

Partial Order Reduction

Julian Brunner

September 23, 2021

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	4
4	Sets	6
5	Basics	10
5.1	Types	10
5.2	Rules	10
5.3	Constants	11
5.4	Theorems for @termcurry and @termsplit	13
6	Relations	14
7	Transition Systems	15
8	Trace Theory	19
9	Transition Systems and Trace Theory	27
10	Functions	29
11	Extended Natural Numbers	30

12 Chain-Complete Partial Orders	31
13 Sets and Extended Natural Numbers	33
14 Coinductive Lists	37
14.1 Index Sets	40
14.2 Selections	41
15 Prefixes on Coinductive Lists	44
16 Stuttering	44
17 Interpreted Transition Systems and Traces	46
18 Abstract Theory of Ample Set Partial Order Reduction	48
19 LTL Formulae	52
20 Correctness Theorem of Partial Order Reduction	52
21 Static Analysis for Partial Order Reduction	53

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  $a \# u \leq b \# v$ 
  <proof>
lemma prefix-finE-item[elim!]:
  assumes  $a \# u \leq b \# v$ 
  obtains  $a = b$   $u \leq v$ 
  <proof>

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

```

<proof>

end

2 Lists

theory *List-Extensions*
imports *HOL-Library.Sublist*
begin

declare *remove1-idem*[*simp*]

lemma *nth-append-simps*[*simp*]:

$i < \text{length } xs \implies (xs @ ys) ! i = xs ! i$

$i \geq \text{length } xs \implies (xs @ ys) ! i = ys ! (i - \text{length } xs)$

<proof>

notation *zip* (**infixr** || 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>

lemma *singleton-list-lengthE*[*elim*]:

assumes $\text{length } xs = 1$

obtains x

where $xs = [x]$

<proof>

lemma *singleton-hd-last*: $\text{length } xs = 1 \implies \text{hd } xs = \text{last } xs$ *<proof>*

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

assumes $\bigwedge i. i < \text{length } xs \implies xs ! i \in S$

shows $\text{set } xs \subseteq S$

<proof>

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

assumes $n \neq 0 \text{ } xs \neq []$

shows $\text{hd } (\text{take } n \text{ } xs) = \text{hd } xs$

<proof>

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

assumes $n < \text{length } xs$

shows $\text{hd } (\text{drop } n \text{ } xs) = xs ! n$

<proof>

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

assumes $n < \text{length } xs$

shows $\text{last } (\text{take } (\text{Suc } n) \text{ } xs) = xs ! n$

<proof>

lemma *split-list-first-unique*:

assumes $u_1 @ [a] @ u_2 = v_1 @ [a] @ v_2$ $a \notin \text{set } u_1$ $a \notin \text{set } v_1$

shows $u_1 = v_1$

<proof>

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** \leq_{FI} 50)

where $u \leq_{FI} v \equiv \exists w. u @- w = v$

lemma *prefix-fininfI[intro]*:

assumes $u @- w = v$

shows $u \leq_{FI} v$

<proof>

lemma *prefix-fininfE[elim]*:

assumes $u \leq_{FI} v$

obtains w

where $v = u @- w$

<proof>

lemma *prefix-fininfI-empty[intro!]*: $[] \leq_{FI} w$ *<proof>*

lemma *prefix-fininfI-item[intro!]*:

assumes $a = b$ $u \leq_{FI} v$

shows $a \# u \leq_{FI} b \#\# v$

<proof>

lemma *prefix-fininfE-item[elim!]*:

assumes $a \# u \leq_{FI} b \#\# v$

obtains $a = b$ $u \leq_{FI} v$

<proof>

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

lemma *prefix-fininf-list[simp]*: $w @ u \leq_{FI} w @- v \longleftrightarrow u \leq_{FI} v$ *<proof>*

lemma *prefix-fininf-conc[intro]*: $u \leq_{FI} u @- v$ *<proof>*

lemma *prefix-fininf-prefix[intro]*: *stake* k $w \leq_{FI} w$ *<proof>*

lemma *prefix-fininf-set-range[dest]*: $u \leq_{FI} v \implies \text{set } u \subseteq \text{sset } v$ *<proof>*

lemma *prefix-fininf-absorb*:

assumes $u \leq_{FI} v @- w$ $\text{length } u \leq \text{length } v$

shows $u \leq v$
<proof>

lemma *prefix-fininf-extend*:

assumes $u \leq_{FI} v @- w$ $length\ v \leq length\ u$
shows $v \leq u$

<proof>

lemma *prefix-fininf-length*:

assumes $u \leq_{FI} w$ $v \leq_{FI} w$ $length\ u \leq length\ v$
shows $u \leq v$

<proof>

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>

lemma *prefix-fin-prefix-fininf-trans*[*trans, intro*]: $u \leq v \implies v \leq_{FI} w \implies u \leq_{FI} w$

<proof>

lemma *prefix-finE-nth*:

assumes $u \leq v$ $i < length\ u$
shows $u ! i = v ! i$

<proof>

lemma *prefix-fininfI-nth*:

assumes $\bigwedge i. i < length\ u \implies u ! i = w !! i$
shows $u \leq_{FI} w$

<proof>

definition *chain* :: $(nat \Rightarrow 'a\ list) \Rightarrow bool$

where $chain\ w \equiv mono\ w \wedge (\forall k. \exists l. k < length\ (w\ l))$

definition *limit* :: $(nat \Rightarrow 'a\ list) \Rightarrow 'a\ stream$

where $limit\ w \equiv smap\ (\lambda k. w\ (SOME\ l. k < length\ (w\ l))\ !\ k)\ nats$

lemma *chainI*[*intro?*]:

assumes $mono\ w$
assumes $\bigwedge k. \exists l. k < length\ (w\ l)$
shows $chain\ w$

<proof>

lemma *chainD-mono*[*dest?*]:

assumes $chain\ w$
shows $mono\ w$

<proof>

lemma *chainE-length*[*elim?*]:

assumes $chain\ w$
obtains l
where $k < length\ (w\ l)$

<proof>

lemma *chain-prefix-limit*:

assumes *chain w*

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

<proof>

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

obtains *Q*

where $\wedge k. P k (Q k) \text{ chain } (f \circ Q)$

<proof>

lemma *chain-construct-2*:

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

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 $\wedge k. P k (Q k) \text{ chain } (f \circ Q) \text{ chain } (g \circ Q)$

<proof>

lemma *chain-construct-2'*:

assumes $P 0 u_0 v_0 \wedge k u v. P k u v \implies \exists u' v'. P (\text{Suc } k) u' v' \wedge u \leq u' \wedge v \leq v'$

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 $\wedge k. P k (u k) (v k) \text{ chain } u \text{ chain } v$

<proof>

end

4 Sets

theory *Set-Extensions*

imports

HOL-Library.Infinite-Set

begin

declare *finite-subset[intro]*

lemma *set-not-emptyI[intro 0]*: $x \in S \implies S \neq \{\}$ *<proof>*

lemma *sets-empty-iffI[intro 0]*:

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

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

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

<proof>

lemma *disjointI[intro 0]*:

assumes $\wedge x. x \in A \implies x \in B \implies \text{False}$

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

<proof>

lemma *range-subsetI[intro 0]*:

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

shows $\text{range } f \subseteq S$

$\langle \text{proof} \rangle$

lemma *finite-imageI-range*:

assumes *finite* (*range f*)

shows *finite* (*f* ' *A*)

$\langle \text{proof} \rangle$

lemma *inf-img-fin-domE'*:

assumes *infinite* *A*

assumes *finite* (*f* ' *A*)

obtains *y*

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

$\langle \text{proof} \rangle$

lemma *vimage-singleton[simp]*: f - ' $\{y\} = \{x. f\ x = y\}$ $\langle \text{proof} \rangle$

lemma *these-alt-def*: *Option.these* *S* = *Some* - ' *S* $\langle \text{proof} \rangle$

lemma *the-vimage-subset*: *the* - ' $\{a\} \subseteq \{None, Some\ a\}$ $\langle \text{proof} \rangle$

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

assumes *finite* *S*

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

shows $P\ S$

$\langle \text{proof} \rangle$

lemma *zero-not-in-Suc-image[simp]*: $0 \notin \text{Suc}$ ' *A* $\langle \text{proof} \rangle$

lemma *Collect-split-Suc*:

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

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

$\langle \text{proof} \rangle$

lemma *Collect-subsume[simp]*:

assumes $\bigwedge x. x \in A \implies P\ x$

shows $\{x \in A. P\ x\} = A$

$\langle \text{proof} \rangle$

lemma *Max-ge'*:

assumes *finite* *A* $A \neq \{\}$

assumes $b \in A$ $a \leq b$

shows $a \leq \text{Max}\ A$

$\langle \text{proof} \rangle$

abbreviation *least* *A* $\equiv \text{LEAST}\ k. k \in A$

lemma *least-contains[intro?, simp]*:

fixes *A* :: 'a :: *wellorder set*

assumes $k \in A$

shows *least* *A* $\in A$

<proof>
lemma *least-contains'*[*intro?*, *simp*]:
 fixes $A :: 'a :: \text{wellorder set}$
 assumes $A \neq \{\}$
 shows $\text{least } A \in A$
 <proof>

lemma *least-least*[*intro?*, *simp*]:
 fixes $A :: 'a :: \text{wellorder set}$
 assumes $k \in A$
 shows $\text{least } A \leq k$
 <proof>

lemma *least-unique*:
 fixes $A :: 'a :: \text{wellorder set}$
 assumes $k \in A$ $k \leq \text{least } A$
 shows $k = \text{least } A$
 <proof>

lemma *least-not-less*:
 fixes $A :: 'a :: \text{wellorder set}$
 assumes $k < \text{least } A$
 shows $k \notin A$
 <proof>

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

lemma *least-singleton*[*simp*]:
 fixes $a :: 'a :: \text{wellorder}$
 shows $\text{least } \{a\} = a$
 <proof>

lemma *least-image*[*simp*]:
 fixes $f :: 'a :: \text{wellorder} \Rightarrow 'b :: \text{wellorder}$
 assumes $A \neq \{\}$ $\bigwedge k l. k \in A \implies l \in A \implies k \leq l \implies f k \leq f l$
 shows $\text{least } (f ` A) = f (\text{least } A)$
 <proof>

lemma *least-le*:
 fixes $A B :: 'a :: \text{wellorder set}$
 assumes $B \neq \{\}$
 assumes $\bigwedge i. i \leq \text{least } A \implies i \leq \text{least } B \implies i \in B \implies i \in A$
 shows $\text{least } A \leq \text{least } B$
 <proof>

lemma *least-eq*:
 fixes $A B :: 'a :: \text{wellorder set}$
 assumes $A \neq \{\}$ $B \neq \{\}$
 assumes $\bigwedge i. i \leq \text{least } A \implies i \leq \text{least } B \implies i \in A \longleftrightarrow i \in B$
 shows $\text{least } A = \text{least } B$

$\langle proof \rangle$

lemma *least-Suc[simp]*:

assumes $A \neq \{\}$

shows $least (Suc \text{ ` } A) = Suc (least A)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

fixes $A :: 'a :: wellorder set$

assumes $finite A \ A - \{least A\} \neq \{\}$

shows $Max (A - \{least A\}) = Max A$

$\langle proof \rangle$

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

fixes $A :: nat set$

assumes $x \in A \ y \in A \ card \{z \in A. z < x\} = card \{z \in A. z < y\}$

shows $x = y$

$\langle proof \rangle$

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

fixes $A :: nat set$

assumes $x \in A \ y \in A \ card \{z \in A. z \leq x\} = card \{z \in A. z \leq y\}$

shows $x = y$

$\langle proof \rangle$

lemma *nat-set-card-mono[simp]*:

fixes $A :: nat set$

assumes $x \in A$

shows $card \{z \in A. z < x\} < card \{z \in A. z < y\} \longleftrightarrow x < y$

$\langle proof \rangle$

lemma *card-one[elim]*:

assumes $card A = 1$

obtains a

where $A = \{a\}$

$\langle proof \rangle$

lemma *image-alt-def*: $f \text{ ` } A = \{f x \mid x. x \in A\}$ $\langle proof \rangle$

lemma *supset-mono-inductive[mono]*:

assumes $\bigwedge x. x \in B \longrightarrow x \in C$

shows $A \subseteq B \longrightarrow A \subseteq C$

$\langle proof \rangle$

lemma *Collect-mono-inductive[mono]*:

assumes $\bigwedge x. P x \longrightarrow Q x$

shows $x \in \{x. P x\} \longrightarrow x \in \{x. Q x\}$

<proof>

lemma *image-union-split*:

assumes $f \text{ ' } (A \cup B) = g \text{ ' } C$

obtains $D E$

where $f \text{ ' } A = g \text{ ' } D$ $f \text{ ' } B = g \text{ ' } E$ $D \subseteq C$ $E \subseteq C$

<proof>

lemma *image-insert-split*:

assumes $\text{inj } g$ $f \text{ ' } \text{insert } a B = g \text{ ' } C$

obtains $d E$

where $f a = g d$ $f \text{ ' } B = g \text{ ' } E$ $d \in C$ $E \subseteq C$

<proof>

end

5 Basics

theory *Basic-Extensions*

imports *HOL-Library.Infinite-Set*

begin

5.1 Types

type-synonym $\text{'a step} = \text{'a} \Rightarrow \text{'a}$

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 $(x, y) \in \text{Id}$

<proof>

lemma (**in** *order*) *order-le-cases*:

assumes $x \leq y$

obtains $(\text{eq}) x = y \mid (\text{lt}) x < y$

<proof>

lemma (**in** *linorder*) *linorder-cases'*:

obtains $(\text{le}) x \leq y \mid (\text{gt}) x > y$

<proof>

lemma *monoI-comp*[*intro*]:

assumes $\text{mono } f$ $\text{mono } g$

shows *mono* ($f \circ g$)
 ⟨*proof*⟩
lemma *strict-monoI-comp*[*intro*]:
assumes *strict-mono* f *strict-mono* g
shows *strict-mono* ($f \circ g$)
 ⟨*proof*⟩

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

lemma *INFM-Suc*[*simp*]: $(\exists_{\infty} i. P (Suc\ i)) \longleftrightarrow (\exists_{\infty} i. P\ i)$
 ⟨*proof*⟩
lemma *INFM-plus*[*simp*]: $(\exists_{\infty} i. P (i + n :: \text{nat})) \longleftrightarrow (\exists_{\infty} i. P\ i)$
 ⟨*proof*⟩
lemma *INFM-minus*[*simp*]: $(\exists_{\infty} i. P (i - n :: \text{nat})) \longleftrightarrow (\exists_{\infty} i. P\ i)$
 ⟨*proof*⟩

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*]: *const* $x\ y = x$ ⟨*proof*⟩
lemma *const2-apply*[*simp*]: *const2* $x\ y\ z = x$ ⟨*proof*⟩
lemma *const3-apply*[*simp*]: *const3* $x\ y\ z\ u = x$ ⟨*proof*⟩
lemma *const4-apply*[*simp*]: *const4* $x\ y\ z\ u\ v = x$ ⟨*proof*⟩
lemma *const5-apply*[*simp*]: *const5* $x\ y\ z\ u\ v\ w = x$ ⟨*proof*⟩

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

lemma *zip-fun-simps*[*simp*]:
 $(f\ ||\ g)\ x = (f\ x, g\ x)$
 $fst \circ (f\ ||\ g) = f$
 $snd \circ (f\ ||\ g) = g$
 $fst \circ h\ ||\ snd \circ h = h$
 $fst\ ' \text{range}\ (f\ ||\ g) = \text{range}\ f$
 $snd\ ' \text{range}\ (f\ ||\ g) = \text{range}\ g$
 ⟨*proof*⟩

lemma *zip-fun-eq*[*dest*]:
assumes $f \parallel g = h \parallel i$
shows $f = h \ g = i$
 \langle *proof* \rangle

lemma *zip-fun-range-subset*[*intro, simp*]: $\text{range } (f \parallel g) \subseteq \text{range } f \times \text{range } g$
 \langle *proof* \rangle

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

lemma *zip-fun-split*:
obtains $f \ g$
where $h = f \parallel g$
 \langle *proof* \rangle

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

lemma *inj-Some2*[*simp, intro*]:
inj None-Some
inj Some-None
inj Some-Some
 \langle *proof* \rangle

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
 \langle *proof* \rangle

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

where $swap\ x \equiv (snd\ x, fst\ x)$

lemma $swap-simps[simp]$: $swap\ (a, b) = (b, a)$ $\langle proof \rangle$

lemma $swap-inj[intro, simp]$: $inj\ swap$ $\langle proof \rangle$

lemma $swap-surj[intro, simp]$: $surj\ swap$ $\langle proof \rangle$

lemma $swap-bij[intro, simp]$: $bij\ swap$ $\langle proof \rangle$

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

where $push\ x \equiv (fst\ (fst\ x), snd\ (fst\ x), snd\ x)$

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

where $pull\ x \equiv ((fst\ x, fst\ (snd\ x)), snd\ (snd\ x))$

lemma $push-simps[simp]$: $push\ ((x, y), z) = (x, y, z)$ $\langle proof \rangle$

lemma $pull-simps[simp]$: $pull\ (x, y, z) = ((x, y), z)$ $\langle proof \rangle$

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

where $label \equiv fst \circ snd$

lemma $label-select[simp]$: $label\ (p, a, q) = a$ $\langle proof \rangle$

5.4 Theorems for @termcurry and @termsplit

lemma $curry-split[simp]$: $curry \circ case-prod = id$ $\langle proof \rangle$

lemma $split-curry[simp]$: $case-prod \circ curry = id$ $\langle proof \rangle$

lemma $curry-le[simp]$: $curry\ f \leq curry\ g \iff f \leq g$ $\langle proof \rangle$

lemma $split-le[simp]$: $case-prod\ f \leq case-prod\ g \iff f \leq g$ $\langle proof \rangle$

lemma $mono-curry-left[simp]$: $mono\ (curry \circ h) \iff mono\ h$
 $\langle proof \rangle$

lemma $mono-split-left[simp]$: $mono\ (case-prod \circ h) \iff mono\ h$
 $\langle proof \rangle$

lemma $mono-curry-right[simp]$: $mono\ (h \circ curry) \iff mono\ h$
 $\langle proof \rangle$

lemma $mono-split-right[simp]$: $mono\ (h \circ case-prod) \iff mono\ h$
 $\langle proof \rangle$

lemma $Collect-curry[simp]$: $\{x. P\ (curry\ x)\} = case-prod\ ' \{x. P\ x\}$ $\langle proof \rangle$

lemma $Collect-split[simp]$: $\{x. P\ (case-prod\ x)\} = curry\ ' \{x. P\ x\}$ $\langle proof \rangle$

lemma $gfp-split-curry[simp]$: $gfp\ (case-prod \circ f \circ curry) = case-prod\ (gfp\ f)$
 $\langle proof \rangle$

lemma $gfp-curry-split[simp]$: $gfp\ (curry \circ f \circ case-prod) = curry\ (gfp\ f)$
 $\langle proof \rangle$

lemma $not-someI$:

assumes $\bigwedge x. P\ x \implies False$

shows $\neg P\ (SOME\ x. P\ x)$

$\langle proof \rangle$

lemma *some-ccontr*:
assumes $(\bigwedge x. \neg P x) \implies False$
shows $P (SOME x. P x)$
 $\langle proof \rangle$

end

6 Relations

theory *Relation-Extensions*

imports

Basic-Extensions

begin

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

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

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

lemma *liftablepI*[*intro*]:
assumes $\bigwedge x\ y. r\ x\ y \implies r\ (f\ x)\ (f\ y)$
shows $liftablep\ r\ f$
 $\langle proof \rangle$

lemma *liftablepE*[*elim*]:
assumes $liftablep\ r\ f$
assumes $r\ x\ y$
obtains $r\ (f\ x)\ (f\ y)$
 $\langle proof \rangle$

lemma *liftablep-rtranclp*:
assumes $liftablep\ r\ f$
shows $liftablep\ r^{**}\ f$
 $\langle proof \rangle$

definition *confluentp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $confluentp\ r \equiv \forall x\ y1\ y2. r^{**}\ x\ y1 \longrightarrow r^{**}\ x\ y2 \longrightarrow (\exists z. r^{**}\ y1\ z \wedge r^{**}\ y2\ z)$

lemma *confluentpI*[*intro*]:
assumes $\bigwedge x\ y1\ y2. r^{**}\ x\ y1 \implies r^{**}\ x\ y2 \implies \exists z. r^{**}\ y1\ z \wedge r^{**}\ y2\ z$
shows $confluentp\ r$
 $\langle proof \rangle$

lemma *confluentpE*[*elim*]:
assumes $confluentp\ r$
assumes $r^{**}\ x\ y1\ r^{**}\ x\ y2$

obtains z
where $r^{**} \ y1 \ z \ r^{**} \ y2 \ z$
 $\langle proof \rangle$

lemma *confluentpI*[*intro*]:
assumes $\bigwedge x \ y1 \ y2. r^{**} \ x \ y1 \ \Longrightarrow \ r \ x \ y2 \ \Longrightarrow \ \exists z. r^{**} \ y1 \ z \ \wedge \ r^{**} \ y2 \ z$
shows *confluentp* r
 $\langle proof \rangle$

lemma *transclp-eq-implies-confluent-imp*:
assumes $r1^{**} = r2^{**}$
assumes *confluentp* $r1$
shows *confluentp* $r2$
 $\langle proof \rangle$

lemma *transclp-eq-implies-confluent-eq*:
assumes $r1^{**} = r2^{**}$
shows *confluentp* $r1 \longleftrightarrow \text{confluentp } r2$
 $\langle proof \rangle$

definition *diamondp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *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 *diamondpI*[*intro*]:
assumes $\bigwedge x \ y1 \ y2. r \ x \ y1 \ \Longrightarrow \ r \ x \ y2 \ \Longrightarrow \ \exists z. r \ y1 \ z \ \wedge \ r \ y2 \ z$
shows *diamondp* r
 $\langle proof \rangle$

lemma *diamondpE*[*elim*]:
assumes *diamondp* r
assumes $r \ x \ y1 \ r \ x \ y2$
obtains z
where $r \ y1 \ z \ r \ y2 \ z$
 $\langle proof \rangle$

lemma *diamondp-implies-confluentp*:
assumes *diamondp* r
shows *confluentp* r
 $\langle proof \rangle$

locale *wellfounded-relation* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
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  $\in$  cycles p
       $\langle$ proof $\rangle$ 
    lemma cyclesE[elim!]:
      assumes w  $\in$  cycles p
      obtains path w p target w p = p
       $\langle$ proof $\rangle$ 

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

    lemma executableI-step[intro!]:
      assumes p  $\in$  nodes enabled a p
      shows a  $\in$  executable
       $\langle$ proof $\rangle$ 
    lemma executableI-words-fin[intro!]:
      assumes p  $\in$  nodes path w p
      shows set w  $\subseteq$  executable
       $\langle$ proof $\rangle$ 
    lemma executableE[elim?]:
      assumes a  $\in$  executable
      obtains p
      where p  $\in$  nodes enabled a p
       $\langle$ proof $\rangle$ 

  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* $\equiv \{a. \exists q. en\ a\ q \wedge int\ q \neq int\ (ex\ a\ q)\}$

lemma *visibleI*[*intro*]:
assumes *en a q int q \neq int (ex a q)*
shows *a \in visible*
 $\langle proof \rangle$

lemma *visibleE*[*elim*]:
assumes *a \in visible*
obtains *q*
where *en a q int q \neq int (ex a q)*
 $\langle proof \rangle$

abbreviation *invisible* $\equiv -\ visible$

lemma *execute-fin-word-invisible*:
assumes *path w p set w \subseteq invisible*
shows *int (target w p) = int p*
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *languageE*[*elim!*]:
assumes *w \in language*
obtains *p v*
where *w = smap int (p ## trace v p) init p run v p*
 $\langle proof \rangle$

```

end

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

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

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

  lemma scut-reachable:
    assumes scut p q
    shows p  $\in$  nodes q  $\in$  nodes
     $\langle$ proof $\rangle$ 

  lemma scut-transcl:
    assumes scut++ p q
    obtains w
    where path w p target w p = q set w  $\cap$  cuts = {} w  $\neq$  []
     $\langle$ proof $\rangle$ 

  sublocale wellfounded-relation scut-1-1
   $\langle$ proof $\rangle$ 

  lemma no-cut-scut:
    assumes p  $\in$  nodes en a p a  $\notin$  cuts
    shows scut-1-1 (ex a p) p
     $\langle$ proof $\rangle$ 

end

locale transition-system-sticky =
  transition-system-complete ex en init int +

```

```

transition-system-cut ex en init sticky
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 sticky :: 'action set
+
assumes executable-visible-sticky: executable  $\cap$  visible  $\subseteq$  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  $\implies$  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  $=_S$  50)
  where swap: ind a b  $\implies$  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  $\implies$  w  $=_S$  v <proof>

lemma eq-swap-length[dest]: w1  $=_S$  w2  $\implies$  length w1 = length w2 <proof>
lemma eq-swap-range[dest]: w1  $=_S$  w2  $\implies$  set w1 = set w2 <proof>

lemma eq-swap-extend:
  assumes w1  $=_S$  w2
  shows u @ w1 @ v  $=_S$  u @ w2 @ v
  <proof>

lemma eq-swap-remove1:
  assumes w1  $=_S$  w2
  obtains (equal) remove1 c w1 = remove1 c w2 | (swap) remove1 c w1  $=_S$ 
remove1 c w2
  <proof>

lemma eq-swap-rev:
  assumes w1  $=_S$  w2

```

shows $rev\ w_1 =_S rev\ w_2$
 $\langle proof \rangle$

abbreviation $eq\text{-}fin :: 'item\ list \Rightarrow 'item\ list \Rightarrow bool$ (**infix** $=_F$ 50)
where $eq\text{-}fin \equiv eq\text{-}swap^{**}$

lemma $eq\text{-}fin\text{-}symp[intro, sym]: u =_F v \Longrightarrow v =_F u$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}length[dest]: w_1 =_F w_2 \Longrightarrow length\ w_1 = length\ w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}range[dest]: w_1 =_F w_2 \Longrightarrow set\ w_1 = set\ w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}remove1$:
assumes $w_1 =_F w_2$
shows $remove1\ c\ w_1 =_F remove1\ c\ w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}rev$:
assumes $w_1 =_F w_2$
shows $rev\ w_1 =_F rev\ w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}concat\text{-}eq\text{-}fin\text{-}start$:
assumes $u @ v_1 =_F u @ v_2$
shows $v_1 =_F v_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}concat$: $u @ w_1 @ v =_F u @ w_2 @ v \longleftrightarrow w_1 =_F w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}concat\text{-}start[iff]$: $w @ w_1 =_F w @ w_2 \longleftrightarrow w_1 =_F w_2$
 $\langle proof \rangle$

lemma $eq\text{-}fin\text{-}concat\text{-}end[iff]$: $w_1 @ w =_F w_2 @ w \longleftrightarrow w_1 =_F w_2$
 $\langle proof \rangle$

lemma $ind\text{-}eq\text{-}fin'$:
assumes $Ind\ \{a\}\ (set\ v)$
shows $[a] @ v =_F v @ [a]$
 $\langle proof \rangle$

lemma $ind\text{-}eq\text{-}fin[intro]$:
assumes $Ind\ (set\ u)\ (set\ v)$
shows $u @ v =_F v @ u$
 $\langle proof \rangle$

definition $le\text{-}fin :: 'item\ list \Rightarrow 'item\ list \Rightarrow bool$ (**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 $w_1 \preceq_F w_2$
 $\langle proof \rangle$

lemma *le-finE*[*elim 0*]:
assumes $w_1 \preceq_F w_2$
obtains v_1
where $w_1 @ v_1 =_F w_2$
 $\langle proof \rangle$

lemma *le-fin-empty*[*simp*]: $[] \preceq_F w \langle proof \rangle$

lemma *le-fin-trivial*[*intro*]: $w_1 =_F w_2 \implies w_1 \preceq_F w_2$
 $\langle proof \rangle$

lemma *le-fin-length*[*dest*]: $w_1 \preceq_F w_2 \implies length\ w_1 \leq length\ w_2 \langle proof \rangle$

lemma *le-fin-range*[*dest*]: $w_1 \preceq_F w_2 \implies set\ w_1 \subseteq set\ w_2 \langle proof \rangle$

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

lemma *le-fin-reflp*[*simp, intro*]: $w \preceq_F w \langle proof \rangle$

lemma *le-fin-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2\ w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *eq-fin-le-fin-transp*[*intro, trans*]:
assumes $w_1 =_F w_2\ w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *le-fin-eq-fin-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2\ w_2 =_F w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *prefix-le-fin-transp*[*intro, trans*]:
assumes $w_1 \leq w_2\ w_2 \preceq_F w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *le-fin-prefix-transp*[*intro, trans*]:
assumes $w_1 \preceq_F w_2\ w_2 \leq w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *prefix-eq-fin-transp*[*intro, trans*]:
assumes $w_1 \leq w_2\ w_2 =_F w_3$
shows $w_1 \preceq_F w_3$
 $\langle proof \rangle$

lemma *le-fin-concat-start*[*iff*]: $w @ w_1 \preceq_F w @ w_2 \longleftrightarrow w_1 \preceq_F w_2$
 $\langle proof \rangle$

lemma *le-fin-concat-end*[*dest*]:

assumes $w_1 \preceq_F w_2$
shows $w_1 \preceq_F w_2 @ w$
 $\langle proof \rangle$

definition $le\text{-fininf} :: 'item\ list \Rightarrow 'item\ stream \Rightarrow bool$ (**infix** \preceq_{FI} 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 $le\text{-fininfI}[intro\ 0]$:
assumes $v_2 \leq_{FI} w_2\ w_1 \preceq_F v_2$
shows $w_1 \preceq_{FI} w_2$
 $\langle proof \rangle$

lemma $le\text{-fininfE}[elim\ 0]$:
assumes $w_1 \preceq_{FI} w_2$
obtains v_2
where $v_2 \leq_{FI} w_2\ w_1 \preceq_F v_2$
 $\langle proof \rangle$

lemma $le\text{-fininf-empty}[simp]$: $[] \preceq_{FI} w$ $\langle proof \rangle$

lemma $le\text{-fininf-range}[dest]$: $w_1 \preceq_{FI} w_2 \Longrightarrow set\ w_1 \subseteq sset\ w_2$ $\langle proof \rangle$

lemma $eq\text{-fin-le-fininf-transp}[intro,\ trans]$:
assumes $w_1 =_F w_2\ w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
 $\langle proof \rangle$

lemma $le\text{-fin-le-fininf-transp}[intro,\ trans]$:
assumes $w_1 \preceq_F w_2\ w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
 $\langle proof \rangle$

lemma $prefix\text{-le-fininf-transp}[intro,\ trans]$:
assumes $w_1 \leq w_2\ w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
 $\langle proof \rangle$

lemma $le\text{-fin-prefix-fininf-transp}[intro,\ trans]$:
assumes $w_1 \preceq_F w_2\ w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
 $\langle proof \rangle$

lemma $eq\text{-fin-prefix-fininf-transp}[intro,\ trans]$:
assumes $w_1 =_F w_2\ w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
 $\langle proof \rangle$

lemma $le\text{-fininf-concat-start}[iff]$: $w @ w_1 \preceq_{FI} w @- w_2 \longleftrightarrow w_1 \preceq_{FI} w_2$
 $\langle proof \rangle$

lemma $le\text{-fininf-singleton}[intro,\ simp]$: $[shd\ v] \preceq_{FI} v$
 $\langle proof \rangle$

definition $le\text{-inf} :: 'item\ stream \Rightarrow 'item\ stream \Rightarrow bool$ (**infix** \preceq_I 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 $\bigwedge v_1. v_1 \leq_{FI} w_1 \implies v_1 \preceq_{FI} w_2$

shows $w_1 \preceq_I w_2$

<proof>

lemma *le-infE*[*elim 0*]:

assumes $w_1 \preceq_I w_2 \quad v_1 \leq_{FI} w_1$

obtains $v_1 \preceq_{FI} w_2$

<proof>

lemma *le-inf-range*[*dest*]:

assumes $w_1 \preceq_I w_2$

shows $sset\ w_1 \subseteq sset\ w_2$

<proof>

lemma *le-inf-reflp*[*simp, intro*]: $w \preceq_I w$ *<proof>*

lemma *prefix-fininf-le-inf-transp*[*intro, trans*]:

assumes $w_1 \leq_{FI} w_2 \quad w_2 \preceq_I w_3$

shows $w_1 \preceq_{FI} w_3$

<proof>

lemma *le-fininf-le-inf-transp*[*intro, trans*]:

assumes $w_1 \preceq_{FI} w_2 \quad w_2 \preceq_I w_3$

shows $w_1 \preceq_{FI} w_3$

<proof>

lemma *le-inf-transp*[*intro, trans*]:

assumes $w_1 \preceq_I w_2 \quad w_2 \preceq_I w_3$

shows $w_1 \preceq_I w_3$

<proof>

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>

lemma *le-infI-chain-left*:

assumes $\text{chain}\ w \wedge k. w\ k \preceq_{FI} v$

shows $\text{limit}\ w \preceq_I v$

<proof>

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>

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>

definition *eq-inf* :: *'item stream* \Rightarrow *'item stream* \Rightarrow *bool* (**infix** =_I 50)

where $w_1 =_I w_2 \equiv w_1 \preceq_I w_2 \wedge w_2 \preceq_I w_1$

lemma *eq-infI*[*intro 0*]:
assumes $w_1 \preceq_I w_2$ $w_2 \preceq_I w_1$
shows $w_1 =_I w_2$
<proof>

lemma *eq-infE*[*elim 0*]:
assumes $w_1 =_I w_2$
obtains $w_1 \preceq_I w_2$ $w_2 \preceq_I w_1$
<proof>

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

lemma *eq-inf-reflp*[*simp, intro*]: $w =_I w$ *<proof>*

lemma *eq-inf-symp*[*intro*]: $w_1 =_I w_2 \implies w_2 =_I w_1$ *<proof>*

lemma *eq-inf-transp*[*intro, trans*]:
assumes $w_1 =_I w_2$ $w_2 =_I w_3$
shows $w_1 =_I w_3$
<proof>

lemma *le-fininf-eq-inf-transp*[*intro, trans*]:
assumes $w_1 \preceq_{FI} w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_{FI} w_3$
<proof>

lemma *le-inf-eq-inf-transp*[*intro, trans*]:
assumes $w_1 \preceq_I w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_I w_3$
<proof>

lemma *eq-inf-le-inf-transp*[*intro, trans*]:
assumes $w_1 =_I w_2$ $w_2 \preceq_I w_3$
shows $w_1 \preceq_I w_3$
<proof>

lemma *prefix-fininf-eq-inf-transp*[*intro, trans*]:
assumes $w_1 \preceq_{FI} w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_{FI} w_3$
<proof>

lemma *le-inf-concat-start*[*iff*]: $w @- w_1 \preceq_I w @- w_2 \iff w_1 \preceq_I w_2$
<proof>

lemma *eq-fin-le-inf-concat-end*[*dest*]: $w_1 =_F w_2 \implies w_1 @- w \preceq_I w_2 @- w$
<proof>

lemma *eq-inf-concat-start*[*iff*]: $w @- w_1 =_I w @- w_2 \iff w_1 =_I w_2$ *<proof>*

lemma *eq-inf-concat-end*[*dest*]: $w_1 =_F w_2 \implies w_1 @- w =_I w_2 @- w$
<proof>

lemma *le-fininf-suffixI*[*intro*]:
assumes $w =_I w_1 @- w_2$
shows $w_1 \preceq_{FI} w$
<proof>

lemma *le-fininf-suffixE[elim]*:

assumes $w_1 \preceq_{FI} w$

obtains w_2

where $w =_I w_1 @- w_2$

<proof>

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>

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>

definition *indoc* :: 'item \Rightarrow 'item list \Rightarrow bool

where $indoc\ a\ u \equiv \exists\ u_1\ u_2. u = u_1 @ [a] @ u_2 \wedge a \notin set\ u_1 \wedge Ind\ \{a\}\ (set\ u_1)$

lemma *indoc-set*: $indoc\ a\ u \implies a \in set\ u$ *<proof>*

lemma *indoc-appendI1[intro]*:

assumes $indoc\ a\ u$

shows $indoc\ a\ (u @ v)$

<proof>

lemma *indoc-appendI2[intro]*:

assumes $a \notin set\ u$ $Ind\ \{a\}\ (set\ u)$ $indoc\ a\ v$

shows $indoc\ a\ (u @ v)$

<proof>

lemma *indoc-appendE[elim!]*:

assumes $indoc\ a\ (u @ v)$

obtains $(first)\ a \in set\ u\ indoc\ a\ u \mid (second)\ a \notin set\ u\ Ind\ \{a\}\ (set\ u)\ indoc\ a\ v$

<proof>

lemma *indoc-single*: $indoc\ a\ [b] \longleftrightarrow a = b$

<proof>

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

$indoc\ a\ u \vee a \notin set\ u \wedge Ind\ \{a\}\ (set\ u) \wedge indoc\ a\ v$ *<proof>*

lemma *indoc-Nil[simp]*: $indoc\ a\ [] \longleftrightarrow False$ *<proof>*

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

<proof>

lemma *eq-swap-indoc*: $u =_S v \implies indoc\ c\ u \implies indoc\ c\ v$ *<proof>*

lemma *eq-fin-indoc*: $u =_F v \implies indoc\ c\ u \implies indoc\ c\ v$ *<proof>*

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>

lemma *eq-fin-ind*:

assumes $u @ v =_F v @ u$ $\text{set } u \cap \text{set } v = \{\}$
shows $\text{Ind } (\text{set } u) (\text{set } v)$

<proof>

lemma *le-fin-member'*:

assumes $[a] \preceq_F u @ v$ $a \in \text{set } u$
shows $[a] \preceq_F u$

<proof>

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

assumes $[a] \preceq_F u @ v$ $a \notin \text{set } u$
shows $[a] \preceq_F v$

<proof>

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

assumes $[a] \preceq_{FI} u @- v$ $a \notin \text{set } u$
shows $[a] \preceq_{FI} v$

<proof>

lemma *le-fin-ind''*:

assumes $[a] \preceq_F w [b] \preceq_F w$ $a \neq b$
shows $\text{ind } a b$

<proof>

lemma *le-fin-ind'*:

assumes $[a] \preceq_F w v \preceq_F w$ $a \notin \text{set } v$
shows $\text{Ind } \{a\} (\text{set } v)$

<proof>

lemma *le-fininf-ind''*:

assumes $[a] \preceq_{FI} w [b] \preceq_{FI} w$ $a \neq b$
shows $\text{ind } a b$

<proof>

lemma *le-fininf-ind'*:

assumes $[a] \preceq_{FI} w v \preceq_{FI} w$ $a \notin \text{set } v$
shows $\text{Ind } \{a\} (\text{set } v)$

<proof>

lemma *indoc-alt-def*: $\text{indoc } a v \longleftrightarrow v =_F [a] @ \text{remove1 } a v$

<proof>

lemma *levi-lemma*:

assumes $t @ u =_F v @ w$

obtains $p r s q$

where $t =_F p @ r$ $u =_F s @ q$ $v =_F p @ s$ $w =_F r @ q$ $\text{Ind } (\text{set } r) (\text{set } s)$

<proof>

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 $\forall v. v \leq_{FI} w \longrightarrow \text{path } v \ p$

shows *run* *w* *p*

<proof>

lemma (**in** *transition-system*) *words-infI-construct'*:

assumes $\bigwedge k. \exists v. v \leq_{FI} w \wedge k < \text{length } v \wedge \text{path } v \ p$

shows *run* *w* *p*

<proof>

lemma (**in** *transition-system*) *words-infI-construct-chain*[*intro*]:

assumes *chain* *w* $\bigwedge k. \text{path } (w \ k) \ p$

shows *run* (*limit* *w*) *p*

<proof>

lemma (**in** *transition-system*) *words-fin-blocked*:

assumes $\bigwedge w. \text{path } w \ p \implies A \cap \text{set } w = \{\} \implies A \cap \{a. \text{enabled } a \ (\text{target } w \ p)\} \subseteq A \cap \{a. \text{enabled } a \ p\}$

assumes *path* *w* *p* $A \cap \{a. \text{enabled } a \ p\} \cap \text{set } w = \{\}$

shows $A \cap \text{set } w = \{\}$

<proof>

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* \longleftrightarrow *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*)

$\langle proof \rangle$
lemma diamond-right:
assumes $ind\ a\ b$
assumes $en\ a\ p\ en\ b\ (ex\ a\ p)$
shows $en\ a\ (ex\ b\ p)\ en\ b\ p\ ex\ b\ (ex\ a\ p) = ex\ a\ (ex\ b\ p)$
 $\langle proof \rangle$
lemma diamond-left:
assumes $ind\ a\ b$
assumes $en\ a\ (ex\ b\ p)\ en\ b\ p$
shows $en\ a\ p\ en\ b\ (ex\ a\ p)\ ex\ b\ (ex\ a\ p) = ex\ a\ (ex\ b\ p)$
 $\langle proof \rangle$

lemma eq-swap-word:
assumes $w_1 =_S w_2\ path\ w_1\ p$
shows $path\ w_2\ p$
 $\langle proof \rangle$
lemma eq-fin-word:
assumes $w_1 =_F w_2\ path\ w_1\ p$
shows $path\ w_2\ p$
 $\langle proof \rangle$
lemma le-fin-word:
assumes $w_1 \preceq_F w_2\ path\ w_2\ p$
shows $path\ w_1\ p$
 $\langle proof \rangle$
lemma le-fininf-word:
assumes $w_1 \preceq_{FI} w_2\ run\ w_2\ p$
shows $path\ w_1\ p$
 $\langle proof \rangle$
lemma le-inf-word:
assumes $w_2 \preceq_I w_1\ run\ w_1\ p$
shows $run\ w_2\ p$
 $\langle proof \rangle$
lemma eq-inf-word:
assumes $w_1 =_I w_2\ run\ w_1\ p$
shows $run\ w_2\ p$
 $\langle proof \rangle$

lemma eq-swap-execute:
assumes $path\ w_1\ p\ w_1 =_S w_2$
shows $fold\ ex\ w_1\ p = fold\ ex\ w_2\ p$
 $\langle proof \rangle$
lemma eq-fin-execute:
assumes $path\ w_1\ p\ w_1 =_F w_2$
shows $fold\ ex\ w_1\ p = fold\ ex\ w_2\ p$
 $\langle proof \rangle$

lemma diamond-fin-word-step:
assumes $Ind\ \{a\}\ (set\ v)\ en\ a\ p\ path\ v\ p$
shows $path\ v\ (ex\ a\ p)$

```

    <proof>
lemma diamond-inf-word-step:
  assumes Ind {a} (sset w) en a p run w p
  shows run w (ex a p)
  <proof>
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)
  <proof>
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
  <proof>

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 <proof>

    lemma injective-on:
      assumes  $x \in A$   $y \in A$   $f x = f y$ 
      shows  $x = y$ 
      <proof>

```

```

end

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

  definition g  $\equiv$  SOME g. injection A B f g

  sublocale injection A B f g  $\langle$ proof $\rangle$ 

end

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

  lemma image-superset[intro]:  $f ' A \supseteq B$ 
     $\langle$ proof $\rangle$ 

  lemma image-eq[simp]:  $f ' A = B$   $\langle$ proof $\rangle$ 

end

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

  definition g  $\equiv$  SOME g. surjection A B f g

  sublocale surjection A B f g  $\langle$ proof $\rangle$ 

end

locale bijection = injection + surjection

lemma inj-on-bijection:
  assumes inj-on f A
  shows bijection A (f ' A) f (inv-into A f)
   $\langle$ proof $\rangle$ 

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 <proof>

lemma le-epred[iff]: l < epred k  $\longleftrightarrow$  eSuc l < k
  <proof>

lemma eq-infI[intro]:
  assumes  $\bigwedge n. \text{enat } n \leq m$ 
  shows  $m = \infty$ 
  <proof>

```

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

lemma infinite-chain-below[dest]:
  assumes Complete-Partial-Order.chain ord C infinite C x  $\in$  C
  assumes finite {y  $\in$  C. ord x y}
  shows infinite {y  $\in$  C. ord y x}
  <proof>

lemma infinite-chain-above[dest]:
  assumes Complete-Partial-Order.chain ord C infinite C x  $\in$  C
  assumes finite {y  $\in$  C. ord y x}
  shows infinite {y  $\in$  C. ord x y}
  <proof>

lemma (in ccpo) ccpo-Sup-upper-inv:
  assumes Complete-Partial-Order.chain less-eq C x >  $\bigsqcup$  C
  shows x  $\notin$  C
  <proof>

lemma (in ccpo) ccpo-Sup-least-inv:
  assumes Complete-Partial-Order.chain less-eq C  $\bigsqcup$  C > x

```

obtains y
where $y \in C \neg y \leq x$
 $\langle proof \rangle$

lemma *ccpo-Sup-least-inv'*:
fixes $C :: 'a :: \{ccpo, linorder\}$ *set*
assumes *Complete-Partial-Order.chain less-eq C* $\sqcup C > x$
obtains y
where $y \in C y > x$
 $\langle proof \rangle$

lemma *mcont2mcont-lessThan*[*THEN lfp.mcont2mcont, simp, cont-intro*]:
shows *mcont-lessThan: mcont Sup less-eq Sup less-eq*
(lessThan :: 'a :: \{ccpo, linorder\} \Rightarrow 'a set)
 $\langle proof \rangle$

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

class *esize-order* = *esize* + *order* +
assumes *esize-finite*[*dest*]: *esize* $x \neq \infty \Rightarrow$ *finite* $\{y. y \leq x\}$
assumes *esize-mono*[*intro*]: $x \leq y \Rightarrow$ *esize* $x \leq$ *esize* y
assumes *esize-strict-mono*[*intro*]: *esize* $x \neq \infty \Rightarrow$ $x < y \Rightarrow$ *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)
 $\langle proof \rangle$

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
 $\langle proof \rangle$

end

class *esize-ccpo* = *esize-order* + *ccpo*
begin

lemma *esize-cont*[*dest*]:
assumes *Complete-Partial-Order.chain less-eq C C* $C \neq \{\}$
shows *esize* $(\sqcup C) = \sqcup$ (*esize* ' C)
 $\langle proof \rangle$

lemma *esize-mcont*: *mcont Sup less-eq Sup less-eq esize*
 $\langle proof \rangle$


```

lemmas mcont2mcont-esize = 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]: $\text{card } \{.. < \text{enat } k\} = \text{card } \{.. < k\}$

<proof>

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

<proof>

lemma *enat-Collect*:

assumes $\infty \notin A$

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

<proof>

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

<proof>

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

begin

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

<proof> **termination** *<proof>*

lemma *esize-iff-empty*[iff]: $\text{esize } A = 0 \longleftrightarrow A = \{\}$ *<proof>*

lemma *esize-iff-infinite*[iff]: $\text{esize } A = \infty \longleftrightarrow \text{infinite } A$ *<proof>*

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

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

instance

<proof>

end

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

assumes *inj-on* f A

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

$\langle \text{proof} \rangle$
lemma *esize-insert1*[simp]: $a \notin A \implies \text{esize} (\text{insert } a \ A) = \text{eSuc} (\text{esize } A)$
 $\langle \text{proof} \rangle$
lemma *esize-insert2*[simp]: $a \in A \implies \text{esize} (\text{insert } a \ A) = \text{esize } A$
 $\langle \text{proof} \rangle$
lemma *esize-remove1*[simp]: $a \notin A \implies \text{esize} (A - \{a\}) = \text{esize } A$
 $\langle \text{proof} \rangle$
lemma *esize-remove2*[simp]: $a \in A \implies \text{esize} (A - \{a\}) = \text{epred} (\text{esize } A)$
 $\langle \text{proof} \rangle$
lemma *esize-union-disjoint*[simp]:
assumes $A \cap B = \{\}$
shows $\text{esize} (A \cup B) = \text{esize } A + \text{esize } B$
 $\langle \text{proof} \rangle$
lemma *esize-lessThan*[simp]: $\text{esize} \{.. < n\} = n$
 $\langle \text{proof} \rangle$
lemma *esize-atMost*[simp]: $\text{esize} \{.. n\} = \text{eSuc } n$
 $\langle \text{proof} \rangle$

lemma *least-eSuc*[simp]:
assumes $A \neq \{\}$
shows $\text{least} (\text{eSuc } 'A) = \text{eSuc} (\text{least } A)$
 $\langle \text{proof} \rangle$

lemma *Inf-enat-eSuc*[simp]: $\bigcap (\text{eSuc } 'A) = \text{eSuc} (\bigcap A)$ $\langle \text{proof} \rangle$

definition *lift* :: *nat set* \Rightarrow *nat set*
where $\text{lift } A \equiv \text{insert } 0 (\text{Suc } 'A)$

lemma *liftI-0*[intro, simp]: $0 \in \text{lift } A$ $\langle \text{proof} \rangle$
lemma *liftI-Suc*[intro]: $a \in A \implies \text{Suc } a \in \text{lift } A$ $\langle \text{proof} \rangle$
lemma *liftE*[elim]:
assumes $b \in \text{lift } A$
obtains $(0) \ b = 0 \mid (\text{Suc } a) \ \text{where } b = \text{Suc } a \ a \in A$
 $\langle \text{proof} \rangle$

lemma *lift-esize*[simp]: $\text{esize} (\text{lift } A) = \text{eSuc} (\text{esize } A)$ $\langle \text{proof} \rangle$
lemma *lift-least*[simp]: $\text{least} (\text{lift } A) = 0$ $\langle \text{proof} \rangle$

primrec *nth-least* :: *'a set* \Rightarrow *nat* \Rightarrow *'a* :: *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 $\text{nth-least } A \ n \in A$
 $\langle \text{proof} \rangle$

lemma *card-wellformed*[intro?, simp]:
fixes $k :: 'a :: \text{wellorder}$

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

lemma *nth-least-strict-mono*:
assumes $\text{enat } l < \text{esize } A$ $k < l$
shows $\text{nth-least } A k < \text{nth-least } A l$
 $\langle \text{proof} \rangle$

lemma *nth-least-mono*[*intro, simp*]:
assumes $\text{enat } l < \text{esize } A$ $k \leq l$
shows $\text{nth-least } A k \leq \text{nth-least } A l$
 $\langle \text{proof} \rangle$

lemma *card-nth-least*[*simp*]:
assumes $\text{enat } n < \text{esize } A$
shows $\text{card } \{k \in A. k < \text{nth-least } A n\} = n$
 $\langle \text{proof} \rangle$

lemma *card-nth-least-le*[*simp*]:
assumes $\text{enat } n < \text{esize } A$
shows $\text{card } \{k \in A. k \leq \text{nth-least } A n\} = \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *nth-least-card*:
fixes $k :: \text{nat}$
assumes $k \in A$
shows $\text{nth-least } A (\text{card } \{i \in A. i < k\}) = k$
 $\langle \text{proof} \rangle$

interpretation *nth-least*:
bounded-function-pair $\{i. \text{enat } i < \text{esize } A\} A$ *nth-least* $A \lambda k. \text{card } \{i \in A. i < k\}$
 $\langle \text{proof} \rangle$

interpretation *nth-least*:
injection $\{i. \text{enat } i < \text{esize } A\} A$ *nth-least* $A \lambda k. \text{card } \{i \in A. i < k\}$
 $\langle \text{proof} \rangle$

interpretation *nth-least*:
surjection $\{i. \text{enat } i < \text{esize } A\} A$ *nth-least* $A \lambda k. \text{card } \{i \in A. i < k\}$
for $A :: \text{nat set}$
 $\langle \text{proof} \rangle$

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

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 $k < l$

<proof>

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>

lemma *nth-least-less-esize-less*:

$\text{enat } n < \text{esize } A \wedge \text{enat } (\text{nth-least } A \ n) < k \longleftrightarrow \text{enat } n < \text{esize } \{i \in A. \text{enat } i < k\}$

<proof>

lemma *nth-least-le*:

assumes $\text{enat } n < \text{esize } A$

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

<proof>

lemma *nth-least-eq*:

assumes $\text{enat } n < \text{esize } A \text{ enat } n < \text{esize } B$

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 $\text{nth-least } A \ n = \text{nth-least } B \ n$

<proof>

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>

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

assumes $A \neq \{\} \wedge i. i \in A \implies \text{enat } i < \text{esize } B$

shows $\text{least } (\text{nth-least } B \ ' A) = \text{nth-least } B \ (\text{least } A)$

<proof>

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 $\text{nth-least } B \ (\text{nth-least } A \ n) = \text{nth-least } (\text{nth-least } B \ ' A) \ n$

<proof>

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

assumes $\text{finite } A \ A \neq \{\}$

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

<proof>

lemma *nth-least-le-Max*:

assumes $\text{finite } A \ A \neq \{\} \text{ enat } n < \text{esize } A$

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

lemma $nth\text{-least-not-contains}$:

fixes $k :: nat$

assumes $enat (Suc \ n) < esize \ A \ nth\text{-least } A \ n < k \ k < nth\text{-least } A \ (Suc \ n)$

shows $k \notin A$

$\langle proof \rangle$

lemma $nth\text{-least-Suc}[simp]$:

assumes $enat \ n < esize \ A$

shows $nth\text{-least } (Suc \ A) \ n = Suc \ (nth\text{-least } A \ n)$

$\langle proof \rangle$

lemma $nth\text{-least-lift}[simp]$:

$nth\text{-least } (lift \ A) \ 0 = 0$

$enat \ n < esize \ A \implies nth\text{-least } (lift \ A) \ (Suc \ n) = Suc \ (nth\text{-least } A \ n)$

$\langle proof \rangle$

lemma $nth\text{-least-list-card}[simp]$:

assumes $enat \ n \leq esize \ A$

shows $card \ \{k \in A. \ k < nth\text{-least } (lift \ A) \ n\} = n$

$\langle proof \rangle$

end

14 Coinductive Lists

theory $Coinductive\text{-List-Extensions}$

imports

$Coinductive.Coinductive\text{-List}$

$Coinductive.Coinductive\text{-List-Prefix}$

$Coinductive.Coinductive\text{-Stream}$

$../Extensions/List\text{-Extensions}$

$../Extensions/ESet\text{-Extensions}$

begin

hide-const (**open**) $Sublist.prefix$

hide-const (**open**) $Sublist.suffix$

declare $list\text{-of-lappend}[simp]$

declare $lnth\text{-lappend1}[simp]$

declare $lnth\text{-lappend2}[simp]$

declare $lprefix\text{-llength-le}[dest]$

declare $Sup\text{-llist-def}[simp]$

declare $length\text{-list-of}[simp]$

declare $llast\text{-linfinite}[simp]$

declare $lnth\text{-ltake}[simp]$

declare $lappend\text{-assoc}[simp]$

declare *lprefix-lappend*[simp]

lemma *lprefix-lSup-revert*: $lSup = Sup\ lprefix = less-eq$ *<proof>*

lemma *admissible-lprefixI*[cont-intro]:
assumes *mcont lub ord lSup lprefix f*
assumes *mcont lub ord lSup lprefix g*
shows *ccpo.admissible lub ord* $(\lambda x. lprefix\ (f\ x)\ (g\ x))$
<proof>

lemma *llist-lift-admissible*:
assumes *ccpo.admissible lSup lprefix P*
assumes $\bigwedge u. u \leq v \implies lfinite\ u \implies P\ u$
shows $P\ v$
<proof>

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

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

syntax *-llist* :: *args* \Rightarrow 'a *llist* ($\langle - \rangle$)

translations
 $\langle a, x \rangle \rightleftharpoons a\ \% \langle x \rangle$
 $\langle a \rangle \rightleftharpoons a\ \% \langle \rangle$

lemma *eq-LNil-conv-lnull*[simp]: $w = \langle \rangle \longleftrightarrow lnull\ w$ *<proof>*

lemma *Collect-lnull*[simp]: $\{w. lnull\ w\} = \{\langle \rangle\}$ *<proof>*

lemma *inj-on-ltake*: *inj-on* $(\lambda k. ltake\ k\ w)\ \{..\ llength\ w\}$
<proof>

lemma *lnth-inf-llist'*[simp]: $lnth\ (inf-llist\ f) = f$ *<proof>*

lemma *not-lnull-lappend-startE*[elim]:
assumes $\neg\ lnull\ w$
obtains $a\ v$
where $w = \langle a \rangle\ \$\ v$
<proof>

lemma *not-lnull-lappend-endE*[elim]:
assumes $\neg\ lnull\ w$
obtains $a\ v$
where $w = v\ \$\ \langle a \rangle$
<proof>

lemma *llength-lappend-startE*[elim]:
assumes $llength\ w \geq eSuc\ n$
obtains $a\ v$

where $w = \langle a \rangle \$ v$ $llength\ v \geq n$
<proof>

lemma *llength-lappend-endE[elim]*:
assumes $llength\ w \geq eSuc\ n$
obtains $a\ v$
where $w = v \$ \langle a \rangle$ $llength\ v \geq n$
<proof>

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

lemma *llength-lappend-end'E[elim]*:
assumes $llength\ w = enat\ (Suc\ n)$
obtains $a\ v$
where $w = v \$ \langle a \rangle$ $llength\ v = enat\ n$
<proof>

lemma *ltake-llast[simp]*:
assumes $enat\ k < llength\ w$
shows $llast\ (ltake\ (enat\ (Suc\ k))\ w) = w\ ?!\ k$
<proof>

lemma *linfinite-llength[dest, simp]*:
assumes $linfinite\ w$
shows $enat\ k < llength\ w$
<proof>

lemma *llist-nth-eqI[intro]*:
assumes $llength\ u = llength\ v$
assumes $\bigwedge i. enat\ i < llength\ u \implies enat\ i < llength\ v \implies u\ ?!\ i = v\ ?!\ i$
shows $u = v$
<proof>

primcorec *lscan* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ llist \Rightarrow 'b \Rightarrow 'b\ llist$
where $lscan\ f\ w\ a = (case\ w\ of\ \langle \rangle \Rightarrow \langle a \rangle \mid x\ \%\ xs \Rightarrow a\ \% lscan\ f\ xs\ (f\ x\ a))$

lemma *lscan-simps[simp]*:
 $lscan\ f\ \langle \rangle\ a = \langle a \rangle$
 $lscan\ f\ (x\ \% xs)\ a = a\ \% lscan\ f\ xs\ (f\ x\ a)$
<proof>

lemma *lscan-lfinite[iff]*: $lfinite\ (lscan\ f\ w\ a) \longleftrightarrow lfinite\ w$
<proof>

lemma *lscan-llength[simp]*: $llength\ (lscan\ f\ w\ a) = eSuc\ (llength\ w)$
<proof>

function *lfold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ llist \Rightarrow 'b \Rightarrow 'b$

where $lfinite\ w \implies lfold\ f\ w = fold\ f\ (list-of\ w) \mid linfinite\ w \implies lfold\ f\ w = id$
 $\langle proof \rangle$ **termination** $\langle proof \rangle$

lemma $lfold-llist-of[simp]$: $lfold\ f\ (llist-of\ xs) = fold\ f\ xs$ $\langle proof \rangle$

lemma $finite-UNIV-llength-eq$:

assumes $finite\ (UNIV :: 'a\ set)$

shows $finite\ \{w :: 'a\ llist.\ llength\ w = enat\ n\}$

$\langle proof \rangle$

lemma $finite-UNIV-llength-le$:

assumes $finite\ (UNIV :: 'a\ set)$

shows $finite\ \{w :: 'a\ llist.\ llength\ w \leq enat\ n\}$

$\langle proof \rangle$

lemma $lprefix-ltake[dest]$: $u \leq v \implies u = ltake\ (llength\ u)\ v$

$\langle proof \rangle$

lemma $prefixes-set$: $\{v.\ v \leq w\} = \{ltake\ k\ w \mid k.\ k \leq llength\ w\}$ $\langle proof \rangle$

lemma $esize-prefixes[simp]$: $esize\ \{v.\ v \leq w\} = eSuc\ (llength\ w)$

$\langle proof \rangle$

lemma $lprefix-subsume$: $v \leq w \implies u \leq w \implies llength\ v \leq llength\ u \implies v \leq u$

$\langle proof \rangle$

lemma $ltake-infinite[simp]$: $ltake\ \infty\ w = w$ $\langle proof \rangle$

lemma $lprefix-infinite$:

assumes $u \leq v\ linfinite\ u$

shows $u = v$

$\langle proof \rangle$

instantiation $llist :: (type)\ esize-order$

begin

definition $[simp]$: $esize \equiv llength$

instance

$\langle proof \rangle$

end

14.1 Index Sets

definition $lisset :: 'a\ set \Rightarrow 'a\ llist \Rightarrow nat\ set$

where $lisset\ A\ w \equiv \{i.\ enat\ i < llength\ w \wedge w\ ?!\ i \in A\}$

lemma $lissetI[intro]$:

assumes $enat\ i < llength\ w\ w\ ?!\ i \in A$

shows $i \in lisset\ A\ w$

$\langle proof \rangle$

lemma $lissetD[dest]$:

assumes $i \in \text{liset } A \ w$
shows $\text{enat } i < \text{llength } w \ w \ ?! \ i \in A$
 $\langle \text{proof} \rangle$

lemma *liset-finite*:
assumes $\text{lfinite } w$
shows $\text{finite } (\text{liset } A \ w)$
 $\langle \text{proof} \rangle$

lemma *liset-nil[simp]*: $\text{liset } A \ \langle \rangle = \{\}$ $\langle \text{proof} \rangle$

lemma *liset-cons-not-member[simp]*:
assumes $a \notin A$
shows $\text{liset } A \ (a \% w) = \text{Suc } \text{' } \text{liset } A \ w$
 $\langle \text{proof} \rangle$

lemma *liset-cons-member[simp]*:
assumes $a \in A$
shows $\text{liset } A \ (a \% w) = \{0\} \cup \text{Suc } \text{' } \text{liset } A \ w$
 $\langle \text{proof} \rangle$

lemma *liset-prefix*:
assumes $i \in \text{liset } A \ v \ u \leq v \ \text{enat } i < \text{llength } u$
shows $i \in \text{liset } A \ u$
 $\langle \text{proof} \rangle$

lemma *liset-suffix*:
assumes $i \in \text{liset } A \ u \ u \leq v$
shows $i \in \text{liset } A \ v$
 $\langle \text{proof} \rangle$

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

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

lemma *liset-cont[dest]*:
assumes $\text{Complete-Partial-Order.chain less-eq } C \ C \neq \{\}$
shows $\text{liset } A \ (\bigsqcup C) = (\bigcup w \in C. \text{liset } A \ w)$
 $\langle \text{proof} \rangle$

lemma *liset-mcont*: $\text{Complete-Partial-Order2.mcont lSup lprefix Sup less-eq}$
 $(\text{liset } A)$
 $\langle \text{proof} \rangle$

lemmas $\text{mcont2mcont-liset} = \text{liset-mcont}[\text{THEN lfp.mcont2mcont, simp, cont-intro}]$

14.2 Selections

abbreviation $\text{lproject } A \equiv \text{lfilter } (\lambda a. a \in A)$

abbreviation $\text{lselect } s \ w \equiv \text{lnths } w \ s$

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

<proof>

lemma *lproject-to-lselect*: $lproject\ A\ w = lselect\ (lset\ A\ w)\ w$

<proof>

lemma *lproject-llength[simp]*: $llength\ (lproject\ A\ w) = esize\ (lset\ A\ w)$

<proof>

lemma *lproject-lfinite[simp]*: $lfinite\ (lproject\ A\ w) \longleftrightarrow finite\ (lset\ A\ w)$

<proof>

lemma *lselect-restrict-indices[simp]*: $lselect\ \{i \in s.\ enat\ i < llength\ w\}\ w = lselect\ s\ w$

<proof>

lemma *lselect-llength*: $llength\ (lselect\ s\ w) = esize\ \{i \in s.\ enat\ i < llength\ w\}$

<proof>

lemma *lselect-llength-le[simp]*: $llength\ (lselect\ s\ w) \leq esize\ s$

<proof>

lemma *least-lselect-llength*:

assumes $\neg\ lnull\ (lselect\ s\ w)$

shows $enat\ (least\ s) < llength\ w$

<proof>

lemma *lselect-lnull*: $lnull\ (lselect\ s\ w) \longleftrightarrow (\forall\ i \in s.\ enat\ i \geq llength\ w)$

<proof>

lemma *lselect-discard-start*:

assumes $\bigwedge\ i.\ i \in s \implies k \leq i$

shows $lselect\ \{i.\ k + i \in s\}\ (ldropn\ k\ w) = lselect\ s\ w$

<proof>

lemma *lselect-discard-end*:

assumes $\bigwedge\ i.\ i \in s \implies i < k$

shows $lselect\ s\ (ltake\ (enat\ k)\ w) = lselect\ s\ w$

<proof>

lemma *lselect-least*:

assumes $\neg\ lnull\ (lselect\ s\ w)$

shows $lselect\ s\ w = w\ ?!\ least\ s\ \% lselect\ (s - \{least\ s\})\ w$

<proof>

lemma *lselect-lnth[simp]*:

assumes $enat\ i < llength\ (lselect\ s\ w)$

shows $lselect\ s\ w\ ?!\ i = w\ ?!\ nth\ least\ s\ i$

<proof>

lemma *lproject-lnth[simp]*:

assumes $enat\ i < llength\ (lproject\ A\ w)$

shows $lproject\ A\ w\ ?!\ i = w\ ?!\ nth\ least\ (lset\ A\ w)\ i$

<proof>

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

lemma *llength-less-llength-lselect-less*:
 $enat\ i < esize\ s \wedge enat\ (nth\ least\ s\ i) < llength\ w \iff enat\ i < llength\ (lselect\ s\ w)$
 $\langle proof \rangle$

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 $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\ least\ s\ 't)\ w$
 $\langle proof \rangle$

lemma *lselect-lselect'[simp]*:
assumes $\bigwedge i. i \in t \implies enat\ i < esize\ s$
shows $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\ least\ s\ 't)\ w$
 $\langle proof \rangle$

lemma *lselect-lselect*:
 $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\ least\ s\ ' \{i \in t. enat\ i < esize\ s\})\ w$
 $\langle proof \rangle$

lemma *lselect-lproject'*:
assumes $\bigwedge i. i \in s \implies enat\ i < llength\ w$
shows $lproject\ A\ (lselect\ s\ w) = lselect\ (s \cap liset\ A\ w)\ w$
 $\langle proof \rangle$

lemma *lselect-lproject[simp]*: $lproject\ A\ (lselect\ s\ w) = lselect\ (s \cap liset\ A\ w)\ w$
 $\langle proof \rangle$

lemma *lproject-lselect-subset[simp]*:
assumes $liset\ A\ w \subseteq s$
shows $lproject\ A\ (lselect\ s\ w) = lproject\ A\ w$
 $\langle proof \rangle$

lemma *lselect-prefix[intro]*:
assumes $u \leq v$
shows $lselect\ s\ u \leq lselect\ s\ v$
 $\langle proof \rangle$

lemma *lproject-prefix[intro]*:
assumes $u \leq v$
shows $lproject\ A\ u \leq lproject\ A\ v$
 $\langle proof \rangle$

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>
  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>

end

```

15 Prefixes on Coinductive Lists

```

theory LList-Prefixes
imports
  Word-Prefixes
  ../Extensions/Coinductive-List-Extensions
begin

lemma unfold-stream-siterate-smap:  $\text{unfold-stream } f \ g = \text{smap } f \circ \text{siterate } g$ 
  <proof>

lemma lappend-stream-of-llist:
  assumes  $\text{lfinite } u$ 
  shows  $\text{stream-of-llist } (u \ \$ \ v) = \text{list-of } u \ @- \ \text{stream-of-llist } v$ 
  <proof>

lemma llist-of-inf-llist-prefix[intro]:  $u \leq_{FI} v \implies \text{llist-of } u \leq \text{llist-of-stream } v$ 
  <proof>
lemma prefix-llist-of-inf-llist[intro]:  $\text{lfinite } u \implies u \leq v \implies \text{list-of } u \leq_{FI} \text{stream-of-llist } v$ 
  <proof>

lemma lproject-prefix-limit-chain:
  assumes  $\text{chain } w \ \bigwedge k. \text{lproject } A \ (\text{llist-of } (w \ k)) \leq x$ 
  shows  $\text{lproject } A \ (\text{llist-of-stream } (\text{limit } w)) \leq x$ 
  <proof>
lemma lproject-eq-limit-chain:
  assumes  $\text{chain } u \ \text{chain } v \ \bigwedge k. \text{project } A \ (u \ k) = \text{project } A \ (v \ k)$ 
  shows  $\text{lproject } A \ (\text{llist-of-stream } (\text{limit } u)) = \text{lproject } A \ (\text{llist-of-stream } (\text{limit } v))$ 
  <proof>

end

```

16 Stuttering

```

theory Stuttering
imports

```

Stuttering-Equivalence.StutterEquivalence
LList-Prefixes

begin

function *nth-least-ext* :: *nat set* \Rightarrow *nat* \Rightarrow *nat*

where

enat $k < \text{esize } A \implies \text{nth-least-ext } A \ k = \text{nth-least } A \ k \mid$

enat $k \geq \text{esize } A \implies \text{nth-least-ext } A \ k = \text{Suc } (\text{Max } A + (k - \text{card } A))$

$\langle \text{proof} \rangle$ **termination** $\langle \text{proof} \rangle$

lemma *nth-least-ext-strict-mono*:

assumes $k < l$

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

$\langle \text{proof} \rangle$

definition *stutter-selection* :: *nat set* \Rightarrow 'a *llist* \Rightarrow *bool*

where *stutter-selection* $s \ w \equiv 0 \in s \wedge$

$(\forall k \ i. \ \text{enat } i < \text{llength } w \longrightarrow \text{enat } (\text{Suc } k) < \text{esize } s \longrightarrow$

$\text{nth-least } s \ k < i \longrightarrow i < \text{nth-least } s \ (\text{Suc } k) \longrightarrow w \ ?! \ i = w \ ?! \ \text{nth-least } s \ k) \wedge$

$(\forall i. \ \text{enat } i < \text{llength } w \longrightarrow \text{finite } s \longrightarrow \text{Max } s < i \longrightarrow w \ ?! \ i = w \ ?! \ \text{Max } s)$

lemma *stutter-selectionI*[*intro*]:

assumes $0 \in s$

assumes $\bigwedge k \ i. \ \text{enat } i < \text{llength } w \implies \text{enat } (\text{Suc } k) < \text{esize } s \implies$

$\text{nth-least } s \ k < i \implies i < \text{nth-least } s \ (\text{Suc } k) \implies w \ ?! \ i = w \ ?! \ \text{nth-least } s \ k$

assumes $\bigwedge i. \ \text{enat } i < \text{llength } w \implies \text{finite } s \implies \text{Max } s < i \implies w \ ?! \ i = w$

$\ ?! \ \text{Max } s$

shows *stutter-selection* $s \ w$

$\langle \text{proof} \rangle$

lemma *stutter-selectionD-0*[*dest*]:

assumes *stutter-selection* $s \ w$

shows $0 \in s$

$\langle \text{proof} \rangle$

lemma *stutter-selectionD-inside*[*dest*]:

assumes *stutter-selection* $s \ w$

assumes $\text{enat } i < \text{llength } w \ \text{enat } (\text{Suc } k) < \text{esize } s$

assumes $\text{nth-least } s \ k < i \ i < \text{nth-least } s \ (\text{Suc } k)$

shows $w \ ?! \ i = w \ ?! \ \text{nth-least } s \ k$

$\langle \text{proof} \rangle$

lemma *stutter-selectionD-infinite*[*dest*]:

assumes *stutter-selection* $s \ w$

assumes $\text{enat } i < \text{llength } w \ \text{finite } s \ \text{Max } s < i$

shows $w \ ?! \ i = w \ ?! \ \text{Max } s$

$\langle \text{proof} \rangle$

lemma *stutter-selection-stutter-sampler*[*intro*]:

assumes *linfinite* $w \ \text{stutter-selection } s \ w$

shows *stutter-sampler* $(\text{nth-least-ext } s) \ (\text{lth } w)$

$\langle \text{proof} \rangle$

```

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⟩

definition stuttering-invariant :: 'a word set ⇒ bool
  where stuttering-invariant A ≡ ∀ u v. u ≈ v ⟶ u ∈ A ⟷ v ∈ A

lemma stuttering-invariant-complement[intro!]:
  assumes stuttering-invariant A
  shows stuttering-invariant (− A)
  ⟨proof⟩

lemma stutter-equiv-forw-subst[trans]: w1 = w2 ⟹ w2 ≈ w3 ⟹ w1 ≈ w3
  ⟨proof⟩

lemma stutter-sampler-build:
  assumes stutter-sampler f w
  shows stutter-sampler (0 ## (Suc ∘ f)) (a ## w)
  ⟨proof⟩
lemma stutter-extend-build:
  assumes u ≈ v
  shows a ## u ≈ a ## v
  ⟨proof⟩
lemma stutter-extend-concat:
  assumes u ≈ v
  shows w ∙ u ≈ w ∙ v
  ⟨proof⟩
lemma build-stutter: w 0 ## w ≈ w
  ⟨proof⟩
lemma replicate-stutter: replicate n (v 0) ∙ v ≈ v
  ⟨proof⟩

lemma replicate-stutter': u ∙ replicate n (v 0) ∙ v ≈ u ∙ v
  ⟨proof⟩

```

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*: $a \in \text{visible} \implies b \in \text{visible} \implies \neg \text{ind } a \ b$
begin

lemma *eq-swap-lproject-visible*:

assumes $u =_S v$
shows $\text{lproject visible (llist-of } u) = \text{lproject visible (llist-of } v)$
 $\langle \text{proof} \rangle$

lemma *eq-fin-lproject-visible*:

assumes $u =_F v$
shows $\text{lproject visible (llist-of } u) = \text{lproject visible (llist-of } v)$
 $\langle \text{proof} \rangle$

lemma *le-fin-lproject-visible*:

assumes $u \preceq_F v$
shows $\text{lproject visible (llist-of } u) \leq \text{lproject visible (llist-of } v)$
 $\langle \text{proof} \rangle$

lemma *le-fininf-lproject-visible*:

assumes $u \preceq_{FI} v$
shows $\text{lproject visible (llist-of } u) \leq \text{lproject visible (llist-of-stream } v)$
 $\langle \text{proof} \rangle$

lemma *le-inf-lproject-visible*:

assumes $u \preceq_I v$
shows $\text{lproject visible (llist-of-stream } u) \leq \text{lproject visible (llist-of-stream } v)$
 $\langle \text{proof} \rangle$

lemma *eq-inf-lproject-visible*:

assumes $u =_I v$
shows $\text{lproject visible (llist-of-stream } u) = \text{lproject visible (llist-of-stream } v)$
 $\langle \text{proof} \rangle$

lemma *stutter-selection-lproject-visible*:

assumes $\text{run } u \ p$
shows $\text{stutter-selection (lift (lset visible (llist-of-stream } u)))}$
 $\quad (\text{llist-of-stream (smap int (p \#\# trace } u \ p)))$
 $\langle \text{proof} \rangle$

lemma *execute-fin-visible*:

assumes $\text{path } u \ q \ \text{path } v \ q \ u \preceq_{FI} w \ v \preceq_{FI} w$
assumes $\text{project visible } u = \text{project visible } v$
shows $\text{int (target } u \ q) = \text{int (target } v \ q)$
 $\langle \text{proof} \rangle$

lemma *execute-inf-visible*:

assumes $\text{run } u \ q \ \text{run } v \ q \ u \preceq_I w \ v \preceq_I w$

```

assumes lproject visible (llist-of-stream u) = lproject visible (llist-of-stream v)
shows snth (smap int (q ## trace u q)) ≈ snth (smap int (q ## trace v q))
⟨proof⟩

```

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 ⇒ 'state ⇒ 'state

and *en* :: 'action ⇒ 'state ⇒ bool

and *int* :: 'state ⇒ 'interpretation

and *ind* :: 'action ⇒ 'action ⇒ bool

and *src* :: 'state ⇒ 'state ⇒ bool

begin

definition *ample-set* :: 'state ⇒ 'action set ⇒ bool

where *ample-set* *q A* ≡

$A \subseteq \{a. \text{en } a \ q\} \wedge$

$(A \subseteq \{a. \text{en } a \ q\} \longrightarrow A \neq \{\}) \wedge$

$(\forall a. A \subseteq \{a. \text{en } a \ q\} \longrightarrow a \in A \longrightarrow \text{src } (\text{ex } a \ q) \ q) \wedge$

$(A \subseteq \{a. \text{en } a \ q\} \longrightarrow A \subseteq \text{invisible}) \wedge$

$(\forall w. A \subseteq \{a. \text{en } a \ q\} \longrightarrow \text{path } w \ q \longrightarrow A \cap \text{set } w = \{\} \longrightarrow \text{Ind } A \ (\text{set } w))$

lemma *ample-subset*:

assumes *ample-set* *q A*

shows $A \subseteq \{a. \text{en } a \ q\}$

⟨*proof*⟩

lemma *ample-nonempty*:

assumes *ample-set* *q A* $A \subseteq \{a. \text{en } a \ q\}$

shows $A \neq \{\}$

⟨*proof*⟩

lemma *ample-wellfounded*:
assumes *ample-set* q A $A \subset \{a. \text{en } a \ q\}$ $a \in A$
shows *src* (*ex* a q) q
 $\langle \text{proof} \rangle$

lemma *ample-invisible*:
assumes *ample-set* q A $A \subset \{a. \text{en } a \ q\}$
shows $A \subseteq \text{invisible}$
 $\langle \text{proof} \rangle$

lemma *ample-independent*:
assumes *ample-set* q A $A \subset \{a. \text{en } a \ q\}$ *path* w q $A \cap \text{set } w = \{\}$
shows *Ind* A (*set* w)
 $\langle \text{proof} \rangle$

lemma *ample-en[intro]*: *ample-set* q $\{a. \text{en } a \ q\}$ $\langle \text{proof} \rangle$

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*: $q \in \text{nodes} \Longrightarrow \text{ample-set } q \ \{a. \text{ren } a \ q\}$
begin

lemma *reduction-words-fin*:
assumes $q \in \text{nodes}$ *R.path* w q
shows *S.path* w q
 $\langle \text{proof} \rangle$

lemma *reduction-words-inf*:
assumes $q \in \text{nodes}$ *R.run* w q
shows *S.run* w q
 $\langle \text{proof} \rangle$

lemma *reduction-step*:
assumes $q \in \text{nodes}$ *run* w q
obtains
(*deferred*) a **where** *ren* a q $[a] \preceq_{FI} w$ |
(*omitted*) $\{a. \text{ren } a \ q\} \subseteq \text{invisible}$ *Ind* $\{a. \text{ren } a \ q\}$ (*sset* w)

$\langle proof \rangle$

lemma *reduction-chunk*:

assumes $q \in nodes$ $run ([a] @- v) q$

obtains $b b_1 b_2 u$

where

$R.path (b @ [a]) q$

$Ind \{a\} (set b) set b \subseteq invisible$

$b =_F b_1 @ b_2 b_1 @- u =_I v Ind (set b_2) (sset u)$

$\langle proof \rangle$

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: $reduced-run q v_1 ([a] @- v_2) l w w_1 w_2 u \Longrightarrow a \in set l \Longrightarrow$

$reduced-run q (v_1 @ [a]) v_2 (remove1 a l) w w_1 w_2 u |$

extend: $reduced-run q v_1 ([a] @- v_2) l w w_1 w_2 u \Longrightarrow a \notin set l \Longrightarrow$

$R.path (b @ [a]) (target w q) \Longrightarrow$

$Ind \{a\} (set b) \Longrightarrow set b \subseteq invisible \Longrightarrow$

$b =_F b_1 @ b_2 \Longrightarrow [a] @- b_1 @- u' =_I u \Longrightarrow Ind (set b_2) (sset u') \Longrightarrow$

$reduced-run q (v_1 @ [a]) v_2 (l @ b_1) (w @ b @ [a]) (w_1 @ b_1 @ [a]) (w_2 @$

$b_2) u'$

lemma *reduced-run-words-fin*:

assumes $reduced-run q v_1 v_2 l w w_1 w_2 u$

shows $R.path w q$

$\langle proof \rangle$

lemma *reduced-run-invar-2*:

assumes $reduced-run q v_1 v_2 l w w_1 w_2 u$

shows $v_2 =_I l @- u$

$\langle proof \rangle$

lemma *reduced-run-invar-1*:

assumes $reduced-run q v_1 v_2 l w w_1 w_2 u$

shows $v_1 @ l =_F w_1$

$\langle proof \rangle$

lemma *reduced-run-invisible*:

assumes $reduced-run q v_1 v_2 l w w_1 w_2 u$

shows $set w_2 \subseteq invisible$

$\langle proof \rangle$

lemma *reduced-run-ind*:

assumes $reduced-run q v_1 v_2 l w w_1 w_2 u$

shows $Ind (set w_2) (sset u)$
 $\langle proof \rangle$

lemma *reduced-run-decompose*:
assumes $reduced-run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $w =_F w_1 @ w_2$
 $\langle proof \rangle$

lemma *reduced-run-project*:
assumes $reduced-run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $project\ visible\ w_1 = project\ visible\ w$
 $\langle proof \rangle$

lemma *reduced-run-length-1*:
assumes $reduced-run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $length\ v_1 \leq length\ w_1$
 $\langle proof \rangle$

lemma *reduced-run-length*:
assumes $reduced-run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $length\ v_1 \leq length\ w$
 $\langle proof \rangle$

lemma *reduced-run-step*:
assumes $q \in nodes\ run\ (v_1 @- [a] @- v_2)\ q$
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 $reduced-run\ q\ (v_1 @ [a])\ v_2\ l'\ (w @ w')\ (w_1 @ w_1')\ (w_2 @ w_2')\ u'$
 $\langle proof \rangle$

lemma *reduction-word*:
assumes $q \in nodes\ run\ v\ q$
obtains $u\ w$
where
 $R.run\ w\ q$
 $v =_I u\ u \preceq_I w$
 $lproject\ visible\ (lstream\ u) = lproject\ visible\ (lstream\ w)$
 $\langle proof \rangle$

lemma *reduction-equivalent*:
assumes $q \in nodes\ run\ u\ q$
obtains v
where $R.run\ v\ q\ snth\ (smap\ int\ (q\ ##\ trace\ u\ q)) \approx snth\ (smap\ int\ (q\ ##\ trace\ v\ q))$
 $\langle proof \rangle$

lemma *reduction-language-subset*: $R.language \subseteq S.language$
 $\langle proof \rangle$

```

lemma reduction-language-stuttering:
  assumes  $u \in S.\text{language}$ 
  obtains  $v$ 
  where  $v \in R.\text{language}$   $\text{snth } u \approx \text{snth } v$ 
   $\langle \text{proof} \rangle$ 

```

end

end

19 LTL Formulae

theory *Formula*

imports

Basics/Stuttering

Stuttering-Equivalence.PLTL

begin

locale *formula* =

fixes $\varphi :: 'a \text{ pltl}$

begin

definition *language* :: $'a \text{ stream set}$

where $\text{language} \equiv \{w. \text{snth } w \models_p \varphi\}$

lemma *language-entails[iff]*: $w \in \text{language} \longleftrightarrow \text{snth } w \models_p \varphi$ $\langle \text{proof} \rangle$

end

locale *formula-next-free* =

formula φ

for $\varphi :: 'a \text{ pltl}$

+

assumes *next-free*: *next-free* φ

begin

lemma *stutter-equivalent-entails[dest]*: $u \approx v \implies u \models_p \varphi \longleftrightarrow v \models_p \varphi$

$\langle \text{proof} \rangle$

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  $\Rightarrow$  '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
    and  $\varphi$  :: 'interpretation plll
  begin

    lemma reduction-language-indistinguishable:
      assumes R.language  $\subseteq$  F.language
      shows S.language  $\subseteq$  F.language
      <proof>

    theorem reduction-correct: S.language  $\subseteq$  F.language  $\longleftrightarrow$  R.language  $\subseteq$  F.language
      <proof>

  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  $\Rightarrow$  'state  $\Rightarrow$  'state
    and en :: 'action  $\Rightarrow$  'state  $\Rightarrow$  bool
    and init :: 'state  $\Rightarrow$  bool
    and int :: 'state  $\Rightarrow$  'interpretation
    and sticky :: 'action set
    and ind :: 'action  $\Rightarrow$  'action  $\Rightarrow$  bool
  begin

```

sublocale *ample-base ex en int ind scut*⁻¹⁻¹ $\langle proof \rangle$

lemma *restrict-ample-set*:

assumes $s \in nodes$
assumes $A \cap \{a. en a s\} \neq \{\}$ $A \cap \{a. en a s\} \cap sticky = \{\}$
assumes $Ind (A \cap \{a. en a s\}) (executable - A)$
assumes $\bigwedge w. path w s \implies A \cap \{a. en a s\} \cap set w = \{\} \implies A \cap set w = \{\}$
shows *ample-set* $s (A \cap \{a. en a s\})$
 $\langle proof \rangle$

end

locale *transition-system-concurrent* =

transition-system-initial ex en init
for $ex :: 'action \Rightarrow 'state \Rightarrow 'state$
and $en :: 'action \Rightarrow 'state \Rightarrow bool$
and $init :: 'state \Rightarrow bool$
 $+$
fixes $procs :: 'state \Rightarrow 'process set$
fixes $pac :: 'process \Rightarrow 'action set$
fixes $psen :: 'process \Rightarrow 'state \Rightarrow 'action set$
assumes *procs-finite*: $s \in nodes \implies finite (procs s)$
assumes *psen-en*: $s \in nodes \implies pac p \cap \{a. en a s\} \subseteq psen p s$
assumes *psen-ex*: $s \in nodes \implies a \in \{a. en a s\} - pac p \implies psen p (ex a s)$
 $= psen p s$
begin

lemma *psen-fin-word*:

assumes $s \in nodes path w s pac p \cap set w = \{\}$
shows $psen p (target w s) = psen p s$
 $\langle proof \rangle$

lemma *en-fin-word*:

assumes $\bigwedge r a b. r \in nodes \implies a \in psen p s - \{a. en a s\} \implies b \in \{a. en a r\} - pac p \implies$
 $en a (ex b r) \implies en a r$
assumes $s \in nodes path w s pac p \cap set w = \{\}$
shows $pac p \cap \{a. en a (target w s)\} \subseteq pac p \cap \{a. en a s\}$
 $\langle proof \rangle$

lemma *pac-en-blocked*:

assumes $\bigwedge r a b. r \in nodes \implies a \in psen p s - \{a. en a s\} \implies b \in \{a. en a r\} - pac p \implies$
 $en a (ex b r) \implies en a r$
assumes $s \in nodes path w s pac p \cap \{a. en a s\} \cap set w = \{\}$
shows $pac p \cap set w = \{\}$
 $\langle proof \rangle$

abbreviation $proc\ a \equiv \{p. a \in pac\ p\}$
abbreviation $Proc\ A \equiv \bigcup a \in A. proc\ a$

lemma *psen-simple*:

assumes $Proc\ (psen\ p\ s) = \{p\}$

assumes $\bigwedge r\ a\ b. r \in nodes \implies a \in psen\ p\ s - \{a. en\ a\ s\} \implies en\ b\ r \implies$
 $proc\ a \cap proc\ b = \{\} \implies en\ a\ (ex\ b\ r) \implies en\ a\ r$

shows $\bigwedge r\ a\ b. r \in nodes \implies a \in psen\ p\ s - \{a. en\ a\ s\} \implies b \in \{a. en\ a$
 $r\} - pac\ p \implies$
 $en\ a\ (ex\ b\ r) \implies en\ a\ r$
 $\langle proof \rangle$

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.