

# Transition Systems and Automata

Julian Brunner

May 20, 2026

## 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

<b>1</b>	<b>Basics</b>	<b>1</b>
1.1	Miscellaneous . . . . .	1
<b>2</b>	<b>Finite and Infinite Sequences</b>	<b>2</b>
2.1	List Basics . . . . .	2
2.2	Stream Basics . . . . .	3
2.3	The scan Function . . . . .	7
2.4	Transposing Streams . . . . .	8
2.5	Distinct Streams . . . . .	9
2.6	Sorted Streams . . . . .	9
<b>3</b>	<b>Linear Temporal Logic on Streams</b>	<b>10</b>
3.1	Basics . . . . .	11
3.2	Infinite Occurrence . . . . .	11
<b>4</b>	<b>Zipping Sequences</b>	<b>13</b>
4.1	Zipping Lists . . . . .	13
4.2	Zipping Streams . . . . .	14
<b>5</b>	<b>Maps</b>	<b>15</b>

<b>6</b>	<b>Basics</b>	<b>15</b>
6.1	Expanding set functions to sets of functions . . . . .	15
6.2	Expanding set maps into sets of maps . . . . .	16
<b>7</b>	<b>Transition Systems</b>	<b>18</b>
7.1	Universal Transition Systems . . . . .	19
7.2	Transition Systems . . . . .	19
7.3	Transition Systems with Initial States . . . . .	21
<b>8</b>	<b>Additional Theorems for Transition Systems</b>	<b>21</b>
<b>9</b>	<b>Constructing Paths and Runs in Transition Systems</b>	<b>23</b>
<b>10</b>	<b>Deterministic Automata</b>	<b>24</b>
<b>11</b>	<b>Deterministic Finite Automata</b>	<b>35</b>
<b>12</b>	<b>Nondeterministic Automata</b>	<b>36</b>
<b>13</b>	<b>Nondeterministic Finite Automata</b>	<b>49</b>
<b>14</b>	<b>Deterministic Büchi Automata</b>	<b>49</b>
<b>15</b>	<b>Deterministic Generalized Büchi Automata</b>	<b>50</b>
<b>16</b>	<b>Deterministic Büchi Automata Combinations</b>	<b>51</b>
<b>17</b>	<b>Deterministic Büchi Transition Automata</b>	<b>53</b>
<b>18</b>	<b>Deterministic Generalized Büchi Transition Automata</b>	<b>54</b>
<b>19</b>	<b>Deterministic Büchi Transition Automata Combinations</b>	<b>54</b>
<b>20</b>	<b>Deterministic Co-Büchi Automata</b>	<b>56</b>
<b>21</b>	<b>Deterministic Co-Generalized Co-Büchi Automata</b>	<b>56</b>
<b>22</b>	<b>Deterministic Co-Büchi Automata Combinations</b>	<b>57</b>
<b>23</b>	<b>Deterministic Rabin Automata</b>	<b>59</b>
<b>24</b>	<b>Deterministic Rabin Automata Combinations</b>	<b>60</b>
<b>25</b>	<b>Relations and Refinement</b>	<b>61</b>
25.1	Predicate to Set Conversion Setup . . . . .	61
25.2	Relation Composition . . . . .	62
25.3	Relation Basics . . . . .	62

25.4 Parametricity . . . . .	63
25.5 Lists . . . . .	63
25.6 Streams . . . . .	64
25.7 Functional Relations . . . . .	65
<b>26 Refinement for Transition Systems</b>	<b>66</b>
<b>27 Relations on Deterministic Rabin Automata</b>	<b>67</b>
<b>28 Implementation</b>	<b>68</b>
28.1 Syntax . . . . .	68
28.2 Monadic Refinement . . . . .	69
28.3 Implementations for Sets Represented by Lists . . . . .	70
28.4 Autoref Setup . . . . .	70
<b>29 Implementation of Deterministic Rabin Automata</b>	<b>74</b>
<b>30 Exploration of Deterministic Rabin Automata</b>	<b>75</b>
<b>31 Explicit Deterministic Rabin Automata</b>	<b>77</b>
<b>32 Explore and Enumerate Nodes of Deterministic Rabin Automata</b>	<b>80</b>
32.1 Syntax . . . . .	80
<b>33 Image on Explicit Automata</b>	<b>80</b>
<b>34 Exploration and Translation</b>	<b>81</b>
<b>35 Nondeterministic Büchi Automata</b>	<b>84</b>
<b>36 Nondeterministic Generalized Büchi Automata</b>	<b>85</b>
<b>37 Nondeterministic Büchi Automata Combinations</b>	<b>85</b>
<b>38 Connecting Nondeterministic Büchi Automata to CAVA Automata Structures</b>	<b>87</b>
38.1 Regular Graphs . . . . .	88
38.2 Indexed Generalized Büchi Graphs . . . . .	88
<b>39 Relations on Nondeterministic Büchi Automata</b>	<b>88</b>
<b>40 Implementation of Nondeterministic Büchi Automata</b>	<b>90</b>

<b>41 Algorithms on Nondeterministic Büchi Automata</b>	<b>91</b>
41.1 Miscellaneous Amendments . . . . .	92
41.2 Operations . . . . .	92
41.3 Implementations . . . . .	92
<b>42 Explicit Nondeterministic Büchi Automata</b>	<b>94</b>
<b>43 Explore and Enumerate Nodes of Nondeterministic Büchi Automata</b>	<b>97</b>
43.1 Syntax . . . . .	97
<b>44 Image on Explicit Automata</b>	<b>97</b>
<b>45 Exploration and Translation</b>	<b>98</b>
<b>46 Connecting Nondeterministic Generalized Büchi Automata to CAVA Automata Structures</b>	<b>100</b>
46.1 Regular Graphs . . . . .	100
46.2 Indexed Generalized Büchi Graphs . . . . .	101
<b>47 Relations on Nondeterministic Generalized Büchi Automata</b>	<b>101</b>
<b>48 Implementation of Nondeterministic Generalized Büchi Automata</b>	<b>102</b>
<b>49 Algorithms on Nondeterministic Generalized Büchi Automata</b>	<b>104</b>
49.1 Operations . . . . .	105
49.2 Implementations . . . . .	105
<b>50 Nondeterministic Büchi Transition Automata</b>	<b>108</b>
<b>51 Nondeterministic Generalized Büchi Transition Automata</b>	<b>109</b>
<b>52 Nondeterministic Büchi Transition Automata Combinations</b>	<b>109</b>

## 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]*:  $A \times B = UNIV \longleftrightarrow A = UNIV \wedge B = UNIV$  *<proof>*

```

lemma prod-singleton:
  fst ‘  $A = \{x\} \implies A = \text{fst} \text{ ‘ } A \times \text{snd} \text{ ‘ } A$ 
  snd ‘  $A = \{y\} \implies A = \text{fst} \text{ ‘ } A \times \text{snd} \text{ ‘ } A$ 
  ⟨proof⟩

lemma infinite-subset[trans]:  $\text{infinite } A \implies A \subseteq B \implies \text{infinite } B$  ⟨proof⟩
lemma finite-subset[trans]:  $A \subseteq B \implies \text{finite } B \implies \text{finite } A$  ⟨proof⟩

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  $\text{infinite } A$ 
  ⟨proof⟩

thm inj-on-subset inj-on-subset

lemma inj-inj-on[dest]:  $\text{inj } f \implies \text{inj-on } f S$  ⟨proof⟩

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  $\text{list-all } P xs$ 
  obtains  $(\text{nil}) \text{ } xs = [] \mid (\text{cons}) \text{ } y \text{ } ys$  where  $xs = y \# ys \text{ } P \text{ } y \text{ } \text{list-all } P \text{ } ys$ 
  ⟨proof⟩

lemma lists-iff-set:  $w \in \text{lists } A \iff \text{set } w \subseteq A$  ⟨proof⟩

lemma fold-const:  $\text{fold } \text{const } xs \text{ } a = \text{last } (a \# xs)$ 
  ⟨proof⟩

```

**lemma** *take-Suc*:  $take (Suc\ n)\ xs = (if\ xs = []\ then\ []\ else\ hd\ xs\ \# \ take\ n\ (tl\ xs))$   
 ⟨proof⟩

**lemma** *bind-map[simp]*:  $map\ f\ xs \ggg\ g = xs \ggg\ g \circ f$  ⟨proof⟩

**lemma** *ball-bind[iff]*:  $Ball\ (set\ (xs \ggg\ f))\ P \longleftrightarrow (\forall\ x \in set\ xs. \forall\ y \in set\ (f\ x). P\ y)$   
 ⟨proof⟩

**lemma** *bex-bind[iff]*:  $Bex\ (set\ (xs \ggg\ f))\ P \longleftrightarrow (\exists\ x \in set\ xs. \exists\ y \in set\ (f\ x). P\ y)$   
 ⟨proof⟩

**lemma** *list-choice*:  $list\ all\ (\lambda\ x. \exists\ y. P\ x\ y)\ xs \longleftrightarrow (\exists\ ys. list\ all2\ P\ xs\ ys)$   
 ⟨proof⟩

**lemma** *listset-member*:  $ys \in listset\ XS \longleftrightarrow list\ all2\ (\in)\ ys\ XS$   
 ⟨proof⟩

**lemma** *listset-empty[iff]*:  $listset\ XS = \{\}\longleftrightarrow \neg\ list\ all\ (\lambda\ A. A \neq \{\})\ XS$   
 ⟨proof⟩

**lemma** *listset-finite[iff]*:  
 assumes  $list\ all\ (\lambda\ A. A \neq \{\})\ XS$   
 shows  $finite\ (listset\ XS) \longleftrightarrow list\ all\ finite\ XS$   
 ⟨proof⟩

**lemma** *listset-finite'[intro]*:  
 assumes  $list\ all\ finite\ XS$   
 shows  $finite\ (listset\ XS)$   
 ⟨proof⟩

**lemma** *listset-card[simp]*:  $card\ (listset\ XS) = prod\ list\ (map\ card\ XS)$   
 ⟨proof⟩

## 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**  $xs = y\ \#\# \ ys\ P\ y\ pred\ stream\ P\ ys$   
 ⟨proof⟩

**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**  $stream\text{-}all2\ P\ u\ v$   
 $\langle proof \rangle$

**lemma**  $stream\text{-}rel\text{-}coinduct\text{-}shift[case\text{-}names\ stream\text{-}rel, consumes\ 1]:$   
**assumes**  $R\ u\ v$   
**assumes**  $\bigwedge u\ v. R\ u\ v \implies$   
 $\exists u_1\ u_2\ v_1\ v_2. u = u_1\ @-\ u_2 \wedge v = v_1\ @-\ v_2 \wedge u_1 \neq [] \wedge v_1 \neq [] \wedge list\text{-}all2$   
 $P\ u_1\ v_1 \wedge R\ u_2\ v_2$   
**shows**  $stream\text{-}all2\ P\ u\ v$   
 $\langle proof \rangle$

**lemma**  $stream\text{-}pred\text{-}coinduct[case\text{-}names\ stream\text{-}pred, coinduct\ pred: pred\text{-}stream]:$   
**assumes**  $R\ w$   
**assumes**  $\bigwedge a\ w. R\ (a\ \#\#\ w) \implies P\ a \wedge R\ w$   
**shows**  $pred\text{-}stream\ P\ w$   
 $\langle proof \rangle$

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

**lemma**  $stream\text{-}pred\text{-}flat\text{-}coinduct[case\text{-}names\ stream\text{-}pred, consumes\ 1]:$   
**assumes**  $R\ ws$   
**assumes**  $\bigwedge w\ ws. R\ (w\ \#\#\ ws) \implies w \neq [] \wedge list\text{-}all\ P\ w \wedge R\ ws$   
**shows**  $pred\text{-}stream\ P\ (flat\ ws)$   
 $\langle proof \rangle$

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

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

**lemma**  $stream\text{-}rel\text{-}shift[iff]:$   
**assumes**  $length\ u_1 = length\ v_1$   
**shows**  $stream\text{-}all2\ P\ (u_1\ @-\ u_2)\ (v_1\ @-\ v_2) \longleftrightarrow list\text{-}all2\ P\ u_1\ v_1 \wedge stream\text{-}all2$   
 $P\ u_2\ v_2$   
 $\langle proof \rangle$

**lemma**  $sset\text{-}subset\text{-}stream\text{-}pred: sset\ w \subseteq A \longleftrightarrow pred\text{-}stream\ (\lambda a. a \in A)\ w$   
 $\langle proof \rangle$

**lemma**  $eq\text{-}scons: w = a\ \#\#\ v \longleftrightarrow a = shd\ w \wedge v = stl\ w\ \langle proof \rangle$   
**lemma**  $scons\text{-}eq: a\ \#\#\ v = w \longleftrightarrow shd\ w = a \wedge stl\ w = v\ \langle proof \rangle$   
**lemma**  $eq\text{-}shift: w = u\ @-\ v \longleftrightarrow stake\ (length\ u)\ w = u \wedge sdrop\ (length\ u)\ w$   
 $= v$

*<proof>*

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

*<proof>*

**lemma** *scons-eq-shift*:  $a \#\# w = u @- v \longleftrightarrow (\square = u \wedge a \#\# w = v) \vee (\exists u'. a \# u' = u \wedge w = u' @- v)$

*<proof>*

**lemma** *shift-eq-scons*:  $u @- v = a \#\# w \longleftrightarrow (u = \square \wedge v = a \#\# w) \vee (\exists u'. u = a \# u' \wedge u' @- v = w)$

*<proof>*

**lemma** *stream-all2-sset1*:

**assumes** *stream-all2*  $P \ x \ ys$

**shows**  $\forall x \in \text{sset } xs. \exists y \in \text{sset } ys. P \ x \ y$

*<proof>*

**lemma** *stream-all2-sset2*:

**assumes** *stream-all2*  $P \ xs \ ys$

**shows**  $\forall y \in \text{sset } ys. \exists x \in \text{sset } xs. P \ x \ y$

*<proof>*

**lemma** *smap-eq-scons*[*iff*]:  $\text{smap } f \ xs = y \#\# \ ys \longleftrightarrow f \ (\text{shd } xs) = y \wedge \text{smap } f \ (\text{stl } xs) = ys$

*<proof>*

**lemma** *scons-eq-smap*[*iff*]:  $y \#\# \ ys = \text{smap } f \ xs \longleftrightarrow y = f \ (\text{shd } xs) \wedge ys = \text{smap } f \ (\text{stl } xs)$

*<proof>*

**lemma** *smap-eq-shift*[*iff*]:

$\text{smap } f \ w = u @- v \longleftrightarrow (\exists w_1 \ w_2. w = w_1 @- w_2 \wedge \text{map } f \ w_1 = u \wedge \text{smap } f \ w_2 = v)$

*<proof>*

**lemma** *shift-eq-smap*[*iff*]:

$u @- v = \text{smap } f \ w \longleftrightarrow (\exists w_1 \ w_2. w = w_1 @- w_2 \wedge u = \text{map } f \ w_1 \wedge v = \text{smap } f \ w_2)$

*<proof>*

**lemma** *szip-eq-scons*[*iff*]:  $\text{szip } xs \ ys = z \#\# \ zs \longleftrightarrow (\text{shd } xs, \text{shd } ys) = z \wedge \text{szip } (\text{stl } xs) \ (\text{stl } ys) = zs$

*<proof>*

**lemma** *scons-eq-szip*[*iff*]:  $z \#\# \ zs = \text{szip } xs \ ys \longleftrightarrow z = (\text{shd } xs, \text{shd } ys) \wedge zs = \text{szip } (\text{stl } xs) \ (\text{stl } ys)$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *snth-0*:  $(a \#\# w) !! 0 = a$   $\langle$ proof $\rangle$

**lemma** *eqI-snth*:

**assumes**  $\bigwedge i. u !! i = v !! i$

**shows**  $u = v$

$\langle$ proof $\rangle$

**lemma** *stream-pred-snth*:  $\text{pred-stream } P w \longleftrightarrow (\forall i. P (w !! i))$

$\langle$ proof $\rangle$

**lemma** *stream-rel-snth*:  $\text{stream-all2 } P u v \longleftrightarrow (\forall i. P (u !! i) (v !! i))$

$\langle$ proof $\rangle$

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

$\langle$ proof $\rangle$

**lemma** *sconst-eq*[*iff*]:  $\text{sconst } x = \text{sconst } y \longleftrightarrow x = y$   $\langle$ proof $\rangle$

**lemma** *stream-pred--sconst*[*iff*]:  $\text{pred-stream } P (\text{sconst } x) \longleftrightarrow P x$

$\langle$ proof $\rangle$

**lemma** *stream-rel-sconst*[*iff*]:  $\text{stream-all2 } P (\text{sconst } x) (\text{sconst } y) \longleftrightarrow P x y$

$\langle$ proof $\rangle$

**lemma** *set-sset-stake*[*intro!*, *simp*]:  $\text{set } (\text{stake } n \text{ } xs) \subseteq \text{sset } xs$

$\langle$ proof $\rangle$

**lemma** *sset-sdrop*[*intro!*, *simp*]:  $\text{sset } (\text{sdrop } n \text{ } xs) \subseteq \text{sset } xs$

$\langle$ proof $\rangle$

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

$\langle$ proof $\rangle$

**lemma** *szip-transfer*[*transfer-rule*]:

**includes** *lifting-syntax*

**shows**  $(\text{stream-all2 } A \implies \text{stream-all2 } B \implies \text{stream-all2 } (\text{rel-prod } A B))$

*szip szip*

$\langle$ proof $\rangle$

**lemma** *siterate-transfer*[*transfer-rule*]:

**includes** *lifting-syntax*

**shows**  $((A \implies A) \implies A \implies \text{stream-all2 } A) \text{ siterate siterate}$

$\langle$ proof $\rangle$

**lemma** *split-stream-first*:

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

**obtains**  $ys \ a \ zs$

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

$\langle$ proof $\rangle$

**lemma** *split-stream-first'*:

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

**obtains**  $ys \ zs$

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

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

**lemma** *streams-int*[*simp*]:  $\text{streams } (A \cap B) = \text{streams } A \cap \text{streams } B$  ⟨proof⟩

**lemma** *streams-Int*[*simp*]:  $\text{streams } (\bigcap S) = \bigcap (\text{streams } 'S)$  ⟨proof⟩

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

**lemma** *pred-stream-streamsp*[*pred-set-conv*]:  $\text{pred-stream} = \text{streamsp}$   
 ⟨proof⟩

## 2.3 The scan Function

**primrec** (*transfer*) *scan* ::  $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \text{ list}$  **where**  
 $\text{scan } f \ [] \ a = [] \mid \text{scan } f \ (x \# xs) \ a = f \ x \ a \ \# \ \text{scan } f \ xs \ (f \ x \ a)$

**lemma** *scan-append*[*simp*]:  $\text{scan } f \ (xs @ ys) \ a = \text{scan } f \ xs \ a @ \text{scan } f \ ys \ (fold \ f \ xs \ a)$   
 ⟨proof⟩

**lemma** *scan-eq-nil*[*iff*]:  $\text{scan } f \ xs \ a = [] \longleftrightarrow xs = []$  ⟨proof⟩

**lemma** *scan-eq-cons*[*iff*]:  
 $\text{scan } f \ xs \ a = b \ \# \ w \longleftrightarrow (\exists \ y \ ys. xs = y \ \# \ ys \wedge f \ y \ a = b \wedge \text{scan } f \ ys \ (f \ y \ a) = w)$   
 ⟨proof⟩

**lemma** *scan-eq-append*[*iff*]:  
 $\text{scan } f \ xs \ a = u @ v \longleftrightarrow (\exists \ ys \ zs. xs = ys @ zs \wedge \text{scan } f \ ys \ a = u \wedge \text{scan } f \ zs \ (fold \ f \ ys \ a) = v)$   
 ⟨proof⟩

**lemma** *scan-length*[*simp*]:  $\text{length } (\text{scan } f \ xs \ a) = \text{length } xs$   
 ⟨proof⟩

**lemma** *scan-last*:  $\text{last } (a \ \# \ \text{scan } f \ xs \ a) = \text{fold } f \ xs \ a$   
 ⟨proof⟩

**lemma** *scan-butlast*[*simp*]:  $\text{scan } f \ (\text{butlast } xs) \ a = \text{butlast } (\text{scan } f \ xs \ a)$   
 ⟨proof⟩

**lemma** *scan-const*[*simp*]:  $\text{scan } \text{const } xs \ a = xs$   
 ⟨proof⟩

**lemma** *scan-nth*[*simp*]:  
**assumes**  $i < \text{length } (\text{scan } f \ xs \ a)$   
**shows**  $\text{scan } f \ xs \ a \ ! \ i = \text{fold } f \ (\text{take } (\text{Suc } i) \ xs) \ a$   
 ⟨proof⟩

**lemma** *scan-map*[*simp*]:  $\text{scan } f \ (\text{map } g \ xs) \ a = \text{scan } (f \circ g) \ xs \ a$   
 ⟨proof⟩

**lemma** *scan-take*[*simp*]:  $\text{take } k \ (\text{scan } f \ xs \ a) = \text{scan } f \ (\text{take } k \ xs) \ a$

*<proof>*

**lemma** *scan-drop[simp]*:  $\text{drop } k (\text{scan } f \text{ } xs \ a) = \text{scan } f (\text{drop } k \ xs) (\text{fold } f (\text{take } k \ xs) \ a)$

*<proof>*

**primcorec** (*transfer*) *sscan* ::  $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \ \text{stream} \Rightarrow 'b \Rightarrow 'b \ \text{stream}$   
**where**

$\text{sscan } f \ xs \ a = f (\text{shd } xs) \ a \ \#\# \ \text{sscan } f (\text{stl } xs) (f (\text{shd } xs) \ a)$

**lemma** *sscan-scons[simp]*:  $\text{sscan } f (x \ \#\# \ xs) \ a = f \ x \ a \ \#\# \ \text{sscan } f \ xs (f \ x \ a)$

*<proof>*

**lemma** *sscan-shift[simp]*:  $\text{sscan } f (xs \ @- \ ys) \ a = \text{scan } f \ xs \ a \ @- \ \text{sscan } f \ ys (\text{fold } f \ xs \ a)$

*<proof>*

**lemma** *sscan-eq-scons[iff]*:

$\text{sscan } f \ xs \ a = b \ \#\# \ w \longleftrightarrow f (\text{shd } xs) \ a = b \wedge \text{sscan } f (\text{stl } xs) (f (\text{shd } xs) \ a) = w$

*<proof>*

**lemma** *scons-eq-sscan[iff]*:

$b \ \#\# \ w = \text{sscan } f \ xs \ a \longleftrightarrow b = f (\text{shd } xs) \ a \wedge w = \text{sscan } f (\text{stl } xs) (f (\text{shd } xs) \ a)$

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

## 2.4 Transposing Streams

**primcorec** (*transfer*) *stranspose* ::  $'a \ \text{stream} \ \text{list} \Rightarrow 'a \ \text{list} \ \text{stream}$  **where**

$\text{stranspose } ws = \text{map } \text{shd } ws \ \#\# \ \text{stranspose } (\text{map } \text{stl } ws)$

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

*<proof>*

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

*<proof>*

**lemma** *stranspose-nil*[simp]: *stranspose* [] = *sconst* [] *<proof>*  
**lemma** *stranspose-cons*[simp]: *stranspose* (w # ws) = *smap2* *Cons* w (*stranspose* ws)  
*<proof>*

**lemma** *snth-stranspose*[simp]: *stranspose* ws !! k = *map* ( $\lambda w. w !! k$ ) ws *<proof>*  
**lemma** *stranspose-nth*[simp]:  
**assumes**  $k < \text{length } ws$   
**shows** *smap* ( $\lambda xs. xs ! k$ ) (*stranspose* ws) = ws ! k  
*<proof>*

## 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** *sdistinct* (x # # xs)  
**obtains**  $x \notin \text{sset } xs \text{ } \text{sdistinct } xs$   
*<proof>*

**lemma** *sdistinct-coinduct*[case-names *sdistinct*, *coinduct* pred: *sdistinct*]:  
**assumes**  $P \text{ } xs$   
**assumes**  $\bigwedge x \text{ } xs. P (x \#\# xs) \Longrightarrow x \notin \text{sset } xs \wedge P \text{ } xs$   
**shows** *sdistinct* xs  
*<proof>*

**lemma** *sdistinct-shift*[intro!]:  
**assumes** *distinct* xs *sdistinct* ys *set* xs  $\cap$  *sset* ys = {}  
**shows** *sdistinct* (xs @- ys)  
*<proof>*

**lemma** *sdistinct-shift-elim*[elim!]:  
**assumes** *sdistinct* (xs @- ys)  
**obtains** *distinct* xs *sdistinct* ys *set* xs  $\cap$  *sset* ys = {}  
*<proof>*

**lemma** *sdistinct-infinite-sset*:  
**assumes** *sdistinct* w  
**shows** *infinite* (*sset* w)  
*<proof>*

**lemma** *not-sdistinct-decomp*:  
**assumes**  $\neg \text{sdistinct } w$   
**obtains** u v a w'  
**where**  $w = u @- a \#\# v @- a \#\# w'$   
*<proof>*

## 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*  $w$   
 $\langle \text{proof} \rangle$

**lemma** *sdescending-scons*:

**assumes** *sdescending*  $(a \## w)$   
**shows** *sdescending*  $w$   
 $\langle \text{proof} \rangle$

**lemma** *sdescending-sappend*:

**assumes** *sdescending*  $(u @- v)$   
**obtains** *sdescending*  $v$   
 $\langle \text{proof} \rangle$

**lemma** *sdescending-sdrop*:

**assumes** *sdescending*  $w$   
**shows** *sdescending*  $(\text{sdrop } k w)$   
 $\langle \text{proof} \rangle$

**lemma** *sdescending-sset-scons*:

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

**lemma** *sdescending-sset-sappend*:

**assumes** *sdescending*  $(u @- v)$   
**assumes**  $a \in \text{set } u \ b \in \text{sset } v$   
**shows**  $a \geq b$   
 $\langle \text{proof} \rangle$

**lemma** *sdescending-snth-antimono*:

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

**lemma** *sdescending-stuck*:

**fixes**  $w :: 'a :: \text{wellorder stream}$   
**assumes** *sdescending*  $w$   
**obtains**  $u a$   
**where**  $w = u @- \text{sconst } a$   
 $\langle \text{proof} \rangle$

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]:  $ev\ P\ (smap\ f\ w) \longleftrightarrow ev\ (P\ \circ\ smap\ f)\ w$  *<proof>*

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

lemma *holds-smap*[iff]:  $holds\ P\ (smap\ f\ w) \longleftrightarrow holds\ (P\ \circ\ f)\ w$  *<proof>*

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

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

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

lemma *alw-ev-scons*[iff]:  $alw\ (ev\ P)\ (a\ \#\#\ w) \longleftrightarrow alw\ (ev\ P)\ w$  *<proof>*

lemma *alw-ev-shift*[iff]:  $alw\ (ev\ P)\ (u\ @-\ v) \longleftrightarrow alw\ (ev\ P)\ v$  *<proof>*

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

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

lemma *ev-alw-scons*[iff]:  $ev\ (alw\ P)\ (a\ \#\#\ w) \longleftrightarrow ev\ (alw\ P)\ w$  *<proof>*

lemma *ev-alw-shift*[iff]:  $ev\ (alw\ P)\ (u\ @-\ v) \longleftrightarrow ev\ (alw\ P)\ v$  *<proof>*

lemma *holds-sconst*[iff]:  $holds\ P\ (sconst\ a) \longleftrightarrow P\ a$  *<proof>*

lemma *HLD-sconst*[iff]:  $HLD\ A\ (sconst\ a) \longleftrightarrow a \in A$  *<proof>*

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

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

lemma *ev-HLD-sset*:  $ev\ (HLD\ A)\ w \longleftrightarrow sset\ w \cap A \neq \{\}$  *<proof>*

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**  $alw (ev \varphi) w$   
 $\langle proof \rangle$

### 3.2 Infinite Occurrence

**abbreviation**  $infs P w \equiv alw (ev (holds P)) w$

**abbreviation**  $fins P w \equiv \neg infs P w$

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

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

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

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

**assumes**  $R w$

**assumes**  $\bigwedge w. R w \implies Bex (sset w) P \wedge ev R (stl w)$

**shows**  $infs P w$

$\langle proof \rangle$

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

**assumes**  $R w$

**assumes**  $\bigwedge w. R w \implies \exists u v. w = u @- v \wedge Bex (set u) P \wedge R v$

**shows**  $infs P w$

$\langle proof \rangle$

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

**assumes**  $R w$

**assumes**  $\bigwedge u v. R (u \#\# v) \implies Bex (set u) P \wedge R v$

**shows**  $infs P (flat w)$

$\langle proof \rangle$

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

**assumes**  $R w a$

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

**shows**  $infs P (a \#\# sscan f w a)$

$\langle proof \rangle$

**lemma**  $infs\text{-}mono$ :  $(\bigwedge a. a \in sset w \implies P a \implies Q a) \implies infs P w \implies infs Q w$

$\langle proof \rangle$

**lemma**  $infs\text{-}mono\text{-}strong$ :  $stream\text{-}all2 (\lambda a b. P a \longrightarrow Q b) u v \implies infs P u \implies infs Q v$

$\langle proof \rangle$

**lemma**  $infs\text{-}all$ :  $Ball (sset w) P \implies infs P w$   $\langle proof \rangle$

**lemma**  $infs\text{-}any$ :  $infs P w \implies Bex (sset w) P$   $\langle proof \rangle$

**lemma**  $infs\text{-}bot$ [ $iff$ ]:  $infs bot w \longleftrightarrow False$   $\langle proof \rangle$

**lemma**  $infs\text{-}top$ [ $iff$ ]:  $infs top w \longleftrightarrow True$   $\langle proof \rangle$

**lemma** *infs-disj*[*iff*]:  $\text{infs } (\lambda a. P a \vee Q a) w \longleftrightarrow \text{infs } P w \vee \text{infs } Q w$   
 <proof>  
**lemma** *infs-bex*[*iff*]:  
**assumes** *finite S*  
**shows**  $\text{infs } (\lambda a. \exists x \in S. P x a) w \longleftrightarrow (\exists x \in S. \text{infs } (P x) w)$   
 <proof>  
**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>  
  
**lemma** *infs-cycle*[*iff*]:  
**assumes**  $w \neq []$   
**shows**  $\text{infs } P (\text{cycle } w) \longleftrightarrow \text{Bex } (\text{set } w) P$   
 <proof>

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>

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

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

**lemma** *zip-eq*[*iff*]:  
**assumes**  $\text{length } u = \text{length } v \text{ length } r = \text{length } s$   
**shows**  $u \parallel v = r \parallel s \longleftrightarrow u = r \wedge v = s$   
 <proof>

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

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

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

**lemma** *list-choice-pair*:  $\text{list-all } (\lambda xy. \text{case-prod } (\lambda x y. \exists z. P x y z) xy) (xs \parallel ys) \longleftrightarrow$   
 $(\exists zs. \text{length } zs = \min (\text{length } xs) (\text{length } ys) \wedge \text{list-all } (\lambda (x, y, z). P x y z) (xs \parallel ys \parallel zs))$   
 $\langle \text{proof} \rangle$

**lemma** *list-rel-zip*[*iff*]:

**assumes**  $\text{length } u = \text{length } v \text{ length } r = \text{length } s$   
**shows**  $\text{list-all2 } (\text{rel-prod } A B) (u \parallel v) (r \parallel s) \longleftrightarrow \text{list-all2 } A u r \wedge \text{list-all2 } B v s$   
 $\langle \text{proof} \rangle$

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

**assumes**  $xs \parallel ys \neq [] \text{ length } xs = \text{length } ys$   
**shows**  $\text{last } (xs \parallel ys) = (\text{last } xs, \text{last } ys)$   
 $\langle \text{proof} \rangle$

## 4.2 Zipping Streams

**notation** *szip* (**infixr**  $\langle ||| \rangle$  51)

**lemmas** [*simp*] = *szip-unfold*

**lemma** *smap-szip-same*:  $\text{smap } f (xs \parallel xs) = \text{smap } (\lambda x. f (x, x)) xs \langle \text{proof} \rangle$

**lemma** *szip-smap*[*simp*]:  $\text{smap } \text{fst } zs \parallel \text{smap } \text{snd } zs = zs \langle \text{proof} \rangle$

**lemma** *szip-smap-fst*[*simp*]:  $\text{smap } \text{fst } (xs \parallel ys) = xs \langle \text{proof} \rangle$

**lemma** *szip-smap-snd*[*simp*]:  $\text{smap } \text{snd } (xs \parallel ys) = ys \langle \text{proof} \rangle$

**lemma** *szip-smap-both*:  $\text{smap } f xs \parallel \text{smap } g ys = \text{smap } (\text{map-prod } f g) (xs \parallel ys)$   
 $\langle \text{proof} \rangle$

**lemma** *szip-smap-left*:  $\text{smap } f xs \parallel ys = \text{smap } (\text{apfst } f) (xs \parallel ys) \langle \text{proof} \rangle$

**lemma** *szip-smap-right*:  $xs \parallel \text{smap } f ys = \text{smap } (\text{apsnd } f) (xs \parallel ys) \langle \text{proof} \rangle$

**lemmas** *szip-smap-fold* = *szip-smap-both szip-smap-left szip-smap-right*

**lemma** *szip-sconst-smap-fst*:  $\text{sconst } a \parallel xs = \text{smap } (\text{Pair } a) xs$   
 $\langle \text{proof} \rangle$

**lemma** *szip-sconst-smap-snd*:  $xs \parallel \text{sconst } a = \text{smap } (\text{prod.swap } \circ \text{Pair } a) xs$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *szip-eq*[*iff*]:  $u \parallel v = r \parallel s \longleftrightarrow u = r \wedge v = s$   
 $\langle \text{proof} \rangle$

**lemma** *stream-rel-szip*[*iff*]:  
 $stream\text{-}all2\ (rel\text{-}prod\ A\ B)\ (u\ |||\ v)\ (r\ |||\ s) \longleftrightarrow stream\text{-}all2\ A\ u\ r \wedge stream\text{-}all2\ B\ v\ s$   
 <proof>

**lemma** *szip-shift*[*simp*]:  
**assumes**  $length\ u = length\ s$   
**shows**  $u\ @-\ v\ |||\ s\ @-\ t = (u\ ||\ s)\ @-\ (v\ |||\ t)$   
 <proof>

**lemma** *szip-sset-fst*[*simp*]:  $fst\ 'sset\ (u\ |||\ v) = sset\ u$  <proof>  
**lemma** *szip-sset-snd*[*simp*]:  $snd\ 'sset\ (u\ |||\ v) = sset\ v$  <proof>  
**lemma** *szip-sset-elim*[*elim*]:  
**assumes**  $(a, b) \in sset\ (u\ |||\ v)$   
**obtains**  $a \in sset\ u\ b \in sset\ v$   
 <proof>

**lemma** *szip-sset*[*simp*]:  $sset\ (u\ |||\ v) \subseteq sset\ u \times sset\ v$  <proof>

**lemma** *sset-szip-finite*[*iff*]:  $finite\ (sset\ (u\ |||\ v)) \longleftrightarrow finite\ (sset\ u) \wedge finite\ (sset\ v)$   
 <proof>

**lemma** *infs-szip-fst*[*iff*]:  $infs\ (P \circ fst)\ (u\ |||\ v) \longleftrightarrow infs\ P\ u$   
 <proof>  
**lemma** *infs-szip-snd*[*iff*]:  $infs\ (P \circ snd)\ (u\ |||\ v) \longleftrightarrow infs\ P\ v$   
 <proof>

end

## 5 Maps

**theory** *Maps*  
**imports** *Sequence-Zip*  
**begin**

## 6 Basics

**lemma** *fun-upd-None*[*simp*]:  
**assumes**  $p \notin dom\ f$   
**shows**  $f\ (p := None) = f$   
 <proof>

**lemma** *finite-set-of-finite-maps'*:  
**assumes**  $finite\ A\ finite\ B$   
**shows**  $finite\ \{m.\ dom\ m \subseteq A \wedge ran\ m \subseteq B\}$   
 <proof>

**lemma** *fold-map-of*:  
**assumes** *distinct xs*  
**shows**  $\text{fold } (\lambda x (k, m). (\text{Suc } k, m (x \mapsto k))) \text{ } xs (k, m) =$   
 $(k + \text{length } xs, m ++ \text{map-of } (xs \parallel [k ..< k + \text{length } xs]))$   
 $\langle \text{proof} \rangle$

## 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**  $\text{expand } (f (x := S)) = (\bigcup y \in S. (\lambda g. g (x := y))) \text{ } \text{'expand } f$   
 $\langle \text{proof} \rangle$

## 6.2 Expanding set maps into sets of maps

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

**lemma** *expand-map-alt-def*:  $\text{expand-map } f =$   
 $\{g. \text{dom } g = \text{dom } f \wedge (\forall x \ S \ y. f \ x = \text{Some } S \longrightarrow g \ x = \text{Some } y \longrightarrow y \in S)\}$   
 $\langle \text{proof} \rangle$

**lemma** *expand-map-dom*:  
**assumes**  $g \in \text{expand-map } f$   
**shows**  $\text{dom } g = \text{dom } f$   
 $\langle \text{proof} \rangle$

**lemma** *expand-map-empty[simp]*:  $\text{expand-map } \text{Map.empty} = \{\text{Map.empty}\}$   $\langle \text{proof} \rangle$

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

**end**

**theory** *Acceptance*

**imports** *Sequence-LTL*

**begin**

**type-synonym**  $'a \text{ pred} = 'a \Rightarrow \text{bool}$   
**type-synonym**  $'a \text{ rabin} = 'a \text{ pred} \times 'a \text{ pred}$   
**type-synonym**  $'a \text{ gen} = 'a \text{ list}$

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

**lemma** *rabin[intro]*:

**assumes**  $IF = (I, F)$  *infs*  $I$  *w fins*  $F$   $w$   
**shows** *rabin*  $IF$   $w$   
 $\langle$ *proof* $\rangle$

**lemma** *rabin-elim*[*elim*]:

**assumes** *rabin*  $IF$   $w$   
**obtains**  $I$   $F$   
**where**  $IF = (I, F)$  *infs*  $I$  *w fins*  $F$   $w$   
 $\langle$ *proof* $\rangle$

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

**lemma** *gen*[*intro*]:

**assumes**  $\bigwedge c. c \in \text{set } cs \implies P$   $c$   $w$   
**shows** *gen*  $P$   $cs$   $w$   
 $\langle$ *proof* $\rangle$

**lemma** *gen-elim*[*elim*]:

**assumes** *gen*  $P$   $cs$   $w$   
**obtains**  $\bigwedge c. c \in \text{set } cs \implies P$   $c$   $w$   
 $\langle$ *proof* $\rangle$

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

**lemma** *cogen*[*intro*]:

**assumes**  $c \in \text{set } cs$   $P$   $c$   $w$   
**shows** *cogen*  $P$   $cs$   $w$   
 $\langle$ *proof* $\rangle$

**lemma** *cogen-elim*[*elim*]:

**assumes** *cogen*  $P$   $cs$   $w$   
**obtains**  $c$   
**where**  $c \in \text{set } cs$   $P$   $c$   $w$   
 $\langle$ *proof* $\rangle$

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

**end**

**theory** *Degeneralization*

**imports**

*Acceptance*

*Sequence-Zip*

**begin**

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

**definition** *degen* ::  $'a \text{ pred} \text{ gen} \Rightarrow 'a \text{ degen} \text{ pred}$  **where**

*degen*  $cs \equiv \lambda (a, k). k \geq \text{length } cs \vee (cs ! k)$   $a$

**lemma** *degen-simps*[*iff*]: *degen*  $cs$   $(a, k) \longleftrightarrow k \geq \text{length } cs \vee (cs ! k)$   $a$   $\langle$ *proof* $\rangle$

**definition** *count* :: 'a pred gen  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  nat **where**

*count cs a k*  $\equiv$   
 if  $k < \text{length } cs$   
 then if  $(cs ! k) a$  then  $\text{Suc } k \bmod \text{length } cs$  else  $k$   
 else if  $cs = []$  then  $k$  else  $0$

**lemma** *count-empty[simp]*: *count [] a k = k*  $\langle \text{proof} \rangle$

**lemma** *count-nonempty[simp]*: *cs  $\neq [] \Rightarrow$  count cs a k  $<$  length cs*  $\langle \text{proof} \rangle$

**lemma** *count-constant-1*:

**assumes**  $k < \text{length } cs$   
**assumes**  $\bigwedge a. a \in \text{set } w \Rightarrow \neg (cs ! k) a$   
**shows** *fold (count cs) w k = k*  
 $\langle \text{proof} \rangle$

**lemma** *count-constant-2*:

**assumes**  $k < \text{length } cs$   
**assumes**  $\bigwedge a. a \in \text{set } (w \parallel k \# \text{scan } (\text{count } cs) w k) \Rightarrow \neg \text{degen } cs a$   
**shows** *fold (count cs) w k = k*  
 $\langle \text{proof} \rangle$

**lemma** *count-step*:

**assumes**  $k < \text{length } cs$   
**assumes**  $(cs ! k) a$   
**shows** *count cs a k = Suc k mod length cs*  
 $\langle \text{proof} \rangle$

**lemma** *degen-skip-condition*:

**assumes**  $k < \text{length } cs$   
**assumes** *infs (degen cs) (w ||| k ## sscan (count cs) w k)*  
**obtains**  $u a v$   
**where**  $w = u @- a ## v \text{ fold } (\text{count } cs) u k = k (cs ! k) a$   
 $\langle \text{proof} \rangle$

**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**  $w = u @- v \text{ fold } (\text{count } cs) u k = l$   
 $\langle \text{proof} \rangle$

**lemma** *degen-skip-arbitrary-condition*:

**assumes**  $l < \text{length } cs$   
**assumes** *infs (degen cs) (w ||| k ## sscan (count cs) w k)*  
**obtains**  $u a v$   
**where**  $w = u @- a ## v \text{ fold } (\text{count } cs) u k = l (cs ! l) a$   
 $\langle \text{proof} \rangle$

**lemma** *gen-degen-step*:

**assumes** *gen infs cs w*  
**obtains**  $u a v$   
**where**  $w = u @- a ## v \text{ degen } cs (a, \text{fold } (\text{count } cs) u k)$   
 $\langle \text{proof} \rangle$

**lemma** *degen-infs[iff]*: *infs (degen cs) (w ||| k ## sscan (count cs) w k)  $\longleftrightarrow$  gen infs cs w*  
 <proof>

**end**

## 7 Transition Systems

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

### 7.1 Universal Transition Systems

**locale** *transition-system-universal* =  
**fixes** *execute* :: *'transition  $\Rightarrow$  'state  $\Rightarrow$  'state*  
**begin**

**abbreviation** *target*  $\equiv$  *fold execute*  
**abbreviation** *states*  $\equiv$  *scan execute*  
**abbreviation** *trace*  $\equiv$  *sscan execute*

**lemma** *target-alt-def*: *target r p = last (p # states r p)* <proof>

**end**

### 7.2 Transition Systems

**locale** *transition-system* =  
*transition-system-universal execute*  
**for** *execute* :: *'transition  $\Rightarrow$  'state  $\Rightarrow$  'state*  
 +  
**fixes** *enabled* :: *'transition  $\Rightarrow$  'state  $\Rightarrow$  bool*  
**begin**

**abbreviation** *successors* *p*  $\equiv$  {*execute a p* | *a. enabled a p*}

**inductive** *path* :: *'transition list  $\Rightarrow$  'state  $\Rightarrow$  bool* **where**  
*nil[intro!]*: *path [] p* |  
*cons[intro!]*: *enabled a p  $\implies$  path r (execute a p)  $\implies$  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*  
 <proof>

**lemma** *path-append-elim[elim!]*:  
**assumes** *path (r @ s) p*

**obtains**  $path\ r\ p\ path\ s\ (target\ r\ p)$   
 $\langle proof \rangle$

**coinductive**  $run :: 'transition\ stream \Rightarrow 'state \Rightarrow bool$  **where**  
 $scons[intro!]:\ enabled\ a\ p \Longrightarrow run\ r\ (execute\ a\ p) \Longrightarrow run\ (a\ \#\#\ r)\ p$

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

**lemma**  $run-shift[intro!]:$   
**assumes**  $path\ r\ p\ run\ s\ (target\ r\ p)$   
**shows**  $run\ (r\ @-\ s)\ p$   
 $\langle proof \rangle$

**lemma**  $run-shift-elim[elim!]:$   
**assumes**  $run\ (r\ @-\ s)\ p$   
**obtains**  $path\ r\ p\ run\ s\ (target\ r\ p)$   
 $\langle proof \rangle$

**lemma**  $run-coinduct[case-names\ run,\ coinduct\ pred:\ run]:$   
**assumes**  $R\ r\ p$   
**assumes**  $\bigwedge a\ r\ p.\ R\ (a\ \#\#\ r)\ p \Longrightarrow enabled\ a\ p \wedge R\ r\ (execute\ a\ p)$   
**shows**  $run\ r\ p$   
 $\langle proof \rangle$

**lemma**  $run-coinduct-shift[case-names\ run,\ consumes\ 1]:$   
**assumes**  $R\ r\ p$   
**assumes**  $\bigwedge r\ p.\ R\ r\ p \Longrightarrow \exists\ s\ t.\ r = s\ @-\ t \wedge s \neq [] \wedge path\ s\ p \wedge R\ t\ (target\ s\ p)$   
**shows**  $run\ r\ p$   
 $\langle proof \rangle$

**lemma**  $run-flat-coinduct[case-names\ run,\ consumes\ 1]:$   
**assumes**  $R\ rs\ p$   
**assumes**  $\bigwedge r\ rs\ p.\ R\ (r\ \#\#\ rs)\ p \Longrightarrow r \neq [] \wedge path\ r\ p \wedge R\ rs\ (target\ r\ p)$   
**shows**  $run\ (flat\ rs)\ p$   
 $\langle proof \rangle$

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

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

**lemma**  $reachable-execute'[intro]:$   
**assumes**  $enabled\ a\ p\ q \in reachable\ (execute\ a\ p)$   
**shows**  $q \in reachable\ p$   
 $\langle proof \rangle$

**lemma**  $reachable-elim'[elim]:$   
**assumes**  $q \in reachable\ p$   
**obtains**  $q = p \mid a$  **where**  $enabled\ a\ p\ q \in reachable\ (execute\ a\ p)$   
 $\langle proof \rangle$

**lemma** *reachable-target*[*intro*]:  
**assumes**  $q \in \text{reachable } p \text{ path } r \ q$   
**shows**  $\text{target } r \ q \in \text{reachable } p$   
 $\langle \text{proof} \rangle$

**lemma** *reachable-target-elim*[*elim*]:  
**assumes**  $q \in \text{reachable } p$   
**obtains**  $r$   
**where**  $\text{path } r \ p \ q = \text{target } r \ p$   
 $\langle \text{proof} \rangle$

**lemma** *reachable-alt-def*:  $\text{reachable } p = \{\text{target } r \ p \mid r. \text{path } r \ p\} \langle \text{proof} \rangle$

**lemma** *reachable-trans*[*trans*]:  $q \in \text{reachable } p \implies s \in \text{reachable } q \implies s \in \text{reachable } p \langle \text{proof} \rangle$

**lemma** *reachable-successors*[*intro!*]:  $\text{successors } p \subseteq \text{reachable } p \langle \text{proof} \rangle$

**lemma** *reachable-step*:  $\text{reachable } p = \text{insert } p \ (\bigcup (\text{reachable } \text{' successors } p)) \langle \text{proof} \rangle$

**end**

### 7.3 Transition Systems with Initial States

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

**inductive-set** *nodes* ::  $\text{'state set}$  **where**  
*initial*[*intro*]:  $\text{initial } p \implies p \in \text{nodes} \mid$   
*execute*[*intro!*]:  $p \in \text{nodes} \implies \text{enabled } a \ p \implies \text{execute } a \ p \in \text{nodes}$

**lemma** *nodes-target*[*intro*]:  
**assumes**  $p \in \text{nodes} \text{ path } r \ p$   
**shows**  $\text{target } r \ p \in \text{nodes}$   
 $\langle \text{proof} \rangle$

**lemma** *nodes-target-elim*[*elim*]:  
**assumes**  $q \in \text{nodes}$   
**obtains**  $r \ p$   
**where**  $\text{initial } p \text{ path } r \ p \ q = \text{target } r \ p$   
 $\langle \text{proof} \rangle$

**lemma** *nodes-alt-def*:  $\text{nodes} = \bigcup (\text{reachable } \text{' Collect } \text{initial}) \langle \text{proof} \rangle$

**lemma** *nodes-trans*[*trans*]:  $p \in \text{nodes} \implies q \in \text{reachable } p \implies q \in \text{nodes} \langle \text{proof} \rangle$

end

end

## 8 Additional Theorems for Transition Systems

**theory** *Transition-System-Extra*

**imports**

*../Basic/Sequence-LTL*

*Transition-System*

**begin**

**context** *transition-system*

**begin**

**definition** *enables*  $p \equiv \{a. \text{enabled } a \ p\}$

**definition** *paths*  $p \equiv \{r. \text{path } r \ p\}$

**definition** *runs*  $p \equiv \{r. \text{run } r \ p\}$

**lemma** *stake-run:*

**assumes**  $\bigwedge k. \text{path } (\text{stake } k \ r) \ p$

**shows**  $\text{run } r \ p$

*<proof>*

**lemma** *snth-run:*

**assumes**  $\bigwedge k. \text{enabled } (r \ !! \ k) \ (\text{target } (\text{stake } k \ r) \ p)$

**shows**  $\text{run } r \ p$

*<proof>*

**lemma** *run-stake:*

**assumes**  $\text{run } r \ p$

**shows**  $\text{path } (\text{stake } k \ r) \ p$

*<proof>*

**lemma** *run-sdrop:*

**assumes**  $\text{run } r \ p$

**shows**  $\text{run } (\text{sdrop } k \ r) \ (\text{target } (\text{stake } k \ r) \ p)$

*<proof>*

**lemma** *run-snth:*

**assumes**  $\text{run } r \ p$

**shows**  $\text{enabled } (r \ !! \ k) \ (\text{target } (\text{stake } k \ r) \ p)$

*<proof>*

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

*<proof>*

**lemma** *reachable-states:*

**assumes**  $q \in \text{reachable } p \ \text{path } r \ q$

**shows**  $\text{set } (\text{states } r \ q) \subseteq \text{reachable } p$

```

    <proof>
lemma reachable-trace:
  assumes  $q \in \text{reachable } p \text{ run } r \ q$ 
  shows  $\text{sset } (\text{trace } r \ q) \subseteq \text{reachable } p$ 
  <proof>

end

context transition-system-initial
begin

  lemma nodes-states:
    assumes  $p \in \text{nodes } \text{path } r \ p$ 
    shows  $\text{set } (\text{states } r \ p) \subseteq \text{nodes}$ 
    <proof>
  lemma nodes-trace:
    assumes  $p \in \text{nodes } \text{run } r \ p$ 
    shows  $\text{sset } (\text{trace } r \ p) \subseteq \text{nodes}$ 
    <proof>

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

```

**lemma** *invariant-run-index*:  
**assumes**  $P\ n\ p \wedge n\ p. P\ n\ p \implies \exists a. \text{enabled } a\ p \wedge P\ (\text{Suc } n)\ (\text{execute } a\ p)$   
 $\wedge Q\ n\ p\ a$   
**obtains**  $r$   
**where**  
 $\text{run } r\ p$   
 $\wedge i. P\ (n + i)\ (\text{target } (\text{stake } i\ r)\ p)$   
 $\wedge i. Q\ (n + i)\ (\text{target } (\text{stake } i\ r)\ p)\ (r\ !!\ i)$   
 $\langle \text{proof} \rangle$

**lemma** *koenig*:  
**assumes**  $\text{infinite } (\text{reachable } p)$   
**assumes**  $\bigwedge q. q \in \text{reachable } p \implies \text{finite } (\text{successors } q)$   
**obtains**  $r$   
**where**  $\text{run } r\ p$   
 $\langle \text{proof} \rangle$

**end**

**end**

## 10 Deterministic Automata

**theory** *Deterministic*

**imports**

$\dots / \text{Transition-Systems} / \text{Transition-System}$   
 $\dots / \text{Transition-Systems} / \text{Transition-System-Extra}$   
 $\dots / \text{Transition-Systems} / \text{Transition-System-Construction}$   
 $\dots / \text{Basic} / \text{Degeneralization}$

**begin**

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

**sublocale** *transition-system-initial*

$\text{transition } A\ \lambda\ a\ p. a \in \text{alphabet } A\ \lambda\ p. p = \text{initial } A$

**for**  $A$

**defines**  $\text{path}' = \text{path}$  **and**  $\text{run}' = \text{run}$  **and**  $\text{reachable}' = \text{reachable}$  **and**  $\text{nodes}'$   
 $= \text{nodes}$

$\langle \text{proof} \rangle$

**lemma** *path-alt-def*:  $\text{path } A \ w \ p \longleftrightarrow w \in \text{lists } (\text{alphabet } A)$   
 $\langle \text{proof} \rangle$

**lemma** *run-alt-def*:  $\text{run } A \ w \ p \longleftrightarrow w \in \text{streams } (\text{alphabet } A)$   
 $\langle \text{proof} \rangle$

**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*  $A \equiv \{w. \text{path } A \ w \ (\text{initial } A) \wedge \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A)) \ (\text{initial } A)\}$

**lemma** *language[intro]*:  
**assumes** *path*  $A \ w \ (\text{initial } A) \ \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A))$   
 $(\text{initial } A)$   
**shows**  $w \in \text{language } A$   
 $\langle \text{proof} \rangle$

**lemma** *language-elim[elim]*:  
**assumes**  $w \in \text{language } A$   
**obtains** *path*  $A \ w \ (\text{initial } A) \ \text{test } (\text{condition } A) \ w \ (\text{states } A \ w \ (\text{initial } A))$   
 $(\text{initial } A)$   
 $\langle \text{proof} \rangle$

**lemma** *language-alphabet*:  $\text{language } A \subseteq \text{lists } (\text{alphabet } A) \ \langle \text{proof} \rangle$

**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*  $A \equiv \{w. \text{run } A \ w \ (\text{initial } A) \wedge \text{test } (\text{condition } A) \ w \ (\text{trace } A \ w \ (\text{initial } A)) \ (\text{initial } A)\}$

**lemma** *language*[*intro*]:  
 assumes *run A w (initial A) test (condition A) w (trace A w (initial A))*  
 (*initial A*)  
 shows *w ∈ language A*  
 ⟨*proof*⟩

**lemma** *language-elim*[*elim*]:  
 assumes *w ∈ language A*  
 obtains *run A w (initial A) test (condition A) w (trace A w (initial A))*  
 (*initial A*)  
 ⟨*proof*⟩

**lemma** *language-alphabet*: *language A ⊆ streams (alphabet A)* ⟨*proof*⟩

**end**

**locale** *automaton-degeneralization* =  
 a: *automaton automaton<sub>1</sub> alphabet<sub>1</sub> initial<sub>1</sub> transition<sub>1</sub> condition<sub>1</sub>* +  
 b: *automaton automaton<sub>2</sub> alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub>*  
 for *automaton<sub>1</sub> :: 'label set ⇒ 'state ⇒ ('label ⇒ 'state ⇒ 'state) ⇒ 'item pred*  
*gen ⇒ 'automaton<sub>1</sub>*  
 and *alphabet<sub>1</sub> initial<sub>1</sub> transition<sub>1</sub> condition<sub>1</sub>*  
 and *automaton<sub>2</sub> :: 'label set ⇒ 'state degen ⇒ ('label ⇒ 'state degen ⇒ 'state*  
*degen) ⇒ 'item-degen pred ⇒ 'automaton<sub>2</sub>*  
 and *alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub>*  
 +  
 fixes *item :: 'state × 'label × 'state ⇒ 'item*  
 fixes *translate :: 'item-degen ⇒ 'item degen*  
**begin**

**definition** *degeneralize* :: *'automaton<sub>1</sub> ⇒ 'automaton<sub>2</sub>* **where**  
*degeneralize A ≡ automaton<sub>2</sub>*  
 (*alphabet<sub>1</sub> A*)  
 (*initial<sub>1</sub> A, 0*)  
 ( $\lambda a (p, k). (transition_1 A a p, count (condition_1 A) (item (p, a, transition_1 A a p)) k)$ )  
 (*degen (condition<sub>1</sub> A) ◦ translate*)

**lemma** *degeneralize-simps*[*simp*]:  
*alphabet<sub>2</sub> (degeneralize A) = alphabet<sub>1</sub> A*  
*initial<sub>2</sub> (degeneralize A) = (initial<sub>1</sub> A, 0)*  
*transition<sub>2</sub> (degeneralize A) a (p, k) =*  
 (*transition<sub>1</sub> A a p, count (condition<sub>1</sub> A) (item (p, a, transition<sub>1</sub> A a p)) k*)  
*condition<sub>2</sub> (degeneralize A) = degen (condition<sub>1</sub> A) ◦ translate*  
 ⟨*proof*⟩

**lemma** *degeneralize-target*[*simp*]: *b.target (degeneralize A) w (p, k) =*  
 (*a.target A w p, fold (count (condition<sub>1</sub> A) ◦ item) (p # a.states A w p) || w*  
 || *a.states A w p*) *k*)

$\langle proof \rangle$   
**lemma** *degeneralize-states[simp]*:  $b.states (degeneralize A) w (p, k) =$   
 $a.states A w p \parallel scan (count (condition_1 A) \circ item) (p \# a.states A w p \parallel w$   
 $\parallel a.states A w p) k$   
 $\langle proof \rangle$   
**lemma** *degeneralize-trace[simp]*:  $b.trace (degeneralize A) w (p, k) =$   
 $a.trace A w p \parallel sscan (count (condition_1 A) \circ item) (p \#\# a.trace A w p \parallel$   
 $w \parallel a.trace A w p) k$   
 $\langle proof \rangle$

**lemma** *degeneralize-path[iff]*:  $b.path (degeneralize A) w (p, k) \longleftrightarrow a.path A w$   
 $p$   
 $\langle proof \rangle$   
**lemma** *degeneralize-run[iff]*:  $b.run (degeneralize A) w (p, k) \longleftrightarrow a.run A w p$   
 $\langle proof \rangle$

**lemma** *degeneralize-reachable-fst[simp]*:  $fst \text{ ' } b.reachable (degeneralize A) (p, k)$   
 $= a.reachable A p$   
 $\langle proof \rangle$   
**lemma** *degeneralize-reachable-snd-empty[simp]*:  
**assumes**  $condition_1 A = []$   
**shows**  $snd \text{ ' } b.reachable (degeneralize A) (p, k) = \{k\}$   
 $\langle proof \rangle$   
**lemma** *degeneralize-reachable-empty[simp]*:  
**assumes**  $condition_1 A = []$   
**shows**  $b.reachable (degeneralize A) (p, k) = a.reachable A p \times \{k\}$   
 $\langle proof \rangle$   
**lemma** *degeneralize-reachable-snd*:  
 $snd \text{ ' } b.reachable (degeneralize A) (p, k) \subseteq insert k \{0 ..< length (condition_1$   
 $A)\}$   
 $\langle proof \rangle$   
**lemma** *degeneralize-reachable*:  
 $b.reachable (degeneralize A) (p, k) \subseteq a.reachable A p \times insert k \{0 ..< length$   
 $(condition_1 A)\}$   
 $\langle proof \rangle$

**lemma** *degeneralize-nodes-fst[simp]*:  $fst \text{ ' } b.nodes (degeneralize A) = a.nodes A$   
 $\langle proof \rangle$   
**lemma** *degeneralize-nodes-snd-empty*:  
**assumes**  $condition_1 A = []$   
**shows**  $snd \text{ ' } b.nodes (degeneralize A) = \{0\}$   
 $\langle proof \rangle$   
**lemma** *degeneralize-nodes-empty*:  
**assumes**  $condition_1 A = []$   
**shows**  $b.nodes (degeneralize A) = a.nodes A \times \{0\}$   
 $\langle proof \rangle$   
**lemma** *degeneralize-nodes-snd*:  
 $snd \text{ ' } b.nodes (degeneralize A) \subseteq insert 0 \{0 ..< length (condition_1 A)\}$   
 $\langle proof \rangle$

**lemma** *degeneralize-nodes*:  
 $b.\text{nodes } (\text{degeneralize } A) \subseteq a.\text{nodes } A \times \text{insert } 0 \{0 \dots < \text{length } (\text{condition}_1 A)\}$   
 ⟨proof⟩

**lemma** *degeneralize-nodes-finite*[*iff*]:  $\text{finite } (b.\text{nodes } (\text{degeneralize } A)) \longleftrightarrow \text{finite } (a.\text{nodes } A)$   
 ⟨proof⟩

**lemma** *degeneralize-nodes-card*:  $\text{card } (b.\text{nodes } (\text{degeneralize } A)) \leq \max 1 (\text{length } (\text{condition}_1 A)) * \text{card } (a.\text{nodes } A)$   
 ⟨proof⟩

**end**

**locale** *automaton-degeneralization-run* =  
*automaton-degeneralization*  
*automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>  
*automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
*item translate* +  
*a*: *automaton-run* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub> +  
*b*: *automaton-run* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>  
**for** *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub>  
**and** *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>  
**and** *item translate*  
 +  
**assumes** *test*[*iff*]:  $\text{test}_2 (\text{degen } cs \circ \text{translate}) w$   
 $(r \parallel \text{sscan } (\text{count } cs \circ \text{item}) (p \text{ \#\# } r \parallel w \parallel r) k) (p, k) \longleftrightarrow \text{test}_1 cs w r p$   
**begin**

**lemma** *degeneralize-language*[*simp*]:  $b.\text{language } (\text{degeneralize } A) = a.\text{language } A$  ⟨proof⟩

**end**

**locale** *automaton-product* =  
*a*: *automaton* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> +  
*b*: *automaton* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> +  
*c*: *automaton* *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>  
**for** *automaton*<sub>1</sub> :: 'label set ⇒ 'state<sub>1</sub> ⇒ ('label ⇒ 'state<sub>1</sub> ⇒ 'state<sub>1</sub>) ⇒ 'condition<sub>1</sub> ⇒ 'automaton<sub>1</sub>  
**and** *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>  
**and** *automaton*<sub>2</sub> :: 'label set ⇒ 'state<sub>2</sub> ⇒ ('label ⇒ 'state<sub>2</sub> ⇒ 'state<sub>2</sub>) ⇒ 'condition<sub>2</sub> ⇒ 'automaton<sub>2</sub>  
**and** *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
**and** *automaton*<sub>3</sub> :: 'label set ⇒ 'state<sub>1</sub> × 'state<sub>2</sub> ⇒ ('label ⇒ 'state<sub>1</sub> × 'state<sub>2</sub> ⇒ 'state<sub>1</sub> × 'state<sub>2</sub>) ⇒ 'condition<sub>3</sub> ⇒ 'automaton<sub>3</sub>  
**and** *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>  
 +  
**fixes** *condition* :: 'condition<sub>1</sub> ⇒ 'condition<sub>2</sub> ⇒ 'condition<sub>3</sub>

**begin**

**definition** *product* :: 'automaton<sub>1</sub> ⇒ 'automaton<sub>2</sub> ⇒ 'automaton<sub>3</sub> **where**  
  *product* *A B* ≡ *automaton*<sub>3</sub>  
    (*alphabet*<sub>1</sub> *A* ∩ *alphabet*<sub>2</sub> *B*)  
    (*initial*<sub>1</sub> *A*, *initial*<sub>2</sub> *B*)  
    (λ *a* (*p*, *q*). (*transition*<sub>1</sub> *A* *a* *p*, *transition*<sub>2</sub> *B* *a* *q*))  
    (*condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*))

**lemma** *product-simps*[*simp*]:  
  *alphabet*<sub>3</sub> (*product* *A B*) = *alphabet*<sub>1</sub> *A* ∩ *alphabet*<sub>2</sub> *B*  
  *initial*<sub>3</sub> (*product* *A B*) = (*initial*<sub>1</sub> *A*, *initial*<sub>2</sub> *B*)  
  *transition*<sub>3</sub> (*product* *A B*) *a* (*p*, *q*) = (*transition*<sub>1</sub> *A* *a* *p*, *transition*<sub>2</sub> *B* *a* *q*)  
  *condition*<sub>3</sub> (*product* *A B*) = *condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*)  
  ⟨*proof*⟩

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

**lemma** *product-states*[*simp*]: *c.states* (*product* *A B*) *w* (*p*, *q*) = *a.states* *A* *w* *p* ||| *b.states* *B* *w* *q*  
  ⟨*proof*⟩

**lemma** *product-trace*[*simp*]: *c.trace* (*product* *A B*) *w* (*p*, *q*) = *a.trace* *A* *w* *p* ||| *b.trace* *B* *w* *q*  
  ⟨*proof*⟩

**lemma** *product-path*[*iff*]: *c.path* (*product* *A B*) *w* (*p*, *q*) ⟷ *a.path* *A* *w* *p* ∧ *b.path* *B* *w* *q*  
  ⟨*proof*⟩

**lemma** *product-run*[*iff*]: *c.run* (*product* *A B*) *w* (*p*, *q*) ⟷ *a.run* *A* *w* *p* ∧ *b.run* *B* *w* *q*  
  ⟨*proof*⟩

**lemma** *product-reachable*[*simp*]: *c.reachable* (*product* *A B*) (*p*, *q*) ⊆ *a.reachable* *A* *p* × *b.reachable* *B* *q*  
  ⟨*proof*⟩

**lemma** *product-nodes*[*simp*]: *c.nodes* (*product* *A B*) ⊆ *a.nodes* *A* × *b.nodes* *B*  
  ⟨*proof*⟩

**lemma** *product-reachable-fst*[*simp*]:  
  **assumes** *alphabet*<sub>1</sub> *A* ⊆ *alphabet*<sub>2</sub> *B*  
  **shows** *fst* ' *c.reachable* (*product* *A B*) (*p*, *q*) = *a.reachable* *A* *p*  
  ⟨*proof*⟩

**lemma** *product-reachable-snd*[*simp*]:  
  **assumes** *alphabet*<sub>1</sub> *A* ⊇ *alphabet*<sub>2</sub> *B*  
  **shows** *snd* ' *c.reachable* (*product* *A B*) (*p*, *q*) = *b.reachable* *B* *q*  
  ⟨*proof*⟩

**lemma** *product-nodes-fst*[*simp*]:  
  **assumes** *alphabet*<sub>1</sub> *A* ⊆ *alphabet*<sub>2</sub> *B*  
  **shows** *fst* ' *c.nodes* (*product* *A B*) = *a.nodes* *A*

```

    <proof>
lemma product-nodes-snd[simp]:
  assumes alphabet1 A  $\supseteq$  alphabet2 B
  shows snd ' c.nodes (product A B) = b.nodes B
  <proof>

lemma product-nodes-finite[intro]:
  assumes finite (a.nodes A) finite (b.nodes B)
  shows finite (c.nodes (product A B))
  <proof>
lemma product-nodes-finite-strong[iff]:
  assumes alphabet1 A = alphabet2 B
  shows finite (c.nodes (product A B))  $\longleftrightarrow$  finite (a.nodes A)  $\wedge$  finite (b.nodes
B)
  <proof>
lemma product-nodes-card[intro]:
  assumes finite (a.nodes A) finite (b.nodes B)
  shows card (c.nodes (product A B))  $\leq$  card (a.nodes A) * card (b.nodes B)
  <proof>
lemma product-nodes-card-strong[intro]:
  assumes alphabet1 A = alphabet2 B
  shows card (c.nodes (product A B))  $\leq$  card (a.nodes A) * card (b.nodes B)
  <proof>

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 s  $\implies$ 
    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 <proof>

end

```

```

locale automaton-union-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 s  $\implies$ 
    test3 (condition c1 c2) w (r || s) (p, q)  $\longleftrightarrow$  test1 c1 w r p  $\vee$  test2 c2 w s q
begin

  lemma product-language[simp]:
    assumes alphabet1 A = alphabet2 B
    shows c.language (product A B) = a.language A  $\cup$  b.language B
    <proof>

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 <proof>

end

locale automaton-union-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
 $\vee$  test2 c2 w s q
  begin

    lemma product-language[simp]:
      assumes alphabet1 A = alphabet2 B
      shows c.language (product A B) = a.language A  $\cup$  b.language B
      <proof>

    end

  locale automaton-product-list =
    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 list  $\Rightarrow$  ('label  $\Rightarrow$  'state list  $\Rightarrow$  'state list)
 $\Rightarrow$  'condition2  $\Rightarrow$  'automaton2
    and alphabet2 initial2 transition2 condition2
    +
    fixes condition :: 'condition1 list  $\Rightarrow$  'condition2
  begin

    definition product :: 'automaton1 list  $\Rightarrow$  'automaton2 where
      product AA  $\equiv$  automaton2
        ( $\bigcap$  (alphabet1 ' set AA))
        (map initial1 AA)
        ( $\lambda$  a ps. map2 ( $\lambda$  A p. transition1 A a p) AA ps)
        (condition (map condition1 AA))

    lemma product-simps[simp]:
      alphabet2 (product AA) =  $\bigcap$  (alphabet1 ' set AA)
      initial2 (product AA) = map initial1 AA
      transition2 (product AA) a ps = map2 ( $\lambda$  A p. transition1 A a p) AA ps

```

$condition_2$  (product  $AA$ ) = condition (map  $condition_1$   $AA$ )  
 ⟨proof⟩

**lemma** *product-trace-smap*:

**assumes**  $length\ ps = length\ AA\ k < length\ AA$

**shows**  $smap\ (\lambda\ ps.\ ps\ !\ k)\ (b.trace\ (product\ AA)\ w\ ps) = a.trace\ (AA\ !\ k)\ w$   
 ( $ps\ !\ k$ )  
 ⟨proof⟩

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

⟨proof⟩

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

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

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

⟨proof⟩

**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⟩

**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**  $test[iff]$ :  $test_2\ (condition\ cs)\ w\ rs\ ps \longleftrightarrow$

$(\forall\ k < length\ cs.\ test_1\ (cs\ !\ k)\ w\ (smap\ (\lambda\ ps.\ ps\ !\ k)\ rs)\ (ps\ !\ k))$

**begin**

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

⟨proof⟩

**end**

**locale** *automaton-union-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 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  $\longleftrightarrow$ 
  ( $\exists k < \text{length } cs. \text{test}_1 (cs ! k) w (\text{smap } (\lambda ps. ps ! k) rs) (ps ! k)$ )
begin

  lemma product-language[simp]:
    assumes  $\bigcap (alphabet_1 \text{ 'set } AA) = \bigcup (alphabet_1 \text{ 'set } AA)$ 
    shows b.language (product AA) =  $\bigcup (a.language \text{ 'set } AA)$ 
    <proof>

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)
    <proof>

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
  <proof>

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
    <proof>

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*  
 <proof>

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

**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*  
 <proof>

**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*  
 <proof>

**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*  
 <proof>

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. snd a  $\lambda$  a p. fst a  $\in$  alphabet A  $\wedge$  snd a  $\in$  transition A (fst a) p  $\lambda$  p. p
 $\in$  initial A
    for A
    defines path' = path and run' = run and reachable' = reachable and nodes'
= nodes
     $\langle$ proof $\rangle$ 

  lemma states-alt-def: states r p = map snd r  $\langle$ proof $\rangle$ 
  lemma trace-alt-def: trace r p = smap snd r  $\langle$ proof $\rangle$ 

  lemma successors-alt-def: successors A p = ( $\bigcup$  a  $\in$  alphabet A. transition A a
p)  $\langle$ proof $\rangle$ 

  lemma reachable-transition[intro]:
    assumes a  $\in$  alphabet A q  $\in$  reachable A p r  $\in$  transition A a q
    shows r  $\in$  reachable A p
     $\langle$ proof $\rangle$ 
  lemma nodes-transition[intro]:
    assumes a  $\in$  alphabet A p  $\in$  nodes A q  $\in$  transition A a p
    shows q  $\in$  nodes A
     $\langle$ proof $\rangle$ 

  lemma path-alphabet:
    assumes length r = length w path A (w || r) p
    shows w  $\in$  lists (alphabet A)
     $\langle$ proof $\rangle$ 
  lemma run-alphabet:
    assumes run A (w ||| r) p
    shows w  $\in$  streams (alphabet A)
     $\langle$ proof $\rangle$ 

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

```

(*condition A*)

**lemma** *restrict-simps*[*simp*]:

*alphabet* (*restrict A*) = *alphabet A*

*initial* (*restrict A*) = *initial A*

*transition* (*restrict A*) *a p* = (if *a* ∈ *alphabet A* then *transition A a p* else {})

*condition* (*restrict A*) = *condition A*

⟨*proof*⟩

**lemma** *restrict-path*[*simp*]: *path* (*restrict A*) = *path A*

⟨*proof*⟩

**lemma** *restrict-run*[*simp*]: *run* (*restrict A*) = *run A*

⟨*proof*⟩

**end**

**locale** *automaton-path* =

*automaton automaton alphabet initial transition condition*

**for** *automaton* :: '*label set* ⇒ '*state set* ⇒ ('*label* ⇒ '*state* ⇒ '*state set*) ⇒ '*condition* ⇒ '*automaton*

**and** *alphabet initial transition condition*

+

**fixes** *test* :: '*condition* ⇒ '*label list* ⇒ '*state list* ⇒ '*state* ⇒ *bool*

**begin**

**definition** *language* :: '*automaton* ⇒ '*label list set* **where**

*language A* ≡ {*w* | *w r p. length r = length w* ∧ *p* ∈ *initial A* ∧ *path A* (*w* || *r*) *p* ∧ *test* (*condition A*) *w r p*}

**lemma** *language*[*intro*]:

**assumes** *length r = length w p* ∈ *initial A path A* (*w* || *r*) *p test* (*condition A*) *w r p*

**shows** *w* ∈ *language A*

⟨*proof*⟩

**lemma** *language-elim*[*elim*]:

**assumes** *w* ∈ *language A*

**obtains** *r p*

**where** *length r = length w p* ∈ *initial A path A* (*w* || *r*) *p test* (*condition A*) *w r p*

⟨*proof*⟩

**lemma** *language-alphabet*: *language A* ⊆ *lists* (*alphabet A*) ⟨*proof*⟩

**lemma** *restrict-language*[*simp*]: *language* (*restrict A*) = *language A* ⟨*proof*⟩

**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*  $A \equiv \{w \mid w \ r \ p. p \in \text{initial } A \wedge \text{run } A \ (w \ ||| \ r) \ p \wedge \text{test} \ (\text{condition}$   
 $A) \ w \ r \ p\}$

**lemma** *language[intro]*:  
**assumes**  $p \in \text{initial } A \ \text{run } A \ (w \ ||| \ r) \ p \ \text{test} \ (\text{condition } A) \ w \ r \ p$   
**shows**  $w \in \text{language } A$   
 $\langle \text{proof} \rangle$

**lemma** *language-elim[elim]*:  
**assumes**  $w \in \text{language } A$   
**obtains**  $r \ p$   
**where**  $p \in \text{initial } A \ \text{run } A \ (w \ ||| \ r) \ p \ \text{test} \ (\text{condition } A) \ w \ r \ p$   
 $\langle \text{proof} \rangle$

**lemma** *language-alphabet*:  $\text{language } A \subseteq \text{streams} \ (\text{alphabet } A) \ \langle \text{proof} \rangle$

**lemma** *restrict-language[simp]*:  $\text{language} \ (\text{restrict } A) = \text{language } A \ \langle \text{proof} \rangle$

**end**

**locale** *automaton-degeneralization* =  
*a*: *automaton* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> +  
*b*: *automaton* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
**for** *automaton*<sub>1</sub> :: 'label set  $\Rightarrow$  'state set  $\Rightarrow$  ('label  $\Rightarrow$  'state  $\Rightarrow$  'state set)  $\Rightarrow$   
'item pred gen  $\Rightarrow$  'automaton<sub>1</sub>  
**and** *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>  
**and** *automaton*<sub>2</sub> :: 'label set  $\Rightarrow$  'state degen set  $\Rightarrow$  ('label  $\Rightarrow$  'state degen  $\Rightarrow$   
'state degen set)  $\Rightarrow$  'item-degen pred  $\Rightarrow$  'automaton<sub>2</sub>  
**and** *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
+  
**fixes** *item* :: 'state  $\times$  'label  $\times$  'state  $\Rightarrow$  'item  
**fixes** *translate* :: 'item-degen  $\Rightarrow$  'item degen  
**begin**

**definition** *degeneralize* :: 'automaton<sub>1</sub>  $\Rightarrow$  'automaton<sub>2</sub> **where**  
*degeneralize*  $A \equiv \text{automaton}_2$   
*(alphabet*<sub>1</sub>  $A)$   
*(initial*<sub>1</sub>  $A \times \{0\})$   
 $(\lambda \ a \ (p, k). \{(q, \text{count} \ (\text{condition}_1 \ A) \ (\text{item} \ (p, a, q)) \ k) \mid q. q \in \text{transition}_1$   
 $A \ a \ p\})$   
*(degen*  $(\text{condition}_1 \ A) \circ \text{translate})$

**lemma** *degeneralize-simps*[simp]:

*alphabet*<sub>2</sub> (*degeneralize* *A*) = *alphabet*<sub>1</sub> *A*

*initial*<sub>2</sub> (*degeneralize* *A*) = *initial*<sub>1</sub> *A* × {0}

*transition*<sub>2</sub> (*degeneralize* *A*) *a* (*p*, *k*) =

{(*q*, *count* (*condition*<sub>1</sub> *A*) (*item* (*p*, *a*, *q*)) *k*) | *q*. *q* ∈ *transition*<sub>1</sub> *A* *a* *p*}

*condition*<sub>2</sub> (*degeneralize* *A*) = *degen* (*condition*<sub>1</sub> *A*) ∘ *translate*

⟨*proof*⟩

**lemma** *run-degeneralize*:

**assumes** *a.run* *A* (*w* ||| *r*) *p*

**shows** *b.run* (*degeneralize* *A*) (*w* ||| *r* ||| *sscan* (*count* (*condition*<sub>1</sub> *A*) ∘ *item*)

(*p* ## *r* ||| *w* ||| *r*) *k*) (*p*, *k*)

⟨*proof*⟩

**lemma** *degeneralize-run*:

**assumes** *b.run* (*degeneralize* *A*) (*w* ||| *rs*) *pk*

**obtains** *r* *s* *p* *k*

**where** *rs* = *r* ||| *s* *pk* = (*p*, *k*) *a.run* *A* (*w* ||| *r*) *p* *s* = *sscan* (*count* (*condition*<sub>1</sub> *A*) ∘ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*

⟨*proof*⟩

**lemma** *degeneralize-nodes*:

*b.nodes* (*degeneralize* *A*) ⊆ *a.nodes* *A* × *insert* 0 {0 ..< *length* (*condition*<sub>1</sub> *A*)}

⟨*proof*⟩

**lemma** *nodes-degeneralize*: *a.nodes* *A* ⊆ *fst* ‘ *b.nodes* (*degeneralize* *A*)

⟨*proof*⟩

**lemma** *degeneralize-nodes-finite*[iff]: *finite* (*b.nodes* (*degeneralize* *A*)) ↔ *finite* (*a.nodes* *A*)

⟨*proof*⟩

**end**

**locale** *automaton-degeneralization-run* =

*automaton-degeneralization*

*automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>

*automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>

*item* *translate* +

*a*: *automaton-run* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub> +

*b*: *automaton-run* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>

**for** *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub>

**and** *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>

**and** *item* *translate*

+

**assumes** *test*[iff]: *test*<sub>2</sub> (*degen* *cs* ∘ *translate*) *w*

(*r* ||| *sscan* (*count* *cs* ∘ *item*) (*p* ## *r* ||| *w* ||| *r*) *k*) (*p*, *k*) ↔ *test*<sub>1</sub> *cs* *w* *r* *p*

**begin**

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

A

⟨proof⟩

**end**

**locale** *automaton-product* =

*a*: *automaton* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> +

*b*: *automaton* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> +

*c*: *automaton* *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>

**for** *automaton*<sub>1</sub> :: 'label set ⇒ 'state<sub>1</sub> set ⇒ ('label ⇒ 'state<sub>1</sub> ⇒ 'state<sub>1</sub> set) ⇒  
'condition<sub>1</sub> ⇒ 'automaton<sub>1</sub>

**and** *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>

**and** *automaton*<sub>2</sub> :: 'label set ⇒ 'state<sub>2</sub> set ⇒ ('label ⇒ 'state<sub>2</sub> ⇒ 'state<sub>2</sub> set)  
⇒ 'condition<sub>2</sub> ⇒ 'automaton<sub>2</sub>

**and** *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>

**and** *automaton*<sub>3</sub> :: 'label set ⇒ ('state<sub>1</sub> × 'state<sub>2</sub>) set ⇒ ('label ⇒ 'state<sub>1</sub> ×  
'state<sub>2</sub> ⇒ ('state<sub>1</sub> × 'state<sub>2</sub>) set) ⇒ 'condition<sub>3</sub> ⇒ 'automaton<sub>3</sub>

**and** *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>

+

**fixes** *condition* :: 'condition<sub>1</sub> ⇒ 'condition<sub>2</sub> ⇒ 'condition<sub>3</sub>

**begin**

**definition** *product* :: 'automaton<sub>1</sub> ⇒ 'automaton<sub>2</sub> ⇒ 'automaton<sub>3</sub> **where**

*product* *A* *B* ≡ *automaton*<sub>3</sub>

(*alphabet*<sub>1</sub> *A* ∩ *alphabet*<sub>2</sub> *B*)

(*initial*<sub>1</sub> *A* × *initial*<sub>2</sub> *B*)

(λ *a* (*p*, *q*). *transition*<sub>1</sub> *A* *a* *p* × *transition*<sub>2</sub> *B* *a* *q*)

(*condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*))

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

*alphabet*<sub>3</sub> (*product* *A* *B*) = *alphabet*<sub>1</sub> *A* ∩ *alphabet*<sub>2</sub> *B*

*initial*<sub>3</sub> (*product* *A* *B*) = *initial*<sub>1</sub> *A* × *initial*<sub>2</sub> *B*

*transition*<sub>3</sub> (*product* *A* *B*) *a* (*p*, *q*) = *transition*<sub>1</sub> *A* *a* *p* × *transition*<sub>2</sub> *B* *a* *q*

*condition*<sub>3</sub> (*product* *A* *B*) = *condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*)

⟨proof⟩

**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*)

⟨proof⟩

**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*

⟨proof⟩

**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⟩

**lemma** *product-nodes*:  $c.nodes (product A B) \subseteq a.nodes A \times b.nodes B$   
*<proof>*

**lemma** *product-nodes-finite*[intro]:  
**assumes** *finite* ( $a.nodes A$ ) *finite* ( $b.nodes B$ )  
**shows** *finite* ( $c.nodes (product A B)$ )  
*<proof>*

**end**

**locale** *automaton-intersection-path* =  
*automaton-product*  
*automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>  
*automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
*automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>  
*condition* +  
*a*: *automaton-path* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub> +  
*b*: *automaton-path* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub> +  
*c*: *automaton-path* *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub> *test*<sub>3</sub>  
**for** *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub>  
**and** *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>  
**and** *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub> *test*<sub>3</sub>  
**and** *condition*  
+  
**assumes** *test*[*iff*]:  $length\ r = length\ w \implies length\ s = length\ w \implies$   
 $test_3 (condition\ c_1\ c_2)\ w\ (r\ ||\ s)\ (p,\ q) \longleftrightarrow test_1\ c_1\ w\ r\ p \wedge test_2\ c_2\ w\ s\ q$   
**begin**

**lemma** *product-language*[*simp*]:  $c.language (product A B) = a.language A \cap b.language B$   
*<proof>*

**end**

**locale** *automaton-intersection-run* =  
*automaton-product*  
*automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>  
*automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>  
*automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>  
*condition* +  
*a*: *automaton-run* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub> +  
*b*: *automaton-run* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub> +  
*c*: *automaton-run* *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub> *test*<sub>3</sub>  
**for** *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> *test*<sub>1</sub>  
**and** *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> *test*<sub>2</sub>  
**and** *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub> *test*<sub>3</sub>  
**and** *condition*  
+

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

**begin**

**lemma** *product-language*[*simp*]: *c.language* (*product* *A* *B*) = *a.language* *A*  $\cap$   
*b.language* *B*  
 ⟨*proof*⟩

**end**

**locale** *automaton-sum* =

*a*: *automaton* *automaton*<sub>1</sub> *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub> +

*b*: *automaton* *automaton*<sub>2</sub> *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub> +

*c*: *automaton* *automaton*<sub>3</sub> *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>

**for** *automaton*<sub>1</sub> :: 'label set  $\Rightarrow$  'state<sub>1</sub> set  $\Rightarrow$  ('label  $\Rightarrow$  'state<sub>1</sub>  $\Rightarrow$  'state<sub>1</sub> set)  $\Rightarrow$   
 'condition<sub>1</sub>  $\Rightarrow$  'automaton<sub>1</sub>

**and** *alphabet*<sub>1</sub> *initial*<sub>1</sub> *transition*<sub>1</sub> *condition*<sub>1</sub>

**and** *automaton*<sub>2</sub> :: 'label set  $\Rightarrow$  'state<sub>2</sub> set  $\Rightarrow$  ('label  $\Rightarrow$  'state<sub>2</sub>  $\Rightarrow$  'state<sub>2</sub> set)  
 $\Rightarrow$  'condition<sub>2</sub>  $\Rightarrow$  'automaton<sub>2</sub>

**and** *alphabet*<sub>2</sub> *initial*<sub>2</sub> *transition*<sub>2</sub> *condition*<sub>2</sub>

**and** *automaton*<sub>3</sub> :: 'label set  $\Rightarrow$  ('state<sub>1</sub> + 'state<sub>2</sub>) set  $\Rightarrow$  ('label  $\Rightarrow$  'state<sub>1</sub> +  
 'state<sub>2</sub>  $\Rightarrow$  ('state<sub>1</sub> + 'state<sub>2</sub>) set)  $\Rightarrow$  'condition<sub>3</sub>  $\Rightarrow$  'automaton<sub>3</sub>

**and** *alphabet*<sub>3</sub> *initial*<sub>3</sub> *transition*<sub>3</sub> *condition*<sub>3</sub>

+

**fixes** *condition* :: 'condition<sub>1</sub>  $\Rightarrow$  'condition<sub>2</sub>  $\Rightarrow$  'condition<sub>3</sub>

**begin**

**definition** *sum* :: 'automaton<sub>1</sub>  $\Rightarrow$  'automaton<sub>2</sub>  $\Rightarrow$  'automaton<sub>3</sub> **where**

*sum* *A* *B*  $\equiv$  *automaton*<sub>3</sub>

(*alphabet*<sub>1</sub> *A*  $\cup$  *alphabet*<sub>2</sub> *B*)

(*initial*<sub>1</sub> *A* <+> *initial*<sub>2</sub> *B*)

( $\lambda$  *a*.  $\lambda$  *Inl* *p*  $\Rightarrow$  *Inl* ' *transition*<sub>1</sub> *A* *a* *p* | *Inr* *q*  $\Rightarrow$  *Inr* ' *transition*<sub>2</sub> *B* *a* *q*)

(*condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*))

**lemma** *sum-simps*[*simp*]:

*alphabet*<sub>3</sub> (*sum* *A* *B*) = *alphabet*<sub>1</sub> *A*  $\cup$  *alphabet*<sub>2</sub> *B*

*initial*<sub>3</sub> (*sum* *A* *B*) = *initial*<sub>1</sub> *A* <+> *initial*<sub>2</sub> *B*

*transition*<sub>3</sub> (*sum* *A* *B*) *a* (*Inl* *p*) = *Inl* ' *transition*<sub>1</sub> *A* *a* *p*

*transition*<sub>3</sub> (*sum* *A* *B*) *a* (*Inr* *q*) = *Inr* ' *transition*<sub>2</sub> *B* *a* *q*

*condition*<sub>3</sub> (*sum* *A* *B*) = *condition* (*condition*<sub>1</sub> *A*) (*condition*<sub>2</sub> *B*)

⟨*proof*⟩

**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*)

⟨*proof*⟩

**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*)

⟨proof⟩  
**lemma** *sum-path*:  
 assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$   
 assumes  $\text{length } rs = \text{length } w.c.\text{path } (\text{sum } A B) (w \parallel rs) pq$   
 obtains  
 (a)  $r p$  **where**  $rs = \text{map } \text{Inl } r pq = \text{Inl } p a.\text{path } A (w \parallel r) p \mid$   
 (b)  $s q$  **where**  $rs = \text{map } \text{Inr } s pq = \text{Inr } q b.\text{path } B (w \parallel s) q$   
 ⟨proof⟩

**lemma** *run-sum-a*:  
 assumes  $a.\text{run } A (w \parallel\parallel r) p$   
 shows  $c.\text{run } (\text{sum } A B) (w \parallel\parallel \text{smap } \text{Inl } r) (\text{Inl } p)$   
 ⟨proof⟩

**lemma** *run-sum-b*:  
 assumes  $b.\text{run } B (w \parallel\parallel s) q$   
 shows  $c.\text{run } (\text{sum } A B) (w \parallel\parallel \text{smap } \text{Inr } s) (\text{Inr } q)$   
 ⟨proof⟩

**lemma** *sum-run*:  
 assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$   
 assumes  $c.\text{run } (\text{sum } A B) (w \parallel\parallel rs) pq$   
 obtains  
 (a)  $r p$  **where**  $rs = \text{smap } \text{Inl } r pq = \text{Inl } p a.\text{run } A (w \parallel\parallel r) p \mid$   
 (b)  $s q$  **where**  $rs = \text{smap } \text{Inr } s pq = \text{Inr } q b.\text{run } B (w \parallel\parallel s) q$   
 ⟨proof⟩

**lemma** *sum-nodes*:  
 assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$   
 shows  $c.\text{nodes } (\text{sum } A B) \subseteq a.\text{nodes } A <+> b.\text{nodes } B$   
 ⟨proof⟩

**lemma** *sum-nodes-finite[intro]*:  
 assumes  $\text{alphabet}_1 A = \text{alphabet}_2 B$   
 assumes  $\text{finite } (a.\text{nodes } A) \text{ finite } (b.\text{nodes } B)$   
 shows  $\text{finite } (c.\text{nodes } (\text{sum } A B))$   
 ⟨proof⟩

**end**

**locale** *automaton-union-path* =  
 automaton-sum  
 automaton<sub>1</sub> alphabet<sub>1</sub> initial<sub>1</sub> transition<sub>1</sub> condition<sub>1</sub>  
 automaton<sub>2</sub> alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub>  
 automaton<sub>3</sub> alphabet<sub>3</sub> initial<sub>3</sub> transition<sub>3</sub> condition<sub>3</sub>  
 condition +  
 a: automaton-path automaton<sub>1</sub> alphabet<sub>1</sub> initial<sub>1</sub> transition<sub>1</sub> condition<sub>1</sub> test<sub>1</sub> +  
 b: automaton-path automaton<sub>2</sub> alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub> test<sub>2</sub> +  
 c: automaton-path automaton<sub>3</sub> alphabet<sub>3</sub> initial<sub>3</sub> transition<sub>3</sub> condition<sub>3</sub> test<sub>3</sub>  
**for** automaton<sub>1</sub> alphabet<sub>1</sub> initial<sub>1</sub> transition<sub>1</sub> condition<sub>1</sub> test<sub>1</sub>  
**and** automaton<sub>2</sub> alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub> test<sub>2</sub>

```

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
<proof>

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
<proof>

end

```

```

locale automaton-product-list =
  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)  $\Rightarrow$ 
'condition1  $\Rightarrow$  'automaton1
and alphabet1 initial1 transition1 condition1

```

**and** *automaton<sub>2</sub>* :: 'label set ⇒ 'state list set ⇒ ('label ⇒ 'state list ⇒ 'state list set) ⇒ 'condition<sub>2</sub> ⇒ 'automaton<sub>2</sub>  
**and** *alphabet<sub>2</sub> initial<sub>2</sub> transition<sub>2</sub> condition<sub>2</sub>*  
 +  
**fixes** *condition* :: 'condition<sub>1</sub> list ⇒ 'condition<sub>2</sub>  
**begin**

**definition** *product* :: 'automaton<sub>1</sub> list ⇒ 'automaton<sub>2</sub> **where**

*product AA* ≡ *automaton<sub>2</sub>*  
 (∩ (alphabet<sub>1</sub> ' set AA))  
 (listset (map initial<sub>1</sub> AA))  
 (λ a ps. listset (map2 (λ A p. transition<sub>1</sub> A a p) AA ps))  
 (condition (map condition<sub>1</sub> AA))

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

*alphabet<sub>2</sub> (product AA)* = ∩ (alphabet<sub>1</sub> ' set AA)  
*initial<sub>2</sub> (product AA)* = listset (map initial<sub>1</sub> AA)  
*transition<sub>2</sub> (product AA) a ps* = listset (map2 (λ A p. transition<sub>1</sub> A a p) AA ps)  
*condition<sub>2</sub> (product AA)* = condition (map condition<sub>1</sub> AA)  
 ⟨proof⟩

**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⟩

**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⟩

**lemma** *run-product*:

**assumes** *length rs = length AA length ps = length AA*  
**assumes**  $\bigwedge k. k < \text{length } AA \implies a.\text{run } (AA ! k) (w ||| rs ! k) (ps ! k)$   
**shows** *b.run (product AA) (w ||| stranspose rs) ps*

⟨proof⟩

**lemma** *product-run*:

**assumes** *length rs = length AA length ps = length AA*  
**assumes** *b.run (product AA) (w ||| stranspose rs) ps*  
**shows**  $k < \text{length } AA \implies a.\text{run } (AA ! k) (w ||| rs ! k) (ps ! k)$

⟨proof⟩

**lemma** *product-nodes*: *b.nodes (product AA) ⊆ listset (map a.nodes AA)*

⟨proof⟩

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

```

assumes list-all (finite ◦ a.nodes) AA
shows finite (b.nodes (product AA))
  ⟨proof⟩
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⟩

```

**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]: length rs = length cs ⇒ length ps = length cs ⇒
  test2 (condition cs) w (stranspose rs) ps ↔ list-all (λ (c, r, p). test1 c w r
p) (cs || rs || ps)
begin

```

```

lemma product-language[simp]: b.language (product AA) = ∩ (a.language ‘ set
AA)
  ⟨proof⟩

```

**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]*:

$\text{alphabet}_2 (\text{sum } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$   
 $\text{initial}_2 (\text{sum } AA) = (\bigcup k < \text{length } AA. \{k\} \times \text{initial}_1 (AA ! k))$   
 $\text{transition}_2 (\text{sum } AA) a (k, p) = \{k\} \times \text{transition}_1 (AA ! k) a p$   
 $\text{condition}_2 (\text{sum } AA) = \text{condition} (\text{map condition}_1 AA)$   
 $\langle \text{proof} \rangle$

**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**  $k < \text{length } AA$   $A = AA ! k$   $b.\text{run} (\text{sum } AA) (w \parallel \text{sconst } k$   
 $\parallel s) (k, p)$   
 $\langle \text{proof} \rangle$

**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**  $r = \text{sconst } k \parallel s$   $a.\text{run} (AA ! k) (w \parallel s) p$   
 $\langle \text{proof} \rangle$

**lemma** *sum-nodes*:

**assumes**  $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$   
**shows**  $b.\text{nodes} (\text{sum } AA) \subseteq (\bigcup k < \text{length } AA. \{k\} \times a.\text{nodes} (AA ! k))$   
 $\langle \text{proof} \rangle$

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

**assumes**  $\bigcap (\text{alphabet}_1 \text{ ' set } AA) = \bigcup (\text{alphabet}_1 \text{ ' set } AA)$   
**assumes**  $\text{list-all} (\text{finite} \circ a.\text{nodes}) AA$   
**shows**  $\text{finite} (b.\text{nodes} (\text{sum } AA))$   
 $\langle \text{proof} \rangle$

**end**

**locale** *automaton-union-list-run* =

*automaton-sum-list*

$\text{automaton}_1 \text{ alphabet}_1 \text{ initial}_1 \text{ transition}_1 \text{ condition}_1$

$\text{automaton}_2 \text{ alphabet}_2 \text{ initial}_2 \text{ transition}_2 \text{ condition}_2$

$\text{condition} +$

$a: \text{automaton-run } \text{automaton}_1 \text{ alphabet}_1 \text{ initial}_1 \text{ transition}_1 \text{ condition}_1 \text{ test}_1 +$

$b: \text{automaton-run } \text{automaton}_2 \text{ alphabet}_2 \text{ initial}_2 \text{ transition}_2 \text{ condition}_2 \text{ test}_2$

**for**  $\text{automaton}_1 \text{ alphabet}_1 \text{ initial}_1 \text{ transition}_1 \text{ condition}_1 \text{ test}_1$

**and**  $\text{automaton}_2 \text{ alphabet}_2 \text{ initial}_2 \text{ transition}_2 \text{ condition}_2 \text{ test}_2$

**and**  $\text{condition}$

$+$

**assumes**  $\text{test}[iff]: k < \text{length } cs \implies \text{test}_2 (\text{condition } cs) w (\text{sconst } k \parallel r) (k,$

```

p)  $\longleftrightarrow$  test1 (cs ! k) w r p
begin

  lemma sum-language[simp]:
    assumes  $\bigcap$  (alphabet1 ' set AA) =  $\bigcup$  (alphabet1 ' set AA)
    shows b.language (sum AA) =  $\bigcup$  (a.language ' set AA)
    <proof>

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
  <proof>

global-interpretation nfa: automaton-path nfa alphabet initial transition ac-
cepting  $\lambda$  P w r p. P (last (p # r))
  defines language = nfa.language
  <proof>

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))
   $\lambda$  c1 c2 (p, q). c1 p  $\wedge$  c2 q
  defines intersect = nfa.product
  <proof>

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
<proof>

```

**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
  <proof>
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
  <proof>

```

```

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*

*<proof>*

**global-interpretation** *dgba: automaton-run dgba alphabet initial transition ac-*  
*cepting  $\lambda P w r p$ . gen infs  $P (p \#\# r)$*

**defines** *language = dgba.language*

*<proof>*

**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  $\lambda P w r p$ . gen infs*  
 *$P (p \#\# r)$*

*dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p$ . infs  $P (p$*   
 *$\#\# r)$*

*fst id*

**defines** *degeneralize = degeneralization.degeneralize*

*<proof>*

**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  $\lambda P w r p$ . infs  $P$*   
 *$(p \#\# r)$*

*dba.dba dba.alphabet dba.initial dba.transition dba.accepting  $\lambda P w r p$ . infs  $P$*   
 *$(p \#\# r)$*

*dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting  $\lambda P w r p$ .*  
*gen infs  $P (p \#\# r)$*

*$\lambda c_1 c_2. [c_1 \circ fst, c_2 \circ snd]$*

**defines** *intersect' = intersection.product*

*<proof>*

**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*  
*<proof>*

**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*  $\lambda P w r p. \text{gen infs } P$  (*p ## r*)  
 $\lambda cs. \text{map } (\lambda k pp. (cs ! k) (pp ! k)) [0 ..< \text{length } cs]$   
**defines** *intersect-list'* = *intersection-list.product*  
*<proof>*

**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*  
*<proof>*

**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*  $intersect\ A\ B \equiv degeneralize\ (intersect'\ A\ B)$

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

*<proof>*

**lemma** *intersect-nodes-finite*:

**assumes**  $finite\ (DBA.nodes\ A)\ finite\ (DBA.nodes\ B)$

**shows**  $finite\ (DBA.nodes\ (intersect\ A\ B))$

*<proof>*

**lemma** *intersect-nodes-card*:

**assumes**  $finite\ (DBA.nodes\ A)\ finite\ (DBA.nodes\ B)$

**shows**  $card\ (DBA.nodes\ (intersect\ A\ B)) \leq 2 * card\ (DBA.nodes\ A) * card\ (DBA.nodes\ B)$

*<proof>*

**abbreviation** *intersect-list where*  $intersect-list\ AA \equiv degeneralize\ (intersect-list'\ AA)$

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

*<proof>*

**lemma** *intersect-list-nodes-finite*:

**assumes**  $list-all\ (finite\ o\ DBA.nodes)\ AA$

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

*<proof>*

**lemma** *intersect-list-nodes-card*:

**assumes**  $list-all\ (finite\ o\ DBA.nodes)\ AA$

**shows**  $card\ (DBA.nodes\ (intersect-list\ AA)) \leq max\ 1\ (length\ AA) * prod-list\ (map\ (card\ o\ DBA.nodes)\ AA)$

*<proof>*

end

## 17 Deterministic Büchi Transition Automata

**theory** *DBTA*

**imports** *../Deterministic*

**begin**

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

(*alphabet: 'label set*)

(*initial: 'state*)

(*transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  'state*)

(*accepting: ('state  $\times$  'label  $\times$  '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
  <proof>
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
  <proof>

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
  <proof>
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
  <proof>

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*  $\lambda P w r p. \text{gen}$   
*infs*  $P (p \#\# r \|\| w \|\| r)$   
*dbta dbta.alphabet dbta.initial dbta.transition dbta.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*  
 $\langle \text{proof} \rangle$

**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*  $\lambda P w r p. \text{infs}$   
 $P (p \#\# r \|\| w \|\| r)$   
*dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting*  $\lambda P w r p. \text{infs}$   
 $P (p \#\# r \|\| w \|\| r)$   
*dgba.dgba dgba.alphabet dgba.initial dgba.transition dgba.accepting*  $\lambda P w r$   
 $p. \text{gen infs } P (p \#\# r \|\| w \|\| r)$   
 $\lambda c_1 c_2. [c_1 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1)), c_2 \circ (\lambda ((p_1, p_2), a, (q_1,$   
 $q_2)). (p_2, a, q_2))]$   
**defines** *intersect'* = *intersection.product*  
 $\langle \text{proof} \rangle$

**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*  $\lambda P w r p. \text{infs}$   
 $P (p \#\# r \|\| w \|\| r)$   
*dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting*  $\lambda P w r p. \text{infs}$   
 $P (p \#\# r \|\| w \|\| r)$   
*dbta.dbta dbta.alphabet dbta.initial dbta.transition dbta.accepting*  $\lambda P w r p. \text{infs}$   
 $P (p \#\# r \|\| w \|\| r)$   
 $\lambda c_1 c_2 pq. (c_1 \circ (\lambda ((p_1, p_2), a, (q_1, q_2)). (p_1, a, q_1))) pq \vee (c_2 \circ (\lambda ((p_1, p_2),$

```

a, (q1, q2)). (p2, a, q2))) pq
defines union = union.product
⟨proof⟩

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 ≡ degeneralize (intersect' A B)

lemma intersect-language[simp]: DBTA.language (intersect A B) = DBTA.language
A ∩ DBTA.language B
⟨proof⟩
lemma intersect-nodes-finite:
assumes finite (DBTA.nodes A) finite (DBTA.nodes B)
shows finite (DBTA.nodes (intersect A B))
⟨proof⟩
lemma intersect-nodes-card:
assumes finite (DBTA.nodes A) finite (DBTA.nodes B)
shows card (DBTA.nodes (intersect A B)) ≤ 2 * card (DBTA.nodes A) * card
(DBTA.nodes B)
⟨proof⟩

end

```

## 20 Deterministic Co-Büchi Automata

```

theory DCA
imports ../Deterministic
begin

datatype ('label, 'state) dca = dca
  (alphabet: 'label set)
  (initial: 'state)
  (transition: 'label ⇒ 'state ⇒ 'state)
  (rejecting: 'state ⇒ 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
⟨proof⟩
global-interpretation dca: automaton-run dca alphabet initial transition reject-
ing λ P w r p. fins P (p ## r)
defines language = dca.language
⟨proof⟩

abbreviation target where target ≡ dca.target
abbreviation states where states ≡ dca.states
abbreviation trace where trace ≡ 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*  
  <*proof*>

**global-interpretation** *dgca*: *automaton-run dgca alphabet initial transition re-*  
*jecting*  $\lambda$  *P w r p. cogen fins P (p ## r)*  
**defines** *language* = *dgca.language*  
  <*proof*>

**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*  $\lambda$  *P w r p. cogen*  
*fins P (p ## r)*  
*dca dca.alphabet dca.initial dca.transition dca.rejecting*  $\lambda$  *P w r p. fins P (p ##*  
*r)*  
  *fst id*  
**defines** *degeneralize* = *degeneralization.degeneralize*  
  <*proof*>

**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*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
*dca.dca dca.alphabet dca.initial dca.transition dca.rejecting*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
*dca.dca dca.alphabet dca.initial dca.transition dca.rejecting*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
 $\lambda c_1 c_2 pq. (c_1 \circ \text{fst}) pq \vee (c_2 \circ \text{snd}) pq$   
**defines** *intersect* = *intersection.product*  
*<proof>*

**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*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
*dca.dca dca.alphabet dca.initial dca.transition dca.rejecting*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
*dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting*  $\lambda P w r p. \text{cogen fins } P (p \text{ ## } r)$   
 $\lambda c_1 c_2. [c_1 \circ \text{fst}, c_2 \circ \text{snd}]$   
**defines** *union'* = *union.product*  
*<proof>*

**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*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
*dca.dca dca.alphabet dca.initial dca.transition dca.rejecting*  $\lambda P w r p. \text{fins } P (p \text{ ## } r)$   
 $\lambda cs pp. \exists k < \text{length } cs. (cs ! k) (pp ! k)$   
**defines** *intersect-list* = *intersection-list.product*  
*<proof>*

**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*  $\lambda P w r p. \text{fins } P (p \text{ \#\# } r)$   
*dgca.dgca dgca.alphabet dgca.initial dgca.transition dgca.rejecting*  $\lambda P w r p. \text{cogen fins } P (p \text{ \#\# } r)$   
 $\lambda cs. \text{map } (\lambda k pp. (cs ! k) (pp ! k)) [0 ..< \text{length } cs]$   
**defines** *union-list'* = *union-list.product*  
 $\langle \text{proof} \rangle$

**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*  $\equiv \text{degeneralize } (\text{union}' A B)$

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

**assumes** *dca.alphabet A = dca.alphabet B*

**shows** *DCA.language (union A B) = DCA.language A  $\cup$  DCA.language B*

$\langle \text{proof} \rangle$

**lemma** *union-nodes-finite*:

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

**shows** *finite (DCA.nodes (union A B))*

$\langle \text{proof} \rangle$

**lemma** *union-nodes-card*:

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

**shows** *card (DCA.nodes (union A B))  $\leq 2 * \text{card } (DCA.nodes A) * \text{card } (DCA.nodes B)$*

$\langle \text{proof} \rangle$

**abbreviation** *union-list where union-list AA*  $\equiv \text{degeneralize } (\text{union-list}' AA)$

**lemma** *union-list-language[simp]*:

**assumes**  $\bigcap (dca.alphabet \text{ ' set } AA) = \bigcup (dca.alphabet \text{ ' set } AA)$

**shows** *DCA.language (union-list AA) =  $\bigcup (DCA.language \text{ ' set } AA)$*

$\langle \text{proof} \rangle$

**lemma** *union-list-nodes-finite*:

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

**shows** *finite (DCA.nodes (union-list AA))*

$\langle \text{proof} \rangle$

**lemma** *union-list-nodes-card*:

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

**shows** *card (DCA.nodes (union-list AA))  $\leq \text{max } 1 (\text{length } AA) * \text{prod-list } (\text{map } (\text{card } \circ \text{DCA.nodes}) AA)$*

$\langle \text{proof} \rangle$

**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
    <proof>
  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
    <proof>

  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
    <proof>

  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*  $\lambda P w r p$ . *cogen*  
*rabin P (p ## r)*  
*dra.dra dra.alphabet dra.initial dra.transition dra.condition*  $\lambda P w r p$ . *cogen*  
*rabin P (p ## r)*  
 $\lambda cs$ . *do* {  $k \leftarrow [0 ..< \text{length } cs]$ ;  $(f, g) \leftarrow cs ! k$ ;  $[(\lambda pp$ .  $f (pp ! k)$ ,  $\lambda pp$ .  $g (pp$   
 $! k))]$  }  
**defines** *union-list = union-list.product*  
 $\langle \text{proof} \rangle$

**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** *bi-unique-pred-set-conv*[*pred-set-conv*]: *bi-unique*  $(\lambda x y. (x, y) \in R) \longleftrightarrow$   
*bijective R*  
 $\langle \text{proof} \rangle$

useful for unfolding equality constants in theorems about predicates

**lemma** *pred-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in \text{Id})$   $\langle \text{proof} \rangle$   
**lemma** *pred-bool-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in (\text{Id} :: \text{bool rel}))$   $\langle \text{proof} \rangle$   
**lemma** *pred-nat-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in (\text{Id} :: \text{nat rel}))$   $\langle \text{proof} \rangle$   
**lemma** *pred-set-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in (\text{Id} :: 'a \text{ set rel}))$   $\langle \text{proof} \rangle$   
**lemma** *pred-list-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in (\text{Id} :: 'a \text{ list rel}))$   $\langle \text{proof} \rangle$   
**lemma** *pred-stream-Id*: *HOL.eq* =  $(\lambda x y. (x, y) \in (\text{Id} :: 'a \text{ stream rel}))$   $\langle \text{proof} \rangle$

**lemma** *eq-onp-Id-on-eq*[*pred-set-conv*]: *eq-onp*  $(\lambda a. a \in A) = (\lambda x y. (x, y) \in$   
*Id-on A*)  
 $\langle \text{proof} \rangle$

**lemma** *rel-fun-fun-rel-eq*[*pred-set-conv*]:

*rel-fun*  $(\lambda x y. (x, y) \in A) (\lambda x y. (x, y) \in B) = (\lambda f g. (f, g) \in A \rightarrow B)$   
*<proof>*

**lemma** *rel-prod-prod-rel-eq*[*pred-set-conv*]:

*rel-prod*  $(\lambda x y. (x, y) \in A) (\lambda x y. (x, y) \in B) = (\lambda f g. (f, g) \in A \times_r B)$   
*<proof>*

**lemma** *rel-sum-sum-rel-eq*[*pred-set-conv*]:

*rel-sum*  $(\lambda x y. (x, y) \in A) (\lambda x y. (x, y) \in B) = (\lambda f g. (f, g) \in \langle A, B \rangle \text{ sum-rel})$   
*<proof>*

**lemma** *rel-set-set-rel-eq*[*pred-set-conv*]:

*rel-set*  $(\lambda x y. (x, y) \in A) = (\lambda f g. (f, g) \in \langle A \rangle \text{ set-rel})$   
*<proof>*

**lemma** *rel-option-option-rel-eq*[*pred-set-conv*]:

*rel-option*  $(\lambda x y. (x, y) \in A) = (\lambda f g. (f, g) \in \langle A \rangle \text{ option-rel})$   
*<proof>*

**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**  $(f, h) \in A_1 \circ A_2$   
*<proof>*

**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>*

**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>*

**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**  $(f, h) \in A_1 \circ A_2 \rightarrow B_1 \circ B_2 \rightarrow C_1 \circ C_2 \rightarrow D_1 \circ D_2$   
*<proof>*

**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**  $(f, h) \in A_1 \circ A_2 \rightarrow B_1 \circ B_2 \rightarrow C_1 \circ C_2 \rightarrow D_1 \circ D_2 \rightarrow E_1 \circ E_2$   
*<proof>*

### 25.3 Relation Basics

**lemma** *inv-fun-rel-eq*[simp]:  $(A \rightarrow B)^{-1} = A^{-1} \rightarrow B^{-1}$   
 ⟨proof⟩  
**lemma** *inv-option-rel-eq*[simp]:  $(\langle K \rangle \text{option-rel})^{-1} = \langle K^{-1} \rangle \text{option-rel}$   
 ⟨proof⟩  
**lemma** *inv-prod-rel-eq*[simp]:  $(P \times_r Q)^{-1} = P^{-1} \times_r Q^{-1}$   
 ⟨proof⟩  
**lemma** *inv-sum-rel-eq*[simp]:  $(\langle P, Q \rangle \text{sum-rel})^{-1} = \langle P^{-1}, Q^{-1} \rangle \text{sum-rel}$   
 ⟨proof⟩  
**lemma** *set-rel-converse*[simp]:  $(\langle A \rangle \text{set-rel})^{-1} = \langle A^{-1} \rangle \text{set-rel}$  ⟨proof⟩

**lemma** *build-rel-domain*[simp]:  $\text{Domain } (br \ \alpha \ I) = \text{Collect } I$  ⟨proof⟩  
**lemma** *build-rel-range*[simp]:  $\text{Range } (br \ \alpha \ I) = \alpha \ ' \ \text{Collect } I$  ⟨proof⟩  
**lemma** *build-rel-image*[simp]:  $br \ \alpha \ I \ ' \ A = \alpha \ ' \ (A \cap \text{Collect } I)$  ⟨proof⟩

**lemma** *prod-rel-domain*[simp]:  $\text{Domain } (A \times_r B) = \text{Domain } A \times \text{Domain } B$   
 ⟨proof⟩  
**lemma** *prod-rel-range*[simp]:  $\text{Range } (A \times_r B) = \text{Range } A \times \text{Range } B$  ⟨proof⟩

**lemma** *member-Id-on*[iff]:  $(x, y) \in \text{Id-on } A \iff x = y \wedge y \in A$  ⟨proof⟩  
**lemma** *bijjective-Id-on*[intro!, simp]: *bijjective*  $(\text{Id-on } A)$  ⟨proof⟩  
**lemma** *relcomp-Id-on*[simp]:  $\text{Id-on } A \circ \text{Id-on } B = \text{Id-on } (A \cap B)$  ⟨proof⟩

**lemma** *prod-rel-Id-on*[simp]:  $\text{Id-on } A \times_r \text{Id-on } B = \text{Id-on } (A \times B)$  ⟨proof⟩  
**lemma** *set-rel-Id-on*[simp]:  $\langle \text{Id-on } S \rangle \text{set-rel} = \text{Id-on } (\text{Pow } S)$  ⟨proof⟩

### 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]:  $(\text{Sigma}, \text{Sigma}) \in \langle A \rangle \text{set-rel} \rightarrow (A \rightarrow \langle B \rangle \text{set-rel})$   
 $\rightarrow \langle A \times_r B \rangle \text{set-rel}$   
 ⟨proof⟩

**lemma** *set-filter-param*[param]:  
 $(\text{Set.filter}, \text{Set.filter}) \in (A \rightarrow \text{bool-rel}) \rightarrow \langle A \rangle \text{set-rel} \rightarrow \langle A \rangle \text{set-rel}$   
 ⟨proof⟩

**lemma** *is-singleton-param*[param]:  
**assumes** *bijjective*  $A$   
**shows**  $(\text{is-singleton}, \text{is-singleton}) \in \langle A \rangle \text{set-rel} \rightarrow \text{bool-rel}$   
 ⟨proof⟩

**lemma** *the-elem-param*[param]:  
**assumes** *is-singleton*  $S$  *is-singleton*  $T$

**assumes**  $(S, T) \in \langle A \rangle$  *set-rel*  
**shows**  $(\text{the-elem } S, \text{the-elem } T) \in A$   
 $\langle \text{proof} \rangle$

## 25.5 Lists

**lemma** *list-all2-list-rel-conv*[*pred-set-conv*]:  
 $\text{list-all2 } (\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in \langle R \rangle \text{ list-rel})$   
 $\langle \text{proof} \rangle$

**lemmas** *list-rel-single-valued*[*iff*] = *list-rel-sv-iff*

**lemmas** *list-rel-simps*[*simp*] =  
 $\text{list.rel-eq-onp}$ [*to-set*]  
 $\text{list.rel-conversep}$ [*to-set, symmetric*]  
 $\text{list.rel-compp}$ [*to-set*]

**lemmas** *list-rel-param*[*param*] =  
 $\text{list.set-transfer}$ [*to-set*]  
 $\text{list.pred-transfer}$ [*unfolded pred-bool-Id, to-set, folded pred-list-listsp*]  
 $\text{list.rel-transfer}$ [*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*]:  $(\text{List.bind}, \text{List.bind}) \in \langle A \rangle \text{ list-rel} \rightarrow (A \rightarrow \langle B \rangle \text{ list-rel}) \rightarrow \langle B \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *set-id-param*[*param*]:  $(\text{set}, \text{id}) \in \langle A \rangle \text{ list-set-rel} \rightarrow \langle A \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

## 25.6 Streams

**definition** *stream-rel* ::  $('a \times 'b) \text{ set} \Rightarrow ('a \text{ stream} \times 'b \text{ stream}) \text{ set}$  **where**  
 $[\text{to-relAPP}]$ :  $\text{stream-rel } R \equiv \{(x, y). \text{stream-all2 } (\lambda x y. (x, y) \in R) x y\}$

**lemma** *stream-all2-stream-rel-conv*[*pred-set-conv*]:  
 $\text{stream-all2 } (\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in \langle R \rangle \text{ stream-rel})$   
 $\langle \text{proof} \rangle$

**lemmas** *stream-rel-coinduct'*[*case-names stream-rel, coinduct set: stream-rel*] =  
 $\text{stream-rel-coinduct}$ [*to-set*]

**lemmas** *stream-rel-intros* =  $\text{stream.rel-intros}$ [*to-set*]

**lemmas** *stream-rel-cases* =  $\text{stream.rel-cases}$ [*to-set*]

**lemmas** *stream-rel-inject*[*iff*] =  $\text{stream.rel-inject}$ [*to-set*]

**lemma** *stream-rel-single-valued*[*iff*]: *single-valued* ( $\langle A \rangle$  *stream-rel*)  $\longleftrightarrow$  *single-valued*  
 $A$   
 $\langle$ *proof* $\rangle$

**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*  $R$ )  
 $\langle$ *proof* $\rangle$

**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*)  $\in \langle R \rangle$  *stream-rel*  $\rightarrow \langle R \rangle$  *stream-rel*  $\rightarrow \text{bool-rel}$   
 $\langle$ *proof* $\rangle$

**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*)  $\in \langle A \rangle$  *set-rel*  $\rightarrow \langle \langle A \rangle$  *stream-rel* $\rangle$   
*set-rel*  
 $\langle$ *proof* $\rangle$

**lemma** *holds-param*[*param*]: (*holds, holds*)  $\in (A \rightarrow \text{bool-rel}) \rightarrow \langle \langle A \rangle$  *stream-rel* $\rangle$   
 $\rightarrow \text{bool-rel}$   
 $\langle$ *proof* $\rangle$

**lemma** *HLD-param*[*param*]:  
**assumes** *single-valued A single-valued* ( $A^{-1}$ )  
**shows** (*HLD, HLD*)  $\in \langle A \rangle$  *set-rel*  $\rightarrow \langle A \rangle$  *stream-rel*  $\rightarrow \text{bool-rel}$   
 $\langle$ *proof* $\rangle$

**lemma** *ev-param*[*param*]:  $(ev, ev) \in (\langle A \rangle \text{ stream-rel} \rightarrow \text{bool-rel}) \rightarrow (\langle A \rangle \text{ stream-rel} \rightarrow \text{bool-rel})$

*<proof>*

**lemma** *alw-param*[*param*]:  $(alw, alw) \in (\langle A \rangle \text{ stream-rel} \rightarrow \text{bool-rel}) \rightarrow (\langle A \rangle \text{ stream-rel} \rightarrow \text{bool-rel})$

*<proof>*

## 25.7 Functional Relations

**lemma** *br-set-rel*:  $\langle br\ f\ P \rangle \text{ set-rel} = br\ (\text{image}\ f)\ (\lambda\ A.\ \text{Ball}\ A\ P)$

*<proof>*

**lemma** *br-list-rel*:  $\langle br\ f\ P \rangle \text{ list-rel} = br\ (\text{map}\ f)\ (\text{list-all}\ P)$

*<proof>*

**lemma** *br-list-set-rel*:  $\langle br\ f\ P \rangle \text{ list-set-rel} = br\ (\text{set} \circ \text{map}\ f)\ (\lambda\ s.\ \text{list-all}\ P\ s \wedge \text{distinct}\ (\text{map}\ f\ s))$

*<proof>*

**lemma** *br-fun-rel1*:  $\text{Id} \rightarrow br\ f\ P = br\ (\text{comp}\ f)\ (\text{All} \circ \text{comp}\ P)$

*<proof>*

**term**  $\text{set} \circ \text{map}\ f \circ \text{map}\ g \circ \text{map}\ h$

**term**  $\text{set} \circ \text{sort}$

**end**

**theory** *Acceptance-Refine*

**imports** *Acceptance Refine*

**begin**

**abbreviation** (*input*)  $\text{pred-rel}\ A \equiv A \rightarrow \text{bool-rel}$

**abbreviation** (*input*)  $\text{rabin-rel}\ A \equiv \text{pred-rel}\ A \times_{\tau} \text{pred-rel}\ A$

**lemma** *rabin-param*[*param*]:  $(rabin, rabin) \in \text{rabin-rel}\ A \rightarrow \text{pred-rel}\ (\langle A \rangle \text{ stream-rel})$

*<proof>*

**lemma** *gen-param*[*param*]:  $(gen, gen) \in (A \rightarrow \text{pred-rel}\ B) \rightarrow (\langle A \rangle \text{ list-rel} \rightarrow \text{pred-rel}\ B)$

*<proof>*

**lemma** *cogen-param*[*param*]:  $(cogen, cogen) \in (A \rightarrow \text{pred-rel}\ B) \rightarrow (\langle A \rangle \text{ list-rel}$

→ *pred-rel B*)  
 ⟨*proof*⟩

**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*) ∈  
 ( $T \rightarrow S \rightarrow S$ ) → ( $T \rightarrow S \rightarrow \text{bool-rel}$ ) → ⟨ $T$ ⟩ *list-rel* →  $S \rightarrow \text{bool-rel}$   
 ⟨*proof*⟩

**lemma** *run-param*[*param*]: (*transition-system.run*, *transition-system.run*) ∈  
 ( $T \rightarrow S \rightarrow S$ ) → ( $T \rightarrow S \rightarrow \text{bool-rel}$ ) → ⟨ $T$ ⟩ *stream-rel* →  $S \rightarrow \text{bool-rel}$   
 ⟨*proof*⟩

**lemma** *paths-param*[*param*]:  
**assumes** [*param*]: (*exa*, *exb*) ∈  $T \rightarrow S \rightarrow S$   
**assumes** (*transition-system.enableds ena*, *transition-system.enableds enb*) ∈  $S$   
 → ⟨ $T$ ⟩ *set-rel*  
**shows** (*transition-system.paths exa ena*, *transition-system.paths exb enb*) ∈  $S$   
 → ⟨⟨ $T$ ⟩ *list-rel*⟩ *set-rel*  
 ⟨*proof*⟩

**lemma** *runs-param*[*param*]:  
**assumes** (*exa*, *exb*) ∈  $T \rightarrow S \rightarrow S$   
**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*⟩

**end**

## 27 Relations on Deterministic Rabin Automata

**theory** *DRA-Refine*

**imports**

*DRA*

*../Basic/Acceptance-Refine*

*../Transition-Systems/Transition-System-Refine*

**begin**

**definition** *dra-rel* :: ( $'\text{label}_1 \times '\text{label}_2$ ) *set* ⇒ ( $'\text{state}_1 \times '\text{state}_2$ ) *set* ⇒

$((\text{'label}_1, \text{'state}_1) \text{ dra} \times (\text{'label}_2, \text{'state}_2) \text{ dra}) \text{ set where}$   
 $[\text{to-relAPP}]: \text{dra-rel } L \ S \equiv \{(A_1, A_2).$   
 $(\text{alphabet } A_1, \text{alphabet } A_2) \in \langle L \rangle \text{ set-rel} \wedge$   
 $(\text{initial } A_1, \text{initial } A_2) \in S \wedge$   
 $(\text{transition } A_1, \text{transition } A_2) \in L \rightarrow S \rightarrow S \wedge$   
 $(\text{condition } A_1, \text{condition } A_2) \in \langle \text{rabin-rel } S \rangle \text{ list-rel}\}$

**lemma**  $\text{dra-param}[\text{param}]$ :

$(\text{dra}, \text{dra}) \in \langle L \rangle \text{ set-rel} \rightarrow S \rightarrow (L \rightarrow S \rightarrow S) \rightarrow \langle \text{rabin-rel } S \rangle \text{ list-rel} \rightarrow$   
 $\langle L, S \rangle \text{ dra-rel}$   
 $(\text{alphabet}, \text{alphabet}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle L \rangle \text{ set-rel}$   
 $(\text{initial}, \text{initial}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S$   
 $(\text{transition}, \text{transition}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow L \rightarrow S \rightarrow S$   
 $(\text{condition}, \text{condition}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle \text{rabin-rel } S \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dra-rel-id}[\text{simp}]$ :  $\langle \text{Id}, \text{Id} \rangle \text{ dra-rel} = \text{Id} \langle \text{proof} \rangle$

**lemma**  $\text{dra-rel-comp}[\text{trans}]$ :

**assumes**  $[\text{param}]: (A, B) \in \langle L_1, S_1 \rangle \text{ dra-rel} (B, C) \in \langle L_2, S_2 \rangle \text{ dra-rel}$   
**shows**  $(A, C) \in \langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle \text{ dra-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dra-rel-converse}[\text{simp}]$ :  $(\langle L, S \rangle \text{ dra-rel})^{-1} = \langle L^{-1}, S^{-1} \rangle \text{ dra-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{dra-rel-eq}$ :  $(A, A) \in \langle \text{Id-on } (\text{alphabet } A), \text{Id-on } (\text{nodes } A) \rangle \text{ dra-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{enableds-param}[\text{param}]$ :  $(\text{dra.enableds}, \text{dra.enableds}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow$   
 $S \rightarrow \langle L \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{paths-param}[\text{param}]$ :  $(\text{dra.paths}, \text{dra.paths}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow \langle \langle L \rangle$   
 $\text{list-rel} \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{runs-param}[\text{param}]$ :  $(\text{dra.runs}, \text{dra.runs}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow \langle \langle L \rangle$   
 $\text{stream-rel} \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{reachable-param}[\text{param}]$ :  $(\text{reachable}, \text{reachable}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow S \rightarrow$   
 $\langle S \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{nodes-param}[\text{param}]$ :  $(\text{nodes}, \text{nodes}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle S \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{language-param}[\text{param}]$ :  $(\text{language}, \text{language}) \in \langle L, S \rangle \text{ dra-rel} \rightarrow \langle \langle L \rangle$   
 $\text{stream-rel} \rangle \text{ set-rel}$   
 $\langle \text{proof} \rangle$

**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 (⟨⟨indent=2 notation=⟨infix do let⟩⟩let
- =/ -)⟩ [1000, 13] 13)
syntax -do-let :: [pttrn, 'a] ⇒ do-bind (⟨⟨indent=2 notation=⟨infix do let⟩⟩let -
= / -)⟩ 13)

```

### 28.2 Monadic Refinement

```

lemmas [refine] = plain-nres-relI

```

```

lemma vcg0:
  assumes (f, g) ∈ ⟨Id⟩ nres-rel
  shows g ≤ h ⇒ f ≤ h
  ⟨proof⟩

```

```

lemma vcg1:
  assumes (f, g) ∈ Id → ⟨Id⟩ nres-rel
  shows g x ≤ h x ⇒ f x ≤ h x
  ⟨proof⟩

```

```

lemma vcg2:
  assumes (f, g) ∈ Id → Id → ⟨Id⟩ nres-rel
  shows g x y ≤ h x y ⇒ f x y ≤ h x y
  ⟨proof⟩

```

```

lemma RETURN-nres-relD:
  assumes (RETURN x, RETURN y) ∈ ⟨A⟩ nres-rel
  shows (x, y) ∈ A
  ⟨proof⟩

```

```

lemma FOREACH-rule-insert:
  assumes finite S
  assumes I { } s
  assumes ⋀ s. I S s ⇒ P s
  assumes ⋀ T x s. T ⊆ S ⇒ I T s ⇒ x ∈ S ⇒ x ∉ T ⇒ f x s ≤ SPEC
(I (insert x T))
  shows FOREACH S f s ≤ SPEC P
  ⟨proof⟩

```

```

lemma FOREACH-rule-map:
  assumes finite (dom g)
  assumes I Map.empty s
  assumes ⋀ s. I g s ⇒ P s

```

**assumes**  $\bigwedge h k v s. h \subseteq_m g \implies I h s \implies g k = \text{Some } v \implies k \notin \text{dom } h \implies f (k, v) s \leq \text{SPEC } (I (h (k \mapsto v)))$

**shows**  $\text{FOREACH } (\text{map-to-set } g) f s \leq \text{SPEC } P$

$\langle \text{proof} \rangle$

**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**  $\text{FOREACH } S f s \leq \text{SPEC } (\text{HOL.eq } t)$

$\langle \text{proof} \rangle$

**lemma** *FOREACH-rule-map-eq*:

**assumes** *finite*  $(\text{dom } g)$

**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**  $\text{FOREACH } (\text{map-to-set } g) f s \leq \text{SPEC } (\text{HOL.eq } t)$

$\langle \text{proof} \rangle$

**lemma** *FOREACH-rule-map-map*:  $(\text{FOREACH } (\text{map-to-set } m) (\lambda (k, v). F k (f k v)),$

$\text{FOREACH } (\text{map-to-set } (\lambda k. \text{map-option } (f k) (m k))) (\lambda (k, v). F k v) \in \text{Id} \rightarrow \langle \text{Id} \rangle \text{nres-rel}$

$\langle \text{proof} \rangle$

### 28.3 Implementations for Sets Represented by Lists

**lemma** *list-set-rel-Id-on[simp]*:  $\langle \text{Id-on } A \rangle \text{list-set-rel} = \langle \text{Id} \rangle \text{list-set-rel} \cap \text{UNIV} \times \text{Pow } A$

$\langle \text{proof} \rangle$

**lemma** *list-set-card[param]*:  $(\text{length}, \text{card}) \in \langle A \rangle \text{list-set-rel} \rightarrow \text{nat-rel}$

$\langle \text{proof} \rangle$

**lemma** *list-set-insert[param]*:

**assumes**  $y \notin Y$

**assumes**  $(x, y) \in A (xs, Y) \in \langle A \rangle \text{list-set-rel}$

**shows**  $(x \# xs, \text{insert } y Y) \in \langle A \rangle \text{list-set-rel}$

$\langle \text{proof} \rangle$

**lemma** *list-set-union[param]*:

**assumes**  $X \cap Y = \{\}$

**assumes**  $(xs, X) \in \langle A \rangle \text{list-set-rel } (ys, Y) \in \langle A \rangle \text{list-set-rel}$

**shows**  $(xs @ ys, X \cup Y) \in \langle A \rangle \text{list-set-rel}$

$\langle \text{proof} \rangle$

**lemma** *list-set-Union[param]*:

**assumes**  $\bigwedge X Y. X \in S \implies Y \in S \implies X \neq Y \implies X \cap Y = \{\}$

**assumes**  $(xs, S) \in \langle \langle A \rangle \text{list-set-rel} \rangle \text{list-set-rel}$

**shows**  $(\text{concat } xs, \text{Union } S) \in \langle A \rangle \text{list-set-rel}$

$\langle \text{proof} \rangle$

**lemma** *list-set-image*[*param*]:

**assumes** *inj-on* *g S*

**assumes**  $(f, g) \in A \rightarrow B$   $(xs, S) \in \langle A \rangle$  *list-set-rel*

**shows**  $(\text{map } f \text{ } xs, g \text{ } S) \in \langle B \rangle$  *list-set-rel*

$\langle \text{proof} \rangle$

**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**  $(xs, S) \in \langle A \rangle$  *list-set-rel*  $(f, g) \in A \rightarrow \langle B \rangle$  *list-set-rel*

**shows**  $(xs \ggg f, S \ggg g) \in \langle B \rangle$  *list-set-rel*

$\langle \text{proof} \rangle$

## 28.4 Autoref Setup

**lemma** *dflt-ahm-rel-finite-nat*: *finite-map-rel* ( $\langle \text{nat-rel}, V \rangle$  *dflt-ahm-rel*)  $\langle \text{proof} \rangle$

**context**

**begin**

**interpretation** *autoref-syn*  $\langle \text{proof} \rangle$

**lemma** [*autoref-op-pat*]:  $(\text{Some } \circ f) \mid X \equiv OP (\lambda f \ X. (\text{Some } \circ f) \mid X) f \ X$   
 $\langle \text{proof} \rangle$

**lemma** [*autoref-op-pat*]:  $\bigcup (m \text{ } S) \equiv OP (\lambda S \ m. \bigcup (m \text{ } S)) S \ m$   $\langle \text{proof} \rangle$

**definition** *gen-UNION* **where**

*gen-UNION* *tol emp un X f*  $\equiv \text{fold } (un \circ f) (tol \ X) \ emp$

**lemma** *gen-UNION*[*autoref-rules-raw*]:

**assumes** *PRIO-TAG-GEN-ALGO*

**assumes** *to-list*: *SIDE-GEN-ALGO* (*is-set-to-list* *A Rs1 tol*)

**assumes** *empty*: *GEN-OP emp*  $\{\}$  ( $\langle B \rangle$  *Rs3*)

**assumes** *union*: *GEN-OP un union* ( $\langle B \rangle$  *Rs2*  $\rightarrow$   $\langle B \rangle$  *Rs3*  $\rightarrow$   $\langle B \rangle$  *Rs3*)

**shows** (*gen-UNION* *tol emp un*,  $\lambda A \ f. \bigcup (f \text{ } A)$ )  $\in \langle A \rangle$  *Rs1*  $\rightarrow (A \rightarrow \langle B \rangle$   
*Rs2*)  $\rightarrow \langle B \rangle$  *Rs3*

$\langle \text{proof} \rangle$

**definition** *gen-Image* **where**

*gen-Image* *tol1 mem2 emp3 ins3 X Y*  $\equiv \text{fold}$

$(\lambda (a, b). \text{if } mem2 \ a \ Y \ \text{then } ins3 \ b \ \text{else } id) (tol1 \ X) \ emp3$

**lemma** *gen-Image*[*autoref-rules*]:

**assumes** *PRIO-TAG-GEN-ALGO*

**assumes** *to-list*: *SIDE-GEN-ALGO* (*is-set-to-list*  $(A \times_r B)$  *Rs1 tol1*)

**assumes** *member*: *GEN-OP mem2*  $(\in) (A \rightarrow \langle A \rangle$  *Rs2*  $\rightarrow$  *bool-rel*)

**assumes** *empty*: *GEN-OP emp3*  $\{\}$  ( $\langle B \rangle$  *Rs3*)

**assumes** *insert*: *GEN-OP ins3* *Set.insert*  $(B \rightarrow \langle B \rangle$  *Rs3*  $\rightarrow$   $\langle B \rangle$  *Rs3*)

**shows** (*gen-Image* *tol1 mem2 emp3 ins3*, *Image*)  $\in \langle A \times_r B \rangle$  *Rs1*  $\rightarrow \langle A \rangle$   
*Rs2*  $\rightarrow \langle B \rangle$  *Rs3*

$\langle \text{proof} \rangle$

**lemma** *list-set-union-autoref*[*autoref-rules*]:  
**assumes** *PRIO-TAG-OPTIMIZATION*  
**assumes** *SIDE-PRECOND-OPT* ( $a' \cap b' = \{\}$ )  
**assumes**  $(a, a') \in \langle R \rangle$  *list-set-rel*  
**assumes**  $(b, b') \in \langle R \rangle$  *list-set-rel*  
**shows**  $(a @ b,$   
 $(OP\ union :: \langle R \rangle\ list-set-rel \rightarrow \langle R \rangle\ list-set-rel \rightarrow \langle R \rangle\ list-set-rel) \$ a' \$ b')$   
 $\in$   
 $\langle R \rangle$  *list-set-rel*  
 $\langle proof \rangle$

**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) ∈ ⟨Ra⟩ list-set-rel*  
**shows** (*map*  $fi\ l,$   
 $(OP\ image :: (Ra \rightarrow Rb) \rightarrow \langle Ra \rangle\ list-set-rel \rightarrow \langle Rb \rangle\ list-set-rel) \$ f \$ s) \in$   
 $\langle Rb \rangle$  *list-set-rel*  
 $\langle proof \rangle$

**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**  $(xs, S) \in \langle A \rangle$  *list-set-rel*  $(f, g) \in A \rightarrow \langle B \rangle$  *list-set-rel*  
**shows**  $(xs \ggg f,$   
 $(OP\ (\lambda A\ f. \bigcup (f\ 'A)) :: \langle A \rangle\ list-set-rel \rightarrow (A \rightarrow \langle B \rangle\ list-set-rel) \rightarrow \langle B \rangle$   
 $list-set-rel) \$ S \$ g) \in$   
 $\langle B \rangle$  *list-set-rel*  
 $\langle proof \rangle$

**definition** *gen-equals where*  
 $gen-equals\ ball\ lu\ eq\ f\ g \equiv$   
 $ball\ f\ (\lambda (k, v). rel-option\ eq\ (lu\ k\ g)\ (Some\ v)) \wedge$   
 $ball\ g\ (\lambda (k, v). rel-option\ eq\ (lu\ k\ f)\ (Some\ v))$

**lemma** *gen-equals*[*autoref-rules*]:  
**assumes** *PRIO-TAG-GEN-ALGO*  
**assumes** *BALL: GEN-OP ball op-map-ball* ( $\langle Rk, Rv \rangle\ Rm \rightarrow (Rk \times_r Rv \rightarrow$   
 $bool-rel) \rightarrow bool-rel$ )  
**assumes** *LU: GEN-OP lu op-map-lookup* ( $Rk \rightarrow \langle Rk, Rv \rangle\ Rm \rightarrow \langle Rv \rangle$   
 $option-rel$ )  
**assumes** *EQ: GEN-OP eq HOL.eq* ( $Rv \rightarrow Rv \rightarrow bool-rel$ )  
**shows** (*gen-equals ball lu eq, HOL.eq*)  $\in \langle Rk, Rv \rangle\ Rm \rightarrow \langle Rk, Rv \rangle\ Rm \rightarrow$   
 $bool-rel$   
 $\langle proof \rangle$

**definition** *op-set-enumerate* :: 'a set  $\Rightarrow$  ('a  $\rightarrow$  nat) nres **where**  
*op-set-enumerate* S  $\equiv$  SPEC ( $\lambda$  f. dom f = S  $\wedge$  inj-on f S)

**lemma** [autoref-itype]: *op-set-enumerate* ::<sub>i</sub>  $\langle A \rangle_i$  i-set  $\rightarrow_i$   $\langle \langle A, i\text{-nat} \rangle_i$  i-map  $\rangle_i$   
i-nres  $\langle$ proof $\rangle$

**lemma** [autoref-hom]: CONSTRAINT *op-set-enumerate* ( $\langle A \rangle$  Rs  $\rightarrow$   $\langle \langle A, \text{nat-rel} \rangle$   
Rm  $\rangle$  nres-rel)  $\langle$ proof $\rangle$

**definition** *gen-enumerate* **where**

*gen-enumerate* tol upd emp S  $\equiv$  snd (fold ( $\lambda$  x (k, m). (Suc k, upd x k m))  
(tol S) (0, 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  $\rightarrow$  nat-rel  $\rightarrow$   $\langle A, \text{nat-rel} \rangle$   
Rm  $\rightarrow$   $\langle A, \text{nat-rel} \rangle$  Rm)

**shows** ( $\lambda$  S. RETURN (*gen-enumerate* tol upd emp S), *op-set-enumerate*)  $\in$   
 $\langle A \rangle$  Rs  $\rightarrow$   $\langle \langle A, \text{nat-rel} \rangle$  Rm  $\rangle$  nres-rel

$\langle$ proof $\rangle$

**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)))

$\langle$ proof $\rangle$

**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 Rs tol)

**assumes** empty: GEN-OP emp op-map-empty ( $\langle A, B \rangle$  Rm)

**assumes** update: GEN-OP upd op-map-update (A  $\rightarrow$  B  $\rightarrow$   $\langle A, B \rangle$  Rm  $\rightarrow$   $\langle A,$   
B  $\rangle$  Rm)

**shows** ( $\lambda$  f X. *gen-build* tol upd emp f X,  $\lambda$  f X. (Some  $\circ$  f) |' X)  $\in$

(A  $\rightarrow$  B)  $\rightarrow$   $\langle A \rangle$  Rs  $\rightarrow$   $\langle A, B \rangle$  Rm

$\langle$ proof $\rangle$

**definition** *to-list* it s  $\equiv$  it s top Cons Nil

**lemma** *map2set-to-list*:

**assumes** GEN-ALGO-tag (is-map-to-list Rk unit-rel R it)

**shows** is-set-to-list Rk (map2set-rel R) (to-list (map-iterator-dom  $\circ$  (foldli  $\circ$   
it)))

$\langle$ proof $\rangle$

**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*)  $\in \langle A \rangle Rs \rightarrow \langle A \rangle list\text{-set-rel}$   
 $\langle proof \rangle$

**lemma** *param-foldli*:

**assumes** (*xs, ys*)  $\in \langle Ra \rangle list\text{-rel}$   
**assumes** (*c, d*)  $\in Rs \rightarrow bool\text{-rel}$   
**assumes**  $\bigwedge x y. (x, y) \in Ra \implies x \in set\ xs \implies y \in set\ ys \implies (f\ x, g\ y) \in$   
 $Rs \rightarrow Rs$

**assumes** (*a, b*)  $\in Rs$   
**shows** (*foldli xs c f a, foldli ys d g b*)  $\in Rs$

$\langle proof \rangle$

**lemma** *det-fold-sorted-set*:

**assumes** 1: *det-fold-set* *ordR c' f'  $\sigma'$  result*  
**assumes** 2: *is-set-to-sorted-list* *ordR Rk Rs tsl*  
**assumes** *SREF*[*param*]: (*s, s'*)  $\in \langle Rk \rangle Rs$   
**assumes** [*param*]: (*c, c'*)  $\in R\sigma \rightarrow Id$   
**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*]: ( *$\sigma, \sigma'$* )  $\in R\sigma$   
**shows** (*foldli (tsl s) c f  $\sigma$ , result s'*)  $\in R\sigma$

$\langle proof \rangle$

**lemma** *det-fold-set*:

**assumes** *det-fold-set* ( $\lambda - . True$ ) *c' f'  $\sigma'$  result*  
**assumes** *is-set-to-list* *Rk Rs tsl*  
**assumes** (*s, s'*)  $\in \langle Rk \rangle Rs$   
**assumes** (*c, c'*)  $\in R\sigma \rightarrow Id$   
**assumes**  $\bigwedge x y. (x, y) \in Rk \implies y \in s' \implies (f\ x, f'\ y) \in R\sigma \rightarrow R\sigma$   
**assumes** ( *$\sigma, \sigma'$* )  $\in R\sigma$   
**shows** (*foldli (tsl s) c f  $\sigma$ , result s'*)  $\in R\sigma$

$\langle proof \rangle$

**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'  $\rightarrow \langle Rk' \rangle Rs2 \rightarrow \langle Rk' \rangle Rs2$* )  
**assumes** *EMPTY*: *GEN-OP* *empty2*  $\{ \}$  ( $\langle Rk' \rangle Rs2$ )  
**assumes**  $\bigwedge xi\ x. (xi, x) \in Rk \implies x \in s \implies (fi\ xi, f\ \$\ x) \in Rk'$   
**assumes** (*l, s*)  $\in \langle Rk \rangle Rs1$   
**shows** (*gen-image* ( $\lambda x. foldli (it1\ x) empty2 ins2 fi\ l,$   
 $(OP\ image\ :: (\langle Rk \rangle Rs1) \rightarrow (\langle Rk' \rangle Rs2))\ \$\ f\ \$\ s) \in (\langle Rk' \rangle Rs2)$ )  
 $\langle proof \rangle$

**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 :: ('label1  $\times$  'label2) set  $\Rightarrow$  ('state1  $\times$  'state2) set  $\Rightarrow$ 
  (('label1, 'state1) drai  $\times$  ('label2, 'state2) drai) set where
[to-relAPP]: drai-rel L S  $\equiv$  {(A1, A2).
  (alphabeti A1, alphabeti A2)  $\in$   $\langle$ L $\rangle$  list-rel  $\wedge$ 
  (initiali A1, initiali A2)  $\in$  S  $\wedge$ 
  (transitioni A1, transitioni A2)  $\in$  L  $\rightarrow$  S  $\rightarrow$  S  $\wedge$ 
  (conditioni A1, conditioni A2)  $\in$   $\langle$ 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
   $\langle$ proof $\rangle$ 

```

```

definition drai-dra-rel :: ('label1  $\times$  'label2) set  $\Rightarrow$  ('state1  $\times$  'state2) set  $\Rightarrow$ 
  (('label1, 'state1) drai  $\times$  ('label2, 'state2) dra) set where
[to-relAPP]: drai-dra-rel L S  $\equiv$  {(A1, A2).
  (alphabeti A1, alphabet A2)  $\in$   $\langle$ L $\rangle$  list-set-rel  $\wedge$ 
  (initiali A1, initial A2)  $\in$  S  $\wedge$ 
  (transitioni A1, transition A2)  $\in$  L  $\rightarrow$  S  $\rightarrow$  S  $\wedge$ 
  (conditioni A1, condition A2)  $\in$   $\langle$ 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)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel  $\rightarrow$   $\langle$ L $\rangle$  list-set-rel
  (initiali, initial)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel  $\rightarrow$  S
  (transitioni, transition)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel  $\rightarrow$  L  $\rightarrow$  S  $\rightarrow$  S
  (conditioni, condition)  $\in$   $\langle$ L, S $\rangle$  drai-dra-rel  $\rightarrow$   $\langle$ rabin-rel S $\rangle$  list-rel
   $\langle$ proof $\rangle$ 

```

```

definition drai-dra :: ('label, 'state) drai  $\Rightarrow$  ('label, 'state) dra where

```

$drai-dra\ A \equiv dra\ (set\ (alphabeti\ A))\ (initiali\ A)\ (transizioni\ A)\ (conditioni\ A)$   
**definition**  $drai-invar :: ('label, 'state)\ drai \Rightarrow bool$  **where**  
 $drai-invar\ A \equiv distinct\ (alphabeti\ A)$

**lemma**  $drai-dra-id-param[param]: (drai-dra, id) \in \langle L, S \rangle\ drai-dra-rel \rightarrow \langle L, S \rangle$   
 $dra-rel$   
 $\langle proof \rangle$

**lemma**  $drai-dra-br: \langle Id, Id \rangle\ drai-dra-rel = br\ drai-dra\ drai-invar$   
 $\langle proof \rangle$

**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**  $dra-G-graph[simp]: graph\ (dra-G\ A)\ \langle proof \rangle$

**lemma**  $dra-G-reachable-nodes: op-reachable\ (dra-G\ A) = nodes\ A$   
 $\langle proof \rangle$

**context**

**begin**

**interpretation**  $autoref-syn\ \langle proof \rangle$

**lemma**  $dra-G-ahs: dra-G\ A = (\mid\ g-V = UNIV, g-E = E-of-succ\ (\lambda\ p.\ CAST$   
 $((\lambda\ a.\ transition\ A\ a\ p :: S)\ 'alphabet\ A :: \langle S \rangle\ ahs-rel\ bhc), g-V0 = \{initial\ A\}\ \mid)$   
 $\langle proof \rangle$

**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**  $(?f :: ?'a, RETURN\ (dra-G\ A)) \in ?A$

$\langle proof \rangle$

**concrete-definition**  $drai-Gi$  **uses**  $drai-Gi$

```

lemma drai-Gi-refine[autoref-rules]:
  fixes S :: ('statei × '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 → S → bool-rel)
  shows (DRA-Nodes.drai-Gi seq bhc hms, dra-G) ∈ ⟨L, S⟩ drai-dra-rel →
  ⟨unit-rel, S⟩ g-impl-rel-ext
  ⟨proof⟩

```

```

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 ⟨proof⟩

```

**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⟩

```

**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 ∧

```

$(\text{initiale } A_1, \text{initiale } A_2) \in S \wedge$   
 $(\text{transitione } A_1, \text{transitione } A_2) \in \langle S \times_r L \times_r S \rangle \text{ set-rel} \wedge$   
 $(\text{conditione } A_1, \text{conditione } A_2) \in \langle \langle S \rangle \text{ set-rel} \times_r \langle S \rangle \text{ set-rel} \rangle \text{ list-rel}$

**lemma** *drae-param*[*param, autoref-rules*]:

$(\text{drae}, \text{drae}) \in \langle L \rangle \text{ set-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel} \rightarrow$   
 $\langle \langle S \rangle \text{ set-rel} \times_r \langle S \rangle \text{ set-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ drae-rel}$   
 $(\text{alphabetete}, \text{alphabetete}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle L \rangle \text{ set-rel}$   
 $(\text{initiale}, \text{initiale}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow S$   
 $(\text{transitione}, \text{transitione}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel}$   
 $(\text{conditione}, \text{conditione}) \in \langle L, S \rangle \text{ drae-rel} \rightarrow \langle \langle S \rangle \text{ set-rel} \times_r \langle S \rangle \text{ set-rel} \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *drae-rel-id*[*simp*]:  $\langle \text{Id}, \text{Id} \rangle \text{ drae-rel} = \text{Id} \langle \text{proof} \rangle$

**lemma** *drae-rel-comp*[*simp*]:  $\langle L_1 \ O \ L_2, S_1 \ O \ S_2 \rangle \text{ drae-rel} = \langle L_1, S_1 \rangle \text{ drae-rel} \ O$   
 $\langle L_2, S_2 \rangle \text{ drae-rel}$   
 $\langle \text{proof} \rangle$

**consts** *i-drae-scheme* :: *interface*  $\Rightarrow$  *interface*  $\Rightarrow$  *interface*

**context**

**begin**

**interpretation** *autoref-syn*  $\langle \text{proof} \rangle$

**lemma** *drae-scheme-itype*[*autoref-itype*]:

$\text{drae} ::_i \langle L \rangle_i \text{ i-set} \rightarrow_i S \rightarrow_i \langle \langle S, \langle L, S \rangle_i \text{ i-prod} \rangle_i \text{ i-prod} \rangle_i \text{ i-set} \rightarrow_i$   
 $\langle \langle \langle S \rangle_i \text{ i-set}, \langle S \rangle_i \text{ i-set} \rangle_i \text{ i-prod} \rangle_i \text{ i-list} \rightarrow_i \langle L, S \rangle_i \text{ i-drae-scheme}$   
 $\text{alphabetete} ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle L \rangle_i \text{ i-set}$   
 $\text{initiale} ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i S$   
 $\text{transitione} ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle \langle S, \langle L, S \rangle_i \text{ i-prod} \rangle_i \text{ i-prod} \rangle_i \text{ i-set}$   
 $\text{conditione} ::_i \langle L, S \rangle_i \text{ i-drae-scheme} \rightarrow_i \langle \langle \langle S \rangle_i \text{ i-set}, \langle S \rangle_i \text{ i-set} \rangle_i \text{ i-prod} \rangle_i \text{ i-list}$   
 $\langle \text{proof} \rangle$

**end**

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

$(\text{alphabetetei}: 'label \ \text{list})$   
 $(\text{initialei}: 'state)$   
 $(\text{transitionei}: ('state \times 'label \times 'state) \ \text{list})$   
 $(\text{conditionei}: ('state \ \text{list} \times 'state \ \text{list}) \ \text{list})$

**definition** *draei-rel* **where**

$[\text{to-relAPP}]: \text{draei-rel } L \ S \equiv \{(A_1, A_2).$   
 $(\text{alphabetetei } A_1, \text{alphabetetei } A_2) \in \langle L \rangle \text{ list-rel} \wedge$   
 $(\text{initialei } A_1, \text{initialei } A_2) \in S \wedge$   
 $(\text{transitionei } A_1, \text{transitionei } A_2) \in \langle S \times_r L \times_r S \rangle \text{ list-rel} \wedge$   
 $(\text{conditionei } A_1, \text{conditionei } A_2) \in \langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle \text{ list-rel}\}$

**lemma** *draei-param*[*param*, *autoref-rules*]:

$(draei, draei) \in \langle L \rangle \text{ list-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-rel} \rightarrow$   
 $\langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ draei-rel}$   
 $(alphabetei, alphabetei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle L \rangle \text{ list-rel}$   
 $(initialei, initialei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow S$   
 $(transitionei, transitionei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-rel}$   
 $(conditionei, conditionei) \in \langle L, S \rangle \text{ draei-rel} \rightarrow \langle \langle S \rangle \text{ list-rel} \times_r \langle S \rangle \text{ list-rel} \rangle$   
 $\text{list-rel}$   
 $\langle \text{proof} \rangle$

**definition** *draei-drae-rel* **where**

$[to\text{-rel}APP]: \text{draei-drae-rel } L \ S \equiv \{(A_1, A_2).$   
 $(alphabetei \ A_1, \ alphabete \ A_2) \in \langle L \rangle \text{ list-set-rel} \wedge$   
 $(initialei \ A_1, \ initiale \ A_2) \in S \wedge$   
 $(transitionei \ A_1, \ transitione \ A_2) \in \langle S \times_r L \times_r S \rangle \text{ list-set-rel} \wedge$   
 $(conditionei \ A_1, \ conditione \ A_2) \in \langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle \text{ list-set-rel} \rangle \text{ list-rel}\}$

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of draei-drae-rel i-drae-scheme*]

**lemma** *draei-drae-param*[*param*, *autoref-rules*]:

$(draei, drae) \in \langle L \rangle \text{ list-set-rel} \rightarrow S \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-set-rel} \rightarrow$   
 $\langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle \text{ list-set-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ draei-drae-rel}$   
 $(alphabetei, alphabete) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle L \rangle \text{ list-set-rel}$   
 $(initialei, initiale) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow S$   
 $(transitionei, transitione) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ list-set-rel}$   
 $(conditionei, conditione) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow \langle \langle S \rangle \text{ list-set-rel} \times_r \langle S \rangle$   
 $\text{list-set-rel} \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**definition** *draei-drae* **where**

$\text{draei-drae } A \equiv \text{drae } (\text{set } (\text{alphabetei } A)) \ (\text{initialei } A)$   
 $(\text{set } (\text{transitionei } A)) \ (\text{map } (\text{map-prod } \text{set } \text{set}) \ (\text{conditionei } A))$

**lemma** *draei-drae-id-param*[*param*]:  $(draei-drae, id) \in \langle L, S \rangle \text{ draei-drae-rel} \rightarrow$   
 $\langle L, S \rangle \text{ drae-rel}$   
 $\langle \text{proof} \rangle$

**abbreviation** *transitions*  $L \ S \ s \equiv \bigcup a \in L. \bigcup p \in S. \{p\} \times \{a\} \times \{s \ a \ p\}$

**abbreviation** *succs*  $T \ a \ p \equiv \text{the-elem } ((T \ \{\{p\}\} \ \{\{a\}\})$

**definition** *wft* :: '*label set*  $\Rightarrow$  '*state set*  $\Rightarrow$  ('*state*  $\times$  '*label*  $\times$  '*state*) *set*  $\Rightarrow$  *bool*  
**where**

$wft \ L \ S \ T \equiv \forall a \in L. \forall p \in S. \text{is-singleton } ((T \ \{\{p\}\} \ \{\{a\}\})$

**lemma** *wft-param*[*param*]:

**assumes** *bijective*  $S$  *bijective*  $L$

**shows**  $(wft, wft) \in \langle L \rangle \text{ set-rel} \rightarrow \langle S \rangle \text{ set-rel} \rightarrow \langle S \times_r L \times_r S \rangle \text{ set-rel} \rightarrow \text{bool-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *wft-transitions*:  $wft\ L\ S\ (transitions\ L\ S\ s)\ \langle proof \rangle$

**definition** *dra-drae* **where**  $dra-drae\ A \equiv drae\ (alphabet\ A)\ (initial\ A)$   
 $(transitions\ (alphabet\ A)\ (nodes\ A)\ (transition\ A))$   
 $(map\ (\lambda\ (P,\ Q).\ (Set.filter\ P\ (nodes\ A),\ Set.filter\ Q\ (nodes\ A)))\ (condition\ A))$

**definition** *drae-dra* **where**  $drae-dra\ A \equiv dra\ (alphabet\ A)\ (initiale\ A)$   
 $(succs\ (transitione\ A))\ (map\ (\lambda\ (I,\ F).\ (\lambda\ p.\ p \in I,\ \lambda\ p.\ p \in F))\ (conditione\ A))$

**lemma** *set-rel-Domain-Range*[*intro!*, *simp*]:  $(Domain\ A,\ Range\ A) \in \langle A \rangle\ set-rel\ \langle proof \rangle$

**lemma** *dra-drae-param*[*param*]:  $(dra-drae,\ dra-drae) \in \langle L,\ S \rangle\ dra-rel \rightarrow \langle L,\ S \rangle\ drae-rel\ \langle proof \rangle$

**lemma** *drae-dra-param*[*param*]:  
**assumes** *bijjective L* *bijjective S*  
**assumes** *wft (Range L) (Range S) (transitione B)*  
**assumes** [*param*]:  $(A,\ B) \in \langle L,\ S \rangle\ drae-rel$   
**shows**  $(drae-dra\ A,\ drae-dra\ B) \in \langle L,\ S \rangle\ dra-rel\ \langle proof \rangle$

**lemma** *succs-transitions-param*[*param*]:  
 $(succs \circ transitions\ L\ S,\ id) \in (Id-on\ L \rightarrow Id-on\ S \rightarrow Id-on\ S) \rightarrow (Id-on\ L \rightarrow Id-on\ S \rightarrow Id-on\ S)\ \langle proof \rangle$

**lemma** *drae-dra-dra-drae-param*[*param*]:  
 $((drae-dra \circ dra-drae)\ A,\ id\ A) \in \langle Id-on\ (alphabet\ A),\ Id-on\ (nodes\ A) \rangle\ dra-rel\ \langle proof \rangle$

**definition** *draei-dra-rel* **where**  
 $[to-relAPP]: draei-dra-rel\ L\ S \equiv \{(Ae,\ A).\ (drae-dra\ (draei-drae\ Ae),\ A) \in \langle L,\ S \rangle\ dra-rel\}$

**lemma** *draei-dra-id*[*param*]:  $(drae-dra \circ draei-drae,\ id) \in \langle L,\ S \rangle\ draei-dra-rel \rightarrow \langle L,\ S \rangle\ dra-rel\ \langle proof \rangle$

**end**

## 32 Explore and Enumerate Nodes of Deterministic Rabin Automata

```
theory DRA-Translate
imports DRA-Explicit
begin
```

### 32.1 Syntax

**no-syntax** *-do-let* :: [pttrn, 'a] ⇒ do-bind (⟨⟨indent=2 notation=⟨infix do let⟩let - =/ -⟩ [1000, 13] 13⟩  
**syntax** *-do-let* :: [pttrn, 'a] ⇒ do-bind (⟨⟨indent=2 notation=⟨infix do let⟩let - =/ -⟩ 13⟩)

## 33 Image on Explicit Automata

**definition** *drae-image* **where** *drae-image*  $f$   $A$  ≡ *drae* (*alphabet*  $A$ ) ( $f$  (*initial*  $A$ ))  
 ((λ ( $p$ ,  $a$ ,  $q$ ). ( $f$   $p$ ,  $a$ ,  $f$   $q$ )) ' *transition*  $A$ ) (*map* (*map-prod* (*image*  $f$ ) (*image*  $f$ )) (*condition*  $A$ ))

**lemma** *drae-image-param*[*param*]: (*drae-image*, *drae-image*) ∈ ( $S$  →  $T$ ) → ⟨ $L$ ,  $S$ ⟩ *drae-rel* → ⟨ $L$ ,  $T$ ⟩ *drae-rel*  
 ⟨*proof*⟩

**lemma** *drae-image-id*[*simp*]: *drae-image* *id* = *id* ⟨*proof*⟩

**lemma** *drae-image-dra-drae*: *drae-image*  $f$  (*dra-drae*  $A$ ) = *drae* (*alphabet*  $A$ ) ( $f$  (*initial*  $A$ ))  
 (∪  $p$  ∈ *nodes*  $A$ . ∪  $a$  ∈ *alphabet*  $A$ .  $f$  ' { $p$ } × { $a$ } ×  $f$  ' {*transition*  $A$   $a$   $p$ })  
 (*map* (λ ( $P$ ,  $Q$ ). ( $f$  ' { $p$  ∈ *nodes*  $A$ .  $P$   $p$ },  $f$  ' { $p$  ∈ *nodes*  $A$ .  $Q$   $p$ })) (*condition*  $A$ ))  
 ⟨*proof*⟩

## 34 Exploration and Translation

**definition** *trans-spec* **where**

*trans-spec*  $A$   $f$  ≡ ∪  $p$  ∈ *nodes*  $A$ . ∪  $a$  ∈ *alphabet*  $A$ .  $f$  ' { $p$ } × { $a$ } ×  $f$  ' {*transition*  $A$   $a$   $p$ }

**definition** *trans-algo* **where**

*trans-algo*  $N$   $L$   $S$   $f$  ≡  
 FOREACH  $N$  (λ  $p$   $T$ . do {  
   ASSERT ( $p$  ∈  $N$ );  
   FOREACH  $L$  (λ  $a$   $T$ . do {  
     ASSERT ( $a$  ∈  $L$ );  
     let  $q$  =  $S$   $a$   $p$ ;  
     ASSERT (( $f$   $p$ ,  $a$ ,  $f$   $q$ ) ∉  $T$ );  
     RETURN (*insert* ( $f$   $p$ ,  $a$ ,  $f$   $q$ )  $T$ ) }  
   )  $T$  }  
 ) {}

**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*  
 ⟨*proof*⟩

**definition** *to-draei* :: ('state, 'label) dra ⇒ ('state, 'label) dra  
 where *to-draei* ≡ id

**schematic-goal** *to-draei-impl*:

```

fixes S :: ('statei × '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⟩ drai-dra-rel
shows (?f :: ?'a, do {
  let N = nodes A;
  f ← 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)))
  }) ∈ ?R
  <proof>

```

**concrete-definition** *to-draei-impl uses to-draei-impl*

**lemma** *to-draei-impl-refine''*:

```

fixes S :: ('statei × '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) ∈ S → S → bool-rel
assumes (Ai, A) ∈ ⟨L, S⟩ drai-dra-rel
shows (RETURN (to-draei-impl seq bhc hms Ai), do {
  f ← op-set-enumerate (nodes A);
  RETURN (drae-image (the ∘ f) (dra-drae A))
  }) ∈ ⟨⟨L, nat-rel⟩ draei-drae-rel⟩ nres-rel
  <proof>

```

**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**  $f' \equiv \text{SOME } f'$ .  
 $(\text{to-draei-impl seq bhc hms } Ai, \text{drae-image } (\text{the } \circ f') (\text{dra-drae } A)) \in \langle \text{Id}, \text{nat-rel} \rangle \text{draei-drae-rel} \wedge$   
 $\text{dom } f' = \text{nodes } A \wedge \text{inj-on } f' (\text{nodes } A)$

**lemma** 1:  $\exists f'$ .  $(\text{to-draei-impl seq bhc hms } Ai, \text{drae-image } (\text{the } \circ f') (\text{dra-drae } A)) \in$   
 $\langle \text{Id}, \text{nat-rel} \rangle \text{draei-drae-rel} \wedge \text{dom } f' = \text{nodes } A \wedge \text{inj-on } f' (\text{nodes } A)$   
 $\langle \text{proof} \rangle$

**lemma**  $f'$ -refine:  $(\text{to-draei-impl seq bhc hms } Ai, \text{drae-image } (\text{the } \circ f') (\text{dra-drae } A)) \in$   
 $\langle \text{Id}, \text{nat-rel} \rangle \text{draei-drae-rel} \langle \text{proof} \rangle$

**lemma**  $f'$ -dom:  $\text{dom } f' = \text{nodes } A \langle \text{proof} \rangle$

**lemma**  $f'$ -inj:  $\text{inj-on } f' (\text{nodes } A) \langle \text{proof} \rangle$

**definition**  $f$  **where**  $f \equiv \text{the } \circ f'$

**definition**  $g$  **where**  $g = \text{inv-into } (\text{nodes } A) f$

**lemma**  $\text{inj-}f[\text{intro!}, \text{simp}]$ :  $\text{inj-on } f (\text{nodes } A)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{inj-}g[\text{intro!}, \text{simp}]$ :  $\text{inj-on } g (f \text{ ' nodes } A)$   
 $\langle \text{proof} \rangle$

**definition**  $\text{rel}$  **where**  $\text{rel} \equiv \{(f p, p) \mid p. p \in \text{nodes } A\}$

**lemma**  $\text{rel-alt-def}$ :  $\text{rel} = (\text{br } f (\lambda p. p \in \text{nodes } A))^{-1}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rel-inv-def}$ :  $\text{rel} = \text{br } g (\lambda k. k \in f \text{ ' nodes } A)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{rel-domain}[\text{simp}]$ :  $\text{Domain } \text{rel} = f \text{ ' nodes } A \langle \text{proof} \rangle$

**lemma**  $\text{rel-range}[\text{simp}]$ :  $\text{Range } \text{rel} = \text{nodes } A \langle \text{proof} \rangle$

**lemma**  $[\text{intro!}, \text{simp}]$ :  $\text{bijective } \text{rel} \langle \text{proof} \rangle$

**lemma**  $[\text{simp}]$ :  $\text{Id-on } (f \text{ ' nodes } A) \circ \text{rel} = \text{rel} \langle \text{proof} \rangle$

**lemma**  $[\text{simp}]$ :  $\text{rel} \circ \text{Id-on } (\text{nodes } A) = \text{rel} \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(f, f) \in \text{Id-on } (\text{Range } \text{rel}) \rightarrow \text{Id-on } (\text{Domain } \text{rel}) \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(g, g) \in \text{Id-on } (\text{Domain } \text{rel}) \rightarrow \text{Id-on } (\text{Range } \text{rel}) \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(\text{id}, f) \in \text{rel} \rightarrow \text{Id-on } (\text{Domain } \text{rel}) \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(f, \text{id}) \in \text{Id-on } (\text{Range } \text{rel}) \rightarrow \text{rel} \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(\text{id}, g) \in \text{Id-on } (\text{Domain } \text{rel}) \rightarrow \text{rel} \langle \text{proof} \rangle$

**lemma**  $[\text{param}]$ :  $(g, \text{id}) \in \text{rel} \rightarrow \text{Id-on } (\text{Range } \text{rel}) \langle \text{proof} \rangle$

**lemma**  $\text{to-draei-impl-refine}'$ :

$(\text{to-draei-impl seq bhc hms } Ai, \text{to-draei } A) \in \langle \text{Id-on } (\text{alphabet } A), \text{rel} \rangle \text{draei-dra-rel}$   
 $\langle \text{proof} \rangle$

**end**

**context**

**begin**

**interpretation** *autoref-syn*  $\langle \text{proof} \rangle$

**lemma** *to-draei-impl-refine*[*autoref-rules*]:

**fixes**  $S :: ('statei \times 'state) \text{ set}$

**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* ( $S \rightarrow S \rightarrow \text{bool-rel}$ )

**assumes**  $(Ai, A) \in \langle Id, S \rangle \text{ drai-dra-rel}$

**shows** (*to-draei-impl seq bhc hms Ai*,

$(OP \text{ to-draei} :: \langle Id, S \rangle \text{ drai-dra-rel} \rightarrow$

$\langle Id\text{-on } (\text{alphabet } A), \text{ rel } Ai \ A \ \text{seq } bhc \ hms \rangle \text{ draei-dra-rel}$ )  $\$ A \in$

$\langle Id\text{-on } (\text{alphabet } A), \text{ rel } Ai \ A \ \text{seq } bhc \ hms \rangle \text{ draei-dra-rel}$

$\langle \text{proof} \rangle$

**end**

**end**

## 35 Nondeterministic Büchi Automata

**theory** *NBA*

**imports** *../Nondeterministic*

**begin**

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

(*alphabet: 'label set*)

(*initial: 'state set*)

(*transition: 'label  $\Rightarrow$  'state  $\Rightarrow$  '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*

$\langle \text{proof} \rangle$

**global-interpretation** *nba: automaton-run nba alphabet initial transition accept-*

*ing  $\lambda P w r p. \text{infs } P (p \ \#\# \ r)$*

**defines** *language = nba.language*

$\langle \text{proof} \rangle$

**abbreviation** *target where target  $\equiv$  nba.target*

**abbreviation** *states where states  $\equiv$  nba.states*

**abbreviation** *trace where trace  $\equiv$  nba.trace*

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

**instantiation** *nba :: (type, type) order*

**begin**

**definition** *less-eq-nba* :: ('a, 'b) nba  $\Rightarrow$  ('a, 'b) nba  $\Rightarrow$  bool **where**  
 $A \leq B \equiv$  alphabet  $A \leq$  alphabet  $B \wedge$  initial  $A \leq$  initial  $B \wedge$   
transition  $A \leq$  transition  $B \wedge$  accepting  $A \leq$  accepting  $B$   
**definition** *less-nba* :: ('a, 'b) nba  $\Rightarrow$  ('a, 'b) nba  $\Rightarrow$  bool **where**  
*less-nba*  $A B \equiv A \leq B \wedge A \neq B$

**instance** <proof>

**end**

**lemma** *nodes-mono*: mono nodes  
<proof>

**lemma** *language-mono*: mono language  
<proof>

**lemma** *simulation-language*:  
**assumes** alphabet  $A \subseteq$  alphabet  $B$   
**assumes**  $\bigwedge p. p \in$  initial  $A \implies \exists q \in$  initial  $B. (p, q) \in R$   
**assumes**  $\bigwedge a p p' q. p' \in$  transition  $A \wedge a p \implies (p, q) \in R \implies \exists q' \in$  transition  
 $B \wedge a q. (p', q') \in R$   
**assumes**  $\bigwedge p q. (p, q) \in R \implies$  accepting  $A p \implies$  accepting  $B q$   
**shows** language  $A \subseteq$  language  $B$   
<proof>

**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  
<proof>

**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  
<proof>

**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*  
*<proof>*

**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*<sub>1</sub>  $\circ$  *fst*, *c*<sub>2</sub>  $\circ$  *snd*]*  
**defines** *intersect'* = *intersection.product*  
*<proof>*

**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*  $\lambda P w r p. \text{infs } P (p \text{ ## } r)$   
*case-sum*  
**defines** *union* = *union.sum*  
 <proof>

**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*  $\lambda P w r p. \text{infs } P (p \text{ ## } r)$   
*ngba ngba.alphabet ngba.initial ngba.transition ngba.accepting*  $\lambda P w r p. \text{gen infs } P (p \text{ ## } r)$   
 $\lambda cs. \text{map } (\lambda k ps. (cs ! k) (ps ! k)) [0 ..< \text{length } cs]$   
**defines** *intersect-list'* = *intersection-list.product*  
 <proof>

**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*  $\lambda P w r p. \text{infs } P (p \text{ ## } r)$   
*nba nba.alphabet nba.initial nba.transition nba.accepting*  $\lambda P w r p. \text{infs } P (p \text{ ## } r)$   
 $\lambda cs (k, p). (cs ! k) p$   
**defines** *union-list* = *union-list.sum*  
 <proof>

**lemmas** *union-list-language* = *union-list.sum-language*  
**lemmas** *union-list-nodes-finite* = *union-list.sum-nodes-finite*

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

**lemma** *intersect-language[simp]*: *NBA.language (intersect A B)* = *NBA.language A*  $\cap$  *NBA.language B*  
 <proof>

**lemma** *intersect-nodes-finite[intro]*:  
**assumes** *finite (NBA.nodes A) finite (NBA.nodes B)*  
**shows** *finite (NBA.nodes (intersect A B))*  
 <proof>

**abbreviation** *intersect-list where* *intersect-list AA*  $\equiv \text{degeneralize } (\text{intersect-list}' AA)$

**lemma** *intersect-list-language[simp]*: *NBA.language (intersect-list AA)* =  $\bigcap$  (*NBA.language*

```

‘ set AA)
  ⟨proof⟩
lemma intersect-list-nodes-finite[intro]:
  assumes list-all (finite ◦ NBA.nodes) AA
  shows finite (NBA.nodes (intersect-list AA))
  ⟨proof⟩

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 ⇒ 'state graph-rec where
  nba-g A ≡ (| g-V = UNIV, g-E = E-of-succ (successors A), g-V0 = initial A |)

```

```

lemma nba-g-graph[simp]: graph (nba-g A) ⟨proof⟩

```

```

lemma nba-g-V0: g-V0 (nba-g A) = initial A ⟨proof⟩

```

```

lemma nba-g-E-rtrancl: (g-E (nba-g A))* = {(p, q). q ∈ reachable A p}
  ⟨proof⟩

```

```

lemma nba-g-rtrancl-path: (g-E (nba-g A))* = {(p, target r p) | r p. NBA.path A
  r p}
  ⟨proof⟩

```

```

lemma nba-g-trancl-path: (g-E (nba-g A))^+ = {(p, target r p) | r p. NBA.path A
  r p ∧ r ≠ []}
  ⟨proof⟩

```

```

lemma nba-g-ipath-run:
  assumes ipath (g-E (nba-g A)) r
  obtains w
  where run A (w ||| smap (r ◦ Suc) nats) (r 0)
  ⟨proof⟩

```

```

lemma nba-g-run-ipath:
  assumes run A (w ||| r) p
  shows ipath (g-E (nba-g A)) (snth (p ## r))
  ⟨proof⟩

```

## 38.2 Indexed Generalized Büchi Graphs

**definition**  $nba\text{-igbg} :: ('label, 'state) nba \Rightarrow 'state\ igb\text{-graph}\text{-rec}$  **where**  
 $nba\text{-igbg}\ A \equiv graph\text{-rec}\text{-extend}\ (nba\text{-g}\ A)$   
 $(\ igb\text{-num}\text{-acc} = 1, igb\text{-acc} = \lambda p.\ \text{if}\ \text{accepting}\ A\ p\ \text{then}\ \{0\}\ \text{else}\ \{\}\ )$

**lemma**  $acc\text{-run}\text{-language}$ :  
**assumes**  $igb\text{-graph}\ (nba\text{-igbg}\ A)$   
**shows**  $Ex\ (igb\text{-graph}\text{-is}\text{-acc}\text{-run}\ (nba\text{-igbg}\ A)) \longleftrightarrow language\ A \neq \{\}$   
 $\langle proof \rangle$

**end**

## 39 Relations on Nondeterministic Büchi Automata

**theory**  $NBA\text{-Refine}$

**imports**

$NBA$

$.././\ Transition\text{-Systems}/\ Transition\text{-System}\text{-Refine}$

**begin**

**definition**  $nba\text{-rel} :: ('label_1 \times 'label_2) set \Rightarrow ('state_1 \times 'state_2) set \Rightarrow$   
 $(('label_1, 'state_1) nba \times ('label_2, 'state_2) nba) set$  **where**  
 $[to\text{-rel}APP]: nba\text{-rel}\ L\ S \equiv \{(A_1, A_2).\$   
 $(alphabet\ A_1, alphabet\ A_2) \in \langle L \rangle\ set\text{-rel} \wedge$   
 $(initial\ A_1, initial\ A_2) \in \langle S \rangle\ set\text{-rel} \wedge$   
 $(transition\ A_1, transition\ A_2) \in L \rightarrow S \rightarrow \langle S \rangle\ set\text{-rel} \wedge$   
 $(accepting\ A_1, accepting\ A_2) \in S \rightarrow bool\text{-rel}\}$

**lemma**  $nba\text{-param}[param]$ :  
 $(nba, nba) \in \langle L \rangle\ set\text{-rel} \rightarrow \langle S \rangle\ set\text{-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle\ set\text{-rel}) \rightarrow (S \rightarrow$   
 $bool\text{-rel}) \rightarrow$   
 $\langle L, S \rangle\ nba\text{-rel}$   
 $(alphabet, alphabet) \in \langle L, S \rangle\ nba\text{-rel} \rightarrow \langle L \rangle\ set\text{-rel}$   
 $(initial, initial) \in \langle L, S \rangle\ nba\text{-rel} \rightarrow \langle S \rangle\ set\text{-rel}$   
 $(transition, transition) \in \langle L, S \rangle\ nba\text{-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle\ set\text{-rel}$   
 $(accepting, accepting) \in \langle L, S \rangle\ nba\text{-rel} \rightarrow S \rightarrow bool\text{-rel}$   
 $\langle proof \rangle$

**lemma**  $nba\text{-rel}\text{-id}[simp]$ :  $\langle Id, Id \rangle\ nba\text{-rel} = Id$   $\langle proof \rangle$

**lemma**  $nba\text{-rel}\text{-comp}[trans]$ :

**assumes**  $[param]: (A, B) \in \langle L_1, S_1 \rangle\ nba\text{-rel}\ (B, C) \in \langle L_2, S_2 \rangle\ nba\text{-rel}$

**shows**  $(A, C) \in \langle L_1\ O\ L_2, S_1\ O\ S_2 \rangle\ nba\text{-rel}$

$\langle proof \rangle$

**lemma**  $nba\text{-rel}\text{-converse}[simp]$ :  $(\langle L, S \rangle\ nba\text{-rel})^{-1} = \langle L^{-1}, S^{-1} \rangle\ nba\text{-rel}$

$\langle proof \rangle$

**lemma**  $nba\text{-rel}\text{-eq}$ :  $(A, A) \in \langle Id\text{-on}\ (alphabet\ A), Id\text{-on}\ (nodes\ A) \rangle\ nba\text{-rel}$

$\langle proof \rangle$

**lemma** *enableds-param*[*param*]: (*nba.enableds*, *nba.enableds*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow$   
 $S \rightarrow \langle L \times_r S \rangle$  *set-rel*  
*<proof>*  
**lemma** *paths-param*[*param*]: (*nba.paths*, *nba.paths*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow S \rightarrow \langle \langle L$   
 $\times_r S \rangle$  *list-rel* *set-rel*  
*<proof>*  
**lemma** *runs-param*[*param*]: (*nba.runs*, *nba.runs*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow S \rightarrow \langle \langle L$   
 $\times_r S \rangle$  *stream-rel* *set-rel*  
*<proof>*  
  
**lemma** *reachable-param*[*param*]: (*reachable*, *reachable*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow S \rightarrow$   
 $\langle S \rangle$  *set-rel*  
*<proof>*  
**lemma** *nodes-param*[*param*]: (*nodes*, *nodes*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow \langle S \rangle$  *set-rel*  
*<proof>*  
  
**lemma** *language-param*[*param*]: (*language*, *language*)  $\in \langle L, S \rangle$  *nba-rel*  $\rightarrow \langle \langle L$   
 $\times_r S \rangle$  *stream-rel* *set-rel*  
*<proof>*

end

## 40 Implementation of Nondeterministic Büchi Automata

**theory** *NBA-Implement*

**imports**

*NBA-Refine*

*../Basic/Implement*

**begin**

**consts** *i-nba-scheme* :: *interface*  $\Rightarrow$  *interface*  $\Rightarrow$  *interface*

**context**

**begin**

**interpretation** *autoref-syn* *<proof>*

**lemma** *nba-scheme-itype*[*autoref-itype*]:

*nba* ::<sub>*i*</sub>  $\langle L \rangle_i$  *i-set*  $\rightarrow_i$   $\langle S \rangle_i$  *i-set*  $\rightarrow_i$  ( $L \rightarrow_i S \rightarrow_i \langle S \rangle_i$  *i-set*)  $\rightarrow_i$   $\langle S \rangle_i$  *i-set*  $\rightarrow_i$   
 $\langle L, S \rangle_i$  *i-nba-scheme*

*alphabet* ::<sub>*i*</sub>  $\langle L, S \rangle_i$  *i-nba-scheme*  $\rightarrow_i$   $\langle L \rangle_i$  *i-set*

*initial* ::<sub>*i*</sub>  $\langle L, S \rangle_i$  *i-nba-scheme*  $\rightarrow_i$   $\langle S \rangle_i$  *i-set*

*transition* ::<sub>*i*</sub>  $\langle L, S \rangle_i$  *i-nba-scheme*  $\rightarrow_i$   $L \rightarrow_i S \rightarrow_i \langle S \rangle_i$  *i-set*

*accepting* ::<sub>*i*</sub>  $\langle L, S \rangle_i$  *i-nba-scheme*  $\rightarrow_i$   $\langle S \rangle_i$  *i-set*

*<proof>*

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<sub>1</sub>  $\times$  'label<sub>2</sub>) set  $\Rightarrow$  ('state<sub>1</sub>  $\times$  'state<sub>2</sub>) set  $\Rightarrow$   
(('label<sub>1</sub>, 'state<sub>1</sub>) nbai  $\times$  ('label<sub>2</sub>, 'state<sub>2</sub>) nbai) set **where**  
[*to-relAPP*]: nbai-rel L S  $\equiv$  {(A<sub>1</sub>, A<sub>2</sub>).  
(*alphabeti* A<sub>1</sub>, *alphabeti* A<sub>2</sub>)  $\in$  <L> list-rel  $\wedge$   
(*initiali* A<sub>1</sub>, *initiali* A<sub>2</sub>)  $\in$  <S> list-rel  $\wedge$   
(*transitioni* A<sub>1</sub>, *transitioni* A<sub>2</sub>)  $\in$  L  $\rightarrow$  S  $\rightarrow$  <S> list-rel  $\wedge$   
(*acceptingi* A<sub>1</sub>, *acceptingi* A<sub>2</sub>)  $\in$  S  $\rightarrow$  bool-rel}

**lemma** nbai-param[*param*, *autoref-rules*]:  
(nbai, nbai)  $\in$  <L> list-rel  $\rightarrow$  <S> list-rel  $\rightarrow$  (L  $\rightarrow$  S  $\rightarrow$  <S> list-rel)  $\rightarrow$   
(S  $\rightarrow$  bool-rel)  $\rightarrow$  <L, S> nbai-rel  
(*alphabeti*, *alphabeti*)  $\in$  <L, S> nbai-rel  $\rightarrow$  <L> list-rel  
(*initiali*, *initiali*)  $\in$  <L, S> nbai-rel  $\rightarrow$  <S> list-rel  
(*transitioni*, *transitioni*)  $\in$  <L, S> nbai-rel  $\rightarrow$  L  $\rightarrow$  S  $\rightarrow$  <S> list-rel  
(*acceptingi*, *acceptingi*)  $\in$  <L, S> nbai-rel  $\rightarrow$  (S  $\rightarrow$  bool-rel)  
<proof>

**definition** nbai-nba-rel :: ('label<sub>1</sub>  $\times$  'label<sub>2</sub>) set  $\Rightarrow$  ('state<sub>1</sub>  $\times$  'state<sub>2</sub>) set  $\Rightarrow$   
(('label<sub>1</sub>, 'state<sub>1</sub>) nbai  $\times$  ('label<sub>2</sub>, 'state<sub>2</sub>) nba) set **where**  
[*to-relAPP*]: nbai-nba-rel L S  $\equiv$  {(A<sub>1</sub>, A<sub>2</sub>).  
(*alphabeti* A<sub>1</sub>, *alphabet* A<sub>2</sub>)  $\in$  <L> list-set-rel  $\wedge$   
(*initiali* A<sub>1</sub>, *initial* A<sub>2</sub>)  $\in$  <S> list-set-rel  $\wedge$   
(*transitioni* A<sub>1</sub>, *transition* A<sub>2</sub>)  $\in$  L  $\rightarrow$  S  $\rightarrow$  <S> list-set-rel  $\wedge$   
(*acceptingi* A<sub>1</sub>, *accepting* A<sub>2</sub>)  $\in$  S  $\rightarrow$  bool-rel}

**lemmas** [*autoref-rel-intf*] = REL-INTFI[*of nbai-nba-rel i-nba-scheme*]

**lemma** nbai-nba-param[*param*, *autoref-rules*]:  
(nbai, nba)  $\in$  <L> list-set-rel  $\rightarrow$  <S> list-set-rel  $\rightarrow$  (L  $\rightarrow$  S  $\rightarrow$  <S> list-set-rel)  $\rightarrow$   
(S  $\rightarrow$  bool-rel)  $\rightarrow$  <L, S> nbai-nba-rel  
(*alphabeti*, *alphabet*)  $\in$  <L, S> nbai-nba-rel  $\rightarrow$  <L> list-set-rel  
(*initiali*, *initial*)  $\in$  <L, S> nbai-nba-rel  $\rightarrow$  <S> list-set-rel  
(*transitioni*, *transition*)  $\in$  <L, S> nbai-nba-rel  $\rightarrow$  L  $\rightarrow$  S  $\rightarrow$  <S> list-set-rel  
(*acceptingi*, *accepting*)  $\in$  <L, S> nbai-nba-rel  $\rightarrow$  S  $\rightarrow$  bool-rel  
<proof>

**definition** nbai-nba :: ('label, 'state) nbai  $\Rightarrow$  ('label, 'state) nba **where**  
nbai-nba A  $\equiv$  nba (set (*alphabeti* A)) (set (*initiali* A)) ( $\lambda$  a p. set (*transitioni*  
A a p)) (*acceptingi* A)

**definition** nbai-invar :: ('label, 'state) nbai  $\Rightarrow$  bool **where**

$nbai\text{-}invar\ A \equiv distinct\ (alphabeti\ A) \wedge distinct\ (initiali\ A) \wedge (\forall\ a\ p.\ distinct\ (transitioni\ A\ a\ p))$

**lemma**  $nbai\text{-}nba\text{-}id\text{-}param[param]: (nbai\text{-}nba, id) \in \langle L, S \rangle\ nbai\text{-}nba\text{-}rel \rightarrow \langle L, S \rangle\ nba\text{-}rel$   
 $\langle proof \rangle$

**lemma**  $nbai\text{-}nba\text{-}br: \langle Id, Id \rangle\ nbai\text{-}nba\text{-}rel = br\ nbai\text{-}nba\ nbai\text{-}invar$   
 $\langle proof \rangle$

**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**  $(in\ igb\text{-}fr\text{-}graph)\ acc\text{-}run\text{-}lasso\text{-}prpl: Ex\ is\text{-}acc\text{-}run \implies Ex\ is\text{-}lasso\text{-}prpl$   
 $\langle proof \rangle$

**lemma**  $(in\ igb\text{-}fr\text{-}graph)\ lasso\text{-}prpl\text{-}acc\text{-}run\text{-}iff: Ex\ is\text{-}lasso\text{-}prpl \iff Ex\ is\text{-}acc\text{-}run$   
 $\langle proof \rangle$

**lemma**  $[autoref\text{-}rel\text{-}intf]: REL\text{-}INTF\ igbg\text{-}impl\text{-}rel\text{-}ext\ i\text{-}igbg\ \langle proof \rangle$

### 41.2 Operations

**definition**  $op\text{-}language\text{-}empty$  **where**  $[simp]: op\text{-}language\text{-}empty\ A \equiv language\ A = \{\}$

**lemmas**  $[autoref\text{-}op\text{-}pat] = op\text{-}language\text{-}empty\text{-}def[symmetric]$

### 41.3 Implementations

**context**

**begin**

**interpretation**  $autoref\text{-}syn\ \langle proof \rangle$

**lemma**  $nba\text{-}g\text{-}ahs: nba\text{-}g\ A = (\mid g\text{-}V = UNIV, g\text{-}E = E\text{-}of\text{-}succ\ (\lambda\ p.\ CAST\ ((\bigcup\ a \in\ alphabet\ A.\ transition\ A\ a\ p :: \langle S \rangle\ list\text{-}set\text{-}rel) :: \langle S \rangle\ ahs\text{-}rel\ bhc)),\ g\text{-}V0 = initial\ A\ \mid)$   
 $\langle proof \rangle$

**schematic-goal** *nbai-gi*:

**notes** [*autoref-ga-rules*] = *map2set-to-list*  
**fixes** *S* :: ('statei × '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*) ∈ *S* → *S* → *bool-rel*  
**assumes** [*autoref-rules*]: (*Ai, A*) ∈ ⟨*L, S*⟩ *nbai-nba-rel*  
**shows** (?*f* :: ?'a, *RETURN (nba-g A)*) ∈ ?*A*  
 ⟨*proof*⟩

**concrete-definition** *nbai-gi uses nbai-gi*

**lemma** *nbai-gi-refine[autoref-rules]*:

**fixes** *S* :: ('statei × '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 → S → bool-rel)*  
**shows** (*NBA-Algorithms.nbai-gi seq bhc hms, nba-g*) ∈  
 ⟨*L, S*⟩ *nbai-nba-rel* → ⟨*unit-rel, S*⟩ *g-impl-rel-ext*  
 ⟨*proof*⟩

**schematic-goal** *nba-nodes*:

**fixes** *S* :: ('statei × 'state) *set*  
**assumes** [*simp*]: *finite ((g-E (nba-g A))\* “ g-V0 (nba-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*⟩ *nbai-nba-rel*  
**shows** (?*f* :: ?'a, *op-reachable (nba-g A)*) ∈ ?*R* ⟨*proof*⟩

**concrete-definition** *nba-nodes uses nba-nodes*

**lemma** *nba-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*⟩ *nbai-nba-rel*  
**shows** (*NBA-Algorithms.nba-nodes seq bhc hms Ai,*  
 (*OP nodes* ∷ ⟨*L, S*⟩ *nbai-nba-rel* → ⟨*S*⟩ *ahs-rel bhc*) \$ *A*) ∈ ⟨*S*⟩ *ahs-rel bhc*  
 ⟨*proof*⟩

**lemma** *nba-igbg-ahs*: *nba-igbg A = (| g-V = UNIV, g-E = E-of-succ (λ p.*  
*CAST*

*((∪ a ∈ alphabet A. transition A a p ∷ ⟨S⟩ list-set-rel) ∷ ⟨S⟩ ahs-rel bhc)),*  
*g-V0 = initial A,*  
*igbg-num-acc = 1, igbg-acc = λ p. if accepting A p then {0} else {} |)*  
 ⟨*proof*⟩

**schematic-goal** *nbai-igbgi*:

**notes** [*autoref-ga-rules*] = *map2set-to-list*

```

fixes  $S :: ('state_i \times 'state) \text{ set}$ 
assumes [autoref-ga-rules]:  $is\text{-bounded}\text{-hashcode } S \text{ seq } bhc$ 
assumes [autoref-ga-rules]:  $is\text{-valid}\text{-def}\text{-hm}\text{-size } TYPE('state_i) \text{ hms}$ 
assumes [autoref-rules]:  $(seq, HOL.eq) \in S \rightarrow S \rightarrow \text{bool}\text{-rel}$ 
assumes [autoref-rules]:  $(Ai, A) \in \langle L, S \rangle \text{ nbai}\text{-nba}\text{-rel}$ 
shows  $(?f :: ?'a, RETURN (nba\text{-igbg } A)) \in ?A$ 
  <proof>
concrete-definition  $nbai\text{-igbg}$  uses  $nbai\text{-igbg}$ 
lemma  $nbai\text{-igbg}\text{-refine}$ [autoref-rules]:
  fixes  $S :: ('state_i \times 'state) \text{ set}$ 
  assumes  $SIDE\text{-GEN}\text{-ALGO } (is\text{-bounded}\text{-hashcode } S \text{ seq } bhc)$ 
  assumes  $SIDE\text{-GEN}\text{-ALGO } (is\text{-valid}\text{-def}\text{-hm}\text{-size } TYPE('state_i) \text{ hms})$ 
  assumes  $GEN\text{-OP } \text{seq } HOL.eq (S \rightarrow S \rightarrow \text{bool}\text{-rel})$ 
  shows  $(NBA\text{-Algorithms.nbai}\text{-igbg } \text{seq } bhc \text{ hms}, nba\text{-igbg}) \in$ 
     $\langle L, S \rangle \text{ nbai}\text{-nba}\text{-rel} \rightarrow \text{igbg}\text{-impl}\text{-rel}\text{-ext } \text{unit}\text{-rel } S$ 
  <proof>

schematic-goal  $nba\text{-language}\text{-empty}$ :
  fixes  $S :: ('state_i \times 'state) \text{ set}$ 
  assumes [simp]:  $igb\text{-fr}\text{-graph } (nba\text{-igbg } A)$ 
  assumes [autoref-ga-rules]:  $is\text{-bounded}\text{-hashcode } S \text{ seq } bhc$ 
  assumes [autoref-ga-rules]:  $is\text{-valid}\text{-def}\text{-hm}\text{-size } TYPE('state_i) \text{ hms}$ 
  assumes [autoref-rules]:  $(seq, HOL.eq) \in S \rightarrow S \rightarrow \text{bool}\text{-rel}$ 
  assumes [autoref-rules]:  $(Ai, A) \in \langle L, S \rangle \text{ nbai}\text{-nba}\text{-rel}$ 
  shows  $(?f :: ?'a, \text{do } \{ r \leftarrow \text{op}\text{-find}\text{-lasso}\text{-spec } (nba\text{-igbg } A); RETURN (r =$ 
     $None)\}) \in ?A$ 
  <proof>
concrete-definition  $nba\text{-language}\text{-empty}$  uses  $nba\text{-language}\text{-empty}$ 
lemma  $nba\text{-language}\text{-empty}\text{-refine}$ [autoref-rules]:
  fixes  $S :: ('state_i \times 'state) \text{ set}$ 
  assumes  $SIDE\text{-PRECOND } (finite (nodes A))$ 
  assumes  $SIDE\text{-GEN}\text{-ALGO } (is\text{-bounded}\text{-hashcode } S \text{ seq } bhc)$ 
  assumes  $SIDE\text{-GEN}\text{-ALGO } (is\text{-valid}\text{-def}\text{-hm}\text{-size } TYPE('state_i) \text{ hms})$ 
  assumes  $GEN\text{-OP } \text{seq } HOL.eq (S \rightarrow S \rightarrow \text{bool}\text{-rel})$ 
  assumes  $(Ai, A) \in \langle L, S \rangle \text{ nbai}\text{-nba}\text{-rel}$ 
  shows  $(NBA\text{-Algorithms.nba}\text{-language}\text{-empty } \text{seq } bhc \text{ hms } Ai,$ 
     $(OP \text{ op}\text{-language}\text{-empty} ::: \langle L, S \rangle \text{ nbai}\text{-nba}\text{-rel} \rightarrow \text{bool}\text{-rel}) \$ A) \in \text{bool}\text{-rel}$ 
  <proof>

end

end

```

## 42 Explicit Nondeterministic Büchi Automata

```

theory  $NBA\text{-Explicit}$ 
imports  $NBA\text{-Algorithms}$ 
begin

```

**datatype** ('label, 'state) nbae = nbae  
 (alphabet: 'label set)  
 (initiale: 'state set)  
 (transitione: ('state × 'label × 'state) set)  
 (acceptinge: 'state set)

**definition** nbae-rel where

[to-relAPP]: nbae-rel L S ≡ {(A<sub>1</sub>, A<sub>2</sub>).  
 (alphabet A<sub>1</sub>, alphabet A<sub>2</sub>) ∈ ⟨L⟩ set-rel ∧  
 (initiale A<sub>1</sub>, initiale A<sub>2</sub>) ∈ ⟨S⟩ set-rel ∧  
 (transitione A<sub>1</sub>, transitione A<sub>2</sub>) ∈ ⟨S ×<sub>r</sub> L ×<sub>r</sub> S⟩ set-rel ∧  
 (acceptinge A<sub>1</sub>, acceptinge A<sub>2</sub>) ∈ ⟨S⟩ set-rel}

**lemma** nbae-param[param, autoref-rules]:

(nbae, nbae) ∈ ⟨L⟩ set-rel → ⟨S⟩ set-rel → ⟨S ×<sub>r</sub> L ×<sub>r</sub> S⟩ set-rel →  
 ⟨S⟩ set-rel → ⟨L, S⟩ nbae-rel  
 (alphabet, alphabet) ∈ ⟨L, S⟩ nbae-rel → ⟨L⟩ set-rel  
 (initiale, initiale) ∈ ⟨L, S⟩ nbae-rel → ⟨S⟩ set-rel  
 (transitione, transitione) ∈ ⟨L, S⟩ nbae-rel → ⟨S ×<sub>r</sub> L ×<sub>r</sub> S⟩ set-rel  
 (acceptinge, acceptinge) ∈ ⟨L, S⟩ nbae-rel → ⟨S⟩ set-rel  
 ⟨proof⟩

**lemma** nbae-rel-id[simp]: ⟨Id, Id⟩ nbae-rel = Id ⟨proof⟩

**lemma** nbae-rel-comp[simp]: ⟨L<sub>1</sub> O L<sub>2</sub>, S<sub>1</sub> O S<sub>2</sub>⟩ nbae-rel = ⟨L<sub>1</sub>, S<sub>1</sub>⟩ nbae-rel O  
 ⟨L<sub>2</sub>, S<sub>2</sub>⟩ nbae-rel  
 ⟨proof⟩

**consts** i-nbae-scheme :: interface ⇒ interface ⇒ interface

**context**

**begin**

**interpretation** autoref-syn ⟨proof⟩

**lemma** nbae-scheme-itype[autoref-itype]:

nbae ::<sub>i</sub> ⟨L⟩<sub>i</sub> i-set →<sub>i</sub> ⟨S⟩<sub>i</sub> i-set →<sub>i</sub> ⟨⟨S, ⟨L, S⟩<sub>i</sub> i-prod⟩<sub>i</sub> i-prod⟩<sub>i</sub> i-set →<sub>i</sub> ⟨S⟩<sub>i</sub>  
 i-set →<sub>i</sub>  
 ⟨L, S⟩<sub>i</sub> i-nbae-scheme  
 alphabet ::<sub>i</sub> ⟨L, S⟩<sub>i</sub> i-nbae-scheme →<sub>i</sub> ⟨L⟩<sub>i</sub> i-set  
 initiale ::<sub>i</sub> ⟨L, S⟩<sub>i</sub> i-nbae-scheme →<sub>i</sub> ⟨S⟩<sub>i</sub> i-set  
 transitione ::<sub>i</sub> ⟨L, S⟩<sub>i</sub> i-nbae-scheme →<sub>i</sub> ⟨⟨S, ⟨L, S⟩<sub>i</sub> i-prod⟩<sub>i</sub> i-prod⟩<sub>i</sub> i-set  
 acceptinge ::<sub>i</sub> ⟨L, S⟩<sub>i</sub> i-nbae-scheme →<sub>i</sub> ⟨S⟩<sub>i</sub> i-set  
 ⟨proof⟩

**end**

**datatype** ('label, 'state) nbaei = nbaei  
 (alphabet: 'label list)

(*initaliaei*: 'state list)  
 (*transitioneiei*: ('state × 'label × 'state) list)  
 (*acceptingeiei*: 'state list)

**definition** *nbaei-rel* where

[*to-relAPP*]:  $nbaei-rel\ L\ S \equiv \{(A_1, A_2).$   
 (*alphabeteei*  $A_1$ , *alphabeteei*  $A_2\} \in \langle L \rangle\ list-rel \wedge$   
 (*initaliaei*  $A_1$ , *initaliaei*  $A_2\} \in \langle S \rangle\ list-rel \wedge$   
 (*transitioneiei*  $A_1$ , *transitioneiei*  $A_2\} \in \langle S \times_r L \times_r S \rangle\ list-rel \wedge$   
 (*acceptingeiei*  $A_1$ , *acceptingeiei*  $A_2\} \in \langle S \rangle\ list-rel\}$

**lemma** *nbaei-param*[*param*, *autoref-rules*]:

(*nbaei*, *nbaei*)  $\in \langle L \rangle\ list-rel \rightarrow \langle S \rangle\ list-rel \rightarrow \langle S \times_r L \times_r S \rangle\ list-rel \rightarrow$   
 $\langle S \rangle\ list-rel \rightarrow \langle L, S \rangle\ nbaei-rel$   
 (*alphabeteei*, *alphabeteei*)  $\in \langle L, S \rangle\ nbaei-rel \rightarrow \langle L \rangle\ list-rel$   
 (*initaliaei*, *initaliaei*)  $\in \langle L, S \rangle\ nbaei-rel \rightarrow \langle S \rangle\ list-rel$   
 (*transitioneiei*, *transitioneiei*)  $\in \langle L, S \rangle\ nbaei-rel \rightarrow \langle S \times_r L \times_r S \rangle\ list-rel$   
 (*acceptingeiei*, *acceptingeiei*)  $\in \langle L, S \rangle\ nbaei-rel \rightarrow \langle S \rangle\ list-rel$   
 <proof>

**definition** *nbaei-nbae-rel* where

[*to-relAPP*]:  $nbaei-nbae-rel\ L\ S \equiv \{(A_1, A_2).$   
 (*alphabeteei*  $A_1$ , *alphabeteei*  $A_2\} \in \langle L \rangle\ list-set-rel \wedge$   
 (*initaliaei*  $A_1$ , *initaliaei*  $A_2\} \in \langle S \rangle\ list-set-rel \wedge$   
 (*transitioneiei*  $A_1$ , *transitioneiei*  $A_2\} \in \langle S \times_r L \times_r S \rangle\ list-set-rel \wedge$   
 (*acceptingeiei*  $A_1$ , *acceptingeiei*  $A_2\} \in \langle S \rangle\ list-set-rel\}$

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of nbaei-nbae-rel i-nbae-scheme*]

**lemma** *nbaei-nbae-param*[*param*, *autoref-rules*]:

(*nbaei*, *nbae*)  $\in \langle L \rangle\ list-set-rel \rightarrow \langle S \rangle\ list-set-rel \rightarrow \langle S \times_r L \times_r S \rangle\ list-set-rel$   
 $\rightarrow \langle S \rangle\ list-set-rel \rightarrow \langle L, S \rangle\ nbaei-nbae-rel$   
 (*alphabeteei*, *alphabeteei*)  $\in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle L \rangle\ list-set-rel$   
 (*initaliaei*, *initaliaei*)  $\in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle S \rangle\ list-set-rel$   
 (*transitioneiei*, *transitioneiei*)  $\in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle S \times_r L \times_r S \rangle\ list-set-rel$   
 (*acceptingeiei*, *acceptingeiei*)  $\in \langle L, S \rangle\ nbaei-nbae-rel \rightarrow \langle S \rangle\ list-set-rel$   
 <proof>

**definition** *nbaei-nbae* where

$nbaei-nbae\ A \equiv nbae\ (set\ (alphabeteei\ A))\ (set\ (initaliaei\ A))$   
 $(set\ (transitioneiei\ A))\ (set\ (acceptingeiei\ A))$

**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>

**abbreviation** *transitions*  $L\ S\ s \equiv \bigcup a \in L. \bigcup p \in S. \{p\} \times \{a\} \times s\ a\ p$

**abbreviation** *succs*  $T\ a\ p \equiv (T\ \{\{p\}\}\ \{\{a\}\})$

**definition** *nba-nbae* **where**  $nba-nbae\ A \equiv nbae\ (alphabet\ A)\ (initial\ A)$   
 $(transitions\ (alphabet\ A)\ (nodes\ A)\ (transition\ A))\ (Set.filter\ (accepting\ A))$   
 $(nodes\ A)$

**definition** *nbae-nba* **where**  $nbae-nba\ A \equiv nba\ (alphabet\ A)\ (initial\ A)$   
 $(succs\ (transition\ A))\ (\lambda\ p.\ p \in\ accepting\ A)$

**lemma** *nba-nbae-param*[*param*]:  $(nba-nbae,\ nba-nbae) \in \langle L,\ S \rangle\ nba-rel \rightarrow \langle L,\ S \rangle\ nbae-rel$

*<proof>*

**lemma** *nbae-nba-param*[*param*]:

**assumes** *bijjective L* *bijjective S*

**shows**  $(nbae-nba,\ nbae-nba) \in \langle L,\ S \rangle\ nbae-rel \rightarrow \langle L,\ S \rangle\ nba-rel$

*<proof>*

**lemma** *nbae-nba-nba-nbae-param*[*param*]:

$((nbae-nba \circ nba-nbae)\ A,\ id\ A) \in \langle Id-on\ (alphabet\ A),\ Id-on\ (nodes\ A) \rangle\ nba-rel$   
*<proof>*

**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*]:  $(nbae-nba \circ nbaei-nbae,\ id) \in \langle L,\ S \rangle\ nbaei-nba-rel \rightarrow \langle L,\ S \rangle\ nba-rel$

*<proof>*

**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**  $(?f,\ nbae-nba) \in \langle L,\ S \rangle\ nbaei-nbae-rel \rightarrow \langle L,\ S \rangle\ nbae-nba-rel$

*<proof>*

**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**  $(nbae-nba-impl\ leq\ seq,\ nbae-nba) \in \langle L,\ S \rangle\ nbaei-nbae-rel \rightarrow \langle L,\ S \rangle\ nbae-nba-rel$

*<proof>*

**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] ⇒ do-bind (⟨⟨indent=2 notation=⟨infix do let⟩let - =/ -⟩ [1000, 13] 13⟩  
**syntax** *-do-let* :: [pttrn, 'a] ⇒ do-bind (⟨⟨indent=2 notation=⟨infix do let⟩let - =/ -⟩ 13⟩

## 44 Image on Explicit Automata

**definition** *nbae-image* **where** *nbae-image* *f* *A* ≡ *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*) ∈ (*S* → *T*) → ⟨*L*, *S*⟩ *nbae-rel* → ⟨*L*, *T*⟩ *nbae-rel*  
 ⟨*proof*⟩

**lemma** *nbae-image-id*[*simp*]: *nbae-image* *id* = *id* ⟨*proof*⟩

**lemma** *nbae-image-nba-nbae*: *nbae-image* *f* (*nba-nbae* *A*) = *nbae* (*alphabet* *A*) (*f* ' *initial* *A*)  
 (∪ *p* ∈ *nodes* *A*. ∪ *a* ∈ *alphabet* *A*. *f* ' {*p*} × {*a*} × *f* ' *transition* *A* *a* *p*)  
 (*f* ' {*p* ∈ *nodes* *A*. *accepting* *A* *p*})  
 ⟨*proof*⟩

## 45 Exploration and Translation

**definition** *trans-spec* **where**  
*trans-spec* *A* *f* ≡ ∪ *p* ∈ *nodes* *A*. ∪ *a* ∈ *alphabet* *A*. *f* ' {*p*} × {*a*} × *f* ' *transition* *A* *a* *p*

**definition** *trans-algo* **where**  
*trans-algo* *N* *L* *S* *f* ≡  
 FOREACH *N* (λ *p* *T*. do {  
 ASSERT (*p* ∈ *N*);  
 FOREACH *L* (λ *a* *T*. do {  
 ASSERT (*a* ∈ *L*);  
 FOREACH (*S* *a* *p*) (λ *q* *T*. do {  
 ASSERT (*q* ∈ *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*  
 ⟨*proof*⟩

**definition** *nba-image* :: ('state<sub>1</sub> ⇒ 'state<sub>2</sub>) ⇒ ('label, 'state<sub>1</sub>) nba ⇒ ('label, 'state<sub>2</sub>) nba **where**  
*nba-image* f A ≡ nba  
 (alphabet A)  
 (f 'initial A)  
 (λ a p. f 'transition A a (inv-into (nodes A) f p))  
 (λ p. accepting A (inv-into (nodes A) f p))

**lemma** *nba-image-rel*[param]:  
**assumes** inj-on f (nodes A)  
**shows** (A, *nba-image* f A) ∈ ⟨Id-on (alphabet A), br f (λ p. p ∈ nodes A)⟩  
*nba-rel*  
 ⟨proof⟩

**lemma** *nba-image-nodes*[simp]:  
**assumes** inj-on f (nodes A)  
**shows** nodes (*nba-image* f A) = f 'nodes A  
 ⟨proof⟩

**lemma** *nba-image-language*[simp]:  
**assumes** inj-on f (nodes A)  
**shows** language (*nba-image* f A) = language A  
 ⟨proof⟩

**lemma** *nba-image-nbae*:  
**assumes** inj-on f (nodes A)  
**shows** nbae-image f (nba-nbae A) = nba-nbae (*nba-image* f A)  
 ⟨proof⟩

**definition** *op-translate* :: ('label, 'state) nba ⇒ ('label, nat) nbae nres **where**  
*op-translate* A ≡ SPEC (λ B. ∃ f. inj-on f (nodes A) ∧ B = nba-nbae (*nba-image* f A))

**lemma** *op-translate-language*:  
**assumes** (RETURN Ai, *op-translate* A) ∈ ⟨⟨Id, nat-rel⟩ nbaei-nbae-rel⟩ nres-rel  
**shows** language (nbae-nba (*nbaei-nbae* Ai)) = language A  
 ⟨proof⟩

**schematic-goal** *to-nbaei-impl*:  
**fixes** S :: ('statei × '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
⟨proof⟩
concrete-definition to-nbaei-impl uses to-nbaei-impl

context
begin

  interpretation autoref-syn ⟨proof⟩

  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⟩

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**  $ngba-g :: ('label, 'state) ngba \Rightarrow 'state \text{ graph-rec where}$   
 $ngba-g A \equiv \langle \langle g-V = UNIV, g-E = E\text{-of-succ (successors } A), g-V0 = \text{initial } A$   
 $\rangle \rangle$

**lemma**  $ngba-g-graph[simp]: graph (ngba-g A) \langle proof \rangle$

**lemma**  $ngba-g-V0: g-V0 (ngba-g A) = \text{initial } A \langle proof \rangle$

**lemma**  $ngba-g-E-rtrancl: (g-E (ngba-g A))^* = \{(p, q). q \in \text{reachable } A p\}$   
 $\langle proof \rangle$

**lemma**  $ngba-g-rtrancl-path: (g-E (ngba-g A))^* = \{(p, \text{target } r p) \mid r p. \text{NGBA.path}$   
 $A r p\}$   
 $\langle proof \rangle$

**lemma**  $ngba-g-trancl-path: (g-E (ngba-g A))^+ = \{(p, \text{target } r p) \mid r p. \text{NGBA.path}$   
 $A r p \wedge r \neq []\}$   
 $\langle proof \rangle$

**lemma**  $ngba-g-ipath-run:$   
**assumes**  $ipath (g-E (ngba-g A)) r$   
**obtains**  $w$   
**where**  $run A (w \parallel smap (r \circ Suc) nats) (r 0)$   
 $\langle proof \rangle$

**lemma**  $ngba-g-run-ipath:$   
**assumes**  $run A (w \parallel r) p$   
**shows**  $ipath (g-E (ngba-g A)) (snth (p \#\# r))$   
 $\langle proof \rangle$

## 46.2 Indexed Generalized Büchi Graphs

**definition**  $ngba-acc :: 'state \text{ pred gen} \Rightarrow 'state \Rightarrow \text{nat set where}$   
 $ngba-acc cs p \equiv \{k \in \{0 ..< \text{length } cs\}. (cs ! k) p\}$

**lemma**  $ngba-acc-param[param]: (ngba-acc, ngba-acc) \in \langle S \rightarrow \text{bool-rel} \rangle \text{list-rel} \rightarrow$   
 $S \rightarrow \langle \text{nat-rel} \rangle \text{set-rel}$   
 $\langle proof \rangle$

**definition**  $ngba-igbg :: ('label, 'state) ngba \Rightarrow 'state \text{ igb-graph-rec where}$   
 $ngba-igbg A \equiv \text{graph-rec.extend (ngba-g A) } \langle \langle \text{igbg-num-acc} = \text{length (accepting}$   
 $A), \text{igbg-acc} = \text{ngba-acc (accepting } A) \rangle \rangle$

**lemma**  $acc-run-language:$   
**assumes**  $igbg-graph (ngba-igbg A)$   
**shows**  $Ex (igbg-graph.is-acc-run (ngba-igbg A)) \longleftrightarrow \text{language } A \neq \{\}$   
 $\langle proof \rangle$

**end**

## 47 Relations on Nondeterministic Generalized Büchi Automata

**theory** *NGBA-Refine*

**imports**

*NGBA*

*../Transition-Systems/Transition-System-Refine*

**begin**

**definition** *ngba-rel* :: ('label<sub>1</sub> × 'label<sub>2</sub>) set ⇒ ('state<sub>1</sub> × 'state<sub>2</sub>) set ⇒  
 (('label<sub>1</sub>, 'state<sub>1</sub>) ngba × ('label<sub>2</sub>, 'state<sub>2</sub>) ngba) set **where**  
 [*to-relAPP*]: *ngba-rel* L S ≡ {(A<sub>1</sub>, A<sub>2</sub>).  
 (alphabet A<sub>1</sub>, alphabet A<sub>2</sub>) ∈ ⟨L⟩ set-rel ∧  
 (initial A<sub>1</sub>, initial A<sub>2</sub>) ∈ ⟨S⟩ set-rel ∧  
 (transition A<sub>1</sub>, transition A<sub>2</sub>) ∈ L → S → ⟨S⟩ set-rel ∧  
 (accepting A<sub>1</sub>, accepting A<sub>2</sub>) ∈ ⟨S → bool-rel⟩ list-rel}

**lemma** *ngba-param*[*param*]:

(ngba, ngba) ∈ ⟨L⟩ set-rel → ⟨S⟩ set-rel → (L → S → ⟨S⟩ set-rel) → ⟨S →  
 bool-rel⟩ list-rel →  
 ⟨L, S⟩ ngba-rel  
 (alphabet, alphabet) ∈ ⟨L, S⟩ ngba-rel → ⟨L⟩ set-rel  
 (initial, initial) ∈ ⟨L, S⟩ ngba-rel → ⟨S⟩ set-rel  
 (transition, transition) ∈ ⟨L, S⟩ ngba-rel → L → S → ⟨S⟩ set-rel  
 (accepting, accepting) ∈ ⟨L, S⟩ ngba-rel → ⟨S → bool-rel⟩ list-rel  
 ⟨proof⟩

**lemma** *ngba-rel-id*[*simp*]: ⟨Id, Id⟩ ngba-rel = Id ⟨proof⟩

**lemma** *enableds-param*[*param*]: (ngba.enableds, ngba.enableds) ∈ ⟨L, S⟩ ngba-rel  
 → S → ⟨L ×<sub>r</sub> S⟩ set-rel  
 ⟨proof⟩

**lemma** *paths-param*[*param*]: (ngba.paths, ngba.paths) ∈ ⟨L, S⟩ ngba-rel → S →  
 ⟨⟨L ×<sub>r</sub> S⟩ list-rel⟩ set-rel  
 ⟨proof⟩

**lemma** *runs-param*[*param*]: (ngba.runs, ngba.runs) ∈ ⟨L, S⟩ ngba-rel → S → ⟨⟨L  
 ×<sub>r</sub> S⟩ stream-rel⟩ set-rel  
 ⟨proof⟩

**lemma** *reachable-param*[*param*]: (reachable, reachable) ∈ ⟨L, S⟩ ngba-rel → S →  
 ⟨S⟩ set-rel  
 ⟨proof⟩

**lemma** *nodes-param*[*param*]: (nodes, nodes) ∈ ⟨L, S⟩ ngba-rel → ⟨S⟩ set-rel  
 ⟨proof⟩

**lemma** *gen-param*[*param*]: (gen, gen) ∈ (A → B → bool-rel) → ⟨A⟩ list-rel → B  
 → bool-rel  
 ⟨proof⟩

**lemma** *language-param*[*param*]: (*language*, *language*) ∈ ⟨*L*, *S*⟩ *ngba-rel* → ⟨⟨*L*⟩  
*stream-rel*⟩ *set-rel*  
 ⟨*proof*⟩

**end**

## 48 Implementation of Nondeterministic Generalized Büchi Automata

**theory** *NGBA-Implement*

**imports**

*NGBA-Refine*

*../Basic/Implement*

**begin**

**consts** *i-ngba-scheme* :: *interface* ⇒ *interface* ⇒ *interface*

**context**

**begin**

**interpretation** *autoref-syn* ⟨*proof*⟩

**lemma** *ngba-scheme-itype*[*autoref-itype*]:

*ngba* ::<sub>*i*</sub> ⟨*L*⟩<sub>*i*</sub> *i-set* →<sub>*i*</sub> ⟨*S*⟩<sub>*i*</sub> *i-set* →<sub>*i*</sub> (*L* →<sub>*i*</sub> *S* →<sub>*i*</sub> ⟨*S*⟩<sub>*i*</sub> *i-set*) →<sub>*i*</sub> ⟨⟨*S*⟩<sub>*i*</sub> *i-set*⟩<sub>*i*</sub>  
*i-list* →<sub>*i*</sub>

⟨*L*, *S*⟩<sub>*i*</sub> *i-ngba-scheme*

*alphabet* ::<sub>*i*</sub> ⟨*L*, *S*⟩<sub>*i*</sub> *i-ngba-scheme* →<sub>*i*</sub> ⟨*L*⟩<sub>*i*</sub> *i-set*

*initial* ::<sub>*i*</sub> ⟨*L*, *S*⟩<sub>*i*</sub> *i-ngba-scheme* →<sub>*i*</sub> ⟨*S*⟩<sub>*i*</sub> *i-set*

*transition* ::<sub>*i*</sub> ⟨*L*, *S*⟩<sub>*i*</sub> *i-ngba-scheme* →<sub>*i*</sub> *L* →<sub>*i*</sub> *S* →<sub>*i*</sub> ⟨*S*⟩<sub>*i*</sub> *i-set*

*accepting* ::<sub>*i*</sub> ⟨*L*, *S*⟩<sub>*i*</sub> *i-ngba-scheme* →<sub>*i*</sub> ⟨⟨*S*⟩<sub>*i*</sub> *i-set*⟩<sub>*i*</sub> *i-list*

⟨*proof*⟩

**end**

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

(*alphabeti*: *'label list*)

(*initiali*: *'state list*)

(*transitioni*: *'label* ⇒ *'state* ⇒ *'state list*)

(*acceptingi*: (*'state* ⇒ *bool*) *list*)

**definition** *ngbai-rel* :: (*'label*<sub>1</sub> × *'label*<sub>2</sub>) *set* ⇒ (*'state*<sub>1</sub> × *'state*<sub>2</sub>) *set* ⇒

((*'label*<sub>1</sub>, *'state*<sub>1</sub>) *ngbai* × (*'label*<sub>2</sub>, *'state*<sub>2</sub>) *ngbai*) *set* **where**

[*to-relAPP*]: *ngbai-rel* *L S* ≡ {(*A*<sub>1</sub>, *A*<sub>2</sub>).

(*alphabeti* *A*<sub>1</sub>, *alphabeti* *A*<sub>2</sub>) ∈ ⟨*L*⟩ *list-rel* ∧

(*initiali* *A*<sub>1</sub>, *initiali* *A*<sub>2</sub>) ∈ ⟨*S*⟩ *list-rel* ∧

(*transitioni* *A*<sub>1</sub>, *transitioni* *A*<sub>2</sub>) ∈ *L* → *S* → ⟨*S*⟩ *list-rel* ∧

(*acceptingi* *A*<sub>1</sub>, *acceptingi* *A*<sub>2</sub>) ∈ ⟨*S* → *bool-rel*⟩ *list-rel*}

**lemma** *ngbai-param*[*param*]:

$(ngbai, ngbai) \in \langle L \rangle \text{ list-rel} \rightarrow \langle S \rangle \text{ list-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle \text{ list-rel}) \rightarrow$   
 $\langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ ngbai-rel}$   
 $(\text{alphabeti}, \text{alphabeti}) \in \langle L, S \rangle \text{ ngbai-rel} \rightarrow \langle L \rangle \text{ list-rel}$   
 $(\text{initiali}, \text{initiali}) \in \langle L, S \rangle \text{ ngbai-rel} \rightarrow \langle S \rangle \text{ list-rel}$   
 $(\text{transitioni}, \text{transitioni}) \in \langle L, S \rangle \text{ ngbai-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ list-rel}$   
 $(\text{acceptingi}, \text{acceptingi}) \in \langle L, S \rangle \text{ ngbai-rel} \rightarrow \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**definition** *ngbai-ngba-rel* :: ('label<sub>1</sub> × 'state<sub>1</sub>) set ⇒ ('state<sub>1</sub> × 'state<sub>2</sub>) set ⇒  
 (('label<sub>1</sub>, 'state<sub>1</sub>) ngbai × ('label<sub>2</sub>, 'state<sub>2</sub>) ngba) set **where**  
 [*to-relAPP*]: *ngbai-ngba-rel* *L S* ≡ {(*A*<sub>1</sub>, *A*<sub>2</sub>).  
 (*alphabeti* *A*<sub>1</sub>, *alphabet* *A*<sub>2</sub>) ∈ ⟨*L*⟩ list-set-rel ∧  
 (*initiali* *A*<sub>1</sub>, *initial* *A*<sub>2</sub>) ∈ ⟨*S*⟩ list-set-rel ∧  
 (*transitioni* *A*<sub>1</sub>, *transition* *A*<sub>2</sub>) ∈ *L* → *S* → ⟨*S*⟩ list-set-rel ∧  
 (*acceptingi* *A*<sub>1</sub>, *accepting* *A*<sub>2</sub>) ∈ ⟨*S* → bool-rel⟩ list-rel}

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of ngbai-ngba-rel i-ngba-scheme*]

**lemma** *ngbai-ngba-param*[*param*, *autoref-rules*]:

$(ngbai, ngba) \in \langle L \rangle \text{ list-set-rel} \rightarrow \langle S \rangle \text{ list-set-rel} \rightarrow (L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel})$   
 $\rightarrow$   
 $\langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel} \rightarrow \langle L, S \rangle \text{ ngbai-ngba-rel}$   
 $(\text{alphabeti}, \text{alphabet}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle L \rangle \text{ list-set-rel}$   
 $(\text{initiali}, \text{initial}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle S \rangle \text{ list-set-rel}$   
 $(\text{transitioni}, \text{transition}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow L \rightarrow S \rightarrow \langle S \rangle \text{ list-set-rel}$   
 $(\text{acceptingi}, \text{accepting}) \in \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \langle S \rightarrow \text{bool-rel} \rangle \text{ list-rel}$   
 $\langle \text{proof} \rangle$

**definition** *ngbai-ngba* :: ('label, 'state) ngbai ⇒ ('label, 'state) ngba **where**  
*ngbai-ngba* *A* ≡ *ngba* (set (*alphabeti* *A*)) (set (*initiali* *A*)) (λ *a p*. set (*transitioni* *A* *a p*)) (*acceptingi* *A*)

**definition** *ngbai-invar* :: ('label, 'state) ngbai ⇒ bool **where**  
*ngbai-invar* *A* ≡ *distinct* (*alphabeti* *A*) ∧ *distinct* (*initiali* *A*) ∧ (∀ *a p*. *distinct* (*transitioni* *A* *a p*))

**lemma** *ngbai-ngba-id-param*[*param*]: (*ngbai-ngba*, *id*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel* →  
 ⟨*L*, *S*⟩ *ngba-rel*  
 ⟨*proof*⟩

**lemma** *ngbai-ngba-br*: ⟨*Id*, *Id*⟩ *ngbai-ngba-rel* = *br ngbai-ngba ngbai-invar*  
 ⟨*proof*⟩

**end**

**theory** *Degeneralization-Refine*

**imports** *Degeneralization Refine*

**begin**

**lemma** *degen-param*[*param*]: (*degen*, *degen*) ∈ ⟨*S* → *bool-rel*⟩ *list-rel* → *S* ×<sub>*r*</sub>  
*nat-rel* → *bool-rel*  
 ⟨*proof*⟩

**lemma** *count-param*[*param*]: (*Degeneralization.count*, *Degeneralization.count*) ∈  
 ⟨*A* → *bool-rel*⟩ *list-rel* → *A* → *nat-rel* → *nat-rel*  
 ⟨*proof*⟩

**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* ⟨*proof*⟩

**lemma** *ngba-g-ahs*: *ngba-g* *A* = (| *g-V* = *UNIV*, *g-E* = *E-of-succ* (λ *p*. *CAST*  
 ((∪ *a* ∈ *ngba.alphabet* *A*. *ngba.transition* *A* *a* *p* :: ⟨*S*⟩ *list-set-rel*) :: ⟨*S*⟩  
*ahs-rel* *bhc*)),  
*g-V0* = *ngba.initial* *A* |)  
 ⟨*proof*⟩

**schematic-goal** *ngbai-gi*:

**notes** [*autoref-ga-rules*] = *map2set-to-list*

**fixes** *S* :: ('*statei* × '*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*) ∈ *S* → *S* → *bool-rel*

**assumes** [*autoref-rules*]: (*Ai*, *A*) ∈ ⟨*L*, *S*⟩ *ngbai-ngba-rel*

**shows** ( $?f :: ?'a, RETURN (ngba-g A) \in ?A$ )  
 $\langle proof \rangle$

**concrete-definition** *ngbai-gi uses ngbai-gi*

**lemma** *ngbai-gi-refine*[*autoref-rules*]:

**fixes**  $S :: ('state_i \times 'state) set$

**assumes** *SIDE-GEN-ALGO* (*is-bounded-hashcode*  $S seq bhc$ )

**assumes** *SIDE-GEN-ALGO* (*is-valid-def-hm-size*  $TYPE('state_i) hms$ )

**assumes** *GEN-OP seq HOL.eq* ( $S \rightarrow S \rightarrow bool-rel$ )

**shows** (*NGBA-Algorithms.ngbai-gi seq bhc hms, ngba-g*)  $\in$

$\langle L, S \rangle ngbai-ngba-rel \rightarrow \langle unit-rel, S \rangle g-impl-rel-ext$

$\langle proof \rangle$

**schematic-goal** *ngba-nodes*:

**fixes**  $S :: ('state_i \times 'state) set$

**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*  $TYPE('state_i) hms$

**assumes** [*autoref-rules*]: (*seq, HOL.eq*)  $\in S \rightarrow S \rightarrow bool-rel$

**assumes** [*autoref-rules*]: ( $A_i, A \in \langle L, S \rangle ngbai-ngba-rel$ )

**shows** ( $?f :: ?'a, op-reachable (ngba-g A) \in ?R$ )  $\langle proof \rangle$

**concrete-definition** *ngba-nodes uses ngba-nodes*

**lemma** *ngba-nodes-refine*[*autoref-rules*]:

**fixes**  $S :: ('state_i \times 'state) set$

**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*  $TYPE('state_i) hms$ )

**assumes** *GEN-OP seq HOL.eq* ( $S \rightarrow S \rightarrow bool-rel$ )

**assumes** ( $A_i, A \in \langle L, S \rangle ngbai-ngba-rel$ )

**shows** (*NGBA-Algorithms.ngba-nodes seq bhc hms*  $A_i,$

$(OP\ NGBA.nodes :: \langle L, S \rangle ngbai-ngba-rel \rightarrow \langle S \rangle ahs-rel bhc) \$ A \in \langle S \rangle$

*ahs-rel bhc*

$\langle proof \rangle$

**lemma** *ngba-igbg-ahs*:  $ngba-igbg A = \langle \mid g-V = UNIV, g-E = E-of-succ (\lambda p.$   
*CAST*

$(\langle \bigcup a \in NGBA.alphabet A. NGBA.transition A a p :: \langle S \rangle list-set-rel \rangle :: \langle S \rangle$   
*ahs-rel bhc*),  $g-V0 = NGBA.initial A,$

$igbg-num-acc = length (NGBA.accepting A), igbg-acc = ngba-acc (NGBA.accepting$   
 $A) \rangle$

$\langle proof \rangle$

**definition**

$ngba-acc-bs\ cs\ p \equiv fold (\lambda (k, c) bs. if\ c\ p\ then\ bs-insert\ k\ bs\ else\ bs)$   
*(indexed-from 0 cs) (bs-empty ())*

**lemma** *ngba-acc-bs-empty*[*simp*]:  $ngba-acc-bs [] p = bs-empty ()$   $\langle proof \rangle$

**lemma** *ngba-acc-bs-insert*[*simp*]:

**assumes**  $c\ p$

**shows**  $ngba-acc-bs (cs @ [c]) p = bs-insert (length cs) (ngba-acc-bs cs p)$

$\langle \text{proof} \rangle$   
**lemma** *ngba-acc-bs-skip*[simp]:  
**assumes**  $\neg c\ p$   
**shows** *ngba-acc-bs* (*cs* @ [*c*]) *p* = *ngba-acc-bs cs p*  
 $\langle \text{proof} \rangle$

**lemma** *ngba-acc-bs-correct*[simp]: *bs- $\alpha$*  (*ngba-acc-bs cs p*) = *ngba-acc cs p*  
 $\langle \text{proof} \rangle$

**lemma** *ngba-acc-impl-bs*[autoref-rules]: (*ngba-acc-bs*, *ngba-acc*)  $\in \langle S \rightarrow \text{bool-rel} \rangle$   
*list-rel*  $\rightarrow S \rightarrow \langle \text{nat-rel} \rangle$  *bs-set-rel*  
 $\langle \text{proof} \rangle$

**schematic-goal** *ngbai-igbgi*:  
**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*)  $\in S \rightarrow S \rightarrow \text{bool-rel}$   
**assumes** [*autoref-rules*]: (*Ai*, *A*)  $\in \langle L, S \rangle$  *ngbai-ngba-rel*  
**shows** (*?f* :: ?'a, *RETURN* (*ngba-igbg A*))  $\in ?A$   
 $\langle \text{proof} \rangle$

**concrete-definition** *ngbai-igbgi* **uses** *ngbai-igbgi*  
**lemma** *ngbai-igbgi-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** (*NGBA-Algorithms.ngbai-igbgi seq bhc hms*, *ngba-igbg*)  $\in$   
 $\langle L, S \rangle$  *ngbai-ngba-rel*  $\rightarrow$  *igbg-impl-rel-ext unit-rel S*  
 $\langle \text{proof} \rangle$

**schematic-goal** *ngba-language-empty*:  
**fixes** *S* :: ('statei  $\times$  'state) set  
**assumes** [*simp*]: *igb-fr-graph* (*ngba-igbg A*)  
**assumes** [*autoref-ga-rules*]: *is-bounded-hashcode S seq bhs*  
**assumes** [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE('statei) hms*  
**assumes** [*autoref-rules*]: (*seq*, *HOL.eq*)  $\in S \rightarrow S \rightarrow \text{bool-rel}$   
**assumes** [*autoref-rules*]: (*Ai*, *A*)  $\in \langle L, S \rangle$  *ngbai-ngba-rel*  
**shows** (*?f* :: ?'a, *do* { *r*  $\leftarrow$  *op-find-lasso-spec* (*ngba-igbg A*); *RETURN* (*r* =  
*None*)} )  $\in ?A$   
 $\langle \text{proof} \rangle$

**concrete-definition** *ngba-language-empty* **uses** *ngba-language-empty*  
**lemma** *nba-language-empty-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-rel*)

**assumes**  $(Ai, A) \in \langle L, S \rangle$  *ngbai-ngba-rel*  
**shows**  $(NGBA-Algorithms.ngba-language-empty \text{ seq } hbc \text{ hms } Ai,$   
 $(OP \text{ op-language-empty} :: \langle L, S \rangle \text{ ngbai-ngba-rel} \rightarrow \text{bool-rel}) \$ A) \in \text{bool-rel}$   
 $\langle \text{proof} \rangle$

**lemma** *degeneralize-alt-def: degeneralize A = nba*  
 $(ngba.alphabet A)$   
 $((\lambda p. (p, 0)) \text{ ' ngba.initial } A)$   
 $(\lambda a (p, k). (\lambda q. (q, Degeneralization.count (ngba.accepting A) p k)) \text{ ' ngba.transition } A a p)$   
 $(degen (ngba.accepting A))$   
 $\langle \text{proof} \rangle$

**schematic-goal** *ngba-degeneralize: (?f :: ?'a, degeneralize) ∈ ?R*  
 $\langle \text{proof} \rangle$

**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**  $(?f, intersect') \in \langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \langle L, T \rangle \text{ nbai-nba-rel} \rightarrow \langle L, S$   
 $\times_r T \rangle \text{ ngbai-ngba-rel}$   
 $\langle \text{proof} \rangle$

**concrete-definition** *nba-intersect' uses nba-intersect'*

**lemma** *nba-intersect'-refine[autoref-rules]:*

**assumes**  $GEN-OP \text{ seq } HOL.eq (L \rightarrow L \rightarrow \text{bool-rel})$

**shows**  $(nba-intersect' \text{ seq}, intersect') \in$

$\langle L, S \rangle \text{ nbai-nba-rel} \rightarrow \langle L, T \rangle \text{ nbai-nba-rel} \rightarrow \langle L, S \times_r T \rangle \text{ ngbai-ngba-rel}$

$\langle \text{proof} \rangle$

**end**

**end**

## 50 Nondeterministic Büchi Transition Automata

**theory** *NBTA*

**imports** *../Nondeterministic*

**begin**

**datatype**  $(\text{'label}, \text{'state}) \text{ nbta} = \text{nbta}$   
 $(alphabet: \text{'label set})$   
 $(initial: \text{'state set})$   
 $(transition: \text{'label} \Rightarrow \text{'state} \Rightarrow \text{'state set})$   
 $(accepting: (\text{'state} \times \text{'label} \times \text{'state}) \text{ pred})$

**global-interpretation** *nbta: automaton nbta alphabet initial transition accepting*

**defines**  $\text{path} = \text{nbta.path}$  **and**  $\text{run} = \text{nbta.run}$  **and**  $\text{reachable} = \text{nbta.reachable}$   
**and**  $\text{nodes} = \text{nbta.nodes}$

```

    <proof>
global-interpretation nbta: automaton-run nbta alphabet initial transition ac-
cepting
    λ P w r p. infs P (p ## r ||| w ||| r)
    defines language = nbta.language
    <proof>

abbreviation target where target ≡ nbta.target
abbreviation states where states ≡ nbta.states
abbreviation trace where trace ≡ nbta.trace
abbreviation successors where successors ≡ 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 ⇒ 'state ⇒ 'state set)
      (accepting: ('state × 'label × '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
    <proof>
    global-interpretation ngbta: automaton-run ngbta alphabet initial transition
accepting
    λ P w r p. gen infs P (p ## r ||| w ||| r)
    defines language = ngbta.language
    <proof>

abbreviation target where target ≡ ngbta.target
abbreviation states where states ≡ ngbta.states
abbreviation trace where trace ≡ ngbta.trace
abbreviation successors where successors ≡ 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*  $P\ (p\ \#\#\ r\ ||| w\ ||| r)$   
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $id\ \lambda\ ((p,\ k),\ a,\ (q,\ l)).\ ((p,\ a,\ q),\ k)$   
**defines** *degeneralize* = *degeneralization.degeneralize*  
 $\langle proof \rangle$

**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*  
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $ngbta\ ngbta.alphabet\ ngbta.initial\ ngbta.transition\ ngbta.accepting\ \lambda\ P\ w\ r\ p.$   
*gen infs*  $P\ (p\ \#\#\ r\ ||| w\ ||| r)$   
 $\lambda\ c_1\ c_2.\ [c_1\ \circ\ (\lambda\ ((p_1,\ p_2),\ a,\ (q_1,\ q_2)).\ (p_1,\ a,\ q_1)),\ c_2\ \circ\ (\lambda\ ((p_1,\ p_2),\ a,\ (q_1,\ q_2)).\ (p_2,\ a,\ q_2))]$   
**defines** *intersect'* = *intersection.product*  
 $\langle proof \rangle$

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

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

**global-interpretation** *union: automaton-union-run*  
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $nbta\ nbta.alphabet\ nbta.initial\ nbta.transition\ nbta.accepting\ \lambda\ P\ w\ r\ p.\ infs\ P$   
 $(p\ \#\#\ r\ ||| w\ ||| r)$   
 $\lambda\ c_1\ c_2\ m.\ case\ m\ of\ (Inl\ p,\ a,\ Inl\ q) \Rightarrow c_1\ (p,\ a,\ q) \mid (Inr\ p,\ a,\ Inr\ q) \Rightarrow c_2\ (p,\ a,\ q)$   
**defines** *union* = *union.sum*  
 $\langle proof \rangle$

**lemmas** *union-language* = *union.sum-language*  
**lemmas** *union-nodes-finite* = *union.sum-nodes-finite*

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

**lemma** *intersect-language[simp]*: *NBTA.language (intersect A B) = NBTA.language A*  $\cap$  *NBTA.language B*  
*<proof>*

**lemma** *intersect-nodes-finite[intro]*:  
**assumes** *finite (NBTA.nodes A)* *finite (NBTA.nodes B)*  
**shows** *finite (NBTA.nodes (intersect A B))*  
*<proof>*

**end**