ b =_F ([c] @ b_1) @ b_2$ **by** *this*


```

show  $([c] @ b_1) @- u =_I v$ 
proof -
  have 10:  $Ind (set [a]) (set [c])$  using 2 by auto
  have 11:  $[a] @ [c] =_F [c] @ [a]$  using 10 by blast
  have  $[a] @- v =_I [c] @- [a] @- v'$  using 3 by this
  also have  $\dots = ([c] @ [a]) @- v'$  by simp
  also have  $\dots =_I ([a] @ [c]) @- v'$  using 11 by blast
  also have  $\dots = [a] @- [c] @- v'$  by simp
  finally have 12:  $[a] @- v =_I [a] @- [c] @- v'$  by this
  have 12:  $v =_I [c] @- v'$  using 12 by blast
  have  $([c] @ b_1) @- u = [c] @- b_1 @- u$  by simp
  also have  $\dots =_I [c] @- v'$  using 9 by blast
  also have  $\dots =_I v$  using 12 by blast
  finally show ?thesis by this
qed
show  $Ind (set b_2) (sset u)$  using 10 by this
qed
qed
next
case (omitted)
have 1:  $\{a. ren a q\} \subseteq invisible$  using omitted(1) by simp
have 2:  $Ind \{a. ren a q\} (sset ([a] @- v))$  using omitted(2) by simp
obtain c where 3:  $ren c q$ 
proof -
  have 1:  $en a q$  using less(4) by auto
  show ?thesis using reduction-ample ample-nonempty less(3) 1 that by
blast
qed
have 4:  $Ind (set [c]) (sset ([a] @- v))$  using 2 3 by auto
have 6:  $path [c] q$  using reduction-ample ample-subset less(3) 3 by auto
have 7:  $run ([a] @- v) (target [c] q)$  using diamond-fin-word-inf-word 4
6 less(4) by this
show ?thesis
proof (rule less(1))
  show  $src (ex c q) q$ 
  using reduction-ample ample-wellfounded ample-subset less(3) 0 3 by
blast
  show  $ex c q \in nodes$  using less(3) 6 by auto
  show  $run ([a] @- v) (ex c q)$  using 7 by auto
next
fix b s b1 b2 u
assume 5:  $R.path (b @ [a]) (ex c q)$ 
assume 6:  $Ind \{a\} (set b)$ 
assume 7:  $set b \subseteq invisible$ 
assume 8:  $b =_F b_1 @ b_2$ 
assume 9:  $b_1 @- u =_I v$ 
assume 10:  $Ind (set b_2) (sset u)$ 
show thesis
proof (rule less(2))

```

```

show  $R.path$  ( $([c] @ b) @ [a]$ )  $q$  using 3 5 by auto
show  $Ind \{a\}$  (set ( $[c] @ b$ ))
proof -
  have 1:  $ind\ c\ a$  using 4 by simp
  have 2:  $ind\ a\ c$  using independence-symmetric 1 by this
  show ?thesis using 6 2 by auto
qed
have 11:  $c \in invisible$  using 1 3 by auto
show set ( $[c] @ b$ )  $\subseteq invisible$  using 7 11 by auto
have 12:  $Ind$  (set  $[c]$ ) (set  $b_1$ )
proof -
  have 1: set  $b_1 \subseteq sset\ v$  using 9 by force
  have 2:  $Ind$  (set  $[c]$ ) (sset  $v$ ) using 4 by simp
  show ?thesis using 1 2 by auto
qed
have  $[c] @ b =_F [c] @ b_1 @ b_2$  using 8 by blast
also have  $\dots = ([c] @ b_1) @ b_2$  by simp
also have  $\dots =_F (b_1 @ [c]) @ b_2$  using 12 by blast
also have  $\dots = b_1 @ [c] @ b_2$  by simp
finally show  $[c] @ b =_F b_1 @ [c] @ b_2$  by this
show  $b_1 @- u =_I v$  using 9 by this
have 13:  $Ind$  (set  $[c]$ ) (sset  $u$ )
proof -
  have 1: sset  $u \subseteq sset\ v$  using 9 by force
  have 2:  $Ind$  (set  $[c]$ ) (sset  $v$ ) using 4 by simp
  show ?thesis using 1 2 by blast
qed
show  $Ind$  (set ( $[c] @ b_2$ )) (sset  $u$ ) using 10 13 by auto
qed
qed
qed
qed
qed

```

inductive *reduced-run* :: 'state \Rightarrow 'action list \Rightarrow 'action stream \Rightarrow 'action list
 \Rightarrow 'action list \Rightarrow 'action list \Rightarrow 'action list \Rightarrow 'action stream \Rightarrow bool
where
init: $reduced-run\ q\ []\ v\ []\ []\ []\ v\ |$
absorb: $reduced-run\ q\ v_1\ ([a] @- v_2)\ l\ w\ w_1\ w_2\ u \implies a \in set\ l \implies$
 $reduced-run\ q\ (v_1 @ [a])\ v_2\ (remove1\ a\ l)\ w\ w_1\ w_2\ u\ |$
extend: $reduced-run\ q\ v_1\ ([a] @- v_2)\ l\ w\ w_1\ w_2\ u \implies a \notin set\ l \implies$
 $R.path\ (b @ [a])\ (target\ w\ q) \implies$
 $Ind\ \{a\}\ (set\ b) \implies set\ b \subseteq invisible \implies$
 $b =_F\ b_1 @ b_2 \implies [a] @- b_1 @- u' =_I\ u \implies Ind\ (set\ b_2)\ (sset\ u') \implies$
 $reduced-run\ q\ (v_1 @ [a])\ v_2\ (l @ b_1)\ (w @ b @ [a])\ (w_1 @ b_1 @ [a])\ (w_2 @$
 $b_2)\ u'$

```

lemma reduced-run-words-fin:
  assumes reduced-run  $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$ 
  shows  $R.path\ w\ q$ 
  using assms by induct auto

lemma reduced-run-invar-2:
  assumes reduced-run  $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$ 
  shows  $v_2 =_I l\ @- u$ 
  using assms
  proof induct
    case (init  $q\ v$ )
    show ?case by simp
  next
    case (absorb  $q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u$ )
    obtain  $l_1\ l_2$  where  $10: l = l_1\ @ [a]\ @ l_2\ a \notin set\ l_1$ 
      using split-list-first[OF absorb(3)] by auto
    have  $11: Ind\ \{a\}\ (set\ l_1)$ 
    proof (rule le-fininf-ind')
      show  $[a] \preceq_{FI} l\ @- u$  using absorb(2) by auto
      show  $l_1 \preceq_{FI} l\ @- u$  unfolding  $10(1)$  by auto
      show  $a \notin set\ l_1$  using  $10(2)$  by this
    qed
    have  $12: Ind\ (set\ [a])\ (set\ l_1)$  using  $11$  by auto
    have  $[a]\ @\ remove1\ a\ l = [a]\ @\ l_1\ @\ l_2$  unfolding  $10(1)$  remove1-append
using  $10(2)$  by auto
    also have  $\dots =_F ([a]\ @\ l_1)\ @\ l_2$  by simp
    also have  $\dots =_F (l_1\ @\ [a])\ @\ l_2$  using  $12$  by blast
    also have  $\dots = l$  unfolding  $10(1)$  by simp
    finally have  $13: [a]\ @\ remove1\ a\ l =_F l$  by this
    have  $[a]\ @- remove1\ a\ l\ @- u = ([a]\ @\ remove1\ a\ l)\ @- u$  unfolding
conc-conc by simp
    also have  $\dots =_I l\ @- u$  using  $13$  by blast
    also have  $\dots =_I [a]\ @- v_2$  using absorb(2) by auto
    finally show ?case by blast
  next
    case (extend  $q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u\ b\ b_1\ b_2\ u'$ )
    have  $11: Ind\ \{a\}\ (set\ l)$ 
    proof (rule le-fininf-ind')
      show  $[a] \preceq_{FI} l\ @- u$  using extend(2) by auto
      show  $l \preceq_{FI} l\ @- u$  by auto
      show  $a \notin set\ l$  using extend(3) by this
    qed
    have  $11: Ind\ (set\ [a])\ (set\ l)$  using  $11$  by auto
    have  $12: eq-fin\ ([a]\ @\ l)\ (l\ @\ [a])$  using  $11$  by blast
    have  $131: set\ b_1 \subseteq set\ b$  using extend(7) by auto
    have  $132: Ind\ (set\ [a])\ (set\ b)$  using extend(5) by auto
    have  $13: Ind\ (set\ [a])\ (set\ b_1)$  using  $131\ 132$  by auto
    have  $14: eq-fin\ ([a]\ @\ b_1)\ (b_1\ @\ [a])$  using  $13$  by blast
    have  $[a]\ @- ((l\ @\ b_1)\ @- u') = ([a]\ @\ l)\ @- b_1\ @- u'$  by simp

```

also have $eq\text{-}inf \dots ((l @ [a]) @- b_1 @- u')$ **using 12 by blast**
 also have $\dots = l @- [a] @- b_1 @- u'$ **by simp**
 also have $eq\text{-}inf \dots (l @- u)$ **using extend(8) by blast**
 also have $eq\text{-}inf \dots ([a] @- v_2)$ **using extend(2) by auto**
 finally show $?case$ **by blast**
qed

lemma reduced-run-invar-1:

assumes $reduced\text{-}run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$

shows $v_1 @ l =_F w_1$

using $assms$

proof $induct$

case $(init\ q\ v)$

show $?case$ **by simp**

next

case $(absorb\ q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u)$

have $1: [a] @- v_2 =_I l @- u$ **using reduced-run-invar-2 absorb(1) by this**

obtain $l_1\ l_2$ **where** $10: l = l_1 @ [a] @ l_2\ a \notin set\ l_1$

using $split\text{-}list\text{-}first[OF\ absorb(3)]$ **by auto**

have $11: Ind\ \{a\}\ (set\ l_1)$

proof $(rule\ le\text{-}fininf\text{-}ind')$

show $[a] \preceq_{FI} l @- u$ **using 1 by auto**

show $l_1 \preceq_{FI} l @- u$ **unfolding 10(1) by auto**

show $a \notin set\ l_1$ **using 10(2) by this**

qed

have $12: Ind\ (set\ [a])\ (set\ l_1)$ **using 11 by auto**

have $[a] @ remove1\ a\ l = [a] @ l_1 @ l_2$ **unfolding 10(1) remove1-append**

using 10(2) by auto

also have $\dots =_F ([a] @ l_1) @ l_2$ **by simp**

also have $\dots =_F (l_1 @ [a]) @ l_2$ **using 12 by blast**

also have $\dots = l$ **unfolding 10(1) by simp**

finally have $13: [a] @ remove1\ a\ l =_F l$ **by this**

have $w_1 =_F v_1 @ l$ **using absorb(2) by auto**

also have $\dots =_F v_1 @ ([a] @ remove1\ a\ l)$ **using 13 by blast**

also have $\dots = (v_1 @ [a]) @ remove1\ a\ l$ **by simp**

finally show $?case$ **by auto**

next

case $(extend\ q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u\ b\ b_1\ b_2\ u')$

have $1: [a] @- v_2 =_I l @- u$ **using reduced-run-invar-2 extend(1) by this**

have $11: Ind\ \{a\}\ (set\ l)$

proof $(rule\ le\text{-}fininf\text{-}ind')$

show $[a] \preceq_{FI} l @- u$ **using 1 by auto**

show $l \preceq_{FI} l @- u$ **by auto**

show $a \notin set\ l$ **using extend(3) by auto**

qed

have $11: Ind\ (set\ [a])\ (set\ l)$ **using 11 by auto**

have $12: eq\text{-}fin\ ([a] @ l)\ (l @ [a])$ **using 11 by blast**

have $131: set\ b_1 \subseteq set\ b$ **using extend(7) by auto**

have $132: Ind\ (set\ [a])\ (set\ b)$ **using extend(5) by auto**

```

have 13: Ind (set [a]) (set b1) using 131 132 by auto
have 14: eq-fin ([a] @ b1) (b1 @ [a]) using 13 by blast
have eq-fin (w1 @ b1 @ [a]) (w1 @ [a] @ b1) using 14 by blast
also have eq-fin ... ((v1 @ l) @ [a] @ b1) using extend(2) by blast
also have eq-fin ... (v1 @ (l @ [a]) @ b1) by simp
also have eq-fin ... (v1 @ ([a] @ l) @ b1) using 12 by blast
also have ... = (v1 @ [a]) @ l @ b1 by simp
finally show ?case by auto
qed

```

lemma *reduced-run-invisible:*

```

assumes reduced-run q v1 v2 l w w1 w2 u
shows set w2 ⊆ invisible
using assms
proof induct
  case (init q v)
    show ?case by simp
next
  case (absorb q v1 a v2 l w w1 w2 u)
    show ?case using absorb(2) by this
next
  case (extend q v1 a v2 l w w1 w2 u b b1 b2 u')
    have 1: set b2 ⊆ set b using extend(7) by auto
    show ?case unfolding set-append using extend(2) extend(6) 1 by blast
qed

```

lemma *reduced-run-ind:*

```

assumes reduced-run q v1 v2 l w w1 w2 u
shows Ind (set w2) (sset u)
using assms
proof induct
  case (init q v)
    show ?case by simp
next
  case (absorb q v1 a v2 l w w1 w2 u)
    show ?case using absorb(2) by this
next
  case (extend q v1 a v2 l w w1 w2 u b b1 b2 u')
    have 1: sset u' ⊆ sset u using extend(8) by force
    show ?case using extend(2) extend(9) 1 unfolding set-append by blast
qed

```

lemma *reduced-run-decompose:*

```

assumes reduced-run q v1 v2 l w w1 w2 u
shows w =F w1 @ w2
using assms
proof induct
  case (init q v)
    show ?case by simp

```

next
 case (*absorb* $q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u$)
 show ?*case* **using** *absorb*(2) **by this**
next
 case (*extend* $q\ v_1\ a\ v_2\ l\ w\ w_1\ w_2\ u\ b\ b_1\ b_2\ u'$)
 have 1: *Ind* (*set* [a]) (*set* b₂) **using** *extend*(5) *extend*(7) **by auto**
 have 2: *Ind* (*set* w₂) (*set* (b₁ @ [a]))
proof –
 have 1: *Ind* (*set* w₂) (*sset* u) **using** *reduced-run-ind* *extend*(1) **by this**
 have 2: $u =_I [a] @- b_1 @- u'$ **using** *extend*(8) **by auto**
 have 3: *sset* u = *sset* ([a] @- b₁ @- u') **using** 2 **by blast**
 show ?*thesis* **unfolding** *set-append* **using** 1 3 **by simp**
qed
 have $w @ b @ [a] =_F (w_1 @ w_2) @ b @ [a]$ **using** *extend*(2) **by blast**
 also have $\dots =_F (w_1 @ w_2) @ (b_1 @ b_2) @ [a]$ **using** *extend*(7) **by blast**
 also have $\dots = w_1 @ w_2 @ b_1 @ (b_2 @ [a])$ **by simp**
 also have $\dots =_F w_1 @ w_2 @ b_1 @ ([a] @ b_2)$ **using** 1 **by blast**
 also have $\dots =_F w_1 @ (w_2 @ (b_1 @ [a])) @ b_2$ **by simp**
 also have $\dots =_F w_1 @ ((b_1 @ [a]) @ w_2) @ b_2$ **using** 2 **by blast**
 also have $\dots =_F (w_1 @ b_1 @ [a]) @ w_2 @ b_2$ **by simp**
finally show ?*case* **by this**
qed

lemma *reduced-run-project*:

assumes *reduced-run* $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
 shows *project visible* $w_1 = \text{project visible } w$

proof –

have 1: $w_1 @ w_2 =_F w$ **using** *reduced-run-decompose* *assms* **by auto**
 have 2: *set* w₂ \subseteq *invisible* **using** *reduced-run-invisible* *assms* **by this**
 have 3: *project visible* w₂ = [] **unfolding** *filter-empty-conv* **using** 2 **by auto**
 have *project visible* w₁ = *project visible* w₁ @ *project visible* w₂ **using** 3 **by**

simp

also have $\dots = \text{project visible } (w_1 @ w_2)$ **by simp**
 also have $\dots = \text{list-of } (\text{lproject visible } (\text{llist-of } (w_1 @ w_2)))$ **by simp**
 also have $\dots = \text{list-of } (\text{lproject visible } (\text{llist-of } w))$
using *eq-fin-lproject-visible* 1 **by metis**
 also have $\dots = \text{project visible } w$ **by simp**
finally show ?*thesis* **by this**
qed

lemma *reduced-run-length-1*:

assumes *reduced-run* $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
 shows *length* v₁ \leq *length* w₁
using *reduced-run-invar-1* *assms* **by force**

lemma *reduced-run-length*:

assumes *reduced-run* $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
 shows *length* v₁ \leq *length* w

proof –

have *length* v₁ \leq *length* w₁ **using** *reduced-run-length-1* *assms* **by this**

also have $\dots \leq \text{length } w$ **using** *reduced-run-decompose* *assms* **by force**
 finally **show** *?thesis* **by this**
qed

lemma *reduced-run-step*:
 assumes $q \in \text{nodes run } (v_1 @- [a] @- v_2) q$
 assumes *reduced-run* $q v_1 ([a] @- v_2) l w w_1 w_2 u$
 obtains $l' w' w_1' w_2' u'$
 where *reduced-run* $q (v_1 @ [a]) v_2 l' (w @ w') (w_1 @ w_1') (w_2 @ w_2') u'$
proof (*cases* $a \in \text{set } l$)
 case *True*
 show *?thesis*
proof (*rule that, rule absorb*)
 show *reduced-run* $q v_1 ([a] @- v_2) l (w @ []) (w_1 @ []) (w_2 @ []) u$ **using**
assms(3) **by simp**
 show $a \in \text{set } l$ **using** *True* **by this**
qed
 next
 case *False*
 have 1: $v_1 @ l =_F w_1$ **using** *reduced-run-invar-1* *assms(3)* **by this**
 have 2: $[a] @- v_2 =_I l @- u$ **using** *reduced-run-invar-2* *assms(3)* **by this**
 have 3: $w =_F w_1 @ w_2$ **using** *reduced-run-decompose* *assms(3)* **by this**
 have $v_1 @ l @ w_2 = (v_1 @ l) @ w_2$ **by simp**
 also have $\dots =_F w_1 @ w_2$ **using** 1 **by blast**
 also have $\dots =_F w$ **using** 3 **by blast**
 finally have 4: $v_1 @ l @ w_2 =_F w$ **by this**
 have 5: *run* $((v_1 @ l) @- w_2 @- u) q$
proof (*rule diamond-fin-word-inf-word'*)
 show *Ind* (*set* w_2) (*sset* u) **using** *reduced-run-ind* *assms(3)* **by this**
 have 6: *R.path* $w q$ **using** *reduced-run-words-fin* *assms(3)* **by this**
 have 7: *path* $w q$ **using** *reduction-words-fin* *assms(1)* 6 **by auto**
 show *path* $((v_1 @ l) @ w_2) q$ **using** *eq-fin-word* 4 7 **by auto**
 have 8: $v_1 @- [a] @- v_2 =_I v_1 @- l @- u$ **using** 2 **by blast**
 show *run* $((v_1 @ l) @- u) q$ **using** *eq-inf-word* *assms(2)* 8 **by auto**
qed
 have 6: *run* $(w @- u) q$ **using** *eq-inf-word* 4 5 **by** (*metis* *eq-inf-concat-end*
shift-append)
 have 7: $[a] \preceq_{FI} l @- u$ **using** 2 **by blast**
 have 8: $[a] \preceq_{FI} u$ **using** *le-fininf-not-member'* 7 *False* **by this**
 obtain u' where 9: $u =_I [a] @- u'$ **using** 8 **by rule**
 have 101: *target* $w q \in \text{nodes}$ **using** *assms(1)* 6 **by auto**
 have 10: *run* $([a] @- u')$ (*target* $w q$) **using** *eq-inf-word* 9 6 **by blast**
 obtain $b b_1 b_2 u''$ where 11:
 R.path $(b @ [a])$ (*target* $w q$)
 Ind $\{a\}$ (*set* b) *set* $b \subseteq \text{invisible}$
 $b =_F b_1 @ b_2 b_1 @- u'' =_I u'$ *Ind* (*set* b_2) (*sset* u'')
 using *reduction-chunk* 101 10 **by this**
 show *?thesis*
proof (*rule that, rule extend*)

show *reduced-run* q v_1 ($[a]$ @- v_2) l w w_1 w_2 u **using** *assms*(3) **by this**
show $a \notin \text{set } l$ **using** *False* **by this**
show $R.\text{path}$ (b @ $[a]$) (target w q) **using** 11(1) **by this**
show $\text{Ind } \{a\}$ (set b) **using** 11(2) **by this**
show $\text{set } b \subseteq \text{invisible}$ **using** 11(3) **by this**
show $b =_F b_1$ @ b_2 **using** 11(4) **by this**
show $[a]$ @- b_1 @- $u'' =_I u$ **using** 9 11(5) **by blast**
show Ind (set b_2) (sset u'') **using** 11(6) **by this**
qed
qed

lemma *reduction-word*:

assumes $q \in \text{nodes}$ *run* v q
obtains u w
where
 $R.\text{run}$ w q
 $v =_I u$ $u \preceq_I w$
 lproject visible ($\text{llist-of-stream } u$) = lproject visible ($\text{llist-of-stream } w$)
proof -
define P **where** $P \equiv \lambda k w w_1. \exists l w_2 u. \text{reduced-run } q$ (stake k v) (*sdrop* k v) l w w_1 w_2 u
obtain w w_1 **where** 1: $\bigwedge k. P$ k (w k) (w_1 k) *chain* w *chain* w_1
proof (*rule chain-construct-2'*[of P])
show P 0 [] [] **unfolding** $P\text{-def}$ **using** *init* **by force**
next
fix k w w_1
assume 1: P k w w_1
obtain l w_2 u **where** 2: *reduced-run* q (stake k v) (*sdrop* k v) l w w_1 w_2 u
using 1 **unfolding** $P\text{-def}$ **by auto**
obtain l' w' w_1' w_2' u' **where** 3:
reduced-run q (stake k v @ $[v !! k]$) (*sdrop* (*Suc* k) v) l' (w @ w') (w_1 @ w_1') (w_2 @ w_2') u'
proof (*rule reduced-run-step*)
show $q \in \text{nodes}$ **using** *assms*(1) **by this**
show *run* (stake k v @- $[v !! k]$ @- *sdrop* (*Suc* k) v) q
using *assms*(2) **by** (*metis shift-append stake-Suc stake-sdrop*)
show *reduced-run* q (stake k v) ($[v !! k]$ @- *sdrop* (*Suc* k) v) l w w_1 w_2 u
using 2 **by** (*metis sdrop-simps shift.simps stream.collapse*)
qed
show $\exists w' w_1'. P$ (*Suc* k) w' w_1' $\wedge w \leq w' \wedge w_1 \leq w_1'$
unfolding $P\text{-def}$ **using** 3 **by** (*metis prefix-fin-append stake-Suc*)
show $k \leq \text{length } w$ **using** *reduced-run-length* 2 **by force**
show $k \leq \text{length } w_1$ **using** *reduced-run-length-1* 2 **by force**
qed rule
obtain l w_2 u **where** 2:
 $\bigwedge k. \text{reduced-run } q$ (stake k v) (*sdrop* k v) (l k) (w k) (w_1 k) (w_2 k) (u k)
using 1(1) **unfolding** $P\text{-def}$ **by metis**
show *?thesis*


```

proof
  show  $R.run (Word-Prefixes.limit w) q$  using reduced-run-words-fin 1(2) 2
by blast
  show  $v =_I Word-Prefixes.limit w_1$ 
  proof
    show  $v \preceq_I Word-Prefixes.limit w_1$ 
    proof (rule le-infI-chain-right')
      show  $chain w_1$  using 1(3) by this
      show  $\bigwedge k. stake k v \preceq_F w_1 k$  using reduced-run-invar-1[OF 2] by auto
    qed
    show  $Word-Prefixes.limit w_1 \preceq_I v$ 
    proof (rule le-infI-chain-left)
      show  $chain w_1$  using 1(3) by this
    next
      fix  $k$ 
      have  $w_1 k =_F stake k v @ l k$  using reduced-run-invar-1 2 by blast
      also have  $\dots \leq_{FI} stake k v @- l k @- u k$  by auto
      also have  $\dots =_I stake k v @- sdrop k v$  using reduced-run-invar-2[OF 2]
    by blast
      also have  $\dots = v$  by simp
      finally show  $w_1 k \preceq_{FI} v$  by this
    qed
  qed
  show  $Word-Prefixes.limit w_1 \preceq_I Word-Prefixes.limit w$ 
  proof (rule le-infI-chain-left)
    show  $chain w_1$  using 1(3) by this
  next
    fix  $k$ 
    have  $w_1 k \preceq_F w k$  using reduced-run-decompose[OF 2] by blast
    also have  $\dots \leq_{FI} Word-Prefixes.limit w$  using chain-prefix-limit 1(2) by
  this
    finally show  $w_1 k \preceq_{FI} Word-Prefixes.limit w$  by this
  qed
  show  $lproject\ visible (l\ list\ of\ stream (Word-Prefixes.limit w_1)) =$ 
     $lproject\ visible (l\ list\ of\ stream (Word-Prefixes.limit w))$ 
    using lproject-eq-limit-chain reduced-run-project 1 unfolding P-def by
  metis
  qed
qed

```

```

lemma reduction-equivalent:
  assumes  $q \in nodes\ run\ u\ q$ 
  obtains  $v$ 
  where  $R.run\ v\ q\ snth (smap\ int (q\ \#\#\ trace\ u\ q)) \approx snth (smap\ int (q\ \#\#\$ 
  trace v q))
  proof –
  obtain  $v\ w$  where 1: R.run w q u =_I v v \preceq_I w
     $lproject\ visible (l\ list\ of\ stream\ v) = lproject\ visible (l\ list\ of\ stream\ w)$ 

```

```

    using reduction-word assms by this
show ?thesis
proof
  show R.run w q using 1(1) by this
  show snth (smap int (q ## trace u q)) ≈ snth (smap int (q ## trace w q))
  proof (rule execute-inf-visible)
    show run u q using assms(2) by this
    show run w q using reduction-words-inf assms(1) 1(1) by auto
    have u =I v using 1(2) by this
    also have ... ≼I w using 1(3) by this
    finally show u ≼I w by this
    show w ≼I w by simp
    have lproject visible (llist-of-stream u) = lproject visible (llist-of-stream v)
      using eq-inf-lproject-visible 1(2) by this
    also have ... = lproject visible (llist-of-stream w) using 1(4) by this
    finally show lproject visible (llist-of-stream u) = lproject visible (llist-of-stream
w) by this
  qed
qed
qed

```

```

lemma reduction-language-subset: R.language ⊆ S.language
  unfolding S.language-def R.language-def using reduction-words-inf by blast

```

```

lemma reduction-language-stuttering:
  assumes u ∈ S.language
  obtains v
  where v ∈ R.language snth u ≈ snth v
proof -
  obtain q v where 1: u = smap int (q ## trace v q) init q S.run v q using
assms by rule
  obtain v' where 2: R.run v' q snth (smap int (q ## trace v q)) ≈ snth
(smap int (q ## trace v' q))
  using reduction-equivalent 1(2, 3) by blast
  show ?thesis
  proof (intro that R.languageI)
    show smap int (q ## trace v' q) = smap int (q ## trace v q) by rule
    show init q using 1(2) by this
    show R.run v' q using 2(1) by this
    show snth u ≈ snth (smap int (q ## trace v' q)) unfolding 1(1) using
2(2) by this
  qed
qed

```

end

end

19 LTL Formulae

```
theory Formula
imports
  Basics/Stuttering
  Stuttering-Equivalence.PLTL
begin

  locale formula =
    fixes  $\varphi :: 'a\ pltl$ 
    begin

      definition language :: 'a stream set
        where language  $\equiv \{w. snth\ w \models_p \varphi\}$ 

      lemma language-entails[iff]:  $w \in language \longleftrightarrow snth\ w \models_p \varphi$  unfolding lan-
        guage-def by simp

    end

    locale formula-next-free =
      formula  $\varphi$ 
      for  $\varphi :: 'a\ pltl$ 
      +
      assumes next-free: next-free  $\varphi$ 
    begin

      lemma stutter-equivalent-entails[dest]:  $u \approx v \implies u \models_p \varphi \longleftrightarrow v \models_p \varphi$ 
        using next-free-stutter-invariant next-free by blast

    end

end
```

20 Correctness Theorem of Partial Order Reduction

```
theory Ample-Correctness
imports
  Ample-Abstract
  Formula
begin

  locale ample-correctness =
    S: transition-system-complete ex en init int +
    R: transition-system-complete ex ren init int +
    F: formula-next-free  $\varphi$  +
    ample-abstract ex en init int ind src ren
```

```

for ex :: 'action ⇒ 'state ⇒ 'state
and en :: 'action ⇒ 'state ⇒ bool
and init :: 'state ⇒ bool
and int :: 'state ⇒ 'interpretation
and ind :: 'action ⇒ 'action ⇒ bool
and src :: 'state ⇒ 'state ⇒ bool
and ren :: 'action ⇒ 'state ⇒ bool
and  $\varphi$  :: 'interpretation pltl
begin

  lemma reduction-language-indistinguishable:
    assumes R.language  $\subseteq$  F.language
    shows S.language  $\subseteq$  F.language
  proof
    fix u
    assume 1: u  $\in$  S.language
    obtain v where 2: v  $\in$  R.language snth u  $\approx$  snth v using reduction-language-stuttering
  1 by this
    have 3: v  $\in$  F.language using assms 2(1) by rule
    show u  $\in$  F.language using 2(2) 3 by auto
  qed

  theorem reduction-correct: S.language  $\subseteq$  F.language  $\longleftrightarrow$  R.language  $\subseteq$  F.language
    using reduction-language-subset reduction-language-indistinguishable by blast

end

end

```

21 Static Analysis for Partial Order Reduction

```

theory Ample-Analysis
imports
  Ample-Abstract
begin

  locale transition-system-ample =
    transition-system-sticky ex en init int sticky +
    transition-system-interpreted-traces ex en int ind
    for ex :: 'action ⇒ 'state ⇒ 'state
    and en :: 'action ⇒ 'state ⇒ bool
    and init :: 'state ⇒ bool
    and int :: 'state ⇒ 'interpretation
    and sticky :: 'action set
    and ind :: 'action ⇒ 'action ⇒ bool
  begin

    sublocale ample-base ex en int ind scut-1-1 by unfold-locales
  end

```

```

lemma restrict-ample-set:
  assumes  $s \in \text{nodes}$ 
  assumes  $A \cap \{a. \text{en } a \ s\} \neq \{\}$   $A \cap \{a. \text{en } a \ s\} \cap \text{sticky} = \{\}$ 
  assumes  $\text{Ind } (A \cap \{a. \text{en } a \ s\})$  (executable -  $A$ )
  assumes  $\bigwedge w. \text{path } w \ s \implies A \cap \{a. \text{en } a \ s\} \cap \text{set } w = \{\} \implies A \cap \text{set } w =$ 
   $\{\}$ 
  shows ample-set  $s$  ( $A \cap \{a. \text{en } a \ s\}$ )
unfolding ample-set-def
proof (intro conjI allI impI)
  show  $A \cap \{a. \text{en } a \ s\} \subseteq \{a. \text{en } a \ s\}$  by simp
next
  show  $A \cap \{a. \text{en } a \ s\} \neq \{\}$  using assms(2) by this
next
  fix  $a$ 
  assume  $1: a \in A \cap \{a. \text{en } a \ s\}$ 
  show  $\text{scut}^{-1-1} (\text{ex } a \ s) \ s$ 
  proof (rule no-cut-scut)
    show  $s \in \text{nodes}$  using assms(1) by this
    show  $\text{en } a \ s$  using  $1$  by simp
    show  $a \notin \text{sticky}$  using assms(3)  $1$  by auto
  qed
next
  have  $1: A \cap \{a. \text{en } a \ s\} \subseteq \text{executable}$  using executable assms(1) by blast
  show  $A \cap \{a. \text{en } a \ s\} \subseteq \text{invisible}$  using executable-visible-sticky assms(3)  $1$ 
by blast
next
  fix  $w$ 
  assume  $1: \text{path } w \ s \ A \cap \{a. \text{en } a \ s\} \cap \text{set } w = \{\}$ 
  have  $2: A \cap \text{set } w = \{\}$  using assms(5)  $1$  by this
  have  $3: \text{set } w \subseteq \text{executable}$  using assms(1)  $1(1)$  by rule
  show  $\text{Ind } (A \cap \{a. \text{en } a \ s\})$  (set  $w$ ) using assms(4)  $2$   $3$  by blast
qed

end

locale transition-system-concurrent =
  transition-system-initial ex en init
  for  $\text{ex} :: 'action \Rightarrow 'state \Rightarrow 'state$ 
  and  $\text{en} :: 'action \Rightarrow 'state \Rightarrow \text{bool}$ 
  and  $\text{init} :: 'state \Rightarrow \text{bool}$ 
  +
  fixes  $\text{procs} :: 'state \Rightarrow 'process \text{ set}$ 
  fixes  $\text{pac} :: 'process \Rightarrow 'action \text{ set}$ 
  fixes  $\text{psen} :: 'process \Rightarrow 'state \Rightarrow 'action \text{ set}$ 
  assumes procs-finite:  $s \in \text{nodes} \implies \text{finite } (\text{procs } s)$ 
  assumes psen-en:  $s \in \text{nodes} \implies \text{pac } p \cap \{a. \text{en } a \ s\} \subseteq \text{psen } p \ s$ 
  assumes psen-ex:  $s \in \text{nodes} \implies a \in \{a. \text{en } a \ s\} - \text{pac } p \implies \text{psen } p (\text{ex } a \ s)$ 
= psen  $p \ s$ 
begin

```

lemma *psen-fin-word*:

assumes $s \in \text{nodes path } w \text{ s pac } p \cap \text{set } w = \{\}$
shows $\text{psen } p (\text{target } w \text{ s}) = \text{psen } p \text{ s}$
using *assms*(2, 1, 3)
proof *induct*
 case (*nil* *s*)
 show *?case* **by** *simp*
next
 case (*cons* *a* *s* *w*)
 have 1: $\text{ex } a \text{ s} \in \text{nodes}$ **using** *cons*(4, 1) **by** *rule*
 have $\text{psen } p (\text{target } (a \# w) \text{ s}) = \text{psen } p (\text{target } w (\text{ex } a \text{ s}))$ **by** *simp*
 also have $\dots = \text{psen } p (\text{ex } a \text{ s})$ **using** *cons* 1 **by** *simp*
 also have $\dots = \text{psen } p \text{ s}$ **using** *psen-ex cons* **by** *simp*
 finally show *?case* **by** *this*
qed

lemma *en-fin-word*:

assumes $\bigwedge r \ a \ b. r \in \text{nodes} \implies a \in \text{psen } p \text{ s} - \{a. \text{en } a \text{ s}\} \implies b \in \{a. \text{en } a \text{ r}\} - \text{pac } p \implies$
 $\text{en } a (\text{ex } b \text{ r}) \implies \text{en } a \text{ r}$
assumes $s \in \text{nodes path } w \text{ s pac } p \cap \text{set } w = \{\}$
shows $\text{pac } p \cap \{a. \text{en } a (\text{target } w \text{ s})\} \subseteq \text{pac } p \cap \{a. \text{en } a \text{ s}\}$
using *assms*
proof (*induct w rule: rev-induct*)
 case *Nil*
 show *?case* **by** *simp*
next
 case (*snoc* *b* *w*)
 show *?case*
 proof (*safe, rule ccontr*)
 fix *a*
 assume 2: $a \in \text{pac } p \text{ en } a (\text{target } (w @ [b]) \text{ s}) \neg \text{en } a \text{ s}$
 have 3: $a \in \text{psen } p \text{ s}$
 proof –
 have 3: $\text{psen } p (\text{target } (w @ [b]) \text{ s}) = \text{psen } p \text{ s}$ **using** *psen-fin-word snoc*(3, 4, 5) **by** *this*
 have 4: $\text{target } (w @ [b]) \text{ s} \in \text{nodes}$ **using** *snoc*(3, 4) **by** *rule*
 have 5: $a \in \text{psen } p (\text{target } (w @ [b]) \text{ s})$ **using** *psen-en* 4 2(1, 2) **by** *auto*
 show *?thesis* **using** 2(1) 3 5 **by** *auto*
 qed
 have 4: $\text{en } a (\text{target } w \text{ s})$
 proof (*rule snoc*(2))
 show $\text{target } w \text{ s} \in \text{nodes}$ **using** *snoc*(3, 4) **by** *auto*
 show $a \in \text{psen } p \text{ s} - \{a. \text{en } a \text{ s}\}$ **using** 2(3) 3 **by** *simp*
 show $b \in \{a. \text{en } a (\text{target } w \text{ s})\} - \text{pac } p$ **using** *snoc*(4, 5) **by** *auto*
 show $\text{en } a (\text{ex } b (\text{target } w \text{ s}))$ **using** 2(2) **by** *simp*
 qed
 have 5: $\text{pac } p \cap \{a. \text{en } a (\text{target } w \text{ s})\} \subseteq \text{pac } p \cap \{a. \text{en } a \text{ s}\}$

```

proof (rule snoc(1))
  show  $\bigwedge r a b. r \in \text{nodes} \implies a \in \text{psen } p s - \{a. \text{en } a s\} \implies b \in \{a. \text{en } a r\} - \text{pac } p \implies$ 
     $\text{en } a (\text{ex } b r) \implies \text{en } a r$  using snoc(2) by this
  show  $s \in \text{nodes}$  using snoc(3) by this
  show  $\text{path } w s$  using snoc(4) by auto
  show  $\text{pac } p \cap \text{set } w = \{\}$  using snoc(5) by auto
qed
have 6:  $\text{en } a s$  using 2(1) 4 5 by auto
show False using 2(3) 6 by simp
qed

```

```

lemma pac-en-blocked:
  assumes  $\bigwedge r a b. r \in \text{nodes} \implies a \in \text{psen } p s - \{a. \text{en } a s\} \implies b \in \{a. \text{en } a r\} - \text{pac } p \implies$ 
     $\text{en } a (\text{ex } b r) \implies \text{en } a r$ 
  assumes  $s \in \text{nodes}$   $\text{path } w s$   $\text{pac } p \cap \{a. \text{en } a s\} \cap \text{set } w = \{\}$ 
  shows  $\text{pac } p \cap \text{set } w = \{\}$ 
  using words-fin-blocked en-fin-word assms by metis

```

```

abbreviation  $\text{proc } a \equiv \{p. a \in \text{pac } p\}$ 
abbreviation  $\text{Proc } A \equiv \bigcup a \in A. \text{proc } a$ 

```

```

lemma psen-simple:
  assumes  $\text{Proc } (\text{psen } p s) = \{p\}$ 
  assumes  $\bigwedge r a b. r \in \text{nodes} \implies a \in \text{psen } p s - \{a. \text{en } a s\} \implies \text{en } b r \implies$ 
     $\text{proc } a \cap \text{proc } b = \{\} \implies \text{en } a (\text{ex } b r) \implies \text{en } a r$ 
  shows  $\bigwedge r a b. r \in \text{nodes} \implies a \in \text{psen } p s - \{a. \text{en } a s\} \implies b \in \{a. \text{en } a r\} - \text{pac } p \implies$ 
     $\text{en } a (\text{ex } b r) \implies \text{en } a r$ 
  using assms by force

```

end

end

References

- [1] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 241–257. Springer Berlin Heidelberg, 1996.
- [2] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.