

Transition Systems and Automata

Julian Brunner

September 13, 2023

Abstract

This entry provides a very abstract theory of transition systems that can be instantiated to express various types of automata. A transition system is typically instantiated by providing a set of initial states, a predicate for enabled transitions, and a transition execution function. From this, it defines the concepts of finite and infinite paths as well as the set of reachable states, among other things. Many useful theorems, from basic path manipulation rules to coinduction and run construction rules, are proven in this abstract transition system context. The library comes with instantiations for DFAs, NFAs, and Büchi automata.

Contents

1	Basics	1
1.1	Miscellaneous	1
2	Finite and Infinite Sequences	2
2.1	List Basics	2
2.2	Stream Basics	4
2.3	The scan Function	8
2.4	Transposing Streams	10
2.5	Distinct Streams	10
2.6	Sorted Streams	11
3	Linear Temporal Logic on Streams	13
3.1	Basics	13
3.2	Infinite Occurrence	14
4	Zipping Sequences	16
4.1	Zipping Lists	16
4.2	Zipping Streams	18
5	Maps	20

6	Basics	20
6.1	Expanding set functions to sets of functions	21
6.2	Expanding set maps into sets of maps	22
7	Transition Systems	29
7.1	Universal Transition Systems	29
7.2	Transition Systems	29
7.3	Transition Systems with Initial States	31
8	Additional Theorems for Transition Systems	32
9	Constructing Paths and Runs in Transition Systems	33
10	Deterministic Automata	36
11	Deterministic Finite Automata	49
12	Nondeterministic Automata	50
13	Nondeterministic Finite Automata	67
14	Deterministic Büchi Automata	68
15	Deterministic Generalized Büchi Automata	69
16	Deterministic Büchi Automata Combinations	69
17	Deterministic Büchi Transition Automata	72
18	Deterministic Generalized Büchi Transition Automata	73
19	Deterministic Büchi Transition Automata Combinations	73
20	Deterministic Co-Büchi Automata	76
21	Deterministic Co-Generalized Co-Büchi Automata	77
22	Deterministic Co-Büchi Automata Combinations	77
23	Deterministic Rabin Automata	80
24	Deterministic Rabin Automata Combinations	81
25	Relations and Refinement	82
25.1	Predicate to Set Conversion Setup	82
25.2	Relation Composition	83
25.3	Relation Basics	84

25.4 Parametricity	84
25.5 Lists	85
25.6 Streams	86
25.7 Functional Relations	88
26 Refinement for Transition Systems	90
27 Relations on Deterministic Rabin Automata	93
28 Implementation	95
28.1 Syntax	95
28.2 Monadic Refinement	95
28.3 Implementations for Sets Represented by Lists	97
28.4 Autoref Setup	99
29 Implementation of Deterministic Rabin Automata	106
30 Exploration of Deterministic Rabin Automata	108
31 Explicit Deterministic Rabin Automata	110
32 Explore and Enumerate Nodes of Deterministic Rabin Automata	115
32.1 Syntax	115
33 Image on Explicit Automata	115
34 Exploration and Translation	115
35 Nondeterministic Büchi Automata	121
36 Nondeterministic Generalized Büchi Automata	123
37 Nondeterministic Büchi Automata Combinations	123
38 Connecting Nondeterministic Büchi Automata to CAVA Automata Structures	126
38.1 Regular Graphs	126
38.2 Indexed Generalized Büchi Graphs	128
39 Relations on Nondeterministic Büchi Automata	129
40 Implementation of Nondeterministic Büchi Automata	131

41 Algorithms on Nondeterministic Büchi Automata	133
41.1 Miscellaneous Amendments	133
41.2 Operations	134
41.3 Implementations	134
42 Explicit Nondeterministic Büchi Automata	137
43 Explore and Enumerate Nodes of Nondeterministic Büchi Automata	140
43.1 Syntax	141
44 Image on Explicit Automata	141
45 Exploration and Translation	141
46 Connecting Nondeterministic Generalized Büchi Automata to CAVA Automata Structures	146
46.1 Regular Graphs	146
46.2 Indexed Generalized Büchi Graphs	148
47 Relations on Nondeterministic Generalized Büchi Automata	150
48 Implementation of Nondeterministic Generalized Büchi Automata	151
49 Algorithms on Nondeterministic Generalized Büchi Automata	154
49.1 Operations	154
49.2 Implementations	154
50 Nondeterministic Büchi Transition Automata	158
51 Nondeterministic Generalized Büchi Transition Automata	159
52 Nondeterministic Büchi Transition Automata Combinations	160

1 Basics

```
theory Basic
imports Main
begin
```

1.1 Miscellaneous

abbreviation (*input*) *const* $x \equiv \lambda -. x$

lemmas [*simp*] = *map-prod.id map-prod.comp[symmetric]*

lemma *prod-UNIV[iff]*: $A \times B = UNIV \longleftrightarrow A = UNIV \wedge B = UNIV$ **by** *auto*

```

lemma prod-singleton:
  fst ‘  $A = \{x\} \implies A = \text{fst} \text{ ‘ } A \times \text{snd} \text{ ‘ } A$ 
  snd ‘  $A = \{y\} \implies A = \text{fst} \text{ ‘ } A \times \text{snd} \text{ ‘ } A$ 
  by force+

lemma infinite-subset[trans]: infinite  $A \implies A \subseteq B \implies \text{infinite } B$  using infinite-super by this
lemma finite-subset[trans]:  $A \subseteq B \implies \text{finite } B \implies \text{finite } A$  using finite-subset by this

declare infinite-coinduct[case-names infinite, coinduct pred: infinite]
lemma infinite-psubset-coinduct[case-names infinite, consumes 1]:
  assumes  $R A$ 
  assumes  $\bigwedge A. R A \implies \exists B \subset A. R B$ 
  shows infinite  $A$ 
proof
  show False if finite  $A$  using that assms by (induct rule: finite-psubset-induct)
(auto)
qed

thm inj-on-subset subset-inj-on

lemma inj-inj-on[dest]: inj  $f \implies \text{inj-on } f S$  using inj-on-subset by auto

end

```

2 Finite and Infinite Sequences

```

theory Sequence
imports
  Basic
  HOL-Library.Stream
  HOL-Library.Monad-Syntax
begin

```

2.1 List Basics

```

declare upt-Suc[simp del]
declare last.simps[simp del]
declare butlast.simps[simp del]
declare Cons-nth-drop-Suc[simp]
declare list.pred-True[simp]

lemma list-pred-cases:
  assumes list-all  $P xs$ 
  obtains (nil)  $xs = [] \mid (\text{cons}) y ys$  where  $xs = y \# ys$   $P y$  list-all  $P ys$ 
  using assms by (cases xs) (auto)

```

lemma *lists-iff-set*: $w \in \text{lists } A \iff \text{set } w \subseteq A$ **by** *auto*

lemma *fold-const*: $\text{fold } \text{const } xs \ a = \text{last } (a \ \# \ xs)$
by (*induct xs arbitrary: a*) (*auto simp: last.simps*)

lemma *take-Suc*: $\text{take } (\text{Suc } n) \ xs = (\text{if } xs = [] \ \text{then } [] \ \text{else } \text{hd } xs \ \# \ \text{take } n \ (\text{tl } xs))$
by (*simp add: take-Suc*)

lemma *bind-map[simp]*: $\text{map } f \ xs \ggg \ g = xs \ggg \ g \circ f$ **unfolding** *List.bind-def*
by *simp*

lemma *ball-bind[iff]*: $\text{Ball } (\text{set } (xs \ggg \ f)) \ P \iff (\forall x \in \text{set } xs. \forall y \in \text{set } (f \ x). P \ y)$
unfolding *set-list-bind* **by** *simp*

lemma *bex-bind[iff]*: $\text{Bex } (\text{set } (xs \ggg \ f)) \ P \iff (\exists x \in \text{set } xs. \exists y \in \text{set } (f \ x). P \ y)$
unfolding *set-list-bind* **by** *simp*

lemma *list-choice*: $\text{list-all } (\lambda x. \exists y. P \ x \ y) \ xs \iff (\exists ys. \text{list-all2 } P \ xs \ ys)$
by (*induct xs*) (*auto simp: list-all2-Cons1*)

lemma *listset-member*: $ys \in \text{listset } XS \iff \text{list-all2 } (\in) \ ys \ XS$
by (*induct XS arbitrary: ys*) (*auto simp: set-Cons-def list-all2-Cons2*)

lemma *listset-empty[iff]*: $\text{listset } XS = \{\} \iff \neg \text{list-all } (\lambda A. A \neq \{\}) \ XS$
by (*induct XS*) (*auto simp: set-Cons-def*)

lemma *listset-finite[iff]*:
assumes $\text{list-all } (\lambda A. A \neq \{\}) \ XS$
shows $\text{finite } (\text{listset } XS) \iff \text{list-all } \text{finite } XS$
using *assms*
proof (*induct XS*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons A XS*)
note [*simp*] = *finite-image-iff finite-cartesian-product-iff*
have $\text{listset } (A \ \# \ XS) = \text{case-prod } \text{Cons } ' (A \times \text{listset } XS)$ **by** (*auto simp: set-Cons-def*)
also have $\text{finite } \dots \iff \text{finite } (A \times \text{listset } XS)$ **by** (*simp add: inj-on-def*)
also have $\dots \iff \text{finite } A \wedge \text{finite } (\text{listset } XS)$ **using** *Cons(2)* **by** *simp*
also have $\text{finite } (\text{listset } XS) \iff \text{list-all } \text{finite } XS$ **using** *Cons* **by** *simp*
also have $\text{finite } A \wedge \dots \iff \text{list-all } \text{finite } (A \ \# \ XS)$ **by** *simp*
finally show *?case* **by** *this*
qed

lemma *listset-finite'[intro]*:
assumes $\text{list-all } \text{finite } XS$
shows $\text{finite } (\text{listset } XS)$
using *infinite-imp-nonempty assms* **by** *blast*

lemma *listset-card[simp]*: $\text{card } (\text{listset } XS) = \text{prod-list } (\text{map } \text{card } XS)$
proof (*induct XS*)

```

    case Nil
    show ?case by simp
  next
  case (Cons A XS)
  have 1: inj (case-prod Cons) unfolding inj-def by simp
  have listset (A # XS) = case-prod Cons ' (A × listset XS) by (auto simp:
set-Cons-def)
  also have card ... = card (A × listset XS) using card-image 1 by auto
  also have ... = card A * card (listset XS) using card-cartesian-product by
this
  also have card (listset XS) = prod-list (map card XS) using Cons by this
  also have card A * ... = prod-list (map card (A # XS)) by simp
  finally show ?case by this
qed

```

2.2 Stream Basics

```

declare stream.map-id[simp]
declare stream.set-map[simp]
declare stream.set-sel(1)[intro!, simp]
declare stream.pred-True[simp]
declare stream.pred-map[iff]
declare stream.rel-map[iff]
declare shift-simps[simp del]
declare stake-sdrop[simp]
declare stake-siterate[simp del]
declare sdrop-snth[simp]

lemma stream-pred-cases:
  assumes pred-stream P xs
  obtains (scons) y ys where xs = y ## ys P y pred-stream P ys
  using assms by (cases xs) (auto)

lemma stream-rel-coinduct[case-names stream-rel, coinduct pred: stream-all2]:
  assumes R u v
  assumes  $\bigwedge a u b v. R (a ## u) (b ## v) \implies P a b \wedge R u v$ 
  shows stream-all2 P u v
  using assms by (coinduct) (metis stream.collapse)
lemma stream-rel-coinduct-shift[case-names stream-rel, consumes 1]:
  assumes R u v
  assumes  $\bigwedge u v. R u v \implies$ 
 $\exists u_1 u_2 v_1 v_2. u = u_1 @- u_2 \wedge v = v_1 @- v_2 \wedge u_1 \neq [] \wedge v_1 \neq [] \wedge$ 
list-all2
P u1 v1  $\wedge$  R u2 v2
  shows stream-all2 P u v
  proof -
    have  $\exists u_1 u_2 v_1 v_2. u = u_1 @- u_2 \wedge v = v_1 @- v_2 \wedge$  list-all2 P u1 v1  $\wedge$  R
u2 v2
    using assms(1) by force
    then show ?thesis using assms(2) by (coinduct) (force elim: list-rel-cases)
  end

```

qed

lemma *stream-pred-coinduct*[*case-names stream-pred, coinduct pred: pred-stream*]:
 assumes $R\ w$
 assumes $\bigwedge a\ w. R\ (a\ \#\# w) \implies P\ a \wedge R\ w$
 shows *pred-stream* $P\ w$
 using *assms unfolding stream.pred-rel eq-onp-def* **by** (*coinduction arbitrary:*
w) (*auto*)

lemma *stream-pred-coinduct-shift*[*case-names stream-pred, consumes 1*]:
 assumes $R\ w$
 assumes $\bigwedge w. R\ w \implies \exists u\ v. w = u\ @-\ v \wedge u \neq [] \wedge \text{list-all}\ P\ u \wedge R\ v$
 shows *pred-stream* $P\ w$

proof –

have $\exists u\ v. w = u\ @-\ v \wedge \text{list-all}\ P\ u \wedge R\ v$
 using *assms(1)* **by** (*metis list-all-simps(2) shift.simps(1)*)
 then show *?thesis* **using** *assms(2)* **by** (*coinduct*) (*force elim: list-pred-cases*)

qed

lemma *stream-pred-flat-coinduct*[*case-names stream-pred, consumes 1*]:
 assumes $R\ ws$
 assumes $\bigwedge w\ ws. R\ (w\ \#\# ws) \implies w \neq [] \wedge \text{list-all}\ P\ w \wedge R\ ws$
 shows *pred-stream* $P\ (\text{flat}\ ws)$
 using *assms*
 by (*coinduction arbitrary: ws rule: stream-pred-coinduct-shift*) (*metis stream.exhaust flat-Stream*)

lemmas *stream-eq-coinduct*[*case-names stream-eq, coinduct pred: HOL.eq*] =
 stream-rel-coinduct[**where** $?P = \text{HOL.eq}$, *unfolded stream.rel-eq*]

lemmas *stream-eq-coinduct-shift*[*case-names stream-eq, consumes 1*] =
 stream-rel-coinduct-shift[**where** $?P = \text{HOL.eq}$, *unfolded stream.rel-eq list.rel-eq*]

lemma *stream-pred-shift*[*iff*]: *pred-stream* $P\ (u\ @-\ v) \longleftrightarrow \text{list-all}\ P\ u \wedge \text{pred-stream}\ P\ v$
 by (*induct u*) (*auto*)

lemma *stream-rel-shift*[*iff*]:

assumes $\text{length}\ u_1 = \text{length}\ v_1$
 shows *stream-all2* $P\ (u_1\ @-\ u_2)\ (v_1\ @-\ v_2) \longleftrightarrow \text{list-all2}\ P\ u_1\ v_1 \wedge \text{stream-all2}\ P\ u_2\ v_2$
 using *assms* **by** (*induct rule: list-induct2*) (*auto*)

lemma *sset-subset-stream-pred*: *sset* $w \subseteq A \longleftrightarrow \text{pred-stream}\ (\lambda a. a \in A)\ w$
 unfolding *stream.pred-set* **by** *auto*

lemma *eq-scons*: $w = a\ \#\# v \longleftrightarrow a = \text{shd}\ w \wedge v = \text{stl}\ w$ **by** *auto*

lemma *scons-eq*: $a\ \#\# v = w \longleftrightarrow \text{shd}\ w = a \wedge \text{stl}\ w = v$ **by** *auto*

lemma *eq-shift*: $w = u\ @-\ v \longleftrightarrow \text{stake}\ (\text{length}\ u)\ w = u \wedge \text{sdrop}\ (\text{length}\ u)\ w = v$
 by (*induct u arbitrary: w*) (*force+*)

lemma *shift-eq*: $u\ @-\ v = w \longleftrightarrow u = \text{stake}\ (\text{length}\ u)\ w \wedge v = \text{sdrop}\ (\text{length}\ u)\ w$

by (induct u arbitrary: w) (force+)
lemma *scons-eq-shift*: $a \#\# w = u @- v \longleftrightarrow (\square = u \wedge a \#\# w = v) \vee (\exists u'. a \# u' = u \wedge w = u' @- v)$
 by (cases u) (auto)
lemma *shift-eq-scons*: $u @- v = a \#\# w \longleftrightarrow (u = \square \wedge v = a \#\# w) \vee (\exists u'. u = a \# u' \wedge u' @- v = w)$
 by (cases u) (auto)

lemma *stream-all2-sset1*:
 assumes *stream-all2* P xs ys
 shows $\forall x \in sset\ xs. \exists y \in sset\ ys. P\ x\ y$
proof –
 have *pred-stream* $(\lambda x. \exists y \in S. P\ x\ y)$ xs **if** $sset\ ys \subseteq S$ **for** S
 using *assms that by* (*coinduction arbitrary: xs ys*) (*force elim: stream.rel-cases*)
 then show *?thesis* **unfolding** *stream.pred-set* **by** *auto*
qed

lemma *stream-all2-sset2*:
 assumes *stream-all2* P xs ys
 shows $\forall y \in sset\ ys. \exists x \in sset\ xs. P\ x\ y$
proof –
 have *pred-stream* $(\lambda y. \exists x \in S. P\ x\ y)$ ys **if** $sset\ xs \subseteq S$ **for** S
 using *assms that by* (*coinduction arbitrary: xs ys*) (*force elim: stream.rel-cases*)
 then show *?thesis* **unfolding** *stream.pred-set* **by** *auto*
qed

lemma *smap-eq-scons[iff]*: $smap\ f\ xs = y \#\# ys \longleftrightarrow f\ (shd\ xs) = y \wedge smap\ f\ (stl\ xs) = ys$
 using *smap-ctr* **by** *metis*
lemma *scons-eq-smap[iff]*: $y \#\# ys = smap\ f\ xs \longleftrightarrow y = f\ (shd\ xs) \wedge ys = smap\ f\ (stl\ xs)$
 using *smap-ctr* **by** *metis*
lemma *smap-eq-shift[iff]*:
 $smap\ f\ w = u @- v \longleftrightarrow (\exists w_1\ w_2. w = w_1 @- w_2 \wedge map\ f\ w_1 = u \wedge smap\ f\ w_2 = v)$
 using *sdrop-smap eq-shift stake-sdrop stake-smap* **by** *metis*
lemma *shift-eq-smap[iff]*:
 $u @- v = smap\ f\ w \longleftrightarrow (\exists w_1\ w_2. w = w_1 @- w_2 \wedge u = map\ f\ w_1 \wedge v = smap\ f\ w_2)$
 using *sdrop-smap eq-shift stake-sdrop stake-smap* **by** *metis*

lemma *szip-eq-scons[iff]*: $szip\ xs\ ys = z \#\# zs \longleftrightarrow (shd\ xs, shd\ ys) = z \wedge szip\ (stl\ xs)\ (stl\ ys) = zs$
 using *szip.ctr stream.inject* **by** *metis*
lemma *scons-eq-szip[iff]*: $z \#\# zs = szip\ xs\ ys \longleftrightarrow z = (shd\ xs, shd\ ys) \wedge zs = szip\ (stl\ xs)\ (stl\ ys)$
 using *szip.ctr stream.inject* **by** *metis*

lemma *siterate-eq-scons[iff]*: $siterate\ f\ s = a \#\# w \longleftrightarrow s = a \wedge siterate\ f\ (f\ s) = w$

using *siterate.ctr stream.inject by metis*
lemma *scons-eq-siterate[iff]*: $a \#\# w = \text{siterate } f \ s \longleftrightarrow a = s \wedge w = \text{siterate } f$
(f s)
using *siterate.ctr stream.inject by metis*

lemma *snth-0*: $(a \#\# w) !! 0 = a$ **by** *simp*
lemma *eqI-snth*:
assumes $\bigwedge i. u !! i = v !! i$
shows $u = v$
using *assms* **by** (*coinduction arbitrary: u v*) (*metis stream.sel snth.simps*)

lemma *stream-pred-snth*: $\text{pred-stream } P \ w \longleftrightarrow (\forall i. P \ (w !! i))$
unfolding *stream.pred-set sset-range* **by** *simp*
lemma *stream-rel-snth*: $\text{stream-all2 } P \ u \ v \longleftrightarrow (\forall i. P \ (u !! i) \ (v !! i))$
proof *safe*
show $P \ (u !! i) \ (v !! i)$ **if** *stream-all2 P u v* **for** i
using *that* **by** (*induct i arbitrary: u v*) (*auto elim: stream.rel-cases*)
show *stream-all2 P u v* **if** $\forall i. P \ (u !! i) \ (v !! i)$
using *that* **by** (*coinduct*) (*metis snth-0 snth-Stream*)
qed

lemma *stream-rel-pred-szip*: $\text{stream-all2 } P \ u \ v \longleftrightarrow \text{pred-stream } (\text{case-prod } P)$
(szip u v)
unfolding *stream-pred-snth stream-rel-snth* **by** *simp*

lemma *sconst-eq[iff]*: $\text{sconst } x = \text{sconst } y \longleftrightarrow x = y$ **by** (*auto*) (*metis siterate.simps(1)*)
lemma *stream-pred--sconst[iff]*: $\text{pred-stream } P \ (\text{sconst } x) \longleftrightarrow P \ x$
unfolding *stream-pred-snth* **by** *simp*
lemma *stream-rel-sconst[iff]*: $\text{stream-all2 } P \ (\text{sconst } x) \ (\text{sconst } y) \longleftrightarrow P \ x \ y$
unfolding *stream-rel-snth* **by** *simp*

lemma *set-sset-stake[intro!, simp]*: $\text{set } (\text{stake } n \ xs) \subseteq \text{sset } xs$
by (*metis sset-shift stake-sdrop sup-ge1*)
lemma *sset-sdrop[intro!, simp]*: $\text{sset } (\text{sdrop } n \ xs) \subseteq \text{sset } xs$
by (*metis sset-shift stake-sdrop sup-ge2*)

lemma *set-stake-snth*: $x \in \text{set } (\text{stake } n \ xs) \longleftrightarrow (\exists i < n. xs !! i = x)$
unfolding *in-set-conv-nth* **by** *auto*

lemma *szip-transfer[transfer-rule]*:
includes *lifting-syntax*
shows ($\text{stream-all2 } A \ \Longrightarrow \ \text{stream-all2 } B \ \Longrightarrow \ \text{stream-all2 } (\text{rel-prod } A \ B)$)
szip szip
by (*intro rel-funI, coinduction*) (*force elim: stream.rel-cases*)
lemma *siterate-transfer[transfer-rule]*:
includes *lifting-syntax*

shows $((A \implies A) \implies A \implies \text{stream-all2 } A)$ *siterate siterate*
by (*intro rel-funI, coinduction*) (*force dest: rel-funD*)

lemma *split-stream-first*:

assumes $A \cap \text{sset } xs \neq \{\}$

obtains $ys \ a \ zs$

where $xs = ys @- a \## zs$ $A \cap \text{set } ys = \{\}$ $a \in A$

proof

let $?n = \text{LEAST } n. xs !! n \in A$

have $1: xs !! n \notin A$ **if** $n < ?n$ **for** n **using** *that* **by** (*metis (full-types)*)
not-less-Least)

show $xs = \text{stake } ?n \ xs @- (xs !! ?n) \## \text{sdrop } (\text{Suc } ?n) \ xs$ **using** *id-stake-snth-sdrop*
by *blast*

show $A \cap \text{set } (\text{stake } ?n \ xs) = \{\}$ **using** 1 **by** (*metis (no-types, lifting) disjoint-iff-not-equal set-stake-snth*)

show $xs !! ?n \in A$ **using** *assms* **unfolding** *sset-range* **by** (*auto intro: LeastI*)

qed

lemma *split-stream-first'*:

assumes $x \in \text{sset } xs$

obtains $ys \ zs$

where $xs = ys @- x \## zs$ $x \notin \text{set } ys$

proof

let $?n = \text{LEAST } n. xs !! n = x$

have $1: xs !! ?n = x$ **using** *assms* **unfolding** *sset-range* **by** (*auto intro: LeastI*)

have $2: xs !! n \neq x$ **if** $n < ?n$ **for** n **using** *that* **by** (*metis (full-types)*)
not-less-Least)

show $xs = \text{stake } ?n \ xs @- x \## \text{sdrop } (\text{Suc } ?n) \ xs$ **using** 1 **by** (*metis id-stake-snth-sdrop*)

show $x \notin \text{set } (\text{stake } ?n \ xs)$ **using** 2 **by** (*meson set-stake-snth*)

qed

lemma *streams-UNIV[iff]*: $\text{streams } A = \text{UNIV} \iff A = \text{UNIV}$

proof

show $A = \text{UNIV} \implies \text{streams } A = \text{UNIV}$ **by** *simp*

next

assume $\text{streams } A = \text{UNIV}$

then have $w \in \text{streams } A$ **for** w **by** *simp*

then have $\text{sset } w \subseteq A$ **for** w **unfolding** *streams-iff-sset* **by** *this*

then have $\text{sset } (\text{sconst } a) \subseteq A$ **for** a **by** *blast*

then have $a \in A$ **for** a **by** *simp*

then show $A = \text{UNIV}$ **by** *auto*

qed

lemma *streams-int[simp]*: $\text{streams } (A \cap B) = \text{streams } A \cap \text{streams } B$ **by** (*auto iff: streams-iff-sset*)

lemma *streams-Int[simp]*: $\text{streams } (\bigcap S) = \bigcap (\text{streams } ` S)$ **by** (*auto iff: streams-iff-sset*)

lemma *pred-list-listsp[pred-set-conv]*: $\text{list-all} = \text{listsp}$

unfolding *list.pred-set* **by** *auto*

lemma *pred-stream-streamsp*[*pred-set-conv*]: *pred-stream* = *streamsp*
unfolding *stream.pred-set streams-iff-sset*[*to-pred*] **by** *auto*

2.3 The scan Function

primrec (*transfer*) *scan* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b list **where**
scan *f* [] *a* = [] | *scan* *f* (*x* # *xs*) *a* = *f* *x* *a* # *scan* *f* *xs* (*f* *x* *a*)

lemma *scan-append*[*simp*]: *scan* *f* (*xs* @ *ys*) *a* = *scan* *f* *xs* *a* @ *scan* *f* *ys* (*fold* *f* *xs* *a*)
by (*induct* *xs* *arbitrary*: *a*) (*auto*)

lemma *scan-eq-nil*[*iff*]: *scan* *f* *xs* *a* = [] ⇔ *xs* = [] **by** (*cases* *xs*) (*auto*)

lemma *scan-eq-cons*[*iff*]:
scan *f* *xs* *a* = *b* # *w* ⇔ (∃ *y* *ys*. *xs* = *y* # *ys* ∧ *f* *y* *a* = *b* ∧ *scan* *f* *ys* (*f* *y* *a*) = *w*)
by (*cases* *xs*) (*auto*)

lemma *scan-eq-append*[*iff*]:
scan *f* *xs* *a* = *u* @ *v* ⇔ (∃ *ys* *zs*. *xs* = *ys* @ *zs* ∧ *scan* *f* *ys* *a* = *u* ∧ *scan* *f* *zs* (*fold* *f* *ys* *a*) = *v*)
by (*induct* *u* *arbitrary*: *xs* *a*) (*auto*, *metis* *append-Cons* *fold-simps*(2), *blast*)

lemma *scan-length*[*simp*]: *length* (*scan* *f* *xs* *a*) = *length* *xs*
by (*induct* *xs* *arbitrary*: *a*) (*auto*)

lemma *scan-last*: *last* (*a* # *scan* *f* *xs* *a*) = *fold* *f* *xs* *a*
by (*induct* *xs* *arbitrary*: *a*) (*auto* *simp*: *last.simps*)

lemma *scan-butlast*[*simp*]: *scan* *f* (*butlast* *xs*) *a* = *butlast* (*scan* *f* *xs* *a*)
by (*induct* *xs* *arbitrary*: *a*) (*auto* *simp*: *butlast.simps*)

lemma *scan-const*[*simp*]: *scan* *const* *xs* *a* = *xs*
by (*induct* *xs* *arbitrary*: *a*) (*auto*)

lemma *scan-nth*[*simp*]:
assumes *i* < *length* (*scan* *f* *xs* *a*)
shows *scan* *f* *xs* *a* ! *i* = *fold* *f* (*take* (*Suc* *i*) *xs*) *a*
using *assms* **by** (*cases* *xs*, *simp*, *induct* *i* *arbitrary*: *xs* *a*, *auto* *simp*: *take-Suc* *neq-Nil-conv*)

lemma *scan-map*[*simp*]: *scan* *f* (*map* *g* *xs*) *a* = *scan* (*f* ∘ *g*) *xs* *a*
by (*induct* *xs* *arbitrary*: *a*) (*auto*)

lemma *scan-take*[*simp*]: *take* *k* (*scan* *f* *xs* *a*) = *scan* *f* (*take* *k* *xs*) *a*
by (*induct* *k* *arbitrary*: *xs* *a*) (*auto* *simp*: *take-Suc* *neq-Nil-conv*)

lemma *scan-drop*[*simp*]: *drop* *k* (*scan* *f* *xs* *a*) = *scan* *f* (*drop* *k* *xs*) (*fold* *f* (*take* *k* *xs*) *a*)
by (*induct* *k* *arbitrary*: *xs* *a*) (*auto* *simp*: *take-Suc* *neq-Nil-conv*)

primcorec (*transfer*) *sscan* :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a stream ⇒ 'b ⇒ 'b stream
where
sscan *f* *xs* *a* = *f* (*shd* *xs*) *a* ## *sscan* *f* (*stl* *xs*) (*f* (*shd* *xs*) *a*)

lemma *sscan-scons*[simp]: $sscan\ f\ (x\ \#\#\ xs)\ a = f\ x\ a\ \#\#\ sscan\ f\ xs\ (f\ x\ a)$
by (*simp add: stream.expand*)
lemma *sscan-shift*[simp]: $sscan\ f\ (xs\ @-\ ys)\ a = scan\ f\ xs\ a\ @-\ sscan\ f\ ys\ (fold\ f\ xs\ a)$
by (*induct xs arbitrary: a*) (*auto*)

lemma *sscan-eq-scons*[iff]:
 $sscan\ f\ xs\ a = b\ \#\#\ w \longleftrightarrow f\ (shd\ xs)\ a = b \wedge sscan\ f\ (stl\ xs)\ (f\ (shd\ xs)\ a)$
 $= w$

using *sscan.ctr stream.inject by metis*

lemma *scons-eq-sscan*[iff]:
 $b\ \#\#\ w = sscan\ f\ xs\ a \longleftrightarrow b = f\ (shd\ xs)\ a \wedge w = sscan\ f\ (stl\ xs)\ (f\ (shd\ xs)\ a)$
using *sscan.ctr stream.inject by metis*

lemma *sscan-const*[simp]: $sscan\ const\ xs\ a = xs$

by (*coinduction arbitrary: xs a*) (*auto*)

lemma *sscan-snth*[simp]: $sscan\ f\ xs\ a\ !!\ i = fold\ f\ (stake\ (Suc\ i)\ xs)\ a$

by (*induct i arbitrary: xs a*) (*auto*)

lemma *sscan-scons-snth*[simp]: $(a\ \#\#\ sscan\ f\ xs\ a)\ !!\ i = fold\ f\ (stake\ i\ xs)\ a$

by (*induct i arbitrary: xs a*) (*auto*)

lemma *sscan-smap*[simp]: $sscan\ f\ (smap\ g\ xs)\ a = sscan\ (f\ \circ\ g)\ xs\ a$

by (*coinduction arbitrary: xs a*) (*auto*)

lemma *sscan-stake*[simp]: $stake\ k\ (sscan\ f\ xs\ a) = scan\ f\ (stake\ k\ xs)\ a$

by (*induct k arbitrary: a xs*) (*auto*)

lemma *sscan-sdrop*[simp]: $sdrop\ k\ (sscan\ f\ xs\ a) = sscan\ f\ (sdrop\ k\ xs)\ (fold\ f\ (stake\ k\ xs)\ a)$

by (*induct k arbitrary: a xs*) (*auto*)

2.4 Transposing Streams

primcorec (*transfer*) *stranspose* :: 'a stream list \Rightarrow 'a list stream **where**

$stranspose\ ws = map\ shd\ ws\ \#\#\ stranspose\ (map\ stl\ ws)$

lemma *stranspose-eq-scons*[iff]: $stranspose\ ws = a\ \#\#\ w \longleftrightarrow map\ shd\ ws = a \wedge stranspose\ (map\ stl\ ws) = w$

using *stranspose.ctr stream.inject by metis*

lemma *scons-eq-stranspose*[iff]: $a\ \#\#\ w = stranspose\ ws \longleftrightarrow a = map\ shd\ ws \wedge w = stranspose\ (map\ stl\ ws)$

using *stranspose.ctr stream.inject by metis*

lemma *stranspose-nil*[simp]: $stranspose\ [] = sconst\ []$ **by** *coinduction auto*

lemma *stranspose-cons*[simp]: $stranspose\ (w\ \#\ ws) = smap2\ Cons\ w\ (stranspose\ ws)$

by (*coinduction arbitrary: w ws*) (*metis list.simps(9) smap2.simps stranspose.simps stream.sel*)

lemma *snth-stranspose*[simp]: $stranspose\ ws\ !!\ k = map\ (\lambda\ w.\ w\ !!\ k)\ ws$ **by** (*induct k arbitrary: ws*) (*auto*)

lemma *stranspose-nth*[*simp*]:
assumes $k < \text{length } ws$
shows $\text{smap } (\lambda xs. xs ! k) (\text{stranspose } ws) = ws ! k$
using *assms* **by** (*auto intro: eqI-snth*)

2.5 Distinct Streams

coinductive *sdistinct* :: 'a stream \Rightarrow bool **where**
scons[*intro!*]: $x \notin \text{sset } xs \Longrightarrow \text{sdistinct } xs \Longrightarrow \text{sdistinct } (x \#\# xs)$

lemma *sdistinct-scons-elim*[*elim!*]:
assumes $\text{sdistinct } (x \#\# xs)$
obtains $x \notin \text{sset } xs \text{ sdistinct } xs$
using *assms* **by** (*auto elim: sdistinct.cases*)

lemma *sdistinct-coinduct*[*case-names sdistinct, coinduct pred: sdistinct*]:
assumes $P \ xs$
assumes $\bigwedge x \ xs. P (x \#\# xs) \Longrightarrow x \notin \text{sset } xs \wedge P \ xs$
shows $\text{sdistinct } xs$
using *stream.collapse sdistinct.coinduct assms* **by** *metis*

lemma *sdistinct-shift*[*intro!*]:
assumes $\text{distinct } xs \text{ sdistinct } ys \text{ set } xs \cap \text{sset } ys = \{\}$
shows $\text{sdistinct } (xs @- ys)$
using *assms* **by** (*induct xs*) (*auto*)

lemma *sdistinct-shift-elim*[*elim!*]:
assumes $\text{sdistinct } (xs @- ys)$
obtains $\text{distinct } xs \text{ sdistinct } ys \text{ set } xs \cap \text{sset } ys = \{\}$
using *assms* **by** (*induct xs*) (*auto*)

lemma *sdistinct-infinite-sset*:
assumes $\text{sdistinct } w$
shows $\text{infinite } (\text{sset } w)$
using *assms* **by** (*coinduction arbitrary: w*) (*force elim: sdistinct.cases*)

lemma *not-sdistinct-decomp*:
assumes $\neg \text{sdistinct } w$
obtains $u \ v \ a \ w'$
where $w = u @- a \#\# v @- a \#\# w'$
proof (*rule ccontr*)
assume *1*: $\neg \text{thesis}$
assume *2*: $w = u @- a \#\# v @- a \#\# w' \Longrightarrow \text{thesis}$ **for** $u \ a \ v \ w'$
have *3*: $\forall u \ v \ a \ w'. w \neq u @- a \#\# v @- a \#\# w'$ **using** *1 2* **by** *auto*
have *4*: $\text{sdistinct } w$ **using** *3* **by** (*coinduct*) (*metis id-stake-snth-sdrop imageE shift.simps sset-range*)
show *False* **using** *assms 4* **by** *auto*
qed

2.6 Sorted Streams

coinductive (in order) *sascending* :: 'a stream \Rightarrow bool **where**
 $a \leq b \implies \text{sascending } (b \#\# w) \implies \text{sascending } (a \#\# b \#\# w)$

coinductive (in order) *sdescending* :: 'a stream \Rightarrow bool **where**
 $a \geq b \implies \text{sdescending } (b \#\# w) \implies \text{sdescending } (a \#\# b \#\# w)$

lemma *sdescending-coinduct*[case-names *sdescending*, coinduct pred: *sdescending*]:

assumes $P w$
assumes $\bigwedge a b w. P (a \#\# b \#\# w) \implies a \geq b \wedge P (b \#\# w)$
shows *sdescending* w
using *stream.collapse sdescending.coinduct assms by (metis (no-types))*

lemma *sdescending-scons*:

assumes *sdescending* $(a \#\# w)$
shows *sdescending* w
using *assms by (auto elim: sdescending.cases)*

lemma *sdescending-sappend*:

assumes *sdescending* $(u @- v)$
obtains *sdescending* v
using *assms by (induct u) (auto elim: sdescending.cases)*

lemma *sdescending-sdrop*:

assumes *sdescending* w
shows *sdescending* $(\text{sdrop } k w)$
using *assms by (metis sdescending-sappend stake-sdrop)*

lemma *sdescending-sset-scons*:

assumes *sdescending* $(a \#\# w)$
assumes $b \in \text{sset } w$
shows $a \geq b$

proof –

have *pred-stream* $(\lambda b. a \geq b) w$ **if** *sdescending* w $a \geq \text{shd } w$ **for** w
using *that by (coinduction arbitrary: w) (auto elim: sdescending.cases)*
then show *?thesis using assms unfolding stream.pred-set by force*

qed

lemma *sdescending-sset-sappend*:

assumes *sdescending* $(u @- v)$
assumes $a \in \text{set } u$ $b \in \text{sset } v$
shows $a \geq b$
using *assms by (induct u) (auto elim: sdescending.cases dest: sdescending-sset-scons)*

lemma *sdescending-snth-antimono*:

assumes *sdescending* w
shows *antimono* $(\text{snth } w)$

unfolding *antimono-iff-le-Suc*

proof

fix k
have *sdescending* $(\text{sdrop } k w)$ **using** *sdescending-sdrop assms by this*

```

    then obtain  $a\ b\ v$  where 2:  $sdrop\ k\ w = a\ \#\#\ b\ \#\#\ v$   $a \geq b$  by rule
    then show  $w !! k \geq w !! Suc\ k$  by (metis sdrop-simps stream.sel)
qed

lemma sdescending-stuck:
  fixes  $w :: 'a :: wellorder\ stream$ 
  assumes sdescending  $w$ 
  obtains  $u\ a$ 
  where  $w = u\ @-\ sconst\ a$ 
using assms
proof (induct shd\ w arbitrary:  $w$  thesis rule: less-induct)
  case less
  show ?case
  proof (cases  $w = sconst\ (shd\ w)$ )
    case True
    show ?thesis using shift-replicate-sconst less(2) True by metis
  next
  case False
  then obtain  $u\ v$  where 1:  $w = u\ @-\ v$   $u \neq []$   $shd\ w \neq shd\ v$ 
  by (metis empty-iff eqI-snth insert-iff sdrop-simps(1) shift.simps(1) snth-sset
sset-sconst stake-sdrop)
  have 2:  $shd\ w \geq shd\ v$  using sdescending-sset-sappend less(3) 1 by (metis
hd-in-set shd-sset shift-simps(1))
  have 3:  $shd\ w > shd\ v$  using 1(3) 2 by simp
  obtain  $s\ a$  where 4:  $v = s\ @-\ sconst\ a$  using sdescending-sappend less(1,
3) 1(1) 3 by metis
  have 5:  $w = (u\ @\ s)\ @-\ sconst\ a$  unfolding 1(1) 4 by simp
  show ?thesis using less(2) 5 by this
qed
qed
end

```

3 Linear Temporal Logic on Streams

```

theory Sequence-LTL
imports
  Sequence
  HOL-Library.Linear-Temporal-Logic-on-Streams
begin

```

3.1 Basics

Avoid destroying the constant *holds* prematurely.

```
lemmas [simp del] = holds.simps holds-eq1 holds-eq2 not-holds-eq
```

```
lemma ev-smap[iff]:  $ev\ P\ (smap\ f\ w) \longleftrightarrow ev\ (P \circ smap\ f)\ w$  using ev-smap
```


unfolding *comp-apply* **by** *this*

lemma *alw-smap*[*iff*]: $alw\ P\ (smap\ f\ w) \longleftrightarrow alw\ (P \circ smap\ f)\ w$ **using** *alw-smap*

unfolding *comp-apply* **by** *this*

lemma *holds-smap*[*iff*]: $holds\ P\ (smap\ f\ w) \longleftrightarrow holds\ (P \circ f)\ w$ **unfolding** *holds.simps* **by** *simp*

lemmas [*iff*] = *ev-sconst alw-sconst hld-smap'*

lemmas [*iff*] = *alw-ev-stl*

lemma *alw-ev-sdrop*[*iff*]: $alw\ (ev\ P)\ (sdrop\ n\ w) \longleftrightarrow alw\ (ev\ P)\ w$

using *alw-ev-sdrop alw-sdrop* **by** *blast*

lemma *alw-ev-scons*[*iff*]: $alw\ (ev\ P)\ (a\ \#\#\ w) \longleftrightarrow alw\ (ev\ P)\ w$ **by** (*metis alw-ev-stl stream.sel(2)*)

lemma *alw-ev-shift*[*iff*]: $alw\ (ev\ P)\ (u\ @-\ v) \longleftrightarrow alw\ (ev\ P)\ v$ **by** (*induct u*) (*auto*)

lemmas [*simp del, iff*] = *ev-alw-stl*

lemma *ev-alw-sdrop*[*iff*]: $ev\ (alw\ P)\ (sdrop\ n\ w) \longleftrightarrow ev\ (alw\ P)\ w$

using *alwD alw-alw alw-sdrop ev-alw-imp-alw-ev not-ev-iff* **by** *metis*

lemma *ev-alw-scons*[*iff*]: $ev\ (alw\ P)\ (a\ \#\#\ w) \longleftrightarrow ev\ (alw\ P)\ w$ **by** (*metis ev-alw-stl stream.sel(2)*)

lemma *ev-alw-shift*[*iff*]: $ev\ (alw\ P)\ (u\ @-\ v) \longleftrightarrow ev\ (alw\ P)\ v$ **by** (*induct u*) (*auto*)

lemma *holds-sconst*[*iff*]: $holds\ P\ (sconst\ a) \longleftrightarrow P\ a$ **unfolding** *holds.simps* **by** *simp*

lemma *HLD-sconst*[*iff*]: $HLD\ A\ (sconst\ a) \longleftrightarrow a \in A$ **unfolding** *HLD-def* *holds.simps* **by** *simp*

lemma *ev-alt-def*: $ev\ \varphi\ w \longleftrightarrow (\exists\ u\ v.\ w = u\ @-\ v \wedge \varphi\ v)$

using *ev.base ev-shift ev-imp-shift* **by** *metis*

lemma *ev-stl-alt-def*: $ev\ \varphi\ (stl\ w) \longleftrightarrow (\exists\ u\ v.\ w = u\ @-\ v \wedge u \neq [] \wedge \varphi\ v)$

unfolding *ev-alt-def* **by** (*cases w*) (*force simp: scons-eq*)

lemma *ev-HLD-sset*: $ev\ (HLD\ A)\ w \longleftrightarrow sset\ w \cap A \neq \{\}$ **unfolding** *HLD-def* *ev-holds-sset* **by** *auto*

lemma *alw-ev-coinduct*[*case-names alw-ev, consumes 1*]:

assumes *R w*

assumes $\bigwedge w.\ R\ w \implies ev\ \varphi\ w \wedge ev\ R\ (stl\ w)$

shows $alw\ (ev\ \varphi)\ w$

proof –

have $ev\ R\ w$ **using** *assms(1)* **by** *rule*

then show *?thesis* **using** *assms(2)* **by** (*coinduct*) (*metis alw-sdrop not-ev-iff sdrop-stl sdrop-wait*)

qed

3.2 Infinite Occurrence

abbreviation $\text{infs } P w \equiv \text{alw } (ev \text{ (holds } P)) w$

abbreviation $\text{fins } P w \equiv \neg \text{infs } P w$

lemma infs-suffix : $\text{infs } P w \longleftrightarrow (\forall u v. w = u @- v \longrightarrow \text{Bex } (sset v) P)$

using $\text{alwD alw-iff-sdrop alw-shift ev-holds-sset stake-sdrop}$ **by** ($\text{metis (mono-tags, opaque-lifting)}$)

lemma infs-snth : $\text{infs } P w \longleftrightarrow (\forall n. \exists k \geq n. P (w !! k))$

by ($\text{auto simp: alw-iff-sdrop ev-iff-sdrop holds.simps intro: le-add1 dest: le-Suc-ex}$)

lemma infs-infm : $\text{infs } P w \longleftrightarrow (\exists_{\infty} i. P (w !! i))$

unfolding $\text{infs-snth INFM-nat-le}$ **by** *rule*

lemma infs-coinduct [$\text{case-names infs, coinduct pred: infs}$]:

assumes $R w$

assumes $\bigwedge w. R w \Longrightarrow \text{Bex } (sset w) P \wedge ev R (stl w)$

shows $\text{infs } P w$

using assms **by** ($\text{coinduct rule: alw-ev-coinduct}$) ($\text{auto simp: ev-holds-sset}$)

lemma $\text{infs-coinduct-shift}$ [$\text{case-names infs, consumes 1}$]:

assumes $R w$

assumes $\bigwedge w. R w \Longrightarrow \exists u v. w = u @- v \wedge \text{Bex } (set u) P \wedge R v$

shows $\text{infs } P w$

using assms **by** (coinduct) ($\text{force simp: ev-stl-alt-def}$)

lemma $\text{infs-flat-coinduct}$ [$\text{case-names infs-flat, consumes 1}$]:

assumes $R w$

assumes $\bigwedge u v. R (u \#\# v) \Longrightarrow \text{Bex } (set u) P \wedge R v$

shows $\text{infs } P (flat w)$

using assms **by** ($\text{coinduction arbitrary: w rule: infs-coinduct-shift}$)

($\text{metis empty-iff flat-Stream list.set(1) stream.exhaust}$)

lemma $\text{infs-sscan-coinduct}$ [$\text{case-names infs-sscan, consumes 1}$]:

assumes $R w a$

assumes $\bigwedge w a. R w a \Longrightarrow P a \wedge (\exists u v. w = u @- v \wedge u \neq [] \wedge R v (fold f u a))$

shows $\text{infs } P (a \#\# \text{sscan } f w a)$

using $\text{assms}(1)$

proof ($\text{coinduction arbitrary: w a rule: infs-coinduct-shift}$)

case ($\text{infs } w a$)

obtain $u v$ **where** $1: P a w = u @- v u \neq [] R v (fold f u a)$ **using** $\text{infs assms}(2)$ **by** *blast*

show $?case$

proof ($\text{intro exI conjI beqI}$)

have $\text{sscan } f w a = \text{scan } f u a @- \text{sscan } f v (fold f u a)$ **unfolding** $1(2)$ **by** *simp*

also have $\text{scan } f u a = \text{butlast } (\text{scan } f u a) @ [fold f u a]$

using $1(3)$ **by** ($\text{metis last-ConsR scan-eq-nil scan-last snoc-eq-iff-butlast}$)

also have $a \#\# \dots @- \text{sscan } f v (fold f u a) =$

$(a \# \text{butlast } (\text{scan } f u a)) @- \text{fold } f u a \#\# \text{sscan } f v (fold f u a)$ **by** *simp*

finally show $a \#\# \text{sscan } f w a = (a \# \text{butlast } (\text{scan } f u a)) @- \text{fold } f u a \#\# \text{sscan } f v (fold f u a)$ **by** *this*

show $P a$ **using** $1(1)$ **by** *this*

```

    show  $a \in \text{set } (a \# \text{butlast } (\text{scan } f \ u \ a))$  by simp
    show  $R \ v \ (\text{fold } f \ u \ a)$  using  $1(4)$  by this
  qed rule
qed

lemma infs-mono:  $(\bigwedge a. a \in \text{sset } w \implies P \ a \implies Q \ a) \implies \text{infs } P \ w \implies \text{infs } Q \ w$ 
  unfolding infs-snth by force
lemma infs-mono-strong: stream-all2  $(\lambda a \ b. P \ a \longrightarrow Q \ b) \ u \ v \implies \text{infs } P \ u \implies \text{infs } Q \ v$ 
  unfolding stream-rel-snth infs-snth by blast

lemma infs-all:  $\text{Ball } (\text{sset } w) \ P \implies \text{infs } P \ w$  unfolding infs-snth by auto
lemma infs-any:  $\text{infs } P \ w \implies \text{Bex } (\text{sset } w) \ P$  unfolding ev-holds-sset by auto

lemma infs-bot[iff]:  $\text{infs } \text{bot } w \longleftrightarrow \text{False}$  using infs-any by auto
lemma infs-top[iff]:  $\text{infs } \text{top } w \longleftrightarrow \text{True}$  by (simp add: infs-all)
lemma infs-disj[iff]:  $\text{infs } (\lambda a. P \ a \vee Q \ a) \ w \longleftrightarrow \text{infs } P \ w \vee \text{infs } Q \ w$ 
  unfolding infs-snth using le-trans le-cases by metis
lemma infs-bex[iff]:
  assumes finite S
  shows  $\text{infs } (\lambda a. \exists x \in S. P \ x \ a) \ w \longleftrightarrow (\exists x \in S. \text{infs } (P \ x) \ w)$ 
  using assms infs-any by induct auto
lemma infs-bex-le-nat[iff]:  $\text{infs } (\lambda a. \exists k < n :: \text{nat}. P \ k \ a) \ w \longleftrightarrow (\exists k < n. \text{infs } (P \ k) \ w)$ 
  proof -
  have  $\text{infs } (\lambda a. \exists k < n. P \ k \ a) \ w \longleftrightarrow \text{infs } (\lambda a. \exists k \in \{k. k < n\}. P \ k \ a) \ w$ 
by simp
  also have  $\dots \longleftrightarrow (\exists k \in \{k. k < n\}. \text{infs } (P \ k) \ w)$  by blast
  also have  $\dots \longleftrightarrow (\exists k < n. \text{infs } (P \ k) \ w)$  by simp
  finally show ?thesis by this
qed

lemma infs-cycle[iff]:
  assumes  $w \neq []$ 
  shows  $\text{infs } P \ (\text{cycle } w) \longleftrightarrow \text{Bex } (\text{set } w) \ P$ 
proof
  show  $\text{infs } P \ (\text{cycle } w) \implies \text{Bex } (\text{set } w) \ P$ 
  using assms by (auto simp: ev-holds-sset dest: alwD)
  show  $\text{Bex } (\text{set } w) \ P \implies \text{infs } P \ (\text{cycle } w)$ 
  using assms by (coinduction rule: infs-coinduct-shift) (blast dest: cycle-decomp)
qed

end

```

4 Zipping Sequences

```

theory Sequence-Zip
imports Sequence-LTL

```

begin

4.1 Zipping Lists

notation *zip* (infixr || 51)

lemmas [*simp*] = *zip-map-fst-snd*

lemma *split-zip*[*no-atp*]: $(\bigwedge x. \text{PROP } P x) \equiv (\bigwedge y z. \text{length } y = \text{length } z \implies \text{PROP } P (y \parallel z))$

proof

fix *y z*

assume 1: $\bigwedge x. \text{PROP } P x$

show *PROP P* (*y || z*) **using** 1 **by** *this*

next

fix *x :: ('a × 'b) list*

assume 1: $\bigwedge y z. \text{length } y = \text{length } z \implies \text{PROP } P (y \parallel z)$

have 2: $\text{length } (\text{map } \text{fst } x) = \text{length } (\text{map } \text{snd } x)$ **by** *simp*

have 3: *PROP P* (*map fst x || map snd x*) **using** 1 2 **by** *this*

show *PROP P x* **using** 3 **by** *simp*

qed

lemma *split-zip-all*[*no-atp*]: $(\forall x. P x) \longleftrightarrow (\forall y z. \text{length } y = \text{length } z \longrightarrow P (y \parallel z))$

by (*fastforce iff: split-zip*)

lemma *split-zip-ex*[*no-atp*]: $(\exists x. P x) \longleftrightarrow (\exists y z. \text{length } y = \text{length } z \wedge P (y \parallel z))$

by (*fastforce iff: split-zip*)

lemma *zip-eq*[*iff*]:

assumes $\text{length } u = \text{length } v \text{ length } r = \text{length } s$

shows $u \parallel v = r \parallel s \longleftrightarrow u = r \wedge v = s$

using *assms zip-eq-conv* **by** *metis*

lemma *list-rel-pred-zip*: $\text{list-all2 } P \text{ } xs \text{ } ys \longleftrightarrow \text{length } xs = \text{length } ys \wedge \text{list-all } (\text{case-prod } P) (xs \parallel ys)$

unfolding *list-all2-conv-all-nth list-all-length* **by** *auto*

lemma *list-choice-zip*: $\text{list-all } (\lambda x. \exists y. P x y) \text{ } xs \longleftrightarrow$

$(\exists ys. \text{length } ys = \text{length } xs \wedge \text{list-all } (\text{case-prod } P) (xs \parallel ys))$

unfolding *list-choice list-rel-pred-zip* **by** *metis*

lemma *list-choice-pair*: $\text{list-all } (\lambda xy. \text{case-prod } (\lambda x y. \exists z. P x y z) \text{ } xy) (xs \parallel ys) \longleftrightarrow$

$(\exists zs. \text{length } zs = \min (\text{length } xs) (\text{length } ys) \wedge \text{list-all } (\lambda (x, y, z). P x y z) (xs \parallel ys \parallel zs))$

proof –

have 1: $\text{list-all } (\lambda (xy, z). \text{case } xy \text{ of } (x, y) \Rightarrow P x y z) ((xs \parallel ys) \parallel zs) \longleftrightarrow$

$\text{list-all } (\lambda (x, y, z). P x y z) (xs \parallel ys \parallel zs)$ **for** *zs*

unfolding *zip-assoc list.pred-map* **by** (*auto intro!: list.pred-cong*)

have 2: $(\lambda (x, y). \exists z. P x y z) = (\lambda xy. \exists z. \text{case } xy \text{ of } (x, y) \Rightarrow P x y z)$ **by**

```

auto
  show ?thesis unfolding list-choice-zip 1 2 by force
qed

lemma list-rel-zip[iff]:
  assumes length u = length v length r = length s
  shows list-all2 (rel-prod A B) (u || v) (r || s)  $\longleftrightarrow$  list-all2 A u r  $\wedge$  list-all2 B
v s
proof safe
  assume [transfer-rule]: list-all2 (rel-prod A B) (u || v) (r || s)
  have list-all2 A (map fst (u || v)) (map fst (r || s)) by transfer-prover
  then show list-all2 A u r using assms by simp
  have list-all2 B (map snd (u || v)) (map snd (r || s)) by transfer-prover
  then show list-all2 B v s using assms by simp
next
  assume [transfer-rule]: list-all2 A u r list-all2 B v s
  show list-all2 (rel-prod A B) (u || v) (r || s) by transfer-prover
qed

lemma zip-last[simp]:
  assumes xs || ys  $\neq$  [] length xs = length ys
  shows last (xs || ys) = (last xs, last ys)
proof -
  have 1: xs  $\neq$  [] ys  $\neq$  [] using assms(1) by auto
  have last (xs || ys) = (xs || ys) ! (length (xs || ys) - 1) using last-conv-nth
assms by blast
  also have ... = (xs ! (length (xs || ys) - 1), ys ! (length (xs || ys) - 1)) using
assms 1 by simp
  also have ... = (xs ! (length xs - 1), ys ! (length ys - 1)) using assms(2)
by simp
  also have ... = (last xs, last ys) using last-conv-nth 1 by metis
  finally show ?thesis by this
qed

```

4.2 Zipping Streams

```

notation szip (infixr ||| 51)

```

```

lemmas [simp] = szip-unfold

```

```

lemma smap-szip-same: smap f (xs ||| xs) = smap ( $\lambda x. f (x, x)$ ) xs by (coinduction
arbitrary: xs) (auto)

```

```

lemma szip-smap[simp]: smap fst zs ||| smap snd zs = zs by (coinduction arbitrary:
zs) (auto)

```

```

lemma szip-smap-fst[simp]: smap fst (xs ||| ys) = xs by (coinduction arbitrary:
xs ys) (auto)

```

```

lemma szip-smap-snd[simp]: smap snd (xs ||| ys) = ys by (coinduction arbitrary:
xs ys) (auto)

```

lemma *szip-smap-both*: $\text{smap } f \text{ } xs \parallel \parallel \text{smap } g \text{ } ys = \text{smap } (\text{map-prod } f \text{ } g) (xs \parallel \parallel ys)$
by (*coinduction arbitrary: xs ys*) (*auto*)
lemma *szip-smap-left*: $\text{smap } f \text{ } xs \parallel \parallel ys = \text{smap } (\text{apfst } f) (xs \parallel \parallel ys)$ **by** (*coinduction arbitrary: xs ys*) (*auto*)
lemma *szip-smap-right*: $xs \parallel \parallel \text{smap } f \text{ } ys = \text{smap } (\text{apsnd } f) (xs \parallel \parallel ys)$ **by** (*coinduction arbitrary: xs ys*) (*auto*)
lemmas *szip-smap-fold* = *szip-smap-both szip-smap-left szip-smap-right*

lemma *szip-sconst-smap-fst*: $\text{sconst } a \parallel \parallel xs = \text{smap } (\text{Pair } a) \text{ } xs$
by (*coinduction arbitrary: xs*) (*auto*)
lemma *szip-sconst-smap-snd*: $xs \parallel \parallel \text{sconst } a = \text{smap } (\text{prod.swap } \circ \text{Pair } a) \text{ } xs$
by (*coinduction arbitrary: xs*) (*auto*)

lemma *split-szip[no-atp]*: $(\bigwedge x. \text{PROP } P \text{ } x) \equiv (\bigwedge y \text{ } z. \text{PROP } P (y \parallel \parallel z))$

proof

fix $y \text{ } z$

assume $1: \bigwedge x. \text{PROP } P \text{ } x$

show $\text{PROP } P (y \parallel \parallel z)$ **using** 1 **by** *this*

next

fix x

assume $1: \bigwedge y \text{ } z. \text{PROP } P (y \parallel \parallel z)$

have $2: \text{PROP } P (\text{smap } \text{fst } x \parallel \parallel \text{smap } \text{snd } x)$ **using** 1 **by** *this*

show $\text{PROP } P \text{ } x$ **using** 2 **by** *simp*

qed

lemma *split-szip-all[no-atp]*: $(\forall x. P \text{ } x) \longleftrightarrow (\forall y \text{ } z. P (y \parallel \parallel z))$ **by** (*fastforce iff: split-szip*)

lemma *split-szip-ex[no-atp]*: $(\exists x. P \text{ } x) \longleftrightarrow (\exists y \text{ } z. P (y \parallel \parallel z))$ **by** (*fastforce iff: split-szip*)

lemma *szip-eq[iff]*: $u \parallel \parallel v = r \parallel \parallel s \longleftrightarrow u = r \wedge v = s$
using *szip-smap-fst szip-smap-snd* **by** *metis*

lemma *stream-rel-szip[iff]*:

$\text{stream-all2 } (\text{rel-prod } A \text{ } B) (u \parallel \parallel v) (r \parallel \parallel s) \longleftrightarrow \text{stream-all2 } A \text{ } u \text{ } r \wedge \text{stream-all2 } B \text{ } v \text{ } s$

proof *safe*

assume [*transfer-rule*]: $\text{stream-all2 } (\text{rel-prod } A \text{ } B) (u \parallel \parallel v) (r \parallel \parallel s)$

have $\text{stream-all2 } A (\text{smap } \text{fst } (u \parallel \parallel v)) (\text{smap } \text{fst } (r \parallel \parallel s))$ **by** *transfer-prover*

then show $\text{stream-all2 } A \text{ } u \text{ } r$ **by** *simp*

have $\text{stream-all2 } B (\text{smap } \text{snd } (u \parallel \parallel v)) (\text{smap } \text{snd } (r \parallel \parallel s))$ **by** *transfer-prover*

then show $\text{stream-all2 } B \text{ } v \text{ } s$ **by** *simp*

next

assume [*transfer-rule*]: $\text{stream-all2 } A \text{ } u \text{ } r \text{ } \text{stream-all2 } B \text{ } v \text{ } s$

show $\text{stream-all2 } (\text{rel-prod } A \text{ } B) (u \parallel \parallel v) (r \parallel \parallel s)$ **by** *transfer-prover*

qed

lemma *szip-shift[simp]*:

assumes $\text{length } u = \text{length } s$

shows $u @- v ||| s @- t = (u || s) @- (v ||| t)$
using *assms* **by** (*simp add: eq-shift stake-shift sdrop-shift*)

lemma *szip-sset-fst[simp]*: $\text{fst } ' \text{sset } (u ||| v) = \text{sset } u$ **by** (*metis stream.set-map szip-smap-fst*)

lemma *szip-sset-snd[simp]*: $\text{snd } ' \text{sset } (u ||| v) = \text{sset } v$ **by** (*metis stream.set-map szip-smap-snd*)

lemma *szip-sset-elim[elim]*:

assumes $(a, b) \in \text{sset } (u ||| v)$

obtains $a \in \text{sset } u$ $b \in \text{sset } v$

using *assms* **by** (*metis image-eqI fst-conv snd-conv szip-sset-fst szip-sset-snd*)

lemma *szip-sset[simp]*: $\text{sset } (u ||| v) \subseteq \text{sset } u \times \text{sset } v$ **by** *auto*

lemma *sset-szip-finite[iff]*: $\text{finite } (\text{sset } (u ||| v)) \longleftrightarrow \text{finite } (\text{sset } u) \wedge \text{finite } (\text{sset } v)$

proof *safe*

assume *1*: $\text{finite } (\text{sset } (u ||| v))$

have *2*: $\text{finite } (\text{fst } ' \text{sset } (u ||| v))$ **using** *1* **by** *blast*

have *3*: $\text{finite } (\text{snd } ' \text{sset } (u ||| v))$ **using** *1* **by** *blast*

show $\text{finite } (\text{sset } u)$ **using** *2* **by** *simp*

show $\text{finite } (\text{sset } v)$ **using** *3* **by** *simp*

next

assume *1*: $\text{finite } (\text{sset } u)$ $\text{finite } (\text{sset } v)$

have $\text{sset } (u ||| v) \subseteq \text{sset } u \times \text{sset } v$ **by** *simp*

also have $\text{finite } \dots$ **using** *1* **by** *simp*

finally show $\text{finite } (\text{sset } (u ||| v))$ **by** *this*

qed

lemma *infs-szip-fst[iff]*: $\text{infs } (P \circ \text{fst}) (u ||| v) \longleftrightarrow \text{infs } P u$

proof *-*

have $\text{infs } (P \circ \text{fst}) (u ||| v) \longleftrightarrow \text{infs } P (\text{smap } \text{fst } (u ||| v))$

by (*simp add: comp-def del: szip-smap-fst*)

also have $\dots \longleftrightarrow \text{infs } P u$ **by** *simp*

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

qed

lemma *infs-szip-snd[iff]*: $\text{infs } (P \circ \text{snd}) (u ||| v) \longleftrightarrow \text{infs } P v$

proof *-*

have $\text{infs } (P \circ \text{snd}) (u ||| v) \longleftrightarrow \text{infs } P (\text{smap } \text{snd } (u ||| v))$

by (*simp add: comp-def del: szip-smap-snd*)

also have $\dots \longleftrightarrow \text{infs } P v$ **by** *simp*

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

qed

end

5 Maps

theory *Maps*

imports *Sequence-Zip*

begin

6 Basics

lemma *fun-upd-None*[simp]:
 assumes $p \notin \text{dom } f$
 shows $f (p := \text{None}) = f$
 using *assms* **by** *auto*

lemma *finite-set-of-finite-maps'*:
 assumes *finite A finite B*
 shows *finite {m. dom m \subseteq A \wedge ran m \subseteq B}*
 proof –
 have $\{m. \text{dom } m \subseteq A \wedge \text{ran } m \subseteq B\} = (\bigcup \mathcal{A} \in \text{Pow } A. \{m. \text{dom } m = \mathcal{A} \wedge \text{ran } m \subseteq B\})$ **by** *auto*
 also have *finite ...* **using** *finite-subset assms* **by** (*auto intro: finite-set-of-finite-maps*)
 finally show *?thesis* **by** *this*
 qed

lemma *fold-map-of*:
 assumes *distinct xs*
 shows $\text{fold } (\lambda x (k, m). (\text{Suc } k, m (x \mapsto k))) \text{ } xs (k, m) =$
 $(k + \text{length } xs, m ++ \text{map-of } (xs \parallel [k ..< k + \text{length } xs]))$
 using *assms*
 proof (*induct xs arbitrary: k m*)
 case *Nil*
 show *?case* **by** *simp*
 next
 case (*Cons x xs*)
 have $\text{fold } (\lambda x (k, m). (\text{Suc } k, m (x \mapsto k))) (x \# xs) (k, m) =$
 $(\text{Suc } k + \text{length } xs, (m ++ \text{map-of } (xs \parallel [\text{Suc } k ..< \text{Suc } k + \text{length } xs]))) (x$
 $\mapsto k)$
 using *Cons* **by** (*fastforce simp add: map-add-upd-left*)
 also have $\dots = (k + \text{length } (x \# xs), m ++ \text{map-of } (x \# xs \parallel [k ..< k +$
 $\text{length } (x \# xs)]))$
 by (*simp add: upt-rec*)
 finally show *?case* **by** *this*
 qed

6.1 Expanding set functions to sets of functions

definition *expand* :: $('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b) \text{ set}$ **where**
 $\text{expand } f = \{g. \forall x. g \ x \in f \ x\}$

lemma *expand-update*[simp]:
 assumes $f \ x \neq \{\}$
 shows $\text{expand } (f (x := S)) = (\bigcup y \in S. (\lambda g. g (x := y))) \text{ } \text{'expand } f$
 unfolding *expand-def*


```

proof (intro equalityI subsetI)
  fix g
  assume 1:  $g \in \{g. \forall y. g y \in (f (x := S)) y\}$ 
  have 2:  $g x \in S \wedge y. x \neq y \implies g y \in f y$  using 1 by (auto split: if-splits)
  obtain y where 3:  $y \in f x$  using assms by auto
  show  $g \in (\bigcup y \in S. (\lambda g. g (x := y))) \text{ ' } \{g. \forall x. g x \in f x\}$ 
  proof (intro UN-I image-eqI)
    show  $g x \in S$  using 2(1) by this
    show  $g (x := y) \in \{g. \forall x. g x \in f x\}$  using 2 3 by auto
    show  $g = g (x := y, x := g x)$  by simp
  qed
next
  fix g
  assume 1:  $g \in (\bigcup y \in S. (\lambda g. g (x := y))) \text{ ' } \{g. \forall x. g x \in f x\}$ 
  show  $g \in \{g. \forall y. g y \in (f (x := S)) y\}$  using 1 by auto
qed

```

6.2 Expanding set maps into sets of maps

definition *expand-map* :: ('a \rightarrow 'b set) \Rightarrow ('a \rightarrow 'b) set **where**
expand-map f \equiv *expand* (case-option {None} (image Some) \circ f)

lemma *expand-map-alt-def*: *expand-map* f =
 $\{g. \text{dom } g = \text{dom } f \wedge (\forall x S y. f x = \text{Some } S \longrightarrow g x = \text{Some } y \longrightarrow y \in S)\}$
unfolding *expand-map-def* *expand-def* **by** (auto split: option.splits) (force+)

lemma *expand-map-dom*:
assumes $g \in \text{expand-map } f$
shows $\text{dom } g = \text{dom } f$
using assms **unfolding** *expand-map-def* *expand-def* **by** (auto split: option.splits)

lemma *expand-map-empty[simp]*: *expand-map* Map.empty = {Map.empty} **unfolding** *expand-map-def* *expand-def* **by** auto

lemma *expand-map-update[simp]*:
 $\text{expand-map } (f (x \mapsto S)) = (\bigcup y \in S. (\lambda g. g (x \mapsto y))) \text{ ' } \text{expand-map } (f (x := \text{None}))$

proof –
let ?m = case-option {None} (image Some)
have 1: $((?m \circ f) (x := \{\text{None}\})) x \neq \{\}$ **by** simp
have *expand-map* (f (x := Some S)) = *expand-map* (f (x := None, x := Some S)) **by** simp
also have ... = *expand* ((?m \circ f) (x := {None}, x := ?m (Some S)))
unfolding *expand-map-def* *fun-upd-comp* **by** simp
also have ... = $(\bigcup y \in ?m (\text{Some } S). (\lambda g. g (x := y))) \text{ ' } \text{expand } ((?m \circ f) (x := \{\text{None}\}))$
using *expand-update* 1 **by** this
also have ... = $(\bigcup y \in S. (\lambda g. g (x \mapsto y))) \text{ ' } \text{expand-map } (f (x := \text{None}))$
unfolding *expand-map-def* *fun-upd-comp* **by** simp
finally show ?thesis **by** this

```

qed

end
theory Acceptance
imports Sequence-LTL
begin

type-synonym 'a pred = 'a  $\Rightarrow$  bool
type-synonym 'a rabin = 'a pred  $\times$  'a pred
type-synonym 'a gen = 'a list

definition rabin :: 'a rabin  $\Rightarrow$  'a stream pred where
  rabin  $\equiv$   $\lambda$  (I, F) w. infs I w  $\wedge$  fins F w

lemma rabin[intro]:
  assumes IF = (I, F) infs I w fins F w
  shows rabin IF w
  using assms unfolding rabin-def by auto
lemma rabin-elim[elim]:
  assumes rabin IF w
  obtains I F
  where IF = (I, F) infs I w fins F w
  using assms unfolding rabin-def by auto

definition gen :: ('a  $\Rightarrow$  'b pred)  $\Rightarrow$  ('a gen  $\Rightarrow$  'b pred) where
  gen P cs w  $\equiv$   $\forall$  c  $\in$  set cs. P c w

lemma gen[intro]:
  assumes  $\bigwedge$  c. c  $\in$  set cs  $\implies$  P c w
  shows gen P cs w
  using assms unfolding gen-def by auto
lemma gen-elim[elim]:
  assumes gen P cs w
  obtains  $\bigwedge$  c. c  $\in$  set cs  $\implies$  P c w
  using assms unfolding gen-def by auto

definition cogen :: ('a  $\Rightarrow$  'b pred)  $\Rightarrow$  ('a gen  $\Rightarrow$  'b pred) where
  cogen P cs w  $\equiv$   $\exists$  c  $\in$  set cs. P c w

lemma cogen[intro]:
  assumes c  $\in$  set cs P c w
  shows cogen P cs w
  using assms unfolding cogen-def by auto
lemma cogen-elim[elim]:
  assumes cogen P cs w
  obtains c
  where c  $\in$  set cs P c w
  using assms unfolding cogen-def by auto

```

lemma *cogen-alt-def*: $\text{cogen } P \text{ cs } w \longleftrightarrow \neg \text{gen } (\lambda c w. \text{Not } (P c w)) \text{ cs } w$ **by**
auto

end

theory *Degeneralization*

imports

Acceptance

Sequence-Zip

begin

type-synonym $'a \text{ degen} = 'a \times \text{nat}$

definition *degen* :: $'a \text{ pred } \text{gen} \Rightarrow 'a \text{ degen } \text{pred}$ **where**
 $\text{degen } cs \equiv \lambda (a, k). k \geq \text{length } cs \vee (cs ! k) a$

lemma *degen-simps*[*iff*]: $\text{degen } cs (a, k) \longleftrightarrow k \geq \text{length } cs \vee (cs ! k) a$ **unfolding**
degen-def **by** *simp*

definition *count* :: $'a \text{ pred } \text{gen} \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{count } cs a k \equiv$
if $k < \text{length } cs$
then *if* $(cs ! k) a$ *then* $\text{Suc } k \text{ mod } \text{length } cs$ *else* k
else *if* $cs = []$ *then* k *else* 0

lemma *count-empty*[*simp*]: $\text{count } [] a k = k$ **unfolding** *count-def* **by** *simp*

lemma *count-nonempty*[*simp*]: $cs \neq [] \Longrightarrow \text{count } cs a k < \text{length } cs$ **unfolding**
count-def **by** *simp*

lemma *count-constant-1*:

assumes $k < \text{length } cs$

assumes $\bigwedge a. a \in \text{set } w \Longrightarrow \neg (cs ! k) a$

shows $\text{fold } (\text{count } cs) w k = k$

using *assms* **unfolding** *count-def* **by** (*induct* w) (*auto*)

lemma *count-constant-2*:

assumes $k < \text{length } cs$

assumes $\bigwedge a. a \in \text{set } (w \parallel k \# \text{scan } (\text{count } cs) w k) \Longrightarrow \neg \text{degen } cs a$

shows $\text{fold } (\text{count } cs) w k = k$

using *assms* **unfolding** *count-def* **by** (*induct* w) (*auto*)

lemma *count-step*:

assumes $k < \text{length } cs$

assumes $(cs ! k) a$

shows $\text{count } cs a k = \text{Suc } k \text{ mod } \text{length } cs$

using *assms* **unfolding** *count-def* **by** *simp*

lemma *degen-skip-condition*:

assumes $k < \text{length } cs$

assumes $\text{infs } (\text{degen } cs) (w \parallel\parallel k \#\#\text{sscan } (\text{count } cs) w k)$

obtains $u a v$

where $w = u @- a \#\# v$ $\text{fold } (\text{count } cs) u k = k (cs ! k) a$

proof –

have 1: $\text{Collect } (\text{degen } cs) \cap \text{sset } (w \parallel k \#\# \text{sscan } (\text{count } cs) w k) \neq \{\}$
using *infs-any assms(2)* **by** *auto*
obtain $ys \ x \ zs$ **where** $2:$
 $w \parallel k \#\# \text{sscan } (\text{count } cs) w k = ys \ @- \ x \#\# \ zs$
 $\text{Collect } (\text{degen } cs) \cap \text{set } ys = \{\}$
 $x \in \text{Collect } (\text{degen } cs)$
using *split-stream-first 1* **by** *this*
define u **where** $u \equiv \text{stake } (\text{length } ys) w$
define a **where** $a \equiv w \ !! \ \text{length } ys$
define v **where** $v \equiv \text{sdrop } (\text{Suc } (\text{length } ys)) w$
have $ys = \text{stake } (\text{length } ys) (w \parallel k \#\# \text{sscan } (\text{count } cs) w k)$ **using** *shift-eq 2(1)* **by** *auto*
also have $\dots = \text{stake } (\text{length } ys) w \ || \ \text{stake } (\text{length } ys) (k \#\# \text{sscan } (\text{count } cs) w k)$ **by** *simp*
also have $\dots = \text{take } (\text{length } ys) u \ || \ \text{take } (\text{length } ys) (k \# \text{scan } (\text{count } cs) u k)$ **unfolding** *u-def*
using *append-eq-conv-conj length-stake length-zip stream.sel*
using *sscan-stake stake.simps(2) stake-Suc stake-szip take-stake*
by *metis*
also have $\dots = \text{take } (\text{length } ys) (u \ || \ k \# \text{scan } (\text{count } cs) u k)$ **using** *take-zip*
by *rule*
also have $\dots = u \ || \ k \# \text{scan } (\text{count } cs) u k$ **unfolding** *u-def* **by** *simp*
finally have $3:$ $ys = u \ || \ k \# \text{scan } (\text{count } cs) u k$ **by** *this*
have $x = (w \parallel k \#\# \text{sscan } (\text{count } cs) w k) \ !! \ \text{length } ys$ **unfolding** $2(1)$ **by** *simp*
also have $\dots = (w \ !! \ \text{length } ys, (k \#\# \text{sscan } (\text{count } cs) w k) \ !! \ \text{length } ys)$ **by** *simp*
also have $\dots = (a, \text{fold } (\text{count } cs) u k)$ **unfolding** *u-def a-def* **by** *simp*
finally have $4:$ $x = (a, \text{fold } (\text{count } cs) u k)$ **by** *this*
have $5:$ $\text{fold } (\text{count } cs) u k = k$ **using** *count-constant-2 assms(1) 2(2)* **unfolding** 3 **by** *blast*
show *?thesis*
proof
show $w = u \ @- \ a \#\# \ v$ **unfolding** *u-def a-def v-def* **using** *id-stake-snth-sdrop*
by *this*
show $\text{fold } (\text{count } cs) u k = k$ **using** 5 **by** *this*
show $(cs \ ! \ k) \ a$ **using** *assms(1) 2(3)* **unfolding** $4 \ 5$ **by** *simp*
qed
qed
lemma *degen-skip-arbitrary:*
assumes $k < \text{length } cs \ l < \text{length } cs$
assumes *infs (degen cs) (w ||| k ## sscan (count cs) w k)*
obtains $u \ v$
where $w = u \ @- \ v \ \text{fold } (\text{count } cs) u k = l$
using *assms*
proof (*induct nat ((int l - int k) mod length cs) arbitrary: l thesis*)
case (0)
have $1:$ $\text{length } cs > 0$ **using** *assms(1)* **by** *auto*
have $2:$ $(\text{int } l - \text{int } k) \ \text{mod } \text{length } cs = 0$ **using** $0(1) \ 1$ **by** (*auto intro: antisym*)

```

have 3:  $\text{int } l \bmod \text{length } cs = \text{int } k \bmod \text{length } cs$  using mod-eq-dvd-iff 2 by
force
have 4:  $k = l$  using 0(3, 4) 3 by simp
show ?case
proof (rule 0(2))
  show  $w = [] @- w$  by simp
  show  $\text{fold } (\text{count } cs) [] k = l$  using 4 by simp
qed
next
case (Suc  $n$ )
have 1:  $\text{length } cs > 0$  using assms(1) by auto
define  $l'$  where  $l' = \text{nat } ((\text{int } l - 1) \bmod \text{length } cs)$ 
obtain  $u v$  where 2:  $w = u @- v$   $\text{fold } (\text{count } cs) u k = l'$ 
proof (rule Suc(1))
  have 2:  $\text{Suc } n < \text{length } cs$  using nat-less-iff Suc(2) 1 by simp
  have  $n = \text{nat } (\text{int } n)$  by simp
  also have  $\text{int } n = (\text{int } (\text{Suc } n) - 1) \bmod \text{length } cs$  using 2 by simp
  also have  $\dots = (\text{int } l - \text{int } k - 1) \bmod \text{length } cs$  using Suc(2) by (simp
add: mod-simps)
  also have  $\dots = (\text{int } l - 1 - \text{int } k) \bmod \text{length } cs$  by (simp add: algebra-simps)
  also have  $\dots = (\text{int } l' - \text{int } k) \bmod \text{length } cs$  using  $l'$ -def 1 by (simp add:
mod-simps)
  finally show  $n = \text{nat } ((\text{int } l' - \text{int } k) \bmod \text{length } cs)$  by this
  show  $k < \text{length } cs$  using Suc(4) by this
  show  $l' < \text{length } cs$  using nat-less-iff  $l'$ -def 1 by simp
  show  $\text{infs } (\text{degen } cs) (w \parallel k \#\# \text{sscan } (\text{count } cs) w k)$  using Suc(6) by this
qed
have 3:  $l' < \text{length } cs$  using nat-less-iff  $l'$ -def 1 by simp
have 4:  $v \parallel l' \#\# \text{sscan } (\text{count } cs) v l' = \text{sdrop } (\text{length } u) (w \parallel k \#\# \text{sscan}$ 
 $(\text{count } cs) w k)$ 
  using 2 eq-scons eq-shift
  by (metis sdrop.simps(2) sdrop-simps sdrop-szip sscan-scons-snth sscan-sdrop
stream.sel(2))
  have 5:  $\text{infs } (\text{degen } cs) (v \parallel l' \#\# \text{sscan } (\text{count } cs) v l')$  using Suc(6)
unfolding 4 by blast
obtain  $vu a vv$  where 6:  $v = vu @- a \#\# vv$   $\text{fold } (\text{count } cs) vu l' = l' (cs !$ 
 $l') a$ 
  using degen-skip-condition 3 5 by this
have  $l = \text{nat } (\text{int } l)$  by simp
also have  $\text{int } l = \text{int } l \bmod \text{length } cs$  using Suc(5) by simp
  also have  $\dots = \text{int } (\text{Suc } l') \bmod \text{length } cs$  using  $l'$ -def 1 by (simp add:
mod-simps)
  finally have 7:  $l = \text{Suc } l' \bmod \text{length } cs$  using nat-mod-as-int by metis
show ?case
proof (rule Suc(3))
  show  $w = (u @ vu @ [a]) @- vv$  unfolding 2(1) 6(1) by simp
  show  $\text{fold } (\text{count } cs) (u @ vu @ [a]) k = l$  using 2(2) 3 6(2, 3) 7 count-step
by simp
qed

```

qed
lemma *degen-skip-arbitrary-condition*:
assumes $l < \text{length } cs$
assumes $\text{infs } (\text{degen } cs) (w \parallel k \#\# \text{sscan } (\text{count } cs) w k)$
obtains $u a v$
where $w = u @- a \#\# v \text{ fold } (\text{count } cs) u k = l (cs ! l) a$
proof –
have $0: cs \neq []$ **using** *assms(1)* **by** *auto*
have $1: \text{count } cs (\text{shd } w) k < \text{length } cs$ **using** 0 **by** *simp*
have $2: \text{infs } (\text{degen } cs) (\text{stl } w \parallel \text{count } cs (\text{shd } w) k \#\# \text{sscan } (\text{count } cs) (\text{stl } w) (\text{count } cs (\text{shd } w) k))$
using *assms(2)* **by** (*metis alw.cases sscan.code stream.sel(2) szip.simps(2)*)
obtain $u v$ **where** $3: \text{stl } w = u @- v \text{ fold } (\text{count } cs) u (\text{count } cs (\text{shd } w) k) = l$
using *degen-skip-arbitrary 1 assms(1) 2 by this*
have $4: v \parallel l \#\# \text{sscan } (\text{count } cs) v l = \text{sdrop } (\text{length } u) (\text{stl } w \parallel \text{count } cs (\text{shd } w) k \#\# \text{sscan } (\text{count } cs) (\text{stl } w) (\text{count } cs (\text{shd } w) k))$
using 3 *eq-scons eq-shift*
by (*metis sdrop.simps(2) sdrop-simps sdrop-szip sscan-scons-snth sscan-sdrop stream.sel(2)*)
have $5: \text{infs } (\text{degen } cs) (v \parallel l \#\# \text{sscan } (\text{count } cs) v l)$ **using** 2 **unfolding** 4 **by** *blast*
obtain $vu a vv$ **where** $6: v = vu @- a \#\# vv \text{ fold } (\text{count } cs) vu l = l (cs ! l) a$
using *degen-skip-condition assms(1) 5 by this*
show *?thesis*
proof
show $w = (\text{shd } w \# u @ vu) @- a \#\# vv$ **using** $3(1) 6(1)$ **by** (*simp add: eq-scons*)
show $\text{fold } (\text{count } cs) (\text{shd } w \# u @ vu) k = l$ **using** $3(2) 6(2)$ **by** *simp*
show $(cs ! l) a$ **using** $6(3)$ **by** *this*
qed
qed
lemma *gen-degen-step*:
assumes $\text{gen } \text{infs } cs w$
obtains $u a v$
where $w = u @- a \#\# v \text{ degen } cs (a, \text{fold } (\text{count } cs) u k)$
proof (*cases k < length cs*)
case *True*
have $1: \text{infs } (cs ! k) w$ **using** *assms True by auto*
have $2: \{a. (cs ! k) a\} \cap \text{sset } w \neq \{\}$ **using** *infs-any 1 by auto*
obtain $u a v$ **where** $3: w = u @- a \#\# v \{a. (cs ! k) a\} \cap \text{set } u = \{a \in \{a. (cs ! k) a\}$
using *split-stream-first 2 by this*
have $4: \text{fold } (\text{count } cs) u k = k$ **using** *count-constant-1 True 3(2)* **by** *auto*
show *?thesis* **using** $3(1, 3) 4$ **that** **by** *simp*
next
case *False*

```

show ?thesis
proof
  show  $w = [] @- shd w ## stl w$  by simp
  show  $degen cs (shd w, fold (count cs) [] k)$  using False by simp
qed
qed

lemma degen-infs[iff]:  $infs (degen cs) (w ||| k ## sscan (count cs) w k) \longleftrightarrow$ 
gen infs cs w
proof
  show gen infs cs w if  $infs (degen cs) (w ||| k ## sscan (count cs) w k)$ 
proof
  fix c
  assume 1:  $c \in set cs$ 
  obtain l where 2:  $c = cs ! l l < length cs$  using in-set-conv-nth 1 by metis
  show  $infs c w$ 
  using that unfolding 2(1)
proof (coinduction arbitrary: w k rule: infs-coinduct-shift)
  case ( $infs w k$ )
  obtain u a v where 3:  $w = u @- a ## v (cs ! l) a$ 
    using degen-skip-arbitrary-condition 2(2) infs by this
  let ?k =  $fold (count cs) u k$ 
  let ?l =  $fold (count cs) (u @ [a]) k$ 
  have 4:  $a ## v ||| ?k ## sscan (count cs) (a ## v) ?k =$ 
     $sdrop (length u) (w ||| k ## sscan (count cs) w k)$ 
    using 3(1) eq-shift sconseq
    by (metis sdrop-simps(1) sdrop-stl sdrop-szip sscan-scons-snth sscan-sdrop
stream.sel(2))
  have 5:  $infs (degen cs) (a ## v ||| ?k ## sscan (count cs) (a ## v) ?k)$ 
    using infs unfolding 4 by blast
  show ?case
proof (intro exI conjI beqI)
  show  $w = (u @ [a]) @- v (cs ! l) a$   $a \in set (u @ [a])$   $v = v$  using 3 by
auto
  show  $infs (degen cs) (v ||| ?l ## sscan (count cs) v ?l)$  using 5 by simp
qed
qed
qed
show  $infs (degen cs) (w ||| k ## sscan (count cs) w k)$  if gen infs cs w
using that
proof (coinduction arbitrary: w k rule: infs-coinduct-shift)
  case ( $infs w k$ )
  obtain u a v where 1:  $w = u @- a ## v degen cs (a, fold (count cs) u k)$ 
    using gen-degen-step infs by this
  let ?u =  $u @ [a] || k \# scan (count cs) u k$ 
  let ?l =  $fold (count cs) (u @ [a]) k$ 
  show ?case
proof (intro exI conjI beqI)
  have  $w ||| k ## sscan (count cs) w k =$ 

```

```

      (u @ [a]) @- v ||| k ## scan (count cs) u k @- ?l ## sscan (count cs)
v ?l
      unfolding 1(1) by simp
      also have ... = ?u @- (v ||| ?l ## sscan (count cs) v ?l)
      by (metis length-Cons length-append-singleton scan-length shift.simps(2)
szip-shift)
      finally show w ||| k ## sscan (count cs) w k = ?u @- (v ||| ?l ## sscan
(count cs) v ?l) by this
      show degenerate cs (a, fold (count cs) u k) using 1(2) by this
      have (a, fold (count cs) u k) = (last (u @ [a]), last (k # scan (count cs) u
k))
      unfolding scan-last by simp
      also have ... = last ?u by (simp add: zip-eq-Nil-iff)
      also have ... ∈ set ?u by (fastforce intro: last-in-set simp: zip-eq-Nil-iff)
      finally show (a, fold (count cs) u k) ∈ set ?u by this
      show v ||| ?l ## sscan (count cs) v ?l = v ||| ?l ## sscan (count cs) v ?l
by rule
      show generate infs cs v using infs unfolding 1(1) by auto
      qed
    qed
  qed
end

```

7 Transition Systems

```

theory Transition-System
imports ../Basic/Sequence
begin

```

7.1 Universal Transition Systems

```

locale transition-system-universal =
  fixes execute :: 'transition ⇒ 'state ⇒ 'state
begin

```

```

  abbreviation target ≡ fold execute
  abbreviation states ≡ scan execute
  abbreviation trace ≡ sscan execute

```

```

  lemma target-alt-def: target r p = last (p # states r p) using scan-last by rule

```

```

end

```

7.2 Transition Systems

```

locale transition-system =
  transition-system-universal execute
  for execute :: 'transition ⇒ 'state ⇒ 'state

```



```

+
fixes enabled :: 'transition ⇒ 'state ⇒ bool
begin

abbreviation successors p ≡ {execute a p | a. enabled a p}

inductive path :: 'transition list ⇒ 'state ⇒ bool where
  nil[intro!]: path [] p |
  cons[intro!]: enabled a p ⇒ path r (execute a p) ⇒ path (a ## r) p

inductive-cases path-cons-elim[elim!]: path (a ## r) p

lemma path-append[intro!]:
  assumes path r p path s (target r p)
  shows path (r @ s) p
  using assms by (induct r arbitrary: p) (auto)
lemma path-append-elim[elim!]:
  assumes path (r @ s) p
  obtains path r p path s (target r p)
  using assms by (induct r arbitrary: p) (auto)

coinductive run :: 'transition stream ⇒ 'state ⇒ bool where
  scons[intro!]: enabled a p ⇒ run r (execute a p) ⇒ run (a ## r) p

inductive-cases run-scons-elim[elim!]: run (a ## r) p

lemma run-shift[intro!]:
  assumes path r p run s (target r p)
  shows run (r @- s) p
  using assms by (induct r arbitrary: p) (auto)
lemma run-shift-elim[elim!]:
  assumes run (r @- s) p
  obtains path r p run s (target r p)
  using assms by (induct r arbitrary: p) (auto)

lemma run-coinduct[case-names run, coinduct pred: run]:
  assumes R r p
  assumes  $\bigwedge a r p. R (a ## r) p \Longrightarrow \text{enabled } a p \wedge R r \text{ (execute } a p)$ 
  shows run r p
  using stream.collapse run.coinduct assms by metis
lemma run-coinduct-shift[case-names run, consumes 1]:
  assumes R r p
  assumes  $\bigwedge r p. R r p \Longrightarrow \exists s t. r = s @- t \wedge s \neq [] \wedge \text{path } s p \wedge R t \text{ (target
s p)
  shows run r p
proof -
  have  $\exists s t. r = s @- t \wedge \text{path } s p \wedge R t \text{ (target } s p)$  using assms(1) by force
  then show ?thesis using assms(2) by (coinduct) (force elim: path.cases)
qed$ 
```

lemma *run-flat-coinduct*[*case-names run, consumes 1*]:
assumes $R\ rs\ p$
assumes $\bigwedge r\ rs\ p. R\ (r\ \#\#\ rs)\ p \implies r \neq [] \wedge path\ r\ p \wedge R\ rs\ (target\ r\ p)$
shows $run\ (flat\ rs)\ p$
using *assms(1)*
proof (*coinduction arbitrary: rs p rule: run-coinduct-shift*)
case (*run rs p*)
then show *?case using assms(2) by (metis stream.exhaust flat-Stream)*
qed

inductive-set *reachable* :: 'state \Rightarrow 'state set **for** *p* **where**
reflexive[*intro!*]: $p \in reachable\ p \mid$
execute[*intro!*]: $q \in reachable\ p \implies enabled\ a\ q \implies execute\ a\ q \in reachable\ p$

inductive-cases *reachable-elim*[*elim*]: $q \in reachable\ p$

lemma *reachable-execute'*[*intro*]:
assumes $enabled\ a\ p\ q \in reachable\ (execute\ a\ p)$
shows $q \in reachable\ p$
using *assms(2, 1) by induct auto*
lemma *reachable-elim'*[*elim*]:
assumes $q \in reachable\ p$
obtains $q = p \mid a$ **where** $enabled\ a\ p\ q \in reachable\ (execute\ a\ p)$
using *assms by induct auto*

lemma *reachable-target*[*intro*]:
assumes $q \in reachable\ p\ path\ r\ q$
shows $target\ r\ q \in reachable\ p$
using *assms by (induct r arbitrary: q) (auto)*

lemma *reachable-target-elim*[*elim*]:
assumes $q \in reachable\ p$
obtains r
where $path\ r\ p\ q = target\ r\ p$
using *assms by induct force+*

lemma *reachable-alt-def*: $reachable\ p = \{target\ r\ p \mid r. path\ r\ p\}$ **by** *auto*

lemma *reachable-trans*[*trans*]: $q \in reachable\ p \implies s \in reachable\ q \implies s \in reachable\ p$ **by** *auto*

lemma *reachable-successors*[*intro!*]: $successors\ p \subseteq reachable\ p$ **by** *auto*

lemma *reachable-step*: $reachable\ p = insert\ p\ (\bigcup (reachable\ ' successors\ p))$ **by** *auto*

end

7.3 Transition Systems with Initial States

```
locale transition-system-initial =
  transition-system execute enabled
  for execute :: 'transition  $\Rightarrow$  'state  $\Rightarrow$  'state
  and enabled :: 'transition  $\Rightarrow$  'state  $\Rightarrow$  bool
  +
  fixes initial :: 'state  $\Rightarrow$  bool
begin

  inductive-set nodes :: 'state set where
    initial[intro]: initial p  $\Longrightarrow$  p  $\in$  nodes |
    execute[intro!]: p  $\in$  nodes  $\Longrightarrow$  enabled a p  $\Longrightarrow$  execute a p  $\in$  nodes

  lemma nodes-target[intro]:
    assumes p  $\in$  nodes path r p
    shows target r p  $\in$  nodes
    using assms by (induct r arbitrary: p) (auto)
  lemma nodes-target-elim[elim]:
    assumes q  $\in$  nodes
    obtains r p
    where initial p path r p q = target r p
    using assms by induct force+

  lemma nodes-alt-def: nodes =  $\bigcup$  (reachable 'Collect initial) by auto

  lemma nodes-trans[trans]: p  $\in$  nodes  $\Longrightarrow$  q  $\in$  reachable p  $\Longrightarrow$  q  $\in$  nodes by
  auto

end

end
```

8 Additional Theorems for Transition Systems

```
theory Transition-System-Extra
imports
  ../Basic/Sequence-LTL
  Transition-System
begin

  context transition-system
  begin

    definition enableds p  $\equiv$  {a. enabled a p}
    definition paths p  $\equiv$  {r. path r p}
    definition runs p  $\equiv$  {r. run r p}

    lemma stake-run:
```

```

assumes  $\bigwedge k. \text{path } (\text{stake } k \ r) \ p$ 
shows  $\text{run } r \ p$ 
using assms by (coinduction arbitrary: r p) (force elim: path.cases)
lemma snth-run:
assumes  $\bigwedge k. \text{enabled } (r \ !! \ k) \ (\text{target } (\text{stake } k \ r) \ p)$ 
shows  $\text{run } r \ p$ 
using assms by (coinduction arbitrary: r p) (metis stream.sel fold-simps
snth.simps stake.simps)

lemma run-stake:
assumes  $\text{run } r \ p$ 
shows  $\text{path } (\text{stake } k \ r) \ p$ 
using assms by (metis run-shift-elim stake-sdrop)
lemma run-sdrop:
assumes  $\text{run } r \ p$ 
shows  $\text{run } (\text{sdrop } k \ r) \ (\text{target } (\text{stake } k \ r) \ p)$ 
using assms by (metis run-shift-elim stake-sdrop)
lemma run-snth:
assumes  $\text{run } r \ p$ 
shows  $\text{enabled } (r \ !! \ k) \ (\text{target } (\text{stake } k \ r) \ p)$ 
using assms by (metis stream.collapse sdrop-simps(1) run-scons-elim run-sdrop)

lemma run-alt-def-snth:  $\text{run } r \ p \longleftrightarrow (\forall k. \text{enabled } (r \ !! \ k) \ (\text{target } (\text{stake } k \ r) \ p))$ 
using snth-run run-snth by blast

lemma reachable-states:
assumes  $q \in \text{reachable } p \ \text{path } r \ q$ 
shows  $\text{set } (\text{states } r \ q) \subseteq \text{reachable } p$ 
using assms by (induct r arbitrary: q) (auto)
lemma reachable-trace:
assumes  $q \in \text{reachable } p \ \text{run } r \ q$ 
shows  $\text{sset } (\text{trace } r \ q) \subseteq \text{reachable } p$ 
using assms unfolding sset-subset-stream-pred
by (coinduction arbitrary: r q) (force elim: run.cases)

end

context transition-system-initial
begin

lemma nodes-states:
assumes  $p \in \text{nodes} \ \text{path } r \ p$ 
shows  $\text{set } (\text{states } r \ p) \subseteq \text{nodes}$ 
using reachable-states assms by blast
lemma nodes-trace:
assumes  $p \in \text{nodes} \ \text{run } r \ p$ 
shows  $\text{sset } (\text{trace } r \ p) \subseteq \text{nodes}$ 
using reachable-trace assms by blast

```

end

end

9 Constructing Paths and Runs in Transition Systems

theory *Transition-System-Construction*

imports

../Basic/Sequence-LTL

Transition-System

begin

context *transition-system*

begin

lemma *invariant-run:*

assumes $P\ p \wedge p. P\ p \implies \exists a. \text{enabled } a\ p \wedge P\ (\text{execute } a\ p) \wedge Q\ p\ a$

obtains r

where $\text{run } r\ p\ \text{pred-stream } P\ (p\ \#\#\ \text{trace } r\ p)\ \text{stream-all2 } Q\ (p\ \#\#\ \text{trace } r\ p)\ r$

proof –

obtain f **where** $1: \text{enabled } (f\ p)\ p\ P\ (\text{execute } (f\ p)\ p)\ Q\ p\ (f\ p)$ **if** $P\ p$ **for** p

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

let $?g = \lambda p. \text{execute } (f\ p)\ p$

let $?r = \lambda p. \text{smap } f\ (\text{siterate } ?g\ p)$

show *?thesis*

proof

show $\text{run } (?r\ p)\ p$ **using** *assms(1)* 1 **by** (*coinduction arbitrary: p*) (*auto*)

show $\text{pred-stream } P\ (p\ \#\#\ \text{trace } (?r\ p)\ p)$ **using** *assms(1)* 1 **by** (*coinduction arbitrary: p*) (*auto*)

show $\text{stream-all2 } Q\ (p\ \#\#\ \text{trace } (?r\ p)\ p)\ (?r\ p)$ **using** *assms(1)* 1 **by** (*coinduction arbitrary: p*) (*auto*)

qed

qed

lemma *recurring-condition:*

assumes $P\ p \wedge p. P\ p \implies \exists r. r \neq [] \wedge \text{path } r\ p \wedge P\ (\text{target } r\ p)$

obtains r

where $\text{run } r\ p\ \text{infs } P\ (p\ \#\#\ \text{trace } r\ p)$

proof –

obtain f **where** $1: f\ p \neq []\ \text{path } (f\ p)\ p\ P\ (\text{target } (f\ p)\ p)$ **if** $P\ p$ **for** p **using** *assms(2)* **by** *metis*

let $?g = \lambda p. \text{target } (f\ p)\ p$

let $?r = \lambda p. \text{flat } (\text{smap } f\ (\text{siterate } ?g\ p))$

have $2: ?r\ p = f\ p\ @- ?r\ (?g\ p)$ **if** $P\ p$ **for** p **using** *that 1(1)* **by** (*simp add: flat-unfold*)

show *?thesis*

proof
show $run\ (?r\ p)\ p$ **using** $assms(1)\ 1\ 2$ **by** (*coinduction arbitrary: p rule: run-coinduct-shift*) (*blast*)
show $infs\ P\ (p\ \#\#\ trace\ (?r\ p)\ p)$ **using** $assms(1)\ 1\ 2$ **by** (*coinduction arbitrary: p rule: infs-sscan-coinduct*) (*blast*)
qed
qed

lemma *invariant-run-index*:
assumes $P\ n\ p \wedge n\ p. P\ n\ p \implies \exists a. enabled\ a\ p \wedge P\ (Suc\ n)\ (execute\ a\ p)$
 $\wedge Q\ n\ p\ a$

obtains r

where

$run\ r\ p$

$\wedge i. P\ (n + i)\ (target\ (stake\ i\ r)\ p)$

$\wedge i. Q\ (n + i)\ (target\ (stake\ i\ r)\ p)\ (r\ !!\ i)$

proof –

define s **where** $s \equiv (n, p)$

have 1: *case-prod* $P\ s$ **using** $assms(1)$ **unfolding** s -def **by** *auto*

obtain f **where** 2:

$\wedge n\ p. P\ n\ p \implies enabled\ (f\ n\ p)\ p$

$\wedge n\ p. P\ n\ p \implies P\ (Suc\ n)\ (execute\ (f\ n\ p)\ p)$

$\wedge n\ p. P\ n\ p \implies Q\ n\ p\ (f\ n\ p)$

using $assms(2)$ **by** *metis*

define g **where** $g \equiv \lambda (n, p). (Suc\ n, execute\ (f\ n\ p)\ p)$

let $?r = smap\ (case-prod\ f)\ (siterate\ g\ s)$

have 3: $run\ ?r\ (snd\ s)$ **using** 1 2(1, 2) **unfolding** g -def **by** (*coinduction arbitrary: s*) (*auto*)

have 4: *case-prod* $P\ (compow\ k\ g\ s)$ **for** k **using** 1 2(2) **unfolding** g -def **by** (*induct* k) (*auto*)

have 5: *case-prod* $Q\ (compow\ k\ g\ s)\ (?r\ !!\ k)$ **for** k **using** 2(3) 4 **by** (*simp add: case-prod-beta*)

have 6: $compow\ k\ g\ (n, p) = (n + k, target\ (stake\ k\ ?r)\ p)$ **for** k

unfolding s -def g -def **by** (*induct* k) (*auto simp add: stake-Suc simp del: stake.simps(2)*)

show $?thesis$ **using** that 3 4 5 **unfolding** s -def 6 **by** *simp*

qed

lemma *koenig*:

assumes *infinite* (*reachable* p)

assumes $\wedge q. q \in reachable\ p \implies finite\ (successors\ q)$

obtains r

where $run\ r\ p$

proof (*rule invariant-run*[**where** $?P = \lambda q. q \in reachable\ p \wedge infinite\ (reachable\ q)$])

```

    show  $p \in \text{reachable } p \wedge \text{infinite } (\text{reachable } p)$  using assms(1) by auto
next
fix q
assume 1:  $q \in \text{reachable } p \wedge \text{infinite } (\text{reachable } q)$ 
have 2: finite (successors q) using assms(2) 1 by auto
have 3: infinite (insert q ( $\bigcup$  (reachable ' (successors q)))) using reachable-step
1 by metis
obtain s where 4:  $s \in \text{successors } q \wedge \text{infinite } (\text{reachable } s)$  using 2 3 by auto
show  $\exists a. \text{enabled } a \wedge q \wedge (\text{execute } a \wedge q \in \text{reachable } p \wedge \text{infinite } (\text{reachable } (\text{execute } a \wedge q))) \wedge \text{True}$ 
using 1 4 by auto
qed

end

end

```

10 Deterministic Automata

theory *Deterministic*

imports

```

  ../Transition-Systems/Transition-System
  ../Transition-Systems/Transition-System-Extra
  ../Transition-Systems/Transition-System-Construction
  ../Basic/Degeneralization

```

begin

locale *automaton* =

fixes *automaton* :: 'label set \Rightarrow 'state \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state) \Rightarrow 'condition \Rightarrow 'automaton

fixes *alphabet initial transition condition*

assumes *automaton[simp]*: *automaton* (*alphabet* *A*) (*initial* *A*) (*transition* *A*) (*condition* *A*) = *A*

assumes *alphabet[simp]*: *alphabet* (*automaton* *a i t c*) = *a*

assumes *initial[simp]*: *initial* (*automaton* *a i t c*) = *i*

assumes *transition[simp]*: *transition* (*automaton* *a i t c*) = *t*

assumes *condition[simp]*: *condition* (*automaton* *a i t c*) = *c*

begin

sublocale *transition-system-initial*

transition *A* λ *a p. a* \in *alphabet* *A* λ *p. p* = *initial* *A*

for *A*

defines *path'* = *path* **and** *run'* = *run* **and** *reachable'* = *reachable* **and** *nodes'* = *nodes*

by *this*

lemma *path-alt-def*: *path* *A* *w p* \longleftrightarrow *w* \in *lists* (*alphabet* *A*)

proof

```

    show  $w \in \text{lists } (\text{alphabet } A)$  if  $\text{path } A \ w \ p$  using that by (induct arbitrary:
  p) (auto)
    show  $\text{path } A \ w \ p$  if  $w \in \text{lists } (\text{alphabet } A)$  using that by (induct arbitrary:
  p) (auto)
  qed
  lemma run-alt-def:  $\text{run } A \ w \ p \longleftrightarrow w \in \text{streams } (\text{alphabet } A)$ 
  proof
    show  $w \in \text{streams } (\text{alphabet } A)$  if  $\text{run } A \ w \ p$ 
      using that by (coinduction arbitrary:  $w \ p$ ) (force elim: run.cases)
    show  $\text{run } A \ w \ p$  if  $w \in \text{streams } (\text{alphabet } A)$ 
      using that by (coinduction arbitrary:  $w \ p$ ) (force elim: streams.cases)
  qed

end

locale automaton-path =
  automaton automaton alphabet initial transition condition
  for automaton :: 'label set  $\Rightarrow$  'state  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state)  $\Rightarrow$  'condition
 $\Rightarrow$  'automaton
  and alphabet initial transition condition
  +
  fixes test :: 'condition  $\Rightarrow$  'label list  $\Rightarrow$  'state list  $\Rightarrow$  'state  $\Rightarrow$  bool
begin

  definition language :: 'automaton  $\Rightarrow$  'label list set where
    language  $A \equiv \{w. \text{path } A \ w \ (\text{initial } A) \wedge \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A)) \}$ 
    ( $\text{initial } A$ ) ( $\text{initial } A$ )

  lemma language[intro]:
    assumes  $\text{path } A \ w \ (\text{initial } A) \ \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A))$ 
    ( $\text{initial } A$ )
    shows  $w \in \text{language } A$ 
    using assms unfolding language-def by auto
  lemma language-elim[elim]:
    assumes  $w \in \text{language } A$ 
    obtains  $\text{path } A \ w \ (\text{initial } A) \ \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A))$ 
    ( $\text{initial } A$ )
    using assms unfolding language-def by auto

  lemma language-alphabet:  $\text{language } A \subseteq \text{lists } (\text{alphabet } A)$  using path-alt-def
  by auto

end

locale automaton-run =
  automaton automaton alphabet initial transition condition
  for automaton :: 'label set  $\Rightarrow$  'state  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state)  $\Rightarrow$  'condition
 $\Rightarrow$  'automaton
  and alphabet initial transition condition

```



```

+
fixes test :: 'condition  $\Rightarrow$  'label stream  $\Rightarrow$  'state stream  $\Rightarrow$  'state  $\Rightarrow$  bool
begin

  definition language :: 'automaton  $\Rightarrow$  'label stream set where
    language A  $\equiv$  {w. run A w (initial A)  $\wedge$  test (condition A) w (trace A w
(initial A)) (initial A)}

  lemma language[intro]:
    assumes run A w (initial A) test (condition A) w (trace A w (initial A))
(initial A)
    shows w  $\in$  language A
    using assms unfolding language-def by auto

  lemma language-elim[elim]:
    assumes w  $\in$  language A
    obtains run A w (initial A) test (condition A) w (trace A w (initial A))
(initial A)
    using assms unfolding language-def by auto

  lemma language-alphabet: language A  $\subseteq$  streams (alphabet A) using run-alt-def
by auto

end

locale automaton-degeneralization =
  a: automaton automaton1 alphabet1 initial1 transition1 condition1 +
  b: automaton automaton2 alphabet2 initial2 transition2 condition2
  for automaton1 :: 'label set  $\Rightarrow$  'state  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state)  $\Rightarrow$  'item pred
gen  $\Rightarrow$  'automaton1
  and alphabet1 initial1 transition1 condition1
  and automaton2 :: 'label set  $\Rightarrow$  'state degen  $\Rightarrow$  ('label  $\Rightarrow$  'state degen  $\Rightarrow$  'state
degen)  $\Rightarrow$  'item-degen pred  $\Rightarrow$  'automaton2
  and alphabet2 initial2 transition2 condition2
  +
  fixes item :: 'state  $\times$  'label  $\times$  'state  $\Rightarrow$  'item
  fixes translate :: 'item-degen  $\Rightarrow$  'item degen
begin

  definition degeneralize :: 'automaton1  $\Rightarrow$  'automaton2 where
    degeneralize A  $\equiv$  automaton2
      (alphabet1 A)
      (initial1 A, 0)
      ( $\lambda$  a (p, k). (transition1 A a p, count (condition1 A) (item (p, a, transition1
A a p)) k))
      (degen (condition1 A)  $\circ$  translate)

  lemma degeneralize-simps[simp]:
    alphabet2 (degeneralize A) = alphabet1 A
    initial2 (degeneralize A) = (initial1 A, 0)

```

$transition_2 (degeneralize\ A)\ a\ (p, k) =$
 $(transition_1\ A\ a\ p, count\ (condition_1\ A)\ (item\ (p, a, transition_1\ A\ a\ p))\ k)$
 $condition_2\ (degeneralize\ A) = degen\ (condition_1\ A) \circ translate$
unfolding *degeneralize-def* **by** *auto*

lemma *degeneralize-target[simp]*: $b.target\ (degeneralize\ A)\ w\ (p, k) =$
 $(a.target\ A\ w\ p, fold\ (count\ (condition_1\ A) \circ item)\ (p\ \# \ a.states\ A\ w\ p\ ||\ w$
 $||\ a.states\ A\ w\ p)\ k)$
by (*induct w arbitrary: p k*) (*auto*)

lemma *degeneralize-states[simp]*: $b.states\ (degeneralize\ A)\ w\ (p, k) =$
 $a.states\ A\ w\ p\ ||\ scan\ (count\ (condition_1\ A) \circ item)\ (p\ \# \ a.states\ A\ w\ p\ ||\ w$
 $||\ a.states\ A\ w\ p)\ k$
by (*induct w arbitrary: p k*) (*auto*)

lemma *degeneralize-trace[simp]*: $b.trace\ (degeneralize\ A)\ w\ (p, k) =$
 $a.trace\ A\ w\ p\ ||| sscan\ (count\ (condition_1\ A) \circ item)\ (p\ \#\#\ a.trace\ A\ w\ p\ |||$
 $w\ ||| \ a.trace\ A\ w\ p)\ k$
by (*coinduction arbitrary: w p k*) (*auto, metis sscan.code*)

lemma *degeneralize-path[iff]*: $b.path\ (degeneralize\ A)\ w\ (p, k) \longleftrightarrow a.path\ A\ w$
 p
unfolding *a.path-alt-def b.path-alt-def* **by** *simp*

lemma *degeneralize-run[iff]*: $b.run\ (degeneralize\ A)\ w\ (p, k) \longleftrightarrow a.run\ A\ w\ p$
unfolding *a.run-alt-def b.run-alt-def* **by** *simp*

lemma *degeneralize-reachable-fst[simp]*: $fst\ ' \ b.reachable\ (degeneralize\ A)\ (p, k)$
 $= a.reachable\ A\ p$
unfolding *a.reachable-alt-def b.reachable-alt-def image-def* **by** *simp*

lemma *degeneralize-reachable-snd-empty[simp]*:
assumes $condition_1\ A = []$
shows $snd\ ' \ b.reachable\ (degeneralize\ A)\ (p, k) = \{k\}$
proof –
have $snd\ ql = k$ **if** $ql \in b.reachable\ (degeneralize\ A)\ (p, k)$ **for** ql
using *that assms* **by** *induct auto*
then show *?thesis* **by** *auto*
qed

lemma *degeneralize-reachable-empty[simp]*:
assumes $condition_1\ A = []$
shows $b.reachable\ (degeneralize\ A)\ (p, k) = a.reachable\ A\ p \times \{k\}$
using *degeneralize-reachable-fst degeneralize-reachable-snd-empty assms*
by (*metis prod-singleton(2)*)

lemma *degeneralize-reachable-snd*:
 $snd\ ' \ b.reachable\ (degeneralize\ A)\ (p, k) \subseteq insert\ k\ \{0\ ..< \ length\ (condition_1$
 $A)\}$
by (*cases condition_1 A = []*) (*auto*)

lemma *degeneralize-reachable*:
 $b.reachable\ (degeneralize\ A)\ (p, k) \subseteq a.reachable\ A\ p \times insert\ k\ \{0\ ..< \ length$
 $(condition_1\ A)\}$
by (*cases condition_1 A = []*) (*auto 0 3*)

```

lemma degeneralize-nodes-fst[simp]: fst ‘ b.nodes (degeneralize A) = a.nodes A
  unfolding a.nodes-alt-def b.nodes-alt-def by simp
lemma degeneralize-nodes-snd-empty:
  assumes condition1 A = []
  shows snd ‘ b.nodes (degeneralize A) = {0}
  using assms unfolding b.nodes-alt-def by auto
lemma degeneralize-nodes-empty:
  assumes condition1 A = []
  shows b.nodes (degeneralize A) = a.nodes A × {0}
  using assms unfolding b.nodes-alt-def by auto
lemma degeneralize-nodes-snd:
  snd ‘ b.nodes (degeneralize A) ⊆ insert 0 {0 ..< length (condition1 A)}
  using degeneralize-reachable-snd unfolding b.nodes-alt-def by auto
lemma degeneralize-nodes:
  b.nodes (degeneralize A) ⊆ a.nodes A × insert 0 {0 ..< length (condition1
A)}
  using degeneralize-reachable unfolding a.nodes-alt-def b.nodes-alt-def by
simp

lemma degeneralize-nodes-finite[iff]: finite (b.nodes (degeneralize A)) ↔ finite
(a.nodes A)
proof
  show finite (a.nodes A) if finite (b.nodes (degeneralize A))
    using that by (auto simp flip: degeneralize-nodes-fst)
  show finite (b.nodes (degeneralize A)) if finite (a.nodes A)
    using finite-subset degeneralize-nodes that by blast
qed
lemma degeneralize-nodes-card: card (b.nodes (degeneralize A)) ≤
  max 1 (length (condition1 A)) * card (a.nodes A)
proof (cases finite (a.nodes A))
  case True
    have card (b.nodes (degeneralize A)) ≤ card (a.nodes A × insert 0 {0 ..<
length (condition1 A)}))
    using degeneralize-nodes True by (blast intro: card-mono)
    also have ... = card (insert 0 {0 ..< length (condition1 A)})) * card (a.nodes
A)
    unfolding card-cartesian-product by simp
    also have card (insert 0 {0 ..< length (condition1 A)})) = max 1 (length
(condition1 A))
    by (simp add: card-insert-if Suc-leI max-absorb2)
    finally show ?thesis by this
  next
  case False
    then have card (a.nodes A) = 0 card (b.nodes (degeneralize A)) = 0 by auto
    then show ?thesis by simp
qed
end

```

locale *automaton-degeneralization-run* =
automaton-degeneralization
*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁
*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂
item translate +
*a: automaton-run automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +
*b: automaton-run automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂
for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁
and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂
and *item translate*
+
assumes *test*[*iff*]: *test*₂ (*degen cs* \circ *translate*) *w*
(*r* ||| *sscan* (*count cs* \circ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*) (*p, k*) \longleftrightarrow *test*₁ *cs w r p*
begin

lemma *degeneralize-language*[*simp*]: *b.language* (*degeneralize A*) = *a.language*
A by force

end

locale *automaton-product* =
*a: automaton automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ +
*b: automaton automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ +
*c: automaton automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃
for *automaton*₁ :: '*label set* \Rightarrow '*state*₁ \Rightarrow ('*label* \Rightarrow '*state*₁ \Rightarrow '*state*₁) \Rightarrow
'*condition*₁ \Rightarrow '*automaton*₁
and *alphabet*₁ *initial*₁ *transition*₁ *condition*₁
and *automaton*₂ :: '*label set* \Rightarrow '*state*₂ \Rightarrow ('*label* \Rightarrow '*state*₂ \Rightarrow '*state*₂) \Rightarrow
'*condition*₂ \Rightarrow '*automaton*₂
and *alphabet*₂ *initial*₂ *transition*₂ *condition*₂
and *automaton*₃ :: '*label set* \Rightarrow '*state*₁ \times '*state*₂ \Rightarrow ('*label* \Rightarrow '*state*₁ \times '*state*₂
 \Rightarrow '*state*₁ \times '*state*₂) \Rightarrow '*condition*₃ \Rightarrow '*automaton*₃
and *alphabet*₃ *initial*₃ *transition*₃ *condition*₃
+
fixes *condition* :: '*condition*₁ \Rightarrow '*condition*₂ \Rightarrow '*condition*₃
begin

definition *product* :: '*automaton*₁ \Rightarrow '*automaton*₂ \Rightarrow '*automaton*₃ **where**
product A B \equiv *automaton*₃
(*alphabet*₁ *A* \cap *alphabet*₂ *B*)
(*initial*₁ *A*, *initial*₂ *B*)
(λ *a* (*p, q*). (*transition*₁ *A a p, transition*₂ *B a q*)
(*condition* (*condition*₁ *A*) (*condition*₂ *B*))

lemma *product-simps*[*simp*]:
*alphabet*₃ (*product A B*) = *alphabet*₁ *A* \cap *alphabet*₂ *B*
*initial*₃ (*product A B*) = (*initial*₁ *A*, *initial*₂ *B*)
*transition*₃ (*product A B*) *a* (*p, q*) = (*transition*₁ *A a p, transition*₂ *B a q*)
*condition*₃ (*product A B*) = *condition* (*condition*₁ *A*) (*condition*₂ *B*)

unfolding *product-def* **by** *auto*

lemma *product-target[simp]*: $c.target (product\ A\ B)\ w\ (p, q) = (a.target\ A\ w\ p, b.target\ B\ w\ q)$

by (*induct w arbitrary: p q*) (*auto*)

lemma *product-states[simp]*: $c.states (product\ A\ B)\ w\ (p, q) = a.states\ A\ w\ p \parallel b.states\ B\ w\ q$

by (*induct w arbitrary: p q*) (*auto*)

lemma *product-trace[simp]*: $c.trace (product\ A\ B)\ w\ (p, q) = a.trace\ A\ w\ p \parallel\parallel b.trace\ B\ w\ q$

by (*coinduction arbitrary: w p q*) (*auto*)

lemma *product-path[iff]*: $c.path (product\ A\ B)\ w\ (p, q) \longleftrightarrow a.path\ A\ w\ p \wedge b.path\ B\ w\ q$

unfolding *a.path-alt-def b.path-alt-def c.path-alt-def* **by** *simp*

lemma *product-run[iff]*: $c.run (product\ A\ B)\ w\ (p, q) \longleftrightarrow a.run\ A\ w\ p \wedge b.run\ B\ w\ q$

unfolding *a.run-alt-def b.run-alt-def c.run-alt-def* **by** *simp*

lemma *product-reachable[simp]*: $c.reachable (product\ A\ B)\ (p, q) \subseteq a.reachable\ A\ p \times b.reachable\ B\ q$

unfolding *c.reachable-alt-def* **by** *auto*

lemma *product-nodes[simp]*: $c.nodes (product\ A\ B) \subseteq a.nodes\ A \times b.nodes\ B$

unfolding *a.nodes-alt-def b.nodes-alt-def c.nodes-alt-def* **by** *auto*

lemma *product-reachable-fst[simp]*:

assumes $alphabet_1\ A \subseteq alphabet_2\ B$

shows $fst\ 'c.reachable (product\ A\ B)\ (p, q) = a.reachable\ A\ p$

using *assms*

unfolding *a.reachable-alt-def a.path-alt-def*

unfolding *b.reachable-alt-def b.path-alt-def*

unfolding *c.reachable-alt-def c.path-alt-def*

by *auto force*

lemma *product-reachable-snd[simp]*:

assumes $alphabet_1\ A \supseteq alphabet_2\ B$

shows $snd\ 'c.reachable (product\ A\ B)\ (p, q) = b.reachable\ B\ q$

using *assms*

unfolding *a.reachable-alt-def a.path-alt-def*

unfolding *b.reachable-alt-def b.path-alt-def*

unfolding *c.reachable-alt-def c.path-alt-def*

by *auto force*

lemma *product-nodes-fst[simp]*:

assumes $alphabet_1\ A \subseteq alphabet_2\ B$

shows $fst\ 'c.nodes (product\ A\ B) = a.nodes\ A$

using *assms product-reachable-fst*

unfolding *a.nodes-alt-def b.nodes-alt-def c.nodes-alt-def*

by *fastforce*

lemma *product-nodes-snd[simp]*:

assumes $alphabet_1\ A \supseteq alphabet_2\ B$

shows $snd\ 'c.nodes (product\ A\ B) = b.nodes\ B$

```

using assms product-reachable-snd
unfolding a.nodes-alt-def b.nodes-alt-def c.nodes-alt-def
by fastforce

lemma product-nodes-finite[intro]:
  assumes finite (a.nodes A) finite (b.nodes B)
  shows finite (c.nodes (product A B))
proof (rule finite-subset)
  show c.nodes (product A B) ⊆ a.nodes A × b.nodes B using product-nodes
by this
  show finite (a.nodes A × b.nodes B) using assms by simp
qed
lemma product-nodes-finite-strong[iff]:
  assumes alphabet1 A = alphabet2 B
  shows finite (c.nodes (product A B)) ↔ finite (a.nodes A) ∧ finite (b.nodes
B)
proof safe
  show finite (a.nodes A) if finite (c.nodes (product A B))
    using product-nodes-fst assms that by (metis finite-imageI equalityD1)
  show finite (b.nodes B) if finite (c.nodes (product A B))
    using product-nodes-snd assms that by (metis finite-imageI equalityD2)
  show finite (c.nodes (product A B)) if finite (a.nodes A) finite (b.nodes B)
    using that by rule
qed
lemma product-nodes-card[intro]:
  assumes finite (a.nodes A) finite (b.nodes B)
  shows card (c.nodes (product A B)) ≤ card (a.nodes A) * card (b.nodes B)
proof –
  have card (c.nodes (product A B)) ≤ card (a.nodes A × b.nodes B)
  proof (rule card-mono)
    show finite (a.nodes A × b.nodes B) using assms by simp
    show c.nodes (product A B) ⊆ a.nodes A × b.nodes B using product-nodes
by this
  qed
  also have ... = card (a.nodes A) * card (b.nodes B) using card-cartesian-product
by this
  finally show ?thesis by this
qed
lemma product-nodes-card-strong[intro]:
  assumes alphabet1 A = alphabet2 B
  shows card (c.nodes (product A B)) ≤ card (a.nodes A) * card (b.nodes B)
proof (cases finite (a.nodes A) ∧ finite (b.nodes B))
  case True
    show ?thesis using True by auto
next
  case False
    have 1: card (c.nodes (product A B)) = 0 using False assms by simp
    have 2: card (a.nodes A) * card (b.nodes B) = 0 using False by auto
    show ?thesis using 1 2 by simp

```

qed

end

locale *automaton-intersection-path* =

automaton-product

*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁

*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂

*automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃

condition +

a: *automaton-path* *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +

b: *automaton-path* *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂ +

c: *automaton-path* *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁

and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

and *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

and *condition*

+

assumes *test*[*iff*]: *length* *r* = *length* *s* \implies

*test*₃ (*condition* *c*₁ *c*₂) *w* (*r* || *s*) (*p*, *q*) \longleftrightarrow *test*₁ *c*₁ *w* *r* *p* \wedge *test*₂ *c*₂ *w* *s* *q*

begin

lemma *product-language*[*simp*]: *c.language* (*product* *A* *B*) = *a.language* *A* \cap *b.language* *B* **by** *force*

end

locale *automaton-union-path* =

automaton-product

*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁

*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂

*automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃

condition +

a: *automaton-path* *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +

b: *automaton-path* *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂ +

c: *automaton-path* *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁

and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

and *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

and *condition*

+

assumes *test*[*iff*]: *length* *r* = *length* *s* \implies

*test*₃ (*condition* *c*₁ *c*₂) *w* (*r* || *s*) (*p*, *q*) \longleftrightarrow *test*₁ *c*₁ *w* *r* *p* \vee *test*₂ *c*₂ *w* *s* *q*

begin

lemma *product-language*[*simp*]:

assumes *alphabet*₁ *A* = *alphabet*₂ *B*

shows *c.language* (*product* *A* *B*) = *a.language* *A* \cup *b.language* *B*

using *assms* **by** (*force simp: a.path-alt-def b.path-alt-def*)

end

locale *automaton-intersection-run* =

automaton-product

*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁

*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂

*automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃

condition +

a: *automaton-run* *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +

b: *automaton-run* *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂ +

c: *automaton-run* *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁

and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

and *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

and *condition*

+

assumes *test*[*iff*]: *test*₃ (*condition* *c*₁ *c*₂) *w* (*r* ||| *s*) (*p*, *q*) \longleftrightarrow *test*₁ *c*₁ *w* *r* *p*
 \wedge *test*₂ *c*₂ *w* *s* *q*

begin

lemma *product-language*[*simp*]: *c.language* (*product* *A* *B*) = *a.language* *A* \cap
b.language *B* **by** *force*

end

locale *automaton-union-run* =

automaton-product

*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁

*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂

*automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃

condition +

a: *automaton-run* *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +

b: *automaton-run* *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂ +

c: *automaton-run* *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁

and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

and *automaton*₃ *alphabet*₃ *initial*₃ *transition*₃ *condition*₃ *test*₃

and *condition*

+

assumes *test*[*iff*]: *test*₃ (*condition* *c*₁ *c*₂) *w* (*r* ||| *s*) (*p*, *q*) \longleftrightarrow *test*₁ *c*₁ *w* *r* *p*
 \vee *test*₂ *c*₂ *w* *s* *q*

begin

lemma *product-language*[*simp*]:

assumes *alphabet*₁ *A* = *alphabet*₂ *B*

shows *c.language* (*product* *A* *B*) = *a.language* *A* \cup *b.language* *B*

using *assms* **by** (*force simp: a.run-alt-def b.run-alt-def*)

end

```
locale automaton-product-list =  
  a: automaton automaton1 alphabet1 initial1 transition1 condition1 +  
  b: automaton automaton2 alphabet2 initial2 transition2 condition2  
  for automaton1 :: 'label set ⇒ 'state ⇒ ('label ⇒ 'state ⇒ 'state) ⇒ 'condition1  
⇒ 'automaton1  
  and alphabet1 initial1 transition1 condition1  
  and automaton2 :: 'label set ⇒ 'state list ⇒ ('label ⇒ 'state list ⇒ 'state list)  
⇒ 'condition2 ⇒ 'automaton2  
  and alphabet2 initial2 transition2 condition2  
  +  
  fixes condition :: 'condition1 list ⇒ 'condition2  
begin
```

```
definition product :: 'automaton1 list ⇒ 'automaton2 where  
  product AA ≡ automaton2  
    (∩ (alphabet1 ' set AA))  
    (map initial1 AA)  
    (λ a ps. map2 (λ A p. transition1 A a p) AA ps)  
    (condition (map condition1 AA))
```

```
lemma product-simps[simp]:  
  alphabet2 (product AA) = ∩ (alphabet1 ' set AA)  
  initial2 (product AA) = map initial1 AA  
  transition2 (product AA) a ps = map2 (λ A p. transition1 A a p) AA ps  
  condition2 (product AA) = condition (map condition1 AA)  
unfolding product-def by auto
```

```
lemma product-trace-smap:  
  assumes length ps = length AA k < length AA  
  shows smap (λ ps. ps ! k) (b.trace (product AA) w ps) = a.trace (AA ! k) w  
(ps ! k)  
  using assms by (coinduction arbitrary: w ps) (force)
```

```
lemma product-nodes: b.nodes (product AA) ⊆ listset (map a.nodes AA)  
proof  
  show ps ∈ listset (map a.nodes AA) if ps ∈ b.nodes (product AA) for ps  
  using that by (induct) (auto simp: listset-member list-all2-conv-all-nth)  
qed
```

```
lemma product-nodes-finite[intro]:  
  assumes list-all (finite ∘ a.nodes) AA  
  shows finite (b.nodes (product AA))  
  using list.pred-map product-nodes assms by (blast dest: finite-subset)  
lemma product-nodes-card:  
  assumes list-all (finite ∘ a.nodes) AA
```

```

  shows card (b.nodes (product AA)) ≤ prod-list (map (card ∘ a.nodes) AA)
proof -
  have card (b.nodes (product AA)) ≤ card (listset (map a.nodes AA))
    using list.pred-map product-nodes assms by (blast intro: card-mono)
  also have ... = prod-list (map (card ∘ a.nodes) AA) by simp
  finally show ?thesis by this
qed

```

end

```

locale automaton-intersection-list-run =
  automaton-product-list
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  condition +
  a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2
for automaton1 alphabet1 initial1 transition1 condition1 test1
and automaton2 alphabet2 initial2 transition2 condition2 test2
and condition
+
assumes test[iff]: test2 (condition cs) w rs ps ↔
  (∀ k < length cs. test1 (cs ! k) w (smap (λ ps. ps ! k) rs) (ps ! k))
begin

```

```

  lemma product-language[simp]: b.language (product AA) = ∩ (a.language ‘ set
AA)
  unfolding a.language-def b.language-def
  unfolding a.run-alt-def b.run-alt-def streams-iff-sset
  by (fastforce simp: set-conv-nth product-trace-smap)

```

end

```

locale automaton-union-list-run =
  automaton-product-list
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  condition +
  a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2
for automaton1 alphabet1 initial1 transition1 condition1 test1
and automaton2 alphabet2 initial2 transition2 condition2 test2
and condition
+
assumes test[iff]: test2 (condition cs) w rs ps ↔
  (∃ k < length cs. test1 (cs ! k) w (smap (λ ps. ps ! k) rs) (ps ! k))
begin

```

```

  lemma product-language[simp]:

```

```

assumes  $\bigcap$  (alphabet1 'set AA) =  $\bigcup$  (alphabet1 'set AA)
shows b.language (product AA) =  $\bigcup$  (a.language 'set AA)
using assms
unfolding a.language-def b.language-def
unfolding a.run-alt-def b.run-alt-def streams-iff-sset
by (fastforce simp: set-conv-nth product-trace-smap)

```

end

```

locale automaton-complement =
  a: automaton automaton1 alphabet1 initial1 transition1 condition1 +
  b: automaton automaton2 alphabet2 initial2 transition2 condition2
  for automaton1 :: 'label set  $\Rightarrow$  'state  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state)  $\Rightarrow$  'condition1
 $\Rightarrow$  'automaton1
  and alphabet1 initial1 transition1 condition1
  and automaton2 :: 'label set  $\Rightarrow$  'state  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state)  $\Rightarrow$  'condition2
 $\Rightarrow$  'automaton2
  and alphabet2 initial2 transition2 condition2
  +
  fixes condition :: 'condition1  $\Rightarrow$  'condition2
begin

```

```

  definition complement :: 'automaton1  $\Rightarrow$  'automaton2 where
    complement A  $\equiv$  automaton2 (alphabet1 A) (initial1 A) (transition1 A)
    (condition (condition1 A))

```

```

lemma combine-simps[simp]:
  alphabet2 (complement A) = alphabet1 A
  initial2 (complement A) = initial1 A
  transition2 (complement A) = transition1 A
  condition2 (complement A) = condition (condition1 A)
  unfolding complement-def by auto

```

end

```

locale automaton-complement-path =
  automaton-complement
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  condition +
  a: automaton-path automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-path automaton2 alphabet2 initial2 transition2 condition2 test2
  for automaton1 alphabet1 initial1 transition1 condition1 test1
  and automaton2 alphabet2 initial2 transition2 condition2 test2
  and condition
  +
  assumes test[iff]: test2 (condition c) w r p  $\longleftrightarrow$   $\neg$  test1 c w r p
begin

```

```

lemma complement-language[simp]: b.language (complement A) = lists (alphabet1
A) - a.language A
  unfolding a.language-def b.language-def a.path-alt-def b.path-alt-def by auto

end

locale automaton-complement-run =
  automaton-complement
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  condition +
  a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2
  for automaton1 alphabet1 initial1 transition1 condition1 test1
  and automaton2 alphabet2 initial2 transition2 condition2 test2
  and condition
  +
  assumes test[iff]: test2 (condition c) w r p  $\longleftrightarrow$   $\neg$  test1 c w r p
begin

  lemma complement-language[simp]: b.language (complement A) = streams
(alphabet1 A) - a.language A
  unfolding a.language-def b.language-def a.run-alt-def b.run-alt-def by auto

end

end

```

11 Deterministic Finite Automata

```

theory DFA
imports ../Deterministic
begin

  datatype ('label, 'state) dfa = dfa
    (alphabet: 'label set)
    (initial: 'state)
    (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
    (accepting: 'state pred)

  global-interpretation dfa: automaton dfa alphabet initial transition accepting
  defines path = dfa.path and run = dfa.run and reachable = dfa.reachable and
nodes = dfa.nodes
  by unfold-locales auto

  global-interpretation dfa: automaton-path dfa alphabet initial transition accept-
ing  $\lambda P w r p. P$  (last (p # r))
  defines language = dfa.language
  by standard

```

abbreviation *target* **where** *target* \equiv *dfa.target*
abbreviation *states* **where** *states* \equiv *dfa.states*
abbreviation *trace* **where** *trace* \equiv *dfa.trace*
abbreviation *successors* **where** *successors* \equiv *dfa.successors* *TYPE('label)*

global-interpretation *intersection: automaton-intersection-path*
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
 $\lambda c_1 c_2 (p, q). c_1 p \wedge c_2 q$
defines *intersect* = *intersection.product*
by (*unfold-locales*) (*auto simp: zip-eq-Nil-iff*)

global-interpretation *union: automaton-union-path*
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
 $\lambda c_1 c_2 (p, q). c_1 p \vee c_2 q$
defines *union* = *union.product*
by (*unfold-locales*) (*auto simp: zip-eq-Nil-iff*)

global-interpretation *complement: automaton-complement-path*
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
dfa *alphabet* *initial* *transition* *accepting* $\lambda P w r p. P$ (*last* (*p* # *r*))
 $\lambda c p. \neg c p$
defines *complement* = *complement.complement*
by *unfold-locales auto*

end

12 Nondeterministic Automata

theory *Nondeterministic*

imports

../Transition-Systems/Transition-System
../Transition-Systems/Transition-System-Extra
../Transition-Systems/Transition-System-Construction
../Basic/Degeneralization

begin

locale *automaton* =
fixes *automaton* :: *'label set* \Rightarrow *'state set* \Rightarrow (*'label* \Rightarrow *'state* \Rightarrow *'state set*) \Rightarrow
'condition \Rightarrow *'automaton*
fixes *alphabet* *initial* *transition* *condition*
assumes *automaton[simp]*: *automaton* (*alphabet* *A*) (*initial* *A*) (*transition* *A*)
(*condition* *A*) = *A*
assumes *alphabet[simp]*: *alphabet* (*automaton* *a i t c*) = *a*
assumes *initial[simp]*: *initial* (*automaton* *a i t c*) = *i*
assumes *transition[simp]*: *transition* (*automaton* *a i t c*) = *t*

```

assumes condition[simp]: condition (automaton a i t c) = c
begin

  sublocale transition-system-initial
     $\lambda a p. \text{snd } a \lambda a p. \text{fst } a \in \text{alphabet } A \wedge \text{snd } a \in \text{transition } A (\text{fst } a) p \lambda p. p$ 
     $\in \text{initial } A$ 
    for A
    defines path' = path and run' = run and reachable' = reachable and nodes'
    = nodes
    by this

  lemma states-alt-def: states r p = map snd r by (induct r arbitrary: p) (auto)
  lemma trace-alt-def: trace r p = smap snd r by (coinduction arbitrary: r p)
  (auto)

  lemma successors-alt-def: successors A p = ( $\bigcup a \in \text{alphabet } A. \text{transition } A a$ 
  p) by auto

  lemma reachable-transition[intro]:
    assumes a  $\in \text{alphabet } A$  q  $\in \text{reachable } A$  p r  $\in \text{transition } A a$  q
    shows r  $\in \text{reachable } A$  p
    using reachable.execute assms by force
  lemma nodes-transition[intro]:
    assumes a  $\in \text{alphabet } A$  p  $\in \text{nodes } A$  q  $\in \text{transition } A a$  p
    shows q  $\in \text{nodes } A$ 
    using nodes.execute assms by force

  lemma path-alphabet:
    assumes length r = length w path A (w || r) p
    shows w  $\in \text{lists } (\text{alphabet } A)$ 
    using assms by (induct arbitrary: p rule: list-induct2) (auto)
  lemma run-alphabet:
    assumes run A (w ||| r) p
    shows w  $\in \text{streams } (\text{alphabet } A)$ 
    using assms by (coinduction arbitrary: w r p) (metis run.cases stream.map
  szip-smap szip-smap-fst)

  definition restrict :: 'automaton  $\Rightarrow$  'automaton where
    restrict A  $\equiv$  automaton
      (alphabet A)
      (initial A)
      ( $\lambda a p. \text{if } a \in \text{alphabet } A \text{ then transition } A a p \text{ else } \{\}$ )
      (condition A)

  lemma restrict-simps[simp]:
    alphabet (restrict A) = alphabet A
    initial (restrict A) = initial A
    transition (restrict A) a p = (if a  $\in \text{alphabet } A$  then transition A a p else  $\{\}$ )
    condition (restrict A) = condition A

```

unfolding *restrict-def* **by** *auto*

lemma *restrict-path[simp]*: $\text{path } (\text{restrict } A) = \text{path } A$
proof (*intro ext iffI*)
 show $\text{path } A \text{ wr } p$ **if** $\text{path } (\text{restrict } A) \text{ wr } p$ **for** $\text{wr } p$ **using** *that* **by** *induct auto*
 show $\text{path } (\text{restrict } A) \text{ wr } p$ **if** $\text{path } A \text{ wr } p$ **for** $\text{wr } p$ **using** *that* **by** *induct auto*
qed

lemma *restrict-run[simp]*: $\text{run } (\text{restrict } A) = \text{run } A$
proof (*intro ext iffI*)
 show $\text{run } A \text{ wr } p$ **if** $\text{run } (\text{restrict } A) \text{ wr } p$ **for** $\text{wr } p$ **using** *that* **by** *coinduct auto*
 show $\text{run } (\text{restrict } A) \text{ wr } p$ **if** $\text{run } A \text{ wr } p$ **for** $\text{wr } p$ **using** *that* **by** *coinduct auto*
qed

end

locale *automaton-path* =
 automaton automaton alphabet initial transition condition
 for *automaton* :: $'\text{label set} \Rightarrow '\text{state set} \Rightarrow (' \text{label} \Rightarrow '\text{state} \Rightarrow '\text{state set}) \Rightarrow$
 'condition $\Rightarrow '\text{automaton}$
 and *alphabet initial transition condition*
 +
 fixes *test* :: $'\text{condition} \Rightarrow '\text{label list} \Rightarrow '\text{state list} \Rightarrow '\text{state} \Rightarrow \text{bool}$
begin

definition *language* :: $'\text{automaton} \Rightarrow '\text{label list set}$ **where**
 language $A \equiv \{w \mid \text{wr } p. \text{length } r = \text{length } w \wedge p \in \text{initial } A \wedge \text{path } A (w \parallel r) p \wedge \text{test } (\text{condition } A) w r p\}$

lemma *language[intro]*:
 assumes $\text{length } r = \text{length } w \text{ } p \in \text{initial } A \text{ path } A (w \parallel r) p \text{ test } (\text{condition } A) w r p$
 shows $w \in \text{language } A$
 using *assms* **unfolding** *language-def* **by** *auto*

lemma *language-elim[elim]*:
 assumes $w \in \text{language } A$
 obtains $r p$
 where $\text{length } r = \text{length } w \text{ } p \in \text{initial } A \text{ path } A (w \parallel r) p \text{ test } (\text{condition } A) w r p$
 using *assms* **unfolding** *language-def* **by** *auto*

lemma *language-alphabet*: $\text{language } A \subseteq \text{lists } (\text{alphabet } A)$ **by** (*auto dest: path-alphabet*)

lemma *restrict-language[simp]*: $\text{language } (\text{restrict } A) = \text{language } A$ **by** *force*

end

locale *automaton-run* =
 automaton automaton alphabet initial transition condition
 for *automaton* :: 'label set \Rightarrow 'state set \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state set) \Rightarrow
 '*condition* \Rightarrow '*automaton*
 and *alphabet initial transition condition*
 +
 fixes *test* :: '*condition* \Rightarrow '*label stream* \Rightarrow '*state stream* \Rightarrow '*state* \Rightarrow *bool*
begin

definition *language* :: '*automaton* \Rightarrow '*label stream set* **where**
 language A \equiv {*w* | *w r p*. *p* \in *initial A* \wedge *run A* (*w* ||| *r*) *p* \wedge *test* (*condition*
 A) *w r p*}

lemma *language[intro]*:
 assumes *p* \in *initial A* *run A* (*w* ||| *r*) *p* *test* (*condition A*) *w r p*
 shows *w* \in *language A*
 using *assms unfolding language-def by auto*
 lemma *language-elim[elim]*:
 assumes *w* \in *language A*
 obtains *r p*
 where *p* \in *initial A* *run A* (*w* ||| *r*) *p* *test* (*condition A*) *w r p*
 using *assms unfolding language-def by auto*

lemma *language-alphabet*: *language A* \subseteq *streams* (*alphabet A*) **by** (*auto dest*:
run-alphabet)

lemma *restrict-language[simp]*: *language* (*restrict A*) = *language A* **by force**

end

locale *automaton-degeneralization* =
 a: automaton automaton₁ alphabet₁ initial₁ transition₁ condition₁ +
 b: automaton automaton₂ alphabet₂ initial₂ transition₂ condition₂
 for *automaton₁* :: 'label set \Rightarrow 'state set \Rightarrow ('label \Rightarrow 'state \Rightarrow 'state set) \Rightarrow
 '*item pred gen* \Rightarrow '*automaton₁*
 and *alphabet₁ initial₁ transition₁ condition₁*
 and *automaton₂* :: 'label set \Rightarrow 'state degen set \Rightarrow ('label \Rightarrow 'state degen \Rightarrow
 '*state degen set*) \Rightarrow '*item-degen pred* \Rightarrow '*automaton₂*
 and *alphabet₂ initial₂ transition₂ condition₂*
 +
 fixes *item* :: 'state \times 'label \times 'state \Rightarrow '*item*
 fixes *translate* :: '*item-degen* \Rightarrow '*item degen*
begin

definition *degeneralize* :: '*automaton₁* \Rightarrow '*automaton₂* **where**
 degeneralize A \equiv *automaton₂*
 (*alphabet₁ A*)

$(initial_1 A \times \{0\})$
 $(\lambda a (p, k). \{(q, count (condition_1 A) (item (p, a, q)) k) \mid q. q \in transition_1 A a p\})$
 $(degen (condition_1 A) \circ translate)$

lemma *degeneralize-simps*[simp]:

$alphabet_2 (degeneralize A) = alphabet_1 A$
 $initial_2 (degeneralize A) = initial_1 A \times \{0\}$
 $transition_2 (degeneralize A) a (p, k) =$
 $\{(q, count (condition_1 A) (item (p, a, q)) k) \mid q. q \in transition_1 A a p\}$
 $condition_2 (degeneralize A) = degen (condition_1 A) \circ translate$
unfolding *degeneralize-def by auto*

lemma *run-degeneralize*:

assumes $a.run A (w \parallel r) p$
shows $b.run (degeneralize A) (w \parallel r \parallel sscan (count (condition_1 A) \circ item) (p \#\# r \parallel w \parallel r) k) (p, k)$
using *assms by (coinduction arbitrary: w r p k) (force elim: a.run.cases)*

lemma *degeneralize-run*:

assumes $b.run (degeneralize A) (w \parallel rs) pk$
obtains $r s p k$
where $rs = r \parallel s$ $pk = (p, k)$ $a.run A (w \parallel r) p$ $s = sscan (count (condition_1 A) \circ item) (p \#\# r \parallel w \parallel r) k$

proof –

obtain $r s p k$ **where** $1: rs = r \parallel s$ $pk = (p, k)$ **using** *szip-smap surjective-pairing by metis*

show *?thesis*

proof

show $rs = r \parallel s$ $pk = (p, k)$ **using** 1 **by** *this*

show $a.run A (w \parallel r) p$

using *assms unfolding 1 by (coinduction arbitrary: w r s p k) (force elim: b.run.cases)*

show $s = sscan (count (condition_1 A) \circ item) (p \#\# r \parallel w \parallel r) k$

using *assms unfolding 1 by (coinduction arbitrary: w r s p k) (erule b.run.cases, force)*

qed

qed

lemma *degeneralize-nodes*:

$b.nodes (degeneralize A) \subseteq a.nodes A \times insert 0 \{0 ..< length (condition_1 A)\}$

proof

fix pk

assume $pk \in b.nodes (degeneralize A)$

then show $pk \in a.nodes A \times insert 0 \{0 ..< length (condition_1 A)\}$

by *(induct) (force, cases condition_1 A = [], auto)*

qed

lemma *nodes-degeneralize*: $a.nodes A \subseteq fst \text{ ` } b.nodes (degeneralize A)$

proof

```

fix p
assume p ∈ a.nodes A
then show p ∈ fst ‘ b.nodes (degeneralize A)
proof induct
  case (initial p)
  have (p, 0) ∈ b.nodes (degeneralize A) using initial by auto
  then show ?case using image-iff fst-conv by force
next
  case (execute p aq)
  obtain k where (p, k) ∈ b.nodes (degeneralize A) using execute(2) by auto
  then have (snd aq, count (condition1 A) (item (p, aq)) k) ∈ b.nodes
(degeneralize A)
  using execute(3) by auto
  then show ?case using image-iff snd-conv by force
qed
qed

lemma degeneralize-nodes-finite[iff]: finite (b.nodes (degeneralize A)) ↔ finite
(a.nodes A)
proof
  show finite (a.nodes A) if finite (b.nodes (degeneralize A))
  using finite-subset nodes-degeneralize that by blast
  show finite (b.nodes (degeneralize A)) if finite (a.nodes A)
  using finite-subset degeneralize-nodes that by blast
qed

end

locale automaton-degeneralization-run =
  automaton-degeneralization
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  item translate +
  a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2
  for automaton1 alphabet1 initial1 transition1 condition1 test1
  and automaton2 alphabet2 initial2 transition2 condition2 test2
  and item translate
  +
  assumes test[iff]: test2 (degen cs ◦ translate) w
  (r ||| sscan (count cs ◦ item) (p ## r ||| w ||| r) k) (p, k) ↔ test1 cs w r p
begin

  lemma degeneralize-language[simp]: b.language (degeneralize A) = a.language
A
  unfolding a.language-def b.language-def by (auto dest: run-degeneralize elim!:
degeneralize-run)

end

```

```

locale automaton-product =
  a: automaton automaton1 alphabet1 initial1 transition1 condition1 +
  b: automaton automaton2 alphabet2 initial2 transition2 condition2 +
  c: automaton automaton3 alphabet3 initial3 transition3 condition3
  for automaton1 :: 'label set ⇒ 'state1 set ⇒ ('label ⇒ 'state1 ⇒ 'state1 set) ⇒
  'condition1 ⇒ 'automaton1
  and alphabet1 initial1 transition1 condition1
  and automaton2 :: 'label set ⇒ 'state2 set ⇒ ('label ⇒ 'state2 ⇒ 'state2 set)
  ⇒ 'condition2 ⇒ 'automaton2
  and alphabet2 initial2 transition2 condition2
  and automaton3 :: 'label set ⇒ ('state1 × 'state2) set ⇒ ('label ⇒ 'state1 ×
  'state2 ⇒ ('state1 × 'state2) set) ⇒ 'condition3 ⇒ 'automaton3
  and alphabet3 initial3 transition3 condition3
  +
  fixes condition :: 'condition1 ⇒ 'condition2 ⇒ 'condition3
begin

```

```

definition product :: 'automaton1 ⇒ 'automaton2 ⇒ 'automaton3 where
  product A B ≡ automaton3
    (alphabet1 A ∩ alphabet2 B)
    (initial1 A × initial2 B)
    (λ a (p, q). transition1 A a p × transition2 B a q)
    (condition (condition1 A) (condition2 B))

```

```

lemma product-simps[simp]:
  alphabet3 (product A B) = alphabet1 A ∩ alphabet2 B
  initial3 (product A B) = initial1 A × initial2 B
  transition3 (product A B) a (p, q) = transition1 A a p × transition2 B a q
  condition3 (product A B) = condition (condition1 A) (condition2 B)
unfolding product-def by auto

```

```

lemma product-target[simp]:
  assumes length w = length r length r = length s
  shows c.target (w || r || s) (p, q) = (a.target (w || r) p, b.target (w || s) q)
  using assms by (induct arbitrary: p q rule: list-induct3) (auto)

```

```

lemma product-path[iff]:
  assumes length w = length r length r = length s
  shows c.path (product A B) (w || r || s) (p, q) ⟷
    a.path A (w || r) p ∧ b.path B (w || s) q
  using assms by (induct arbitrary: p q rule: list-induct3) (auto)

```

```

lemma product-run[iff]: c.run (product A B) (w ||| r ||| s) (p, q) ⟷
  a.run A (w ||| r) p ∧ b.run B (w ||| s) q

```

```

proof safe
  show a.run A (w ||| r) p if c.run (product A B) (w ||| r ||| s) (p, q)
    using that by (coinduction arbitrary: w r s p q) (force elim: c.run.cases)
  show b.run B (w ||| s) q if c.run (product A B) (w ||| r ||| s) (p, q)
    using that by (coinduction arbitrary: w r s p q) (force elim: c.run.cases)

```

```

show c.run (product A B) (w ||| r ||| s) (p, q) if a.run A (w ||| r) p b.run B
(w ||| s) q
using that by (coinduction arbitrary: w r s p q) (auto elim: a.run.cases
b.run.cases)
qed

```

```

lemma product-nodes: c.nodes (product A B) ⊆ a.nodes A × b.nodes B
proof
fix pq
assume pq ∈ c.nodes (product A B)
then show pq ∈ a.nodes A × b.nodes B by induct auto
qed

```

```

lemma product-nodes-finite[intro]:
assumes finite (a.nodes A) finite (b.nodes B)
shows finite (c.nodes (product A B))
using finite-subset product-nodes assms by blast

```

end

```

locale automaton-intersection-path =
automaton-product
automaton1 alphabet1 initial1 transition1 condition1
automaton2 alphabet2 initial2 transition2 condition2
automaton3 alphabet3 initial3 transition3 condition3
condition +
a: automaton-path automaton1 alphabet1 initial1 transition1 condition1 test1 +
b: automaton-path automaton2 alphabet2 initial2 transition2 condition2 test2 +
c: automaton-path automaton3 alphabet3 initial3 transition3 condition3 test3
for automaton1 alphabet1 initial1 transition1 condition1 test1
and automaton2 alphabet2 initial2 transition2 condition2 test2
and automaton3 alphabet3 initial3 transition3 condition3 test3
and condition
+
assumes test[iff]: length r = length w ⇒ length s = length w ⇒
test3 (condition c1 c2) w (r || s) (p, q) ⇔ test1 c1 w r p ∧ test2 c2 w s q
begin

```

```

lemma product-language[simp]: c.language (product A B) = a.language A ∩
b.language B
unfolding a.language-def b.language-def c.language-def by (force iff: split-zip)

```

end

```

locale automaton-intersection-run =
automaton-product
automaton1 alphabet1 initial1 transition1 condition1
automaton2 alphabet2 initial2 transition2 condition2
automaton3 alphabet3 initial3 transition3 condition3

```

condition +
a: automaton-run automaton₁ alphabet₁ initial₁ transition₁ condition₁ test₁ +
b: automaton-run automaton₂ alphabet₂ initial₂ transition₂ condition₂ test₂ +
c: automaton-run automaton₃ alphabet₃ initial₃ transition₃ condition₃ test₃
for *automaton₁ alphabet₁ initial₁ transition₁ condition₁ test₁*
and *automaton₂ alphabet₂ initial₂ transition₂ condition₂ test₂*
and *automaton₃ alphabet₃ initial₃ transition₃ condition₃ test₃*
and *condition*
+
assumes *test[iff]: test₃ (condition c₁ c₂) w (r ||| s) (p, q) \longleftrightarrow test₁ c₁ w r p*
 \wedge *test₂ c₂ w s q*
begin

lemma *product-language[simp]: c.language (product A B) = a.language A \cap b.language B*
unfolding *a.language-def b.language-def c.language-def* **by** (*fastforce iff: split-zip*)

end

locale *automaton-sum =*
a: automaton automaton₁ alphabet₁ initial₁ transition₁ condition₁ +
b: automaton automaton₂ alphabet₂ initial₂ transition₂ condition₂ +
c: automaton automaton₃ alphabet₃ initial₃ transition₃ condition₃
for *automaton₁ :: 'label set \Rightarrow 'state₁ set \Rightarrow ('label \Rightarrow 'state₁ \Rightarrow 'state₁ set) \Rightarrow*
'condition₁ \Rightarrow 'automaton₁
and *alphabet₁ initial₁ transition₁ condition₁*
and *automaton₂ :: 'label set \Rightarrow 'state₂ set \Rightarrow ('label \Rightarrow 'state₂ \Rightarrow 'state₂ set)*
 \Rightarrow *'condition₂ \Rightarrow 'automaton₂*
and *alphabet₂ initial₂ transition₂ condition₂*
and *automaton₃ :: 'label set \Rightarrow ('state₁ + 'state₂) set \Rightarrow ('label \Rightarrow 'state₁ +*
'state₂ \Rightarrow ('state₁ + 'state₂) set) \Rightarrow 'condition₃ \Rightarrow 'automaton₃
and *alphabet₃ initial₃ transition₃ condition₃*
+
fixes *condition :: 'condition₁ \Rightarrow 'condition₂ \Rightarrow 'condition₃*
begin

definition *sum :: 'automaton₁ \Rightarrow 'automaton₂ \Rightarrow 'automaton₃ where*
sum A B \equiv automaton₃
(alphabet₁ A \cup alphabet₂ B)
(initial₁ A $\langle + \rangle$ initial₂ B)
(λ a. λ Inl p \Rightarrow Inl ' transition₁ A a p | Inr q \Rightarrow Inr ' transition₂ B a q)
(condition (condition₁ A) (condition₂ B))

lemma *sum-simps[simp]:*
alphabet₃ (sum A B) = alphabet₁ A \cup alphabet₂ B
initial₃ (sum A B) = initial₁ A $\langle + \rangle$ initial₂ B
transition₃ (sum A B) a (Inl p) = Inl ' transition₁ A a p
transition₃ (sum A B) a (Inr q) = Inr ' transition₂ B a q

$condition_3 (sum A B) = condition (condition_1 A) (condition_2 B)$
unfolding *sum-def* **by** *auto*

lemma *path-sum-a*:

assumes $length\ r = length\ w\ a.path\ A\ (w\ ||\ r)\ p$
shows $c.path\ (sum\ A\ B)\ (w\ ||\ map\ Inl\ r)\ (Inl\ p)$
using *assms* **by** (*induct arbitrary: p rule: list-induct2*) (*auto*)

lemma *path-sum-b*:

assumes $length\ s = length\ w\ b.path\ B\ (w\ ||\ s)\ q$
shows $c.path\ (sum\ A\ B)\ (w\ ||\ map\ Inr\ s)\ (Inr\ q)$
using *assms* **by** (*induct arbitrary: q rule: list-induct2*) (*auto*)

lemma *sum-path*:

assumes $alphabet_1\ A = alphabet_2\ B$
assumes $length\ rs = length\ w\ c.path\ (sum\ A\ B)\ (w\ ||\ rs)\ pq$
obtains

(a) $r\ p$ **where** $rs = map\ Inl\ r\ pq = Inl\ p\ a.path\ A\ (w\ ||\ r)\ p$ |
(b) $s\ q$ **where** $rs = map\ Inr\ s\ pq = Inr\ q\ b.path\ B\ (w\ ||\ s)\ q$

proof (*cases pq*)

case (*Inl p*)

have 1: $rs = map\ Inl\ (map\ projl\ rs)$

using *assms*(2, 3) **unfolding** *Inl* **by** (*induct arbitrary: p rule: list-induct2*)

(*auto*)

have 2: $a.path\ A\ (w\ ||\ map\ projl\ rs)\ p$

using *assms*(2, 1, 3) **unfolding** *Inl* **by** (*induct arbitrary: p rule: list-induct2*)

(*auto*)

show *?thesis* **using** a 1 *Inl* 2 **by** *this*

next

case (*Inr q*)

have 1: $rs = map\ Inr\ (map\ projr\ rs)$

using *assms*(2, 3) **unfolding** *Inr* **by** (*induct arbitrary: q rule: list-induct2*)

(*auto*)

have 2: $b.path\ B\ (w\ ||\ map\ projr\ rs)\ q$

using *assms*(2, 1, 3) **unfolding** *Inr* **by** (*induct arbitrary: q rule: list-induct2*)

(*auto*)

show *?thesis* **using** b 1 *Inr* 2 **by** *this*

qed

lemma *run-sum-a*:

assumes $a.run\ A\ (w\ ||| r)\ p$
shows $c.run\ (sum\ A\ B)\ (w\ ||| smap\ Inl\ r)\ (Inl\ p)$
using *assms* **by** (*coinduction arbitrary: w r p*) (*force elim: a.run.cases*)

lemma *run-sum-b*:

assumes $b.run\ B\ (w\ ||| s)\ q$
shows $c.run\ (sum\ A\ B)\ (w\ ||| smap\ Inr\ s)\ (Inr\ q)$
using *assms* **by** (*coinduction arbitrary: w s q*) (*force elim: b.run.cases*)

lemma *sum-run*:

assumes $alphabet_1\ A = alphabet_2\ B$
assumes $c.run\ (sum\ A\ B)\ (w\ ||| rs)\ pq$
obtains

```

    (a)  $r p$  where  $rs = \text{smap Inl } r \text{ } pq = \text{Inl } p \text{ } a.\text{run } A (w \parallel r) p \mid$ 
    (b)  $s q$  where  $rs = \text{smap Inr } s \text{ } pq = \text{Inr } q \text{ } b.\text{run } B (w \parallel s) q$ 
proof (cases pq)
  case (Inl p)
    have 1:  $rs = \text{smap Inl } (\text{smap projl } rs)$ 
      using assms(2) unfolding Inl by (coinduction arbitrary:  $w \text{ } rs \text{ } p$ ) (force
elim: c.run.cases)
    have 2:  $a.\text{run } A (w \parallel \text{smap projl } rs) p$ 
      using assms unfolding Inl by (coinduction arbitrary:  $w \text{ } rs \text{ } p$ ) (force elim:
c.run.cases)
    show ?thesis using a 1 Inl 2 by this
  next
  case (Inr q)
    have 1:  $rs = \text{smap Inr } (\text{smap projr } rs)$ 
      using assms(2) unfolding Inr by (coinduction arbitrary:  $w \text{ } rs \text{ } q$ ) (force
elim: c.run.cases)
    have 2:  $b.\text{run } B (w \parallel \text{smap projr } rs) q$ 
      using assms unfolding Inr by (coinduction arbitrary:  $w \text{ } rs \text{ } q$ ) (force elim:
c.run.cases)
    show ?thesis using b 1 Inr 2 by this
qed

```

lemma *sum-nodes*:

```

  assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$ 
  shows  $c.\text{nodes } (\text{sum } A \text{ } B) \subseteq a.\text{nodes } A \langle + \rangle b.\text{nodes } B$ 

```

proof

```

  fix pq

```

```

  assume  $pq \in c.\text{nodes } (\text{sum } A \text{ } B)$ 

```

```

  then show  $pq \in a.\text{nodes } A \langle + \rangle b.\text{nodes } B$  using assms by (induct) (auto
0 3)
qed

```

lemma *sum-nodes-finite[intro]*:

```

  assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$ 
  assumes finite ( $a.\text{nodes } A$ ) finite ( $b.\text{nodes } B$ )
  shows finite ( $c.\text{nodes } (\text{sum } A \text{ } B)$ )
  using finite-subset sum-nodes assms by (auto intro: finite-Plus)

```

end

locale *automaton-union-path* =

```

  automaton-sum

```

```

  automaton1 alphabet1 initial1 transition1 condition1

```

```

  automaton2 alphabet2 initial2 transition2 condition2

```

```

  automaton3 alphabet3 initial3 transition3 condition3

```

```

  condition +

```

```

a: automaton-path automaton1 alphabet1 initial1 transition1 condition1 test1 +

```

```

b: automaton-path automaton2 alphabet2 initial2 transition2 condition2 test2 +

```

```

c: automaton-path automaton3 alphabet3 initial3 transition3 condition3 test3

```

```

for automaton1 alphabet1 initial1 transition1 condition1 test1
and automaton2 alphabet2 initial2 transition2 condition2 test2
and automaton3 alphabet3 initial3 transition3 condition3 test3
and condition
+
assumes test1[iff]: length r = length w  $\implies$  test3 (condition c1 c2) w (map Inl
r) (Inl p)  $\longleftrightarrow$  test1 c1 w r p
assumes test2[iff]: length s = length w  $\implies$  test3 (condition c1 c2) w (map Inr
s) (Inr q)  $\longleftrightarrow$  test2 c2 w s q
begin

lemma sum-language[simp]:
assumes alphabet1 A = alphabet2 B
shows c.language (sum A B) = a.language A  $\cup$  b.language B
using assms unfolding a.language-def b.language-def c.language-def
by (force intro: path-sum-a path-sum-b elim!: sum-path)

end

locale automaton-union-run =
automaton-sum
automaton1 alphabet1 initial1 transition1 condition1
automaton2 alphabet2 initial2 transition2 condition2
automaton3 alphabet3 initial3 transition3 condition3
condition +
a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2 +
c: automaton-run automaton3 alphabet3 initial3 transition3 condition3 test3
for automaton1 alphabet1 initial1 transition1 condition1 test1
and automaton2 alphabet2 initial2 transition2 condition2 test2
and automaton3 alphabet3 initial3 transition3 condition3 test3
and condition
+
assumes test1[iff]: test3 (condition c1 c2) w (smap Inl r) (Inl p)  $\longleftrightarrow$  test1 c1
w r p
assumes test2[iff]: test3 (condition c1 c2) w (smap Inr s) (Inr q)  $\longleftrightarrow$  test2 c2
w s q
begin

lemma sum-language[simp]:
assumes alphabet1 A = alphabet2 B
shows c.language (sum A B) = a.language A  $\cup$  b.language B
using assms unfolding a.language-def b.language-def c.language-def
by (auto intro: run-sum-a run-sum-b elim!: sum-run)

end

locale automaton-product-list =
a: automaton automaton1 alphabet1 initial1 transition1 condition1 +

```



```

    b: automaton automaton2 alphabet2 initial2 transition2 condition2
    for automaton1 :: 'label set ⇒ 'state set ⇒ ('label ⇒ 'state ⇒ 'state set) ⇒
'condition1 ⇒ 'automaton1
    and alphabet1 initial1 transition1 condition1
    and automaton2 :: 'label set ⇒ 'state list set ⇒ ('label ⇒ 'state list ⇒ 'state
list set) ⇒ 'condition2 ⇒ 'automaton2
    and alphabet2 initial2 transition2 condition2
    +
    fixes condition :: 'condition1 list ⇒ 'condition2
begin

```

```

definition product :: 'automaton1 list ⇒ 'automaton2 where

```

```

product AA ≡ automaton2
  (∩ (alphabet1 ' set AA))
  (listset (map initial1 AA))
  (λ a ps. listset (map2 (λ A p. transition1 A a p) AA ps))
  (condition (map condition1 AA))

```

```

lemma product-simps[simp]:

```

```

  alphabet2 (product AA) = ∩ (alphabet1 ' set AA)
  initial2 (product AA) = listset (map initial1 AA)
  transition2 (product AA) a ps = listset (map2 (λ A p. transition1 A a p) AA
ps)
  condition2 (product AA) = condition (map condition1 AA)
unfolding product-def by auto

```

```

lemma product-run-length:

```

```

  assumes length ps = length AA
  assumes b.run (product AA) (w ||| r) ps
  assumes qs ∈ sset r
  shows length qs = length AA

```

```

proof –

```

```

  have pred-stream (λ qs. length qs = length AA) r
    using assms(1, 2) by (coinduction arbitrary: w r ps)
    (force elim: b.run.cases simp: listset-member list-all2-conv-all-nth)
  then show ?thesis using assms(3) unfolding stream.pred-set by auto

```

```

qed

```

```

lemma product-run-stranspose:

```

```

  assumes length ps = length AA
  assumes b.run (product AA) (w ||| r) ps
  obtains rs where r = stranspose rs length rs = length AA

```

```

proof

```

```

  define rs where rs ≡ map (λ k. smap (λ ps. ps ! k) r) [0 ..< length AA]

```

```

  have length qs = length AA if qs ∈ sset r for qs using product-run-length
  asms that by this

```

```

  then show r = stranspose rs

```

```

    unfolding rs-def by (coinduction arbitrary: r) (force intro: nth-equalityI

```

```

simp: comp-def)

```

```

  show length rs = length AA unfolding rs-def by auto

```

qed

lemma *run-product*:

assumes $length\ rs = length\ AA$ $length\ ps = length\ AA$

assumes $\bigwedge k. k < length\ AA \implies a.run\ (AA\ !\ k)\ (w\ ||| rs\ !\ k)\ (ps\ !\ k)$

shows $b.run\ (product\ AA)\ (w\ ||| stranspose\ rs)\ ps$

using *assms*

proof (*coinduction arbitrary: w rs ps*)

case (*run ap r*)

then show ?*case*

proof (*intro conjI exI*)

show $fst\ ap \in alphabet_2\ (product\ AA)$

using *run by* (*force elim: a.run.cases simp: set-conv-nth*)

show $snd\ ap \in transition_2\ (product\ AA)\ (fst\ ap)\ ps$

using *run by* (*force elim: a.run.cases simp: listset-member list-all2-conv-all-nth*)

show $\forall k < length\ AA. a.run'\ (AA\ !\ k)\ (stl\ w\ ||| map\ stl\ rs\ !\ k)\ (map\ shd$

rs\ !\ k)

using *run by* (*force elim: a.run.cases*)

qed *auto*

qed

lemma *product-run*:

assumes $length\ rs = length\ AA$ $length\ ps = length\ AA$

assumes $b.run\ (product\ AA)\ (w\ ||| stranspose\ rs)\ ps$

shows $k < length\ AA \implies a.run\ (AA\ !\ k)\ (w\ ||| rs\ !\ k)\ (ps\ !\ k)$

using *assms*

proof (*coinduction arbitrary: w rs ps*)

case (*run ap wr*)

then show ?*case*

proof (*intro exI conjI*)

show $fst\ ap \in alphabet_1\ (AA\ !\ k)$

using *run by* (*force elim: b.run.cases*)

show $snd\ ap \in transition_1\ (AA\ !\ k)\ (fst\ ap)\ (ps\ !\ k)$

using *run by* (*force elim: b.run.cases simp: listset-member list-all2-conv-all-nth*)

show $b.run'\ (product\ AA)\ (stl\ w\ ||| stranspose\ (map\ stl\ rs))\ (shd\ (stranspose$

rs))

using *run by* (*force elim: b.run.cases*)

qed *auto*

qed

lemma *product-nodes*: $b.nodes\ (product\ AA) \subseteq listset\ (map\ a.nodes\ AA)$

proof

show $ps \in listset\ (map\ a.nodes\ AA)$ if $ps \in b.nodes\ (product\ AA)$ for *ps*

using *that by* (*induct*) (*auto 0 3 simp: listset-member list-all2-conv-all-nth*)

qed

lemma *product-nodes-finite*[*intro*]:

assumes *list-all* (*finite* \circ *a.nodes*) *AA*

shows *finite* (*b.nodes* (*product* *AA*))

using *list.pred-map product-nodes assms by* (*blast dest: finite-subset*)

lemma *product-nodes-card*:

assumes *list-all* (*finite* \circ *a.nodes*) *AA*

shows *card* (*b.nodes* (*product AA*)) \leq *prod-list* (*map* (*card* \circ *a.nodes*) *AA*)

proof –

have *card* (*b.nodes* (*product AA*)) \leq *card* (*listset* (*map a.nodes AA*))

using *list.pred-map product-nodes assms* **by** (*blast intro: card-mono*)

also have $\dots =$ *prod-list* (*map* (*card* \circ *a.nodes*) *AA*) **by** *simp*

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

qed

end

locale *automaton-intersection-list-run* =

automaton-product-list

*automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁

*automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂

condition +

a: *automaton-run* *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁ +

b: *automaton-run* *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

for *automaton*₁ *alphabet*₁ *initial*₁ *transition*₁ *condition*₁ *test*₁

and *automaton*₂ *alphabet*₂ *initial*₂ *transition*₂ *condition*₂ *test*₂

and *condition*

+

assumes *test*[*iff*]: *length rs* = *length cs* \implies *length ps* = *length cs* \implies

*test*₂ (*condition cs*) *w* (*stranspose rs*) *ps* \longleftrightarrow *list-all* (λ (*c*, *r*, *p*). *test*₁ *c w r*
p) (*cs* || *rs* || *ps*)

begin

lemma *product-language*[*simp*]: *b.language* (*product AA*) = \bigcap (*a.language* ‘*set AA*)

proof *safe*

fix *A w*

assume *1*: *w* \in *b.language* (*product AA*) *A* \in *set AA*

obtain *r ps* **where** *2*:

ps \in *initial*₂ (*product AA*)

b.run (*product AA*) (*w* ||| *r*) *ps*

*test*₂ (*condition*₂ (*product AA*)) *w r ps*

using *1(1)* **by** *auto*

have *3*: *length ps* = *length AA* **using** *2(1)* **by** (*simp add: listset-member list-all2-conv-all-nth*)

obtain *rs* **where** *4*: *r* = *stranspose rs* *length rs* = *length AA*

using *product-run-stranspose 3 2(2)* **by** *this*

obtain *k* **where** *5*: *k* < *length AA* *A* = *AA* ! *k* **using** *1(2)* **unfolding**
set-conv-nth **by** *auto*

show *w* \in *a.language A*

proof

show *ps* ! *k* \in *initial*₁ *A* **using** *2(1)* *5* **by** (*auto simp: listset-member list-all2-conv-all-nth*)

show *a.run A* (*w* ||| *rs* ! *k*) (*ps* ! *k*) **using** *2(2)* *3 4 5* **by** (*auto intro:*

```

product-run)
  show test1 (condition1 A) w (rs ! k) (ps ! k) using 2(3) 3 4 5 by (simp
add: list-all-length)
  qed
next
fix w
assume 1: w ∈ ∩ (a.language ' set AA)
have 2: ∀ A ∈ set AA. ∃ r p. p ∈ initial1 A ∧ a.run A (w ||| r) p ∧ test1
(condition1 A) w r p
  using 1 by blast
obtain rs ps where 3:
  length rs = length AA length ps = length AA
  ∧ k. k < length AA ⇒ ps ! k ∈ initial1 (AA ! k)
  ∧ k. k < length AA ⇒ a.run (AA ! k) (w ||| rs ! k) (ps ! k)
  ∧ k. k < length AA ⇒ test1 (condition1 (AA ! k)) w (rs ! k) (ps ! k)
  using 2
  unfolding Ball-set list-choice-zip list-choice-pair
  unfolding list.pred-set set-conv-nth
  by force
show w ∈ b.language (product AA)
proof
  show ps ∈ initial2 (product AA) using 3 by (auto simp: listset-member
list-all2-conv-all-nth)
  show b.run (product AA) (w ||| stranspose rs) ps using 3 by (auto intro:
run-product)
  show test2 (condition2 (product AA)) w (stranspose rs) ps using 3 by (auto
simp: list-all-length)
  qed
  qed
end

locale automaton-sum-list =
  a: automaton automaton1 alphabet1 initial1 transition1 condition1 +
  b: automaton automaton2 alphabet2 initial2 transition2 condition2
  for automaton1 :: 'label set ⇒ 'state set ⇒ ('label ⇒ 'state ⇒ 'state set) ⇒
'condition1 ⇒ 'automaton1
  and alphabet1 initial1 transition1 condition1
  and automaton2 :: 'label set ⇒ (nat × 'state) set ⇒ ('label ⇒ nat × 'state ⇒
(nat × 'state) set) ⇒ 'condition2 ⇒ 'automaton2
  and alphabet2 initial2 transition2 condition2
  +
  fixes condition :: 'condition1 list ⇒ 'condition2
begin

definition sum :: 'automaton1 list ⇒ 'automaton2 where
  sum AA ≡ automaton2
  (∪ (alphabet1 ' set AA))
  (∪ k < length AA. {k} × initial1 (AA ! k))

```

$(\lambda a (k, p). \{k\} \times \text{transition}_1 (AA ! k) a p)$
 $(\text{condition} (\text{map condition}_1 AA))$

lemma *sum-simps[simp]*:

$\text{alphabet}_2 (\text{sum } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$
 $\text{initial}_2 (\text{sum } AA) = (\bigcup k < \text{length } AA. \{k\} \times \text{initial}_1 (AA ! k))$
 $\text{transition}_2 (\text{sum } AA) a (k, p) = \{k\} \times \text{transition}_1 (AA ! k) a p$
 $\text{condition}_2 (\text{sum } AA) = \text{condition} (\text{map condition}_1 AA)$
unfolding *sum-def* **by** *auto*

lemma *run-sum*:

assumes $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$
assumes $A \in \text{set } AA$
assumes $a.\text{run } A (w \parallel s) p$
obtains k **where** $k < \text{length } AA$ $A = AA ! k$ $b.\text{run} (\text{sum } AA) (w \parallel \text{sconst } k \parallel s) (k, p)$
proof –
obtain k **where** $1: k < \text{length } AA$ $A = AA ! k$ **using** *assms(2)* **unfolding**
set-conv-nth **by** *auto*
show *?thesis*
proof
show $k < \text{length } AA$ $A = AA ! k$ **using** 1 **by** *this*
show $b.\text{run} (\text{sum } AA) (w \parallel \text{sconst } k \parallel s) (k, p)$
using *assms 1(2)* **by** (*coinduction arbitrary: w s p*) (*force elim: a.run.cases*)
qed
qed

lemma *sum-run*:

assumes $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$
assumes $k < \text{length } AA$
assumes $b.\text{run} (\text{sum } AA) (w \parallel r) (k, p)$
obtains s **where** $r = \text{sconst } k \parallel s$ $a.\text{run} (AA ! k) (w \parallel s) p$
proof
show $r = \text{sconst } k \parallel \text{smap snd } r$
using *assms* **by** (*coinduction arbitrary: w r p*) (*force elim: b.run.cases*)
show $a.\text{run} (AA ! k) (w \parallel \text{smap snd } r) p$
using *assms* **by** (*coinduction arbitrary: w r p*) (*force elim: b.run.cases*)
qed

lemma *sum-nodes*:

assumes $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$
shows $b.\text{nodes} (\text{sum } AA) \subseteq (\bigcup k < \text{length } AA. \{k\} \times a.\text{nodes} (AA ! k))$
proof
show $kp \in (\bigcup k < \text{length } AA. \{k\} \times a.\text{nodes} (AA ! k))$ **if** $kp \in b.\text{nodes} (\text{sum } AA)$ **for** kp
using *that assms* **by** (*induct*) (*auto 0 4*)
qed

lemma *sum-nodes-finite[intro]*:

assumes $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$

```

    assumes list-all (finite ◦ a.nodes) AA
    shows finite (b.nodes (sum AA))
  proof (rule finite-subset)
    show b.nodes (sum AA) ⊆ (⋃ k < length AA. {k} × a.nodes (AA ! k))
      using sum-nodes assms(1) by this
    show finite (⋃ k < length AA. {k} × a.nodes' (AA ! k))
      using assms(2) unfolding list-all-length by auto
  qed

end

locale automaton-union-list-run =
  automaton-sum-list
  automaton1 alphabet1 initial1 transition1 condition1
  automaton2 alphabet2 initial2 transition2 condition2
  condition +
  a: automaton-run automaton1 alphabet1 initial1 transition1 condition1 test1 +
  b: automaton-run automaton2 alphabet2 initial2 transition2 condition2 test2
  for automaton1 alphabet1 initial1 transition1 condition1 test1
  and automaton2 alphabet2 initial2 transition2 condition2 test2
  and condition
  +
  assumes test[iff]: k < length cs ⇒ test2 (condition cs) w (sconst k ||| r) (k,
  p) ⇔ test1 (cs ! k) w r p
  begin

  lemma sum-language[simp]:
    assumes ⋂ (alphabet1 ' set AA) = ⋃ (alphabet1 ' set AA)
    shows b.language (sum AA) = ⋃ (a.language ' set AA)
  proof
    show b.language (sum AA) ⊆ ⋃ (a.language ' set AA)
      using assms unfolding a.language-def b.language-def by (force elim:
  sum-run)
    show ⋃ (a.language ' set AA) ⊆ b.language (sum AA)
      using assms unfolding a.language-def b.language-def by (force elim!:
  run-sum)
  qed

end

end

```

13 Nondeterministic Finite Automata

```

theory NFA
imports ../Nondeterministic
begin

datatype ('label, 'state) nfa = nfa

```

(*alphabet*: 'label set)
 (*initial*: 'state set)
 (*transition*: 'label \Rightarrow 'state \Rightarrow 'state set)
 (*accepting*: 'state pred)

global-interpretation *nfa*: automaton *nfa* alphabet initial transition accepting
defines *path* = *nfa.path* **and** *run* = *nfa.run* **and** *reachable* = *nfa.reachable* **and**
nodes = *nfa.nodes*
by *unfold-locales auto*
global-interpretation *nfa*: automaton-path *nfa* alphabet initial transition ac-
 cepting $\lambda P w r p. P$ (*last* (*p # r*))
defines *language* = *nfa.language*
by *standard*

abbreviation *target* **where** *target* \equiv *nfa.target*
abbreviation *states* **where** *states* \equiv *nfa.states*
abbreviation *trace* **where** *trace* \equiv *nfa.trace*
abbreviation *successors* **where** *successors* \equiv *nfa.successors* *TYPE*('label)

global-interpretation *nfa*: automaton-intersection-path
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
 $\lambda c_1 c_2 (p, q). c_1 p \wedge c_2 q$
defines *intersect* = *nfa.product*
by (*unfold-locales*) (*auto simp*: *zip-eq-Nil-iff*)

global-interpretation *nfa*: automaton-union-path
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
nfa alphabet initial transition accepting $\lambda P w r p. P$ (*last* (*p # r*))
case-sum
defines *union* = *nfa.sum*
by (*unfold-locales*) (*auto simp*: *last-map*)

end

14 Deterministic Büchi Automata

theory *DBA*
imports *../Deterministic*
begin

datatype ('label, 'state) *dba* = *dba*
 (*alphabet*: 'label set)
 (*initial*: 'state)
 (*transition*: 'label \Rightarrow 'state \Rightarrow 'state)
 (*accepting*: 'state pred)

```

global-interpretation dba: automaton dba alphabet initial transition accepting
  defines path = dba.path and run = dba.run and reachable = dba.reachable
and nodes = dba.nodes
  by unfold-locales auto
global-interpretation dba: automaton-run dba alphabet initial transition accept-
ing  $\lambda P w r p. \text{infs } P (p \#\# r)$ 
  defines language = dba.language
  by standard

abbreviation target where target  $\equiv$  dba.target
abbreviation states where states  $\equiv$  dba.states
abbreviation trace where trace  $\equiv$  dba.trace

abbreviation successors where successors  $\equiv$  dba.successors TYPE('label)

end

```

15 Deterministic Generalized Büchi Automata

```

theory DGBA
imports ../Deterministic
begin

  datatype ('label, 'state) dgba = dgba
    (alphabet: 'label set)
    (initial: 'state)
    (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
    (accepting: 'state pred gen)

  global-interpretation dgba: automaton dgba alphabet initial transition accepting
    defines path = dgba.path and run = dgba.run and reachable = dgba.reachable
and nodes = dgba.nodes
  by unfold-locales auto
  global-interpretation dgba: automaton-run dgba alphabet initial transition ac-
cepting  $\lambda P w r p. \text{gen infs } P (p \#\# r)$ 
    defines language = dgba.language
    by standard

  abbreviation target where target  $\equiv$  dgba.target
  abbreviation states where states  $\equiv$  dgba.states
  abbreviation trace where trace  $\equiv$  dgba.trace
  abbreviation successors where successors  $\equiv$  dgba.successors TYPE('label)

end

```


16 Deterministic Büchi Automata Combinations

```
theory DBA-Combine
imports DBA DGBA
begin
```

```
global-interpretation degeneralization: automaton-degeneralization-run
  dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting  $\lambda P w r p. \text{gen infs}$ 
  P (p ## r)
  dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P (p}$ 
  ## r)
  fst id
defines degeneralize = degeneralization.degeneralize
by (unfold-locales) (auto simp flip: sscan-smap)
```

```
lemmas degeneralize-language[simp] = degeneralization.degeneralize-language[folded
DBA.language-def]
lemmas degeneralize-nodes-finite[iff] = degeneralization.degeneralize-nodes-finite[folded
DBA.nodes-def]
lemmas degeneralize-nodes-card = degeneralization.degeneralize-nodes-card[folded
DBA.nodes-def]
```

```
global-interpretation intersection: automaton-intersection-run
  dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P}$ 
  (p ## r)
  dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P}$ 
  (p ## r)
  dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting  $\lambda P w r p. \text{gen infs P (p ## r)}$ 
   $\lambda c_1 c_2. [c_1 \circ \text{fst}, c_2 \circ \text{snd}]$ 
defines intersect' = intersection.product
by unfold-locales auto
```

```
lemmas intersect'-language[simp] = intersection.product-language[folded DGBA.language-def]
lemmas intersect'-nodes-finite = intersection.product-nodes-finite[folded DGBA.nodes-def]
lemmas intersect'-nodes-card = intersection.product-nodes-card[folded DGBA.nodes-def]
```

```
global-interpretation union: automaton-union-run
  dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P}$ 
  (p ## r)
  dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P}$ 
  (p ## r)
  dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p. \text{infs P}$ 
  (p ## r)
   $\lambda c_1 c_2 pq. (c_1 \circ \text{fst}) pq \vee (c_2 \circ \text{snd}) pq$ 
defines union = union.product
by (unfold-locales) (simp del: comp-apply)
```

```
lemmas union-language = union.product-language
```

lemmas *union-nodes-finite* = *union.product-nodes-finite*
lemmas *union-nodes-card* = *union.product-nodes-card*

global-interpretation *intersection-list: automaton-intersection-list-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting $\lambda P w r p. \text{infs } P$
 $(p \#\# r)$
dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting $\lambda P w r p.$
gen infs $P (p \#\# r)$
 $\lambda cs. \text{map } (\lambda k pp. (cs ! k) (pp ! k)) [0 ..< \text{length } cs]$
defines *intersect-list'* = *intersection-list.product*
by (*unfold-locales*) (*auto simp: gen-def comp-def*)

lemmas *intersect-list'-language*[*simp*] = *intersection-list.product-language*[*folded*
DGBA.language-def]
lemmas *intersect-list'-nodes-finite* = *intersection-list.product-nodes-finite*[*folded*
DGBA.nodes-def]
lemmas *intersect-list'-nodes-card* = *intersection-list.product-nodes-card*[*folded*
DGBA.nodes-def]

global-interpretation *union-list: automaton-union-list-run*
dba.dba dba.alphabet dba.initial dba.transition dba.accepting $\lambda P w r p. \text{infs } P$
 $(p \#\# r)$
dba.dba dba.alphabet dba.initial dba.transition dba.accepting $\lambda P w r p. \text{infs } P$
 $(p \#\# r)$
 $\lambda cs pp. \exists k < \text{length } cs. (cs ! k) (pp ! k)$
defines *union-list* = *union-list.product*
by (*unfold-locales*) (*simp add: comp-def*)

lemmas *union-list-language* = *union-list.product-language*
lemmas *union-list-nodes-finite* = *union-list.product-nodes-finite*
lemmas *union-list-nodes-card* = *union-list.product-nodes-card*

abbreviation *intersect where* *intersect* $A B \equiv \text{degeneralize } (\text{intersect}' A B)$

lemma *intersect-language*[*simp*]: *DBA.language* (*intersect* $A B$) = *DBA.language*
 $A \cap \text{DBA.language } B$
by *simp*
lemma *intersect-nodes-finite*:
assumes *finite* (*DBA.nodes* A) *finite* (*DBA.nodes* B)
shows *finite* (*DBA.nodes* (*intersect* $A B$))
using *intersect'-nodes-finite* *assms* **by** *simp*
lemma *intersect-nodes-card*:
assumes *finite* (*DBA.nodes* A) *finite* (*DBA.nodes* B)
shows *card* (*DBA.nodes* (*intersect* $A B$)) $\leq 2 * \text{card } (\text{DBA.nodes } A) * \text{card}$
 $(\text{DBA.nodes } B)$
proof –
have *card* (*DBA.nodes* (*intersect* $A B$)) \leq

```

    max 1 (length (dgba.accepting (intersect' A B))) * card (DGBA.nodes (intersect'
A B))
    using degeneralize-nodes-card by this
    also have length (dgba.accepting (intersect' A B)) = 2 by simp
    also have card (DGBA.nodes (intersect' A B)) ≤ card (DBA.nodes A) * card
(DBA.nodes B)
    using intersect'-nodes-card assms by this
    finally show ?thesis by simp
qed

```

abbreviation *intersect-list* **where** *intersect-list* AA ≡ *degeneralize* (*intersect-list*' AA)

lemma *intersect-list-language*[simp]: *DBA.language* (*intersect-list* AA) = \bigcap (*DBA.language* 'set AA)

by simp

lemma *intersect-list-nodes-finite*:

assumes *list-all* (*finite* ∘ *DBA.nodes*) AA

shows *finite* (*DBA.nodes* (*intersect-list* AA))

using *intersect-list'-nodes-finite* assms **by** simp

lemma *intersect-list-nodes-card*:

assumes *list-all* (*finite* ∘ *DBA.nodes*) AA

shows *card* (*DBA.nodes* (*intersect-list* AA)) ≤ *max* 1 (*length* AA) * *prod-list* (*map* (*card* ∘ *DBA.nodes*) AA)

proof –

have *card* (*DBA.nodes* (*intersect-list* AA)) ≤

max 1 (*length* (dgba.accepting (intersect-list' AA))) * *card* (DGBA.nodes (intersect-list' AA))

using *degeneralize-nodes-card* **by** this

also have *length* (dgba.accepting (intersect-list' AA)) = *length* AA **by** simp

also have *card* (DGBA.nodes (intersect-list' AA)) ≤ *prod-list* (*map* (*card* ∘ *DBA.nodes*) AA)

using *intersect-list'-nodes-card* assms **by** this

finally show ?thesis **by** simp

qed

end

17 Deterministic Büchi Transition Automata

theory *DBTA*

imports ../Deterministic

begin

datatype ('label, 'state) *dbta* = *dbta*

(*alphabet*: 'label set)

(*initial*: 'state)

(*transition*: 'label ⇒ 'state ⇒ 'state)

(*accepting*: ('state × 'label × 'state) pred)

```

global-interpretation dbta: automaton dbta alphabet initial transition accepting
  defines path = dbta.path and run = dbta.run and reachable = dbta.reachable
and nodes = dbta.nodes
  by unfold-locales auto
global-interpretation dbta: automaton-run dbta alphabet initial transition ac-
cepting
   $\lambda P w r p. \text{infs } P (p \#\# r \|\| w \|\| r)$ 
  defines language = dbta.language
  by standard

abbreviation target where target  $\equiv$  dbta.target
abbreviation states where states  $\equiv$  dbta.states
abbreviation trace where trace  $\equiv$  dbta.trace
abbreviation successors where successors  $\equiv$  dbta.successors TYPE('label)

end

```

18 Deterministic Generalized Büchi Transition Automata

```

theory DGBTA
imports ../Deterministic
begin

  datatype ('label, 'state) dgba = dgba
    (alphabet: 'label set)
    (initial: 'state)
    (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
    (accepting: ('state  $\times$  'label  $\times$  'state) pred gen)

  global-interpretation dgba: automaton dgba alphabet initial transition accept-
ing
  defines path = dgba.path and run = dgba.run and reachable = dgba.reachable
and nodes = dgba.nodes
  by unfold-locales auto
  global-interpretation dgba: automaton-run dgba alphabet initial transition
accepting
   $\lambda P w r p. \text{gen infs } P (p \#\# r \|\| w \|\| r)$ 
  defines language = dgba.language
  by standard

  abbreviation target where target  $\equiv$  dgba.target
  abbreviation states where states  $\equiv$  dgba.states
  abbreviation trace where trace  $\equiv$  dgba.trace
  abbreviation successors where successors  $\equiv$  dgba.successors TYPE('label)

end

```

19 Deterministic Büchi Transition Automata Combinations

```
theory DBTA-Combine
imports DBTA DGBTA
begin
```

```
global-interpretation degeneralization: automaton-degeneralization-run
  dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting  $\lambda P w r p.$  gen
infs P (p ## r ||| w ||| r)
  dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting  $\lambda P w r p.$  infs P
(p ## r ||| w ||| r)
  id  $\lambda ((p, k), a, (q, l)). ((p, a, q), k)$ 
defines degeneralize = degeneralization.degeneralize
proof
  fix w :: 'a stream
  fix r :: 'b stream
  fix cs p k
  let ?f =  $\lambda ((p, k), a, (q, l)). ((p, a, q), k)$ 
  let ?s = sscan (count cs  $\circ$  id) (p ## r ||| w ||| r) k
  have infs (degen cs  $\circ$  ?f) ((p, k) ## (r ||| ?s) ||| w ||| (r ||| ?s))  $\longleftrightarrow$ 
    infs (degen cs) (smap ?f ((p, k) ## (r ||| ?s) ||| w ||| (r ||| ?s)))
    by (simp add: comp-def)
  also have smap ?f ((p, k) ## (r ||| ?s) ||| w ||| (r ||| ?s)) = (p ## r ||| w |||
r) ||| k ## ?s
    by (coinduction arbitrary: p k r w) (auto simp: eq-scons simp flip: szip-unfold
sscan-scons)
  also have ... = (p ## r ||| w ||| r) ||| k ## sscan (count cs) (p ## r ||| w
||| r) k by simp
  also have infs (degen cs) ... = gen infs cs (p ## r ||| w ||| r) using degen-infs
by this
  finally show infs (degen cs  $\circ$  ?f) ((p, k) ## (r ||| ?s) ||| w ||| (r ||| ?s))  $\longleftrightarrow$ 
    gen infs cs (p ## r ||| w ||| r) by this
qed
```

```
lemmas degeneralize-language[simp] = degeneralization.degeneralize-language[folded
DBTA.language-def]
lemmas degeneralize-nodes-finite[iff] = degeneralization.degeneralize-nodes-finite[folded
DBTA.nodes-def]
lemmas degeneralize-nodes-card = degeneralization.degeneralize-nodes-card[folded
DBTA.nodes-def]
```

```
global-interpretation intersection: automaton-intersection-run
  dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting  $\lambda P w r p.$  infs
P (p ## r ||| w ||| r)
  dgba.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting  $\lambda P w r p.$  infs
P (p ## r ||| w ||| r)
  dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting  $\lambda P w r$ 
```

p. gen infs $P (p \#\# r \parallel w \parallel r)$
 $\lambda c_1 c_2. [c_1 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)), c_2 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_2, a, q_2))]$
defines *intersect'* = *intersection.product*
proof
fix $w :: 'a \text{ stream}$
fix $u :: 'b \text{ stream}$
fix $v :: 'c \text{ stream}$
fix $c_1 c_2 p q$
let $?tfst = \lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)$
let $?tsnd = \lambda ((p_1, p_2), a, (q_1, q_2)). (p_2, a, q_2)$
have *gen infs* $[c_1 \circ ?tfst, c_2 \circ ?tsnd] ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) \longleftrightarrow$
 $infs\ c_1 (smap\ ?tfst\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v)) \wedge$
 $infs\ c_2 (smap\ ?tsnd\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v))$
unfolding *gen-def* **by** (*simp add: comp-def*)
also have $smap\ ?tfst\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) = p \#\# u \parallel w \parallel u$
by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)
also have $smap\ ?tsnd\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) = q \#\# v \parallel w \parallel v$
by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)
finally show *gen infs* $[c_1 \circ ?tfst, c_2 \circ ?tsnd] ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) \longleftrightarrow$
 $infs\ c_1 (p \#\# u \parallel w \parallel u) \wedge infs\ c_2 (q \#\# v \parallel w \parallel v)$ **by this**
qed

lemmas *intersect'-language[simp]* = *intersection.product-language[folded DGBTA.language-def]*

lemmas *intersect'-nodes-finite* = *intersection.product-nodes-finite[folded DGBTA.nodes-def]*

lemmas *intersect'-nodes-card* = *intersection.product-nodes-card[folded DGBTA.nodes-def]*

global-interpretation *union: automaton-union-run*

$dbta.dbta\ dbta.alphabet\ dbta.initial\ dbta.transition\ dbta.accepting\ \lambda P\ w\ r\ p.\ infs$
 $P (p \#\# r \parallel w \parallel r)$
 $dbta.dbta\ dbta.alphabet\ dbta.initial\ dbta.transition\ dbta.accepting\ \lambda P\ w\ r\ p.\ infs$
 $P (p \#\# r \parallel w \parallel r)$
 $dbta.dbta\ dbta.alphabet\ dbta.initial\ dbta.transition\ dbta.accepting\ \lambda P\ w\ r\ p.\ infs$
 $P (p \#\# r \parallel w \parallel r)$
 $\lambda c_1 c_2 pq. (c_1 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)))\ pq \vee (c_2 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_2, a, q_2)))\ pq$
defines *union* = *union.product*

proof

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

fix $u :: 'b \text{ stream}$

fix $v :: 'c \text{ stream}$

fix $c_1 c_2 p q$

let $?tfst = \lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)$

let $?tsnd = \lambda ((p_1, p_2), a, (q_1, q_2)). (p_2, a, q_2)$

have $infs\ (\lambda pq. (c_1 \circ (\lambda ((p_1, p_2), a, q_1, q_2)). (p_1, a, q_1)))\ pq \vee$

$(c_2 \circ (\lambda ((p_1, p_2), a, q_1, q_2). (p_2, a, q_2))) pq) ((p, q) \#\# (u \parallel v) \parallel w \parallel (u \parallel v)) \longleftrightarrow$
 $infs\ c_1\ (smap\ ?tfst\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v)) \vee$
 $infs\ c_2\ (smap\ ?tsnd\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v))$
by (*simp add: comp-def*)
also have $smap\ ?tfst\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) = p \#\# u \parallel w \parallel u$
by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)
also have $smap\ ?tsnd\ ((p, q) \#\# (u \parallel v) \parallel w \parallel u \parallel v) = q \#\# v \parallel w \parallel v$
by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)
finally show $infs\ (\lambda pq. (c_1 \circ (\lambda ((p_1, p_2), a, q_1, q_2). (p_1, a, q_1))) pq \vee$
 $(c_2 \circ (\lambda ((p_1, p_2), a, q_1, q_2). (p_2, a, q_2))) pq) ((p, q) \#\# (u \parallel v) \parallel w \parallel (u \parallel v)) \longleftrightarrow$
 $infs\ c_1\ (p \#\# u \parallel w \parallel u) \vee infs\ c_2\ (q \#\# v \parallel w \parallel v)$ **by this**
qed

lemmas *union-language = union.product-language*
lemmas *union-nodes-finite = union.product-nodes-finite*
lemmas *union-nodes-card = union.product-nodes-card*

abbreviation *intersect where intersect A B \equiv degeneralize (intersect' A B)*

lemma *intersect-language[*simp*]: DBTA.language (intersect A B) = DBTA.language A \cap DBTA.language B*

by *simp*

lemma *intersect-nodes-finite:*

assumes *finite (DBTA.nodes A) finite (DBTA.nodes B)*

shows *finite (DBTA.nodes (intersect A B))*

using *intersect'-nodes-finite* **assms** **by** *simp*

lemma *intersect-nodes-card:*

assumes *finite (DBTA.nodes A) finite (DBTA.nodes B)*

shows $card\ (DBTA.nodes\ (intersect\ A\ B)) \leq 2 * card\ (DBTA.nodes\ A) * card\ (DBTA.nodes\ B)$

proof –

have $card\ (DBTA.nodes\ (intersect\ A\ B)) \leq$

$max\ 1\ (length\ (dgbta.accepting\ (intersect'\ A\ B))) * card\ (DGBTA.nodes\ (intersect'\ A\ B))$

using *degeneralize-nodes-card* **by this**

also have $length\ (dgbta.accepting\ (intersect'\ A\ B)) = 2$ **by** *simp*

also have $card\ (DGBTA.nodes\ (intersect'\ A\ B)) \leq card\ (DBTA.nodes\ A) * card\ (DBTA.nodes\ B)$

using *intersect'-nodes-card* **assms** **by this**

finally show *?thesis* **by** *simp*

qed

end

20 Deterministic Co-Büchi Automata

```
theory DCA
imports ../Deterministic
begin

  datatype ('label, 'state) dca = dca
    (alphabet: 'label set)
    (initial: 'state)
    (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
    (rejecting: 'state  $\Rightarrow$  bool)

  global-interpretation dca: automaton dca alphabet initial transition rejecting
  defines path = dca.path and run = dca.run and reachable = dca.reachable
and nodes = dca.nodes
  by unfold-locales auto
  global-interpretation dca: automaton-run dca alphabet initial transition reject-
ing  $\lambda P w r p. fins P (p \#\# r)$ 
  defines language = dca.language
  by standard

  abbreviation target where target  $\equiv$  dca.target
  abbreviation states where states  $\equiv$  dca.states
  abbreviation trace where trace  $\equiv$  dca.trace
  abbreviation successors where successors  $\equiv$  dca.successors TYPE('label)

end
```

21 Deterministic Co-Generalized Co-Büchi Automata

```
theory DGCA
imports ../Deterministic
begin

  datatype ('label, 'state) dgca = dgca
    (alphabet: 'label set)
    (initial: 'state)
    (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
    (rejecting: 'state pred gen)

  global-interpretation dgca: automaton dgca alphabet initial transition rejecting
  defines path = dgca.path and run = dgca.run and reachable = dgca.reachable
and nodes = dgca.nodes
  by unfold-locales auto
  global-interpretation dgca: automaton-run dgca alphabet initial transition reject-
ing  $\lambda P w r p. cogen fins P (p \#\# r)$ 
  defines language = dgca.language
  by standard
```


abbreviation *target* **where** *target* \equiv *dgca.target*
abbreviation *states* **where** *states* \equiv *dgca.states*
abbreviation *trace* **where** *trace* \equiv *dgca.trace*
abbreviation *successors* **where** *successors* \equiv *dgca.successors* *TYPE('label)*

end

22 Deterministic Co-Büchi Automata Combinations

theory *DCA-Combine*
imports *DCA DGCA*
begin

global-interpretation *degeneralization: automaton-degeneralization-run*
dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting λ P w r p. cogen
fins P (p ## r)
dca dca.alphabet dca.initial dca.transition dca.rejecting λ P w r p. fins P (p ##
r)
fst id
defines *degeneralize* = *degeneralization.degeneralize*
by (*unfold-locales*) (*auto simp flip: sscan-smap*)

lemmas *degeneralize-language[simp]* = *degeneralization.degeneralize-language[folded*
DCA.language-def]
lemmas *degeneralize-nodes-finite[iff]* = *degeneralization.degeneralize-nodes-finite[folded*
DCA.nodes-def]
lemmas *degeneralize-nodes-card* = *degeneralization.degeneralize-nodes-card[folded*
DCA.nodes-def]

global-interpretation *intersection: automaton-intersection-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ P w r p. fins P (p
r)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ P w r p. fins P (p
r)
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ P w r p. fins P (p
r)
 λ c₁ c₂ pq. (c₁ \circ fst) pq \vee (c₂ \circ snd) pq
defines *intersect* = *intersection.product*
by (*unfold-locales*) (*simp del: comp-apply*)

lemmas *intersect-language* = *intersection.product-language*
lemmas *intersect-nodes-finite* = *intersection.product-nodes-finite*
lemmas *intersect-nodes-card* = *intersection.product-nodes-card*

global-interpretation *union: automaton-union-run*
dca.dca dca.alphabet dca.initial dca.transition dca.rejecting λ P w r p. fins P (p
r)

```

    dca.dca dca.alphabet dca.initial dca.transition dca.rejecting  $\lambda P w r p$ . fins  $P (p$ 
##  $r)$ 
    dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting  $\lambda P w r p$ .
cogen fins  $P (p ## r)$ 
     $\lambda c_1 c_2$ . [ $c_1 \circ fst$ ,  $c_2 \circ snd$ ]
    defines union' = union.product
    by unfold-locales auto

```

```

lemmas union'-language[simp] = union.product-language[folded DGCA.language-def]
lemmas union'-nodes-finite = union.product-nodes-finite[folded DGCA.nodes-def]
lemmas union'-nodes-card = union.product-nodes-card[folded DGCA.nodes-def]

```

```

global-interpretation intersection-list: automaton-intersection-list-run
    dca.dca dca.alphabet dca.initial dca.transition dca.rejecting  $\lambda P w r p$ . fins  $P (p$ 
##  $r)$ 
    dca.dca dca.alphabet dca.initial dca.transition dca.rejecting  $\lambda P w r p$ . fins  $P (p$ 
##  $r)$ 
     $\lambda cs pp$ .  $\exists k < length\ cs$ . ( $cs ! k$ ) ( $pp ! k$ )
    defines intersect-list = intersection-list.product
    by (unfold-locales) (simp add: comp-def)

```

```

lemmas intersect-list-language = intersection-list.product-language
lemmas intersect-list-nodes-finite = intersection-list.product-nodes-finite
lemmas intersect-list-nodes-card = intersection-list.product-nodes-card

```

```

global-interpretation union-list: automaton-union-list-run
    dca.dca dca.alphabet dca.initial dca.transition dca.rejecting  $\lambda P w r p$ . fins  $P (p$ 
##  $r)$ 
    dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting  $\lambda P w r p$ .
cogen fins  $P (p ## r)$ 
     $\lambda cs$ . map ( $\lambda k pp$ . ( $cs ! k$ ) ( $pp ! k$ )) [ $0 .. < length\ cs$ ]
    defines union-list' = union-list.product
    by (unfold-locales) (auto simp: cogen-def comp-def)

```

```

lemmas union-list'-language[simp] = union-list.product-language[folded DGCA.language-def]
lemmas union-list'-nodes-finite = union-list.product-nodes-finite[folded DGCA.nodes-def]
lemmas union-list'-nodes-card = union-list.product-nodes-card[folded DGCA.nodes-def]

```

```

abbreviation union where union  $A B \equiv degeneralize (union' A B)$ 

```

```

lemma union-language[simp]:
  assumes dca.alphabet  $A = dca.alphabet B$ 
  shows DCA.language (union  $A B$ ) = DCA.language  $A \cup DCA.language B$ 
  using assms by simp
lemma union-nodes-finite:
  assumes finite (DCA.nodes  $A$ ) finite (DCA.nodes  $B$ )
  shows finite (DCA.nodes (union  $A B$ ))
  using union'-nodes-finite assms by simp
lemma union-nodes-card:

```

```

assumes finite (DCA.nodes A) finite (DCA.nodes B)
shows card (DCA.nodes (union A B)) ≤ 2 * card (DCA.nodes A) * card
(DCA.nodes B)
proof –
  have card (DCA.nodes (union A B)) ≤
    max 1 (length (dgca.rejecting (union' A B))) * card (DGCA.nodes (union' A
B))
  using degeneralize-nodes-card by this
  also have length (dgca.rejecting (union' A B)) = 2 by simp
  also have card (DGCA.nodes (union' A B)) ≤ card (DCA.nodes A) * card
(DCA.nodes B)
  using union'-nodes-card assms by this
  finally show ?thesis by simp
qed

```

abbreviation *union-list* **where** *union-list* AA ≡ *degeneralize* (*union-list'* AA)

```

lemma union-list-language[simp]:
  assumes  $\bigcap$  (dca.alphabet ' set AA) =  $\bigcup$  (dca.alphabet ' set AA)
  shows DCA.language (union-list AA) =  $\bigcup$  (DCA.language ' set AA)
  using assms by simp
lemma union-list-nodes-finite:
  assumes list-all (finite ∘ DCA.nodes) AA
  shows finite (DCA.nodes (union-list AA))
  using union-list'-nodes-finite assms by simp
lemma union-list-nodes-card:
  assumes list-all (finite ∘ DCA.nodes) AA
  shows card (DCA.nodes (union-list AA)) ≤ max 1 (length AA) * prod-list (map
(card ∘ DCA.nodes) AA)
  proof –
    have card (DCA.nodes (union-list AA)) ≤
      max 1 (length (dgca.rejecting (union-list' AA))) * card (DGCA.nodes (union-list'
AA))
    using degeneralize-nodes-card by this
    also have length (dgca.rejecting (union-list' AA)) = length AA by simp
    also have card (DGCA.nodes (union-list' AA)) ≤ prod-list (map (card ∘
DCA.nodes) AA)
    using union-list'-nodes-card assms by this
    finally show ?thesis by simp
  qed

```

end

23 Deterministic Rabin Automata

```

theory DRA
imports ../Deterministic
begin

```

```

datatype ('label, 'state) dra = dra
  (alphabet: 'label set)
  (initial: 'state)
  (transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state)
  (condition: 'state rabin gen)

global-interpretation dra: automaton dra alphabet initial transition condition
  defines path = dra.path and run = dra.run and reachable = dra.reachable
and nodes = dra.nodes
  by unfold-locales auto
global-interpretation dra: automaton-run dra alphabet initial transition condi-
tion  $\lambda P w r p$ . cogen rabin P (p ## r)
  defines language = dra.language
  by standard

abbreviation target where target  $\equiv$  dra.target
abbreviation states where states  $\equiv$  dra.states
abbreviation trace where trace  $\equiv$  dra.trace
abbreviation successors where successors  $\equiv$  dra.successors TYPE('label)

end

```

24 Deterministic Rabin Automata Combinations

```

theory DRA-Combine
imports DRA ../DBA/DBA ../DCA/DCA
begin

  global-interpretation intersection-bc: automaton-intersection-run
    dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p$ . infs P
  (p ## r)
    dca.dca dca.alphabet dca.initial dca.transition dca.rejecting  $\lambda P w r p$ . fins P (p
  ## r)
    dra.dra dra.alphabet dra.initial dra.transition dra.condition  $\lambda P w r p$ . cogen
  rabin P (p ## r)
     $\lambda c_1 c_2$ . [(c1  $\circ$  fst, c2  $\circ$  snd)]
  defines intersect-bc = intersection-bc.product
  by (unfold-locales) (simp add: cogen-def rabin-def)

  lemmas intersect-bc-language[simp] = intersection-bc.product-language[folded DCA.language-def
  DRA.language-def]
  lemmas intersect-bc-nodes-finite = intersection-bc.product-nodes-finite[folded DCA.nodes-def
  DRA.nodes-def]
  lemmas intersect-bc-nodes-card = intersection-bc.product-nodes-card[folded DCA.nodes-def
  DRA.nodes-def]

  global-interpretation union-list: automaton-union-list-run

```

```

    dra.dra dra.alphabet dra.initial dra.transition dra.condition λ P w r p. cogen
rabin P (p ## r)
    dra.dra dra.alphabet dra.initial dra.transition dra.condition λ P w r p. cogen
rabin P (p ## r)
    λ cs. do { k ← [0 ..< length cs]; (f, g) ← cs ! k; [(λ pp. f (pp ! k), λ pp. g (pp
! k))] }
    defines union-list = union-list.product
    by (unfold-locales) (auto simp: cogen-def rabin-def comp-def split-beta)

lemmas union-list-language = union-list.product-language
lemmas union-list-nodes-finite = union-list.product-nodes-finite
lemmas union-list-nodes-card = union-list.product-nodes-card

```

end

25 Relations and Refinement

theory *Refine*

imports

Automatic-Refinement.Automatic-Refinement

Refine-Monadic.Refine-Foreach

Sequence-LTL

Maps

begin

25.1 Predicate to Set Conversion Setup

lemma *right-unique-pred-set-conv*[*pred-set-conv*]: *right-unique* = *single-valuedp*

unfolding *right-unique-def single-valuedp-def* **by** *auto*

lemma *bi-unique-pred-set-conv*[*pred-set-conv*]: *bi-unique* (λ x y. (x, y) ∈ R) ↔
bijjective R

unfolding *bi-unique-def bijjective-def* **by** *blast*

useful for unfolding equality constants in theorems about predicates

lemma *pred-Id*: *HOL.eq* = (λ x y. (x, y) ∈ *Id*) **by** *simp*

lemma *pred-bool-Id*: *HOL.eq* = (λ x y. (x, y) ∈ (*Id* :: *bool rel*)) **by** *simp*

lemma *pred-nat-Id*: *HOL.eq* = (λ x y. (x, y) ∈ (*Id* :: *nat rel*)) **by** *simp*

lemma *pred-set-Id*: *HOL.eq* = (λ x y. (x, y) ∈ (*Id* :: '*a set rel*)) **by** *simp*

lemma *pred-list-Id*: *HOL.eq* = (λ x y. (x, y) ∈ (*Id* :: '*a list rel*)) **by** *simp*

lemma *pred-stream-Id*: *HOL.eq* = (λ x y. (x, y) ∈ (*Id* :: '*a stream rel*)) **by** *simp*

lemma *eq-onp-Id-on-eq*[*pred-set-conv*]: *eq-onp* (λ a. a ∈ A) = (λ x y. (x, y) ∈
Id-on A)

unfolding *eq-onp-def* **by** *auto*

lemma *rel-fun-fun-rel-eq*[*pred-set-conv*]:

rel-fun (λ x y. (x, y) ∈ A) (λ x y. (x, y) ∈ B) = (λ f g. (f, g) ∈ A → B)

by (*force simp: rel-fun-def fun-rel-def*)

lemma *rel-prod-prod-rel-eq*[*pred-set-conv*]:

$rel\text{-}prod (\lambda x y. (x, y) \in A) (\lambda x y. (x, y) \in B) = (\lambda f g. (f, g) \in A \times_r B)$
by (*force simp: prod-rel-def elim: rel-prod.cases*)
lemma *rel-sum-sum-rel-eq*[*pred-set-conv*]:
 $rel\text{-}sum (\lambda x y. (x, y) \in A) (\lambda x y. (x, y) \in B) = (\lambda f g. (f, g) \in \langle A, B \rangle sum\text{-}rel)$
by (*force simp: sum-rel-def elim: rel-sum.cases*)
lemma *rel-set-set-rel-eq*[*pred-set-conv*]:
 $rel\text{-}set (\lambda x y. (x, y) \in A) = (\lambda f g. (f, g) \in \langle A \rangle set\text{-}rel)$
unfolding *rel-set-def set-rel-def* **by** *simp*
lemma *rel-option-option-rel-eq*[*pred-set-conv*]:
 $rel\text{-}option (\lambda x y. (x, y) \in A) = (\lambda f g. (f, g) \in \langle A \rangle option\text{-}rel)$
by (*force simp: option-rel-def elim: option.rel-cases*)

thm *image-transfer image-transfer*[*to-set*]
thm *fun-upd-transfer fun-upd-transfer*[*to-set*]

25.2 Relation Composition

lemma *relcomp-trans-1*[*trans*]:
assumes $(f, g) \in A_1$
assumes $(g, h) \in A_2$
shows $(f, h) \in A_1 \circ A_2$
using *relcompI assms* **by** *this*
lemma *relcomp-trans-2*[*trans*]:
assumes $(f, g) \in A_1 \rightarrow B_1$
assumes $(g, h) \in A_2 \rightarrow B_2$
shows $(f, h) \in A_1 \circ A_2 \rightarrow B_1 \circ B_2$
proof –
note *assms(1)*
also note *assms(2)*
also note
fun-rel-comp-dist
finally show *?thesis* **by** *this*
qed
lemma *relcomp-trans-3*[*trans*]:
assumes $(f, g) \in A_1 \rightarrow B_1 \rightarrow C_1$
assumes $(g, h) \in A_2 \rightarrow B_2 \rightarrow C_2$
shows $(f, h) \in A_1 \circ A_2 \rightarrow B_1 \circ B_2 \rightarrow C_1 \circ C_2$
proof –
note *assms(1)*
also note *assms(2)*
also note
fun-rel-mono[*OF order-refl*
fun-rel-comp-dist]
finally show *?thesis* **by** *this*
qed
lemma *relcomp-trans-4*[*trans*]:
assumes $(f, g) \in A_1 \rightarrow B_1 \rightarrow C_1 \rightarrow D_1$
assumes $(g, h) \in A_2 \rightarrow B_2 \rightarrow C_2 \rightarrow D_2$

shows $(f, h) \in A_1 \ O \ A_2 \rightarrow B_1 \ O \ B_2 \rightarrow C_1 \ O \ C_2 \rightarrow D_1 \ O \ D_2$
proof –
note *assms(1)*
also note *assms(2)*
also note
fun-rel-mono[OF order-refl
fun-rel-mono[OF order-refl
fun-rel-comp-dist]]
finally show *?thesis by this*
qed
lemma *relcomp-trans-5[trans]*:
assumes $(f, g) \in A_1 \rightarrow B_1 \rightarrow C_1 \rightarrow D_1 \rightarrow E_1$
assumes $(g, h) \in A_2 \rightarrow B_2 \rightarrow C_2 \rightarrow D_2 \rightarrow E_2$
shows $(f, h) \in A_1 \ O \ A_2 \rightarrow B_1 \ O \ B_2 \rightarrow C_1 \ O \ C_2 \rightarrow D_1 \ O \ D_2 \rightarrow E_1 \ O \ E_2$
proof –
note *assms(1)*
also note *assms(2)*
also note
fun-rel-mono[OF order-refl
fun-rel-mono[OF order-refl
fun-rel-mono[OF order-refl
fun-rel-comp-dist]]]
finally show *?thesis by this*
qed

25.3 Relation Basics

lemma *inv-fun-rel-eq[simp]*: $(A \rightarrow B)^{-1} = A^{-1} \rightarrow B^{-1}$
by *(auto dest: fun-relD)*
lemma *inv-option-rel-eq[simp]*: $(\langle K \rangle \text{option-rel})^{-1} = \langle K^{-1} \rangle \text{option-rel}$
by *(auto simp: option-rel-def)*
lemma *inv-prod-rel-eq[simp]*: $(P \times_r Q)^{-1} = P^{-1} \times_r Q^{-1}$
by *(auto)*
lemma *inv-sum-rel-eq[simp]*: $(\langle P, Q \rangle \text{sum-rel})^{-1} = \langle P^{-1}, Q^{-1} \rangle \text{sum-rel}$
by *(auto simp: sum-rel-def)*
lemma *set-rel-converse[simp]*: $(\langle A \rangle \text{set-rel})^{-1} = \langle A^{-1} \rangle \text{set-rel}$ **unfolding** *set-rel-def*
by *auto*

lemma *build-rel-domain[simp]*: $\text{Domain} (br \ \alpha \ I) = \text{Collect } I$ **unfolding** *build-rel-def*
by *auto*
lemma *build-rel-range[simp]*: $\text{Range} (br \ \alpha \ I) = \alpha \ ' \ \text{Collect } I$ **unfolding** *build-rel-def*
by *auto*
lemma *build-rel-image[simp]*: $br \ \alpha \ I \ ' \ A = \alpha \ ' \ (A \cap \text{Collect } I)$ **unfolding**
build-rel-def **by** *auto*

lemma *prod-rel-domain[simp]*: $\text{Domain} (A \times_r B) = \text{Domain } A \times \text{Domain } B$
unfolding *prod-rel-def* **by** *auto*
lemma *prod-rel-range[simp]*: $\text{Range} (A \times_r B) = \text{Range } A \times \text{Range } B$ **unfolding**
prod-rel-def **by** *auto*

lemma *member-Id-on*[*iff*]: $(x, y) \in \text{Id-on } A \iff x = y \wedge y \in A$ **unfolding**
Id-on-def **by** *auto*
lemma *bijjective-Id-on*[*intro!*, *simp*]: *bijjective* (*Id-on* *A*) **unfolding** *bijjective-def*
by *auto*
lemma *relcomp-Id-on*[*simp*]: *Id-on* *A* *O* *Id-on* *B* = *Id-on* (*A* \cap *B*) **by** *auto*

lemma *prod-rel-Id-on*[*simp*]: *Id-on* *A* \times_r *Id-on* *B* = *Id-on* (*A* \times *B*) **by** *auto*
lemma *set-rel-Id-on*[*simp*]: $\langle \text{Id-on } S \rangle \text{ set-rel} = \text{Id-on } (\text{Pow } S)$ **unfolding** *set-rel-def*
by *auto*

25.4 Parametricity

lemmas *basic-param*[*param*] =
option.rel-transfer[*unfolded pred-bool-Id, to-set*]
All-transfer[*unfolded pred-bool-Id, to-set*]
Ex-transfer[*unfolded pred-bool-Id, to-set*]
Union-transfer[*to-set*]
image-transfer[*to-set*]
Image-parametric[*to-set*]

lemma *Sigma-param*[*param*]: $(\text{Sigma}, \text{Sigma}) \in \langle A \rangle \text{ set-rel} \rightarrow (A \rightarrow \langle B \rangle \text{ set-rel})$
 $\rightarrow \langle A \times_r B \rangle \text{ set-rel}$
unfolding *Sigma-def* **by** *parametricity*

lemma *set-filter-param*[*param*]:
 $(\text{Set.filter}, \text{Set.filter}) \in (A \rightarrow \text{bool-rel}) \rightarrow \langle A \rangle \text{ set-rel} \rightarrow \langle A \rangle \text{ set-rel}$
unfolding *Set.filter-def fun-rel-def set-rel-def* **by** *blast*
lemma *is-singleton-param*[*param*]:
assumes *bijjective A*
shows $(\text{is-singleton}, \text{is-singleton}) \in \langle A \rangle \text{ set-rel} \rightarrow \text{bool-rel}$
using *assms* **unfolding** *is-singleton-def set-rel-def bijjective-def* **by** *auto blast+*
lemma *the-elem-param*[*param*]:
assumes *is-singleton S is-singleton T*
assumes $(S, T) \in \langle A \rangle \text{ set-rel}$
shows $(\text{the-elem } S, \text{the-elem } T) \in A$
using *assms* **unfolding** *set-rel-def is-singleton-def* **by** *auto*

25.5 Lists

lemma *list-all2-list-rel-conv*[*pred-set-conv*]:
 $\text{list-all2 } (\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in \langle R \rangle \text{ list-rel})$
unfolding *list-rel-def* **by** *simp*

lemmas *list-rel-single-valued*[*iff*] = *list-rel-sv-iff*

lemmas *list-rel-simps*[*simp*] =
list.rel-eq-onp[*to-set*]
list.rel-conversep[*to-set, symmetric*]
list.rel-compp[*to-set*]

lemmas *list-rel-param*[*param*] =
list.set-transfer[*to-set*]
list.pred-transfer[*unfolded pred-bool-Id, to-set, folded pred-list-listsp*]
list.rel-transfer[*unfolded pred-bool-Id, to-set*]

lemmas *null-param*[*param*] = *null-transfer*[*unfolded pred-bool-Id, to-set*]

thm *param-set list.set-transfer*[*to-set*]

lemmas *scan-param*[*param*] = *scan.transfer*[*to-set*]

lemma *bind-param*[*param*]: (*List.bind, List.bind*) $\in \langle A \rangle$ *list-rel* $\rightarrow (A \rightarrow \langle B \rangle$
list-rel) $\rightarrow \langle B \rangle$ *list-rel*

unfolding *List.bind-def* **by** *parametricity*

lemma *set-id-param*[*param*]: (*set, id*) $\in \langle A \rangle$ *list-set-rel* $\rightarrow \langle A \rangle$ *set-rel*

unfolding *list-set-rel-def relcomp-unfold in-br-conv* **by** *auto parametricity*

25.6 Streams

definition *stream-rel* :: ('a \times 'b) *set* \Rightarrow ('a *stream* \times 'b *stream*) *set* **where**
[*to-relAPP*]: *stream-rel* *R* $\equiv \{(x, y). \text{stream-all2 } (\lambda x y. (x, y) \in R) x y\}$

lemma *stream-all2-stream-rel-conv*[*pred-set-conv*]:

stream-all2 ($\lambda x y. (x, y) \in R$) = ($\lambda x y. (x, y) \in \langle R \rangle$ *stream-rel*)

unfolding *stream-rel-def* **by** *simp*

lemmas *stream-rel-coinduct'*[*case-names stream-rel, coinduct set: stream-rel*] =
stream-rel-coinduct[*to-set*]

lemmas *stream-rel-intros* = *stream.rel-intros*[*to-set*]

lemmas *stream-rel-cases* = *stream.rel-cases*[*to-set*]

lemmas *stream-rel-inject*[*iff*] = *stream.rel-inject*[*to-set*]

lemma *stream-rel-single-valued*[*iff*]: *single-valued* ($\langle A \rangle$ *stream-rel*) \iff *single-valued*
A

proof

show *single-valued A* **if** *single-valued* ($\langle A \rangle$ *stream-rel*)

proof (*intro single-valuedI*)

fix *x y z*

assume (*x, y*) $\in A$ (*x, z*) $\in A$

then have (*sconst x, sconst y*) $\in \langle A \rangle$ *stream-rel* (*sconst x, sconst z*) $\in \langle A \rangle$

stream-rel

unfolding *stream-rel-sconst*[*to-set*] **by** *this*

then have *sconst y = sconst z* **using** *single-valuedD* **that by** *metis*

then show *y = z* **by** *simp*

qed

show *single-valued* $A \implies \text{single-valued } \langle\langle A \rangle\rangle \text{ stream-rel}$
using *stream.right-unique-rel*[*to-set*, *to-set*] **by this**
qed

lemmas *stream-rel-simps*[*simp*] =
stream.rel-eq[*unfolded pred-Id*, *THEN IdD*, *to-set*]
stream.rel-eq-onp[*to-set*]
stream.rel-conversep[*to-set*]
stream.rel-compp[*to-set*]

lemmas *stream-rel-param*[*param*] =
stream.ctr-transfer[*to-set*]
stream.sel-transfer[*to-set*]
stream.pred-transfer[*unfolded pred-bool-Id*, *to-set*, *folded pred-stream-streamsp*]
stream.rel-transfer[*unfolded pred-bool-Id*, *to-set*]
stream.map-transfer[*to-set*]
stream.set-transfer[*to-set*]
stream.case-transfer[*to-set*]
stream.corec-transfer[*unfolded pred-bool-Id*, *to-set*]

lemma *stream-Rangep-rel*: *Rangep* (*stream-all2* R) = *pred-stream* (*Rangep* R)
proof –
have 1: *pred-stream* (*Rangep* R) v **if** *stream-all2* R u v **for** u v
using *that* **by** (*coinduction arbitrary*: u v) (*auto elim*: *stream.rel-cases*)
have 2: *stream-all2* R (*smap* ($\lambda y.$ *SOME* $x.$ R x y) v) v **if** *pred-stream* (*Rangep* R) v **for** v
using *that* **by** (*coinduction arbitrary*: v) (*auto intro*: *someI*)
show *?thesis* **using** 1 2 **by** *blast*
qed

lemmas *stream-rel-domain*[*simp*] = *stream.Domainp-rel*[*to-set*]
lemmas *stream-rel-range*[*simp*] = *stream-Rangep-rel*[*to-set*]

lemma *stream-param*[*param*]:
assumes (*HOL.eq*, *HOL.eq*) $\in R \rightarrow R \rightarrow \text{bool-rel}$
shows (*HOL.eq*, *HOL.eq*) $\in \langle R \rangle \text{ stream-rel} \rightarrow \langle R \rangle \text{ stream-rel} \rightarrow \text{bool-rel}$
proof –
have (*stream-all2* *HOL.eq*, *stream-all2* *HOL.eq*) $\in \langle R \rangle \text{ stream-rel} \rightarrow \langle R \rangle \text{ stream-rel} \rightarrow \text{bool-rel}$
using *assms* **by** *parametricity*
then show *?thesis* **unfolding** *stream.rel-eq* **by this**
qed

lemmas *szip-param*[*param*] = *szip-transfer*[*to-set*]
lemmas *siterate-param*[*param*] = *siterate-transfer*[*to-set*]
lemmas *sscan-param*[*param*] = *sscan.transfer*[*to-set*]

lemma *streams-param*[*param*]: (*streams*, *streams*) $\in \langle A \rangle \text{ set-rel} \rightarrow \langle\langle A \rangle\rangle \text{ stream-rel}$

```

set-rel
proof (intro fun-relI set-relI)
  fix S T
  assume 1: (S, T) ∈ ⟨A⟩ set-rel
  obtain f where 2:  $\bigwedge x. x \in S \implies f x \in T \wedge (x, f x) \in A$ 
    using 1 unfolding set-rel-def by auto metis
  have 3:  $f \text{ ` } S \subseteq T \text{ (id, f) } \in \text{Id-on } S \rightarrow A$  using 2 by auto
  obtain g where 4:  $\bigwedge y. y \in T \implies g y \in S \wedge (g y, y) \in A$ 
    using 1 unfolding set-rel-def by auto metis
  have 5:  $g \text{ ` } T \subseteq S \text{ (g, id) } \in \text{Id-on } T \rightarrow A$  using 4 by auto
  show  $\exists v \in \text{streams } T. (u, v) \in \langle A \rangle \text{ stream-rel}$  if  $u \in \text{streams } S$  for u
  proof
    show  $\text{smap } f u \in \text{streams } T$  using smap-streams 3 that by blast
    have  $(\text{smap id } u, \text{smap } f u) \in \langle A \rangle \text{ stream-rel}$  using 3 that by parametricity
  auto
  then show  $(u, \text{smap } f u) \in \langle A \rangle \text{ stream-rel}$  by simp
  qed
  show  $\exists u \in \text{streams } S. (u, v) \in \langle A \rangle \text{ stream-rel}$  if  $v \in \text{streams } T$  for v
  proof
    show  $\text{smap } g v \in \text{streams } S$  using smap-streams 5 that by blast
    have  $(\text{smap } g v, \text{smap id } v) \in \langle A \rangle \text{ stream-rel}$  using 5 that by parametricity
  auto
  then show  $(\text{smap } g v, v) \in \langle A \rangle \text{ stream-rel}$  by simp
  qed
  qed

lemma holds-param[param]: (holds, holds) ∈ (A → bool-rel) → (⟨A⟩ stream-rel → bool-rel)
  unfolding holds.simps by parametricity
lemma HLD-param[param]:
  assumes single-valued A single-valued (A-1)
  shows (HLD, HLD) ∈ ⟨A⟩ set-rel → ⟨A⟩ stream-rel → bool-rel
  using assms unfolding HLD-def by parametricity
lemma ev-param[param]: (ev, ev) ∈ (⟨A⟩ stream-rel → bool-rel) → (⟨A⟩ stream-rel → bool-rel)
  proof safe
    fix P Q u v
    assume 1: (P, Q) ∈ ⟨A⟩ stream-rel → bool-rel (u, v) ∈ ⟨A⟩ stream-rel
    note 2 = 1[param-fo] stream-rel-param(3)[param-fo]
    show ev Q v if ev P u using that 2 by (induct arbitrary: v) (blast+)
    show ev P u if ev Q v using that 2 by (induct arbitrary: u) (blast+)
  qed
lemma alw-param[param]: (alw, alw) ∈ (⟨A⟩ stream-rel → bool-rel) → (⟨A⟩ stream-rel → bool-rel)
  proof safe
    fix P Q u v
    assume 1: (P, Q) ∈ ⟨A⟩ stream-rel → bool-rel (u, v) ∈ ⟨A⟩ stream-rel
    note 2 = 1[param-fo] stream-rel-param(3)[param-fo]
    show alw Q v if alw P u using that 2 by (coinduction arbitrary: u v) (auto,

```

```

blast)
  show  $alw P u$  if  $alw Q v$  using that 2 by (coinduction arbitrary:  $u v$ ) (auto,
blast)
qed

```

25.7 Functional Relations

```

lemma br-set-rel:  $\langle br f P \rangle set-rel = br (image f) (\lambda A. Ball A P)$ 
  using br-set-rel-alt by (auto simp: build-rel-def)

```

```

lemma br-list-rel:  $\langle br f P \rangle list-rel = br (map f) (list-all P)$ 
proof safe

```

```

  fix  $u v$ 

```

```

  show  $(u, v) \in br (map f) (list-all P)$  if  $(u, v) \in \langle br f P \rangle list-rel$ 
  using that unfolding build-rel-def by induct auto

```

```

  show  $(u, v) \in \langle br f P \rangle list-rel$  if  $(u, v) \in br (map f) (list-all P)$ 

```

```

  using that unfolding build-rel-def by (induct u arbitrary:  $v$ ) (auto)

```

```

qed

```

```

lemma br-list-set-rel:  $\langle br f P \rangle list-set-rel = br (set \circ map f) (\lambda s. list-all P s \wedge
distinct (map f s))$ 

```

```

  unfolding list-set-rel-def br-list-rel

```

```

  unfolding br-chain

```

```

  by rule

```

```

lemma br-fun-rel1:  $Id \rightarrow br f P = br (comp f) (All \circ comp P)$ 
  unfolding fun-rel-def Ball-def by (auto simp: build-rel-def)

```

```

term  $set \circ map f \circ map g \circ map h$ 

```

```

term  $set \circ sort$ 

```

```

end

```

```

theory Acceptance-Refine

```

```

imports Acceptance Refine

```

```

begin

```

```

  abbreviation (input)  $pred-rel A \equiv A \rightarrow bool-rel$ 

```

```

  abbreviation (input)  $rabin-rel A \equiv pred-rel A \times_r pred-rel A$ 

```

```

lemma rabin-param[param]: (rabin, rabin) ∈ rabin-rel A → pred-rel (⟨A⟩ stream-rel)
  unfolding rabin-def by parametricity
lemma gen-param[param]: (gen, gen) ∈ (A → pred-rel B) → (⟨A⟩ list-rel →
pred-rel B)
  unfolding gen-def by parametricity
lemma cogen-param[param]: (cogen, cogen) ∈ (A → pred-rel B) → (⟨A⟩ list-rel
→ pred-rel B)
  unfolding cogen-def by parametricity

end

```

26 Refinement for Transition Systems

```

theory Transition-System-Refine

```

```

imports

```

```

  Transition-System

```

```

  Transition-System-Extra

```

```

  ../Basic/Refine

```

```

begin

```

```

lemma path-param[param]: (transition-system.path, transition-system.path) ∈
(T → S → S) → (T → S → bool-rel) → ⟨T⟩ list-rel → S → bool-rel
proof (rule, rule)
  fix exa exb ena enb
  assume [param]: (exa, exb) ∈ T → S → S (ena, enb) ∈ T → S → bool-rel
  interpret A: transition-system exa ena by this
  interpret B: transition-system exb enb by this
  have [param]: (A.path [] p, B.path [] q) ∈ bool-rel for p q by auto
  have [param]: (A.path (a # r) p, B.path (b # s) q) ∈ bool-rel
    if (ena a p, enb b q) ∈ bool-rel (A.path r (exa a p), B.path s (exb b q)) ∈
bool-rel
    for a r p b s q
    using that by auto
  show (A.path, B.path) ∈ ⟨T⟩ list-rel → S → bool-rel
  proof (intro fun-relI)
    show (A.path r p, B.path s q) ∈ bool-rel if (r, s) ∈ ⟨T⟩ list-rel (p, q) ∈ S for
r s p q
    using that by (induct arbitrary: p q) (parametricity+)
  qed
qed
lemma run-param[param]: (transition-system.run, transition-system.run) ∈
(T → S → S) → (T → S → bool-rel) → ⟨T⟩ stream-rel → S → bool-rel
proof (rule, rule)
  fix exa exb ena enb
  assume 1: (exa, exb) ∈ T → S → S (ena, enb) ∈ T → S → bool-rel
  interpret A: transition-system exa ena by this
  interpret B: transition-system exb enb by this
  show (A.run, B.run) ∈ ⟨T⟩ stream-rel → S → bool-rel
  proof safe

```

```

    show  $B.run\ s\ q$  if  $(r, s) \in \langle T \rangle$  stream-rel  $(p, q) \in S$   $A.run\ r\ p$  for  $r\ s\ p\ q$ 
      using 1[param-fo] that by (coinduction arbitrary:  $r\ s\ p\ q$ ) (blast elim:
stream-rel-cases)
    show  $A.run\ r\ p$  if  $(r, s) \in \langle T \rangle$  stream-rel  $(p, q) \in S$   $B.run\ s\ q$  for  $r\ s\ p\ q$ 
      using 1[param-fo] that by (coinduction arbitrary:  $r\ s\ p\ q$ ) (blast elim:
stream-rel-cases)
  qed
  qed

lemma paths-param[param]:
  assumes [param]:  $(exa, exb) \in T \rightarrow S \rightarrow S$ 
  assumes (transition-system.enableds ena, transition-system.enableds enb)  $\in S$ 
 $\rightarrow \langle T \rangle$  set-rel
  shows (transition-system.paths exa ena, transition-system.paths exb enb)  $\in S$ 
 $\rightarrow \langle \langle T \rangle$  list-rel  $\rangle$  set-rel
  proof -
    note assms = assms[param-fo, unfolded transition-system.enableds-def]
    interpret A: transition-system exa ena by this
    interpret B: transition-system exb enb by this
    have 1:  $\exists s. (r, s) \in \langle T \rangle$  list-rel  $\wedge B.path\ s\ q$  if  $(p, q) \in S$   $A.path\ r\ p$  for  $p\ q\ r$ 
      using that(2, 1)
    proof (induct arbitrary: q)
      case (nil p)
        show ?case by auto
      next
        case (cons a p r)
          obtain b where 1:  $(a, b) \in T$  enb  $b\ q$  using assms(2) cons(1, 4) by (blast
elim: set-relE1)
          have 2:  $(exa\ a\ p, exb\ b\ q) \in S$  using cons(4) 1(1) by parametricity
          obtain s where 3:  $(r, s) \in \langle T \rangle$  list-rel  $B.path\ s\ (exb\ b\ q)$  using cons(3) 2
by auto
          show ?case using 1 3 by force
        qed
      have 2:  $\exists r. (r, s) \in \langle T \rangle$  list-rel  $\wedge A.path\ r\ p$  if  $(p, q) \in S$   $B.path\ s\ q$  for  $p\ q\ s$ 
        using that(2, 1)
      proof (induct arbitrary: p)
        case (nil q)
          show ?case by auto
        next
          case (cons b q s)
            obtain a where 1:  $(a, b) \in T$  ena  $a\ p$  using assms(2) cons(1, 4) by (blast
elim: set-relE2)
            have 2:  $(exa\ a\ p, exb\ b\ q) \in S$  using cons(4) 1(1) by parametricity
            obtain r where 3:  $(r, s) \in \langle T \rangle$  list-rel  $A.path\ r\ (exa\ a\ p)$  using cons(3) 2
by auto
            show ?case using 1 3 by force
          qed
        show ?thesis unfolding transition-system.paths-def set-rel-def using 1 2 by
blast

```

```

qed
lemma runs-param[param]:
  assumes (exa, exb) ∈ T → S → S
  assumes (transition-system.enableds ena, transition-system.enableds enb) ∈ S
  → ⟨T⟩ set-rel
  shows (transition-system.runs exa ena, transition-system.runs exb enb) ∈ S →
  ⟨⟨T⟩ stream-rel⟩ set-rel
  proof -
    note assms = assms[param-fo, unfolded transition-system.enableds-def]
    interpret A: transition-system exa ena by this
    interpret B: transition-system exb enb by this
    have 1: ∃ s. (r, s) ∈ ⟨T⟩ stream-rel ∧ B.run s q if (p, q) ∈ S A.run r p for p
    q r
    proof -
      define P where P ≡ λ (p, q, r). (p, q) ∈ S ∧ A.run r p
      define Q where Q ≡ λ (p :: 'b, q, r) a. (shd r, a) ∈ T ∧ enb a q
      have 1: P (p, q, r) using that unfolding P-def by auto
      have ∃ a. Q x a if P x for x
        using assms(2) that unfolding P-def Q-def by (force elim: set-relE1
        A.run.cases)
      then obtain f where 2: ∧ x. P x ⇒ Q x (f x) by metis
      define g where g ≡ λ (p, q, r). (exa (shd r) p, exb (f (p, q, r)) q, stl r)
      have 3: P (g x) if P x for x
        using assms(1) 2 that unfolding P-def Q-def g-def by (auto elim:
        A.run.cases)
      show ?thesis
      proof (intro exI conjI)
        show (r, smap f (siterate g (p, q, r))) ∈ ⟨T⟩ stream-rel
          using 1 2 3 unfolding Q-def g-def by (coinduction arbitrary: p q r)
          (fastforce)
        show B.run (smap f (siterate g (p, q, r))) q
          using 1 2 3 unfolding Q-def g-def by (coinduction arbitrary: p q r)
          (fastforce)
      qed
    qed
    have 2: ∃ r. (r, s) ∈ ⟨T⟩ stream-rel ∧ A.run r p if (p, q) ∈ S B.run s q for p
    q s
    proof -
      define P where P ≡ λ (p, q, s). (p, q) ∈ S ∧ B.run s q
      define Q where Q ≡ λ (p, q :: 'd, s) b. (b, shd s) ∈ T ∧ ena b p
      have 1: P (p, q, s) using that unfolding P-def by auto
      have ∃ a. Q x a if P x for x
        using assms(2) that unfolding P-def Q-def by (force elim: set-relE2
        B.run.cases)
      then obtain f where 2: ∧ x. P x ⇒ Q x (f x) by metis
      define g where g ≡ λ (p, q, s). (exa (f (p, q, s)) p, exb (shd s) q, stl s)
      have 3: P (g x) if P x for x
        using assms(1) 2 that unfolding P-def Q-def g-def by (auto elim:
        B.run.cases)

```

```

show ?thesis
proof (intro exI conjI)
  show (smap f (siterate g (p, q, s)), s) ∈ ⟨T⟩ stream-rel
    using 1 2 3 unfolding Q-def g-def by (coinduction arbitrary: p q s)
(fastforce)
  show A.run (smap f (siterate g (p, q, s))) p
    using 1 2 3 unfolding Q-def g-def by (coinduction arbitrary: p q s)
(fastforce)
  qed
qed
show ?thesis unfolding transition-system.runs-def set-rel-def using 1 2 by
force
qed

end

```

27 Relations on Deterministic Rabin Automata

theory DRA-Refine

imports

DRA

../Basic/Acceptance-Refine

../Transition-Systems/Transition-System-Refine

begin

definition dra-rel :: ('label₁ × 'label₂) set ⇒ ('state₁ × 'state₂) set ⇒
 ((('label₁, 'state₁) dra × ('label₂, 'state₂) dra) set **where**
 [to-relAPP]: dra-rel L S ≡ {(A₁, A₂).
 (alphabet A₁, alphabet A₂) ∈ ⟨L⟩ set-rel ∧
 (initial A₁, initial A₂) ∈ S ∧
 (transition A₁, transition A₂) ∈ L → S → S ∧
 (condition A₁, condition A₂) ∈ ⟨rabin-rel S⟩ list-rel}

lemma dra-param[param]:

(dra, dra) ∈ ⟨L⟩ set-rel → S → (L → S → S) → ⟨rabin-rel S⟩ list-rel →
 ⟨L, S⟩ dra-rel
 (alphabet, alphabet) ∈ ⟨L, S⟩ dra-rel → ⟨L⟩ set-rel
 (initial, initial) ∈ ⟨L, S⟩ dra-rel → S
 (transition, transition) ∈ ⟨L, S⟩ dra-rel → L → S → S
 (condition, condition) ∈ ⟨L, S⟩ dra-rel → ⟨rabin-rel S⟩ list-rel
unfolding dra-rel-def fun-rel-def **by** auto

lemma dra-rel-id[simp]: ⟨Id, Id⟩ dra-rel = Id **unfolding** dra-rel-def **using**
 dra.expand **by** auto

lemma dra-rel-comp[trans]:

assumes [param]: (A, B) ∈ ⟨L₁, S₁⟩ dra-rel (B, C) ∈ ⟨L₂, S₂⟩ dra-rel
shows (A, C) ∈ ⟨L₁ O L₂, S₁ O S₂⟩ dra-rel

proof –

have (condition A, condition B) ∈ ⟨rabin-rel S₁⟩ list-rel **by** parametricity

also have $(\text{condition } B, \text{condition } C) \in \langle \text{rabin-rel } S_2 \rangle \text{ list-rel}$ **by** *parametricity*
finally have $1: (\text{condition } A, \text{condition } C) \in \langle \text{rabin-rel } S_1 \text{ } O \text{ rabin-rel } S_2 \rangle$
list-rel **by** *simp*
have $2: \text{rabin-rel } S_1 \text{ } O \text{ rabin-rel } S_2 \subseteq \text{rabin-rel } (S_1 \text{ } O \text{ } S_2)$ **by** (*force simp: fun-rel-def*)
have $3: (\text{condition } A, \text{condition } C) \in \langle \text{rabin-rel } (S_1 \text{ } O \text{ } S_2) \rangle \text{ list-rel}$ **using** $1 \ 2$
list-rel-mono **by** *blast*
have $(\text{transition } A, \text{transition } B) \in L_1 \rightarrow S_1 \rightarrow S_1$ **by** *parametricity*
also have $(\text{transition } B, \text{transition } C) \in L_2 \rightarrow S_2 \rightarrow S_2$ **by** *parametricity*
finally have $4: (\text{transition } A, \text{transition } C) \in L_1 \text{ } O \text{ } L_2 \rightarrow S_1 \text{ } O \text{ } S_2 \rightarrow S_1 \text{ } O \text{ } S_2$ **by** *this*
show *?thesis*
unfolding *dra-rel-def mem-Collect-eq prod.case set-rel-compp*
using $3 \ 4$
using *dra-param(2 - 5)[THEN fun-relD, OF assms(1)]*
using *dra-param(2 - 5)[THEN fun-relD, OF assms(2)]*
by *auto*
qed
lemma *dra-rel-converse[simp]*: $(\langle L, S \rangle \text{ dra-rel})^{-1} = \langle L^{-1}, S^{-1} \rangle \text{ dra-rel}$
proof –
have $1: \langle L \rangle \text{ set-rel} = (\langle L^{-1} \rangle \text{ set-rel})^{-1}$ **by** *simp*
have $2: \langle S \rangle \text{ set-rel} = (\langle S^{-1} \rangle \text{ set-rel})^{-1}$ **by** *simp*
have $3: L \rightarrow S \rightarrow S = (L^{-1} \rightarrow S^{-1} \rightarrow S^{-1})^{-1}$ **by** *simp*
have $4: \langle \text{rabin-rel } S \rangle \text{ list-rel} = (\langle \text{rabin-rel } (S^{-1}) \rangle \text{ list-rel})^{-1}$ **by** *simp*
show *?thesis* **unfolding** *dra-rel-def* **unfolding** 3 **unfolding** $1 \ 2 \ 4$ **by** *fastforce*
qed

lemma *dra-rel-eq*: $(A, A) \in \langle \text{Id-on } (\text{alphabet } A), \text{Id-on } (\text{nodes } A) \rangle \text{ dra-rel}$
unfolding *dra-rel-def prod-rel-def* **using** *list-all2-same[to-set]* **by** *auto*

lemma *enableds-param[param]*: $(\text{dra.enableds}, \text{dra.enableds}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow \langle L \rangle \text{ set-rel}$
unfolding *dra.enableds-def Collect-mem-eq* **by** *parametricity*
lemma *paths-param[param]*: $(\text{dra.paths}, \text{dra.paths}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow (\langle L \rangle \text{ list-rel}) \text{ set-rel}$
using *enableds-param[param-fo]* **by** *parametricity*
lemma *runs-param[param]*: $(\text{dra.runs}, \text{dra.runs}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow (\langle L \rangle \text{ stream-rel}) \text{ set-rel}$
using *enableds-param[param-fo]* **by** *parametricity*

lemma *reachable-param[param]*: $(\text{reachable}, \text{reachable}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow \langle S \rangle \text{ set-rel}$
proof –
have $1: \text{reachable } A \ p = (\lambda w. \text{target } A \ w \ p) \text{ `dra.paths } A \ p$ **for** $A :: ('label, 'state) \text{ dra}$ **and** p
unfolding *dra.reachable-alt-def dra.paths-def* **by** *auto*
show *?thesis* **unfolding** 1 **using** *enableds-param[param-fo]* **by** *parametricity*
qed
lemma *nodes-param[param]*: $(\text{nodes}, \text{nodes}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle S \rangle \text{ set-rel}$

```

proof –
  have 1: nodes A = reachable A (initial A) for A :: ('label, 'state) dra
    unfolding dra.nodes-alt-def by simp
  show ?thesis unfolding 1 by parametricity
qed

```

```

lemma language-param[param]: (language, language) ∈ ⟨L, S⟩ dra-rel → ⟨⟨L⟩
stream-rel⟩ set-rel

```

```

proof –
  have 1: language A = (⋃ w ∈ dra.runs A (initial A).
    if cogen rabin (condition A) (initial A ## trace A w (initial A)) then {w}
  else {})
  for A :: ('label, 'state) dra
  unfolding dra.language-def dra.runs-def by auto
  show ?thesis unfolding 1 using enableds-param[param-fo] by parametricity
qed

```

end

28 Implementation

```

theory Implement
imports
  HOL-Library.Monad-Syntax
  Collections.Refine-Dflt
  Refine
begin

```

28.1 Syntax

```

no-syntax -do-let :: [pttrn, 'a] ⇒ do-bind ((2let - =/ -) [1000, 13] 13)
syntax -do-let :: [pttrn, 'a] ⇒ do-bind ((2let - =/ -) 13)

```

28.2 Monadic Refinement

```

lemmas [refine] = plain-nres-relI

```

```

lemma vcg0:
  assumes (f, g) ∈ ⟨Id⟩ nres-rel
  shows g ≤ h ⇒ f ≤ h
  using order-trans nres-relD[OF assms[param-fo, OF], THEN refine-IdD] by
  this

```

```

lemma vcg1:
  assumes (f, g) ∈ Id → ⟨Id⟩ nres-rel
  shows g x ≤ h x ⇒ f x ≤ h x
  using order-trans nres-relD[OF assms[param-fo, OF Id], THEN refine-IdD]
by this

```

```

lemma vcg2:
  assumes (f, g) ∈ Id → Id → ⟨Id⟩ nres-rel

```

shows $g x y \leq h x y \implies f x y \leq h x y$
using *order-trans nres-relD*[*OF assms*[*param-fo*, *OF Idi IdI*], *THEN refine-IdD*]
by this

lemma *RETURN-nres-relD*:

assumes $(RETURN\ x, RETURN\ y) \in \langle A \rangle\ nres-rel$
shows $(x, y) \in A$
using *assms unfolding nres-rel-def by simp*

lemma *FOREACH-rule-insert*:

assumes *finite S*
assumes $I\ \{\}\ s$
assumes $\bigwedge s. I\ S\ s \implies P\ s$
assumes $\bigwedge T\ x\ s. T \subseteq S \implies I\ T\ s \implies x \in S \implies x \notin T \implies f\ x\ s \leq SPEC$
(I (insert x T))
shows $FOREACH\ S\ f\ s \leq SPEC\ P$
proof (*rule FOREACH-rule*[**where** $I = \lambda T\ s. I\ (S - T)\ s$])
show *finite S* **using** *assms(1) by this*
show $I\ (S - S)\ s$ **using** *assms(2) by simp*
show $P\ s$ **if** $I\ (S - \{\})\ s$ **for** s **using** *assms(3) that by simp*
next
fix $x\ T\ s$
assume $1: x \in T\ T \subseteq S\ I\ (S - T)\ s$
have $f\ x\ s \leq SPEC\ (I\ (insert\ x\ (S - T)))$ **using** *assms(4) 1 by blast*
also have $insert\ x\ (S - T) = S - (T - \{x\})$ **using** $1(1, 2)$ **by** (*simp add: it-step-insert-iff*)
finally show $f\ x\ s \leq SPEC\ (I\ (S - (T - \{x\})))$ **by this**

qed

lemma *FOREACH-rule-map*:

assumes *finite (dom g)*
assumes $I\ Map.empty\ s$
assumes $\bigwedge s. I\ g\ s \implies P\ s$
assumes $\bigwedge h\ k\ v\ s. h \subseteq_m g \implies I\ h\ s \implies g\ k = Some\ v \implies k \notin dom\ h \implies$
 $f\ (k, v)\ s \leq SPEC\ (I\ (h\ (k \mapsto v)))$
shows $FOREACH\ (map-to-set\ g)\ f\ s \leq SPEC\ P$
proof (*rule FOREACH-rule-insert*[**where** $I = \lambda H\ s. I\ (set-to-map\ H)\ s$])
show *finite (map-to-set g)* **unfolding** *finite-map-to-set* **using** *assms(1) by this*
show $I\ (set-to-map\ \{\})\ s$ **using** *assms(2) by simp*
show $P\ s$ **if** $I\ (set-to-map\ (map-to-set\ g))\ s$ **for** s
using *assms(3) that unfolding map-to-set-inverse by this*

next

fix $H\ x\ s$
assume $1: H \subseteq map-to-set\ g\ I\ (set-to-map\ H)\ s\ x \in map-to-set\ g\ x \notin H$
obtain $k\ v$ **where** $2: x = (k, v)$ **by force**
have $3: inj-on\ fst\ H$ **using** *inj-on-fst-map-to-set inj-on-subset 1(1) by blast*
have $f\ x\ s = f\ (k, v)\ s$ **unfolding** 2 **by rule**
also have $\dots \leq SPEC\ (I\ ((set-to-map\ H)\ (k \mapsto v)))$
proof (*rule assms(4)*)
show $set-to-map\ H \subseteq_m g$

using 1(1) 3
by (*metis inj-on-fst-map-to-set map-leI map-to-set-inverse set-to-map-simp subset-eq*)
show I (*set-to-map H*) s **using** 1(2) **by this**
show $g k = \text{Some } v$ **using** 1(3) **unfolding** 2 *map-to-set-def* **by simp**
show $k \notin \text{dom } (\text{set-to-map } H)$
using 1(1, 3, 4) **unfolding** 2 *set-to-map-dom*
by (*metis fst-conv inj-on-fst-map-to-set inj-on-image-mem-iff*)
qed
also have (*set-to-map H*) ($k \mapsto v$) = (*set-to-map H*) (*fst x* \mapsto *snd x*) **unfolding**
2 **by simp**
also have $\dots = \text{set-to-map } (\text{insert } x H)$
using 1(1, 3, 4) **by** (*metis inj-on-fst-map-to-set inj-on-image-mem-iff set-to-map-insert*)
finally show $f x s \leq \text{SPEC } (I (\text{set-to-map } (\text{insert } x H)))$ **by this**
qed
lemma *FOREACH-rule-insert-eq*:
assumes *finite S*
assumes $X \{\} = s$
assumes $X S = t$
assumes $\bigwedge T x. T \subseteq S \implies x \in S \implies x \notin T \implies f x (X T) \leq \text{SPEC } (\text{HOL.eq } (X (\text{insert } x T)))$
shows *FOREACH S f s* $\leq \text{SPEC } (\text{HOL.eq } t)$
by (*rule FOREACH-rule-insert[where I = HOL.eq o X]*) (*use assms in auto*)
lemma *FOREACH-rule-map-eq*:
assumes *finite (dom g)*
assumes $X \text{Map.empty} = s$
assumes $X g = t$
assumes $\bigwedge h k v. h \subseteq_m g \implies g k = \text{Some } v \implies k \notin \text{dom } h \implies$
 $f (k, v) (X h) \leq \text{SPEC } (\text{HOL.eq } (X (h (k \mapsto v))))$
shows *FOREACH (map-to-set g) f s* $\leq \text{SPEC } (\text{HOL.eq } t)$
by (*rule FOREACH-rule-map[where I = HOL.eq o X]*) (*use assms in auto*)

lemma *FOREACH-rule-map-map*: (*FOREACH (map-to-set m)* ($\lambda (k, v). F k (f k v)$),
FOREACH (map-to-set ($\lambda k. \text{map-option } (f k) (m k)$)) ($\lambda (k, v). F k v$)) $\in \text{Id}$
 $\rightarrow \langle \text{Id} \rangle \text{nres-rel}$
proof *refine-vcg*
show *inj-on* ($\lambda (k, v). (k, f k v)$) (*map-to-set m*)
unfolding *map-to-set-def* **by rule auto**
show *map-to-set* ($\lambda k. \text{map-option } (f k) (m k)$) = ($\lambda (k, v). (k, f k v)$) ‘
map-to-set m
unfolding *map-to-set-def* **by auto**
qed *auto*

28.3 Implementations for Sets Represented by Lists

lemma *list-set-rel-Id-on[simp]*: $\langle \text{Id-on } A \rangle \text{list-set-rel} = \langle \text{Id} \rangle \text{list-set-rel} \cap \text{UNIV} \times \text{Pow } A$
unfolding *list-set-rel-def relcomp-unfold in-br-conv* **by auto**

```

lemma list-set-card[param]: (length, card) ∈ ⟨A⟩ list-set-rel → nat-rel
  unfolding list-set-rel-def relcomp-unfold in-br-conv
  by (auto simp: distinct-card list-rel-imp-same-length)
lemma list-set-insert[param]:
  assumes y ∉ Y
  assumes (x, y) ∈ A (xs, Y) ∈ ⟨A⟩ list-set-rel
  shows (x # xs, insert y Y) ∈ ⟨A⟩ list-set-rel
  using assms unfolding list-set-rel-def relcomp-unfold in-br-conv
  by (auto) (metis refine-list(2)[param-fo] distinct.simps(2) list.simps(15))
lemma list-set-union[param]:
  assumes X ∩ Y = {}
  assumes (xs, X) ∈ ⟨A⟩ list-set-rel (ys, Y) ∈ ⟨A⟩ list-set-rel
  shows (xs @ ys, X ∪ Y) ∈ ⟨A⟩ list-set-rel
  using assms unfolding list-set-rel-def relcomp-unfold in-br-conv
  by (auto) (meson param-append[param-fo] distinct-append set-union-code)
lemma list-set-Union[param]:
  assumes ∧ X Y. X ∈ S ⇒ Y ∈ S ⇒ X ≠ Y ⇒ X ∩ Y = {}
  assumes (xs, S) ∈ ⟨⟨A⟩ list-set-rel⟩ list-set-rel
  shows (concat xs, Union S) ∈ ⟨A⟩ list-set-rel
proof –
  note distinct-map[iff]
  obtain zs where 1: (xs, zs) ∈ ⟨⟨A⟩ list-set-rel⟩ list-rel S = set zs distinct zs
    using assms(2) unfolding list-set-rel-def relcomp-unfold in-br-conv by auto
  obtain ys where 2: (xs, ys) ∈ ⟨⟨A⟩ list-rel⟩ list-rel zs = map set ys list-all
    distinct ys
    using 1(1)
    unfolding list-set-rel-def list-rel-compp
    unfolding relcomp-unfold mem-Collect-eq prod.case
    unfolding br-list-rel in-br-conv
    by auto
  have 20: set a ∈ S set b ∈ S set a ≠ set b if a ∈ set ys b ∈ set ys a ≠ b for a b
    using 1(3) that unfolding 1(2) 2(2) by (auto dest: inj-onD)
  have 3: set a ∩ set b = {} if a ∈ set ys b ∈ set ys a ≠ b for a b
    using assms(1) 20 that by auto
  have 4: Union S = set (concat ys) unfolding 1(2) 2(2) by simp
  have 5: distinct (concat ys)
    using 1(3) 2(2, 3) 3 unfolding list.pred-set by (blast intro: distinct-concat)
  have 6: (concat xs, concat ys) ∈ ⟨A⟩ list-rel using 2(1) by parametricity
  show ?thesis unfolding list-set-rel-def relcomp-unfold in-br-conv using 4 5 6
by blast
qed
lemma list-set-image[param]:
  assumes inj-on g S
  assumes (f, g) ∈ A → B (xs, S) ∈ ⟨A⟩ list-set-rel
  shows (map f xs, g ‘ S) ∈ ⟨B⟩ list-set-rel
  using assms unfolding list-set-rel-def relcomp-unfold in-br-conv
  using param-map[param-fo] distinct-map by fastforce
lemma list-set-bind[param]:

```

assumes $\bigwedge x y. x \in S \implies y \in S \implies x \neq y \implies g x \cap g y = \{\}$
assumes $(xs, S) \in \langle A \rangle \text{ list-set-rel } (f, g) \in A \rightarrow \langle B \rangle \text{ list-set-rel}$
shows $(xs \ggg f, S \ggg g) \in \langle B \rangle \text{ list-set-rel}$
proof –
note $[param] = \text{list-set-autoref-filter list-set-autoref-isEmpty}$
let $?xs = \text{filter } (Not \circ \text{is-Nil} \circ f) xs$
let $?S = \text{op-set-filter } (Not \circ \text{op-set-isEmpty} \circ g) S$
have $1: \text{inj-on } g \text{ ?S using } \text{assms}(1) \text{ by } (\text{fastforce intro: inj-onI})$
have $xs \ggg f = \text{concat } (\text{map } f \text{ ?xs}) \text{ by } (\text{induct } xs) (\text{auto split: list.split})$
also have $(\dots, \bigcup (g \text{ ' ?S})) \in \langle B \rangle \text{ list-set-rel using } \text{assms } 1 \text{ by parametricity}$
auto
also have $\bigcup (g \text{ ' ?S}) = S \ggg g \text{ by auto auto}$
finally show $?thesis \text{ by this}$
qed

28.4 Autoref Setup

lemma *dflt-ahm-rel-finite-nat: finite-map-rel* ($\langle nat\text{-rel}, V \rangle \text{ dflt-ahm-rel}$) **by** *tagged-solver*

context

begin

interpretation *autoref-syn* **by** *this*

lemma $[\text{autoref-op-pat}]: (Some \circ f) \mid ' X \equiv OP (\lambda f X. (Some \circ f) \mid ' X) f X$
by *simp*

lemma $[\text{autoref-op-pat}]: \bigcup (m \text{ ' } S) \equiv OP (\lambda S m. \bigcup (m \text{ ' } S)) S m \text{ by } \text{simp}$

definition *gen-UNION* **where**

gen-UNION tol emp un X f $\equiv \text{fold } (un \circ f) (tol X) emp$

lemma *gen-UNION* $[\text{autoref-rules-raw}]$:

assumes *PRIO-TAG-GEN-ALGO*

assumes *to-list: SIDE-GEN-ALGO* (*is-set-to-list* $A \text{ Rs1 tol}$)

assumes *empty: GEN-OP emp* $\{\} (\langle B \rangle \text{ Rs3})$

assumes *union: GEN-OP un union* ($\langle B \rangle \text{ Rs2} \rightarrow \langle B \rangle \text{ Rs3} \rightarrow \langle B \rangle \text{ Rs3}$)

shows (*gen-UNION tol emp un*, $\lambda A f. \bigcup (f \text{ ' } A) \in \langle A \rangle \text{ Rs1} \rightarrow (A \rightarrow \langle B \rangle \text{ Rs2}) \rightarrow \langle B \rangle \text{ Rs3}$)

proof (*intro fun-relI*)

note $[\text{unfolded autoref-tag-defs, param}] = \text{empty union}$

fix $f g T S$

assume $1[param]: (T, S) \in \langle A \rangle \text{ Rs1 } (g, f) \in A \rightarrow \langle B \rangle \text{ Rs2}$

obtain tsl' **where**

$[param]: (tol T, tsl') \in \langle A \rangle \text{ list-rel}$

and IT' : *RETURN* $tsl' \leq \text{it-to-sorted-list } (\lambda - . \text{True}) S$

using *to-list* $[\text{unfolded autoref-tag-defs is-set-to-list-def}] 1(1)$

by (*rule is-set-to-sorted-listE*)

from IT' **have** $10: S = \text{set } tsl' \text{ distinct } tsl' \text{ unfolding } \text{it-to-sorted-list-def}$

by *simp-all*

have *gen-UNION tol emp un T g* $= \text{fold } (un \circ g) (tol T) emp \text{ unfolding}$

gen-UNION-def **by rule**

also have $(\dots, \text{fold } (\text{union} \circ f) \text{ } \text{tsl}' \ \{\}) \in \langle B \rangle \text{ Rs3}$ **by parametricity**
also have $\text{fold } (\text{union} \circ f) \text{ } \text{tsl}' \ X = \bigcup (f \ ' \ S) \cup X$ **for** X
unfolding $10(1)$ **by** $(\text{induct } \text{tsl}' \ \text{arbitrary: } X) \ (\text{auto})$
also have $\bigcup (f \ ' \ S) \cup \{\} = \bigcup (f \ ' \ S)$ **by simp**
finally show $(\text{gen-UNION } \text{tol } \text{emp } \text{un } T \ g, \bigcup (f \ ' \ S)) \in \langle B \rangle \text{ Rs3}$ **by this**
qed

definition *gen-Image* **where**

gen-Image $\text{tol1 } \text{mem2 } \text{emp3 } \text{ins3 } X \ Y \equiv \text{fold}$
 $(\lambda (a, b). \text{if } \text{mem2 } a \ Y \text{ then } \text{ins3 } b \ \text{else } \text{id}) (\text{tol1 } X) \ \text{emp3}$

lemma *gen-Image*[*autoref-rules*]:

assumes *PRIO-TAG-GEN-ALGO*
assumes *to-list: SIDE-GEN-ALGO* $(\text{is-set-to-list } (A \times_r B) \ \text{Rs1 } \text{tol1})$
assumes *member: GEN-OP* $\text{mem2 } (\in) (A \rightarrow \langle A \rangle \ \text{Rs2} \rightarrow \text{bool-rel})$
assumes *empty: GEN-OP* $\text{emp3 } \{\} (\langle B \rangle \ \text{Rs3})$
assumes *insert: GEN-OP* $\text{ins3 } \text{Set.insert } (B \rightarrow \langle B \rangle \ \text{Rs3} \rightarrow \langle B \rangle \ \text{Rs3})$
shows $(\text{gen-Image } \text{tol1 } \text{mem2 } \text{emp3 } \text{ins3}, \text{Image}) \in \langle A \times_r B \rangle \ \text{Rs1} \rightarrow \langle A \rangle$

$\text{Rs2} \rightarrow \langle B \rangle \ \text{Rs3}$

proof (*intro fun-relI*)

note $[\text{unfolded } \text{autoref-tag-defs}, \text{param}] = \text{member } \text{empty } \text{insert}$

fix $T \ S \ X \ Y$

assume $1[\text{param}]: (T, S) \in \langle A \times_r B \rangle \ \text{Rs1} \ (Y, X) \in \langle A \rangle \ \text{Rs2}$

obtain tsl' **where**

$[\text{param}]: (\text{tol1 } T, \text{tsl}') \in \langle A \times_r B \rangle \ \text{list-rel}$

and $IT': \text{RETURN } \text{tsl}' \leq \text{it-to-sorted-list } (\lambda \ -. \ \text{True}) \ S$

using *to-list*[*unfolded autoref-tag-defs is-set-to-list-def*] $1(1)$

by (*rule is-set-to-sorted-listE*)

from IT' **have** $10: S = \text{set } \text{tsl}' \ \text{distinct } \text{tsl}'$ **unfolding** *it-to-sorted-list-def*

by *simp-all*

have *gen-Image* $\text{tol1 } \text{mem2 } \text{emp3 } \text{ins3 } T \ Y =$

$\text{fold } (\lambda (a, b). \text{if } \text{mem2 } a \ Y \text{ then } \text{ins3 } b \ \text{else } \text{id}) (\text{tol1 } T) \ \text{emp3}$

unfolding *gen-Image-def* **by rule**

also have $(\dots, \text{fold } (\lambda (a, b). \text{if } a \in X \text{ then } \text{Set.insert } b \ \text{else } \text{id}) \ \text{tsl}' \ \{\}) \in$
 $\langle B \rangle \ \text{Rs3}$

by parametricity

also have $\text{fold } (\lambda (a, b). \text{if } a \in X \text{ then } \text{Set.insert } b \ \text{else } \text{id}) \ \text{tsl}' \ M = S \ \text{`` } X$
 $\cup M$ **for** M

unfolding $10(1)$ **by** $(\text{induct } \text{tsl}' \ \text{arbitrary: } M) \ (\text{auto } \text{split: } \text{prod.splits})$

also have $S \ \text{`` } X \cup \{\} = S \ \text{`` } X$ **by simp**

finally show $(\text{gen-Image } \text{tol1 } \text{mem2 } \text{emp3 } \text{ins3 } T \ Y, S \ \text{`` } X) \in \langle B \rangle \ \text{Rs3}$ **by**

this

qed

lemma *list-set-union-autoref*[*autoref-rules*]:

assumes *PRIO-TAG-OPTIMIZATION*

assumes *SIDE-PRECOND-OPT* $(a' \cap b' = \{\})$

assumes $(a, a') \in \langle R \rangle \ \text{list-set-rel}$

assumes $(b, b') \in \langle R \rangle \text{ list-set-rel}$
shows $(a @ b,$
 $(OP \text{ union} :: \langle R \rangle \text{ list-set-rel} \rightarrow \langle R \rangle \text{ list-set-rel} \rightarrow \langle R \rangle \text{ list-set-rel}) \$ a' \$ b')$
 \in
 $\langle R \rangle \text{ list-set-rel}$
using *assms list-set-union unfolding autoref-tag-defs by blast*
lemma *list-set-image-autoref[autoref-rules]:*
assumes *PRIO-TAG-OPTIMIZATION*
assumes *INJ: SIDE-PRECOND-OPT (inj-on f s)*
assumes $\bigwedge xi x. (xi, x) \in Ra \implies x \in s \implies (fi xi, f \$ x) \in Rb$
assumes *LP: $(l, s) \in \langle Ra \rangle \text{list-set-rel}$*
shows $(map fi l,$
 $(OP \text{ image} :: (Ra \rightarrow Rb) \rightarrow \langle Ra \rangle \text{ list-set-rel} \rightarrow \langle Rb \rangle \text{ list-set-rel}) \$ f \$ s) \in$
 $\langle Rb \rangle \text{ list-set-rel}$
proof –
from *LP* **obtain** l' **where** $1: (l, l') \in \langle Ra \rangle \text{list-rel}$ **and** $L'S: (l', s) \in br \text{ set distinct}$
unfolding *list-set-rel-def* **by** *auto*
have $2: s = set l'$ **using** $L'S$ **unfolding** *in-br-conv* **by** *auto*
have $(map fi l, map f l') \in \langle Rb \rangle \text{list-rel}$
using $1 L'S \text{ assms } (3)$ **unfolding** 2 in-br-conv **by** *induct auto*
also from *INJ* $L'S$ **have** $(map f l', f's) \in br \text{ set distinct}$
by *(induct l' arbitrary: s) (auto simp: br-def dest: injD)*
finally *(relcompI)* **show** *?thesis* **unfolding** *autoref-tag-defs list-set-rel-def* **by**
this
qed
lemma *list-set-UNION-autoref[autoref-rules]:*
assumes *PRIO-TAG-OPTIMIZATION*
assumes *SIDE-PRECOND-OPT $(\forall x \in S. \forall y \in S. x \neq y \longrightarrow g x \cap g y =$*
 $\{\})$
assumes $(xs, S) \in \langle A \rangle \text{ list-set-rel}$ $(f, g) \in A \rightarrow \langle B \rangle \text{ list-set-rel}$
shows $(xs \ggg f,$
 $(OP (\lambda A f. \bigcup (f ' A)) :: \langle A \rangle \text{ list-set-rel} \rightarrow (A \rightarrow \langle B \rangle \text{ list-set-rel}) \rightarrow \langle B \rangle$
 $\text{list-set-rel}) \$ S \$ g) \in$
 $\langle B \rangle \text{ list-set-rel}$
using *assms list-set-bind unfolding bind-UNION autoref-tag-defs by metis*

definition *gen-equals* **where**
 $gen-equals \text{ ball } lu \text{ eq } f \ g \equiv$
 $\text{ball } f (\lambda (k, v). \text{rel-option eq } (lu \ k \ g) (\text{Some } v)) \wedge$
 $\text{ball } g (\lambda (k, v). \text{rel-option eq } (lu \ k \ f) (\text{Some } v))$

lemma *gen-equals[autoref-rules]:*
assumes *PRIO-TAG-GEN-ALGO*
assumes *BALL: GEN-OP ball op-map-ball $(\langle Rk, Rv \rangle Rm \rightarrow (Rk \times_r Rv \rightarrow$*
 $\text{bool-rel}) \rightarrow \text{bool-rel}$
assumes *LU: GEN-OP lu op-map-lookup $(Rk \rightarrow \langle Rk, Rv \rangle Rm \rightarrow \langle Rv \rangle$*
 $\text{option-rel})$
assumes *EQ: GEN-OP eq HOL.eq $(Rv \rightarrow Rv \rightarrow \text{bool-rel})$*
shows $(gen-equals \text{ ball } lu \text{ eq}, \text{HOL.eq}) \in \langle Rk, Rv \rangle Rm \rightarrow \langle Rk, Rv \rangle Rm \rightarrow$

bool-rel

proof (*intro fun-relI*)

note [*unfolded autoref-tag-defs, param*] = *BALL LU EQ*

fix *fi f gi g*

assume [*param*]: $(fi, f) \in \langle Rk, Rv \rangle Rm$ $(gi, g) \in \langle Rk, Rv \rangle Rm$

have *gen-equals ball lu eq fi gi* \longleftrightarrow *ball fi* $(\lambda (k, v). \text{rel-option eq } (lu\ k\ gi))$
(Some v)) \wedge
ball gi $(\lambda (k, v). \text{rel-option eq } (lu\ k\ fi))$ *(Some v)*)

unfolding *gen-equals-def* **by** *rule*

also have *ball fi* $(\lambda (k, v). \text{rel-option eq } (lu\ k\ gi))$ *(Some v)*) \longleftrightarrow
op-map-ball f $(\lambda (k, v). \text{rel-option HOL.eq } (op\text{-map-lookup } k\ g))$ *(Some v)*)
by (*rule IdD*) (*parametricity*)

also have *ball gi* $(\lambda (k, v). \text{rel-option eq } (lu\ k\ fi))$ *(Some v)*) \longleftrightarrow
op-map-ball g $(\lambda (k, v). \text{rel-option HOL.eq } (op\text{-map-lookup } k\ f))$ *(Some v)*)
by (*rule IdD*) (*parametricity*)

also have *op-map-ball f* $(\lambda (k, v). \text{rel-option HOL.eq } (op\text{-map-lookup } k\ g))$
(Some v)) \wedge
op-map-ball g $(\lambda (k, v). \text{rel-option HOL.eq } (op\text{-map-lookup } k\ f))$ *(Some v)*)
 \longleftrightarrow
 $(\forall a\ b. f\ a = \text{Some } b \longleftrightarrow g\ a = \text{Some } b)$

unfolding *op-map-ball-def map-to-set-def option.rel-eq op-map-lookup-def*

by *auto*

also have $(\forall a\ b. f\ a = \text{Some } b \longleftrightarrow g\ a = \text{Some } b) \longleftrightarrow f = g$ **using**
option.exhaust ext **by** *metis*

finally show (*gen-equals ball lu eq fi gi, f = g*) \in *bool-rel* **by** *simp*

qed

definition *op-set-enumerate* :: '*a set* \Rightarrow (*a* \rightarrow *nat*) *nres* **where**
op-set-enumerate S \equiv *SPEC* $(\lambda f. \text{dom } f = S \wedge \text{inj-on } f\ S)$

lemma [*autoref-itype*]: *op-set-enumerate* ::_{*i*} $\langle A \rangle_i$ *i-set* \rightarrow_i $\langle \langle A, i\text{-nat} \rangle_i$ *i-map*_{*i*}
i-nres **by** *simp*

lemma [*autoref-hom*]: *CONSTRAINT* *op-set-enumerate* $(\langle A \rangle\ Rs \rightarrow \langle \langle A, \text{nat-rel} \rangle$
Rm) *nres-rel*) **by** *simp*

definition *gen-enumerate* **where**
gen-enumerate tol upd emp S \equiv *snd* $(\text{fold } (\lambda x\ (k, m). (\text{Suc } k, \text{upd } x\ k\ m))$
(tol S) $(0, \text{emp}))$

lemma *gen-enumerate[autoref-rules-raw]*:

assumes *PRIO-TAG-GEN-ALGO*

assumes *to-list: SIDE-GEN-ALGO* (*is-set-to-list A Rs tol*)

assumes *empty: GEN-OP emp op-map-empty* $(\langle A, \text{nat-rel} \rangle\ Rm)$

assumes *update: GEN-OP upd op-map-update* $(A \rightarrow \text{nat-rel} \rightarrow \langle A, \text{nat-rel} \rangle$
Rm $\rightarrow \langle A, \text{nat-rel} \rangle\ Rm)$

shows $(\lambda S. \text{RETURN } (\text{gen-enumerate } tol\ \text{upd } \text{emp } S), \text{op-set-enumerate}) \in$
 $\langle A \rangle\ Rs \rightarrow \langle \langle A, \text{nat-rel} \rangle\ Rm \rangle$ *nres-rel*

proof
note $[unfolding\ autoref-tag-defs, param] = empty\ update$
fix $T\ S$
assume $1: (T, S) \in \langle A \rangle R_s$
obtain tsl' **where**
 $[param]: (tol\ T, tsl') \in \langle A \rangle list-rel$
and $IT': RETURN\ tsl' \leq it-to-sorted-list\ (\lambda -. True)\ S$
using $to-list[unfolding\ autoref-tag-defs\ is-set-to-list-def]\ 1$
by $(rule\ is-set-to-sorted-listE)$
from IT' **have** $10: S = set\ tsl'\ distinct\ tsl'$ **unfolding** $it-to-sorted-list-def$
by $simp-all$
have $2: dom\ (snd\ (fold\ (\lambda\ x\ (k, m). (Suc\ k, m\ (x \mapsto k)))\ tsl'\ (k, m))) = dom\ m \cup set\ tsl'$
for $k\ m$ **by** $(induct\ tsl'\ arbitrary: k\ m)\ (auto)$
have $3: inj-on\ (snd\ (fold\ (\lambda\ x\ (k, m). (Suc\ k, m\ (x \mapsto k)))\ tsl'\ (0, Map.empty)))\ (set\ tsl')$
using $10(2)$ **by** $(auto\ intro!: inj-onI\ simp: fold-map-of\ (metis\ diff-zero\ distinct-Ex1\ distinct-upt\ length-upt\ map-of-zip-nth\ option.simps(1)))$
let $?f = RETURN\ (snd\ (fold\ (\lambda\ x\ (k, m). (Suc\ k, op-map-update\ x\ k\ m))\ tsl'\ (0, op-map-empty)))$
have $(RETURN\ (gen-enumerate\ tol\ upd\ emp\ T), ?f) \in \langle \langle A, nat-rel \rangle R_m \rangle nres-rel$
unfolding $gen-enumerate-def$ **by** $parametricity$
also **have** $(?f, op-set-enumerate\ S) \in \langle Id \rangle nres-rel$
unfolding $op-set-enumerate-def$ **using** $2\ 3\ 10$ **by** $refine-vcg\ auto$
finally **show** $(RETURN\ (gen-enumerate\ tol\ upd\ emp\ T), op-set-enumerate\ S) \in \langle \langle A, nat-rel \rangle R_m \rangle nres-rel$ **unfolding** $nres-rel-comp$ **by** $simp$
qed

lemma $gen-enumerate-it-to-list[refine-transfer-post-simp]:$
 $gen-enumerate\ (it-to-list\ it) =$
 $(\lambda\ upd\ emp\ S. snd\ (foldli\ (it-to-list\ it)\ S)\ (\lambda -. True))$
 $(\lambda\ x\ s. case\ s\ of\ (k, m) \Rightarrow (Suc\ k, upd\ x\ k\ m))\ (0, emp))$
unfolding $gen-enumerate-def$
unfolding $foldl-conv-fold[symmetric]$
unfolding $foldli-foldl[symmetric]$
by $rule$

definition $gen-build$ **where**

$gen-build\ tol\ upd\ emp\ f\ X \equiv fold\ (\lambda\ x. upd\ x\ (f\ x))\ (tol\ X)\ emp$

lemma $gen-build[autoref-rules]:$

assumes $PRIO-TAG-GEN-ALGO$

assumes $to-list: SIDE-GEN-ALGO\ (is-set-to-list\ A\ R_s\ tol)$

assumes $empty: GEN-OP\ emp\ op-map-empty\ (\langle A, B \rangle R_m)$

assumes $update: GEN-OP\ upd\ op-map-update\ (A \rightarrow B \rightarrow \langle A, B \rangle R_m \rightarrow \langle A, B \rangle R_m)$

shows $(\lambda f X. \text{gen-build tol upd emp } f X, \lambda f X. (\text{Some } \circ f) \mid' X) \in$
 $(A \rightarrow B) \rightarrow \langle A \rangle R_s \rightarrow \langle A, B \rangle R_m$

proof (*intro fun-relI*)

note [*unfolded autoref-tag-defs, param*] = *empty update*

fix $f g T S$

assume $1[\text{param}]: (g, f) \in A \rightarrow B (T, S) \in \langle A \rangle R_s$

obtain tsl' **where**

$[\text{param}]: (\text{tol } T, \text{tsl}') \in \langle A \rangle \text{list-rel}$

and $IT': \text{RETURN } \text{tsl}' \leq \text{it-to-sorted-list } (\lambda - . \text{True}) S$

using *to-list[unfolded autoref-tag-defs is-set-to-list-def]* 1(2)

by (*rule is-set-to-sorted-listE*)

from IT' **have** $10: S = \text{set } \text{tsl}' \text{ distinct } \text{tsl}'$ **unfolding** *it-to-sorted-list-def*

by *simp-all*

have $\text{gen-build tol upd emp } g T = \text{fold } (\lambda x. \text{upd } x (g x)) (\text{tol } T) \text{ emp}$

unfolding *gen-build-def* **by** *rule*

also have $(\dots, \text{fold } (\lambda x. \text{op-map-update } x (f x)) \text{tsl}' \text{ op-map-empty}) \in \langle A,$

$B \rangle R_m$

by *parametricity*

also have $\text{fold } (\lambda x. \text{op-map-update } x (f x)) \text{tsl}' m = m ++ (\text{Some } \circ f) \mid' S$

for m

unfolding *10 op-map-update-def*

by (*induct tsl' arbitrary: m rule: rev-induct*) (*auto simp add: restrict-map-insert*)

also have $\text{op-map-empty } ++ (\text{Some } \circ f) \mid' S = (\text{Some } \circ f) \mid' S$ **by** *simp*

finally show $(\text{gen-build tol upd emp } g T, (\text{Some } \circ f) \mid' S) \in \langle A, B \rangle R_m$ **by**

this

qed

definition *to-list* $\text{it } s \equiv \text{it } s \text{ top Cons Nil}$

lemma *map2set-to-list*:

assumes *GEN-ALGO-tag* (*is-map-to-list Rk unit-rel R it*)

shows $\text{is-set-to-list } Rk (\text{map2set-rel } R) (\text{to-list } (\text{map-iterator-dom } \circ (\text{foldli } \circ \text{it})))$

unfolding *is-set-to-list-def is-set-to-sorted-list-def*

proof *safe*

fix $f g$

assume $1: (f, g) \in \langle Rk \rangle \text{map2set-rel } R$

obtain xs **where** $2: (\text{it-to-list } (\text{map-iterator-dom } \circ (\text{foldli } \circ \text{it})) f, xs) \in \langle Rk \rangle$

list-rel

$\text{RETURN } xs \leq \text{it-to-sorted-list } (\lambda - . \text{True}) g$

using *map2set-to-list[OF assms]* 1

unfolding *is-set-to-list-def is-set-to-sorted-list-def*

by *auto*

have $3: \text{map-iterator-dom } (\text{foldli } xs) \text{ top } (\#) a =$

$\text{rev } (\text{map-iterator-dom } (\text{foldli } xs) (\lambda - . \text{True}) (\lambda x l. l @ [x])) (\text{rev } a)$

for $xs :: ('k \times \text{unit}) \text{list}$ **and** a

unfolding *map-iterator-dom-def set-iterator-image-def set-iterator-image-filter-def*

by (*induct xs arbitrary: a*) (*auto*)

show $\exists xs. (\text{to-list } (\text{map-iterator-dom } \circ (\text{foldli } \circ \text{it})) f, xs) \in \langle Rk \rangle \text{list-rel} \wedge$

```

RETURN xs ≤ it-to-sorted-list (λ - -. True) g
proof (intro exI conjI)
  have to-list (map-iterator-dom ∘ (foldli ∘ it)) f =
    rev (it-to-list (map-iterator-dom ∘ (foldli ∘ it)) f)
  unfolding to-list-def it-to-list-def by (simp add: 3)
  also have (rev (it-to-list (map-iterator-dom ∘ (foldli ∘ it)) f), rev xs) ∈
⟨Rk⟩ list-rel
  using 2(1) by parametricity
  finally show (to-list (map-iterator-dom ∘ (foldli ∘ it)) f, rev xs) ∈ ⟨Rk⟩
list-rel by this
  show RETURN (rev xs) ≤ it-to-sorted-list (λ - -. True) g
  using 2(2) unfolding it-to-sorted-list-def by auto
qed
qed

```

```

lemma CAST-to-list[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes SIDE-GEN-ALGO (is-set-to-list A Rs tol)
  shows (tol, CAST) ∈ ⟨A⟩ Rs → ⟨A⟩ list-set-rel
  using assms(2) unfolding autoref-tag-defs is-set-to-list-def
by (auto simp: it-to-sorted-list-def list-set-rel-def in-br-conv elim!: is-set-to-sorted-listE)

```

```

lemma param-foldli:
  assumes (xs, ys) ∈ ⟨Ra⟩ list-rel
  assumes (c, d) ∈ Rs → bool-rel
  assumes ∧ x y. (x, y) ∈ Ra ⇒ x ∈ set xs ⇒ y ∈ set ys ⇒ (f x, g y) ∈
Rs → Rs
  assumes (a, b) ∈ Rs
  shows (foldli xs c f a, foldli ys d g b) ∈ Rs
using assms
proof (induct arbitrary: a b)
  case 1
  then show ?case by simp
next
  case (2 x y xs ys)
  show ?case
  proof (cases c a)
    case True
    have 10: (c a, d b) ∈ bool-rel using 2 by parametricity
    have 20: d b using 10 True by auto
    have 30: (foldli xs c f (f x a), foldli ys d g (g y b)) ∈ Rs
      by (auto intro!: 2 2(5)[THEN fun-relD])
    show ?thesis using True 20 30 by simp
  next
  case False
  have 10: (c a, d b) ∈ bool-rel using 2 by parametricity
  have 20: ¬ d b using 10 False by auto
  show ?thesis unfolding foldli.simps using False 20 2 by simp

```

qed
 qed
 lemma *det-fold-sorted-set*:
 assumes 1: *det-fold-set* ordR c' f' σ' result
 assumes 2: *is-set-to-sorted-list* ordR Rk Rs tsl
 assumes SREF[*param*]: (s,s') ∈ ⟨Rk⟩Rs
 assumes [*param*]: (c,c') ∈ Rσ → Id
 assumes [*param*]: $\bigwedge x y. (x, y) \in Rk \implies y \in s' \implies (f x, f' y) \in R\sigma \rightarrow R\sigma$
 assumes [*param*]: (σ,σ') ∈ Rσ
 shows (foldli (tsl s) c f σ, result s') ∈ Rσ
 proof –
 obtain tsl' where
 n[*param*]: (tsl s, tsl') ∈ ⟨Rk⟩list-rel
 and IT: RETURN tsl' ≤ *it-to-sorted-list* ordR s'
 using 2 SREF
 by (rule *is-set-to-sorted-listE*)
 from IT have suen: s' = set tsl'
 unfolding *it-to-sorted-list-def* by *simp-all*
 have (foldli (tsl s) c f σ, foldli tsl' c' f' σ') ∈ Rσ
 using *assms*(4, 5, 6) n unfolding suen
 using *param-foldli*[OF n *assms*(4)] *assms* by *simp*
 also have foldli tsl' c' f' σ' = result s'
 using 1 IT
 unfolding *det-fold-set-def* *it-to-sorted-list-def*
 by *simp*
 finally show ?thesis .

qed
 lemma *det-fold-set*:
 assumes *det-fold-set* (λ-. True) c' f' σ' result
 assumes *is-set-to-list* Rk Rs tsl
 assumes (s,s') ∈ ⟨Rk⟩Rs
 assumes (c,c') ∈ Rσ → Id
 assumes $\bigwedge x y. (x, y) \in Rk \implies y \in s' \implies (f x, f' y) \in R\sigma \rightarrow R\sigma$
 assumes (σ,σ') ∈ Rσ
 shows (foldli (tsl s) c f σ, result s') ∈ Rσ
 using *assms* unfolding *is-set-to-list-def* by (rule *det-fold-sorted-set*)

lemma *gen-image*[*autoref-rules-raw*]:
 assumes PRIO-TAG-GEN-ALGO
 assumes IT: SIDE-GEN-ALGO (*is-set-to-list* Rk Rs1 it1)
 assumes INS: GEN-OP ins2 Set.insert (Rk' → ⟨Rk'⟩Rs2 → ⟨Rk'⟩Rs2)
 assumes EMPTY: GEN-OP empty2 {} (⟨Rk'⟩Rs2)
 assumes $\bigwedge xi x. (xi, x) \in Rk \implies x \in s \implies (fi xi, f \$ x) \in Rk'$
 assumes (l, s) ∈ ⟨Rk⟩Rs1
 shows (*gen-image* (λ x. foldli (it1 x) empty2 ins2 fi l,
 (OP image :: (Rk → Rk') → (⟨Rk⟩Rs1) → (⟨Rk'⟩Rs2)) \$ f \$ s) ∈ (⟨Rk'⟩Rs2)
 proof –
 note [unfolding *autoref-tag-defs*, *param*] = INS EMPTY
 note 1 = *det-fold-set*[OF foldli-image IT[unfolding *autoref-tag-defs*]]
 show ?thesis using *assms* 1 unfolding *gen-image-def* *autoref-tag-defs* by

parametricity
qed

end

end

29 Implementation of Deterministic Rabin Automata

theory *DRA-Implement*

imports

DRA-Refine

../Basic/Implement

begin

datatype (*'label*, *'state*) *drai* = *drai*
 (*alphabeti*: *'label list*)
 (*initiali*: *'state*)
 (*transitioni*: *'label* \Rightarrow *'state* \Rightarrow *'state*)
 (*conditioni*: *'state rabin gen*)

definition *drai-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow
 ((*'label*₁, *'state*₁) *drai* \times (*'label*₂, *'state*₂) *drai*) *set* **where**
 [*to-relAPP*]: *drai-rel* *L S* \equiv {(*A*₁, *A*₂).
 (*alphabeti* *A*₁, *alphabeti* *A*₂) \in $\langle L \rangle$ *list-rel* \wedge
 (*initiali* *A*₁, *initiali* *A*₂) \in *S* \wedge
 (*transitioni* *A*₁, *transitioni* *A*₂) \in *L* \rightarrow *S* \rightarrow *S* \wedge
 (*conditioni* *A*₁, *conditioni* *A*₂) \in \langle *rabin-rel S* \rangle *list-rel*}

lemma *drai-param*[*param*]:

(*drai*, *drai*) \in $\langle L \rangle$ *list-rel* \rightarrow *S* \rightarrow (*L* \rightarrow *S* \rightarrow *S*) \rightarrow
 \langle *rabin-rel S* \rangle *list-rel* \rightarrow $\langle L, S \rangle$ *drai-rel*
 (*alphabeti*, *alphabeti*) \in $\langle L, S \rangle$ *drai-rel* \rightarrow $\langle L \rangle$ *list-rel*
 (*initiali*, *initiali*) \in $\langle L, S \rangle$ *drai-rel* \rightarrow *S*
 (*transitioni*, *transitioni*) \in $\langle L, S \rangle$ *drai-rel* \rightarrow *L* \rightarrow *S* \rightarrow *S*
 (*conditioni*, *conditioni*) \in $\langle L, S \rangle$ *drai-rel* \rightarrow \langle *rabin-rel S* \rangle *list-rel*
unfolding *drai-rel-def fun-rel-def* **by** *auto*

definition *drai-dra-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow
 ((*'label*₁, *'state*₁) *drai* \times (*'label*₂, *'state*₂) *dra*) *set* **where**
 [*to-relAPP*]: *drai-dra-rel* *L S* \equiv {(*A*₁, *A*₂).
 (*alphabeti* *A*₁, *alphabet* *A*₂) \in $\langle L \rangle$ *list-set-rel* \wedge
 (*initiali* *A*₁, *initial* *A*₂) \in *S* \wedge
 (*transitioni* *A*₁, *transition* *A*₂) \in *L* \rightarrow *S* \rightarrow *S* \wedge
 (*conditioni* *A*₁, *condition* *A*₂) \in \langle *rabin-rel S* \rangle *list-rel*}

lemma *drai-dra-param*[*param*, *autoref-rules*]:

(*drai*, *dra*) \in $\langle L \rangle$ *list-set-rel* \rightarrow *S* \rightarrow (*L* \rightarrow *S* \rightarrow *S*) \rightarrow
 \langle *rabin-rel S* \rangle *list-rel* \rightarrow $\langle L, S \rangle$ *drai-dra-rel*

```

(alphabeti, alphabet) ∈ ⟨L, S⟩ drai-dra-rel → ⟨L⟩ list-set-rel
(initiali, initial) ∈ ⟨L, S⟩ drai-dra-rel → S
(transitioni, transition) ∈ ⟨L, S⟩ drai-dra-rel → L → S → S
(conditioni, condition) ∈ ⟨L, S⟩ drai-dra-rel → ⟨rabin-rel S⟩ list-rel
unfolding drai-dra-rel-def fun-rel-def by auto

```

```

definition drai-dra :: ('label', 'state') drai ⇒ ('label', 'state') dra where
  drai-dra A ≡ dra (set (alphabeti A)) (initiali A) (transitioni A) (conditioni A)
definition drai-invar :: ('label', 'state') drai ⇒ bool where
  drai-invar A ≡ distinct (alphabeti A)

```

```

lemma drai-dra-id-param[param]: (drai-dra, id) ∈ ⟨L, S⟩ drai-dra-rel → ⟨L, S⟩
dra-rel

```

```

proof
  fix Ai A
  assume 1: (Ai, A) ∈ ⟨L, S⟩ drai-dra-rel
  have 2: drai-dra Ai = dra (set (alphabeti Ai)) (initiali Ai) (transitioni Ai)
(conditioni Ai)
  unfolding drai-dra-def by rule
  have 3: id A = dra (id (alphabet A)) (initial A) (transition A) (condition A)
by simp
  show (drai-dra Ai, id A) ∈ ⟨L, S⟩ dra-rel unfolding 2 3 using 1 by para-
metricity
qed

```

```

lemma drai-dra-br: ⟨Id, Id⟩ drai-dra-rel = br drai-dra drai-invar

```

```

proof safe
  show (A, B) ∈ ⟨Id, Id⟩ drai-dra-rel if (A, B) ∈ br drai-dra drai-invar
  for A and B :: ('a', 'b') dra
  using that unfolding drai-dra-rel-def drai-dra-def drai-invar-def
  by (auto simp: in-br-conv list-set-rel-def)
  show (A, B) ∈ br drai-dra drai-invar if (A, B) ∈ ⟨Id, Id⟩ drai-dra-rel
  for A and B :: ('a', 'b') dra
  proof –
  have 1: (drai-dra A, id B) ∈ ⟨Id, Id⟩ dra-rel using that by parametricity
  have 2: drai-invar A
  using drai-dra-param(2 - 5)[param-fo, OF that]
  by (auto simp: in-br-conv list-set-rel-def drai-invar-def)
  show ?thesis using 1 2 unfolding in-br-conv by auto
qed
qed

```

end

30 Exploration of Deterministic Rabin Automata

```

theory DRA-Nodes

```

```

imports

```

```

  DFS-Framework.Reachable-Nodes

```

DRA-Implement
begin

definition *dra-G* :: ('label, 'state) dra \Rightarrow 'state graph-rec **where**
dra-G A \equiv (\mid g-V = UNIV, g-E = E-of-succ (successors A), g-V0 = {initial A} \mid)

lemma *dra-G-graph[simp]*: graph (dra-G A) **unfolding** *dra-G-def graph-def* **by** *simp*

lemma *dra-G-reachable-nodes*: op-reachable (dra-G A) = nodes A
unfolding *op-reachable-def dra-G-def graph-rec.simps E-of-succ-def*
proof *safe*

show $p \in$ nodes A **if** (initial A, p) \in {(u, v). v \in successors A u}* **for** p
using that **by** *induct auto*
show (initial A, p) \in {(u, v). v \in successors A u}* **if** p \in nodes A **for** p
using that **by** (*induct*) (*auto intro: rtrancl-into-rtrancl*)

qed

context
begin

interpretation *autoref-syn* **by** *this*

lemma *dra-G-ahs*: *dra-G* A = (\mid g-V = UNIV, g-E = E-of-succ (λ p. CAST
(λ a. transition A a p :: S) 'alphabet A :: \langle S \rangle ahs-rel bhc)), g-V0 = {initial
A} \mid)

unfolding *dra-G-def CAST-def id-apply E-of-succ-def autoref-tag-defs* **by** *auto*

schematic-goal *drai-Gi*:

notes *map2set-to-list[autoref-ga-rules]*

fixes S :: ('statei \times 'state) set

assumes [*autoref-ga-rules*]: *is-bounded-hashcode* S seq bhc

assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* TYPE('statei) hms

assumes [*autoref-rules*]: (seq, HOL.eq) \in S \rightarrow S \rightarrow bool-rel

assumes [*autoref-rules*]: (Ai, A) \in \langle L, S \rangle drai-dra-rel

shows (?f :: ?'a, RETURN (dra-G A)) \in ?A

unfolding *dra-G-ahs[where S = S and bhc = bhc]* **by** (*autoref-monadic*
(*plain*))

concrete-definition *drai-Gi* **uses** *drai-Gi*

lemma *drai-Gi-refine[autoref-rules]*:

fixes S :: ('statei \times 'state) set

assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* S seq bhc)

assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* TYPE('statei) hms)

assumes *GEN-OP* seq HOL.eq (S \rightarrow S \rightarrow bool-rel)

shows (DRA-Nodes.drai-Gi seq bhc hms, dra-G) \in \langle L, S \rangle drai-dra-rel \rightarrow
 \langle unit-rel, S \rangle g-impl-rel-ext

using *drai-Gi.refine[THEN RETURN-nres-relD]* *assms* **unfolding** *autoref-tag-defs*
by *blast*


```

schematic-goal dra-nodes:
  fixes S :: ('statei × 'state) set
  assumes [simp]: finite ((g-E (dra-G A))* “ g-V0 (dra-G A))
  assumes [autoref-ga-rules]: is-bounded-hashcode S seq bhc
  assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE('statei) hms
  assumes [autoref-rules]: (seq, HOL.eq) ∈ S → S → bool-rel
  assumes [autoref-rules]: (Ai, A) ∈ ⟨L, S⟩ drai-dra-rel
  shows (?f :: ?'a, op-reachable (dra-G A)) ∈ ?R by autoref
concrete-definition dra-nodes uses dra-nodes
lemma dra-nodes-refine[autoref-rules]:
  fixes S :: ('statei × 'state) set
  assumes SIDE-PRECOND (finite (nodes A))
  assumes SIDE-GEN-ALGO (is-bounded-hashcode S seq bhc)
  assumes SIDE-GEN-ALGO (is-valid-def-hm-size TYPE('statei) hms)
  assumes GEN-OP seq HOL.eq (S → S → bool-rel)
  assumes (Ai, A) ∈ ⟨L, S⟩ drai-dra-rel
  shows (DRA-Nodes.dra-nodes seq bhc hms Ai,
    (OP nodes :: ⟨L, S⟩ drai-dra-rel → ⟨S⟩ ahs-rel bhc) $ A) ∈ ⟨S⟩ ahs-rel bhc
proof –
  have finite ((g-E (dra-G A))* “ g-V0 (dra-G A))
  using assms(1) unfolding autoref-tag-defs dra-G-reachable-nodes[symmetric]
by simp
  then show ?thesis using dra-nodes.refine assms
  unfolding autoref-tag-defs dra-G-reachable-nodes[symmetric] by blast
qed

end

end

```

31 Explicit Deterministic Rabin Automata

```

theory DRA-Explicit
imports DRA-Nodes
begin

```

```

datatype ('label, 'state) drae = drae
  (alphabet: 'label set)
  (initiale: 'state)
  (transitione: ('state × 'label × 'state) set)
  (conditione: ('state set × 'state set) list)

```

definition drae-rel **where**

```

[to-relAPP]: drae-rel L S ≡ {(A1, A2).
  (alphabet A1, alphabet A2) ∈ ⟨L⟩ set-rel ∧
  (initiale A1, initiale A2) ∈ S ∧
  (transitione A1, transitione A2) ∈ ⟨S ×r L ×r S⟩ set-rel ∧
  (conditione A1, conditione A2) ∈ ⟨⟨S⟩ set-rel ×r ⟨S⟩ set-rel⟩ list-rel}

```

lemma *drae-param*[*param, autoref-rules*]:
 $(drae, drae) \in \langle L \rangle \text{ set-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel} \rightarrow$
 $\langle \langle S \rangle \text{ set-rel} \times_r \langle S \rangle \text{ set-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ drae-rel}$
 $(\text{alphabet}, \text{alphabet}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle L \rangle \text{ set-rel}$
 $(\text{initiale}, \text{initiale}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow S$
 $(\text{transition}, \text{transition}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel}$
 $(\text{condition}, \text{condition}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle \langle S \rangle \text{ set-rel} \times_r \langle S \rangle \text{ set-rel} \rangle \text{ list-rel}$
unfolding *drae-rel-def* **by** *auto*

lemma *drae-rel-id*[*simp*]: $\langle Id, Id \rangle \text{ drae-rel} = Id$ **unfolding** *drae-rel-def* **using** *drae.expand* **by** *auto*

lemma *drae-rel-comp*[*simp*]: $\langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle \text{ drae-rel} = \langle L_1, S_1 \rangle \text{ drae-rel} \ O$
 $\langle L_2, S_2 \rangle \text{ drae-rel}$

proof *safe*

fix *A B*

assume *1*: $(A, B) \in \langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle \text{ drae-rel}$

obtain *a b c d* **where** *2*:

$(\text{alphabet} \ A, a) \in \langle L_1 \rangle \text{ set-rel}$ $(a, \text{alphabet} \ B) \in \langle L_2 \rangle \text{ set-rel}$

$(\text{initiale} \ A, b) \in S_1$ $(b, \text{initiale} \ B) \in S_2$

$(\text{transition} \ A, c) \in \langle S_1 \times_r L_1 \times_r S_1 \rangle \text{ set-rel}$ $(c, \text{transition} \ B) \in \langle S_2 \times_r L_2$
 $\times_r S_2 \rangle \text{ set-rel}$

$(\text{condition} \ A, d) \in \langle \langle S_1 \rangle \text{ set-rel} \times_r \langle S_1 \rangle \text{ set-rel} \rangle \text{ list-rel}$

$(d, \text{condition} \ B) \in \langle \langle S_2 \rangle \text{ set-rel} \times_r \langle S_2 \rangle \text{ set-rel} \rangle \text{ list-rel}$

using *1* **unfolding** *drae-rel-def prod-rel-compp set-rel-compp* **by** *auto*

show $(A, B) \in \langle L_1, S_1 \rangle \text{ drae-rel} \ O \ \langle L_2, S_2 \rangle \text{ drae-rel}$

proof

show $(A, \text{drae} \ a \ b \ c \ d) \in \langle L_1, S_1 \rangle \text{ drae-rel}$ **using** *2* **unfolding** *drae-rel-def*

by *auto*

show $(\text{drae} \ a \ b \ c \ d, B) \in \langle L_2, S_2 \rangle \text{ drae-rel}$ **using** *2* **unfolding** *drae-rel-def*

by *auto*

qed

next

show $(A, C) \in \langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle \text{ drae-rel}$

if $(A, B) \in \langle L_1, S_1 \rangle \text{ drae-rel}$ $(B, C) \in \langle L_2, S_2 \rangle \text{ drae-rel}$ **for** *A B C*

using *that* **unfolding** *drae-rel-def prod-rel-compp set-rel-compp* **by** *auto*

qed

consts *i-drae-scheme* :: *interface* \Rightarrow *interface* \Rightarrow *interface*

context

begin

interpretation *autoref-syn* **by** *this*

lemma *drae-scheme-itype*[*autoref-itype*]:

drae :: $\langle L \rangle_i \text{ i-set} \rightarrow_i S \rightarrow_i \langle \langle S, \langle L, S \rangle_i \text{ i-prod} \rangle_i \text{ i-prod} \rangle_i \text{ i-set} \rightarrow_i$

$\langle \langle \langle S \rangle_i \text{ i-set}, \langle S \rangle_i \text{ i-set} \rangle_i \text{ i-prod} \rangle_i \text{ i-list} \rightarrow_i \langle L, S \rangle_i \text{ i-drae-scheme}$

$alphabet_e ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle L \rangle_i \text{ i-set}$
 $initiale ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i S$
 $transizione ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle \langle S, \langle L, S \rangle_i \text{ i-prod} \rangle_i \text{ i-prod} \rangle_i \text{ i-set}$
 $condizione ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle \langle \langle S \rangle_i \text{ i-set}, \langle S \rangle_i \text{ i-set} \rangle_i \text{ i-prod} \rangle_i \text{ i-list}$
by auto

end

datatype ('label, 'state) draei = draei
 (alphabet_ei: 'label list)
 (initialei: 'state)
 (transizionei: ('state × 'label × 'state) list)
 (condizionei: ('state list × 'state list) list)

definition draei-rel where

[to-relAPP]: draei-rel $L S \equiv \{(A_1, A_2).$
 $(alphabet_ei A_1, alphabet_ei A_2) \in \langle L \rangle \text{ list-rel} \wedge$
 $(initialei A_1, initialei A_2) \in S \wedge$
 $(transizionei A_1, transizionei A_2) \in \langle S \times_r L \times_r S \rangle \text{ list-rel} \wedge$
 $(condizionei A_1, condizionei A_2) \in \langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle \text{ list-rel}\}$

lemma draei-param[param, autoref-rules]:

$(draei, draei) \in \langle L \rangle \text{ list-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-rel} \rightarrow$
 $\langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ draei-rel}$
 $(alphabet_ei, alphabet_ei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle L \rangle \text{ list-rel}$
 $(initialei, initialei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow S$
 $(transizionei, transizionei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-rel}$
 $(condizionei, condizionei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle$

list-rel

unfolding draei-rel-def by auto

definition draei-drae-rel where

[to-relAPP]: draei-drae-rel $L S \equiv \{(A_1, A_2).$
 $(alphabet_ei A_1, alphabet_e A_2) \in \langle L \rangle \text{ list-set-rel} \wedge$
 $(initialei A_1, initiale A_2) \in S \wedge$
 $(transizionei A_1, transizione A_2) \in \langle S \times_r L \times_r S \rangle \text{ list-set-rel} \wedge$
 $(condizionei A_1, condizione A_2) \in \langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle \text{ list-set-rel} \rangle \text{ list-rel}\}$

lemmas [autoref-rel-intf] = REL-INTFI[of draei-drae-rel i-drae-scheme]

lemma draei-drae-param[param, autoref-rules]:

$(draei, drae) \in \langle L \rangle \text{ list-set-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-set-rel} \rightarrow$
 $\langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle \text{ list-set-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ draei-drae-rel}$
 $(alphabet_ei, alphabet_e) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle L \rangle \text{ list-set-rel}$
 $(initialei, initiale) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow S$
 $(transizionei, transizione) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-set-rel}$
 $(condizionei, condizione) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle$

list-set-rel) list-rel

unfolding draei-drae-rel-def by auto

definition draei-drae where

$\text{draei-drae } A \equiv \text{drae } (\text{set } (\text{alphabet} A)) (\text{initiale} A)$
 $(\text{set } (\text{transition} A)) (\text{map } (\text{map-prod } \text{set } \text{set}) (\text{condition} A))$

lemma draei-drae-id-param[*param*]: $(\text{draei-drae}, \text{id}) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle L, S \rangle \text{ drae-rel}$

proof

fix *Ai A*

assume *1: (Ai, A) ∈ ⟨L, S⟩ draei-drae-rel*

have *2: draei-drae Ai = drae (set (alphabet Ai)) (initiale Ai)*

(set (transition Ai)) (map (map-prod set set) (condition Ai)) **unfolding**

draei-drae-def **by rule**

have *3: id A = drae (id (alphabet A)) (initiale A)*

(id (transition A)) (map (map-prod id id) (condition A)) **by simp**

show $(\text{draei-drae } Ai, \text{id } A) \in \langle L, S \rangle \text{ drae-rel}$ **unfolding 2 3 using 1 by parametricity**

qed

abbreviation transitions $L S s \equiv \bigcup a \in L. \bigcup p \in S. \{p\} \times \{a\} \times \{s \ a \ p\}$

abbreviation succs $T a p \equiv \text{the-elem } ((T \text{ “ } \{p\} \text{ “ } \{a\})$

definition wft $:: 'label \text{ set} \Rightarrow 'state \text{ set} \Rightarrow ('state \times 'label \times 'state) \text{ set} \Rightarrow \text{bool}$
where

$\text{wft } L S T \equiv \forall a \in L. \forall p \in S. \text{is-singleton } ((T \text{ “ } \{p\} \text{ “ } \{a\})$

lemma wft-param[*param*]:

assumes *bijjective S bijjective L*

shows $(\text{wft}, \text{wft}) \in \langle L \rangle \text{ set-rel} \rightarrow \langle S \rangle \text{ set-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel} \rightarrow \text{bool-rel}$

using *assms* **unfolding wft-def by parametricity**

lemma wft-transitions: $\text{wft } L S (\text{transitions } L S s)$ **unfolding wft-def is-singleton-def by auto**

definition dra-drae where $\text{dra-drae } A \equiv \text{drae } (\text{alphabet } A) (\text{initial } A)$

$(\text{transitions } (\text{alphabet } A) (\text{nodes } A) (\text{transition } A))$

$(\text{map } (\lambda (P, Q). (\text{Set.filter } P (\text{nodes } A), \text{Set.filter } Q (\text{nodes } A)))) (\text{condition } A))$

definition drae-dra where $\text{drae-dra } A \equiv \text{dra } (\text{alphabet } A) (\text{initiale } A)$

$(\text{succs } (\text{transition } A)) (\text{map } (\lambda (I, F). (\lambda p. p \in I, \lambda p. p \in F)) (\text{condition } A))$

lemma set-rel-Domain-Range[*intro!*, *simp*]: $(\text{Domain } A, \text{Range } A) \in \langle A \rangle \text{ set-rel}$
unfolding set-rel-def by auto

lemma dra-drae-param[*param*]: $(\text{dra-drae}, \text{dra-drae}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle L, S \rangle \text{ drae-rel}$

unfolding dra-drae-def by parametricity

lemma drae-dra-param[*param*]:

assumes *bijjective L bijjective S*

```

assumes wft (Range L) (Range S) (transitione B)
assumes [param]: (A, B) ∈ ⟨L, S⟩ drae-rel
shows (drae-dra A, drae-dra B) ∈ ⟨L, S⟩ dra-rel
proof –
  have 1: (wft (Domain L) (Domain S) (transitione A), wft (Range L) (Range
S) (transitione B)) ∈ bool-rel
    using assms(1, 2) by parametricity auto
  have 2: wft (Domain L) (Domain S) (transitione A) using assms(3) 1 by
simp
  show ?thesis
    using assms(1 – 3) 2 assms(2)[unfolding bijective-alt]
    unfolding drae-dra-def wft-def
    by parametricity force+
qed

```

```

lemma succs-transitions-param[param]:
  (succs ∘ transitions L S, id) ∈ (Id-on L → Id-on S → Id-on S) → (Id-on L →
Id-on S → Id-on S)

```

```

proof
  fix f g
  assume 1[param]: (f, g) ∈ Id-on L → Id-on S → Id-on S
  show ((succs ∘ transitions L S) f, id g) ∈ Id-on L → Id-on S → Id-on S
  proof safe
    fix a p
    assume 2: a ∈ L p ∈ S
    have (succs ∘ transitions L S) f a p = succs (transitions L S f) a p by simp
    also have (transitions L S f “ {p}” “ {a} = {f a p}” using 2 by auto
    also have the-elem ... = f a p by simp
    also have (... , g a p) ∈ Id-on S using 2 by parametricity auto
    finally show (succs ∘ transitions L S) f a p = id g a p by simp
    show id g a p ∈ S using 1[param-fo] 2 by simp
  qed

```

```

qed
lemma drae-dra-dra-drae-param[param]:
  ((drae-dra ∘ dra-drae) A, id A) ∈ ⟨Id-on (alphabet A), Id-on (nodes A)⟩ dra-rel
proof –

```

```

  have [param]: (λ (P, Q). (λ p. p ∈ Set.filter P (nodes A), λ p. p ∈ Set.filter Q
(nodes A)), id) ∈
    pred-rel (Id-on (nodes A)) ×r pred-rel (Id-on (nodes A)) → rabin-rel (Id-on
(nodes A))
    unfolding fun-rel-def Id-on-def by auto
  have (drae-dra ∘ dra-drae) A = dra (alphabet A) (initial A)
    ((succs ∘ transitions (alphabet A) (nodes A)) (transition A))
    (map (λ (P, Q). (λ p. p ∈ Set.filter P (nodes A), λ p. p ∈ Set.filter Q (nodes
A)))) (condition A))
    unfolding drae-dra-def dra-drae-def by auto
  also have (... , dra (alphabet A) (initial A) (id (transition A)) (map id (condition
A))) ∈
    ⟨Id-on (alphabet A), Id-on (nodes A)⟩ dra-rel using dra-rel-eq by parametricity

```

auto
also have dra (*alphabet* A) (*initial* A) (*id* (*transition* A)) (*map id* (*condition* A)) = $\text{id } A$ **by simp**
finally show *?thesis* **by this**
qed

definition draei-dra-rel **where**
 $[\text{to-relAPP}]: \text{draei-dra-rel } L S \equiv \{(Ae, A). (\text{drae-dra } (\text{draei-drae } Ae), A) \in \langle L, S \rangle \text{ dra-rel}\}$
lemma draei-dra-id [*param*]: $(\text{drae-dra} \circ \text{draei-drae}, \text{id}) \in \langle L, S \rangle \text{ draei-dra-rel} \rightarrow \langle L, S \rangle \text{ dra-rel}$
unfolding draei-dra-rel-def **by auto**

end

32 Explore and Enumerate Nodes of Deterministic Rabin Automata

theory DRA-Translate
imports DRA-Explicit
begin

32.1 Syntax

no-syntax *-do-let* :: $[p\text{trn}, 'a] \Rightarrow \text{do-bind } ((\text{?let } - = / -) [1000, 13] 13)$
syntax *-do-let* :: $[p\text{trn}, 'a] \Rightarrow \text{do-bind } ((\text{?let } - = / -) 13)$

33 Image on Explicit Automata

definition drae-image **where** $\text{drae-image } f A \equiv \text{drae}$ (*alphabet* A) (f (*initiale* A))
 $((\lambda (p, a, q). (f p, a, f q)) \text{ 'transitione } A)$ (*map* (*map-prod* (*image* f) (*image* f))) (*conditione* A)

lemma drae-image-param [*param*]: $(\text{drae-image}, \text{drae-image}) \in (S \rightarrow T) \rightarrow \langle L, S \rangle \text{ drae-rel} \rightarrow \langle L, T \rangle \text{ drae-rel}$
unfolding drae-image-def **by parametricity**

lemma drae-image-id [*simp*]: $\text{drae-image id} = \text{id}$ **unfolding** drae-image-def **by auto**

lemma $\text{drae-image-dra-drae}$: $\text{drae-image } f (\text{dra-drae } A) = \text{drae}$ (*alphabet* A) (f (*initial* A))
 $(\bigcup p \in \text{nodes } A. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\})$
 $(\text{map } (\lambda (P, Q). (f \text{ ' } \{p \in \text{nodes } A. P p\}, f \text{ ' } \{p \in \text{nodes } A. Q p\})))$ (*condition* A)
unfolding dra-drae-def drae-image-def drae.simps Set.filter-def **by force**

34 Exploration and Translation

definition *trans-spec where*

$$\text{trans-spec } A f \equiv \bigcup p \in \text{nodes } A. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}$$

definition *trans-algo where*

$$\begin{aligned} \text{trans-algo } N L S f \equiv & \\ & \text{FOREACH } N (\lambda p T. \text{do } \{ \\ & \quad \text{ASSERT } (p \in N); \\ & \quad \text{FOREACH } L (\lambda a T. \text{do } \{ \\ & \quad \quad \text{ASSERT } (a \in L); \\ & \quad \quad \text{let } q = S a p; \\ & \quad \quad \text{ASSERT } ((f p, a, f q) \notin T); \\ & \quad \quad \text{RETURN } (\text{insert } (f p, a, f q) T) \} \\ & \quad \} T \} \\ & \} \} \end{aligned}$$

lemma *trans-algo-refine:*

assumes *finite (nodes A) finite (alphabet A) inj-on f (nodes A)*

assumes $N = \text{nodes } A$ $L = \text{alphabet } A$ $S = \text{transition } A$

shows $(\text{trans-algo } N L S f, \text{SPEC } (\text{HOL.eq } (\text{trans-spec } A f))) \in \langle \text{Id} \rangle \text{ nres-rel}$

unfolding *trans-algo-def trans-spec-def assms(4-6)*

proof (*refine-vcg FOREACH-rule-insert-eq*)

show *finite (nodes A) using assms(1) by this*

show $(\bigcup p \in \text{nodes } A. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}) =$

$$(\bigcup p \in \text{nodes } A. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\})$$

by rule

show $(\bigcup p \in \{\}. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}) =$

$\{\}$ **by simp**

fix $T x$

assume 1: $T \subseteq \text{nodes } A$ $x \in \text{nodes } A$ $x \notin T$

show *finite (alphabet A) using assms(2) by this*

show $(\bigcup a \in \{\}. f \text{ ' } \{x\} \times \{a\} \times f \text{ ' } \{\text{transition } A a x\}) \cup$

$$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}) =$$

$$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\})$$

$$(\bigcup a \in \text{alphabet } A. f \text{ ' } \{x\} \times \{a\} \times f \text{ ' } \{\text{transition } A a x\}) \cup$$

$$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}) =$$

$$(\bigcup p \in \text{insert } x T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\})$$

by auto

fix $Ta xa$

assume 2: $Ta \subseteq \text{alphabet } A$ $xa \in \text{alphabet } A$ $xa \notin Ta$

show $(f x, xa, f (\text{transition } A xa x)) \notin (\bigcup a \in Ta. f \text{ ' } \{x\} \times \{a\} \times f \text{ ' } \{\text{transition } A a x\}) \cup$

$$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\})$$

using 1 2(3) assms(3) by (auto dest: inj-onD)

show $(\bigcup a \in \text{insert } xa Ta. f \text{ ' } \{x\} \times \{a\} \times f \text{ ' } \{\text{transition } A a x\}) \cup$

$$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition } A a p\}) =$$

```

      insert (f x, xa, f (transition A xa x)) (( $\bigcup a \in Ta. f \text{ ' } \{x\} \times \{a\} \times f \text{ ' }$ 
{transition A a x})  $\cup$ 
      ( $\bigcup p \in T. \bigcup a \in \text{alphabet } A. f \text{ ' } \{p\} \times \{a\} \times f \text{ ' } \{\text{transition A a p}\}$ ))
    by simp
  qed

```

definition *to-draei* :: ('state, 'label) dra \Rightarrow ('state, 'label) dra
 where *to-draei* \equiv id

schematic-goal *to-draei-impl*:

```

  fixes S :: ('statei  $\times$  'state) set
  assumes [simp]: finite (nodes A)
  assumes [autoref-ga-rules]: is-bounded-hashcode S seq bhc
  assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE('statei) hms
  assumes [autoref-rules]: (seq, HOL.eq)  $\in$  S  $\rightarrow$  S  $\rightarrow$  bool-rel
  assumes [autoref-rules]: (Ai, A)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel
  shows (?f :: ?'a, do {
    let N = nodes A;
    f  $\leftarrow$  op-set-enumerate N;
    ASSERT (dom f = N);
    ASSERT (f (initial A)  $\neq$  None);
    ASSERT ( $\forall a \in \text{alphabet } A. \forall p \in \text{dom } f. f (\text{transition A a p}) \neq \text{None}$ );
    T  $\leftarrow$  trans-algo N (alphabet A) (transition A) ( $\lambda x. \text{the } (f x)$ );
    RETURN (drae (alphabet A) (( $\lambda x. \text{the } (f x)$ ) (initial A)) T
      (map ( $\lambda (P, Q). ((\lambda x. \text{the } (f x)) \text{ ' } \{p \in N. P p\}, (\lambda x. \text{the } (f x)) \text{ ' } \{p \in$ 
N. Q p})) (condition A))))
  })  $\in$  ?R

```

unfolding *trans-algo-def* **by** (autoref-monadic (plain))

concrete-definition *to-draei-impl* **uses** *to-draei-impl*

lemma *to-draei-impl-refine'*:

```

  fixes S :: ('statei  $\times$  'state) set
  assumes finite (nodes A)
  assumes is-bounded-hashcode S seq bhc
  assumes is-valid-def-hm-size TYPE('statei) hms
  assumes (seq, HOL.eq)  $\in$  S  $\rightarrow$  S  $\rightarrow$  bool-rel
  assumes (Ai, A)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel
  shows (RETURN (to-draei-impl seq bhc hms Ai), do {
    f  $\leftarrow$  op-set-enumerate (nodes A);
    RETURN (drae-image (the  $\circ$  f) (dra-drae A))
  })  $\in$   $\langle$  $\langle$ L, nat-rel $\rangle$  draei-drae-rel $\rangle$  nres-rel

```

proof –

```

  have 1: finite (alphabet A)
  using drai-dra-param(2)[param-fo, OF assms(5)] list-set-rel-finite
  unfolding finite-set-rel-def by auto
  note to-draei-impl.refine[OF assms]
  also have (do {
    let N = nodes A;
    f  $\leftarrow$  op-set-enumerate N;

```



```

    ASSERT (dom f = N);
    ASSERT (f (initial A) ≠ None);
    ASSERT (∀ a ∈ alphabet A. ∀ p ∈ dom f. f (transition A a p) ≠ None);
    T ← trans-algo N (alphabet A) (transition A) (λ x. the (f x));
    RETURN (drae (alphabet A) ((λ x. the (f x)) (initial A)) T
      (map (λ (P, Q). ((λ x. the (f x)) ‘ {p ∈ N. P p}, (λ x. the (f x)) ‘ {p ∈
N. Q p})) (condition A)))
  }, do {
    f ← op-set-enumerate (nodes A);
    T ← SPEC (HOL.eq (trans-spec A (λ x. the (f x))));
    RETURN (drae (alphabet A) ((λ x. the (f x)) (initial A)) T
      (map (λ (P, Q). ((λ x. the (f x)) ‘ {p ∈ nodes A. P p}, (λ x. the (f x)) ‘
{p ∈ nodes A. Q p})) (condition A)))
  }) ∈ ⟨Id⟩ nres-rel
  unfolding Let-def comp-apply op-set-enumerate-def using assms(1) 1
  by (refine-vcg vcg0[OF trans-algo-refine]) (auto intro!: inj-on-map-the[unfolded
comp-apply])
  also have (do {
    f ← op-set-enumerate (nodes A);
    T ← SPEC (HOL.eq (trans-spec A (λ x. the (f x))));
    RETURN (drae (alphabet A) ((λ x. the (f x)) (initial A)) T
      (map (λ (P, Q). ((λ x. the (f x)) ‘ {p ∈ nodes A. P p}, (λ x. the (f x)) ‘
{p ∈ nodes A. Q p})) (condition A)))
  }, do {
    f ← op-set-enumerate (nodes A);
    RETURN (drae-image (the ∘ f) (dra-drae A))
  }) ∈ ⟨Id⟩ nres-rel
  unfolding trans-spec-def drae-image-dra-drae by refine-vcg force
  finally show ?thesis unfolding nres-rel-comp by simp
qed

```

context

```

fixes Ai A
fixes seq bhc hms
fixes S :: ('statei × 'state) set
assumes a: finite (nodes A)
assumes b: is-bounded-hashcode S seq bhc
assumes c: is-valid-def-hm-size TYPE('statei) hms
assumes d: (seq, HOL.eq) ∈ S → S → bool-rel
assumes e: (Ai, A) ∈ ⟨Id, S⟩ drai-dra-rel

```

begin

definition f' **where** $f' \equiv \text{SOME } f'$.

```

  (to-draei-impl seq bhc hms Ai, drae-image (the ∘ f') (dra-drae A)) ∈ ⟨Id,
nat-rel⟩ draei-drae-rel ∧
  dom f' = nodes A ∧ inj-on f' (nodes A)

```

lemma 1: $\exists f'. (\text{to-draei-impl seq bhc hms Ai, drae-image (the ∘ f') (dra-drae$

$A)) \in$
 $\langle Id, nat-rel \rangle$ draei-drae-rel \wedge dom $f' = nodes\ A \wedge inj-on\ f' (nodes\ A)$
using to-draei-impl-refine''[
OF $a\ b\ c\ d\ e,$
unfolded op-set-enumerate-def bind-RES-RETURN-eq,
THEN nres-relD,
THEN RETURN-ref-SPECD]
by force

lemma f' -refine: (to-draei-impl seq $bhc\ hms\ Ai,$ drae-image (the $\circ f'$) (dra-drae
 $A)) \in$
 $\langle Id, nat-rel \rangle$ draei-drae-rel **using** someI-ex[*OF* 1, folded f' -def] **by auto**
lemma f' -dom: dom $f' = nodes\ A$ **using** someI-ex[*OF* 1, folded f' -def] **by auto**
lemma f' -inj: inj-on $f' (nodes\ A)$ **using** someI-ex[*OF* 1, folded f' -def] **by auto**

definition f **where** $f \equiv the \circ f'$
definition g **where** $g = inv-into (nodes\ A)\ f$
lemma inj- f [intro!, simp]: inj-on $f (nodes\ A)$
using f' -inj f' -dom **unfolding** f -def **by** (simp add: inj-on-map-the)
lemma inj- g [intro!, simp]: inj-on $g (f' \text{ nodes } A)$
unfolding g -def **by** (simp add: inj-on-inv-into)

definition rel **where** $rel \equiv \{(f\ p, p) \mid p. p \in nodes\ A\}$
lemma rel-alt-def: $rel = (br\ f\ (\lambda\ p. p \in nodes\ A))^{-1}$
unfolding rel-def **by** (auto simp: in-br-conv)
lemma rel-inv-def: $rel = br\ g\ (\lambda\ k. k \in f' \text{ nodes } A)$
unfolding rel-alt-def g -def **by** (auto simp: in-br-conv)
lemma rel-domain[simp]: Domain rel = $f' \text{ nodes } A$ **unfolding** rel-def **by force**
lemma rel-range[simp]: Range rel = nodes A **unfolding** rel-def **by auto**
lemma [intro!, simp]: bijective rel **unfolding** rel-inv-def **by** (simp add: bijective-alt)

lemma [simp]: Id-on ($f' \text{ nodes } A$) $O\ rel = rel$ **unfolding** rel-def **by auto**
lemma [simp]: rel $O\ Id-on (nodes\ A) = rel$ **unfolding** rel-def **by auto**

lemma [param]: (f, f) $\in Id-on (Range\ rel) \rightarrow Id-on (Domain\ rel)$ **unfolding**
rel-alt-def **by auto**
lemma [param]: (g, g) $\in Id-on (Domain\ rel) \rightarrow Id-on (Range\ rel)$ **unfolding**
rel-inv-def **by auto**
lemma [param]: (id, f) $\in rel \rightarrow Id-on (Domain\ rel)$ **unfolding** rel-alt-def **by**
(auto simp: in-br-conv)
lemma [param]: (f, id) $\in Id-on (Range\ rel) \rightarrow rel$ **unfolding** rel-alt-def **by**
(auto simp: in-br-conv)
lemma [param]: (id, g) $\in Id-on (Domain\ rel) \rightarrow rel$ **unfolding** rel-inv-def **by**
(auto simp: in-br-conv)
lemma [param]: (g, id) $\in rel \rightarrow Id-on (Range\ rel)$ **unfolding** rel-inv-def **by**
(auto simp: in-br-conv)

lemma to-draei-impl-refine':
(to-draei-impl seq $bhc\ hms\ Ai,$ to-draei A) $\in \langle Id-on (alphabet\ A), rel \rangle$ draei-dra-rel

proof –

have 1: (*draei-drae* (*to-draei-impl seq bhc hms Ai*), *id* (*drae-image f (dra-drae A)*)) ∈
 ⟨*Id*, *nat-rel*⟩ *drae-rel* **using** *f'-refine*[*folded f-def*] **by** *parametricity*

have 2: (*draei-drae* (*to-draei-impl seq bhc hms Ai*), *id* (*drae-image f (dra-drae A)*)) ∈
 ⟨*Id-on* (*alphabet A*), *Id-on* (*f ' nodes A*)⟩ *drae-rel*
using 1 **unfolding** *drae-rel-def dra-drae-def drae-image-def* **by** *auto*

have 3: *wft* (*alphabet A*) (*nodes A*) (*transitione (dra-drae A)*)
using *wft-transitions unfolding dra-drae-def drae.sel* **by** *this*

have 4: (*wft* (*alphabet A*) (*f ' nodes A*) (*transitione (drae-image f (dra-drae A))*),
wft (*alphabet A*) (*id ' nodes A*) (*transitione (drae-image id (dra-drae A))*))
 ∈ *bool-rel*
using *dra-rel-eq* **by** *parametricity auto*

have 5: *wft* (*alphabet A*) (*f ' nodes A*) (*transitione (drae-image f (dra-drae A))*)
 using 3 4 **by** *simp*

have (*drae-dra* (*draei-drae (to-draei-impl seq bhc hms Ai)*), *drae-dra* (*id* (*drae-image f (dra-drae A)*))) ∈
 ⟨*Id-on* (*alphabet A*), *Id-on* (*f ' nodes A*)⟩ *dra-rel* **using** 2 5 **by** *parametricity auto*

also have (*drae-dra* (*id* (*drae-image f (dra-drae A)*)), *drae-dra* (*id* (*drae-image id (dra-drae A)*))) ∈
 ⟨*Id-on* (*alphabet A*), *rel*⟩ *dra-rel* **using** *dra-rel-eq 3* **by** *parametricity auto*

also have *drae-dra* (*id* (*drae-image id (dra-drae A)*)) = (*drae-dra* ∘ *dra-drae*)
A **by** *simp*

also have (*...*, *id A*) ∈ ⟨*Id-on* (*alphabet A*), *Id-on* (*nodes A*)⟩ *dra-rel* **by**
parametricity

also have *id A = to-draei A* **unfolding** *to-draei-def* **by** *simp*

finally show *?thesis* **unfolding** *draei-dra-rel-def* **by** *simp*

qed

end

context
begin

interpretation *autoref-syn* **by** *this*

lemma *to-draei-impl-refine*[*autoref-rules*]:
fixes *S* :: (*'statei* × *'state*) *set*
assumes *SIDE-PRECOND* (*finite (nodes A)*)
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE('statei) hms*)
assumes *GEN-OP seq HOL.eq* (*S* → *S* → *bool-rel*)
assumes (*Ai*, *A*) ∈ ⟨*Id*, *S*⟩ *drai-dra-rel*
shows (*to-draei-impl seq bhc hms Ai*,

```

(OP to-draei :: ⟨Id, S⟩ drai-dra-rel →
⟨Id-on (alphabet A), rel Ai A seq bhc hms⟩ draei-dra-rel) $ A) ∈
⟨Id-on (alphabet A), rel Ai A seq bhc hms⟩ draei-dra-rel
using to-draei-impl-refine' assms unfolding autoref-tag-defs by this

```

end

end

35 Nondeterministic Büchi Automata

theory NBA

imports ../Nondeterministic

begin

```

datatype ('label, 'state) nba = nba
  (alphabet: 'label set)
  (initial: 'state set)
  (transition: 'label ⇒ 'state ⇒ 'state set)
  (accepting: 'state pred)

```

```

global-interpretation nba: automaton nba alphabet initial transition accepting
defines path = nba.path and run = nba.run and reachable = nba.reachable
and nodes = nba.nodes
by unfold-locales auto
global-interpretation nba: automaton-run nba alphabet initial transition accept-
ing λ P w r p. infs P (p ## r)
defines language = nba.language
by standard

```

abbreviation target **where** target ≡ nba.target

abbreviation states **where** states ≡ nba.states

abbreviation trace **where** trace ≡ nba.trace

abbreviation successors **where** successors ≡ nba.successors TYPE('label)

instantiation nba :: (type, type) order

begin

```

definition less-eq-nba :: ('a, 'b) nba ⇒ ('a, 'b) nba ⇒ bool where
  A ≤ B ≡ alphabet A ≤ alphabet B ∧ initial A ≤ initial B ∧
    transition A ≤ transition B ∧ accepting A ≤ accepting B
definition less-nba :: ('a, 'b) nba ⇒ ('a, 'b) nba ⇒ bool where
  less-nba A B ≡ A ≤ B ∧ A ≠ B

```

instance **by** (intro-classes) (auto simp: less-eq-nba-def less-nba-def nba.expand)

end

lemma nodes-mono: mono nodes

proof
fix $A B :: ('label, 'state) nba$
assume $1: A \leq B$
have $2: \text{alphabet } A \subseteq \text{alphabet } B$ **using** 1 **unfolding** less-eq-nba-def **by** auto
have $3: \text{initial } A \subseteq \text{initial } B$ **using** 1 **unfolding** less-eq-nba-def **by** auto
have $4: \text{transition } A \ a \ p \subseteq \text{transition } B \ a \ p$ **for** $a \ p$ **using** 1 **unfolding**
 less-eq-nba-def le-fun-def **by** auto
have $5: p \in \text{nodes } B$ **if** $p \in \text{nodes } A$ **for** p **using** $\text{that } 2 \ 3 \ 4$ **by** induct fastforce+
show $\text{nodes } A \subseteq \text{nodes } B$ **using** 5 **by** auto
qed

lemma $\text{language-mono: mono language}$

proof
fix $A B :: ('label, 'state) nba$
assume $1: A \leq B$
have $2: \text{alphabet } A \subseteq \text{alphabet } B$ **using** 1 **unfolding** less-eq-nba-def **by** auto
have $3: \text{initial } A \subseteq \text{initial } B$ **using** 1 **unfolding** less-eq-nba-def **by** auto
have $4: \text{transition } A \ a \ p \subseteq \text{transition } B \ a \ p$ **for** $a \ p$ **using** 1 **unfolding**
 less-eq-nba-def le-fun-def **by** auto
have $5: \text{accepting } A \ p \implies \text{accepting } B \ p$ **for** p **using** 1 **unfolding** less-eq-nba-def
by auto
have $6: \text{run } B \ wr \ p$ **if** $\text{run } A \ wr \ p$ **for** $wr \ p$ **using** $\text{that } 2 \ 4$ **by** coinduct auto
have $7: \text{infs } (\text{accepting } B) \ w$ **if** $\text{infs } (\text{accepting } A) \ w$ **for** w **using** infs-mono
 $\text{that } 5$ **by** metis
show $\text{language } A \subseteq \text{language } B$ **using** $3 \ 6 \ 7$ **by** blast
qed

lemma $\text{simulation-language:}$

assumes $\text{alphabet } A \subseteq \text{alphabet } B$
assumes $\bigwedge p. p \in \text{initial } A \implies \exists q \in \text{initial } B. (p, q) \in R$
assumes $\bigwedge a \ p \ p' \ q. p' \in \text{transition } A \ a \ p \implies (p, q) \in R \implies \exists q' \in \text{transition}$
 $B \ a \ q. (p', q') \in R$
assumes $\bigwedge p \ q. (p, q) \in R \implies \text{accepting } A \ p \implies \text{accepting } B \ q$
shows $\text{language } A \subseteq \text{language } B$

proof

fix w
assume $1: w \in \text{language } A$
obtain $r \ p$ **where** $2: p \in \text{initial } A$ $\text{run } A \ (w \ ||| \ r) \ p$ $\text{infs } (\text{accepting } A) \ (p \ ##$
 $r)$ **using** 1 **by** rule
define P **where** $P \ n \ q \equiv (\text{target } (\text{stake } n \ (w \ ||| \ r)) \ p, q) \in R$ **for** $n \ q$
obtain q **where** $3: q \in \text{initial } B$ $(p, q) \in R$ **using** $\text{assms}(2)$ $2(1)$ **by** auto
obtain ws **where** $4:$
 $\text{run } B \ ws \ q \ \wedge \ i. P \ (0 + i) \ (\text{target } (\text{stake } i \ ws) \ q) \ \wedge \ i. \text{fst } (ws \ !! \ i) = w \ !! \ (0$
 $+ i)$

proof $(\text{rule } nba.\text{invariant-run-index})$

have $\text{stake } k \ (w \ ||| \ r) \ @- \ (w \ !! \ k, \text{target } (\text{stake } (\text{Suc } k) \ (w \ ||| \ r)) \ p) \ ##$

$\text{sdrop } (\text{Suc } k) \ (w \ ||| \ r) = w \ ||| \ r$ **for** k

by $(\text{metis } id\text{-stake-snth-sdrop } snth\text{-szip } sscan\text{-snth } szip\text{-smap-snd } nba.\text{trace-alt-def})$

also have $\text{run } A \ \dots \ p$ **using** $2(2)$ **by** this

```

finally show  $\exists a. (fst\ a \in\ alphabet\ B \wedge\ snd\ a \in\ transition\ B\ (fst\ a)\ q) \wedge$ 
   $P\ (Suc\ n)\ (snd\ a) \wedge\ fst\ a = w\ !!\ n$  if  $P\ n\ q$  for  $n\ q$ 
  using  $assms(1, 3)$  that unfolding  $P$ -def by  $fastforce$ 
  show  $P\ 0\ q$  unfolding  $P$ -def using  $3(2)$  by  $auto$ 
qed  $rule$ 
obtain  $s$  where  $5: ws = w\ |||\ s$  using  $4(3)$  by  $(metis\ add.left-neutral\ eqI-snth$ 
 $snth-smap\ szip-smap)$ 
show  $w \in\ language\ B$ 
proof
  show  $q \in\ initial\ B$  using  $3(1)$  by  $this$ 
  show  $run\ B\ (w\ |||\ s)\ q$  using  $4(1)$  unfolding  $5$  by  $this$ 
  have  $6: (\lambda\ a\ b. (a, b) \in\ R) \leq (\lambda\ a\ b. accepting\ A\ a \longrightarrow accepting\ B\ b)$  using
 $assms(4)$  by  $auto$ 
  have  $7: stream-all2\ (\lambda\ p\ q. (p, q) \in\ R)\ (trace\ (w\ |||\ r)\ p)\ (trace\ (w\ |||\ s)\ q)$ 
using  $4(2)$  unfolding  $P$ -def  $5$  by  $(simp\ add: stream-rel-snth\ del: stake.simps(2))$ 
  have  $8: stream-all2\ (\lambda\ a\ b. accepting\ A\ a \longrightarrow accepting\ B\ b)\ r\ s$ 
using  $stream.rel-mono\ 6\ 7$  unfolding  $nba.trace-alt-def$  by  $auto$ 
  show  $infs\ (accepting\ B)\ (q\ \#\#\ s)$  using  $infs-mono-strong\ 8\ 2(3)$  by  $simp$ 
qed
qed
end

```

36 Nondeterministic Generalized Büchi Automata

theory $NGBA$

imports $../Nondeterministic$

begin

```

datatype  $(label, state)\ ngba = ngba$ 
   $(alphabet: label\ set)$ 
   $(initial: state\ set)$ 
   $(transition: label \Rightarrow state \Rightarrow state\ set)$ 
   $(accepting: state\ pred\ gen)$ 

```

```

global-interpretation  $ngba: automaton\ ngba\ alphabet\ initial\ transition\ accepting$ 
  defines  $path = ngba.path$  and  $run = ngba.run$  and  $reachable = ngba.reachable$ 
and  $nodes = ngba.nodes$ 
  by  $unfold-locales\ auto$ 
global-interpretation  $ngba: automaton-run\ ngba\ alphabet\ initial\ transition\ ac-$ 
 $cepting\ \lambda\ P\ w\ r\ p. gen\ infs\ P\ (p\ \#\#\ r)$ 
  defines  $language = ngba.language$ 
  by  $standard$ 

```

abbreviation $target$ **where** $target \equiv ngba.target$

abbreviation $states$ **where** $states \equiv ngba.states$

abbreviation $trace$ **where** $trace \equiv ngba.trace$

abbreviation $successors$ **where** $successors \equiv ngba.successors\ TYPE(label)$

end

37 Nondeterministic Büchi Automata Combinations

theory *NBA-Combine*
imports *NBA NGBA*
begin

global-interpretation *degeneralization: automaton-degeneralization-run*
 ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting $\lambda P w r p$. gen
infs P (p ## r)
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 fst id
defines *degeneralize = degeneralization.degeneralize*
by (*unfold-locales*) (*auto simp flip: sscan-smap*)

lemmas *degeneralize-language[simp] = degeneralization.degeneralize-language[folded*
NBA.language-def]
lemmas *degeneralize-nodes-finite[iff] = degeneralization.degeneralize-nodes-finite[folded*
NBA.nodes-def]

global-interpretation *intersection: automaton-intersection-run*
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting $\lambda P w r p$. gen
infs P (p ## r)
 $\lambda c_1 c_2. [c_1 \circ fst, c_2 \circ snd]$
defines *intersect' = intersection.product*
by *unfold-locales auto*

lemmas *intersect'-language[simp] = intersection.product-language[folded NGBA.language-def]*
lemmas *intersect'-nodes-finite[intro] = intersection.product-nodes-finite[folded*
NGBA.nodes-def]

global-interpretation *union: automaton-union-run*
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p$. infs P (p
r)
 case-sum
defines *union = union.sum*
by (*unfold-locales*) (*auto simp: comp-def*)

lemmas *union-language* = *union.sum-language*
lemmas *union-nodes-finite* = *union.sum-nodes-finite*

global-interpretation *intersection-list: automaton-intersection-list-run*

nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p. \text{infs } P (p \#\# r)$

ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting $\lambda P w r p. \text{gen infs } P (p \#\# r)$

$\lambda cs. \text{map } (\lambda k ps. (cs ! k) (ps ! k)) [0 ..< \text{length } cs]$

defines *intersect-list'* = *intersection-list.product*

proof *unfold-locales*

fix *cs* :: ('b \Rightarrow bool) *list* **and** *rs* :: 'b *stream list* **and** *w* :: 'a *stream* **and** *ps* :: 'b *list*

assume 1: *length rs* = *length cs* *length ps* = *length cs*

have *gen infs* (*map* ($\lambda k pp. (cs ! k) (pp ! k)$) [*0 ..< length cs*]) (*ps* $\#\#$ *stranspose rs*) \longleftrightarrow

($\forall k < \text{length } cs. \text{infs } (\lambda pp. (cs ! k) (pp ! k)) (ps \#\# \text{stranspose } rs)$)

by (*auto simp: gen-def*)

also have ... \longleftrightarrow ($\forall k < \text{length } cs. \text{infs } (cs ! k) (\text{smap } (\lambda pp. pp ! k) (ps \#\# \text{stranspose } rs))$)

by (*simp add: comp-def*)

also have ... \longleftrightarrow ($\forall k < \text{length } cs. \text{infs } (cs ! k) (rs ! k)$) **using** 1 **by** *simp*

also have ... \longleftrightarrow *list-all* ($\lambda (c, r, p). \text{infs } c (p \#\# r)$) (*cs* || *rs* || *ps*)

using 1 **unfolding** *list-all-length* **by** *simp*

finally show *gen infs* (*map* ($\lambda k ps. (cs ! k) (ps ! k)$) [*0 ..< length cs*]) (*ps* $\#\#$ *stranspose rs*) \longleftrightarrow

list-all ($\lambda (c, r, p). \text{infs } c (p \#\# r)$) (*cs* || *rs* || *ps*) **by** *this*

qed

lemmas *intersect-list'-language*[*simp*] = *intersection-list.product-language*[*folded NGBA.language-def*]

lemmas *intersect-list'-nodes-finite*[*intro*] = *intersection-list.product-nodes-finite*[*folded NGBA.nodes-def*]

global-interpretation *union-list: automaton-union-list-run*

nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p. \text{infs } P (p \#\# r)$

nba nba.alphabet nba.initial nba.transition nba.accepting $\lambda P w r p. \text{infs } P (p \#\# r)$

$\lambda cs (k, p). (cs ! k) p$

defines *union-list* = *union-list.sum*

by (*unfold-locales*) (*auto simp: szip-sconst-smap-fst comp-def*)

lemmas *union-list-language* = *union-list.sum-language*

lemmas *union-list-nodes-finite* = *union-list.sum-nodes-finite*

abbreviation *intersect where* *intersect A B* \equiv *degeneralize (intersect' A B)*

lemma *intersect-language*[simp]: $NBA.language (intersect A B) = NBA.language A \cap NBA.language B$

by *simp*

lemma *intersect-nodes-finite*[intro]:

assumes *finite* (NBA.nodes A) *finite* (NBA.nodes B)

shows *finite* (NBA.nodes (intersect A B))

using *intersect'-nodes-finite* *assms* **by** *simp*

abbreviation *intersect-list* **where** *intersect-list* AA \equiv *degeneralize* (*intersect-list'* AA)

lemma *intersect-list-language*[simp]: $NBA.language (intersect-list AA) = \bigcap (NBA.language \text{'set } AA)$

by *simp*

lemma *intersect-list-nodes-finite*[intro]:

assumes *list-all* (*finite* \circ NBA.nodes) AA

shows *finite* (NBA.nodes (intersect-list AA))

using *intersect-list'-nodes-finite* *assms* **by** *simp*

end

38 Connecting Nondeterministic Büchi Automata to CAVA Automata Structures

theory *NBA-Graphs*

imports

NBA

CAVA-Automata.Automata-Impl

begin

no-notation *build* (**infixr** ## 65)

38.1 Regular Graphs

definition *nba-g* :: ('label, 'state) *nba* \Rightarrow 'state *graph-rec* **where**

nba-g A \equiv (λ *g-V* = UNIV, *g-E* = *E-of-succ* (successors A), *g-V0* = *initial* A λ)

lemma *nba-g-graph*[simp]: *graph* (*nba-g* A) **unfolding** *nba-g-def* *graph-def* **by** *simp*

lemma *nba-g-V0*: *g-V0* (*nba-g* A) = *initial* A **unfolding** *nba-g-def* **by** *simp*

lemma *nba-g-E-rtrancl*: (*g-E* (*nba-g* A))* = {(*p*, *q*). *q* \in *reachable* A *p*}

unfolding *nba-g-def* *graph-rec.simps* *E-of-succ-def*

proof *safe*

show (*p*, *q*) \in {(*p*, *q*). *q* \in *successors* A *p*}* **if** *q* \in *reachable* A *p* **for** *p* *q*

using *that* **by** (*induct*) (*auto* *intro*: *rtrancl-into-rtrancl*)

show *q* \in *reachable* A *p* **if** (*p*, *q*) \in {(*p*, *q*). *q* \in *successors* A *p*}* **for** *p* *q*

using *that* **by** *induct* *auto*

qed

lemma *nba-g-rtrancl-path*: $(g-E (nba-g A))^* = \{(p, target\ r\ p) \mid r\ p.\ NBA.path\ A\ r\ p\}$

unfolding *nba-g-E-rtrancl* **by** *blast*

lemma *nba-g-trancl-path*: $(g-E (nba-g A))^+ = \{(p, target\ r\ p) \mid r\ p.\ NBA.path\ A\ r\ p\ \wedge\ r\ \neq\ []\}$

unfolding *nba-g-def graph-rec.simps E-of-succ-def*

proof *safe*

show $\exists\ r\ p.\ (x, y) = (p, target\ r\ p) \wedge NBA.path\ A\ r\ p\ \wedge\ r\ \neq\ []$

if $(x, y) \in \{(p, q). q \in successors\ A\ p\}^+$ **for** $x\ y$

using *that*

proof *induct*

case (*base y*)

obtain a **where** $1: a \in alphabet\ A\ y \in transition\ A\ a\ x$ **using** *base by auto*

show *?case*

proof (*intro exI conjI*)

show $(x, y) = (x, target\ [(a, y)]\ x)$ **by** *simp*

show $NBA.path\ A\ [(a, y)]\ x$ **using** 1 **by** *auto*

show $[(a, y)] \neq []$ **by** *simp*

qed

next

case (*step y z*)

obtain r **where** $1: y = target\ r\ x\ NBA.path\ A\ r\ x\ r \neq []$ **using** *step(3) by auto*

obtain a **where** $2: a \in alphabet\ A\ z \in transition\ A\ a\ y$ **using** *step(2) by auto*

show *?case*

proof (*intro exI conjI*)

show $(x, z) = (x, target\ (r\ @\ [(a, z)])\ x)$ **by** *simp*

show $NBA.path\ A\ (r\ @\ [(a, z)])\ x$ **using** $1\ 2$ **by** *auto*

show $r\ @\ [(a, z)] \neq []$ **by** *simp*

qed

qed

show $(p, target\ r\ p) \in \{(u, v). v \in successors\ A\ u\}^+$ **if** $NBA.path\ A\ r\ p\ r \neq []$

for $r\ p$

using *that by (induct) (fastforce intro: trancl-into-trancl2)+*

qed

lemma *nba-g-ipath-run*:

assumes *ipath* $(g-E (nba-g A))\ r$

obtains w

where $run\ A\ (w\ ||| smap\ (r\ o\ Suc)\ nats)\ (r\ 0)$

proof –

have $1: \exists\ a \in alphabet\ A.\ r\ (Suc\ i) \in transition\ A\ a\ (r\ i)$ **for** i

using *assms unfolding ipath-def nba-g-def E-of-succ-def* **by** *auto*

obtain wr **where** $2: run\ A\ wr\ (r\ 0) \wedge i.\ target\ (stake\ i\ wr)\ (r\ 0) = r\ i$

proof (*rule nba.invariant-run-index*)

show $\exists\ aq.\ (fst\ aq \in alphabet\ A \wedge snd\ aq \in transition\ A\ (fst\ aq)\ p) \wedge snd\ aq$

```

= r (Suc i) ∧ True
  if p = r i for i p using that 1 by auto
  show r 0 = r 0 by rule
qed auto
have 3: smap (r ∘ Suc) nats = smap snd wr
proof (rule eqI-snth)
  fix i
  have smap (r ∘ Suc) nats !! i = r (Suc i) by simp
  also have ... = target (stake (Suc i) wr) (r 0) unfolding 2(2) by rule
  also have ... = (r 0 ## trace wr (r 0)) !! Suc i by simp
  also have ... = smap snd wr !! i unfolding nba.trace-alt-def by simp
  finally show smap (r ∘ Suc) nats !! i = smap snd wr !! i by this
qed
show ?thesis
proof
  show run A (smap fst wr ||| smap (r ∘ Suc) nats) (r 0) using 2(1) unfolding
3 by auto
qed
qed
lemma nba-g-run-ipath:
  assumes run A (w ||| r) p
  shows ipath (g-E (nba-g A)) (snth (p ## r))
proof
  fix i
  have 1: w !! i ∈ alphabet A r !! i ∈ transition A (w !! i) (target (stake i (w |||
r)) p)
  using assms by (auto dest: nba.run-snth)
  have 2: r !! i ∈ successors A ((p ## r) !! i)
  using 1 unfolding sscan-scons-snth[symmetric] nba.trace-alt-def by auto
  show ((p ## r) !! i, (p ## r) !! Suc i) ∈ g-E (nba-g A)
  using 2 unfolding nba-g-def graph-rec.simps E-of-succ-def by simp
qed

```

38.2 Indexed Generalized Büchi Graphs

definition $nba\text{-igbg} :: ('label, 'state) nba \Rightarrow 'state\ igb\text{-graph-rec}$ **where**
 $nba\text{-igbg } A \equiv graph\text{-rec.extend } (nba\text{-g } A)$
 $(\ | igb\text{-num-acc} = 1, igb\text{-acc} = \lambda p. \text{if accepting } A\ p \text{ then } \{0\} \text{ else } \{\} \ |)$

lemma $acc\text{-run-language}$:

```

assumes igb-graph (nba-igbg A)
shows Ex (igb-graph.is-acc-run (nba-igbg A)) ⟷ language A ≠ {}
proof
  interpret igb-graph nba-igbg A using assms by this
  have [simp]: V0 = g-V0 (nba-g A) E = g-E (nba-g A)
  num-acc = 1 0 ∈ acc p ⟷ accepting A p for p
  unfolding nba-igbg-def graph-rec.defs by simp+
  show language A ≠ {} if run: Ex is-acc-run
proof -

```

```

    obtain r where 1: is-acc-run r using run by rule
    have 2: r 0 ∈ V0 ipath E r is-acc r
      using 1 unfolding is-acc-run-def graph-defs.is-run-def by auto
    obtain w where 3: run A (w ||| smap (r ∘ Suc) nats) (r 0) using nba-g-ipath-run
    2(2) by auto
      have 4: r 0 ## smap (r ∘ Suc) nats = smap r nats by (simp) (metis
    stream.map-comp smap-siterate)
      have 5: infs (accepting A) (r 0 ## smap (r ∘ Suc) nats)
        using 2(3) unfolding infs-infm is-acc-def 4 by simp
      have w ∈ language A
    proof
      show r 0 ∈ initial A using nba-g-V0 2(1) by force
      show run A (w ||| smap (r ∘ Suc) nats) (r 0) using 3 by this
      show infs (accepting A) (r 0 ## smap (r ∘ Suc) nats) using 5 by simp
    qed
    then show ?thesis by auto
  qed
  show Ex is-acc-run if language: language A ≠ {}
  proof -
    obtain w where 1: w ∈ language A using language by auto
    obtain r p where 2: p ∈ initial A run A (w ||| r) p infs (accepting A) (p
  ## r) using 1 by rule
    have is-acc-run (snth (p ## r))
    unfolding is-acc-run-def graph-defs.is-run-def
    proof safe
      show (p ## r) !! 0 ∈ V0 using nba-g-V0 2(1) by force
      show ipath E (snth (p ## r)) using nba-g-run-ipath 2(2) by force
      show is-acc (snth (p ## r)) using 2(3) unfolding infs-infm is-acc-def by
    simp
    qed
    then show ?thesis by auto
  qed
  qed
end

```

39 Relations on Nondeterministic Büchi Automata

```

theory NBA-Refine
imports
  NBA
  ../Transition-Systems/Transition-System-Refine
begin

```

```

definition nba-rel :: ('label1 × 'label2) set ⇒ ('state1 × 'state2) set ⇒
  (('label1, 'state1) nba × ('label2, 'state2) nba) set where
  [to-relAPP]: nba-rel L S ≡ {(A1, A2).
    (alphabet A1, alphabet A2) ∈ ⟨L⟩ set-rel ∧
    (initial A1, initial A2) ∈ ⟨S⟩ set-rel ∧

```

$(\text{transition } A_1, \text{transition } A_2) \in L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel} \wedge$
 $(\text{accepting } A_1, \text{accepting } A_2) \in S \rightarrow \text{bool-rel}\}$

lemma *nba-param*[*param*]:

$(\text{nba}, \text{nba}) \in \langle L \rangle \text{ set-rel} \rightarrow \langle S \rangle \text{ set-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel}) \rightarrow (S \rightarrow$
bool-rel) \rightarrow

$\langle L, S \rangle \text{ nba-rel}$

$(\text{alphabet}, \text{alphabet}) \in \langle L, S \rangle \text{ nba-rel} \rightarrow \langle L \rangle \text{ set-rel}$

$(\text{initial}, \text{initial}) \in \langle L, S \rangle \text{ nba-rel} \rightarrow \langle S \rangle \text{ set-rel}$

$(\text{transition}, \text{transition}) \in \langle L, S \rangle \text{ nba-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel}$

$(\text{accepting}, \text{accepting}) \in \langle L, S \rangle \text{ nba-rel} \rightarrow S \rightarrow \text{bool-rel}$

unfolding *nba-rel-def fun-rel-def* **by** *auto*

lemma *nba-rel-id*[*simp*]: $\langle \text{Id}, \text{Id} \rangle \text{ nba-rel} = \text{Id}$ **unfolding** *nba-rel-def* **using**
nba.expand **by** *auto*

lemma *nba-rel-comp*[*trans*]:

assumes [*param*]: $(A, B) \in \langle L_1, S_1 \rangle \text{ nba-rel}$ $(B, C) \in \langle L_2, S_2 \rangle \text{ nba-rel}$

shows $(A, C) \in \langle L_1 \text{ O } L_2, S_1 \text{ O } S_2 \rangle \text{ nba-rel}$

proof –

have $(\text{accepting } A, \text{accepting } B) \in S_1 \rightarrow \text{bool-rel}$ **by** *parametricity*

also have $(\text{accepting } B, \text{accepting } C) \in S_2 \rightarrow \text{bool-rel}$ **by** *parametricity*

finally have 1: $(\text{accepting } A, \text{accepting } C) \in S_1 \text{ O } S_2 \rightarrow \text{bool-rel}$ **by** *simp*

have $(\text{transition } A, \text{transition } B) \in L_1 \rightarrow S_1 \rightarrow \langle S_1 \rangle \text{ set-rel}$ **by** *parametricity*

also have $(\text{transition } B, \text{transition } C) \in L_2 \rightarrow S_2 \rightarrow \langle S_2 \rangle \text{ set-rel}$ **by** *para-*
metricity

finally have 2: $(\text{transition } A, \text{transition } C) \in L_1 \text{ O } L_2 \rightarrow S_1 \text{ O } S_2 \rightarrow \langle S_1 \rangle$
set-rel $\text{ O } \langle S_2 \rangle \text{ set-rel}$ **by** *simp*

show *?thesis*

unfolding *nba-rel-def mem-Collect-eq prod.case set-rel-comp*

using 1 2

using *nba-param*(2 – 5)[*THEN fun-relD, OF assms*(1)]

using *nba-param*(2 – 5)[*THEN fun-relD, OF assms*(2)]

by *auto*

qed

lemma *nba-rel-converse*[*simp*]: $(\langle L, S \rangle \text{ nba-rel})^{-1} = \langle L^{-1}, S^{-1} \rangle \text{ nba-rel}$

proof –

have 1: $\langle L \rangle \text{ set-rel} = (\langle L^{-1} \rangle \text{ set-rel})^{-1}$ **by** *simp*

have 2: $\langle S \rangle \text{ set-rel} = (\langle S^{-1} \rangle \text{ set-rel})^{-1}$ **by** *simp*

have 3: $L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel} = (L^{-1} \rightarrow S^{-1} \rightarrow \langle S^{-1} \rangle \text{ set-rel})^{-1}$ **by** *simp*

have 4: $S \rightarrow \text{bool-rel} = (S^{-1} \rightarrow \text{bool-rel})^{-1}$ **by** *simp*

show *?thesis* **unfolding** *nba-rel-def* **unfolding** 3 **unfolding** 1 2 4 **by** *fastforce*
qed

lemma *nba-rel-eq*: $(A, A) \in \langle \text{Id-on } (\text{alphabet } A), \text{Id-on } (\text{nodes } A) \rangle \text{ nba-rel}$

unfolding *nba-rel-def* **by** *auto*

lemma *enableds-param*[*param*]: $(\text{nba.enableds}, \text{nba.enableds}) \in \langle L, S \rangle \text{ nba-rel} \rightarrow$
 $S \rightarrow \langle L \times_r S \rangle \text{ set-rel}$

using *nba-param*(2, 4) **unfolding** *nba.enableds-def fun-rel-def set-rel-def* **by**

```

fastforce
lemma paths-param[param]: (nba.paths, nba.paths) ∈ ⟨L, S⟩ nba-rel → S → ⟨⟨L
×r S⟩ list-rel⟩ set-rel
  using enableds-param[param-fo] by parametricity
lemma runs-param[param]: (nba.runs, nba.runs) ∈ ⟨L, S⟩ nba-rel → S → ⟨⟨L
×r S⟩ stream-rel⟩ set-rel
  using enableds-param[param-fo] by parametricity

lemma reachable-param[param]: (reachable, reachable) ∈ ⟨L, S⟩ nba-rel → S →
⟨S⟩ set-rel
proof –
  have 1: reachable A p = (λ wr. target wr p) ‘ nba.paths A p for A :: ('label,
'state) nba and p
    unfolding nba.reachable-alt-def nba.paths-def by auto
    show ?thesis unfolding 1 using enableds-param[param-fo] by parametricity
  qed
lemma nodes-param[param]: (nodes, nodes) ∈ ⟨L, S⟩ nba-rel → ⟨S⟩ set-rel
  unfolding nba.nodes-alt-def Collect-mem-eq by parametricity

lemma language-param[param]: (language, language) ∈ ⟨L, S⟩ nba-rel → ⟨⟨L
stream-rel⟩ set-rel⟩
proof –
  have 1: language A = (⋃ p ∈ initial A. ⋃ wr ∈ nba.runs A p.
    if infs (accepting A) (p ## smap snd wr) then {smap fst wr} else {})
    for A :: ('label, 'state) nba
    unfolding nba.language-def nba.runs-def image-def
    by (auto iff: split-szip-ex simp del: alw-smap)
  show ?thesis unfolding 1 using enableds-param[param-fo] by parametricity
qed
end

```

40 Implementation of Nondeterministic Büchi Automata

```

theory NBA-Implement
imports
  NBA-Refine
  ../Basic/Implement
begin

```

```

  consts i-nba-scheme :: interface ⇒ interface ⇒ interface

```

```

  context
  begin

```

```

    interpretation autoref-syn by this
  end

```

lemma *nba-scheme-itype*[*autoref-itype*]:
 $nba ::_i \langle L \rangle_i \text{ i-set} \rightarrow_i \langle S \rangle_i \text{ i-set} \rightarrow_i (L \rightarrow_i S \rightarrow_i \langle S \rangle_i \text{ i-set}) \rightarrow_i \langle S \rangle_i \text{ i-set} \rightarrow_i$
 $\langle L, S \rangle_i \text{ i-nba-scheme}$
 $alphabet ::_i \langle L, S \rangle_i \text{ i-nba-scheme} \rightarrow_i \langle L \rangle_i \text{ i-set}$
 $initial ::_i \langle L, S \rangle_i \text{ i-nba-scheme} \rightarrow_i \langle S \rangle_i \text{ i-set}$
 $transition ::_i \langle L, S \rangle_i \text{ i-nba-scheme} \rightarrow_i L \rightarrow_i S \rightarrow_i \langle S \rangle_i \text{ i-set}$
 $accepting ::_i \langle L, S \rangle_i \text{ i-nba-scheme} \rightarrow_i \langle S \rangle_i \text{ i-set}$
by *auto*

end

datatype (*'label*, *'state*) *nbai* = *nbai*
(*alphabeti*: *'label list*)
(*initiali*: *'state list*)
(*transitioni*: *'label* \Rightarrow *'state* \Rightarrow *'state list*)
(*acceptingi*: *'state* \Rightarrow *bool*)

definition *nbai-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow
((*'label*₁, *'state*₁) *nbai* \times (*'label*₂, *'state*₂) *nbai*) *set* **where**
[*to-relAPP*]: *nbai-rel* *L S* \equiv {(*A*₁, *A*₂).
(*alphabeti* *A*₁, *alphabeti* *A*₂) \in $\langle L \rangle$ *list-rel* \wedge
(*initiali* *A*₁, *initiali* *A*₂) \in $\langle S \rangle$ *list-rel* \wedge
(*transitioni* *A*₁, *transitioni* *A*₂) \in $L \rightarrow S \rightarrow \langle S \rangle$ *list-rel* \wedge
(*acceptingi* *A*₁, *acceptingi* *A*₂) \in $S \rightarrow \text{bool-rel}$ }

lemma *nbai-param*[*param*, *autoref-rules*]:
(*nbai*, *nbai*) \in $\langle L \rangle$ *list-rel* \rightarrow $\langle S \rangle$ *list-rel* \rightarrow ($L \rightarrow S \rightarrow \langle S \rangle$ *list-rel*) \rightarrow
($S \rightarrow \text{bool-rel}$) \rightarrow $\langle L, S \rangle$ *nbai-rel*
(*alphabeti*, *alphabeti*) \in $\langle L, S \rangle$ *nbai-rel* \rightarrow $\langle L \rangle$ *list-rel*
(*initiali*, *initiali*) \in $\langle L, S \rangle$ *nbai-rel* \rightarrow $\langle S \rangle$ *list-rel*
(*transitioni*, *transitioni*) \in $\langle L, S \rangle$ *nbai-rel* \rightarrow $L \rightarrow S \rightarrow \langle S \rangle$ *list-rel*
(*acceptingi*, *acceptingi*) \in $\langle L, S \rangle$ *nbai-rel* \rightarrow ($S \rightarrow \text{bool-rel}$)
unfolding *nbai-rel-def fun-rel-def* **by** *auto*

definition *nbai-nba-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow
((*'label*₁, *'state*₁) *nbai* \times (*'label*₂, *'state*₂) *nba*) *set* **where**
[*to-relAPP*]: *nbai-nba-rel* *L S* \equiv {(*A*₁, *A*₂).
(*alphabeti* *A*₁, *alphabet* *A*₂) \in $\langle L \rangle$ *list-set-rel* \wedge
(*initiali* *A*₁, *initial* *A*₂) \in $\langle S \rangle$ *list-set-rel* \wedge
(*transitioni* *A*₁, *transition* *A*₂) \in $L \rightarrow S \rightarrow \langle S \rangle$ *list-set-rel* \wedge
(*acceptingi* *A*₁, *acceptingi* *A*₂) \in $S \rightarrow \text{bool-rel}$ }

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nbai-nba-rel i-nba-scheme*]

lemma *nbai-nba-param*[*param*, *autoref-rules*]:
(*nbai*, *nba*) \in $\langle L \rangle$ *list-set-rel* \rightarrow $\langle S \rangle$ *list-set-rel* \rightarrow ($L \rightarrow S \rightarrow \langle S \rangle$ *list-set-rel*) \rightarrow
($S \rightarrow \text{bool-rel}$) \rightarrow $\langle L, S \rangle$ *nbai-nba-rel*
(*alphabeti*, *alphabet*) \in $\langle L, S \rangle$ *nbai-nba-rel* \rightarrow $\langle L \rangle$ *list-set-rel*

$(initiali, initial) \in \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \langle S \rangle \text{ list-set-rel}$
 $(transitioni, transition) \in \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel}$
 $(acceptingi, accepting) \in \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow S \rightarrow \text{bool-rel}$
unfolding *nbai-nba-rel-def fun-rel-def* **by** *auto*

definition *nbai-nba* :: ('label, 'state) *nbai* \Rightarrow ('label, 'state) *nba* **where**
 $\text{nbai-nba } A \equiv \text{nba } (\text{set } (\text{alphabeti } A)) (\text{set } (\text{initiali } A)) (\lambda a p. \text{set } (\text{transitioni } A a p)) (\text{acceptingi } A)$

definition *nbai-invar* :: ('label, 'state) *nbai* \Rightarrow *bool* **where**
 $\text{nbai-invar } A \equiv \text{distinct } (\text{alphabeti } A) \wedge \text{distinct } (\text{initiali } A) \wedge (\forall a p. \text{distinct } (\text{transitioni } A a p))$

lemma *nbai-nba-id-param*[*param*]: (*nbai-nba, id*) $\in \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \langle L, S \rangle \text{ nba-rel}$

proof

fix *Ai A*

assume *1*: (*Ai, A*) $\in \langle L, S \rangle \text{ nbai-nba-rel}$

have *2*: $\text{nbai-nba } Ai = \text{nba } (\text{set } (\text{alphabeti } Ai)) (\text{set } (\text{initiali } Ai))$

$(\lambda a p. \text{set } (\text{transitioni } Ai a p)) (\text{acceptingi } Ai)$ **unfolding** *nbai-nba-def* **by**

rule

have *3*: $\text{id } A = \text{nba } (\text{id } (\text{alphabet } A)) (\text{id } (\text{initial } A))$

$(\lambda a p. \text{id } (\text{transition } A a p)) (\text{accepting } A)$ **by** *simp*

show (*nbai-nba Ai, id A*) $\in \langle L, S \rangle \text{ nba-rel}$ **unfolding** *2 3* **using** *1* **by** *parametricity*

qed

lemma *nbai-nba-br*: $\langle Id, Id \rangle \text{ nbai-nba-rel} = \text{br } \text{nbai-nba } \text{nbai-invar}$

proof *safe*

show (*A, B*) $\in \langle Id, Id \rangle \text{ nbai-nba-rel}$ **if** (*A, B*) $\in \text{br } \text{nbai-nba } \text{nbai-invar}$

for *A* **and** *B* :: ('a, 'b) *nba*

using *that* **unfolding** *nbai-nba-rel-def nbai-nba-def nbai-invar-def*

by (*auto simp: in-br-conv list-set-rel-def*)

show (*A, B*) $\in \text{br } \text{nbai-nba } \text{nbai-invar}$ **if** (*A, B*) $\in \langle Id, Id \rangle \text{ nbai-nba-rel}$

for *A* **and** *B* :: ('a, 'b) *nba*

proof –

have *1*: (*nbai-nba A, id B*) $\in \langle Id, Id \rangle \text{ nba-rel}$ **using** *that* **by** *parametricity*

have *2*: *nbai-invar A*

using *nbai-nba-param(2 - 5)[param-fo, OF that]*

by (*auto simp: in-br-conv list-set-rel-def nbai-invar-def*)

show *?thesis* **using** *1 2* **unfolding** *in-br-conv* **by** *auto*

qed

qed

end

41 Algorithms on Nondeterministic Büchi Automata

theory *NBA-Algorithms*

imports

NBA-Graphs
NBA-Implement
DFS-Framework.Reachable-Nodes
Gabow-SCC.Gabow-GBG-Code

begin

41.1 Miscellaneous Amendments

lemma (in *igb-fr-graph*) *acc-run-lasso-prpl*: $Ex\ is\ acc\ run \implies Ex\ is\ lasso\ prpl$
using *accepted-lasso is-lasso-prpl-of-lasso* **by** *blast*

lemma (in *igb-fr-graph*) *lasso-prpl-acc-run-iff*: $Ex\ is\ lasso\ prpl \iff Ex\ is\ acc\ run$
using *acc-run-lasso-prpl lasso-prpl-acc-run* **by** *auto*

lemma [*autoref-rel-intf*]: *REL-INTF igbg-impl-rel-ext i-igbg* **by** (rule *REL-INTFI*)

41.2 Operations

definition *op-language-empty* **where** [*simp*]: *op-language-empty A* \equiv *language A = {}*

lemmas [*autoref-op-pat*] = *op-language-empty-def*[*symmetric*]

41.3 Implementations

context

begin

interpretation *autoref-syn* **by** *this*

lemma *nba-g-ahs*: $nba\ g\ A = (\mid g\ V = UNIV, g\ E = E\ of\ succ\ (\lambda\ p.\ CAST\ ((\bigcup\ a \in\ alphabet\ A.\ transition\ A\ a\ p :: \langle S \rangle\ list\ set\ rel) :: \langle S \rangle\ ahs\ rel\ bhc)),\ g\ V0 = initial\ A \mid)$

unfolding *nba-g-def nba.successors-alt-def CAST-def id-apply autoref-tag-defs*
by *rule*

schematic-goal *nbai-gi*:

notes [*autoref-ga-rules*] = *map2set-to-list*

fixes $S :: ('statei \times 'state)\ set$

assumes [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhc*

assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE('statei) hms*

assumes [*autoref-rules*]: $(seq, HOL.eq) \in S \rightarrow S \rightarrow bool\ rel$

assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle\ nbai\ nba\ rel$

shows $(?f :: ?'a, RETURN\ (nba\ g\ A)) \in ?A$

unfolding *nba-g-ahs*[**where** $S = S$ **and** $bhc = bhc$] **by** (*autoref-monadic* (*plain*))

concrete-definition *nbai-gi* **uses** *nbai-gi*

lemma *nbai-gi-refine*[*autoref-rules*]:

fixes $S :: ('statei \times 'state)\ set$

assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode S seq bhc*)

assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE('statei) hms*)

assumes *GEN-OP seq HOL.eq* ($S \rightarrow S \rightarrow \text{bool-rel}$)
shows (*NBA-Algorithms.nbai-gi seq bhc hms, nba-g*) \in
 $\langle L, S \rangle \text{nbai-nba-rel} \rightarrow \langle \text{unit-rel}, S \rangle \text{g-impl-rel-ext}$
using *nbai-gi.refine[THEN RETURN-nres-relD]* *assms unfolding autoref-tag-defs*
by *blast*

schematic-goal *nba-nodes*:

fixes $S :: ('statei \times 'state) \text{set}$
assumes [*simp*]: *finite* ($(g-E (nba-g A))^* \text{“ } g-V0 (nba-g A)$)
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $S \text{ seq bhc}$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('statei) \text{ hms}$
assumes [*autoref-rules*]: $(\text{seq}, \text{HOL.eq}) \in S \rightarrow S \rightarrow \text{bool-rel}$
assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle \text{nbai-nba-rel}$
shows ($?f :: ?'a, \text{op-reachable} (nba-g A) \in ?R$ **by** *autoref*

concrete-definition *nba-nodes uses nba-nodes*

lemma *nba-nodes-refine[autoref-rules]*:

fixes $S :: ('statei \times 'state) \text{set}$
assumes *SIDE-PRECOND* (*finite* ($\text{nodes } A$))
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* $S \text{ seq bhc}$)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE('statei) \text{ hms}$)
assumes *GEN-OP seq HOL.eq* ($S \rightarrow S \rightarrow \text{bool-rel}$)
assumes $(Ai, A) \in \langle L, S \rangle \text{nbai-nba-rel}$
shows (*NBA-Algorithms.nba-nodes seq bhc hms Ai,*
 $(\text{OP nodes} :: \langle L, S \rangle \text{nbai-nba-rel} \rightarrow \langle S \rangle \text{ahs-rel bhc}) \$ A \in \langle S \rangle \text{ahs-rel bhc}$

proof –

have 1: $\text{nodes } A = \text{op-reachable} (nba-g A)$ **by** (*auto simp: nba-g-V0 nba-g-E-rtrancl*)

have 2: *finite* ($(g-E (nba-g A))^* \text{“ } g-V0 (nba-g A)$) **using** *assms(1) unfolding*

1 **by** *simp*

show *?thesis using nba-nodes.refine assms 2 unfolding autoref-tag-defs 1*

by *blast*

qed

lemma *nba-igbg-ahs*: $\text{nba-igbg } A = (\text{ } g-V = \text{UNIV}, g-E = \text{E-of-succ } (\lambda p. \text{CAST}$

$(\bigcup a \in \text{alphabet } A. \text{transition } A a p :: \langle S \rangle \text{list-set-rel}) :: \langle S \rangle \text{ahs-rel bhc}),$
 $g-V0 = \text{initial } A,$

$\text{igbg-num-acc} = 1, \text{igbg-acc} = \lambda p. \text{if accepting } A p \text{ then } \{0\} \text{ else } \{\}$)

unfolding *nba-g-def nba-igbg-def nba.successors-alt-def CAST-def id-apply*
autoref-tag-defs

unfolding *graph-rec.defs*

by *simp*

schematic-goal *nbai-igbgi*:

notes [*autoref-ga-rules*] = *map2set-to-list*

fixes $S :: ('statei \times 'state) \text{set}$

assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $S \text{ seq bhc}$

assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('statei) \text{ hms}$

assumes [*autoref-rules*]: $(\text{seq}, \text{HOL.eq}) \in S \rightarrow S \rightarrow \text{bool-rel}$

assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle \text{nbai-nba-rel}$

shows ($?f :: ?'a, \text{RETURN } (nba\text{-igbg } A) \in ?A$
unfolding $nba\text{-igbg}\text{-ahs}[\text{where } S = S \text{ and } bhc = bhc]$ **by** (*autoref-monadic*
(plain))

concrete-definition $nba\text{-igbgi}$ **uses** $nba\text{-igbgi}$
lemma $nba\text{-igbgi}\text{-refine}[\text{autoref-rules}]$:
fixes $S :: ('statei \times 'state)$ *set*
assumes $\text{SIDE-GEN-ALGO } (is\text{-bounded-hashcode } S \text{ seq } bhc)$
assumes $\text{SIDE-GEN-ALGO } (is\text{-valid-def-hm-size } \text{TYPE}('statei) \text{ hms})$
assumes $\text{GEN-OP seq HOL.eq } (S \rightarrow S \rightarrow \text{bool-rel})$
shows ($\text{NBA-Algorithms.nbai-igbgi seq bhc hms, nba-igbg} \in$
 $\langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \text{igbg-impl-rel-ext unit-rel } S$
using $nba\text{-igbgi.refine}[\text{THEN RETURN-nres-relD}]$ *assms* **unfolding** *au-*
toref-tag-defs **by** *blast*

schematic-goal $nba\text{-language-empty}$:
fixes $S :: ('statei \times 'state)$ *set*
assumes [*simp*]: $\text{igb-fr-graph } (nba\text{-igbg } A)$
assumes [*autoref-ga-rules*]: $is\text{-bounded-hashcode } S \text{ seq bhc}$
assumes [*autoref-ga-rules*]: $is\text{-valid-def-hm-size } \text{TYPE}('statei) \text{ hms}$
assumes [*autoref-rules*]: $(\text{seq, HOL.eq}) \in S \rightarrow S \rightarrow \text{bool-rel}$
assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle \text{ nbai-nba-rel}$
shows ($?f :: ?'a, \text{do } \{ r \leftarrow \text{op-find-lasso-spec } (nba\text{-igbg } A); \text{RETURN } (r =$
 $\text{None}) \} \in ?A$
by (*autoref-monadic (plain)*)

concrete-definition $nba\text{-language-empty}$ **uses** $nba\text{-language-empty}$
lemma $nba\text{-language-empty}\text{-refine}[\text{autoref-rules}]$:
fixes $S :: ('statei \times 'state)$ *set*
assumes $\text{SIDE-PRECOND } (finite (nodes A))$
assumes $\text{SIDE-GEN-ALGO } (is\text{-bounded-hashcode } S \text{ seq } bhc)$
assumes $\text{SIDE-GEN-ALGO } (is\text{-valid-def-hm-size } \text{TYPE}('statei) \text{ hms})$
assumes $\text{GEN-OP seq HOL.eq } (S \rightarrow S \rightarrow \text{bool-rel})$
assumes $(Ai, A) \in \langle L, S \rangle \text{ nbai-nba-rel}$
shows ($\text{NBA-Algorithms.nba-language-empty seq bhc hms } Ai,$
 $(\text{OP op-language-empty} :: \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \text{bool-rel}) \$ A) \in \text{bool-rel}$

proof –
have 1: $nodes A = \text{op-reachable } (nba\text{-g } A)$ **by** (*auto simp: nba-g-V0 nba-g-E-rtrancl*)
have 2: $finite ((g\text{-E } (nba\text{-g } A))^* \text{ “} g\text{-V0 } (nba\text{-g } A))$ **using** *assms(1)* **unfolding**
1 **by** *simp*
interpret $\text{igb-fr-graph } nba\text{-igbg } A$
using 2 **unfolding** $nba\text{-igbg-def nba-g-def graph-rec.defs}$ **by** *unfold-locales*
auto
have ($\text{RETURN } (\text{NBA-Algorithms.nba-language-empty seq bhc hms } Ai),$
 $\text{do } \{ r \leftarrow \text{find-lasso-spec}; \text{RETURN } (r = \text{None}) \} \in \langle \text{bool-rel} \rangle \text{ nres-rel}$
using $nba\text{-language-empty.refine assms igb-fr-graph-axioms}$ **by** *simp*
also have ($\text{do } \{ r \leftarrow \text{find-lasso-spec}; \text{RETURN } (r = \text{None}) \},$
 $\text{RETURN } (\neg \text{Ex is-lasso-prpl}) \in \langle \text{bool-rel} \rangle \text{ nres-rel}$
unfolding $\text{find-lasso-spec-def}$ **by** (*refine-vcg (auto split: option.splits)*)
finally have $\text{NBA-Algorithms.nba-language-empty seq bhc hms } Ai \longleftrightarrow \neg \text{Ex}$
is-lasso-prpl

```

    unfolding nres-rel-comp using RETURN-nres-relD by force
    also have ...  $\longleftrightarrow \neg \exists x$  is-acc-run using lasso-prpl-acc-run-iff by auto
    also have ...  $\longleftrightarrow$  language  $A = \{\}$  using acc-run-language-is-igb-graph by
auto
    finally show ?thesis by simp
  qed

end

end

```

42 Explicit Nondeterministic Büchi Automata

```

theory NBA-Explicit
imports NBA-Algorithms
begin

```

```

datatype ('label, 'state) nbae = nbae
  (alphabet: 'label set)
  (initiale: 'state set)
  (transitione: ('state  $\times$  'label  $\times$  'state) set)
  (acceptinge: 'state set)

```

definition *nbae-rel* **where**

```

[to-relAPP]: nbae-rel L S  $\equiv$   $\{(A_1, A_2).$ 
  (alphabet A1, alphabet A2)  $\in$   $\langle L \rangle$  set-rel  $\wedge$ 
  (initiale A1, initiale A2)  $\in$   $\langle S \rangle$  set-rel  $\wedge$ 
  (transitione A1, transitione A2)  $\in$   $\langle S \times_r L \times_r S \rangle$  set-rel  $\wedge$ 
  (acceptinge A1, acceptinge A2)  $\in$   $\langle S \rangle$  set-rel $\}$ 

```

lemma *nbae-param*[*param*, *autoref-rules*]:

```

(nbae, nbae)  $\in$   $\langle L \rangle$  set-rel  $\rightarrow$   $\langle S \rangle$  set-rel  $\rightarrow$   $\langle S \times_r L \times_r S \rangle$  set-rel  $\rightarrow$ 
 $\langle S \rangle$  set-rel  $\rightarrow$   $\langle L, S \rangle$  nbae-rel
(alphabet, alphabet)  $\in$   $\langle L, S \rangle$  nbae-rel  $\rightarrow$   $\langle L \rangle$  set-rel
(initiale, initiale)  $\in$   $\langle L, S \rangle$  nbae-rel  $\rightarrow$   $\langle S \rangle$  set-rel
(transitione, transitione)  $\in$   $\langle L, S \rangle$  nbae-rel  $\rightarrow$   $\langle S \times_r L \times_r S \rangle$  set-rel
(acceptinge, acceptinge)  $\in$   $\langle L, S \rangle$  nbae-rel  $\rightarrow$   $\langle S \rangle$  set-rel

```

unfolding *nbae-rel-def* **by** *auto*

lemma *nbae-rel-id*[*simp*]: $\langle Id, Id \rangle$ nbae-rel = *Id* **unfolding** *nbae-rel-def* **using** *nbae.expand* **by** *auto*

lemma *nbae-rel-comp*[*simp*]: $\langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle$ nbae-rel = $\langle L_1, S_1 \rangle$ nbae-rel *O* $\langle L_2, S_2 \rangle$ nbae-rel

proof *safe*

fix *A B*

assume *1*: $(A, B) \in \langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle$ nbae-rel

obtain *a b c d* **where** *2*:

```

(alphabet A, a)  $\in$   $\langle L_1 \rangle$  set-rel (a, alphabet B)  $\in$   $\langle L_2 \rangle$  set-rel
(initiale A, b)  $\in$   $\langle S_1 \rangle$  set-rel (b, initiale B)  $\in$   $\langle S_2 \rangle$  set-rel

```

```

    (transitione A, c) ∈ ⟨S1 ×r L1 ×r S1⟩ set-rel (c, transitione B) ∈ ⟨S2 ×r L2
    ×r S2⟩ set-rel
    (acceptinge A, d) ∈ ⟨S1⟩ set-rel (d, acceptinge B) ∈ ⟨S2⟩ set-rel
    using 1 unfolding nbae-rel-def prod-rel-compp set-rel-compp by auto
    show (A, B) ∈ ⟨L1, S1⟩ nbae-rel O ⟨L2, S2⟩ nbae-rel
  proof
    show (A, nbae a b c d) ∈ ⟨L1, S1⟩ nbae-rel using 2 unfolding nbae-rel-def
  by auto
    show (nbae a b c d, B) ∈ ⟨L2, S2⟩ nbae-rel using 2 unfolding nbae-rel-def
  by auto
  qed
next
  show (A, C) ∈ ⟨L1 O L2, S1 O S2⟩ nbae-rel
  if (A, B) ∈ ⟨L1, S1⟩ nbae-rel (B, C) ∈ ⟨L2, S2⟩ nbae-rel for A B C
  using that unfolding nbae-rel-def prod-rel-compp set-rel-compp by auto
qed

```

consts *i-nbae-scheme* :: *interface* ⇒ *interface* ⇒ *interface*

context
begin

interpretation *autoref-syn* **by** *this*

lemma *nbae-scheme-itype*[*autoref-itype*]:
nbae ::_i ⟨L⟩_i *i-set* →_i ⟨S⟩_i *i-set* →_i ⟨⟨S, ⟨L, S⟩_i *i-prod*⟩_i *i-prod*⟩_i *i-set* →_i ⟨S⟩_i
i-set →_i
 ⟨L, S⟩_i *i-nbae-scheme*
alphabetei ::_i ⟨L, S⟩_i *i-nbae-scheme* →_i ⟨L⟩_i *i-set*
initialei ::_i ⟨L, S⟩_i *i-nbae-scheme* →_i ⟨S⟩_i *i-set*
transitionei ::_i ⟨L, S⟩_i *i-nbae-scheme* →_i ⟨⟨S, ⟨L, S⟩_i *i-prod*⟩_i *i-prod*⟩_i *i-set*
acceptingei ::_i ⟨L, S⟩_i *i-nbae-scheme* →_i ⟨S⟩_i *i-set*
by *auto*

end

datatype (*'label*, *'state*) *nbaei* = *nbaei*
 (*alphabetei*: *'label list*)
 (*initialei*: *'state list*)
 (*transitionei*: (*'state* × *'label* × *'state*) *list*)
 (*acceptingei*: *'state list*)

definition *nbaei-rel* **where**

[*to-relAPP*]: *nbaei-rel* L S ≡ {(A₁, A₂).
 (*alphabetei* A₁, *alphabetei* A₂) ∈ ⟨L⟩ *list-rel* ∧
 (*initialei* A₁, *initialei* A₂) ∈ ⟨S⟩ *list-rel* ∧
 (*transitionei* A₁, *transitionei* A₂) ∈ ⟨S ×_r L ×_r S⟩ *list-rel* ∧
 (*acceptingei* A₁, *acceptingei* A₂) ∈ ⟨S⟩ *list-rel*}

lemma *nbaei-param*[*param, autoref-rules*]:

(*nbaei, nbaei*) $\in \langle L \rangle$ *list-rel* $\rightarrow \langle S \rangle$ *list-rel* $\rightarrow \langle S \times_r L \times_r S \rangle$ *list-rel* \rightarrow
 $\langle S \rangle$ *list-rel* $\rightarrow \langle L, S \rangle$ *nbaei-rel*
(*alphabetei, alphabetei*) $\in \langle L, S \rangle$ *nbaei-rel* $\rightarrow \langle L \rangle$ *list-rel*
(*initialei, initialei*) $\in \langle L, S \rangle$ *nbaei-rel* $\rightarrow \langle S \rangle$ *list-rel*
(*transitionei, transitionei*) $\in \langle L, S \rangle$ *nbaei-rel* $\rightarrow \langle S \times_r L \times_r S \rangle$ *list-rel*
(*acceptingei, acceptingei*) $\in \langle L, S \rangle$ *nbaei-rel* $\rightarrow \langle S \rangle$ *list-rel*
unfolding *nbaei-rel-def* **by auto**

definition *nbaei-nbae-rel* **where**

[*to-relAPP*]: *nbaei-nbae-rel* $L S \equiv \{(A_1, A_2).$
(*alphabetei* A_1 , *alphabetete* A_2) $\in \langle L \rangle$ *list-set-rel* \wedge
(*initialei* A_1 , *initiale* A_2) $\in \langle S \rangle$ *list-set-rel* \wedge
(*transitionei* A_1 , *transitione* A_2) $\in \langle S \times_r L \times_r S \rangle$ *list-set-rel* \wedge
(*acceptingei* A_1 , *acceptinge* A_2) $\in \langle S \rangle$ *list-set-rel*}

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of nbaei-nbae-rel i-nbae-scheme*]

lemma *nbaei-nbae-param*[*param, autoref-rules*]:

(*nbaei, nbae*) $\in \langle L \rangle$ *list-set-rel* $\rightarrow \langle S \rangle$ *list-set-rel* $\rightarrow \langle S \times_r L \times_r S \rangle$ *list-set-rel*
 \rightarrow
 $\langle S \rangle$ *list-set-rel* $\rightarrow \langle L, S \rangle$ *nbaei-nbae-rel*
(*alphabetei, alphabetete*) $\in \langle L, S \rangle$ *nbaei-nbae-rel* $\rightarrow \langle L \rangle$ *list-set-rel*
(*initialei, initiale*) $\in \langle L, S \rangle$ *nbaei-nbae-rel* $\rightarrow \langle S \rangle$ *list-set-rel*
(*transitionei, transitione*) $\in \langle L, S \rangle$ *nbaei-nbae-rel* $\rightarrow \langle S \times_r L \times_r S \rangle$ *list-set-rel*
(*acceptingei, acceptinge*) $\in \langle L, S \rangle$ *nbaei-nbae-rel* $\rightarrow \langle S \rangle$ *list-set-rel*
unfolding *nbaei-nbae-rel-def* **by auto**

definition *nbaei-nbae* **where**

nbaei-nbae $A \equiv$ *nbae* (*set* (*alphabetei* A)) (*set* (*initialei* A))
(*set* (*transitionei* A)) (*set* (*acceptingei* A))

lemma *nbaei-nbae-id-param*[*param*]: (*nbaei-nbae, id*) $\in \langle L, S \rangle$ *nbaei-nbae-rel* \rightarrow
 $\langle L, S \rangle$ *nbae-rel*

proof

fix $A_i A$

assume 1: (A_i, A) $\in \langle L, S \rangle$ *nbaei-nbae-rel*

have 2: *nbaei-nbae* $A_i =$ *nbae* (*set* (*alphabetei* A_i)) (*set* (*initialei* A_i))

(*set* (*transitionei* A_i)) (*set* (*acceptingei* A_i)) **unfolding** *nbaei-nbae-def* **by rule**

have 3: *id* $A =$ *nbae* (*id* (*alphabetete* A)) (*id* (*initiale* A))

(*id* (*transitione* A)) (*id* (*acceptinge* A)) **by simp**

show (*nbaei-nbae* $A_i, id A$) $\in \langle L, S \rangle$ *nbae-rel* **unfolding** 2 3 **using** 1 **by**
parametricity

qed

abbreviation *transitions* $L S s \equiv \bigcup a \in L. \bigcup p \in S. \{p\} \times \{a\} \times s a p$

abbreviation *succs* $T a p \equiv (T \text{ `` } \{p\} \text{ '' } \{a\}$

definition *nba-nbae* **where** $nba-nbae\ A \equiv nbae\ (\text{alphabet } A)\ (\text{initial } A)$
 $(\text{transitions } (\text{alphabet } A)\ (\text{nodes } A)\ (\text{transition } A))\ (\text{Set.filter } (\text{accepting } A))$
 $(\text{nodes } A)$

definition *nbae-nba* **where** $nbae-nba\ A \equiv nba\ (\text{alphabet } A)\ (\text{initial } A)$
 $(\text{succs } (\text{transition } A))\ (\lambda p. p \in \text{accepting } A)$

lemma *nba-nbae-param*[*param*]: $(nba-nbae, nba-nbae) \in \langle L, S \rangle\ nba-rel \rightarrow \langle L, S \rangle\ nbae-rel$

unfolding *nba-nbae-def* **by** *parametricity*

lemma *nbae-nba-param*[*param*]:

assumes *bijjective L* *bijjective S*

shows $(nbae-nba, nbae-nba) \in \langle L, S \rangle\ nbae-rel \rightarrow \langle L, S \rangle\ nba-rel$

using *assms* *assms(2)*[*unfolded bijjective-alt*] **unfolding** *nbae-nba-def* **by** *parametricity auto*

lemma *nbae-nba-nba-nbae-param*[*param*]:

$((nbae-nba \circ nba-nbae)\ A, id\ A) \in \langle Id-on\ (\text{alphabet } A), Id-on\ (\text{nodes } A) \rangle\ nba-rel$

proof –

have $(nbae-nba \circ nba-nbae)\ A = nba\ (\text{alphabet } A)\ (\text{initial } A)$

$(\text{succs } (\text{transitions } (\text{alphabet } A)\ (\text{nodes } A)\ (\text{transition } A)))\ (\lambda p. p \in \text{Set.filter } (\text{accepting } A)\ (\text{nodes } A))$

unfolding *nbae-nba-def* *nba-nbae-def* **by** *simp*

also have $(\dots, nba\ (\text{alphabet } A)\ (\text{initial } A)\ (\text{transition } A)\ (\text{accepting } A)) \in$

$\langle Id-on\ (\text{alphabet } A), Id-on\ (\text{nodes } A) \rangle\ nba-rel$

using *nba-rel-eq* **by** *parametricity auto*

also have $nba\ (\text{alphabet } A)\ (\text{initial } A)\ (\text{transition } A)\ (\text{accepting } A) = id\ A$ **by** *simp*

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

qed

definition *nbaei-nba-rel* **where**

$[to-relAPP]: nbaei-nba-rel\ L\ S \equiv \{(Ae, A). (nbae-nba\ (nbaei-nbae\ Ae), A) \in \langle L, S \rangle\ nba-rel\}$

lemma *nbaei-nba-id*[*param*]: $(nbae-nba \circ nbaei-nbae, id) \in \langle L, S \rangle\ nbaei-nba-rel \rightarrow \langle L, S \rangle\ nba-rel$

unfolding *nbaei-nba-rel-def* **by** *auto*

schematic-goal *nbae-nba-impl*:

assumes [*autoref-rules*]: $(leq, HOL.eq) \in L \rightarrow L \rightarrow bool-rel$

assumes [*autoref-rules*]: $(seq, HOL.eq) \in S \rightarrow S \rightarrow bool-rel$

shows $(?f, nbae-nba) \in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle L, S \rangle\ nbai-nba-rel$

unfolding *nbae-nba-def* **by** *autoref*

concrete-definition *nbae-nba-impl* **uses** *nbae-nba-impl*

lemma *nbae-nba-impl-refine*[*autoref-rules*]:

assumes *GEN-OP leq* *HOL.eq* $(L \rightarrow L \rightarrow bool-rel)$

assumes *GEN-OP seq* *HOL.eq* $(S \rightarrow S \rightarrow bool-rel)$

shows $(nbae-nba-impl\ leq\ seq, nbae-nba) \in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle L, S \rangle\ nbai-nba-rel$

using *nbae-nba-impl.refine* *assms* **unfolding** *autoref-tag-defs* **by** *this*

end

43 Explore and Enumerate Nodes of Nondeterministic Büchi Automata

theory *NBA-Translate*
imports *NBA-Explicit*
begin

43.1 Syntax

no-syntax *-do-let* :: [*pttrn*, 'a] \Rightarrow *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
syntax *-do-let* :: [*pttrn*, 'a] \Rightarrow *do-bind* ((*2let* - =/ -) *13*)

44 Image on Explicit Automata

definition *nbae-image* **where** *nbae-image* *f* *A* \equiv *nbae* (*alphabet* *A*) (*f* ' *initiale* *A*)

((λ (*p*, *a*, *q*). (*f* *p*, *a*, *f* *q*)) ' *transitione* *A*) (*f* ' *acceptinge* *A*)

lemma *nbae-image-param*[*param*]: (*nbae-image*, *nbae-image*) \in (*S* \rightarrow *T*) \rightarrow (*L*, *S*) *nbae-rel* \rightarrow (*L*, *T*) *nbae-rel*

unfolding *nbae-image-def* **by** *parametricity*

lemma *nbae-image-id*[*simp*]: *nbae-image* *id* = *id* **unfolding** *nbae-image-def* **by** *auto*

lemma *nbae-image-nba-nbae*: *nbae-image* *f* (*nba-nbae* *A*) = *nbae* (*alphabet* *A*) (*f* ' *initial* *A*)

(\bigcup *p* \in *nodes* *A*. \bigcup *a* \in *alphabet* *A*. *f* ' {*p*} \times {*a*} \times *f* ' *transition* *A* *a* *p*)

(*f* ' {*p* \in *nodes* *A*. *accepting* *A* *p*})

unfolding *nba-nbae-def* *nbae-image-def* *nbae.simps* *Set.filter-def* **by** *force*

45 Exploration and Translation

definition *trans-spec* **where**

trans-spec *A* *f* \equiv \bigcup *p* \in *nodes* *A*. \bigcup *a* \in *alphabet* *A*. *f* ' {*p*} \times {*a*} \times *f* ' *transition* *A* *a* *p*

definition *trans-algo* **where**

trans-algo *N* *L* *S* *f* \equiv

FOREACH *N* (λ *p* *T*. *do* {

ASSERT (*p* \in *N*);

FOREACH *L* (λ *a* *T*. *do* {

ASSERT (*a* \in *L*);

FOREACH (*S* *a* *p*) (λ *q* *T*. *do* {

ASSERT (*q* \in *S* *a* *p*);


```

    ASSERT ((f p, a, f q) ∉ T);
    RETURN (insert (f p, a, f q) T) }
  ) T }
) T }
) {}

```

lemma *trans-algo-refine*:

```

assumes finite (nodes A) finite (alphabet A) inj-on f (nodes A)
assumes N = nodes A L = alphabet A S = transition A
shows (trans-algo N L S f, SPEC (HOL.eq (trans-spec A f))) ∈ ⟨Id⟩ nres-rel
unfolding trans-algo-def trans-spec-def assms(4-6)
proof (refine-vcg FOREACH-rule-insert-eq)
  show finite (nodes A) using assms(1) by this
  show (⋃ p ∈ nodes A. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a
p) =
    (⋃ p ∈ nodes A. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) by
rule
  show (⋃ p ∈ {}. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) =
{} by simp
  fix T x
  assume 1: T ⊆ nodes A x ∈ nodes A x ∉ T
  show finite (alphabet A) using assms(2) by this
  show (⋃ a ∈ {}. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) =
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p)
    (⋃ a ∈ alphabet A. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) =
    (⋃ p ∈ insert x T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p)
by auto
  fix Ta xa
  assume 2: Ta ⊆ alphabet A xa ∈ alphabet A xa ∉ Ta
  show finite (transition A xa x) using 1 2 assms(1) by (meson infinite-subset
nba.nodes-transition subsetI)
  show (f ‘ {x} × {xa} × f ‘ transition A xa x) ∪
    (⋃ a ∈ Ta. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) =
    (⋃ a ∈ insert xa Ta. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p)
  by auto
  show (f ‘ {x} × {xa} × f ‘ {}) ∪
    (⋃ a ∈ Ta. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p) =
    (⋃ a ∈ Ta. f ‘ {x} × {a} × f ‘ transition A a x) ∪
    (⋃ p ∈ T. ⋃ a ∈ alphabet A. f ‘ {p} × {a} × f ‘ transition A a p)
  by auto
  fix Tb xb
  assume 3: Tb ⊆ transition A xa x xb ∈ transition A xa x xb ∉ Tb
  show (f x, xa, f xb) ∉ f ‘ {x} × {xa} × f ‘ Tb ∪
    (⋃ a ∈ Ta. f ‘ {x} × {a} × f ‘ transition A a x) ∪

```

$(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f' \{p\} \times \{a\} \times f' \text{ transition } A a p)$
using 1 2 3 *assms*(3) **by** (*blast dest: inj-onD*)
show $f' \{x\} \times \{xa\} \times f' \text{ insert } xb Tb \cup$
 $(\bigcup a \in Ta. f' \{x\} \times \{a\} \times f' \text{ transition } A a x) \cup$
 $(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f' \{p\} \times \{a\} \times f' \text{ transition } A a p) =$
 $\text{insert } (f x, xa, f xb) (f' \{x\} \times \{xa\} \times f' Tb \cup$
 $(\bigcup a \in Ta. f' \{x\} \times \{a\} \times f' \text{ transition } A a x) \cup$
 $(\bigcup p \in T. \bigcup a \in \text{alphabet } A. f' \{p\} \times \{a\} \times f' \text{ transition } A a p))$
by auto
qed

definition *nba-image* :: ('state₁ ⇒ 'state₂) ⇒ ('label, 'state₁) *nba* ⇒ ('label, 'state₂) *nba* **where**
nba-image *f* *A* ≡ *nba*
(alphabet *A)*
(f' initial *A)*
 $(\lambda a p. f' \text{ transition } A a (\text{inv-into } (\text{nodes } A) f p))$
 $(\lambda p. \text{accepting } A (\text{inv-into } (\text{nodes } A) f p))$

lemma *nba-image-rel*[*param*]:
assumes *inj-on* *f* (*nodes* *A*)
shows $(A, \text{nba-image } f A) \in \langle \text{Id-on } (\text{alphabet } A), \text{br } f (\lambda p. p \in \text{nodes } A) \rangle$
nba-rel
proof –
have $A = \text{nba } (\text{alphabet } A) (\text{initial } A) (\text{transition } A) (\text{accepting } A)$ **by** *simp*
also have $(\dots, \text{nba-image } f A) \in \langle \text{Id-on } (\text{alphabet } A), \text{br } f (\lambda p. p \in \text{nodes } A) \rangle$
nba-rel
using *assms unfolding nba-image-def*
by (*parametricity*) (*auto intro: nba-rel-eq simp: in-br-conv br-set-rel-alt*)
finally show ?thesis **by this**
qed

lemma *nba-image-nodes*[*simp*]:
assumes *inj-on* *f* (*nodes* *A*)
shows $\text{nodes } (\text{nba-image } f A) = f' \text{ nodes } A$
proof –
have $(\text{nodes } A, \text{nodes } (\text{nba-image } f A)) \in \langle \text{br } f (\lambda p. p \in \text{nodes } A) \rangle \text{ set-rel}$
using *assms by parametricity*
then show ?thesis **unfolding br-set-rel-alt by simp**
qed

lemma *nba-image-language*[*simp*]:
assumes *inj-on* *f* (*nodes* *A*)
shows $\text{language } (\text{nba-image } f A) = \text{language } A$
proof –
have $(\text{language } A, \text{language } (\text{nba-image } f A)) \in \langle \langle \text{Id-on } (\text{alphabet } A) \rangle \text{ stream-rel} \rangle$
set-rel
using *assms by parametricity*
then show ?thesis **by simp**

qed

lemma *nba-image-nbae*:

assumes *inj-on f (nodes A)*
shows *nbae-image f (nba-nbae A) = nba-nbae (nba-image f A)*
unfolding *nbae-image-nba-nbae*
unfolding *nba-nbae-def*
unfolding *nba-image-nodes[OF assms]*
unfolding *nbae.simps*
unfolding *nba-image-def*
unfolding *nba.sel*
using *assms* **by** *auto*

definition *op-translate* :: (*'label, 'state*) *nba* \Rightarrow (*'label, nat*) *nbae nres* **where**
op-translate A \equiv *SPEC* ($\lambda B. \exists f. \text{inj-on } f \text{ (nodes } A) \wedge B = \text{nba-nbae (nba-image } f \text{ } A)$)

lemma *op-translate-language*:

assumes (*RETURN Ai, op-translate A*) \in $\langle\langle \text{Id, nat-rel} \rangle \text{nbaei-nbae-rel}\rangle$ *nres-rel*
shows *language (nbae-nba (nbaei-nbae Ai)) = language A*
proof –

obtain *f* **where** *1*:

(*Ai, nba-nbae (nba-image f A)*) \in $\langle \text{Id, nat-rel} \rangle$ *nbaei-nbae-rel inj-on f (nodes A)*
using *assms[unfolded in-nres-rel-iff op-translate-def, THEN RETURN-ref-SPEC]*
by *metis*
let *?C = nba-image f A*
have (*nbae-nba (nbaei-nbae Ai), nbae-nba (id (nba-nbae ?C))*) \in $\langle \text{Id, nat-rel} \rangle$ *nba-rel*
using *1(1)* **by** *parametricity auto*
also have *nbae-nba (id (nba-nbae ?C)) = (nbae-nba \circ nba-nbae) ?C* **by** *simp*
also have ($\dots, \text{id } ?C$) \in $\langle \text{Id-on (alphabet } ?C), \text{Id-on (nodes } ?C) \rangle$ *nba-rel* **by** *parametricity*
finally have *2: (nbae-nba (nbaei-nbae Ai), ?C) \in*
 $\langle \text{Id-on (alphabet } ?C), \text{Id-on (nodes } ?C) \rangle$ nba-rel **by** *simp*
have (*language (nbae-nba (nbaei-nbae Ai)), language ?C*) \in
 $\langle\langle \text{Id-on (alphabet } ?C) \rangle \text{stream-rel} \rangle$ *set-rel*
using *2* **by** *parametricity*
also have *language ?C = language A* **using** *1(2)* **by** *simp*
finally show *?thesis* **by** *simp*
qed

schematic-goal *to-nbaei-impl*:

fixes *S* :: (*'statei* \times *'state*) *set*

```

assumes [simp]: finite (nodes A)
assumes [autoref-ga-rules]: is-bounded-hashcode S seq bhc
assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE('statei) hms
assumes [autoref-rules]: (seq, HOL.eq) ∈ S → S → bool-rel
assumes [autoref-rules]: (Ai, A) ∈ ⟨L, S⟩ nbai-nba-rel
shows (?f :: ?'a, do {
  let N = nodes A;
  f ← op-set-enumerate N;
  ASSERT (dom f = N);
  ASSERT (∀ p ∈ initial A. f p ≠ None);
  ASSERT (∀ a ∈ alphabet A. ∀ p ∈ dom f. ∀ q ∈ transition A a p. f q ≠
None);
  T ← trans-algo N (alphabet A) (transition A) (λ x. the (f x));
  RETURN (nbae (alphabet A) ((λ x. the (f x)) ' initial A) T
((λ x. the (f x)) ' {p ∈ N. accepting A p}))
}) ∈ ?R
unfolding trans-algo-def by (autoref-monadic (plain))
concrete-definition to-nbaei-impl uses to-nbaei-impl

context
begin

interpretation autoref-syn by this

lemma to-nbaei-impl-refine[autoref-rules]:
  fixes S :: ('statei × 'state) set
  assumes SIDE-PRECOND (finite (nodes A))
  assumes SIDE-GEN-ALGO (is-bounded-hashcode S seq bhc)
  assumes SIDE-GEN-ALGO (is-valid-def-hm-size TYPE('statei) hms)
  assumes GEN-OP seq HOL.eq (S → S → bool-rel)
  assumes (Ai, A) ∈ ⟨L, S⟩ nbai-nba-rel
  shows (RETURN (to-nbaei-impl seq bhc hms Ai),
(OP op-translate ::: ⟨L, S⟩ nbai-nba-rel → ⟨⟨L, nat-rel⟩ nbaei-nbae-rel
nres-rel) § A) ∈
⟨⟨L, nat-rel⟩ nbaei-nbae-rel⟩ nres-rel
proof –
  have 1: finite (alphabet A)
  using nbai-nba-param(2)[param-fo, OF assms(5)] list-set-rel-finite
  unfolding finite-set-rel-def by auto
  note to-nbaei-impl.refine[OF assms[unfolded autoref-tag-defs]]
  also have (do {
    let N = nodes A;
    f ← op-set-enumerate N;
    ASSERT (dom f = N);
    ASSERT (∀ p ∈ initial A. f p ≠ None);
    ASSERT (∀ a ∈ alphabet A. ∀ p ∈ dom f. ∀ q ∈ transition A a p. f q ≠
None);
    T ← trans-algo N (alphabet A) (transition A) (λ x. the (f x));
    RETURN (nbae (alphabet A) ((λ x. the (f x)) ' initial A) T ((λ x. the (f

```

```

x)) ‘ {p ∈ N. accepting A p}
  }, do {
    f ← op-set-enumerate (nodes A);
    T ← SPEC (HOL.eq (trans-spec A (λ x. the (f x))));
    RETURN (nbae (alphabet A) ((λ x. the (f x)) ‘ initial A) T ((λ x. the (f
x)) ‘ {p ∈ nodes A. accepting A p}))
  }) ∈ ⟨Id⟩ nres-rel
  unfolding Let-def comp-apply op-set-enumerate-def using assms(1) 1
  by (refine-vcg vcg0[OF trans-algo-refine]) (auto intro!: inj-on-map-the[unfolded
comp-apply])
  also have (do {
    f ← op-set-enumerate (nodes A);
    T ← SPEC (HOL.eq (trans-spec A (λ x. the (f x))));
    RETURN (nbae (alphabet A) ((λ x. the (f x)) ‘ initial A) T ((λ x. the (f
x)) ‘ {p ∈ nodes A. accepting A p}))
  }) ∈ ⟨Id⟩ nres-rel
  unfolding trans-spec-def nbae-image-nba-nbae by refine-vcg force
  also have (do {
    f ← op-set-enumerate (nodes A);
    RETURN (nbae-image (the ∘ f) (nba-nbae A))
  }, do {
    f ← op-set-enumerate (nodes A);
    RETURN (nba-nbae (nba-image (the ∘ f) A))
  }) ∈ ⟨Id⟩ nres-rel
  unfolding op-set-enumerate-def by (refine-vcg) (simp add: inj-on-map-the
nba-image-nbae)
  also have (do {
    f ← op-set-enumerate (nodes A);
    RETURN (nba-nbae (nba-image (the ∘ f) A))
  }, op-translate A) ∈ ⟨Id⟩ nres-rel
  unfolding op-set-enumerate-def op-translate-def
  by (refine-vcg) (metis Collect-mem-eq inj-on-map-the subset-Collect-conv)
  finally show ?thesis unfolding nres-rel-comp by simp
qed

end

end

```

46 Connecting Nondeterministic Generalized Büchi Automata to CAVA Automata Structures

```

theory NGBA-Graphs
imports
  NGBA

```

CAVA-Automata.Automata-Impl
begin

no-notation build (infixr ## 65)

46.1 Regular Graphs

definition $ngba-g :: ('label, 'state) ngba \Rightarrow 'state \text{ graph-rec}$ **where**
 $ngba-g A \equiv (\mid g-V = UNIV, g-E = E\text{-of-succ } (successors A), g-V0 = initial A$
 $\mid)$

lemma $ngba-g\text{-graph}[simp]: graph (ngba-g A)$ **unfolding** $ngba-g\text{-def graph-def}$ **by** $simp$

lemma $ngba-g-V0: g-V0 (ngba-g A) = initial A$ **unfolding** $ngba-g\text{-def}$ **by** $simp$

lemma $ngba-g-E\text{-rtrancl}: (g-E (ngba-g A))^* = \{(p, q). q \in reachable A p\}$

unfolding $ngba-g\text{-def graph-rec.simps E-of-succ-def}$

proof $safe$

show $(p, q) \in \{(p, q). q \in successors A p\}^*$ **if** $q \in reachable A p$ **for** $p q$
using $that$ **by** $(induct) (auto intro: rtrancl\text{-into-rtrancl})$

show $q \in reachable A p$ **if** $(p, q) \in \{(p, q). q \in successors A p\}^*$ **for** $p q$
using $that$ **by** $induct auto$

qed

lemma $ngba-g\text{-rtrancl-path}: (g-E (ngba-g A))^* = \{(p, target r p) \mid r p. NGBA.path A r p\}$

unfolding $ngba-g\text{-E-rtrancl}$ **by** $blast$

lemma $ngba-g\text{-trancl-path}: (g-E (ngba-g A))^+ = \{(p, target r p) \mid r p. NGBA.path A r p \wedge r \neq []\}$

unfolding $ngba-g\text{-def graph-rec.simps E-of-succ-def}$

proof $safe$

show $\exists r p. (x, y) = (p, target r p) \wedge NGBA.path A r p \wedge r \neq []$
if $(x, y) \in \{(p, q). q \in successors A p\}^+$ **for** $x y$

using $that$

proof $induct$

case $(base y)$

obtain a **where** $1: a \in alphabet A y \in transition A a x$ **using** $base$ **by** $auto$

show $?case$

proof $(intro exI conjI)$

show $(x, y) = (x, target [(a, y)] x)$ **by** $simp$

show $NGBA.path A [(a, y)] x$ **using** 1 **by** $auto$

show $[(a, y)] \neq []$ **by** $simp$

qed

next

case $(step y z)$

obtain r **where** $1: y = target r x NGBA.path A r x r \neq []$ **using** $step(3)$ **by** $auto$

obtain a **where** $2: a \in alphabet A z \in transition A a y$ **using** $step(2)$ **by** $auto$

```

show ?case
proof (intro exI conjI)
  show (x, z) = (x, target (r @ [(a, z)]) x) by simp
  show NGBA.path A (r @ [(a, z)]) x using 1 2 by auto
  show r @ [(a, z)] ≠ [] by simp
qed
qed
show (p, target r p) ∈ {(u, v). v ∈ successors A u}+ if NGBA.path A r p r ≠
[] for r p
  using that by (induct) (fastforce intro: trancl-into-trancl2)+
qed

lemma ngba-g-ipath-run:
  assumes ipath (g-E (ngba-g A)) r
  obtains w
  where run A (w ||| smap (r ∘ Suc) nats) (r 0)
proof -
  have 1: ∃ a ∈ alphabet A. r (Suc i) ∈ transition A a (r i) for i
  using assms unfolding ipath-def ngba-g-def E-of-succ-def by auto
  obtain wr where 2: run A wr (r 0) ∧ i. target (stake i wr) (r 0) = r i
  proof (rule ngba.invariant-run-index)
  show ∃ aq. (fst aq ∈ alphabet A ∧ snd aq ∈ transition A (fst aq) p) ∧ snd aq
= r (Suc i) ∧ True
  if p = r i for i p using that 1 by auto
  show r 0 = r 0 by rule
qed auto
have 3: smap (r ∘ Suc) nats = smap snd wr
proof (rule eqI-snth)
  fix i
  have smap (r ∘ Suc) nats !! i = r (Suc i) by simp
  also have ... = target (stake (Suc i) wr) (r 0) unfolding 2(2) by rule
  also have ... = (r 0 ## trace wr (r 0)) !! Suc i by simp
  also have ... = smap snd wr !! i unfolding ngba.trace-alt-def by simp
  finally show smap (r ∘ Suc) nats !! i = smap snd wr !! i by this
qed
show ?thesis
proof
  show run A (smap fst wr ||| smap (r ∘ Suc) nats) (r 0) using 2(1) unfolding
3 by auto
qed
qed

lemma ngba-g-run-ipath:
  assumes run A (w ||| r) p
  shows ipath (g-E (ngba-g A)) (snth (p ## r))
proof
  fix i
  have 1: w !! i ∈ alphabet A r !! i ∈ transition A (w !! i) (target (stake i (w |||
r)) p)
  using assms by (auto dest: ngba.run-snth)

```

```

have 2:  $r !! i \in \text{successors } A ((p \#\# r) !! i)$ 
  using 1 unfolding sscan-scons-snth[symmetric] ngba.trace-alt-def by auto
show  $((p \#\# r) !! i, (p \#\# r) !! \text{Suc } i) \in g\text{-}E (ngba\text{-}g A)$ 
  using 2 unfolding ngba-g-def graph-rec.simps E-of-succ-def by simp
qed

```

46.2 Indexed Generalized Büchi Graphs

definition *ngba-acc* :: 'state pred gen \Rightarrow 'state \Rightarrow nat set **where**
ngba-acc cs p $\equiv \{k \in \{0 ..< \text{length } cs\}. (cs ! k) p\}$

lemma *ngba-acc-param*[*param*]: $(ngba\text{-}acc, ngba\text{-}acc) \in \langle S \rightarrow \text{bool-rel} \rangle \text{list-rel} \rightarrow S \rightarrow \langle \text{nat-rel} \rangle \text{set-rel}$
unfolding *ngba-acc-def list-rel-def list-all2-conv-all-nth fun-rel-def* **by** *auto*

definition *ngba-igbg* :: ('label, 'state) *ngba* \Rightarrow 'state *igb-graph-rec* **where**
ngba-igbg A $\equiv \text{graph-rec.extend } (ngba\text{-}g A) (\lambda \text{igbg_num_acc} = \text{length } (\text{accepting } A), \text{igbg_acc} = ngba\text{-}acc (\text{accepting } A))$

lemma *acc-run-language*:

assumes *igb-graph* (*ngba-igbg A*)

shows $Ex (\text{igb-graph.is-acc-run } (ngba\text{-}igbg A)) \longleftrightarrow \text{language } A \neq \{\}$

proof

interpret *igb-graph ngba-igbg A* **using** *assms* **by** *this*

have [*simp*]: $V0 = g\text{-}V0 (ngba\text{-}g A) E = g\text{-}E (ngba\text{-}g A) \text{num-acc} = \text{length } (\text{accepting } A)$

$k \in \text{acc } p \longleftrightarrow k < \text{length } (\text{accepting } A) \wedge (\text{accepting } A ! k) p$ **for** $p k$

unfolding *ngba-igbg-def ngba-acc-def graph-rec.defs* **by** *simp+*

show $\text{language } A \neq \{\}$ **if** *run*: $Ex \text{is-acc-run}$

proof –

obtain r **where** 1: *is-acc-run r* **using** *run* **by** *rule*

have 2: $r 0 \in V0 \text{ipath } E r \text{is-acc } r$

using 1 **unfolding** *is-acc-run-def graph-defs.is-run-def* **by** *auto*

obtain w **where** 3: $\text{run } A (w \parallel \text{smap } (r \circ \text{Suc}) \text{nats}) (r 0)$ **using** *ngba-g-ipath-run 2(2)* **by** *auto*

have 4: $r 0 \#\# \text{smap } (r \circ \text{Suc}) \text{nats} = \text{smap } r \text{nats}$ **by** (*simp*) (*metis stream.map-comp smap-siterate*)

have 5: $\text{infs } (\text{accepting } A ! k) (r 0 \#\# \text{smap } (r \circ \text{Suc}) \text{nats})$ **if** $k < \text{length } (\text{accepting } A)$ **for** k

using 2(3) **that** **unfolding** *infs-infm is-acc-def 4* **by** *simp*

have $w \in \text{language } A$

proof

show $r 0 \in \text{initial } A$ **using** *ngba-g-V0 2(1)* **by** *force*

show $\text{run } A (w \parallel \text{smap } (r \circ \text{Suc}) \text{nats}) (r 0)$ **using** 3 **by** *this*

show $\text{gen infs } (\text{accepting } A) (r 0 \#\# \text{smap } (r \circ \text{Suc}) \text{nats})$

unfolding *gen-def all-set-conv-all-nth* **using** 5 **by** *simp*

qed

then **show** *?thesis* **by** *auto*

qed


```

show Ex is-acc-run if language: language A ≠ {}
proof –
  obtain w where 1: w ∈ language A using language by auto
  obtain r p where 2: p ∈ initial A run A (w ||| r) p gen infs (accepting A)
(p ## r) using 1 by rule
  have is-acc-run (snth (p ## r))
  unfolding is-acc-run-def graph-defs.is-run-def
  proof safe
    show (p ## r) !! 0 ∈ V0 using ngba-g-V0 2(1) by force
    show ipath E (snth (p ## r)) using ngba-g-run-ipath 2(2) by force
    show is-acc (snth (p ## r)) using 2(3) unfolding gen-def infs-infm
is-acc-def by simp
  qed
  then show ?thesis by auto
  qed
qed
end

```

47 Relations on Nondeterministic Generalized Büchi Automata

theory *NGBA-Refine*

imports

NGBA

../Transition-Systems/Transition-System-Refine

begin

definition *ngba-rel* :: $(\text{'label}_1 \times \text{'label}_2) \text{ set} \Rightarrow (\text{'state}_1 \times \text{'state}_2) \text{ set} \Rightarrow$
 $(\text{'label}_1, \text{'state}_1) \text{ ngba} \times (\text{'label}_2, \text{'state}_2) \text{ ngba}) \text{ set}$ **where**
 $[to\text{-relAPP}]$: $\text{ngba-rel } L \ S \equiv \{(A_1, A_2).$
 $(\text{alphabet } A_1, \text{alphabet } A_2) \in \langle L \rangle \text{ set-rel} \wedge$
 $(\text{initial } A_1, \text{initial } A_2) \in \langle S \rangle \text{ set-rel} \wedge$
 $(\text{transition } A_1, \text{transition } A_2) \in L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel} \wedge$
 $(\text{accepting } A_1, \text{accepting } A_2) \in \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}\}$

lemma *ngba-param[param]*:

$(\text{ngba}, \text{ngba}) \in \langle L \rangle \text{ set-rel} \rightarrow \langle S \rangle \text{ set-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel}) \rightarrow \langle S \rightarrow$
 $\text{bool-rel} \rangle \text{ list-rel} \rightarrow$

$\langle L, S \rangle \text{ ngba-rel}$

$(\text{alphabet}, \text{alphabet}) \in \langle L, S \rangle \text{ ngba-rel} \rightarrow \langle L \rangle \text{ set-rel}$

$(\text{initial}, \text{initial}) \in \langle L, S \rangle \text{ ngba-rel} \rightarrow \langle S \rangle \text{ set-rel}$

$(\text{transition}, \text{transition}) \in \langle L, S \rangle \text{ ngba-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ set-rel}$

$(\text{accepting}, \text{accepting}) \in \langle L, S \rangle \text{ ngba-rel} \rightarrow \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}$

unfolding *ngba-rel-def fun-rel-def by auto*

lemma *ngba-rel-id[simp]*: $\langle \text{Id}, \text{Id} \rangle \text{ ngba-rel} = \text{Id}$ **unfolding** *ngba-rel-def using*
ngba.expand by auto

```

lemma enableds-param[param]: (ngba.enableds, ngba.enableds) ∈ ⟨L, S⟩ ngba-rel
→ S → ⟨L ×r S⟩ set-rel
  using ngba-param(2, 4) unfolding ngba.enableds-def fun-rel-def set-rel-def by
fastforce
lemma paths-param[param]: (ngba.paths, ngba.paths) ∈ ⟨L, S⟩ ngba-rel → S →
⟨⟨L ×r S⟩ list-rel⟩ set-rel
  using enableds-param[param-fo] by parametricity
lemma runs-param[param]: (ngba.runs, ngba.runs) ∈ ⟨L, S⟩ ngba-rel → S → ⟨⟨L
×r S⟩ stream-rel⟩ set-rel
  using enableds-param[param-fo] by parametricity

lemma reachable-param[param]: (reachable, reachable) ∈ ⟨L, S⟩ ngba-rel → S →
⟨S⟩ set-rel
proof –
  have 1: reachable A p = (λ wr. target wr p) ‘ ngba.paths A p for A :: ('label,
'state) ngba and p
    unfolding ngba.reachable-alt-def ngba.paths-def by auto
    show ?thesis unfolding 1 using enableds-param[param-fo] by parametricity
  qed
lemma nodes-param[param]: (nodes, nodes) ∈ ⟨L, S⟩ ngba-rel → ⟨S⟩ set-rel
  unfolding ngba.nodes-alt-def Collect-mem-eq by parametricity

lemma gen-param[param]: (gen, gen) ∈ (A → B → bool-rel) → ⟨A⟩ list-rel → B
→ bool-rel
  unfolding gen-def by parametricity

lemma language-param[param]: (language, language) ∈ ⟨L, S⟩ ngba-rel → ⟨⟨L⟩
stream-rel⟩ set-rel
proof –
  have 1: language A = (∪ p ∈ initial A. ∪ wr ∈ ngba.runs A p.
    if gen infs (accepting A) (p ## smap snd wr) then {smap fst wr} else {})
    for A :: ('label, 'state) ngba
    unfolding ngba.language-def ngba.runs-def image-def
    by (auto iff: split-szip-ex simp del: alw-smap)
    show ?thesis unfolding 1 using enableds-param[param-fo] by parametricity
  qed

```

end

48 Implementation of Nondeterministic Generalized Büchi Automata

```

theory NGBA-Implement
imports
  NGBA-Refine
  ../Basic/Implement

```

begin

consts *i-ngba-scheme* :: *interface* \Rightarrow *interface* \Rightarrow *interface*

context

begin

interpretation *autoref-syn* **by** *this*

lemma *ngba-scheme-itype*[*autoref-itype*]:

ngba ::_{*i*} $\langle L \rangle_i$ *i-set* \rightarrow_i $\langle S \rangle_i$ *i-set* \rightarrow_i ($L \rightarrow_i S \rightarrow_i \langle S \rangle_i$ *i-set*) \rightarrow_i $\langle \langle S \rangle_i$ *i-set* \rangle_i
i-list \rightarrow_i

$\langle L, S \rangle_i$ *i-ngba-scheme*

alphabet ::_{*i*} $\langle L, S \rangle_i$ *i-ngba-scheme* \rightarrow_i $\langle L \rangle_i$ *i-set*

initial ::_{*i*} $\langle L, S \rangle_i$ *i-ngba-scheme* \rightarrow_i $\langle S \rangle_i$ *i-set*

transition ::_{*i*} $\langle L, S \rangle_i$ *i-ngba-scheme* \rightarrow_i $L \rightarrow_i S \rightarrow_i \langle S \rangle_i$ *i-set*

accepting ::_{*i*} $\langle L, S \rangle_i$ *i-ngba-scheme* \rightarrow_i $\langle \langle S \rangle_i$ *i-set* \rangle_i *i-list*

by *auto*

end

datatype (*'label*, *'state*) *ngbai* = *ngbai*

(*alphabeti*: *'label list*)

(*initiali*: *'state list*)

(*transitioni*: *'label* \Rightarrow *'state* \Rightarrow *'state list*)

(*acceptingi*: (*'state* \Rightarrow *bool*) *list*)

definition *ngbai-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow

((*'label*₁, *'state*₁) *ngbai* \times (*'label*₂, *'state*₂) *ngbai*) *set* **where**

[*to-relAPP*]: *ngbai-rel* L $S \equiv \{(A_1, A_2).$

(*alphabeti* A_1 , *alphabeti* $A_2) \in \langle L \rangle$ *list-rel* \wedge

(*initiali* A_1 , *initiali* $A_2) \in \langle S \rangle$ *list-rel* \wedge

(*transitioni* A_1 , *transitioni* $A_2) \in L \rightarrow S \rightarrow \langle S \rangle$ *list-rel* \wedge

(*acceptingi* A_1 , *acceptingi* $A_2) \in \langle S \rightarrow \text{bool-rel} \rangle$ *list-rel*}

lemma *ngbai-param*[*param*]:

(*ngbai*, *ngbai*) $\in \langle L \rangle$ *list-rel* $\rightarrow \langle S \rangle$ *list-rel* $\rightarrow (L \rightarrow S \rightarrow \langle S \rangle$ *list-rel*) \rightarrow

$\langle S \rightarrow \text{bool-rel} \rangle$ *list-rel* $\rightarrow \langle L, S \rangle$ *ngbai-rel*

(*alphabeti*, *alphabeti*) $\in \langle L, S \rangle$ *ngbai-rel* $\rightarrow \langle L \rangle$ *list-rel*

(*initiali*, *initiali*) $\in \langle L, S \rangle$ *ngbai-rel* $\rightarrow \langle S \rangle$ *list-rel*

(*transitioni*, *transitioni*) $\in \langle L, S \rangle$ *ngbai-rel* $\rightarrow L \rightarrow S \rightarrow \langle S \rangle$ *list-rel*

(*acceptingi*, *acceptingi*) $\in \langle L, S \rangle$ *ngbai-rel* $\rightarrow \langle S \rightarrow \text{bool-rel} \rangle$ *list-rel*

unfolding *ngbai-rel-def fun-rel-def* **by** *auto*

definition *ngbai-ngba-rel* :: (*'label*₁ \times *'label*₂) *set* \Rightarrow (*'state*₁ \times *'state*₂) *set* \Rightarrow

((*'label*₁, *'state*₁) *ngbai* \times (*'label*₂, *'state*₂) *ngba*) *set* **where**

[*to-relAPP*]: *ngbai-ngba-rel* L $S \equiv \{(A_1, A_2).$

(*alphabeti* A_1 , *alphabet* $A_2) \in \langle L \rangle$ *list-set-rel* \wedge

(*initiali* A_1 , *initial* $A_2) \in \langle S \rangle$ *list-set-rel* \wedge

$(\text{transitioni } A_1, \text{transition } A_2) \in L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel} \wedge$
 $(\text{acceptingi } A_1, \text{accepting } A_2) \in \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}$

lemmas [autoref-rel-intf] = REL-INTFI[of ngbai-ngba-rel i-ngba-scheme]

lemma ngbai-ngba-param[param, autoref-rules]:

$(\text{ngbai}, \text{ngba}) \in \langle L \rangle \text{ list-set-rel} \rightarrow \langle S \rangle \text{ list-set-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel})$
 \rightarrow
 $\langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ ngbai-ngba-rel}$
 $(\text{alphabeti}, \text{alphabet}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle L \rangle \text{ list-set-rel}$
 $(\text{initiali}, \text{initial}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle S \rangle \text{ list-set-rel}$
 $(\text{transitioni}, \text{transition}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel}$
 $(\text{acceptingi}, \text{accepting}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}$
unfolding ngbai-ngba-rel-def fun-rel-def **by** auto

definition ngbai-ngba :: ('label, 'state) ngbai \Rightarrow ('label, 'state) ngba **where**
ngbai-ngba A \equiv ngba (set (alphabeti A)) (set (initiali A)) (λ a p. set (transitioni A a p)) (acceptingi A)

definition ngbai-invar :: ('label, 'state) ngbai \Rightarrow bool **where**
ngbai-invar A \equiv distinct (alphabeti A) \wedge distinct (initiali A) \wedge (\forall a p. distinct (transitioni A a p))

lemma ngbai-ngba-id-param[param]: (ngbai-ngba, id) $\in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle L, S \rangle \text{ ngba-rel}$

proof

fix Ai A

assume 1: (Ai, A) $\in \langle L, S \rangle \text{ ngbai-ngba-rel}$

have 2: ngbai-ngba Ai = ngba (set (alphabeti Ai)) (set (initiali Ai))

(λ a p. set (transitioni Ai a p)) (acceptingi Ai) **unfolding** ngbai-ngba-def **by** rule

have 3: id A = ngba (id (alphabet A)) (id (initial A))

(λ a p. id (transition A a p)) (accepting A) **by** simp

show (ngbai-ngba Ai, id A) $\in \langle L, S \rangle \text{ ngba-rel}$ **unfolding** 2 3 **using** 1 **by** parametricity

qed

lemma ngbai-ngba-br: $\langle \text{Id}, \text{Id} \rangle \text{ ngbai-ngba-rel} = \text{br ngbai-ngba ngbai-invar}$

proof safe

show (A, B) $\in \langle \text{Id}, \text{Id} \rangle \text{ ngbai-ngba-rel}$ **if** (A, B) $\in \text{br ngbai-ngba ngbai-invar}$

for A **and** B :: ('a, 'b) ngba

using that **unfolding** ngbai-ngba-rel-def ngbai-ngba-def ngbai-invar-def

by (auto simp: in-br-conv list-set-rel-def)

show (A, B) $\in \text{br ngbai-ngba ngbai-invar}$ **if** (A, B) $\in \langle \text{Id}, \text{Id} \rangle \text{ ngbai-ngba-rel}$

for A **and** B :: ('a, 'b) ngba

proof –

have 1: (ngbai-ngba A, id B) $\in \langle \text{Id}, \text{Id} \rangle \text{ ngba-rel}$ **using** that **by** parametricity

have 2: ngbai-invar A

using ngbai-ngba-param(2 – 5)[param-fo, OF that]

by (auto simp: in-br-conv list-set-rel-def ngbai-invar-def)

```

    show ?thesis using 1 2 unfolding in-br-conv by auto
  qed
qed

end
theory Degeneralization-Refine
imports Degeneralization Refine
begin

  lemma degen-param[param]: (degen, degen) ∈ ⟨S → bool-rel⟩ list-rel → S ×r
  nat-rel → bool-rel
  proof (intro fun-reII)
    fix cs ds ak bl
    assume (cs, ds) ∈ ⟨S → bool-rel⟩ list-rel (ak, bl) ∈ S ×r nat-rel
    then show (degen cs ak, degen ds bl) ∈ bool-rel
      unfolding degen-def list-rel-def fun-rel-def list-all2-conv-all-nth
      by (cases snd ak < length cs) (auto 0 3)
  qed

  lemma count-param[param]: (Degeneralization.count, Degeneralization.count) ∈
  ⟨A → bool-rel⟩ list-rel → A → nat-rel → nat-rel
  unfolding count-def null-def[symmetric] by parametricity

end

```

49 Algorithms on Nondeterministic Generalized Büchi Automata

```

theory NGBA-Algorithms
imports
  NGBA-Graphs
  NGBA-Implement
  NBA-Combine
  NBA-Algorithms
  Degeneralization-Refine
begin

```

49.1 Operations

```

  definition op-language-empty where [simp]: op-language-empty A ≡ NGBA.language
  A = {}

```

```

  lemmas [autoref-op-pat] = op-language-empty-def[symmetric]

```

49.2 Implementations

```

  context
  begin

```

interpretation *autoref-syn* by *this*

lemma *ngba-g-ahs*: $ngba-g\ A = \langle \rangle\ g-V = UNIV, g-E = E\text{-of-succ}\ (\lambda\ p.\ CAST\ ((\bigcup\ a \in ngba.alphabet\ A.\ ngba.transition\ A\ a\ p :: \langle S \rangle\ list\text{-set-rel}) :: \langle S \rangle\ ahs\text{-rel}\ bhc)),$
 $g-V0 = ngba.initial\ A\ \langle \rangle$
unfolding *ngba-g-def ngba.successors-alt-def CAST-def id-apply autoref-tag-defs*
by *rule*

schematic-goal *ngbai-gi*:

notes [*autoref-ga-rules*] = *map2set-to-list*
fixes $S :: ('statei \times 'state)\ set$
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $S\ seq\ bhc$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('statei)\ hms$
assumes [*autoref-rules*]: $(seq, HOL.eq) \in S \rightarrow S \rightarrow bool\text{-rel}$
assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle\ ngbai\text{-ngba-rel}$
shows $(?f :: ?'a, RETURN\ (ngba-g\ A)) \in ?A$
unfolding *ngba-g-ahs[where* $S = S$ **and** $bhc = bhc$ **by** (*autoref-monadic*
(plain))

concrete-definition *ngbai-gi* **uses** *ngbai-gi*

lemma *ngbai-gi-refine[autoref-rules]*:
fixes $S :: ('statei \times 'state)\ set$
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* $S\ seq\ bhc$)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE('statei)\ hms$)
assumes *GEN-OP* $seq\ HOL.eq\ (S \rightarrow S \rightarrow bool\text{-rel})$
shows $(NGBA-Algorithms.ngbai-gi\ seq\ bhc\ hms, ngba-g) \in$
 $\langle L, S \rangle\ ngbai\text{-ngba-rel} \rightarrow \langle unit\text{-rel}, S \rangle\ g\text{-impl-rel-ext}$
using *ngbai-gi.refine[THEN RETURN-nres-relD]* **assms** **unfolding** *autoref-tag-defs*
by *blast*

schematic-goal *ngba-nodes*:

fixes $S :: ('statei \times 'state)\ set$
assumes [*simp*]: *finite* $((g-E\ (ngba-g\ A))^* \text{ “ } g-V0\ (ngba-g\ A))$
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $S\ seq\ bhc$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('statei)\ hms$
assumes [*autoref-rules*]: $(seq, HOL.eq) \in S \rightarrow S \rightarrow bool\text{-rel}$
assumes [*autoref-rules*]: $(Ai, A) \in \langle L, S \rangle\ ngbai\text{-ngba-rel}$
shows $(?f :: ?'a, op\text{-reachable}\ (ngba-g\ A)) \in ?R$ **by** *autoref*
concrete-definition *ngba-nodes* **uses** *ngba-nodes*
lemma *ngba-nodes-refine[autoref-rules]*:
fixes $S :: ('statei \times 'state)\ set$
assumes *SIDE-PRECOND* (*finite* $(NGBA.nodes\ A)$)
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* $S\ seq\ bhc$)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE('statei)\ hms$)
assumes *GEN-OP* $seq\ HOL.eq\ (S \rightarrow S \rightarrow bool\text{-rel})$
assumes $(Ai, A) \in \langle L, S \rangle\ ngbai\text{-ngba-rel}$
shows $(NGBA-Algorithms.ngba-nodes\ seq\ bhc\ hms\ Ai,$
 $(OP\ NGBA.nodes :: \langle L, S \rangle\ ngbai\text{-ngba-rel} \rightarrow \langle S \rangle\ ahs\text{-rel}\ bhc)\ \$\ A) \in \langle S \rangle$
 $ahs\text{-rel}\ bhc$

proof –
have 1: $NGBA.nodes\ A = op\text{-}reachable\ (ngba\text{-}g\ A)$ **by** (auto simp: ngba-g-V0 ngba-g-E-rtrancl)
have 2: finite $((g\text{-}E\ (ngba\text{-}g\ A))^* \text{ “ } g\text{-}V0\ (ngba\text{-}g\ A))$ **using** *assms(1)* **unfolding** 1 **by** *simp*
show ?thesis **using** ngba-nodes.refine *assms 2* **unfolding** autoref-tag-defs 1 **by** *blast*
qed

lemma ngba-igbg-ahs: $ngba\text{-}igbg\ A = (\downarrow\ g\text{-}V = UNIV, g\text{-}E = E\text{-of}\text{-}succ\ (\lambda\ p.\ CAST\ ((\bigcup\ a \in\ NGBA.alphabet\ A.\ NGBA.transition\ A\ a\ p :: \langle S \rangle\ list\text{-}set\text{-}rel) :: \langle S \rangle\ ahs\text{-}rel\ bhc)), g\text{-}V0 = NGBA.initial\ A,$
 $igbg\text{-}num\text{-}acc = length\ (NGBA.accepting\ A), igbg\text{-}acc = ngba\text{-}acc\ (NGBA.accepting\ A)$)
unfolding ngba-g-def ngba-igbg-def ngba.successors-alt-def CAST-def id-apply autoref-tag-defs
unfolding graph-rec.defs
by *simp*

definition ngba-acc-bs $cs\ p \equiv fold\ (\lambda\ (k, c)\ bs.\ if\ c\ p\ then\ bs\text{-}insert\ k\ bs\ else\ bs)\ (List.enumerate\ 0\ cs)\ (bs\text{-}empty\ ())$

lemma ngba-acc-bs-empty[*simp*]: $ngba\text{-}acc\text{-}bs\ []\ p = bs\text{-}empty\ ()$ **unfolding** ngba-acc-bs-def **by** *simp*

lemma ngba-acc-bs-insert[*simp*]:

assumes $c\ p$

shows $ngba\text{-}acc\text{-}bs\ (cs\ @\ [c])\ p = bs\text{-}insert\ (length\ cs)\ (ngba\text{-}acc\text{-}bs\ cs\ p)$

using *assms* **unfolding** ngba-acc-bs-def **by** (*simp* add: enumerate-append-eq)

lemma ngba-acc-bs-skip[*simp*]:

assumes $\neg\ c\ p$

shows $ngba\text{-}acc\text{-}bs\ (cs\ @\ [c])\ p = ngba\text{-}acc\text{-}bs\ cs\ p$

using *assms* **unfolding** ngba-acc-bs-def **by** (*simp* add: enumerate-append-eq)

lemma ngba-acc-bs-correct[*simp*]: $bs\text{-}\alpha\ (ngba\text{-}acc\text{-}bs\ cs\ p) = ngba\text{-}acc\ cs\ p$

proof (induct *cs* rule: rev-induct)

case Nil

show ?case **unfolding** ngba-acc-def **by** *simp*

next

case (snoc *c cs*)

show ?case **using** less-Suc-eq snoc **by** (cases *c p*) (force *simp*: ngba-acc-def)+

qed

lemma ngba-acc-impl-bs[autoref-rules]: $(ngba\text{-}acc\text{-}bs, ngba\text{-}acc) \in \langle S \rightarrow bool\text{-}rel \rangle\ list\text{-}rel \rightarrow S \rightarrow \langle nat\text{-}rel \rangle\ bs\text{-}set\text{-}rel$

proof –

have $(ngba\text{-}acc\text{-}bs, ngba\text{-}acc) \in \langle Id \rightarrow bool\text{-}rel \rangle\ list\text{-}rel \rightarrow Id \rightarrow \langle nat\text{-}rel \rangle\ bs\text{-}set\text{-}rel$

by (auto *simp*: bs-set-rel-def in-br-conv)

also have $(ngba\text{-}acc, ngba\text{-}acc) \in \langle S \rightarrow bool\text{-}rel \rangle list\text{-}rel \rightarrow S \rightarrow \langle nat\text{-}rel \rangle$
set-rel by parametricity
finally show *?thesis by simp*
qed

schematic-goal *ngbai-igbgi*:

notes $[autoref\text{-}ga\text{-}rules] = map2set\text{-}to\text{-}list$

fixes $S :: ('statei \times 'state) set$

assumes $[autoref\text{-}ga\text{-}rules]: is\text{-}bounded\text{-}hashcode\ S\ seq\ bhc$

assumes $[autoref\text{-}ga\text{-}rules]: is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('statei)\ hms$

assumes $[autoref\text{-}rules]: (seq, HOL.eq) \in S \rightarrow S \rightarrow bool\text{-}rel$

assumes $[autoref\text{-}rules]: (Ai, A) \in \langle L, S \rangle ngbai\text{-}ngba\text{-}rel$

shows $(?f :: ?'a, RETURN (ngba\text{-}igbg\ A)) \in ?A$

unfolding $ngba\text{-}igbg\text{-}ahs[where\ S = S\ and\ bhc = bhc]$ **by** $(autoref\text{-}monadic\ (plain))$

concrete-definition *ngbai-igbgi uses ngbai-igbgi*

lemma *ngbai-igbgi-refine[autoref-rules]*:

fixes $S :: ('statei \times 'state) set$

assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}bounded\text{-}hashcode\ S\ seq\ bhc)$

assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('statei)\ hms)$

assumes $GEN\text{-}OP\ seq\ HOL.eq\ (S \rightarrow S \rightarrow bool\text{-}rel)$

shows $(NGBA\text{-}Algorithms.ngbai\text{-}igbgi\ seq\ bhc\ hms, ngba\text{-}igbg) \in$

$\langle L, S \rangle ngbai\text{-}ngba\text{-}rel \rightarrow igbg\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ S$

using $ngbai\text{-}igbgi.refine[THEN\ RETURN\text{-}nres\text{-}relD]$ **assms** **unfolding** *autoref-tag-defs by blast*

schematic-goal *ngba-language-empty*:

fixes $S :: ('statei \times 'state) set$

assumes $[simp]: igb\text{-}fr\text{-}graph\ (ngba\text{-}igbg\ A)$

assumes $[autoref\text{-}ga\text{-}rules]: is\text{-}bounded\text{-}hashcode\ S\ seq\ bhs$

assumes $[autoref\text{-}ga\text{-}rules]: is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('statei)\ hms$

assumes $[autoref\text{-}rules]: (seq, HOL.eq) \in S \rightarrow S \rightarrow bool\text{-}rel$

assumes $[autoref\text{-}rules]: (Ai, A) \in \langle L, S \rangle ngbai\text{-}ngba\text{-}rel$

shows $(?f :: ?'a, do \{ r \leftarrow op\text{-}find\text{-}lasso\text{-}spec\ (ngba\text{-}igbg\ A); RETURN\ (r = None) \}) \in ?A$

by $(autoref\text{-}monadic\ (plain))$

concrete-definition *ngba-language-empty uses ngba-language-empty*

lemma *ngba-language-empty-refine[autoref-rules]*:

fixes $S :: ('statei \times 'state) set$

assumes $SIDE\text{-}PRECOND\ (finite\ (NGBA.nodes\ A))$

assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}bounded\text{-}hashcode\ S\ seq\ bhc)$

assumes $SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('statei)\ hms)$

assumes $GEN\text{-}OP\ seq\ HOL.eq\ (S \rightarrow S \rightarrow bool\text{-}rel)$

assumes $(Ai, A) \in \langle L, S \rangle ngbai\text{-}ngba\text{-}rel$

shows $(NGBA\text{-}Algorithms.ngba\text{-}language\text{-}empty\ seq\ bhc\ hms\ Ai,$

$(OP\ op\text{-}language\text{-}empty ::: \langle L, S \rangle ngbai\text{-}ngba\text{-}rel \rightarrow bool\text{-}rel)\ \$\ A) \in bool\text{-}rel$

proof –

have $1: NGBA.nodes\ A = op\text{-}reachable\ (ngba\text{-}g\ A)$ **by** $(auto\ simp: ngba\text{-}g\text{-}V0\ ngba\text{-}g\text{-}E\text{-}rtrancl)$

have 2: *finite* $((g-E (ngba-g A))^* \text{ “ } g-V0 (ngba-g A))$ **using** *assms(1)* **unfolding** 1 **by** *simp*
interpret *igb-fr-graph ngba-igbg A*
using 2 **unfolding** *ngba-igbg-def ngba-g-def graph-rec.defs ngba-acc-def* **by** *unfold-locales auto*
have (*RETURN (NGBA-Algorithms.ngba-language-empty seq bhc hms Ai)*,
do { r ← find-lasso-spec; RETURN (r = None) }) $\in \langle \text{bool-rel} \rangle$ *nres-rel*
using *ngba-language-empty.refine assms igb-fr-graph-axioms* **by** *simp*
also have (*do { r ← find-lasso-spec; RETURN (r = None) }*,
RETURN ($\neg Ex$ is-lasso-prpl)) $\in \langle \text{bool-rel} \rangle$ *nres-rel*
unfolding *find-lasso-spec-def* **by** (*refine-vcg*) (*auto split: option.splits*)
finally have *NGBA-Algorithms.ngba-language-empty seq bhc hms Ai* $\longleftrightarrow \neg$
Ex is-lasso-prpl
unfolding *nres-rel-comp* **using** *RETURN-nres-relD* **by** *force*
also have $\dots \longleftrightarrow \neg Ex$ *is-acc-run* **using** *lasso-prpl-acc-run-iff* **by** *auto*
also have $\dots \longleftrightarrow$ *NGBA.language A = { }* **using** *NGBA-Graphs.acc-run-language*
is-igb-graph **by** *auto*
finally show *?thesis* **by** *simp*
qed

lemma *degeneralize-alt-def: degeneralize A = nba*
(ngba.alphabet A)
*(($\lambda p. (p, 0)$) ‘ *ngba.initial A*)*
($\lambda a (p, k). (\lambda q. (q, Degeneralization.count (ngba.accepting A) p k))$) ‘
ngba.transition A a p)
(degen (ngba.accepting A))
unfolding *degeneralization.degeneralize-def* **by** *auto*

schematic-goal *ngba-degeneralize: (?f :: ?'a, degeneralize) \in ?R*
unfolding *degeneralize-alt-def*
using *degen-param[autoref-rules] count-param[autoref-rules]*
by *autoref*

concrete-definition *ngba-degeneralize* **uses** *ngba-degeneralize*
lemmas *ngba-degeneralize-refine[autoref-rules] = ngba-degeneralize.refine*

schematic-goal *nba-intersect'*:
assumes [*autoref-rules*]: *(seq, HOL.eq) $\in L \rightarrow L \rightarrow \text{bool-rel}$*
shows $(?f, intersect') \in \langle L, S \rangle$ *nba-nba-rel* $\rightarrow \langle L, T \rangle$ *nba-nba-rel* $\rightarrow \langle L, S$
 $\times_r T \rangle$ *ngbai-ngba-rel*
unfolding *intersection.product-def* **by** *autoref*
concrete-definition *nba-intersect'* **uses** *nba-intersect'*
lemma *nba-intersect'-refine[autoref-rules]*:
assumes *GEN-OP seq HOL.eq (L $\rightarrow L \rightarrow \text{bool-rel}$)*
shows $(nba-intersect' seq, intersect') \in$
 $\langle L, S \rangle$ *nba-nba-rel* $\rightarrow \langle L, T \rangle$ *nba-nba-rel* $\rightarrow \langle L, S \times_r T \rangle$ *ngbai-ngba-rel*
using *nba-intersect'.refine assms* **unfolding** *autoref-tag-defs* **by** *this*

end

end

50 Nondeterministic Büchi Transition Automata

theory *NBTA*

imports *../Nondeterministic*

begin

datatype ('label, 'state) *nbta* = *nbta*
 (*alphabet*: 'label set)
 (*initial*: 'state set)
 (*transition*: 'label \Rightarrow 'state \Rightarrow 'state set)
 (*accepting*: ('state \times 'label \times 'state) pred)

global-interpretation *nbta*: automaton *nbta* alphabet initial transition accepting
 defines *path* = *nbta.path* **and** *run* = *nbta.run* **and** *reachable* = *nbta.reachable*
and *nodes* = *nbta.nodes*

by *unfold-locales auto*

global-interpretation *nbta*: automaton-run *nbta* alphabet initial transition ac-
cepting

$\lambda P w r p. \text{infs } P (p \#\# r \|\| w \|\| r)$

defines *language* = *nbta.language*

by *standard*

abbreviation *target* **where** *target* \equiv *nbta.target*

abbreviation *states* **where** *states* \equiv *nbta.states*

abbreviation *trace* **where** *trace* \equiv *nbta.trace*

abbreviation *successors* **where** *successors* \equiv *nbta.successors* *TYPE*('label)

end

51 Nondeterministic Generalized Büchi Transition Automata

theory *NGBTA*

imports *../Nondeterministic*

begin

datatype ('label, 'state) *ngbta* = *ngbta*
 (*alphabet*: 'label set)
 (*initial*: 'state set)
 (*transition*: 'label \Rightarrow 'state \Rightarrow 'state set)
 (*accepting*: ('state \times 'label \times 'state) pred gen)

global-interpretation *ngbta*: automaton *ngbta* alphabet initial transition accept-
ing

defines *path* = *ngbta.path* **and** *run* = *ngbta.run* **and** *reachable* = *ngbta.reachable*
and *nodes* = *ngbta.nodes*

```

    by unfold-locales auto
global-interpretation ngbta: automaton-run ngbta alphabet initial transition
accepting
   $\lambda P w r p. \text{gen infs } P (p \#\# r \|\| w \|\| r)$ 
  defines language = ngbta.language
  by standard

abbreviation target where target  $\equiv$  ngbta.target
abbreviation states where states  $\equiv$  ngbta.states
abbreviation trace where trace  $\equiv$  ngbta.trace
abbreviation successors where successors  $\equiv$  ngbta.successors TYPE('label)

end

```

52 Nondeterministic Büchi Transition Automata Combinations

```

theory NBTA-Combine
imports NBTA NGBTA
begin

```

```

  global-interpretation degeneralization: automaton-degeneralization-run
    ngbta ngbta.alphabet ngbta.initial ngbta.transition ngbta.accepting  $\lambda P w r p.$ 
gen infs  $P (p \#\# r \|\| w \|\| r)$ 
    nbt nbt.alphabet nbt.initial nbt.transition nbt.accepting  $\lambda P w r p. \text{infs } P$ 
( $p \#\# r \|\| w \|\| r$ )
    id  $\lambda ((p, k), a, (q, l)). ((p, a, q), k)$ 
    defines degeneralize = degeneralization.degeneralize
  proof
    fix w :: 'a stream
    fix r :: 'b stream
    fix cs p k
    let ?f =  $\lambda ((p, k), a, (q, l)). ((p, a, q), k)$ 
    let ?s = sscan (count cs  $\circ$  id) (p  $\#\#$  r  $\|\|$  w  $\|\|$  r) k
    have infs (degen cs  $\circ$  ?f) ((p, k)  $\#\#$  (r  $\|\|$  ?s)  $\|\|$  w  $\|\|$  (r  $\|\|$  ?s))  $\longleftrightarrow$ 
      infs (degen cs) (smap ?f ((p, k)  $\#\#$  (r  $\|\|$  ?s)  $\|\|$  w  $\|\|$  (r  $\|\|$  ?s)))
      by (simp add: comp-def)
    also have smap ?f ((p, k)  $\#\#$  (r  $\|\|$  ?s)  $\|\|$  w  $\|\|$  (r  $\|\|$  ?s)) = (p  $\#\#$  r  $\|\|$  w  $\|\|$ 
      r)  $\|\|$  k  $\#\#$  ?s
      by (coinduction arbitrary: p k r w) (auto simp: eq-scons simp flip: szip-unfold
        sscan-scons)
    also have  $\dots = (p \#\# r \|\| w \|\| r) \|\| k \#\# \text{sscan (count cs) (p \#\# r \|\| w \|\| r) k}$ 
by simp
    also have infs (degen cs)  $\dots = \text{gen infs cs (p \#\# r \|\| w \|\| r)$  using degen-infs
by this
    finally show infs (degen cs  $\circ$  ?f) ((p, k)  $\#\#$  (r  $\|\|$  ?s)  $\|\|$  w  $\|\|$  (r  $\|\|$  ?s))  $\longleftrightarrow$ 
      gen infs cs (p  $\#\#$  r  $\|\|$  w  $\|\|$  r) by this
  qed

```

lemmas *degeneralize-language*[simp] = *degeneralization.degeneralize-language*[folded *NBTA.language-def*]
lemmas *degeneralize-nodes-finite*[iff] = *degeneralization.degeneralize-nodes-finite*[folded *NBTA.nodes-def*]

global-interpretation *intersection: automaton-intersection-run*

nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting $\lambda P w r p. \text{infs } P$
 $(p \## r \parallel w \parallel r)$
nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting $\lambda P w r p. \text{infs } P$
 $(p \## r \parallel w \parallel r)$
ngbta ngbta.alphabet ngbta.initial ngbta.transition ngbta.accepting $\lambda P w r p.$
gen infs $P (p \## r \parallel w \parallel r)$
 $\lambda c_1 c_2. [c_1 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)), c_2 \circ (\lambda ((p_1, p_2), a, (q_1,$
 $q_2)). (p_2, a, q_2))]$

defines *intersect'* = *intersection.product*

proof

fix *w* :: 'a stream

fix *u* :: 'b stream

fix *v* :: 'c stream

fix *c*₁ *c*₂ *p* *q*

let *?tfst* = $\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)$

let *?tsnd* = $\lambda ((p_1, p_2), a, (q_1, q_2)). (p_2, a, q_2)$

have *gen infs* [*c*₁ \circ *?tfst*, *c*₂ \circ *?tsnd*] $((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v) \longleftrightarrow$
 $\text{infs } c_1 (\text{smap } ?tfst ((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v)) \wedge$
 $\text{infs } c_2 (\text{smap } ?tsnd ((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v))$

unfolding *gen-def* **by** (*simp add: comp-def*)

also have *smap ?tfst* $((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v) = p \## u \parallel w \parallel u$

by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)

also have *smap ?tsnd* $((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v) = q \## v \parallel w \parallel v$

by (*coinduction arbitrary: p q u v w*) (*auto simp flip: szip-unfold, metis stream.collapse*)

finally show *gen infs* [*c*₁ \circ *?tfst*, *c*₂ \circ *?tsnd*] $((p, q) \## (u \parallel v) \parallel w \parallel u \parallel v) \longleftrightarrow$

$\text{infs } c_1 (p \## u \parallel w \parallel u) \wedge \text{infs } c_2 (q \## v \parallel w \parallel v)$ **by this**

qed

lemmas *intersect'-language*[simp] = *intersection.product-language*[folded *NG-BTA.language-def*]

lemmas *intersect'-nodes-finite*[intro] = *intersection.product-nodes-finite*[folded *NGBTA.nodes-def*]

global-interpretation *union: automaton-union-run*

nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting $\lambda P w r p. \text{infs } P$
 $(p \## r \parallel w \parallel r)$

nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting $\lambda P w r p. \text{infs } P$
 $(p \## r \parallel w \parallel r)$

nbta nbta.alphabet nbta.initial nbta.transition nbta.accepting $\lambda P w r p. \text{infs } P$

```

(p ## r ||| w ||| r)
  λ c1 c2 m. case m of (Inl p, a, Inl q) ⇒ c1 (p, a, q) | (Inr p, a, Inr q) ⇒ c2
(p, a, q)
  defines union = union.sum
  by (unfold-locales) (auto simp add: szip-smap-fold comp-def case-prod-unfold
simp flip: stream.map)

lemmas union-language = union.sum-language
lemmas union-nodes-finite = union.sum-nodes-finite

abbreviation intersect where intersect A B ≡ degeneralize (intersect' A B)

lemma intersect-language[simp]: NBTA.language (intersect A B) = NBTA.language
A ∩ NBTA.language B
  by simp
lemma intersect-nodes-finite[intro]:
  assumes finite (NBTA.nodes A) finite (NBTA.nodes B)
  shows finite (NBTA.nodes (intersect A B))
  using intersect'-nodes-finite assms by simp
end

```