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

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 $a = b$ $u \le v$
shows $a \# u \le b \# v$
⟨*proof*⟩
lemma *prefix-finE-item*[*elim*!]:
assumes $a \# u \le b \# v$
obtains $a = b$ $u \le v$
⟨*proof*⟩

lemma *prefix-fin-append*[*intro*]: $u \le u @ v$ ⟨*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*]:
$i < length\ xs \Longrightarrow (xs\ @\ ys)\ !\ i = xs\ !\ i$
$i \geq length\ xs \Longrightarrow (xs\ @\ ys)\ !\ i = ys\ !\ (i - length\ xs)$
⟨*proof*⟩

notation *zip* (**infixr** ‹||› *51*)

abbreviation $project\ A \equiv filter\ (\lambda\ a.\ a \in A)$
abbreviation $select\ s\ w \equiv nths\ w\ s$

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

lemma *singleton-list-lengthE*[*elim*]:
assumes $length\ xs = 1$
obtains x
where $xs = [x]$
⟨*proof*⟩

lemma *singleton-hd-last*: $length\ xs = 1 \Longrightarrow hd\ xs = last\ xs$ ⟨*proof*⟩

lemma *set-subsetI*[*intro*]:
assumes $\bigwedge\ i.\ i < length\ xs \Longrightarrow xs\ !\ i \in S$
shows $set\ xs \subseteq S$
⟨*proof*⟩

lemma *hd-take*[*simp*]:
assumes $n \neq 0\ xs \neq []$
shows $hd\ (take\ n\ xs) = hd\ xs$
⟨*proof*⟩
lemma *hd-drop*[*simp*]:
assumes $n < length\ xs$
shows $hd\ (drop\ n\ xs) = xs\ !\ n$
⟨*proof*⟩
lemma *last-take*[*simp*]:
assumes $n < length\ xs$
shows $last\ (take\ (Suc\ n)\ xs) = xs\ !\ n$

⟨*proof*⟩

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*⟩

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*]: $u \leq_{FI} v \Longrightarrow set\ u \subseteq sset\ v$ ⟨*proof*⟩

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

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*]: $u \leq v \Longrightarrow v \leq_{FI} w \Longrightarrow u \leq_{FI}$
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* $u \Longrightarrow u$! $i = w$!! i
shows $u \leq_{FI} w$
⟨*proof*⟩

definition *chain* :: (*nat* ⇒ *'a list*) ⇒ *bool*
where *chain* $w \equiv$ *mono* $w \wedge (\forall\ k.\ \exists\ l.\ k <$ *length* $(w\ l))$
definition *limit* :: (*nat* ⇒ *'a list*) ⇒ *'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 $w\ k \leq_{FI}$ *limit w*
⟨*proof*⟩

lemma *chain-construct-1*:
assumes $P\ 0\ x_0$ $\bigwedge k\ x.\ P\ k\ x \Longrightarrow \exists\ x'.\ P\ (Suc\ k)\ x' \wedge f\ x \leq f\ x'$
assumes $\bigwedge k\ x.\ P\ k\ x \Longrightarrow k \leq length\ (f\ x)$
obtains Q
where $\bigwedge k.\ P\ k\ (Q\ k)$ $chain\ (f \circ Q)$
⟨*proof*⟩
lemma *chain-construct-2*:
assumes $P\ 0\ x_0$ $\bigwedge k\ x.\ P\ k\ x \Longrightarrow \exists\ x'.\ P\ (Suc\ k)\ x' \wedge f\ x \leq f\ x' \wedge g\ x \leq g\ x'$
assumes $\bigwedge k\ x.\ P\ k\ x \Longrightarrow k \leq length\ (f\ x)$ $\bigwedge k\ x.\ P\ k\ x \Longrightarrow k \leq length\ (g\ x)$
obtains Q
where $\bigwedge k.\ P\ k\ (Q\ k)$ $chain\ (f \circ Q)$ $chain\ (g \circ Q)$
⟨*proof*⟩
lemma *chain-construct-2'*:
assumes $P\ 0\ u_0\ v_0$ $\bigwedge k\ u\ v.\ P\ k\ u\ v \Longrightarrow \exists\ u'\ v'.\ P\ (Suc\ k)\ u'\ v' \wedge u \leq u' \wedge v \leq v'$
assumes $\bigwedge k\ u\ v.\ P\ k\ u\ v \Longrightarrow k \leq length\ u$ $\bigwedge k\ u\ v.\ P\ k\ u\ v \Longrightarrow k \leq length\ v$
obtains $u\ v$
where $\bigwedge k.\ P\ k\ (u\ k)\ (v\ k)$ $chain\ u$ $chain\ v$
⟨*proof*⟩

end

4 Sets

theory *Set-Extensions*
imports
HOL−Library.Infinite-Set
begin

declare *finite-subset*[*intro*]

lemma *set-not-emptyI*[*intro 0*]: $x \in S \Longrightarrow S \neq \{\}$ ⟨*proof*⟩
lemma *sets-empty-iffI*[*intro 0*]:
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\ b.\ b \in B$
assumes $\bigwedge b.\ b \in B \Longrightarrow \exists\ a.\ a \in A$
shows $A = \{\} \longleftrightarrow B = \{\}$
⟨*proof*⟩
lemma *disjointI*[*intro 0*]:
assumes $\bigwedge x.\ x \in A \Longrightarrow x \in B \Longrightarrow False$
shows $A \cap B = \{\}$
⟨*proof*⟩
lemma *range-subsetI*[*intro 0*]:
assumes $\bigwedge x.\ f\ x \in S$
shows $range\ f \subseteq S$

$\langle proof \rangle$

lemma *finite-imageI-range*:
assumes *finite* (*range f*)
shows *finite* (f ‘ A)
$\langle proof \rangle$

lemma *inf-img-fin-domE′*:
assumes *infinite A*
assumes *finite* (f ‘ A)
obtains y
where $y \in f$ ‘ A *infinite* ($A \cap f$ −‘ $\{y\}$)
$\langle proof \rangle$

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

lemma *these-alt-def*: *Option.these* S = *Some* −‘ S $\langle proof \rangle$
lemma *the-vimage-subset*: *the* −‘ $\{a\} \subseteq \{None,\ Some\ a\}$ $\langle proof \rangle$

lemma *finite-induct-reverse*[*consumes 1*, *case-names remove*]:
assumes *finite S*
assumes $\bigwedge S.\ finite\ S \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow P\ (S - \{x\})) \Longrightarrow P\ S$
shows $P\ S$
$\langle proof \rangle$

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

lemma *Collect-split-Suc*:
$\neg P\ 0 \Longrightarrow \{i.\ P\ i\} = Suc$ ‘ $\{i.\ P\ (Suc\ i)\}$
$P\ 0 \Longrightarrow \{i.\ P\ i\} = \{0\} \cup Suc$ ‘ $\{i.\ P\ (Suc\ i)\}$
$\langle proof \rangle$

lemma *Collect-subsume*[*simp*]:
assumes $\bigwedge x.\ x \in A \Longrightarrow P\ x$
shows $\{x \in A.\ P\ x\} = A$
$\langle proof \rangle$

lemma *Max-ge′*:
assumes *finite A* $A \neq \{\}$
assumes $b \in A$ $a \leq b$
shows $a \leq Max\ A$
$\langle proof \rangle$

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* $A \in A$

$\langle proof \rangle$

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

lemma *least-least*[*intro?*, *simp*]:
fixes A :: $'a$ :: *wellorder set*
assumes $k \in A$
shows *least* $A \leq k$
$\langle proof \rangle$

lemma *least-unique*:
fixes A :: $'a$ :: *wellorder set*
assumes $k \in A$ $k \leq$ *least* A
shows $k =$ *least* A
$\langle proof \rangle$

lemma *least-not-less*:
fixes A :: $'a$ :: *wellorder set*
assumes $k <$ *least* A
shows $k \notin A$
$\langle proof \rangle$

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

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

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

lemma *least-le*:
fixes A B :: $'a$ :: *wellorder set*
assumes $B \neq \{\}$
assumes $\bigwedge i.\ i \leq$ *least* $A \Longrightarrow i \leq$ *least* $B \Longrightarrow i \in B \Longrightarrow i \in A$
shows *least* $A \leq$ *least* B
$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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 $x \in A\ y \in A\ card\ \{z \in A.\ z \le x\} = card\ \{z \in A.\ z \le y\}$
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*: $f\ `\ A = \{f\ x\ |x.\ x \in A\}$ $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *image-union-split*:
assumes f ‘ $(A \cup B) = g$ ‘ C
obtains D E
where f ‘ $A = g$ ‘ D f ‘ $B = g$ ‘ E $D \subseteq C$ $E \subseteq C$
$\langle proof \rangle$
lemma *image-insert-split*:
assumes *inj* g f ‘ *insert* a $B = g$ ‘ C
obtains d E
where $f\ a = g\ d$ f ‘ $B = g$ ‘ E $d \in C$ $E \subseteq C$
$\langle proof \rangle$

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 $(x, y) \in Id$
$\langle proof \rangle$

lemma (**in** *order*) *order-le-cases*:
assumes $x \leq y$
obtains (*eq*) $x = y$ | (*lt*) $x < y$
$\langle proof \rangle$

lemma (**in** *linorder*) *linorder-cases'*:
obtains (*le*) $x \leq y$ | (*gt*) $x > y$
$\langle proof \rangle$

lemma *monoI-comp*[*intro*]:
assumes *mono* f *mono* g

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

lemma *eq-le-absorb*[*simp*]:
fixes $x\ y :: 'a :: order$
shows $x = y \wedge x \leq y \longleftrightarrow x = y$ $x \leq y \wedge x = y \longleftrightarrow x = y$
$\langle proof \rangle$

lemma *INFM-Suc*[*simp*]: $(\exists_\infty\ i.\ P\ (Suc\ i)) \longleftrightarrow (\exists_\infty\ i.\ P\ i)$
$\langle proof \rangle$
lemma *INFM-plus*[*simp*]: $(\exists_\infty\ i.\ P\ (i + n :: nat)) \longleftrightarrow (\exists_\infty\ i.\ P\ i)$
$\langle proof \rangle$
lemma *INFM-minus*[*simp*]: $(\exists_\infty\ i.\ P\ (i - n :: nat)) \longleftrightarrow (\exists_\infty\ i.\ P\ i)$
$\langle proof \rangle$

5.3 Constants

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

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

definition $zip\text{-}fun :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infixr** ‹‖› *51*)
where $f \parallel g \equiv \lambda\ x.\ (f\ x,\ g\ x)$

lemma *zip-fun-simps*[*simp*]:
$(f \parallel g)\ x = (f\ x,\ g\ x)$
$fst \circ (f \parallel g) = f$
$snd \circ (f \parallel g) = g$
$fst \circ h \parallel snd \circ h = h$
fst ‘ $range\ (f \parallel g) = range\ f$
snd ‘ $range\ (f \parallel g) = range\ g$
$\langle proof \rangle$

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

lemma *zip-fun-range-subset*[*intro*, *simp*]: *range* $(f \parallel g) \subseteq$ *range* $f \times$ *range* g
  ⟨*proof*⟩
lemma *zip-fun-range-finite*[*elim*]:
  assumes *finite* (*range* $(f \parallel g)$)
  obtains *finite* (*range* f) *finite* (*range* g)
⟨*proof*⟩

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

abbreviation *None-None* ≡ (*None*, *None*)
abbreviation *None-Some* ≡ λ (*y*). (*None*, *Some y*)
abbreviation *Some-None* ≡ λ (*x*). (*Some x*, *None*)
abbreviation *Some-Some* ≡ λ (*x*, *y*). (*Some x*, *Some y*)

abbreviation *None-None-None* ≡ (*None*, *None*, *None*)
abbreviation *None-None-Some* ≡ λ (*z*). (*None*, *None*, *Some z*)
abbreviation *None-Some-None* ≡ λ (*y*). (*None*, *Some y*, *None*)
abbreviation *None-Some-Some* ≡ λ (*y*, *z*). (*None*, *Some y*, *Some z*)
abbreviation *Some-None-None* ≡ λ (*x*). (*Some x*, *None*, *None*)
abbreviation *Some-None-Some* ≡ λ (*x*, *z*). (*Some x*, *None*, *Some z*)
abbreviation *Some-Some-None* ≡ λ (*x*, *y*). (*Some x*, *Some y*, *None*)
abbreviation *Some-Some-Some* ≡ λ (*x*, *y*, *z*). (*Some x*, *Some y*, *Some z*)

lemma *inj-Some2*[*simp*, *intro*]:
  inj None-Some
  inj Some-None
  inj Some-Some
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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

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

lemma *swap-simps*[*simp*]: $swap\ (a,\ b) = (b,\ a)$ ⟨*proof*⟩
lemma *swap-inj*[*intro*, *simp*]: *inj swap* ⟨*proof*⟩
lemma *swap-surj*[*intro*, *simp*]: *surj swap* ⟨*proof*⟩
lemma *swap-bij*[*intro*, *simp*]: *bij swap* ⟨*proof*⟩

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*]: $push\ ((x,\ y),\ z) = (x,\ y,\ z)$ ⟨*proof*⟩
lemma *pull-simps*[*simp*]: $pull\ (x,\ y,\ z) = ((x,\ y),\ z)$ ⟨*proof*⟩

definition *label* :: $'vertex \times 'label \times 'vertex \Rightarrow 'label$
where *label* $\equiv fst \circ snd$

lemma *label-select*[*simp*]: $label\ (p,\ a,\ q) = a$ ⟨*proof*⟩

5.4 Theorems for @termcurry and @termsplit

lemma *curry-split*[*simp*]: $curry \circ case\text{-}prod = id$ ⟨*proof*⟩
lemma *split-curry*[*simp*]: $case\text{-}prod \circ curry = id$ ⟨*proof*⟩

lemma *curry-le*[*simp*]: $curry\ f \le curry\ g \longleftrightarrow f \le g$ ⟨*proof*⟩
lemma *split-le*[*simp*]: $case\text{-}prod\ f \le case\text{-}prod\ g \longleftrightarrow f \le g$ ⟨*proof*⟩

lemma *mono-curry-left*[*simp*]: $mono\ (curry \circ h) \longleftrightarrow mono\ h$
⟨*proof*⟩
lemma *mono-split-left*[*simp*]: $mono\ (case\text{-}prod \circ h) \longleftrightarrow mono\ h$
⟨*proof*⟩
lemma *mono-curry-right*[*simp*]: $mono\ (h \circ curry) \longleftrightarrow mono\ h$
⟨*proof*⟩
lemma *mono-split-right*[*simp*]: $mono\ (h \circ case\text{-}prod) \longleftrightarrow mono\ h$
⟨*proof*⟩

lemma *Collect-curry*[*simp*]: $\{x.\ P\ (curry\ x)\} = case\text{-}prod\ `\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *Collect-split*[*simp*]: $\{x.\ P\ (case\text{-}prod\ x)\} = curry\ `\ \{x.\ P\ x\}$ ⟨*proof*⟩

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

lemma *not-someI*:
assumes $\bigwedge x.\ P\ x \Longrightarrow False$
shows $\neg\ P\ (SOME\ x.\ P\ x)$
⟨*proof*⟩

lemma *some-ccontr*:
assumes $(\bigwedge x. \neg P\ x) \Longrightarrow False$
shows $P\ (SOME\ x.\ P\ x)$
⟨*proof*⟩

end

6 Relations

theory *Relation-Extensions*
imports
Basic-Extensions
begin

abbreviation *rev-lex-prod* (**infixr** ‹<∗rlex∗>› *80*)
where r_1 <∗rlex∗> r_2 ≡ *inv-image* (r_2 <∗lex∗> 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 $\bigwedge x\ y.\ r\ x\ y \Longrightarrow r\ (f\ x)\ (f\ y)$
shows *liftablep r f*
⟨*proof*⟩
lemma *liftablepE*[*elim*]:
assumes *liftablep r f*
assumes *r x y*
obtains $r\ (f\ x)\ (f\ y)$
⟨*proof*⟩

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

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 $\bigwedge x\ y1\ y2.\ r^{**}\ x\ y1 \Longrightarrow r^{**}\ x\ y2 \Longrightarrow \exists\ z.\ r^{**}\ y1\ z \wedge r^{**}\ y2\ z$
shows *confluentp r*
⟨*proof*⟩

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* $\longleftrightarrow$ *confluentp r2*
$\langle proof \rangle$

definition *diamondp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
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 $R :: 'a \Rightarrow 'a \Rightarrow bool$
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 $w \in cycles\ p$
⟨*proof*⟩
lemma *cyclesE*[*elim*!]:
assumes $w \in cycles\ p$
obtains $path\ w\ p$ $target\ w\ p = p$
⟨*proof*⟩

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

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

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

lemma *visibleI*[*intro*]:
  assumes $en\ a\ q$ $int\ q \neq int\ (ex\ a\ q)$
  shows $a \in visible$
  ⟨*proof*⟩
lemma *visibleE*[*elim*]:
  assumes $a \in visible$
  obtains q
  where $en\ a\ q$ $int\ q \neq int\ (ex\ a\ q)$
  ⟨*proof*⟩

abbreviation *invisible* ≡ − *visible*

lemma *execute-fin-word-invisible*:
  assumes $path\ w\ p$ $set\ w \subseteq invisible$
  shows $int\ (target\ w\ p) = int\ p$
  ⟨*proof*⟩
lemma *execute-inf-word-invisible*:
  assumes $run\ w\ p$ $k \leq l$ $\bigwedge i.\ k \leq i \Longrightarrow i < l \Longrightarrow w\ !!\ i \notin visible$
  shows $int\ ((p\ \#\#\ trace\ w\ p)\ !!\ k) = int\ ((p\ \#\#\ trace\ w\ p)\ !!\ l)$
⟨*proof*⟩

end

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

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

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

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*: $p \in nodes \Longrightarrow w \in cycles\ p \Longrightarrow w \neq [] \Longrightarrow set\ w \cap cuts$
$\neq \{\}$
begin

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

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

  lemma *scut-reachable*:
    assumes *scut p q*
    shows $p \in nodes$ $q \in nodes$
    ⟨*proof*⟩
  lemma *scut-trancl*:
    assumes $scut^{++}\ p\ q$
    obtains *w*
    where $path\ w\ p$ $target\ w\ p = q$ $set\ w \cap cuts = \{\}$ $w \neq []$
  ⟨*proof*⟩

  sublocale *wellfounded-relation* $scut^{-1-1}$
  ⟨*proof*⟩

  lemma *no-cut-scut*:
    assumes $p \in nodes$ $en\ a\ p$ $a \notin cuts$
    shows $scut^{-1-1}\ (ex\ a\ p)\ p$
    ⟨*proof*⟩

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* ⇒ *'item* ⇒ *bool*
assumes *independence-symmetric*[*sym*]: *ind a b* ⟹ *ind b a*
begin

abbreviation *Ind* :: *'item set* ⇒ *'item set* ⇒ *bool*
where *Ind A B* ≡ ∀ *a* ∈ *A*. ∀ *b* ∈ *B*. *ind a b*

inductive *eq-swap* :: *'item list* ⇒ *'item list* ⇒ *bool* (**infix** ‹$=_S$› *50*)
where *swap*: *ind a b* ⟹ *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* ⟹ *w* $=_S$ *v* ⟨*proof*⟩

lemma *eq-swap-length*[*dest*]: $w_1 =_S w_2$ ⟹ *length* w_1 = *length* w_2 ⟨*proof*⟩
lemma *eq-swap-range*[*dest*]: $w_1 =_S w_2$ ⟹ *set* w_1 = *set* w_2 ⟨*proof*⟩

lemma *eq-swap-extend*:
assumes $w_1 =_S w_2$
shows *u* @ w_1 @ *v* $=_S$ *u* @ w_2 @ *v*
⟨*proof*⟩

lemma *eq-swap-remove1*:
assumes $w_1 =_S w_2$
obtains (*equal*) *remove1 c* w_1 = *remove1 c* w_2 | (*swap*) *remove1 c* $w_1 =_S$ *remove1 c* w_2
⟨*proof*⟩

lemma *eq-swap-rev*:
assumes $w_1 =_S w_2$

shows *rev* $w_1 =_S$ *rev* w_2
⟨*proof*⟩

abbreviation *eq-fin* :: *'item list* ⇒ *'item list* ⇒ *bool* (**infix** ‹$=_F$› *50*)
where *eq-fin* ≡ *eq-swap***

lemma *eq-fin-symp*[*intro*, *sym*]: $u =_F v \Longrightarrow v =_F u$
⟨*proof*⟩

lemma *eq-fin-length*[*dest*]: $w_1 =_F w_2 \Longrightarrow$ *length* w_1 = *length* w_2
⟨*proof*⟩
lemma *eq-fin-range*[*dest*]: $w_1 =_F w_2 \Longrightarrow$ *set* w_1 = *set* w_2
⟨*proof*⟩

lemma *eq-fin-remove1*:
assumes $w_1 =_F w_2$
shows *remove1 c* $w_1 =_F$ *remove1 c* w_2
⟨*proof*⟩

lemma *eq-fin-rev*:
assumes $w_1 =_F w_2$
shows *rev* $w_1 =_F$ *rev* w_2
⟨*proof*⟩

lemma *eq-fin-concat-eq-fin-start*:
assumes u @ $v_1 =_F u$ @ v_2
shows $v_1 =_F v_2$
⟨*proof*⟩

lemma *eq-fin-concat*: u @ w_1 @ $v =_F u$ @ w_2 @ $v \longleftrightarrow w_1 =_F w_2$
⟨*proof*⟩
lemma *eq-fin-concat-start*[*iff*]: w @ $w_1 =_F w$ @ $w_2 \longleftrightarrow w_1 =_F w_2$
⟨*proof*⟩
lemma *eq-fin-concat-end*[*iff*]: w_1 @ $w =_F w_2$ @ $w \longleftrightarrow w_1 =_F w_2$
⟨*proof*⟩

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

lemma *ind-eq-fin*[*intro*]:
assumes *Ind* (*set* u) (*set* v)
shows u @ $v =_F v$ @ u
⟨*proof*⟩

definition *le-fin* :: *'item list* ⇒ *'item list* ⇒ *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 $w_1 \preceq_F w_2$
  ⟨*proof*⟩
lemma *le-finE*[*elim 0*]:
  assumes $w_1 \preceq_F w_2$
  obtains v_1
  where w_1 @ $v_1 =_F w_2$
  ⟨*proof*⟩

lemma *le-fin-empty*[*simp*]: $[] \preceq_F w$ ⟨*proof*⟩
lemma *le-fin-trivial*[*intro*]: $w_1 =_F w_2 \Longrightarrow w_1 \preceq_F w_2$
⟨*proof*⟩

lemma *le-fin-length*[*dest*]: $w_1 \preceq_F w_2 \Longrightarrow$ *length* $w_1 \leq$ *length* w_2 ⟨*proof*⟩
lemma *le-fin-range*[*dest*]: $w_1 \preceq_F w_2 \Longrightarrow$ *set* $w_1 \subseteq$ *set* w_2 ⟨*proof*⟩

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*⟩

lemma *le-fin-reflp*[*simp*, *intro*]: $w \preceq_F w$ ⟨*proof*⟩
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*⟩
lemma *eq-fin-le-fin-transp*[*intro*, *trans*]:
  assumes $w_1 =_F w_2$ $w_2 \preceq_F w_3$
  shows $w_1 \preceq_F w_3$
  ⟨*proof*⟩
lemma *le-fin-eq-fin-transp*[*intro*, *trans*]:
  assumes $w_1 \preceq_F w_2$ $w_2 =_F w_3$
  shows $w_1 \preceq_F w_3$
  ⟨*proof*⟩
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*⟩
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*⟩
lemma *prefix-eq-fin-transp*[*intro*, *trans*]:
  assumes $w_1 \leq w_2$ $w_2 =_F w_3$
  shows $w_1 \preceq_F w_3$
  ⟨*proof*⟩

lemma *le-fin-concat-start*[*iff*]: w @ $w_1 \preceq_F w$ @ $w_2 \longleftrightarrow w_1 \preceq_F w_2$
⟨*proof*⟩
lemma *le-fin-concat-end*[*dest*]:

assumes $w_1 \preceq_F w_2$
shows $w_1 \preceq_F w_2 \mathbin{@} w$
⟨*proof*⟩

definition *le-fininf* :: *'item list* ⇒ *'item stream* ⇒ *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-fininfI*[*intro 0*]:
assumes $v_2 \leq_{FI} w_2$ $w_1 \preceq_F v_2$
shows $w_1 \preceq_{FI} w_2$
⟨*proof*⟩
lemma *le-fininfE*[*elim 0*]:
assumes $w_1 \preceq_{FI} w_2$
obtains v_2
where $v_2 \leq_{FI} w_2$ $w_1 \preceq_F v_2$
⟨*proof*⟩

lemma *le-fininf-empty*[*simp*]: $[] \preceq_{FI} w$ ⟨*proof*⟩

lemma *le-fininf-range*[*dest*]: $w_1 \preceq_{FI} w_2 \Longrightarrow$ *set* $w_1 \subseteq$ *sset* w_2 ⟨*proof*⟩

lemma *eq-fin-le-fininf-transp*[*intro*, *trans*]:
assumes $w_1 =_F w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *le-fin-le-fininf-transp*[*intro*, *trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *prefix-le-fininf-transp*[*intro*, *trans*]:
assumes $w_1 \leq w_2$ $w_2 \preceq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *le-fin-prefix-fininf-transp*[*intro*, *trans*]:
assumes $w_1 \preceq_F w_2$ $w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *eq-fin-prefix-fininf-transp*[*intro*, *trans*]:
assumes $w_1 =_F w_2$ $w_2 \leq_{FI} w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩

lemma *le-fininf-concat-start*[*iff*]: $w \mathbin{@} w_1 \preceq_{FI} w \mathbin{@-} w_2 \longleftrightarrow w_1 \preceq_{FI} w_2$
⟨*proof*⟩

lemma *le-fininf-singleton*[*intro*, *simp*]: $[shd\ v] \preceq_{FI} v$
⟨*proof*⟩

definition *le-inf* :: *'item stream* ⇒ *'item stream* ⇒ *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 $\bigwedge v_1.\ v_1 \leq_{FI} w_1 \Longrightarrow v_1 \preceq_{FI} w_2$
shows $w_1 \preceq_I w_2$
⟨*proof*⟩
lemma *le-infE*[*elim 0*]:
assumes $w_1 \preceq_I w_2$ $v_1 \leq_{FI} w_1$
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 $w_1 \leq_{FI} w_2$ $w_2 \preceq_I w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *le-fininf-le-inf-transp*[*intro*, *trans*]:
assumes $w_1 \preceq_{FI} w_2$ $w_2 \preceq_I w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩
lemma *le-inf-transp*[*intro*, *trans*]:
assumes $w_1 \preceq_I w_2$ $w_2 \preceq_I w_3$
shows $w_1 \preceq_I w_3$
⟨*proof*⟩

lemma *le-infI'*:
assumes $\bigwedge k.\ \exists\ v.\ v \leq_{FI} w_1 \wedge k < \mathit{length}\ v \wedge v \preceq_{FI} w_2$
shows $w_1 \preceq_I w_2$
⟨*proof*⟩

lemma *le-infI-chain-left*:
assumes *chain* w $\bigwedge k.\ w\ k \preceq_{FI} v$
shows *limit* $w \preceq_I v$
⟨*proof*⟩
lemma *le-infI-chain-right*:
assumes *chain* w $\bigwedge u.\ u \leq_{FI} v \Longrightarrow u \preceq_F w\ (l\ u)$
shows $v \preceq_I$ *limit* w
⟨*proof*⟩
lemma *le-infI-chain-right'*:
assumes *chain* w $\bigwedge k.\ \mathit{stake}\ k\ v \preceq_F w\ (l\ k)$
shows $v \preceq_I$ *limit* w
⟨*proof*⟩

definition *eq-inf* :: *'item stream* ⇒ *'item stream* ⇒ *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*]: $w_1 =_I w_2 \Longrightarrow$ *sset* w_1 = *sset* w_2 ⟨*proof*⟩

lemma *eq-inf-reflp*[*simp*, *intro*]: $w =_I w$ ⟨*proof*⟩
lemma *eq-inf-symp*[*intro*]: $w_1 =_I w_2 \Longrightarrow w_2 =_I w_1$ ⟨*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 $w_1 \leq_{FI} w_2$ $w_2 =_I w_3$
shows $w_1 \preceq_{FI} w_3$
⟨*proof*⟩

lemma *le-inf-concat-start*[*iff*]: w @− $w_1 \preceq_I w$ @− $w_2 \longleftrightarrow w_1 \preceq_I w_2$
⟨*proof*⟩
lemma *eq-fin-le-inf-concat-end*[*dest*]: $w_1 =_F w_2 \Longrightarrow w_1$ @− $w \preceq_I w_2$ @− w
⟨*proof*⟩

lemma *eq-inf-concat-start*[*iff*]: w @− $w_1 =_I w$ @− $w_2 \longleftrightarrow w_1 =_I w_2$ ⟨*proof*⟩
lemma *eq-inf-concat-end*[*dest*]: $w_1 =_F w_2 \Longrightarrow w_1$ @− $w =_I w_2$ @− 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* ⇒ *'item list* ⇒ *bool*
where *indoc a u* ≡ ∃ u_1 u_2. $u = u_1$ @ [a] @ u_2 ∧ $a \notin$ *set* u_1 ∧ *Ind* {a} (*set* u_1)

lemma *indoc-set*: *indoc a u* ⟹ $a \in$ *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* | (*second*) $a \notin$ *set u* *Ind* {a} (*set u*) *indoc a v*
⟨*proof*⟩

lemma *indoc-single*: *indoc a* [b] ⟷ $a = b$
⟨*proof*⟩

lemma *indoc-append*[*simp*]: *indoc a* (u @ v) ⟷
indoc a u ∨ $a \notin$ *set u* ∧ *Ind* {a} (*set u*) ∧ *indoc a v* ⟨*proof*⟩
lemma *indoc-Nil*[*simp*]: *indoc a* [] ⟷ *False* ⟨*proof*⟩
lemma *indoc-Cons*[*simp*]: *indoc a* (b # v) ⟷ $a = b$ ∨ $a \neq b$ ∧ *ind a b* ∧ *indoc a v*
⟨*proof*⟩

lemma *eq-swap-indoc*: $u =_S v$ ⟹ *indoc c u* ⟹ *indoc c v* ⟨*proof*⟩
lemma *eq-fin-indoc*: $u =_F v$ ⟹ *indoc c u* ⟹ *indoc c v* ⟨*proof*⟩

lemma *eq-fin-ind'*:
  assumes $[a]$ @ $u =_F u_1$ @ $[a]$ @ u_2 $a \notin set\ u_1$
  shows $Ind\ \{a\}\ (set\ u_1)$
⟨*proof*⟩
lemma *eq-fin-ind*:
  assumes u @ $v =_F v$ @ u $set\ u \cap set\ v = \{\}$
  shows $Ind\ (set\ u)\ (set\ v)$
⟨*proof*⟩

lemma *le-fin-member'*:
  assumes $[a] \preceq_F u$ @ v $a \in set\ u$
  shows $[a] \preceq_F u$
⟨*proof*⟩
lemma *le-fin-not-member'*:
  assumes $[a] \preceq_F u$ @ v $a \notin set\ u$
  shows $[a] \preceq_F v$
⟨*proof*⟩
lemma *le-fininf-not-member'*:
  assumes $[a] \preceq_{FI} u$ @$-$ v $a \notin set\ u$
  shows $[a] \preceq_{FI} v$
⟨*proof*⟩

lemma *le-fin-ind''*:
  assumes $[a] \preceq_F w$ $[b] \preceq_F w$ $a \neq b$
  shows $ind\ a\ b$
⟨*proof*⟩
lemma *le-fin-ind'*:
  assumes $[a] \preceq_F w$ $v \preceq_F w$ $a \notin set\ v$
  shows $Ind\ \{a\}\ (set\ v)$
⟨*proof*⟩
lemma *le-fininf-ind''*:
  assumes $[a] \preceq_{FI} w$ $[b] \preceq_{FI} w$ $a \neq b$
  shows $ind\ a\ b$
  ⟨*proof*⟩
lemma *le-fininf-ind'*:
  assumes $[a] \preceq_{FI} w$ $v \preceq_{FI} w$ $a \notin set\ v$
  shows $Ind\ \{a\}\ (set\ v)$
  ⟨*proof*⟩

lemma *indoc-alt-def*: $indoc\ a\ v \longleftrightarrow v =_F [a]$ @ $remove1\ a\ v$
⟨*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 $Ind\ (set\ r)\ (set\ s)$
⟨*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 $\forall\ v.\ v \leq_{FI} w \longrightarrow path\ v\ p$
shows *run w p*
⟨*proof*⟩

lemma (**in** *transition-system*) *words-infI-construct′*:
assumes $\bigwedge\ k.\ \exists\ v.\ v \leq_{FI} w \wedge k < length\ v \wedge path\ v\ p$
shows *run w p*
⟨*proof*⟩

lemma (**in** *transition-system*) *words-infI-construct-chain*[*intro*]:
assumes $chain\ w\ \bigwedge\ k.\ path\ (w\ k)\ p$
shows *run* (*limit w*) *p*
⟨*proof*⟩

lemma (**in** *transition-system*) *words-fin-blocked*:
assumes $\bigwedge\ w.\ path\ w\ p \Longrightarrow A \cap set\ w = \{\} \Longrightarrow A \cap \{a.\ enabled\ a\ (target\ w\ p)\} \subseteq A \cap \{a.\ enabled\ a\ p\}$
assumes $path\ w\ p\ A \cap \{a.\ enabled\ a\ p\} \cap set\ w = \{\}$
shows $A \cap set\ w = \{\}$
⟨*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*: $ind\ a\ b \Longrightarrow en\ a\ p \Longrightarrow en\ b\ p \longleftrightarrow en\ b\ (ex\ a\ p)$
assumes *ex*: $ind\ a\ b \Longrightarrow en\ a\ p \Longrightarrow en\ b\ p \Longrightarrow ex\ b\ (ex\ a\ p) = ex\ a\ (ex\ b\ p)$
begin

lemma *diamond-bottom*:
assumes *ind a b*
assumes *en a p en b p*
shows $en\ a\ (ex\ b\ p)\ en\ b\ (ex\ a\ p)\ ex\ b\ (ex\ a\ p) = ex\ a\ (ex\ b\ p)$

⟨*proof*⟩

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)*
  ⟨*proof*⟩

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)*
  ⟨*proof*⟩

lemma *eq-swap-word*:
  assumes $w_1 =_S w_2$ *path* w_1 *p*
  shows *path* w_2 *p*
  ⟨*proof*⟩

lemma *eq-fin-word*:
  assumes $w_1 =_F w_2$ *path* w_1 *p*
  shows *path* w_2 *p*
  ⟨*proof*⟩

lemma *le-fin-word*:
  assumes $w_1 \preceq_F w_2$ *path* w_2 *p*
  shows *path* w_1 *p*
  ⟨*proof*⟩

lemma *le-fininf-word*:
  assumes $w_1 \preceq_{FI} w_2$ *run* w_2 *p*
  shows *path* w_1 *p*
  ⟨*proof*⟩

lemma *le-inf-word*:
  assumes $w_2 \preceq_I w_1$ *run* w_1 *p*
  shows *run* w_2 *p*
  ⟨*proof*⟩

lemma *eq-inf-word*:
  assumes $w_1 =_I w_2$ *run* w_1 *p*
  shows *run* w_2 *p*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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

$\langle proof \rangle$

lemma *diamond-inf-word-step*:
assumes *Ind* $\{a\}$ (*sset w*) *en a p run w p*
shows *run w* (*ex a p*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

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*
$\langle proof \rangle$

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* $\langle proof \rangle$

lemma *injective-on*:
assumes $x \in A$ $y \in A$ $f\,x = f\,y$
shows $x = y$
$\langle proof \rangle$

end

locale *injective* = *bounded-function* +
assumes *injection*: ∃ *g*. *injection A B f g*
begin

definition *g* ≡ *SOME g*. *injection A B f g*

sublocale *injection A B f g* ⟨*proof*⟩

end

locale *surjection* = *bounded-function-pair* +
assumes *right-inverse*[*simp*]: *y* ∈ *B* ⟹ *f* (*g y*) = *y*
begin

lemma *image-superset*[*intro*]: *f* ‘ *A* ⊇ *B*
⟨*proof*⟩

lemma *image-eq*[*simp*]: *f* ‘ *A* = *B* ⟨*proof*⟩

end

locale *surjective* = *bounded-function* +
assumes *surjection*: ∃ *g*. *surjection A B f g*
begin

definition *g* ≡ *SOME g*. *surjection A B f g*

sublocale *surjection A B f g* ⟨*proof*⟩

end

locale *bijection* = *injection* + *surjection*

lemma *inj-on-bijection*:
assumes *inj-on f A*
shows *bijection A* (*f* ‘ *A*) *f* (*inv-into A f*)
⟨*proof*⟩

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 $\bigwedge n.\ enat\ n \leq m$
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 $C = \{y \in C.\ ord\ x\ y\} \cup \{y \in C.\ ord\ y\ x\}$
⟨*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* $\bigsqcup\ C > x$
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*]: $esize\ x \neq \infty \Longrightarrow finite\ \{y.\ y \leq x\}$
assumes *esize-mono*[*intro*]: $x \leq y \Longrightarrow esize\ x \leq esize\ y$
assumes *esize-strict-mono*[*intro*]: $esize\ x \neq \infty \Longrightarrow x < y \Longrightarrow esize\ x < esize\ y$
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* $C \neq \{\}$
shows $esize\ (\bigsqcup\ C) = \bigsqcup\ (esize\ `\ C)$
$\langle proof \rangle$

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*]: *card* $\{..< enat\ k\} = card\ \{..< k\}$
⟨*proof*⟩
lemma *card-atMost-enat*[*simp*]: *card* $\{.. enat\ k\} = card\ \{.. k\}$
⟨*proof*⟩

lemma *enat-Collect*:
assumes $\infty \notin A$
shows $\{i.\ enat\ i \in A\} = the\text{-}enat$ ‘ A
⟨*proof*⟩

lemma *Collect-lessThan*: $\{i.\ enat\ i < n\} = the\text{-}enat$ ‘ $\{..< n\}$
⟨*proof*⟩

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

function *esize-set* **where** *finite A* $\Longrightarrow$ *esize A* = *enat* (*card A*) | *infinite A* $\Longrightarrow$ *esize A* = ∞
⟨*proof*⟩ **termination** ⟨*proof*⟩

lemma *esize-iff-empty*[*iff*]: *esize A* = *0* $\longleftrightarrow$ *A* = {} ⟨*proof*⟩
lemma *esize-iff-infinite*[*iff*]: *esize A* = ∞ $\longleftrightarrow$ *infinite A* ⟨*proof*⟩
lemma *esize-singleton*[*simp*]: *esize* {*a*} = *eSuc 0* ⟨*proof*⟩
lemma *esize-infinite-enat*[*dest*, *simp*]: *infinite A* $\Longrightarrow$ *enat k* < *esize A* ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *esize-image*[*simp*, *intro*]:
assumes *inj-on f A*
shows *esize* (*f* ‘ *A*) = *esize A*

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

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

lemma *Inf-enat-eSuc*[*simp*]: $\bigsqcap\ (eSuc\ `\ A) = eSuc\ (\bigsqcap\ A)$ $\langle proof \rangle$

definition *lift* :: *nat set* $\Rightarrow$ *nat set*
where $lift\ A \equiv insert\ 0\ (Suc\ `\ A)$

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

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

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

lemma *nth-least-wellformed*[*intro?*, *simp*]:
assumes $enat\ n < esize\ A$
shows $nth\text{-}least\ A\ n \in A$
$\langle proof \rangle$

lemma *card-wellformed*[*intro?*, *simp*]:
fixes k :: *'a* :: *wellorder*

assumes $k \in A$
shows *enat* (*card* $\{i \in A.\ i < k\}$) $<$ *esize* A
⟨*proof*⟩

lemma *nth-least-strict-mono*:
assumes *enat* $l <$ *esize* A $k < l$
shows *nth-least* A $k <$ *nth-least* A l
⟨*proof*⟩

lemma *nth-least-mono*[*intro*, *simp*]:
assumes *enat* $l <$ *esize* A $k \le l$
shows *nth-least* A $k \le$ *nth-least* A l
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *nth-least-strict-mono-inverse*:
  fixes A :: *nat set*
  assumes $enat\ k < esize\ A$ $enat\ l < esize\ A$ $nth\text{-}least\ A\ k < nth\text{-}least\ A\ l$
  shows $k < l$
  ⟨*proof*⟩

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

lemma *nth-least-less-esize-less*:
  $enat\ n < esize\ A \wedge enat\ (nth\text{-}least\ A\ n) < k \longleftrightarrow enat\ n < esize\ \{i \in A.\ enat\ i < k\}$
  ⟨*proof*⟩

lemma *nth-least-le*:
  assumes $enat\ n < esize\ A$
  shows $n \leq nth\text{-}least\ A\ n$
⟨*proof*⟩

lemma *nth-least-eq*:
  assumes $enat\ n < esize\ A$ $enat\ n < esize\ B$
  assumes $\bigwedge i.\ i \leq nth\text{-}least\ A\ n \Longrightarrow i \leq nth\text{-}least\ B\ n \Longrightarrow i \in A \longleftrightarrow i \in B$
  shows $nth\text{-}least\ A\ n = nth\text{-}least\ B\ n$
⟨*proof*⟩

lemma *nth-least-restrict*[*simp*]:
  assumes $enat\ i < esize\ \{i \in s.\ enat\ i < k\}$
  shows $nth\text{-}least\ \{i \in s.\ enat\ i < k\}\ i = nth\text{-}least\ s\ i$
⟨*proof*⟩

lemma *least-nth-least*[*simp*]:
  assumes $A \neq \{\}$ $\bigwedge i.\ i \in A \Longrightarrow enat\ i < esize\ B$
  shows $least\ (nth\text{-}least\ B\ `\ A) = nth\text{-}least\ B\ (least\ A)$
  ⟨*proof*⟩

lemma *nth-least-nth-least*[*simp*]:
  assumes $enat\ n < esize\ A$ $\bigwedge i.\ i \in A \Longrightarrow enat\ i < esize\ B$
  shows $nth\text{-}least\ B\ (nth\text{-}least\ A\ n) = nth\text{-}least\ (nth\text{-}least\ B\ `\ A)\ n$
⟨*proof*⟩

lemma *nth-least-Max*[*simp*]:
  assumes $finite\ A$ $A \neq \{\}$
  shows $nth\text{-}least\ A\ (card\ A - 1) = Max\ A$
⟨*proof*⟩

lemma *nth-least-le-Max*:
  assumes $finite\ A$ $A \neq \{\}$ $enat\ n < esize\ A$

shows *nth-least A n* $\leq$ *Max A*
⟨*proof*⟩

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* $\notin$ *A*
⟨*proof*⟩

lemma *nth-least-Suc*[*simp*]:
assumes *enat n* < *esize A*
shows *nth-least* (*Suc* ‘ *A*) *n* = *Suc* (*nth-least A n*)
⟨*proof*⟩

lemma *nth-least-lift*[*simp*]:
nth-least (*lift A*) *0* = *0*
enat n < *esize A* $\Longrightarrow$ *nth-least* (*lift A*) (*Suc n*) = *Suc* (*nth-least A n*)
⟨*proof*⟩

lemma *nth-least-list-card*[*simp*]:
assumes *enat n* $\leq$ *esize A*
shows *card* {*k* $\in$ *A*. *k* < *nth-least* (*lift A*) *n*} = *n*
⟨*proof*⟩

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* ⟨*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* (*λ x. lprefix* (*f x*) (*g x*))
  ⟨*proof*⟩
lemma *llist-lift-admissible*:
  assumes *ccpo.admissible lSup lprefix P*
  assumes ⋀ *u. u* ≤ *v* ⟹ *lfinite u* ⟹ *P u*
  shows *P v*
  ⟨*proof*⟩

abbreviation *linfinite w* ≡ ¬ *lfinite w*

notation *LNil* (‹<>›)
notation *LCons* (**infixr** ‹%› *65*)
notation *lzip* (**infixr** ‹||› *51*)
notation *lappend* (**infixr** ‹$› *65*)
notation *lnth* (**infixl** ‹?!› *100*)

syntax *-llist* :: *args* ⇒ *'a llist* (‹<->›)
syntax-consts *-llist* ⇌ *LCons*
translations
  <*a*, *x*> ⇌ *a* % <*x*>
  <*a*> ⇌ *a* % <>

lemma *eq-LNil-conv-lnull*[*simp*]: *w* = <> ⟷ *lnull w* ⟨*proof*⟩
lemma *Collect-lnull*[*simp*]: {*w. lnull w*} = {<>} ⟨*proof*⟩

lemma *inj-on-ltake*: *inj-on* (*λ 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 ¬ *lnull w*
  obtains *a v*
  where *w* = <*a*> $ *v*
  ⟨*proof*⟩
lemma *not-lnull-lappend-endE*[*elim*]:
  assumes ¬ *lnull w*
  obtains *a v*
  where *w* = *v* $ <*a*>
⟨*proof*⟩

lemma *llength-lappend-startE*[*elim*]:
  assumes *llength w* ≥ *eSuc n*

obtains *a v*
where *w* = <*a*> $ *v* *llength v* ≥ *n*
⟨*proof*⟩
lemma *llength-lappend-endE*[*elim*]:
assumes *llength w* ≥ *eSuc n*
obtains *a v*
where *w* = *v* $ <*a*> *llength v* ≥ *n*
⟨*proof*⟩

lemma *llength-lappend-start'E*[*elim*]:
assumes *llength w* = *enat* (*Suc n*)
obtains *a v*
where *w* = <*a*> $ *v* *llength v* = *enat n*
⟨*proof*⟩
lemma *llength-lappend-end'E*[*elim*]:
assumes *llength w* = *enat* (*Suc n*)
obtains *a v*
where *w* = *v* $ <*a*> *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 ⋀ *i*. *enat i* < *llength u* ⟹ *enat i* < *llength v* ⟹ *u* ?! *i* = *v* ?! *i*
shows *u* = *v*
⟨*proof*⟩

primcorec *lscan* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a llist* ⇒ *'b* ⇒ *'b llist*
where *lscan f w a* = (*case w of* <> ⇒ <*a*> | *x* % *xs* ⇒ *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*)
⟨*proof*⟩

lemma *lscan-lfinite*[*iff*]: *lfinite* (*lscan f w a*) ⟷ *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 $lfinite\ w \Longrightarrow lfold\ f\ w = fold\ f\ (list\text{-}of\ w) \mid linfinite\ w \Longrightarrow lfold\ f\ w = id$
⟨*proof*⟩ **termination** ⟨*proof*⟩

lemma *lfold-llist-of*[*simp*]: $lfold\ f\ (llist\text{-}of\ xs) = fold\ f\ xs$ ⟨*proof*⟩

lemma *finite-UNIV-llength-eq*:
assumes $finite\ (UNIV :: 'a\ set)$
shows $finite\ \{w :: 'a\ llist.\ llength\ w = enat\ n\}$
⟨*proof*⟩
lemma *finite-UNIV-llength-le*:
assumes $finite\ (UNIV :: 'a\ set)$
shows $finite\ \{w :: 'a\ llist.\ llength\ w \leq enat\ n\}$
⟨*proof*⟩

lemma *lprefix-ltake*[*dest*]: $u \leq v \Longrightarrow u = ltake\ (llength\ u)\ v$
⟨*proof*⟩
lemma *prefixes-set*: $\{v.\ v \leq w\} = \{ltake\ k\ w\ |k.\ k \leq llength\ w\}$ ⟨*proof*⟩
lemma *esize-prefixes*[*simp*]: $esize\ \{v.\ v \leq w\} = eSuc\ (llength\ w)$
⟨*proof*⟩
lemma *prefix-subsume*: $v \leq w \Longrightarrow u \leq w \Longrightarrow llength\ v \leq llength\ u \Longrightarrow v \leq u$
⟨*proof*⟩

lemma *ltake-infinite*[*simp*]: $ltake\ \infty\ w = w$ ⟨*proof*⟩

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

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

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

instance
⟨*proof*⟩

end

14.1 Index Sets

definition *liset* :: $'a\ set \Rightarrow 'a\ llist \Rightarrow nat\ set$
where $liset\ A\ w \equiv \{i.\ enat\ i < llength\ w \wedge w\ ?!\ i \in A\}$

lemma *lisetI*[*intro*]:
assumes $enat\ i < llength\ w$ $w\ ?!\ i \in A$
shows $i \in liset\ A\ w$
⟨*proof*⟩

lemma *lisetD*[*dest*]:
  assumes $i \in$ *liset A w*
  shows *enat i* $<$ *llength w w ?! i* $\in A$
  ⟨*proof*⟩

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

lemma *liset-nil*[*simp*]: *liset A* <> = {} ⟨*proof*⟩
lemma *liset-cons-not-member*[*simp*]:
  assumes $a \notin A$
  shows *liset A* (*a* % *w*) = *Suc* ‘ *liset A w*
⟨*proof*⟩
lemma *liset-cons-member*[*simp*]:
  assumes $a \in A$
  shows *liset A* (*a* % *w*) = {*0*} ∪ *Suc* ‘ *liset A w*
⟨*proof*⟩

lemma *liset-prefix*:
  assumes $i \in$ *liset A v u* $\leq$ *v enat i* $<$ *llength u*
  shows $i \in$ *liset A u*
⟨*proof*⟩
lemma *liset-suffix*:
  assumes $i \in$ *liset A u u* $\leq v$
  shows $i \in$ *liset A v*
⟨*proof*⟩

lemma *liset-ltake*[*simp*]: *liset A* (*ltake* (*enat k*) *w*) = *liset A w* ∩ {..< *k*}
⟨*proof*⟩

lemma *liset-mono*[*dest*]: $u \leq v \Longrightarrow$ *liset A u* $\subseteq$ *liset A v*
  ⟨*proof*⟩
lemma *liset-cont*[*dest*]:
  assumes *Complete-Partial-Order.chain less-eq C C* $\neq$ {}
  shows *liset A* ($\bigsqcup$ *C*) = ($\bigcup$ *w* $\in$ *C*. *liset A w*)
⟨*proof*⟩

lemma *liset-mcont*: *Complete-Partial-Order2.mcont lSup lprefix Sup less-eq* (*liset A*)
  ⟨*proof*⟩

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

14.2 Selections

abbreviation *lproject A* ≡ *lfilter* (λ *a*. *a* ∈ *A*)
abbreviation *lselect s w* ≡ *lnths w s*

lemma *lselect-to-lproject*: *lselect s w* = *lmap fst* (*lproject* (*UNIV* × *s*) (*w* || *iterates Suc 0*))
⟨*proof*⟩
lemma *lproject-to-lselect*: *lproject A w* = *lselect* (*liset A w*) *w*
⟨*proof*⟩

lemma *lproject-llength*[*simp*]: *llength* (*lproject A w*) = *esize* (*liset A w*)
⟨*proof*⟩
lemma *lproject-lfinite*[*simp*]: *lfinite* (*lproject A w*) ⟷ *finite* (*liset A w*)
⟨*proof*⟩

lemma *lselect-restrict-indices*[*simp*]: *lselect* {*i* ∈ *s*. *enat i* < *llength w*} *w* = *lselect s w*
⟨*proof*⟩

lemma *lselect-llength*: *llength* (*lselect s w*) = *esize* {*i* ∈ *s*. *enat i* < *llength w*}
⟨*proof*⟩
lemma *lselect-llength-le*[*simp*]: *llength* (*lselect s w*) ≤ *esize s*
⟨*proof*⟩
lemma *least-lselect-llength*:
assumes ¬ *lnull* (*lselect s w*)
shows *enat* (*least s*) < *llength w*
⟨*proof*⟩
lemma *lselect-lnull*: *lnull* (*lselect s w*) ⟷ (∀ *i* ∈ *s*. *enat i* ≥ *llength w*)
⟨*proof*⟩

lemma *lselect-discard-start*:
assumes ⋀ *i*. *i* ∈ *s* ⟹ *k* ≤ *i*
shows *lselect* {*i*. *k* + *i* ∈ *s*} (*ldropn k w*) = *lselect s w*
⟨*proof*⟩
lemma *lselect-discard-end*:
assumes ⋀ *i*. *i* ∈ *s* ⟹ *i* < *k*
shows *lselect s* (*ltake* (*enat k*) *w*) = *lselect s w*
⟨*proof*⟩

lemma *lselect-least*:
assumes ¬ *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* (*liset A w*) *i*
⟨*proof*⟩

lemma *lproject-ltake*[*simp*]:
assumes $enat\ k \le llength\ (lproject\ A\ w)$
shows $lproject\ A\ (ltake\ (enat\ (nth\text{-}least\ (lift\ (liset\ A\ w))\ k))\ w) =$
$ltake\ (enat\ k)\ (lproject\ A\ w)$
⟨*proof*⟩

lemma *llength-less-llength-lselect-less*:
$enat\ i < esize\ s \wedge enat\ (nth\text{-}least\ s\ i) < llength\ w \longleftrightarrow enat\ i < llength\ (lselect\ s\ w)$
⟨*proof*⟩

lemma *lselect-lselect''*:
assumes $\bigwedge i.\ i \in s \Longrightarrow enat\ i < llength\ w$
assumes $\bigwedge i.\ i \in t \Longrightarrow enat\ i < llength\ (lselect\ s\ w)$
shows $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\text{-}least\ s\ `\ t)\ w$
⟨*proof*⟩

lemma *lselect-lselect'*[*simp*]:
assumes $\bigwedge i.\ i \in t \Longrightarrow enat\ i < esize\ s$
shows $lselect\ t\ (lselect\ s\ w) = lselect\ (nth\text{-}least\ s\ `\ t)\ w$
⟨*proof*⟩

lemma *lselect-lselect*:
$lselect\ t\ (lselect\ s\ w) = lselect\ (nth\text{-}least\ s\ `\ \{i \in t.\ enat\ i < esize\ s\})\ w$
⟨*proof*⟩

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

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

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

lemma *lselect-prefix*[*intro*]:
assumes $u \le v$
shows $lselect\ s\ u \le lselect\ s\ v$
⟨*proof*⟩
lemma *lproject-prefix*[*intro*]:
assumes $u \le v$
shows $lproject\ A\ u \le lproject\ A\ v$
⟨*proof*⟩

lemma *lproject-prefix-limit*[*intro?*]:
assumes $\bigwedge v.\ v \le w \Longrightarrow$ *lfinite* $v \Longrightarrow$ *lproject* $A\ v \le x$
shows *lproject* $A\ w \le x$
⟨*proof*⟩
lemma *lproject-prefix-limit′*:
assumes $\bigwedge k.\ \exists\ v.\ v \le w \wedge$ *enat* $k <$ *llength* $v \wedge$ *lproject* $A\ v \le x$
shows *lproject* $A\ w \le x$
⟨*proof*⟩

end

15 Prefixes on Coinductive Lists

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

lemma *unfold-stream-siterate-smap*: *unfold-stream* $f\ g =$ *smap* $f \circ$ *siterate* g
⟨*proof*⟩

lemma *lappend-stream-of-llist*:
assumes *lfinite* u
shows *stream-of-llist* $(u\ \$\ v) =$ *list-of* u @− *stream-of-llist* v
⟨*proof*⟩

lemma *llist-of-inf-llist-prefix*[*intro*]: $u \le_{FI} v \Longrightarrow$ *llist-of* $u \le$ *llist-of-stream* v
⟨*proof*⟩
lemma *prefix-llist-of-inf-llist*[*intro*]: *lfinite* $u \Longrightarrow u \le v \Longrightarrow$ *list-of* $u \le_{FI}$ *stream-of-llist* v
⟨*proof*⟩

lemma *lproject-prefix-limit-chain*:
assumes *chain* $w\ \bigwedge k.$ *lproject* A (*llist-of* $(w\ k)) \le x$
shows *lproject* A (*llist-of-stream* (*limit* w)) $\le x$
⟨*proof*⟩
lemma *lproject-eq-limit-chain*:
assumes *chain* u *chain* $v\ \bigwedge k.$ *project* $A\ (u\ k) =$ *project* $A\ (v\ k)$
shows *lproject* A (*llist-of-stream* (*limit* u)) $=$ *lproject* A (*llist-of-stream* (*limit* v))
⟨*proof*⟩

end

16 Stuttering

theory *Stuttering*

imports
Stuttering-Equivalence.StutterEquivalence
LList-Prefixes
begin

function *nth-least-ext* :: *nat set* ⇒ *nat* ⇒ *nat*
where
$enat\ k < esize\ A \Longrightarrow nth\text{-}least\text{-}ext\ A\ k = nth\text{-}least\ A\ k\ |$
$enat\ k \geq esize\ A \Longrightarrow nth\text{-}least\text{-}ext\ A\ k = Suc\ (Max\ A + (k - card\ A))$
⟨*proof*⟩ **termination** ⟨*proof*⟩

lemma *nth-least-ext-strict-mono*:
assumes $k < l$
shows $nth\text{-}least\text{-}ext\ s\ k < nth\text{-}least\text{-}ext\ s\ l$
⟨*proof*⟩

definition *stutter-selection* :: *nat set* ⇒ *'a llist* ⇒ *bool*
where $stutter\text{-}selection\ s\ w \equiv 0 \in s\ \wedge$
$(\forall\ k\ i.\ enat\ i < llength\ w \longrightarrow enat\ (Suc\ k) < esize\ s \longrightarrow$
$nth\text{-}least\ s\ k < i \longrightarrow i < nth\text{-}least\ s\ (Suc\ k) \longrightarrow w\ ?!\ i = w\ ?!\ nth\text{-}least\ s\ k)\ \wedge$
$(\forall\ i.\ enat\ i < llength\ w \longrightarrow finite\ s \longrightarrow Max\ s < i \longrightarrow w\ ?!\ i = w\ ?!\ Max\ s)$

lemma *stutter-selectionI*[*intro*]:
assumes $0 \in s$
assumes $\bigwedge k\ i.\ enat\ i < llength\ w \Longrightarrow enat\ (Suc\ k) < esize\ s \Longrightarrow$
$nth\text{-}least\ s\ k < i \Longrightarrow i < nth\text{-}least\ s\ (Suc\ k) \Longrightarrow w\ ?!\ i = w\ ?!\ nth\text{-}least\ s\ k$
assumes $\bigwedge i.\ enat\ i < llength\ w \Longrightarrow finite\ s \Longrightarrow Max\ s < i \Longrightarrow w\ ?!\ i = w$
$?!\ Max\ s$
shows *stutter-selection s w*
⟨*proof*⟩
lemma *stutter-selectionD-0*[*dest*]:
assumes *stutter-selection s w*
shows $0 \in s$
⟨*proof*⟩
lemma *stutter-selectionD-inside*[*dest*]:
assumes *stutter-selection s w*
assumes $enat\ i < llength\ w\ enat\ (Suc\ k) < esize\ s$
assumes $nth\text{-}least\ s\ k < i\ i < nth\text{-}least\ s\ (Suc\ k)$
shows $w\ ?!\ i = w\ ?!\ nth\text{-}least\ s\ k$
⟨*proof*⟩
lemma *stutter-selectionD-infinite*[*dest*]:
assumes *stutter-selection s w*
assumes $enat\ i < llength\ w\ finite\ s\ Max\ s < i$
shows $w\ ?!\ i = w\ ?!\ Max\ s$
⟨*proof*⟩

lemma *stutter-selection-stutter-sampler*[*intro*]:
assumes *linfinite w stutter-selection s w*
shows *stutter-sampler* (*nth-least-ext s*) (*lnth w*)

⟨*proof*⟩

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*]: $w_1 = w_2 \Longrightarrow w_2 \approx w_3 \Longrightarrow w_1 \approx w_3$
⟨*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* ⇒ *'state* ⇒ *'state*
and *en* :: *'action* ⇒ *'state* ⇒ *bool*
and *int* :: *'state* ⇒ *'interpretation*
and *ind* :: *'action* ⇒ *'action* ⇒ *bool*
+
assumes *independence-invisible*: $a \in visible \Longrightarrow b \in visible \Longrightarrow \neg\ ind\ a\ b$
begin

lemma *eq-swap-lproject-visible*:
assumes $u =_S v$
shows *lproject visible* (*llist-of u*) = *lproject visible* (*llist-of v*)
⟨*proof*⟩
lemma *eq-fin-lproject-visible*:
assumes $u =_F v$
shows *lproject visible* (*llist-of u*) = *lproject visible* (*llist-of v*)
⟨*proof*⟩
lemma *le-fin-lproject-visible*:
assumes $u \preceq_F v$
shows *lproject visible* (*llist-of u*) ≤ *lproject visible* (*llist-of v*)
⟨*proof*⟩
lemma *le-fininf-lproject-visible*:
assumes $u \preceq_{FI} v$
shows *lproject visible* (*llist-of u*) ≤ *lproject visible* (*llist-of-stream v*)
⟨*proof*⟩
lemma *le-inf-lproject-visible*:
assumes $u \preceq_I v$
shows *lproject visible* (*llist-of-stream u*) ≤ *lproject visible* (*llist-of-stream v*)
⟨*proof*⟩
lemma *eq-inf-lproject-visible*:
assumes $u =_I v$
shows *lproject visible* (*llist-of-stream u*) = *lproject visible* (*llist-of-stream v*)
⟨*proof*⟩

lemma *stutter-selection-lproject-visible*:
assumes *run u p*
shows *stutter-selection* (*lift* (*liset visible* (*llist-of-stream u*)))
(*llist-of-stream* (*smap int* (*p* ## *trace u p*)))
⟨*proof*⟩

lemma *execute-fin-visible*:
assumes *path u q path v q* $u \preceq_{FI} w$ $v \preceq_{FI} w$
assumes *project visible u* = *project visible v*
shows *int* (*target u q*) = *int* (*target v q*)
⟨*proof*⟩
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 ## 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* :: $'action \Rightarrow 'state \Rightarrow 'state$
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* ≡
$A \subseteq \{a.\ en\ a\ q\} \wedge$
$(A \subset \{a.\ en\ a\ q\} \longrightarrow A \neq \{\}) \wedge$
$(\forall\ a.\ A \subset \{a.\ en\ a\ q\} \longrightarrow a \in A \longrightarrow src\ (ex\ a\ q)\ q) \wedge$
$(A \subset \{a.\ en\ a\ q\} \longrightarrow A \subseteq invisible) \wedge$
$(\forall\ w.\ A \subset \{a.\ en\ a\ q\} \longrightarrow path\ w\ q \longrightarrow A \cap set\ w = \{\} \longrightarrow Ind\ A\ (set\ w))$

lemma *ample-subset*:
assumes *ample-set q A*
shows $A \subseteq \{a.\ en\ a\ q\}$
⟨*proof*⟩

lemma *ample-nonempty*:
assumes *ample-set q A* $A \subset \{a.\ en\ a\ q\}$
shows $A \neq \{\}$
⟨*proof*⟩

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

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

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

lemma *ample-en*[*intro*]: *ample-set* q $\{a.\ en\ a\ q\}$ ⟨*proof*⟩

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

lemma *reduction-words-fin*:
  assumes $q \in nodes$ *R.path* w q
  shows *S.path* w q
  ⟨*proof*⟩
lemma *reduction-words-inf*:
  assumes $q \in nodes$ *R.run* w q
  shows *S.run* w q
  ⟨*proof*⟩

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

(*omitted*) $\{a.\ ren\ a\ q\} \subseteq invisible\ Ind\ \{a.\ ren\ a\ q\}\ (sset\ w)$
⟨*proof*⟩

lemma *reduction-chunk*:
assumes $q \in nodes\ run\ ([a]\ @-\ v)\ q$
obtains $b\ b_1\ b_2\ u$
where
$R.path\ (b\ @\ [a])\ q$
$Ind\ \{a\}\ (set\ b)\ set\ b \subseteq invisible$
$b =_F b_1\ @\ b_2\ b_1\ @-\ u =_I v\ Ind\ (set\ b_2)\ (sset\ u)$
⟨*proof*⟩

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: $reduced\text{-}run\ q\ []\ v\ []\ []\ []\ []\ v\ |$
absorb: $reduced\text{-}run\ q\ v_1\ ([a]\ @-\ v_2)\ l\ w\ w_1\ w_2\ u \Longrightarrow a \in set\ l \Longrightarrow$
$reduced\text{-}run\ q\ (v_1\ @\ [a])\ v_2\ (remove1\ a\ l)\ w\ w_1\ w_2\ u\ |$
extend: $reduced\text{-}run\ q\ v_1\ ([a]\ @-\ v_2)\ l\ w\ w_1\ w_2\ u \Longrightarrow a \notin set\ l \Longrightarrow$
$R.path\ (b\ @\ [a])\ (target\ w\ q) \Longrightarrow$
$Ind\ \{a\}\ (set\ b) \Longrightarrow set\ b \subseteq invisible \Longrightarrow$
$b =_F b_1\ @\ b_2 \Longrightarrow [a]\ @-\ b_1\ @-\ u' =_I u \Longrightarrow Ind\ (set\ b_2)\ (sset\ u') \Longrightarrow$
$reduced\text{-}run\ q\ (v_1\ @\ [a])\ v_2\ (l\ @\ b_1)\ (w\ @\ b\ @\ [a])\ (w_1\ @\ b_1\ @\ [a])\ (w_2\ @\ b_2)\ u'$

lemma *reduced-run-words-fin*:
assumes $reduced\text{-}run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $R.path\ w\ q$
⟨*proof*⟩

lemma *reduced-run-invar-2*:
assumes $reduced\text{-}run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $v_2 =_I l\ @-\ u$
⟨*proof*⟩

lemma *reduced-run-invar-1*:
assumes $reduced\text{-}run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $v_1\ @\ l =_F w_1$
⟨*proof*⟩

lemma *reduced-run-invisible*:
assumes $reduced\text{-}run\ q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$
shows $set\ w_2 \subseteq invisible$
⟨*proof*⟩

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)
⟨*proof*⟩

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

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
⟨*proof*⟩

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
⟨*proof*⟩
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
⟨*proof*⟩

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'
⟨*proof*⟩

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 (*llist-of-stream* u) = *lproject visible* (*llist-of-stream* w)
⟨*proof*⟩

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*⟩

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

⟨*proof*⟩

lemma *reduction-language-stuttering*:
assumes $u \in S.language$
obtains v
where $v \in R.language$ $snth\ u \approx snth\ v$
⟨*proof*⟩

end

end

19 LTL Formulae

theory *Formula*
imports
Basics/Stuttering
Stuttering-Equivalence.PLTL
begin

locale *formula* =
fixes φ :: *'a pltl*
begin

definition *language* :: *'a stream set*
where $language \equiv \{w.\ snth\ w \models_p \varphi\}$

lemma *language-entails*[*iff*]: $w \in language \longleftrightarrow snth\ w \models_p \varphi$ ⟨*proof*⟩

end

locale *formula-next-free* =
formula φ
for φ :: *'a pltl*
\+
assumes *next-free*: *next-free* φ
begin

lemma *stutter-equivalent-entails*[*dest*]: $u \approx v \Longrightarrow u \models_p \varphi \longleftrightarrow v \models_p \varphi$
⟨*proof*⟩

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* φ +
    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 φ :: *'interpretation pltl*
  begin

    lemma *reduction-language-indistinguishable*:
      assumes *R.language* ⊆ *F.language*
      shows *S.language* ⊆ *F.language*
    ⟨*proof*⟩

    theorem *reduction-correct*: *S.language* ⊆ *F.language* ⟷ *R.language* ⊆ *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* ⇒ *'state* ⇒ *'state*
    and *en* :: *'action* ⇒ *'state* ⇒ *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* $scut^{-1-1}$ ⟨*proof*⟩

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 $\bigwedge w.\ path\ w\ s \Longrightarrow A \cap \{a.\ en\ a\ s\} \cap set\ w = \{\} \Longrightarrow A \cap set\ w =$
$\{\}$
shows $ample\text{-}set\ s\ (A \cap \{a.\ en\ a\ s\})$
⟨*proof*⟩

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*: $s \in nodes \Longrightarrow finite\ (procs\ s)$
assumes *psen-en*: $s \in nodes \Longrightarrow pac\ p \cap \{a.\ en\ a\ s\} \subseteq psen\ p\ s$
assumes *psen-ex*: $s \in nodes \Longrightarrow a \in \{a.\ en\ a\ s\} - pac\ p \Longrightarrow psen\ p\ (ex\ a\ s)$
$= psen\ p\ s$
begin

lemma *psen-fin-word*:
assumes $s \in nodes$ $path\ w\ s$ $pac\ p \cap set\ w = \{\}$
shows $psen\ p\ (target\ w\ s) = psen\ p\ s$
⟨*proof*⟩

lemma *en-fin-word*:
assumes $\bigwedge r\ a\ b.\ r \in nodes \Longrightarrow a \in psen\ p\ s - \{a.\ en\ a\ s\} \Longrightarrow b \in \{a.\ en$
$a\ r\} - pac\ p \Longrightarrow$
$en\ a\ (ex\ b\ r) \Longrightarrow en\ a\ r$
assumes $s \in nodes$ $path\ w\ s$ $pac\ p \cap set\ w = \{\}$
shows $pac\ p \cap \{a.\ en\ a\ (target\ w\ s)\} \subseteq pac\ p \cap \{a.\ en\ a\ s\}$
⟨*proof*⟩

lemma *pac-en-blocked*:
assumes $\bigwedge r\ a\ b.\ r \in nodes \Longrightarrow a \in psen\ p\ s - \{a.\ en\ a\ s\} \Longrightarrow b \in \{a.\ en$

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

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.