

The CAVA Automata Library

Peter Lammich

March 17, 2025

Abstract

We report on the graph and automata library that is used in the fully verified LTL model checker CAVA. As most components of CAVA use some type of graphs or automata, a common automata library simplifies assembly of the components and reduces redundancy.

The CAVA Automata Library provides a hierarchy of graph and automata classes, together with some standard algorithms. Its object oriented design allows for sharing of algorithms, theorems, and implementations between its classes, and also simplifies extensions of the library. Moreover, it is integrated into the Automatic Refinement Framework, supporting automatic refinement of the abstract automata types to efficient data structures.

Note that the CAVA Automata Library is work in progress. Currently, it is very specifically tailored towards the requirements of the CAVA model checker. Nevertheless, the formalization techniques presented here allow an extension of the library to a wider scope. Moreover, they are not limited to graph libraries, but apply to class hierarchies in general.

The CAVA Automata Library is described in the paper: Peter Lammich, The CAVA Automata Library, Isabelle Workshop 2014, to appear.

Contents

1	Relations interpreted as Directed Graphs	3
1.1	Paths	3
1.2	Infinite Paths	7
1.3	Strongly Connected Components	8
2	Directed Graphs	11
2.1	Directed Graphs with Explicit Node Set and Set of Initial Nodes	11
3	Automata	15
3.1	Generalized Buchi Graphs	15
3.2	Generalized Buchi Automata	17
3.3	Buchi Graphs	19
3.4	Buchi Automata	19
3.5	Indexed acceptance classes	20
3.5.1	Indexing Conversion	22
3.5.2	Degeneralization	25
3.6	System Automata	27
3.6.1	Product Construction	28
4	Lassos	29
4.1	Implementing runs by lassos	35
5	Simulation	36
6	Simulation	36
6.1	Functional Relations	36
6.2	Relation between Runs	36
6.3	Simulation	37
6.4	Bisimulation	38
7	Implementing Graphs	44
7.1	Directed Graphs by Successor Function	44
7.1.1	Restricting Edges	45
7.2	Rooted Graphs	47
7.2.1	Operation Identification Setup	47
7.2.2	Generic Implementation	48
7.2.3	Implementation with list-set for Nodes	48
7.2.4	Implementation with Cfun for Nodes	49
7.2.5	Renaming	50
7.3	Graphs from Lists	52

8	Implementing Automata	53
8.1	Indexed Generalized Buchi Graphs	53
8.1.1	Implementation with bit-set	55
8.2	Indexed Generalized Buchi Automata	56
8.2.1	Implementation as function	57
8.3	Generalized Buchi Graphs	58
8.3.1	Implementation with list of lists	60
8.4	GBAs	61
8.4.1	Implementation as function	62
8.5	Buchi Graphs	63
8.5.1	Implementation with Characteristic Functions	64
8.6	System Automata	65
8.6.1	Implementation with Function	67
8.7	Index Conversion	68
8.8	Degeneralization	69
8.9	Product Construction	70

1 Relations interpreted as Directed Graphs

```
theory Digraph-Basic
imports
  Automatic-Refinement.Misc
  Automatic-Refinement.Refine-Util
  HOL-Library.Omega-Words-Fun
begin
```

This theory contains some basic graph theory on directed graphs which are modeled as a relation between nodes.

The theory here is very fundamental, and also used by non-directly graph-related applications like the theory of tail-recursion in the Refinement Framework. Thus, we decided to put it in the basic theories of the refinement framework.

Directed graphs are modeled as a relation on nodes

```
type-synonym 'v digraph = ('v × 'v) set
```

```
locale digraph = fixes E :: 'v digraph
```

1.1 Paths

Path are modeled as list of nodes, the last node of a path is not included into the list. This formalization allows for nice concatenation and splitting of paths.

```
inductive path :: 'v digraph ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool for E where
  path0: path E u [] u
| path-prepend:  $\llbracket (u,v) \in E; \text{path } E \ v \ l \ w \rrbracket \implies \text{path } E \ u \ (u\#l) \ w$ 
```

```
lemma path1:  $(u,v) \in E \implies \text{path } E \ u \ [u] \ v$ 
  <proof>
```

```
lemma path-empty-conv[simp]:
  path E u [] v  $\longleftrightarrow u=v$ 
  <proof>
```

```
inductive-cases path-uncons: path E u (u'\#l) w
```

```
inductive-simps path-cons-conv: path E u (u'\#l) w
```

```
lemma path-no-edges[simp]: path {} u p v  $\longleftrightarrow (u=v \wedge p=[])$ 
  <proof>
```

```
lemma path-conc:
  assumes P1: path E u la v
  assumes P2: path E v lb w
  shows path E u (la@lb) w
  <proof>
```

lemma *path-append*:

$\llbracket \text{path } E \ u \ l \ v; (v,w) \in E \rrbracket \implies \text{path } E \ u \ (l@[v]) \ w$
<proof>

lemma *path-unconc*:

assumes $\text{path } E \ u \ (la@lb) \ w$
obtains v **where** $\text{path } E \ u \ la \ v$ **and** $\text{path } E \ v \ lb \ w$
<proof>

lemma *path-conc-conv*:

$\text{path } E \ u \ (la@lb) \ w \longleftrightarrow (\exists v. \text{path } E \ u \ la \ v \wedge \text{path } E \ v \ lb \ w)$
<proof>

lemma (**in** $-$) *path-append-conv*: $\text{path } E \ u \ (p@[v]) \ w \longleftrightarrow (\text{path } E \ u \ p \ v \wedge (v,w) \in E)$
<proof>

lemmas *path-simps* = *path-empty-conv path-cons-conv path-conc-conv*

lemmas *path-trans*[*trans*] = *path-prepend path-conc path-append*

lemma *path-from-edges*: $\llbracket (u,v) \in E; (v,w) \in E \rrbracket \implies \text{path } E \ u \ [u] \ v$
<proof>

lemma *path-edge-cases*[*case-names no-use split*]:

assumes $\text{path } (\text{insert } (u,v) \ E) \ w \ p \ x$
obtains
 $\text{path } E \ w \ p \ x$
 $| \ p1 \ p2$ **where** $\text{path } E \ w \ p1 \ u \ \text{path } (\text{insert } (u,v) \ E) \ v \ p2 \ x$
<proof>

lemma *path-edge-rev-cases*[*case-names no-use split*]:

assumes $\text{path } (\text{insert } (u,v) \ E) \ w \ p \ x$
obtains
 $\text{path } E \ w \ p \ x$
 $| \ p1 \ p2$ **where** $\text{path } (\text{insert } (u,v) \ E) \ w \ p1 \ u \ \text{path } E \ v \ p2 \ x$
<proof>

lemma *path-mono*:

assumes $S: E \subseteq E'$
assumes $P: \text{path } E \ u \ p \ v$
shows $\text{path } E' \ u \ p \ v$
<proof>

lemma *path-is-rtrancl*:

assumes $\text{path } E \ u \ l \ v$
shows $(u,v) \in E^*$

<proof>

lemma *rtrancl-is-path*:

assumes $(u,v) \in E^*$

obtains l **where** $\text{path } E \ u \ l \ v$

<proof>

lemma *path-is-trancl*:

assumes $\text{path } E \ u \ l \ v$

and $l \neq []$

shows $(u,v) \in E^+$

<proof>

lemma *trancl-is-path*:

assumes $(u,v) \in E^+$

obtains l **where** $l \neq []$ **and** $\text{path } E \ u \ l \ v$

<proof>

lemma *path-nth-conv*: $\text{path } E \ u \ p \ v \longleftrightarrow (\text{let } p' = p@[v] \text{ in}$

$u = p'^!0 \wedge$

$(\forall i < \text{length } p' - 1. (p'^!i, p'^!Suc \ i) \in E))$

<proof>

lemma *path-mapI*:

assumes $\text{path } E \ u \ p \ v$

shows $\text{path } (\text{pairsel } f \ ' \ E) \ (f \ u) \ (\text{map } f \ p) \ (f \ v)$

<proof>

lemma *path-restrict*:

assumes $\text{path } E \ u \ p \ v$

shows $\text{path } (E \cap \text{set } p \times \text{insert } v \ (\text{set } (\text{tl } p))) \ u \ p \ v$

<proof>

lemma *path-restrict-closed*:

assumes *CLOSED*: $E \ ' \ D \subseteq D$

assumes *I*: $v \in D$ **and** *P*: $\text{path } E \ v \ p \ v'$

shows $\text{path } (E \cap D \times D) \ v \ p \ v'$

<proof>

lemma *path-set-induct*:

assumes $\text{path } E \ u \ p \ v$ **and** $u \in I$ **and** $E \ ' \ I \subseteq I$

shows $\text{set } p \subseteq I$

<proof>

lemma *path-nodes-reachable*: $\text{path } E \ u \ p \ v \implies \text{insert } v \ (\text{set } p) \subseteq E^* \ ' \ \{u\}$

<proof>

lemma *path-nodes-edges*: $\text{path } E \ u \ p \ v \implies \text{set } p \subseteq \text{fst}' E$

$\langle proof \rangle$

lemma *path-tl-nodes-edges*:

assumes $path\ E\ u\ p\ v$

shows $set\ (tl\ p) \subseteq fst'E \cap snd'E$

$\langle proof \rangle$

lemma *path-loop-shift*:

assumes $P: path\ E\ u\ p\ u$

assumes $S: v \in set\ p$

obtains p' **where** $set\ p' = set\ p \quad path\ E\ v\ p'\ v$

$\langle proof \rangle$

lemma *path-hd*:

assumes $p \neq [] \quad path\ E\ v\ p\ w$

shows $hd\ p = v$

$\langle proof \rangle$

lemma *path-last-is-edge*:

assumes $path\ E\ x\ p\ y$

and $p \neq []$

shows $(last\ p, y) \in E$

$\langle proof \rangle$

lemma *path-member-reach-end*:

assumes $P: path\ E\ x\ p\ y$

and $v: v \in set\ p$

shows $(v, y) \in E^+$

$\langle proof \rangle$

lemma *path-tl-induct*[consumes 2, case-names single step]:

assumes $P: path\ E\ x\ p\ y$

and $NE: x \neq y$

and $S: \bigwedge u. (x, u) \in E \implies P\ x\ u$

and $ST: \bigwedge u\ v. [(x, u) \in E^+; (u, v) \in E; P\ x\ u] \implies P\ x\ v$

shows $P\ x\ y \wedge (\forall v \in set\ (tl\ p). P\ x\ v)$

$\langle proof \rangle$

lemma *path-restrict-tl*:

$[[\ u \notin R; path\ (E \cap UNIV \times -R)\ u\ p\ v\] \implies path\ (rel-restrict\ E\ R)\ u\ p\ v$

$\langle proof \rangle$

lemma *path1-restr-conv*: $path\ (E \cap UNIV \times -R)\ u\ (x \# xs)\ v$

$\longleftrightarrow (\exists w. w \notin R \wedge x = u \wedge (u, w) \in E \wedge path\ (rel-restrict\ E\ R)\ w\ xs\ v)$

$\langle proof \rangle$

lemma *dropWhileNot-path*:
assumes $p \neq []$
and $\text{path } E \ w \ p \ x$
and $v \in \text{set } p$
and $\text{dropWhile } ((\neq) \ v) \ p = c$
shows $\text{path } E \ v \ c \ x$
 $\langle \text{proof} \rangle$

lemma *takeWhileNot-path*:
assumes $p \neq []$
and $\text{path } E \ w \ p \ x$
and $v \in \text{set } p$
and $\text{takeWhile } ((\neq) \ v) \ p = c$
shows $\text{path } E \ w \ c \ v$
 $\langle \text{proof} \rangle$

1.2 Infinite Paths

definition $\text{ipath} :: 'q \ \text{digraph} \Rightarrow 'q \ \text{word} \Rightarrow \text{bool}$
— Predicate for an infinite path in a digraph
where $\text{ipath } E \ r \equiv \forall i. (r \ i, r \ (\text{Suc } i)) \in E$

lemma *ipath-conc-conv*:
 $\text{ipath } E \ (u \ \frown \ v) \longleftrightarrow (\exists a. \text{path } E \ a \ u \ (v \ 0) \ \wedge \ \text{ipath } E \ v)$
 $\langle \text{proof} \rangle$

lemma *ipath-iter-conv*:
assumes $p \neq []$
shows $\text{ipath } E \ (p^\omega) \longleftrightarrow (\text{path } E \ (\text{hd } p) \ p \ (\text{hd } p))$
 $\langle \text{proof} \rangle$

lemma *ipath-to-rtrancl*:
assumes $R: \text{ipath } E \ r$
assumes $I: i1 \leq i2$
shows $(r \ i1, r \ i2) \in E^*$
 $\langle \text{proof} \rangle$

lemma *ipath-to-trancl*:
assumes $R: \text{ipath } E \ r$
assumes $I: i1 < i2$
shows $(r \ i1, r \ i2) \in E^+$
 $\langle \text{proof} \rangle$

lemma *run-limit-two-connectedI*:
assumes $A: \text{ipath } E \ r$
assumes $B: a \in \text{limit } r \quad b \in \text{limit } r$
shows $(a, b) \in E^+$

$\langle proof \rangle$

lemma *ipath-subpath*:

assumes P : $ipath\ E\ r$

assumes LE : $l \leq u$

shows $path\ E\ (r\ l)\ (map\ r\ [l..<u])\ (r\ u)$

$\langle proof \rangle$

lemma *ipath-restrict-eq*: $ipath\ (E \cap (E^* \{r\ 0\} \times E^* \{r\ 0\}))\ r \longleftrightarrow ipath\ E\ r$

$\langle proof \rangle$

lemma *ipath-restrict*: $ipath\ E\ r \implies ipath\ (E \cap (E^* \{r\ 0\} \times E^* \{r\ 0\}))\ r$

$\langle proof \rangle$

lemma *ipathI[intro?]*: $[[\bigwedge i. (r\ i, r\ (Suc\ i)) \in E]] \implies ipath\ E\ r$

$\langle proof \rangle$

lemma *ipathD*: $ipath\ E\ r \implies (r\ i, r\ (Suc\ i)) \in E$

$\langle proof \rangle$

lemma *ipath-in-Domain*: $ipath\ E\ r \implies r\ i \in Domain\ E$

$\langle proof \rangle$

lemma *ipath-in-Range*: $[ipath\ E\ r; i \neq 0] \implies r\ i \in Range\ E$

$\langle proof \rangle$

lemma *ipath-suffix*: $ipath\ E\ r \implies ipath\ E\ (suffix\ i\ r)$

$\langle proof \rangle$

1.3 Strongly Connected Components

A strongly connected component is a maximal mutually connected set of nodes

definition *is-scc* :: $'q\ digraph \Rightarrow 'q\ set \Rightarrow bool$

where $is-scc\ E\ U \longleftrightarrow U \times U \subseteq E^* \wedge (\forall V. V \supset U \longrightarrow \neg (V \times V \subseteq E^*))$

lemma *scc-non-empty[simp]*: $\neg is-scc\ E\ \{\}$ $\langle proof \rangle$

lemma *scc-non-empty'[simp]*: $is-scc\ E\ U \implies U \neq \{\}$ $\langle proof \rangle$

lemma *is-scc-closed*:

assumes SCC : $is-scc\ E\ U$

assumes MEM : $x \in U$

assumes P : $(x, y) \in E^* \quad (y, x) \in E^*$

shows $y \in U$

$\langle proof \rangle$

lemma *is-scc-connected*:

assumes *SCC*: *is-scc E U*
assumes *MEM*: $x \in U \quad y \in U$
shows $(x, y) \in E^*$
 <proof>

In the following, we play around with alternative characterizations, and prove them all equivalent .

A common characterization is to define an equivalence relation „mutually connected” on nodes, and characterize the SCCs as its equivalence classes:

definition *mconn* :: $('a \times 'a)$ set $\Rightarrow ('a \times 'a)$ set
 — Mutually connected relation on nodes
where $mconn\ E = E^* \cap (E^{-1})^*$

lemma *mconn-pointwise*:
 $mconn\ E = \{(u, v). (u, v) \in E^* \wedge (v, u) \in E^*\}$
 <proof>

mconn is an equivalence relation:

lemma *mconn-refl[simp]*: $Id \subseteq mconn\ E$
 <proof>

lemma *mconn-sym*: $mconn\ E = (mconn\ E)^{-1}$
 <proof>

lemma *mconn-trans*: $mconn\ E\ O\ mconn\ E = mconn\ E$
 <proof>

lemma *mconn-refl'*: $refl\ (mconn\ E)$
 <proof>

lemma *mconn-sym'*: $sym\ (mconn\ E)$
 <proof>

lemma *mconn-trans'*: $trans\ (mconn\ E)$
 <proof>

lemma *mconn-equiv*: $equiv\ UNIV\ (mconn\ E)$
 <proof>

lemma *is-scc-mconn-eqclasses*: $is-scc\ E\ U \iff U \in UNIV // mconn\ E$
 — The strongly connected components are the equivalence classes of the mutually-connected relation on nodes
 <proof>

lemma *is-scc E U* $\iff U \in UNIV // (E^* \cap (E^{-1})^*)$
 <proof>

We can also restrict the notion of "reachability" to nodes inside the SCC

lemma *find-outside-node*:

assumes $(u,v) \in E^*$
assumes $(u,v) \notin (E \cap U \times U)^*$
assumes $u \in U \quad v \in U$
shows $\exists u'. u' \notin U \wedge (u,u') \in E^* \wedge (u',v) \in E^*$
 $\langle proof \rangle$

lemma *is-scc-restrict1*:

assumes *SCC*: *is-scc* $E \ U$
shows $U \times U \subseteq (E \cap U \times U)^*$
 $\langle proof \rangle$

lemma *is-scc-restrict2*:

assumes *SCC*: *is-scc* $E \ U$
assumes $V \supset U$
shows $\neg (V \times V \subseteq (E \cap V \times V)^*)$
 $\langle proof \rangle$

lemma *is-scc-restrict3*:

assumes *SCC*: *is-scc* $E \ U$
shows $((E^* \setminus ((E^* \setminus U) - U)) \cap U = \{\})$
 $\langle proof \rangle$

lemma *is-scc-alt-restrict-path*:

is-scc $E \ U \iff U \neq \{\} \wedge$
 $(U \times U \subseteq (E \cap U \times U)^*) \wedge ((E^* \setminus ((E^* \setminus U) - U)) \cap U = \{\})$
 $\langle proof \rangle$

lemma *is-scc-pointwise*:

is-scc $E \ U \iff$
 $U \neq \{\}$
 $\wedge (\forall u \in U. \forall v \in U. (u,v) \in (E \cap U \times U)^*)$
 $\wedge (\forall u \in U. \forall v. (v \notin U \wedge (u,v) \in E^*) \longrightarrow (\forall u' \in U. (v,u') \notin E^*))$
— Alternative, pointwise characterization
 $\langle proof \rangle$

lemma *is-scc-unique*:

assumes *SCC*: *is-scc* $E \ scc \quad is-scc \ E \ scc'$
and $v: v \in scc \quad v \in scc'$
shows $scc = scc'$
 $\langle proof \rangle$

lemma *is-scc-ex1*:

$\exists ! scc. is-scc \ E \ scc \wedge v \in scc$
 $\langle proof \rangle$

lemma *is-scc-ex*:

$\exists scc. is-scc \ E \ scc \wedge v \in scc$

<proof>

lemma *is-scc-connected'*:

$\llbracket \text{is-scc } E \text{ scc}; x \in \text{scc}; y \in \text{scc} \rrbracket \implies (x,y) \in (\text{Restr } E \text{ scc})^*$

<proof>

definition *scc-of* :: ('v × 'v) set ⇒ 'v ⇒ 'v set

where

$\text{scc-of } E \ v = (\text{THE } \text{scc}. \text{is-scc } E \ \text{scc} \wedge v \in \text{scc})$

lemma *scc-of-is-scc[simp]*:

$\text{is-scc } E \ (\text{scc-of } E \ v)$

<proof>

lemma *node-in-scc-of-node[simp]*:

$v \in \text{scc-of } E \ v$

<proof>

lemma *scc-of-unique*:

assumes $w \in \text{scc-of } E \ v$

shows $\text{scc-of } E \ v = \text{scc-of } E \ w$

<proof>

end

2 Directed Graphs

theory *Digraph*

imports

CAVA-Base.CAVA-Base

Digraph-Basic

begin

2.1 Directed Graphs with Explicit Node Set and Set of Initial Nodes

record 'v *graph-rec* =

g-V :: 'v set

g-E :: 'v digraph

g-V0 :: 'v set

definition *graph-restrict* :: ('v, 'more) *graph-rec-scheme* ⇒ 'v set ⇒ ('v, 'more) *graph-rec-scheme*

where *graph-restrict* *G R* ≡

(

g-V = *g-V G*,

g-E = *rel-restrict (g-E G) R*,

g-V0 = *g-V0 G - R*,

... = *graph-rec.more G*

)

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

$g-V$ (*graph-restrict* G R) = $g-V$ G

$g-E$ (*graph-restrict* G R) = *rel-restrict* ($g-E$ G) R

$g-V0$ (*graph-restrict* G R) = $g-V0$ $G - R$

graph-rec.more (*graph-restrict* G R) = *graph-rec.more* G

<proof>

lemma *graph-restrict-trivial*[simp]: *graph-restrict* G {} = G *<proof>*

locale *graph-defs* =

fixes G :: (' v , ' $more$) *graph-rec-scheme*

begin

abbreviation $V \equiv g-V$ G

abbreviation $E \equiv g-E$ G

abbreviation $V0 \equiv g-V0$ G

abbreviation *reachable* $\equiv E^*$ “ $V0$

abbreviation *succ* $v \equiv E$ “ { v }

lemma *finite-V0*: *finite reachable* \implies *finite V0* *<proof>*

definition *is-run*

— Infinite run, i.e., a rooted infinite path

where *is-run* $r \equiv r$ $0 \in V0 \wedge ipath$ E r

lemma *run-ipath*: *is-run* $r \implies ipath$ E r *<proof>*

lemma *run-V0*: *is-run* $r \implies r$ $0 \in V0$ *<proof>*

lemma *run-reachable*: *is-run* $r \implies range$ $r \subseteq reachable$
<proof>

end

locale *graph* =

graph-defs G

for G :: (' v , ' $more$) *graph-rec-scheme*

+

assumes *V0-ss*: $V0 \subseteq V$

assumes *E-ss*: $E \subseteq V \times V$

begin

lemma *reachable-V*: *reachable* $\subseteq V$ *<proof>*

lemma *finite-E*: *finite V* \implies *finite E* *<proof>*

end

```

locale fb-graph =
  graph G
  for G :: ('v, 'more) graph-rec-scheme
  +
  assumes finite-V0[simp, intro!]: finite V0
  assumes finitely-branching[simp, intro]:  $v \in \text{reachable} \implies \text{finite } (\text{succ } v)$ 
begin

  lemma fb-graph-subset:
    assumes g-V  $G' = V$ 
    assumes g-E  $G' \subseteq E$ 
    assumes finite (g-V0  $G'$ )
    assumes g-V0  $G' \subseteq \text{reachable}$ 
    shows fb-graph  $G'$ 
  <proof>

  lemma fb-graph-restrict: fb-graph (graph-restrict G R)
  <proof>

end

lemma (in graph) fb-graphI-fr:
  assumes finite reachable
  shows fb-graph G
  <proof>

abbreviation rename-E  $f E \equiv (\lambda(u,v). (f\ u, f\ v))\ 'E$ 

definition fr-rename-ext ecnv  $f G \equiv ($ 
  g-V =  $f'(g-V\ G)$ ,
  g-E = rename-E  $f (g-E\ G)$ ,
  g-V0 =  $(f'g-V0\ G)$ ,
  ... = ecnv G
   $)$ 

locale g-rename-precond =
  graph G
  for G :: ('u, 'more) graph-rec-scheme
  +
  fixes f :: 'u  $\Rightarrow$  'v
  fixes ecnv :: ('u, 'more) graph-rec-scheme  $\Rightarrow$  'more'
  assumes INJ: inj-on f V
begin

  abbreviation  $G' \equiv \text{fr-rename-ext } \text{ecnv } f\ G$ 

  lemma  $G'$ -fields:

```

$g-V\ G' = f \cdot V$
 $g-V0\ G' = f \cdot V0$
 $g-E\ G' = \text{rename-}E\ f\ E$
 <proof>

definition $fi \equiv \text{the-inv-into } V\ f$

lemma

$fi-f: x \in V \implies fi\ (f\ x) = x$ **and**
 $f-fi: y \in f \cdot V \implies f\ (fi\ y) = y$ **and**
 $fi-f\text{-eq}: \llbracket f\ x = y; x \in V \rrbracket \implies fi\ y = x$
 <proof>

lemma $E'-to-E: (u,v) \in g-E\ G' \implies (fi\ u, fi\ v) \in E$
 <proof>

lemma $V0'-to-V0: v \in g-V0\ G' \implies fi\ v \in V0$
 <proof>

lemma $rtrancl-E'-sim:$

assumes $(f\ u, v') \in (g-E\ G')^*$
assumes $u \in V$
shows $\exists v. v' = f\ v \wedge v \in V \wedge (u, v) \in E^*$
 <proof>

lemma $rtrancl-E'-to-E: \text{assumes } (u,v) \in (g-E\ G')^* \text{ shows } (fi\ u, fi\ v) \in E^*$
 <proof>

lemma $G'-invar: \text{graph } G'$
 <proof>

sublocale $G': \text{graph } G'$ <proof>

lemma $G'-finite-reachable:$

assumes $finite\ ((g-E\ G')^* \text{ `` } g-V0\ G)$
shows $finite\ ((g-E\ G')^* \text{ `` } g-V0\ G')$
 <proof>

lemma $V'-to-V: v \in G'.V \implies fi\ v \in V$
 <proof>

lemma $ipath-sim1: ipath\ E\ r \implies ipath\ G'.E\ (f\ o\ r)$
 <proof>

lemma $ipath-sim2: ipath\ G'.E\ r \implies ipath\ E\ (fi\ o\ r)$
 <proof>

lemma $run-sim1: is-run\ r \implies G'.is-run\ (f\ o\ r)$

<proof>

lemma *run-sim2*: $G'.is-run\ r \implies is-run\ (fi\ o\ r)$
<proof>

end

end

3 Automata

theory *Automata*
imports *Digraph*
begin

In this theory, we define Generalized Buchi Automata and Buchi Automata based on directed graphs

hide-const (**open**) *prod*

3.1 Generalized Buchi Graphs

A generalized Buchi graph is a graph where each node belongs to a set of acceptance classes. An infinite run on this graph is accepted, iff it visits nodes from each acceptance class infinitely often.

The standard encoding of acceptance classes is as a set of sets of nodes, each inner set representing one acceptance class.

record *'Q gb-graph-rec* = *'Q graph-rec* +
gbg-F :: *'Q set set*

locale *gb-graph* =
graph G
for $G :: ('Q, 'more)\ gb-graph-rec-scheme +$
assumes *finite-F[simp, intro!]*: *finite (gbg-F G)*
assumes *F-ss*: $gbg-F\ G \subseteq Pow\ V$
begin
abbreviation $F \equiv gbg-F\ G$

lemma *is-gb-graph*: *gb-graph G* *<proof>*

definition

is-acc :: *'Q word* $\Rightarrow bool$ **where** *is-acc r* $\equiv (\forall A \in F. \exists_{\infty} i. r\ i \in A)$

definition *is-acc-run r* $\equiv is-run\ r \wedge is-acc\ r$

lemma *is-acc-run* $r \equiv is-run\ r \wedge (\forall A \in F. \exists_{\infty} i. r\ i \in A)$
 ⟨proof⟩

lemma *acc-run-run*: $is-acc-run\ r \implies is-run\ r$
 ⟨proof⟩

lemmas *acc-run-reachable* = *run-reachable*[*OF acc-run-run*]

lemma *acc-eq-limit*:
assumes *FIN*: *finite* (*range* r)
shows $is-acc\ r \longleftrightarrow (\forall A \in F. limit\ r \cap A \neq \{\})$
 ⟨proof⟩

lemma *is-acc-run-limit-alt*:
assumes *finite* (E^* “ $V0$)
shows $is-acc-run\ r \longleftrightarrow is-run\ r \wedge (\forall A \in F. limit\ r \cap A \neq \{\})$
 ⟨proof⟩

lemma *is-acc-suffix*[*simp*]: $is-acc\ (suffix\ i\ r) \longleftrightarrow is-acc\ r$
 ⟨proof⟩

lemma *finite-V-Fe*:
assumes *finite* $V \quad A \in F$
shows *finite* A
 ⟨proof⟩

end

definition *gb-rename-ecnv* $ecnv\ f\ G \equiv \langle$
 $gbg-F = \{ f'A \mid A. A \in gbg-F\ G \}, \dots = ecnv\ G$
 \rangle

abbreviation *gb-rename-ext* $ecnv\ f \equiv fr-rename-ext\ (gb-rename-ecnv\ ecnv\ f)\ f$

locale *gb-rename-precond* =
gb-graph G +
g-rename-precond $G\ f\ gb-rename-ecnv\ ecnv\ f$
for $G :: ('u, 'more)\ gb-graph-rec-scheme$
and $f :: 'u \Rightarrow 'v$ **and** $ecnv$
begin
lemma $G'-gb-fields$: $gbg-F\ G' = \{ f'A \mid A. A \in F \}$
 ⟨proof⟩

sublocale G' : *gb-graph* G'

$\langle proof \rangle$

lemma *acc-sim1*: $is-acc\ r \implies G'.is-acc\ (f\ o\ r)$
 $\langle proof \rangle$

lemma *acc-sim2*:
assumes $G'.is-acc\ r$ **shows** $is-acc\ (fi\ o\ r)$
 $\langle proof \rangle$

lemma *acc-run-sim1*: $is-acc-run\ r \implies G'.is-acc-run\ (f\ o\ r)$
 $\langle proof \rangle$

lemma *acc-run-sim2*: $G'.is-acc-run\ r \implies is-acc-run\ (fi\ o\ r)$
 $\langle proof \rangle$

end

3.2 Generalized Buchi Automata

A GBA is obtained from a GBG by adding a labeling function, that associates each state with a set of labels. A word is accepted if there is an accepting run that can be labeled with this word.

record $(\prime Q, \prime L)$ *gba-rec* = $\prime Q$ *gb-graph-rec* +
gba-L :: $\prime Q \Rightarrow \prime L \Rightarrow bool$

locale *gba* =
gb-graph G
for G :: $(\prime Q, \prime L, \prime more)$ *gba-rec-scheme* +
assumes L -*ss*: $gba-L\ G\ q\ l \implies q \in V$

begin

abbreviation $L \equiv gba-L\ G$

lemma *is-gba*: $gba\ G\ \langle proof \rangle$

definition *accept* $w \equiv \exists r. is-acc-run\ r \wedge (\forall i. L\ (r\ i)\ (w\ i))$

lemma *acceptI*[*intro?*]: $\llbracket is-acc-run\ r; \bigwedge i. L\ (r\ i)\ (w\ i) \rrbracket \implies accept\ w$
 $\langle proof \rangle$

definition *lang* $\equiv Collect\ (accept)$

lemma *langI*[*intro?*]: $accept\ w \implies w \in lang\ \langle proof \rangle$

end

definition *gba-rename-ecnv* $ecnv\ f\ G \equiv (\$
 $gba-L = \lambda q\ l.$
 $\text{if } q \in f'g-V\ G \text{ then}$
 $\quad gba-L\ G\ (the-inv-into\ (g-V\ G)\ f\ q)\ l$
 else
 $\quad False,$
 $\dots = ecnv\ G$

)

abbreviation *gba-rename-ext ecnv f* \equiv *gb-rename-ext (gba-rename-ecnv ecnv f) f*

locale *gba-rename-precond* =

gb-rename-precond G f gba-rename-ecnv ecnv f + gba G

for *G* :: ('u,'L,'more) *gba-rec-scheme*

and *f* :: 'u \Rightarrow 'v **and** *ecnv*

begin

lemma *G'-gba-fields*: *gba-L G' = (λq l.*

if $q \in f^{\cdot}V$ then $L (fi q) l$ else False)

<proof>

sublocale *G'*: *gba G'*

<proof>

lemma *L-sim1*: $\llbracket \text{range } r \subseteq V; L (r i) l \rrbracket \Longrightarrow G'.L (f (r i)) l$

<proof>

lemma *L-sim2*: $\llbracket \text{range } r \subseteq f^{\cdot}V; G'.L (r i) l \rrbracket \Longrightarrow L (fi (r i)) l$

<proof>

lemma *accept-eq[simp]*: *G'.accept = accept*

<proof>

lemma *lang-eq[simp]*: *G'.lang = lang*

<proof>

lemma *finite-G'-V*:

assumes *finite V*

shows *finite G'.V*

<proof>

end

abbreviation *gba-rename* \equiv *gba-rename-ext ($\lambda.$ ())*

lemma *gba-rename-correct*:

fixes *G* :: ('v,'l,'m) *gba-rec-scheme*

assumes *gba G*

assumes *INJ*: *inj-on f (g-V G)*

defines *G' \equiv gba-rename f G*

shows *gba G'*

and *finite (g-V G) \Longrightarrow finite (g-V G')*

and *gba.accept G' = gba.accept G*

and *gba.lang G' = gba.lang G*

<proof>

3.3 Buchi Graphs

A Buchi graph has exactly one acceptance class

record $'Q$ *b-graph-rec* = $'Q$ *graph-rec* +
bg-F :: $'Q$ *set*

locale *b-graph* =
graph *G*
for G :: ($'Q$, $'more$) *b-graph-rec-scheme*
+
assumes F -ss: $bg-F\ G \subseteq V$
begin
abbreviation F **where** $F \equiv bg-F\ G$

lemma *is-b-graph*: *b-graph* G $\langle proof \rangle$

definition *to-gbg-ext* m
 \equiv (\mid $g-V = V$,
 $g-E = E$,
 $g-V0 = V0$,
 $gbg-F =$ *if* $F = UNIV$ *then* $\{\}$ *else* $\{F\}$,
 $\dots = m$ \mid)

abbreviation *to-gbg* $\equiv to-gbg-ext$ $()$

sublocale *gbg*: *gb-graph* *to-gbg-ext* m
 $\langle proof \rangle$

definition *is-acc* :: $'Q$ *word* \Rightarrow *bool* **where** *is-acc* $r \equiv (\exists_{\infty} i. r\ i \in F)$

definition *is-acc-run* **where** *is-acc-run* $r \equiv is-run\ r \wedge is-acc\ r$

lemma *to-gbg-alt*:
 $gbg.V\ T\ m = V$
 $gbg.E\ T\ m = E$
 $gbg.V0\ T\ m = V0$
 $gbg.F\ T\ m =$ (*if* $F = UNIV$ *then* $\{\}$ *else* $\{F\}$)
 $gbg.is-run\ T\ m = is-run$
 $gbg.is-acc\ T\ m = is-acc$
 $gbg.is-acc-run\ T\ m = is-acc-run$
 $\langle proof \rangle$

end

3.4 Buchi Automata

Buchi automata are labeled Buchi graphs

record ($'Q$, $'L$) *ba-rec* = $'Q$ *b-graph-rec* +
ba-L :: $'Q \Rightarrow 'L \Rightarrow bool$

locale *ba* =
bg?: *b-graph* *G*
for *G* :: ('*Q*, '*L*, '*more*) *ba-rec-scheme*
+
assumes *L-ss*: *ba-L* *G* *q l* \implies *q* \in *V*
begin
abbreviation *L* **where** *L* \equiv *ba-L* *G*

lemma *is-ba*: *ba* *G* *<proof>*

abbreviation *to-gba-ext* *m* \equiv *to-gbg-ext* (*gba-L* = *L*, ... = *m*)
abbreviation *to-gba* \equiv *to-gba-ext* ()

sublocale *gba*: *gba* *to-gba-ext* *m*
<proof>

lemma *ba-acc-simps*[*simp*]: *gba.L* *T* *m* = *L*
<proof>

definition *accept* *w* \equiv (\exists *r*. *is-acc-run* *r* \wedge (\forall *i*. *L* (*r* *i*) (*w* *i*)))
definition *lang* \equiv *Collect* *accept*

lemma *to-gba-alt-accept*:
gba.accept *T* *m* = *accept*
<proof>

lemma *to-gba-alt-lang*:
gba.lang *T* *m* = *lang*
<proof>

lemmas *to-gba-alt* = *to-gbg-alt* *to-gba-alt-accept* *to-gba-alt-lang*
end

3.5 Indexed acceptance classes

record '*Q* *igb-graph-rec* = '*Q* *graph-rec* +
igbg-num-acc :: *nat*
igbg-acc :: '*Q* \Rightarrow *nat set*

locale *igb-graph* =
graph *G*
for *G* :: ('*Q*, '*more*) *igb-graph-rec-scheme*
+
assumes *acc-bound*: \bigcup (*range* (*igbg-acc* *G*)) \subseteq {0..*(igbg-num-acc* *G*)}
assumes *acc-ss*: *igbg-acc* *G* *q* \neq {} \implies *q* \in *V*
begin
abbreviation *num-acc* **where** *num-acc* \equiv *igbg-num-acc* *G*
abbreviation *acc* **where** *acc* \equiv *igbg-acc* *G*

lemma *is-igb-graph*: *igb-graph* G \langle *proof* \rangle

lemma *acc-boundI*[*simp, intro*]: $x \in \text{acc } q \implies x < \text{num-acc}$
 \langle *proof* \rangle

definition *accn* $i \equiv \{q . i \in \text{acc } q\}$

definition $F \equiv \{ \text{accn } i \mid i. i < \text{num-acc} \}$

definition *to-gbg-ext* m
 $\equiv (\lambda g. V = V, g.E = E, g.V0 = V0, \text{gbg}.F = F, \dots = m)$

sublocale *gbg*: *gb-graph to-gbg-ext* m
 \langle *proof* \rangle

lemma *to-gbg-alt1*:

$\text{gbg}.E \ T \ m = E$

$\text{gbg}.V0 \ T \ m = V0$

$\text{gbg}.F \ T \ m = F$

\langle *proof* \rangle

lemma *F-fin*[*simp, intro!*]: *finite* F
 \langle *proof* \rangle

definition *is-acc* :: $'Q \ \text{word} \Rightarrow \text{bool}$

where *is-acc* $r \equiv (\forall n < \text{num-acc}. \exists_{\infty} i. n \in \text{acc } (r \ i))$

definition *is-acc-run* $r \equiv \text{is-run } r \wedge \text{is-acc } r$

lemma *is-run-gbg*:

$\text{gbg}.is\text{-run} \ T \ m = \text{is-run}$

\langle *proof* \rangle

lemma *is-acc-gbg*:

$\text{gbg}.is\text{-acc} \ T \ m = \text{is-acc}$

\langle *proof* \rangle

lemma *is-acc-run-gbg*:

$\text{gbg}.is\text{-acc-run} \ T \ m = \text{is-acc-run}$

\langle *proof* \rangle

lemmas *to-gbg-alt = to-gbg-alt1 is-run-gbg is-acc-gbg is-acc-run-gbg*

lemma *acc-limit-alt*:

assumes *FIN*: *finite* (*range* r)

shows *is-acc* $r \longleftrightarrow (\forall n < \text{num-acc}. \text{limit } r \cap \text{accn } n \neq \{\})$

\langle *proof* \rangle

lemma *acc-limit-alt'*:

finite (*range* *r*) \implies *is-acc* *r* \iff (\bigcup (*acc* ' *limit* *r*) = {0..*num-acc*})
 ⟨*proof*⟩

end

record ('*Q*, '*L*) *igba-rec* = '*Q* *igb-graph-rec* +
igba-L :: '*Q* \Rightarrow '*L* \Rightarrow *bool*

locale *igba* =
igbg?: *igb-graph* *G*
for *G* :: ('*Q*, '*L*, '*more*) *igba-rec-scheme*
 +
assumes *L-ss*: *igba-L* *G* *q* *l* \implies *q* \in *V*
begin
abbreviation *L* **where** *L* \equiv *igba-L* *G*

lemma *is-igba*: *igba* *G* ⟨*proof*⟩

abbreviation *to-gba-ext* *m* \equiv *to-gbg-ext* (\lfloor *gba-L* = *igba-L* *G*, ...=*m* \rfloor)

sublocale *gba*: *gba* *to-gba-ext* *m*
 ⟨*proof*⟩

lemma *to-gba-alt-L*:
gba.L *T* *m* = *L*
 ⟨*proof*⟩

definition *accept* *w* \equiv $\exists r$. *is-acc-run* *r* \wedge ($\forall i$. *L* (*r* *i*) (*w* *i*))

definition *lang* \equiv *Collect* *accept*

lemma *accept-gba-alt*: *gba.accept* *T* *m* = *accept*
 ⟨*proof*⟩

lemma *lang-gba-alt*: *gba.lang* *T* *m* = *lang*
 ⟨*proof*⟩

lemmas *to-gba-alt* = *to-gbg-alt* *to-gba-alt-L* *accept-gba-alt* *lang-gba-alt*

end

3.5.1 Indexing Conversion

definition *F-to-idx* :: '*Q* *set* *set* \Rightarrow (*nat* \times ('*Q* \Rightarrow *nat* *set*)) *nres* **where**

F-to-idx *F* \equiv *do* {
Flist \leftarrow *SPEC* (λ *Flist*. *distinct* *Flist* \wedge *set* *Flist* = *F*);
let *num-acc* = *length* *Flist*;
let *acc* = (λv . {*i* . *i* < *num-acc* \wedge *v* \in *Flist*!*i*});
RETURN (*num-acc*, *acc*)

}

lemma *F-to-idx-correct*:

shows $F\text{-to-idx } F \leq \text{SPEC } (\lambda(\text{num-acc}, \text{acc}). F = \{ \{q. i \in \text{acc } q\} \mid i. i < \text{num-acc} \})$
 $\wedge \bigcup (\text{range } \text{acc}) \subseteq \{0..<\text{num-acc}\}$
 <proof>

definition *mk-acc-impl Flist* \equiv *do* {

let *acc* = *Map.empty*;

(-, *acc*) \leftarrow *nfoldli Flist* ($\lambda\cdot$. *True*) (λA (*i*, *acc*). *do* {
acc \leftarrow *FOREACHi* (λit *acc'*.
acc' = (λv .
 if $v \in A$ - *it* then
 Some (*insert i* (*the-default* {} (*acc v*)))
 else
 acc v
)
)
)
A (λv *acc*. *RETURN* (*acc*($v \mapsto$ *insert i* (*the-default* {} (*acc v*)))))) *acc*;
RETURN (*Suc i*, *acc*)
 }) (*0*, *acc*);
RETURN (λx . *the-default* {} (*acc x*))
 }

lemma *mk-acc-impl-correct*:

assumes *F*: (*Flist'*, *Flist*) \in *Id*
assumes *FIN*: $\forall A \in \text{set } Flist. \text{finite } A$
shows $\text{mk-acc-impl } Flist' \leq \Downarrow \text{Id } (\text{RETURN } (\lambda v. \{i. i < \text{length } Flist \wedge v \in Flist!i\}))$
 <proof>

definition *F-to-idx-impl* $:: 'Q \text{ set set} \Rightarrow (\text{nat} \times ('Q \Rightarrow \text{nat set})) \text{ nres}$ **where**

F-to-idx-impl F \equiv *do* {
Flist \leftarrow *SPEC* ($\lambda Flist. \text{distinct } Flist \wedge \text{set } Flist = F$);
let *num-acc* = *length Flist*;
acc \leftarrow *mk-acc-impl Flist*;
RETURN (*num-acc*, *acc*)
 }

lemma *F-to-idx-refine*:

assumes *FIN*: $\forall A \in F. \text{finite } A$
shows $F\text{-to-idx-impl } F \leq \Downarrow \text{Id } (F\text{-to-idx } F)$
 <proof>

definition *gbg-to-idx-ext*

$:: - \Rightarrow ('a, 'more) \text{ gb-graph-rec-scheme} \Rightarrow ('a, 'more) \text{ igb-graph-rec-scheme nres}$
where *gbg-to-idx-ext ecnv A* = *do* {
 (*num-acc*, *acc*) \leftarrow *F-to-idx-impl* (*gbg-F A*);


```

RETURN (
  g-V = g-V A,
  g-E = g-E A,
  g-V0 = g-V0 A,
  igbg-num-acc = num-acc,
  igbg-acc = acc,
  ... = ecnv A
)
}

```

lemma (in *gb-graph*) *gbg-to-idx-ext-correct*:
assumes [*simp*, *intro*]: $\bigwedge A. A \in F \implies \text{finite } A$
shows *gbg-to-idx-ext ecnv G* \leq *SPEC* ($\lambda G'.$
igb-graph.is-acc-run G' = is-acc-run
 \wedge *g-V G' = V*
 \wedge *g-E G' = E*
 \wedge *g-V0 G' = V0*
 \wedge *igb-graph-rec.more G' = ecnv G*
 \wedge *igb-graph G'*
)
<proof>

abbreviation *gbg-to-idx* :: (*'q,-*) *gb-graph-rec-scheme* \Rightarrow *'q igb-graph-rec nres*
where *gbg-to-idx* \equiv *gbg-to-idx-ext* ($\lambda-. ()$)

definition *ti-Lcnev* **where** *ti-Lcnev ecnv A* \equiv (*igba-L = gba-L A, ... = ecnv A*)

abbreviation *gba-to-idx-ext ecnv* \equiv *gbg-to-idx-ext* (*ti-Lcnev ecnv*)

abbreviation *gba-to-idx* \equiv *gba-to-idx-ext* ($\lambda-. ()$)

lemma (in *gba*) *gba-to-idx-ext-correct*:
assumes [*simp*, *intro*]: $\bigwedge A. A \in F \implies \text{finite } A$
shows *gba-to-idx-ext ecnv G* \leq
SPEC ($\lambda G'.$
igba.accept G' = accept
 \wedge *g-V G' = V*
 \wedge *g-E G' = E*
 \wedge *g-V0 G' = V0*
 \wedge *igba-rec.more G' = ecnv G*
 \wedge *igba G'*
)
<proof>

corollary (in *gba*) *gba-to-idx-ext-lang-correct*:

assumes [*simp*, *intro*]: $\bigwedge A. A \in F \implies \text{finite } A$
shows *gba-to-idx-ext ecnv G* \leq
SPEC ($\lambda G'. \text{igba.lang } G' = \text{lang} \wedge \text{igba-rec.more } G' = \text{ecnv } G \wedge \text{igba } G'$)
<proof>

3.5.2 Degeneralization

context *igb-graph*

begin

definition *degeneralize-ext* :: $- \Rightarrow ('Q \times \text{nat}, -)$ *b-graph-rec-scheme* **where**

degeneralize-ext ecnv \equiv
 if *num-acc* = 0 then \langle
 $g\text{-}V = V \times \{0\},$
 $g\text{-}E = \{((q,0),(q',0)) \mid q \ q'. (q,q') \in E\},$
 $g\text{-}V0 = V0 \times \{0\},$
 $bg\text{-}F = V \times \{0\},$
 $\dots = \text{ecnv } G$
 \rangle
 else \langle
 $g\text{-}V = V \times \{0..<\text{num-acc}\},$
 $g\text{-}E = \{((q,i),(q',i')) \mid i \ i' \ q \ q'.$
 $\quad i < \text{num-acc}$
 $\quad \wedge (q,q') \in E$
 $\quad \wedge i' = (\text{if } i \in \text{acc } q \text{ then } (i+1) \text{ mod } \text{num-acc} \text{ else } i) \},$
 $g\text{-}V0 = V0 \times \{0\},$
 $bg\text{-}F = \{(q,0) \mid q. 0 \in \text{acc } q\},$
 $\dots = \text{ecnv } G$
 \rangle

abbreviation *degeneralize* **where** *degeneralize* \equiv *degeneralize-ext* ($\lambda\text{-}.$ ())

lemma *degen-more*[*simp*]: *b-graph-rec.more* (*degeneralize-ext ecnv*) = *ecnv G*
 $\langle \text{proof} \rangle$

lemma *degen-invar*: *b-graph* (*degeneralize-ext ecnv*)
 $\langle \text{proof} \rangle$

sublocale *degen*: *b-graph* *degeneralize-ext m* $\langle \text{proof} \rangle$

lemma *degen-finite-reachable*:

assumes [*simp*, *intro*]: *finite* (E^* “ $V0$)

shows *finite* ($(g\text{-}E \text{ (degeneralize-ext ecnv)})^* \text{ “ } g\text{-}V0 \text{ (degeneralize-ext ecnv)}$)

$\langle \text{proof} \rangle$

lemma *degen-is-run-sound*:

degen.is-run $T \ m \ r \implies \text{is-run } (\text{fst } o \ r)$

$\langle \text{proof} \rangle$

lemma *degen-path-sound*:

assumes *path* (*degen.E T m*) $u \ p \ v$

shows *path* $E \ (\text{fst } u) \ (\text{map } \text{fst } p) \ (\text{fst } v)$

$\langle \text{proof} \rangle$

lemma *degen-V0-sound*:

assumes $u \in \text{degen.V0 } T m$
shows $\text{fst } u \in V0$
 $\langle \text{proof} \rangle$

lemma *degen-visit-acc*:
assumes $\text{path } (\text{degen.E } T m) (q,n) p (q',n')$
assumes $n \neq n'$
shows $\exists qa. (qa,n) \in \text{set } p \wedge n \in \text{acc } qa$
 $\langle \text{proof} \rangle$

lemma *degen-run-complete0*:
assumes $[\text{simp}]: \text{num-acc} = 0$
assumes $R: \text{is-run } r$
shows $\text{degen.is-run } T m (\lambda i. (r i, 0))$
 $\langle \text{proof} \rangle$

lemma *degen-acc-run-complete0*:
assumes $[\text{simp}]: \text{num-acc} = 0$
assumes $R: \text{is-acc-run } r$
shows $\text{degen.is-acc-run } T m (\lambda i. (r i, 0))$
 $\langle \text{proof} \rangle$

lemma *degen-run-complete*:
assumes $[\text{simp}]: \text{num-acc} \neq 0$
assumes $R: \text{is-run } r$
shows $\exists r'. \text{degen.is-run } T m r' \wedge r = \text{fst } o r'$
 $\langle \text{proof} \rangle$

lemma *degen-run-bound*:
assumes $[\text{simp}]: \text{num-acc} \neq 0$
assumes $R: \text{degen.is-run } T m r$
shows $\text{snd } (r i) < \text{num-acc}$
 $\langle \text{proof} \rangle$

lemma *degen-acc-run-complete-aux1*:
assumes $\text{NN0}[\text{simp}]: \text{num-acc} \neq 0$
assumes $R: \text{degen.is-run } T m r$
assumes $\text{EXJ}: \exists j \geq i. n \in \text{acc } (\text{fst } (r j))$
assumes $\text{RI}: r i = (q,n)$
shows $\exists j \geq i. \exists q'. r j = (q',n) \wedge n \in \text{acc } q'$
 $\langle \text{proof} \rangle$

lemma *degen-acc-run-complete-aux1'*:
assumes $\text{NN0}[\text{simp}]: \text{num-acc} \neq 0$
assumes $R: \text{degen.is-run } T m r$
assumes $\text{ACC}: \forall n < \text{num-acc}. \exists \infty i. n \in \text{acc } (\text{fst } (r i))$
assumes $\text{RI}: r i = (q,n)$
shows $\exists j \geq i. \exists q'. r j = (q',n) \wedge n \in \text{acc } q'$

<proof>

lemma *degen-acc-run-complete-aux2*:

assumes *NN0[simp]*: $num-acc \neq 0$

assumes *R*: *degen.is-run* $T\ m\ r$

assumes *ACC*: $\forall n < num-acc. \exists_{\infty} i. n \in acc\ (fst\ (r\ i))$

assumes *RI*: $r\ i = (q, n)$ **and** *OFS*: $ofs < num-acc$

shows $\exists j \geq i. \exists q'$.

$r\ j = (q', (n + ofs) \bmod num-acc) \wedge (n + ofs) \bmod num-acc \in acc\ q'$

<proof>

lemma *degen-acc-run-complete*:

assumes *AR*: *is-acc-run* r

obtains r'

where *degen.is-acc-run* $T\ m\ r'$ **and** $r = fst\ o\ r'$

<proof>

lemma *degen-run-find-change*:

assumes *NN0[simp]*: $num-acc \neq 0$

assumes *R*: *degen.is-run* $T\ m\ r$

assumes *A*: $i \leq j$ $r\ i = (q, n)$ $r\ j = (q', n')$ $n \neq n'$

obtains $k\ qk$ **where** $i \leq k$ $k < j$ $r\ k = (qk, n)$ $n \in acc\ qk$

<proof>

lemma *degen-run-find-acc-aux*:

assumes *NN0[simp]*: $num-acc \neq 0$

assumes *AR*: *degen.is-acc-run* $T\ m\ r$

assumes *A*: $r\ i = (q, 0)$ $0 \in acc\ q$ $n < num-acc$

shows $\exists j\ qj. i \leq j \wedge r\ j = (qj, n) \wedge n \in acc\ qj$

<proof>

lemma *degen-acc-run-sound*:

assumes *A*: *degen.is-acc-run* $T\ m\ r$

shows *is-acc-run* $(fst\ o\ r)$

<proof>

lemma *degen-acc-run-iff*:

$is-acc-run\ r \longleftrightarrow (\exists r'. fst\ o\ r' = r \wedge degen.is-acc-run\ T\ m\ r')$

<proof>

end

3.6 System Automata

System automata are (finite) rooted graphs with a labeling function. They are used to describe the model (system) to be checked.

record $('Q, 'L)$ *sa-rec* = $'Q$ *graph-rec* +

sa-L :: $'Q \Rightarrow 'L$

locale *sa* =
g?: *graph G*
for *G* :: ('*Q*, '*L*, '*more*) *sa-rec-scheme*
begin

abbreviation *L* **where** $L \equiv sa-L\ G$

definition $accept\ w \equiv \exists r. is-run\ r \wedge w = L\ o\ r$

lemma $acceptI[intro?]: \llbracket is-run\ r; w = L\ o\ r \rrbracket \implies accept\ w\ \langle proof \rangle$

definition $lang \equiv Collect\ accept$

lemma $langI[intro?]: accept\ w \implies w \in lang\ \langle proof \rangle$

end

3.6.1 Product Construction

In this section we formalize the product construction between a GBA and a system automaton. The result is a GBG and a projection function, such that projected runs of the GBG correspond to words accepted by the GBA and the system.

locale *igba-sys-prod-precond* = *igba: igba G* + *sa: sa S* **for**
G :: ('*q*, '*l*, '*moreG*) *igba-rec-scheme*
and *S* :: ('*s*, '*l*, '*moreS*) *sa-rec-scheme*
begin

definition $prod \equiv (\mid$
 $g-V = igba.V \times sa.V,$
 $g-E = \{ ((q,s),(q',s')).$
 $igba.L\ q\ (sa.L\ s) \wedge (q,q') \in igba.E \wedge (s,s') \in sa.E \},$
 $g-V0 = igba.V0 \times sa.V0,$
 $igbg-num-acc = igba.num-acc,$
 $igbg-acc = (\lambda(q,s). if\ s \in sa.V\ then\ igba.acc\ q\ else\ \{\}) \mid)$

lemma *prod-invar: igb-graph prod*
 $\langle proof \rangle$

sublocale *prod: igb-graph prod* $\langle proof \rangle$

lemma *prod-finite-reachable:*
assumes $finite\ (igba.E^* \text{ `` } igba.V0)$ $finite\ (sa.E^* \text{ `` } sa.V0)$
shows $finite\ ((g-E\ prod)^* \text{ `` } g-V0\ prod)$
 $\langle proof \rangle$

lemma *prod-fields:*

$prod.V = igba.V \times sa.V$
 $prod.E = \{ ((q,s),(q',s')) .$
 $igba.L q (sa.L s) \wedge (q,q') \in igba.E \wedge (s,s') \in sa.E \}$
 $prod.V0 = igba.V0 \times sa.V0$
 $prod.num-acc = igba.num-acc$
 $prod.acc = (\lambda(q,s). \text{ if } s \in sa.V \text{ then } igba.acc q \text{ else } \{ \})$
 <proof>

lemma *prod-run*: $prod.is-run r \longleftrightarrow$
 $igba.is-run (fst o r)$
 $\wedge sa.is-run (snd o r)$
 $\wedge (\forall i. igba.L (fst (r i)) (sa.L (snd (r i))))$ (**is** ?L=?R)
 <proof>

lemma *prod-acc*:
assumes A : $range (snd o r) \subseteq sa.V$
shows $prod.is-acc r \longleftrightarrow igba.is-acc (fst o r)$
 <proof>

lemma *gsp-correct1*:
assumes A : $prod.is-acc-run r$
shows $sa.is-run (snd o r) \wedge (sa.L o snd o r \in igba.lang)$
 <proof>

lemma *gsp-correct2*:
assumes A : $sa.is-run r \quad sa.L o r \in igba.lang$
shows $\exists r'. r = snd o r' \wedge prod.is-acc-run r'$
 <proof>

end

end

4 Lassos

theory *Lasso*
imports *Automata*
begin

record $'v$ *lasso* =
 $lasso-reach :: 'v list$
 $lasso-va :: 'v$
 $lasso-cysfx :: 'v list$

definition $lasso-v0 L \equiv case\ lasso-reach\ L\ of\ [] \Rightarrow lasso-va\ L \mid (v0\ \#) \Rightarrow v0$

definition $lasso-cycle$ **where** $lasso-cycle\ L = lasso-va\ L\ \#\ lasso-cysfx\ L$

definition $lasso-of-prpl\ prpl \equiv case\ prpl\ of\ (pr,pl) \Rightarrow ()$

$lasso-reach = pr,$
 $lasso-va = hd\ pl,$
 $lasso-cysfx = tl\ pl\ \rangle$

definition $prpl\text{-of-lasso}\ L \equiv (lasso-reach\ L, lasso-va\ L \# lasso-cysfx\ L)$

lemma $prpl\text{-of-lasso-simps}[simp]:$

$fst\ (prpl\text{-of-lasso}\ L) = lasso-reach\ L$
 $snd\ (prpl\text{-of-lasso}\ L) = lasso-va\ L \# lasso-cysfx\ L$
 $\langle proof \rangle$

lemma $lasso\text{-of-prpl-simps}[simp]:$

$lasso-reach\ (lasso\text{-of-prpl}\ prpl) = fst\ prpl$
 $snd\ prpl \neq [] \implies lasso-cycle\ (lasso\text{-of-prpl}\ prpl) = snd\ prpl$
 $\langle proof \rangle$

definition $run\text{-of-lasso} :: 'q\ lasso \Rightarrow 'q\ word$

— Run described by a lasso

where $run\text{-of-lasso}\ L \equiv lasso-reach\ L \frown (lasso-cycle\ L)^\omega$

lemma $run\text{-of-lasso-of-prpl}:$

$pl \neq [] \implies run\text{-of-lasso}\ (lasso\text{-of-prpl}\ (pr, pl)) = pr \frown pl^\omega$
 $\langle proof \rangle$

definition $map\text{-lasso}\ f\ L \equiv \langle$

$lasso-reach = map\ f\ (lasso-reach\ L),$

$lasso-va = f\ (lasso-va\ L),$

$lasso-cysfx = map\ f\ (lasso-cysfx\ L)$

\rangle

lemma $map\text{-lasso-simps}[simp]:$

$lasso-reach\ (map\text{-lasso}\ f\ L) = map\ f\ (lasso-reach\ L)$

$lasso-va\ (map\text{-lasso}\ f\ L) = f\ (lasso-va\ L)$

$lasso-cysfx\ (map\text{-lasso}\ f\ L) = map\ f\ (lasso-cysfx\ L)$

$lasso-v0\ (map\text{-lasso}\ f\ L) = f\ (lasso-v0\ L)$

$lasso-cycle\ (map\text{-lasso}\ f\ L) = map\ f\ (lasso-cycle\ L)$

$\langle proof \rangle$

lemma $map\text{-lasso-run}[simp]:$

shows $run\text{-of-lasso}\ (map\text{-lasso}\ f\ L) = f\ o\ (run\text{-of-lasso}\ L)$

$\langle proof \rangle$

context $graph\ begin$

definition $is\text{-lasso-pre} :: 'v\ lasso \Rightarrow bool$

where $is\text{-lasso-pre}\ L \equiv$

$lasso-v0\ L \in V0$

$\wedge \text{path } E \text{ (lasso-v0 } L \text{) (lasso-reach } L \text{) (lasso-va } L \text{)}$
 $\wedge \text{path } E \text{ (lasso-va } L \text{) (lasso-cycle } L \text{) (lasso-va } L \text{)}$

definition *is-lasso-prpl-pre* *prpl* \equiv case *prpl* of (*pr*, *pl*) $\Rightarrow \exists v0 \text{ va.}$

$v0 \in V0$
 $\wedge \text{pl} \neq []$
 $\wedge \text{path } E \text{ } v0 \text{ } pr \text{ } va$
 $\wedge \text{path } E \text{ } va \text{ } pl \text{ } va$

lemma *is-lasso-pre-prpl-of-lasso*[*simp*]:
is-lasso-prpl-pre (*prpl-of-lasso* *L*) \longleftrightarrow *is-lasso-pre* *L*
 $\langle \text{proof} \rangle$

lemma *is-lasso-prpl-pre-conv*:
is-lasso-prpl-pre *prpl*
 \longleftrightarrow (*snd prpl* $\neq []$ \wedge *is-lasso-pre* (*lasso-of-prpl prpl*))
 $\langle \text{proof} \rangle$

lemma *is-lasso-pre-empty*[*simp*]: $V0 = \{\}$ \Longrightarrow \neg *is-lasso-pre* *L*
 $\langle \text{proof} \rangle$

lemma *run-of-lasso-pre*:
assumes *is-lasso-pre* *L*
shows *is-run* (*run-of-lasso* *L*)
and *run-of-lasso* *L* $0 \in V0$
 $\langle \text{proof} \rangle$

end

context *gb-graph* **begin**

definition *is-lasso*
 $:: 'Q \text{ lasso} \Rightarrow \text{bool}$
— Predicate that defines a lasso
where *is-lasso* *L* \equiv
is-lasso-pre *L*
 $\wedge (\forall A \in F. (\text{set } (\text{lasso-cycle } L)) \cap A \neq \{\})$

definition *is-lasso-prpl* *prpl* \equiv
is-lasso-prpl-pre *prpl*
 $\wedge (\forall A \in F. \text{set } (\text{snd } prpl) \cap A \neq \{\})$

lemma *is-lasso-prpl-of-lasso*[*simp*]:
is-lasso-prpl (*prpl-of-lasso* *L*) \longleftrightarrow *is-lasso* *L*
 $\langle \text{proof} \rangle$

lemma *is-lasso-prpl-conv*:
is-lasso-prpl *prpl* \longleftrightarrow (*snd prpl* $\neq []$ \wedge *is-lasso* (*lasso-of-prpl prpl*))

$\langle proof \rangle$

lemma *is-lasso-empty*[simp]: $V0 = \{\} \implies \neg is-lasso\ L$
 $\langle proof \rangle$

lemma *lasso-accepted*:
assumes $L: is-lasso\ L$
shows *is-acc-run* (*run-of-lasso* L)
 $\langle proof \rangle$

lemma *lasso-prpl-acc-run*:
is-lasso-prpl (pr, pl) $\implies is-acc-run\ (pr \frown iter\ pl)$
 $\langle proof \rangle$

end

context *gb-graph*

begin

lemma *accepted-lasso*:
assumes [*simp, intro*]: *finite* (E^* “ $V0$)
assumes $A: is-acc-run\ r$
shows $\exists L. is-lasso\ L$
 $\langle proof \rangle$

end

context *b-graph*

begin

definition *is-lasso* **where** *is-lasso* $L \equiv$
is-lasso-pre L
 $\wedge (set\ (lasso-cycle\ L)) \cap F \neq \{\}$

definition *is-lasso-prpl* **where** *is-lasso-prpl* $L \equiv$
is-lasso-prpl-pre L
 $\wedge (set\ (snd\ L)) \cap F \neq \{\}$

lemma *is-lasso-pre-ext*[simp]:
gbg.is-lasso-pre $T\ m = is-lasso-pre$
 $\langle proof \rangle$

lemma *is-lasso-gbg*:
gbg.is-lasso $T\ m = is-lasso$
 $\langle proof \rangle$

lemmas *lasso-accepted* = *gbg.lasso-accepted*[*unfolded to-gbg-alt is-lasso-gbg*]

lemmas *accepted-lasso* = *gbg.accepted-lasso*[*unfolded to-gbg-alt is-lasso-gbg*]

lemma *is-lasso-prpl-of-lasso*[simp]:
is-lasso-prpl (*prpl-of-lasso* L) $\longleftrightarrow is-lasso\ L$

$\langle proof \rangle$

lemma *is-lasso-prpl-conv*:

$is-lasso-prpl\ prpl \longleftrightarrow (snd\ prpl \neq [] \wedge is-lasso\ (lasso-of-prpl\ prpl))$
 $\langle proof \rangle$

lemma *lasso-prpl-acc-run*:

$is-lasso-prpl\ (pr, pl) \implies is-acc-run\ (pr \frown iter\ pl)$
 $\langle proof \rangle$

end

context *igb-graph* **begin**

definition *is-lasso* $L \equiv$

$is-lasso-pre\ L$
 $\wedge (\forall i < num-acc. \exists q \in set\ (lasso-cycle\ L). i \in acc\ q)$

definition *is-lasso-prpl* $L \equiv$

$is-lasso-prpl-pre\ L$
 $\wedge (\forall i < num-acc. \exists q \in set\ (snd\ L). i \in acc\ q)$

lemma *is-lasso-prpl-of-lasso[simp]*:

$is-lasso-prpl\ (prpl-of-lasso\ L) \longleftrightarrow is-lasso\ L$
 $\langle proof \rangle$

lemma *is-lasso-prpl-conv*:

$is-lasso-prpl\ prpl \longleftrightarrow (snd\ prpl \neq [] \wedge is-lasso\ (lasso-of-prpl\ prpl))$
 $\langle proof \rangle$

lemma *is-lasso-pre-ext[simp]*:

$gbg.is-lasso-pre\ T\ m = is-lasso-pre$
 $\langle proof \rangle$

lemma *is-lasso-gbg*: $gbg.is-lasso\ T\ m = is-lasso$

$\langle proof \rangle$

lemmas *lasso-accepted* = $gbg.lasso-accepted[unfolded\ to-gbg-alt\ is-lasso-gbg]$

lemmas *accepted-lasso* = $gbg.accepted-lasso[unfolded\ to-gbg-alt\ is-lasso-gbg]$

lemma *lasso-prpl-acc-run*:

$is-lasso-prpl\ (pr, pl) \implies is-acc-run\ (pr \frown iter\ pl)$
 $\langle proof \rangle$

lemma *degen-lasso-sound*:

assumes A : $degen.is-lasso\ T\ m\ L$

shows $is-lasso\ (map-lasso\ fst\ L)$

$\langle proof \rangle$

end

definition *lasso-rel-ext-internal-def*: $\bigwedge Re R. \text{lasso-rel-ext } Re R \equiv \{$
 $(\langle \text{lasso-reach} = r', \text{lasso-va} = va', \text{lasso-cysfx} = \text{cysfx}', \dots = m' \rangle,$
 $\langle \text{lasso-reach} = r, \text{lasso-va} = va, \text{lasso-cysfx} = \text{cysfx}, \dots = m \rangle) \mid$
 $r' r \text{ va}' va \text{ cysfx}' \text{ cysfx } m' m.$
 $(r', r) \in \langle R \rangle \text{list-rel}$
 $\wedge (va', va) \in R$
 $\wedge (\text{cysfx}', \text{cysfx}) \in \langle R \rangle \text{list-rel}$
 $\wedge (m', m) \in Re$
 $\}$

lemma *lasso-rel-ext-def*: $\bigwedge Re R. \langle Re, R \rangle \text{lasso-rel-ext} = \{$
 $(\langle \text{lasso-reach} = r', \text{lasso-va} = va', \text{lasso-cysfx} = \text{cysfx}', \dots = m' \rangle,$
 $\langle \text{lasso-reach} = r, \text{lasso-va} = va, \text{lasso-cysfx} = \text{cysfx}, \dots = m \rangle) \mid$
 $r' r \text{ va}' va \text{ cysfx}' \text{ cysfx } m' m.$
 $(r', r) \in \langle R \rangle \text{list-rel}$
 $\wedge (va', va) \in R$
 $\wedge (\text{cysfx}', \text{cysfx}) \in \langle R \rangle \text{list-rel}$
 $\wedge (m', m) \in Re$
 $\}$
 $\langle \text{proof} \rangle$

lemma *lasso-rel-ext-sv[relator-props]*:
 $\bigwedge Re R. \llbracket \text{single-valued } Re; \text{single-valued } R \rrbracket \implies \text{single-valued } (\langle Re, R \rangle \text{lasso-rel-ext})$
 $\langle \text{proof} \rangle$

lemma *lasso-rel-ext-id[relator-props]*:
 $\bigwedge Re R. \llbracket Re = Id; R = Id \rrbracket \implies \langle Re, R \rangle \text{lasso-rel-ext} = Id$
 $\langle \text{proof} \rangle$

consts *i-lasso-ext* :: *interface* \Rightarrow *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of lasso-rel-ext i-lasso-ext*]

find-consts (-, -) *lasso-scheme*

term *lasso-reach-update*

lemma *lasso-param[param, autoref-rules]*:
 $\bigwedge Re R. (\text{lasso-reach}, \text{lasso-reach}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle R \rangle \text{list-rel}$
 $\bigwedge Re R. (\text{lasso-va}, \text{lasso-va}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow R$
 $\bigwedge Re R. (\text{lasso-cysfx}, \text{lasso-cysfx}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle R \rangle \text{list-rel}$
 $\bigwedge Re R. (\text{lasso-ext}, \text{lasso-ext})$
 $\in \langle R \rangle \text{list-rel} \rightarrow R \rightarrow \langle R \rangle \text{list-rel} \rightarrow Re \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso-reach-update}, \text{lasso-reach-update})$
 $\in (\langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso-va-update}, \text{lasso-va-update})$
 $\in (R \rightarrow R) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$

$\wedge Re R. (lasso-cysfx-update, lasso-cysfx-update)$
 $\in (\langle R \rangle list-rel \rightarrow \langle R \rangle list-rel) \rightarrow \langle Re, R \rangle lasso-rel-ext \rightarrow \langle Re, R \rangle lasso-rel-ext$
 $\wedge Re R. (lasso.more-update, lasso.more-update)$
 $\in (Re \rightarrow Re) \rightarrow \langle Re, R \rangle lasso-rel-ext \rightarrow \langle Re, R \rangle lasso-rel-ext$
 $\langle proof \rangle$

lemma *lasso-param2*[*param, autoref-rules*]:

$\wedge Re R. (lasso-v0, lasso-v0) \in \langle Re, R \rangle lasso-rel-ext \rightarrow R$
 $\wedge Re R. (lasso-cycle, lasso-cycle) \in \langle Re, R \rangle lasso-rel-ext \rightarrow \langle R \rangle list-rel$
 $\wedge Re R. (map-lasso, map-lasso)$
 $\in (R \rightarrow R') \rightarrow \langle Re, R \rangle lasso-rel-ext \rightarrow \langle unit-rel, R' \rangle lasso-rel-ext$
 $\langle proof \rangle$

lemma *lasso-of-prpl-param*: $\llbracket (l', l) \in \langle R \rangle list-rel \times_r \langle R \rangle list-rel; snd\ l \neq [] \rrbracket$

$\implies (lasso-of-prpl\ l', lasso-of-prpl\ l) \in \langle unit-rel, R \rangle lasso-rel-ext$
 $\langle proof \rangle$

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

lemma *lasso-of-prpl-autoref*[*autoref-rules*]:

assumes *SIDE-PRECOND* ($snd\ l \neq []$)
assumes $(l', l) \in \langle R \rangle list-rel \times_r \langle R \rangle list-rel$
shows $(lasso-of-prpl\ l',$
 $(OP\ lasso-of-prpl$
 $\quad \dots \langle R \rangle list-rel \times_r \langle R \rangle list-rel \rightarrow \langle unit-rel, R \rangle lasso-rel-ext) \l
 $\in \langle unit-rel, R \rangle lasso-rel-ext$
 $\langle proof \rangle$

end

4.1 Implementing runs by lassos

definition *lasso-run-rel-def-internal*:

$lasso-run-rel\ R \equiv br\ run-of-lasso\ (\lambda-. True)\ O\ (nat-rel \rightarrow R)$

lemma *lasso-run-rel-def*:

$\langle R \rangle lasso-run-rel = br\ run-of-lasso\ (\lambda-. True)\ O\ (nat-rel \rightarrow R)$
 $\langle proof \rangle$

lemma *lasso-run-rel-sv*[*relator-props*]:

$single-valued\ R \implies single-valued\ (\langle R \rangle lasso-run-rel)$
 $\langle proof \rangle$

consts *i-run* :: *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of lasso-run-rel i-run*]

definition [*simp*]: *op-map-run* \equiv (*o*)

lemma *[autoref-op-pat]*: $(o) \equiv op\text{-map-run} \langle proof \rangle$

lemma *map-lasso-run-refine[autoref-rules]*:
shows $(map\text{-lasso}, op\text{-map-run}) \in (R \rightarrow R') \rightarrow \langle R \rangle\text{lasso-run-rel} \rightarrow \langle R' \rangle\text{lasso-run-rel}$
 $\langle proof \rangle$

end

5 Simulation

theory *Simulation*
imports *Automata*
begin

lemma *finite-ImageI*:
assumes *finite A*
assumes $\bigwedge a. a \in A \implies \text{finite } (R''\{a\})$
shows *finite (R''A)*
 $\langle proof \rangle$

6 Simulation

6.1 Functional Relations

definition *the-br- α* $R \equiv \lambda x. \text{SOME } y. (x, y) \in R$
abbreviation (*input*) *the-br-invar* $R \equiv \lambda x. x \in \text{Domain } R$

lemma *the-br[simp]*:
assumes *single-valued R*
shows $br (the\text{-br-}\alpha R) (the\text{-br-invar } R) = R$
 $\langle proof \rangle$

lemma *the-br-br[simp]*:
 $I x \implies the\text{-br-}\alpha (br \alpha I) x = \alpha x$
 $the\text{-br-invar } (br \alpha I) = I$
 $\langle proof \rangle$

6.2 Relation between Runs

definition *run-rel* :: $('a \times 'b) \text{ set} \Rightarrow ('a \text{ word} \times 'b \text{ word}) \text{ set}$ **where**
 $run\text{-rel } R \equiv \{(ra, rb). \forall i. (ra\ i, rb\ i) \in R\}$

lemma *run-rel-converse[simp]*: $(ra, rb) \in run\text{-rel } (R^{-1}) \iff (rb, ra) \in run\text{-rel } R$
 $\langle proof \rangle$

lemma *run-rel-single-valued*: *single-valued R*

$\implies (ra, rb) \in \text{run-rel } R \iff ((\forall i. \text{the-br-invar } R (ra\ i)) \wedge rb = \text{the-br-}\alpha\ R\ o\ ra)$
 <proof>

6.3 Simulation

locale simulation =
 a: graph A +
 b: graph B
for R :: ('a × 'b) set
and A :: ('a, -) graph-rec-scheme
and B :: ('b, -) graph-rec-scheme
 +
assumes nodes-sim: $a \in a.V \implies (a, b) \in R \implies b \in b.V$
assumes init-sim: $a0 \in a.V0 \implies \exists b0. b0 \in b.V0 \wedge (a0, b0) \in R$
assumes step-sim: $(a, a') \in a.E \implies (a, b) \in R \implies \exists b'. (b, b') \in b.E \wedge (a', b') \in R$
begin

lemma simulation-this: simulation R A B <proof>

lemma run-sim:
assumes arun: a.is-run ra
obtains rb **where** b.is-run rb $(ra, rb) \in \text{run-rel } R$
 <proof>

lemma stuck-sim:
assumes $(a, b) \in R$
assumes $b \notin \text{Domain } b.E$
shows $a \notin \text{Domain } a.E$
 <proof>

lemma run-Domain: a.is-run r $\implies r\ i \in \text{Domain } R$
 <proof>

lemma br-run-sim:
assumes $R = \text{br } \alpha\ I$
assumes a.is-run r
shows b.is-run $(\alpha\ o\ r)$
 <proof>

lemma is-reachable-sim: $a \in a.E^* \text{ `` } a.V0 \implies \exists b. (a, b) \in R \wedge b \in b.E^* \text{ `` } b.V0$
 <proof>

lemma reachable-sim: $a.E^* \text{ `` } a.V0 \subseteq R^{-1} \text{ `` } b.E^* \text{ `` } b.V0$
 <proof>

lemma reachable-finite-sim:

```

assumes finite (b.E* “ b.V0)
assumes  $\bigwedge b. b \in b.E^* \text{ “ } b.V0 \implies \textit{finite} (R^{-1} \text{ “ } \{b\})$ 
shows finite (a.E* “ a.V0)
<proof>

```

end

```

lemma simulation-trans[trans]:
assumes simulation R1 A B
assumes simulation R2 B C
shows simulation (R1 O R2) A C
<proof>

```

```

lemma (in graph) simulation-refl[simp]: simulation Id G G <proof>

```

```

locale lsimulation =
  a: sa A +
  b: sa B +
  simulation R A B
for R :: ('a × 'b) set
and A :: ('a, 'l, -) sa-rec-scheme
and B :: ('b, 'l, -) sa-rec-scheme
  +
assumes labeling-consistent: (a, b) ∈ R  $\implies$  a.L a = b.L b
begin

```

```

lemma lsimulation-this: lsimulation R A B <proof>

```

```

lemma run-rel-consistent: (ra, rb) ∈ run-rel R  $\implies$  a.L o ra = b.L o rb
<proof>

```

```

lemma accept-sim: a.accept w  $\implies$  b.accept w
<proof>

```

end

```

lemma lsimulation-trans[trans]:
assumes lsimulation R1 A B
assumes lsimulation R2 B C
shows lsimulation (R1 O R2) A C
<proof>

```

```

lemma (in sa) lsimulation-refl[simp]: lsimulation Id G G <proof>

```

6.4 Bisimulation

```

locale bisimulation =
  a: graph A +
  b: graph B +

```

```

s1: simulation R A B +
s2: simulation R-1 B A
for R :: ('a × 'b) set
and A :: ('a, -) graph-rec-scheme
and B :: ('b, -) graph-rec-scheme
begin

lemma bisimulation-this: bisimulation R A B ⟨proof⟩

lemma converse: bisimulation (R-1) B A
⟨proof⟩

lemma br-run-conv:
  assumes R = br α I
  shows b.is-run rb ⟷ (∃ ra. rb=α o ra ∧ a.is-run ra)
  ⟨proof⟩

lemma bri-run-conv:
  assumes R = (br γ I)-1
  shows a.is-run ra ⟷ (∃ rb. ra=γ o rb ∧ b.is-run rb)
  ⟨proof⟩

lemma inj-map-run-eg:
  assumes inj α
  assumes E: α o r1 = α o r2
  shows r1 = r2
  ⟨proof⟩

lemma br-inj-run-conv:
  assumes INJ: inj α
  assumes [simp]: R = br α I
  shows b.is-run (α o ra) ⟷ a.is-run ra
  ⟨proof⟩

lemma single-valued-run-conv:
  assumes single-valued R
  shows b.is-run rb
    ⟷ (∃ ra. rb=the-br-α R o ra ∧ a.is-run ra)
  ⟨proof⟩

lemma stuck-bisim:
  assumes A: (a, b) ∈ R
  shows a ∈ Domain a.E ⟷ b ∈ Domain b.E
  ⟨proof⟩

end

lemma bisimulation-trans[trans]:
  assumes bisimulation R1 A B

```


assumes *bisimulation* $R2\ B\ C$
shows *bisimulation* $(R1\ O\ R2)\ A\ C$
 $\langle proof \rangle$

lemma (*in graph*) *bisimulation-refl[simp]*: *bisimulation* $Id\ G\ G\ \langle proof \rangle$

locale *lbisimulation* =
a: *sa* $A\ +$
b: *sa* $B\ +$
s1: *lsimulation* $R\ A\ B\ +$
s2: *lsimulation* $R^{-1}\ B\ A\ +$
bisimulation $R\ A\ B$
for $R :: ('a \times 'b)\ set$
and $A :: ('a, 'l, -)\ sa-rec-scheme$
and $B :: ('b, 'l, -)\ sa-rec-scheme$
begin

lemma *lbisimulation-this*: *lbisimulation* $R\ A\ B\ \langle proof \rangle$

lemma *accept-bisim*: $a.accept = b.accept$
 $\langle proof \rangle$

end

lemma *lbisimulation-trans[trans]*:
assumes *lbisimulation* $R1\ A\ B$
assumes *lbisimulation* $R2\ B\ C$
shows *lbisimulation* $(R1\ O\ R2)\ A\ C$
 $\langle proof \rangle$

lemma (*in sa*) *bisimulation-refl[simp]*: *bisimulation* $Id\ G\ G\ \langle proof \rangle$

end

theory *Step-Conv*

imports *Main*

begin

definition *rel-of-pred* $s \equiv \{(a,b). s\ a\ b\}$
definition *rel-of-succ* $s \equiv \{(a,b). b \in s\ a\}$

definition *pred-of-rel* $s \equiv \lambda a. \{(a,b) \in s\}$
definition *pred-of-succ* $s \equiv \lambda a. b \in s\ a$

definition *succ-of-rel* $s \equiv \lambda a. \{b. (a,b) \in s\}$
definition *succ-of-pred* $s \equiv \lambda a. \{b. s\ a\ b\}$

lemma *rps-expand[simp]*:
 $(a,b) \in rel-of-pred\ p \longleftrightarrow p\ a\ b$

$$(a,b) \in \text{rel-of-succ } s \iff b \in s a$$

$$\begin{aligned} \text{pred-of-rel } r a b &\iff (a,b) \in r \\ \text{pred-of-succ } s a b &\iff b \in s a \end{aligned}$$

$$\begin{aligned} b \in \text{succ-of-rel } r a &\iff (a,b) \in r \\ b \in \text{succ-of-pred } p a &\iff p a b \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *rps-conv[simp]*:

$$\begin{aligned} \text{rel-of-pred } (\text{pred-of-rel } r) &= r \\ \text{rel-of-pred } (\text{pred-of-succ } s) &= \text{rel-of-succ } s \end{aligned}$$

$$\begin{aligned} \text{rel-of-succ } (\text{succ-of-rel } r) &= r \\ \text{rel-of-succ } (\text{succ-of-pred } p) &= \text{rel-of-pred } p \end{aligned}$$

$$\begin{aligned} \text{pred-of-rel } (\text{rel-of-pred } p) &= p \\ \text{pred-of-rel } (\text{rel-of-succ } s) &= \text{pred-of-succ } s \end{aligned}$$

$$\begin{aligned} \text{pred-of-succ } (\text{succ-of-pred } p) &= p \\ \text{pred-of-succ } (\text{succ-of-rel } r) &= \text{pred-of-rel } r \end{aligned}$$

$$\begin{aligned} \text{succ-of-rel } (\text{rel-of-succ } s) &= s \\ \text{succ-of-rel } (\text{rel-of-pred } p) &= \text{succ-of-pred } p \end{aligned}$$

$$\begin{aligned} \text{succ-of-pred } (\text{pred-of-succ } s) &= s \\ \text{succ-of-pred } (\text{pred-of-rel } r) &= \text{succ-of-rel } r \\ \langle \text{proof} \rangle \end{aligned}$$

definition *m2r-rel* :: ('a × 'a option) set ⇒ 'a option rel

where *m2r-rel* *r* ≡ {(Some a,b) | a b. (a,b) ∈ r}

definition *m2r-pred* :: ('a ⇒ 'a option ⇒ bool) ⇒ 'a option ⇒ 'a option ⇒ bool

where *m2r-pred* *p* ≡ λNone ⇒ λ-. False | Some a ⇒ p a

definition *m2r-succ* :: ('a ⇒ 'a option set) ⇒ 'a option ⇒ 'a option set

where *m2r-succ* *s* ≡ λNone ⇒ {} | Some a ⇒ s a

lemma *m2r-expand[simp]*:

$$\begin{aligned} (a,b) \in \text{m2r-rel } r &\iff (\exists a'. a = \text{Some } a' \wedge (a',b) \in r) \\ \text{m2r-pred } p a b &\iff (\exists a'. a = \text{Some } a' \wedge p a' b) \\ b \in \text{m2r-succ } s a &\iff (\exists a'. a = \text{Some } a' \wedge b \in s a') \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *m2r-conv[simp]*:

$$\begin{aligned} \text{m2r-rel } (\text{rel-of-succ } s) &= \text{rel-of-succ } (\text{m2r-succ } s) \\ \text{m2r-rel } (\text{rel-of-pred } p) &= \text{rel-of-pred } (\text{m2r-pred } p) \end{aligned}$$

$m2r\text{-pred } (\text{pred-of-succ } s) = \text{pred-of-succ } (m2r\text{-succ } s)$
 $m2r\text{-pred } (\text{pred-of-rel } r) = \text{pred-of-rel } (m2r\text{-rel } r)$

$m2r\text{-succ } (\text{succ-of-pred } p) = \text{succ-of-pred } (m2r\text{-pred } p)$
 $m2r\text{-succ } (\text{succ-of-rel } r) = \text{succ-of-rel } (m2r\text{-rel } r)$
 $\langle \text{proof} \rangle$

definition $\text{rel-of-enex } enex \equiv \text{let } (en, ex) = enex \text{ in } \{(s, ex \ a \ s) \mid s \ a. \ a \in en \ s\}$

definition $\text{pred-of-enex } enex \equiv \lambda s \ s'. \ \text{let } (en, ex) = enex \text{ in } \exists a \in en \ s. \ s' = ex \ a \ s$

definition $\text{succ-of-enex } enex \equiv \lambda s. \ \text{let } (en, ex) = enex \text{ in } \{s'. \ \exists a \in en \ s. \ s' = ex \ a \ s\}$

lemma $x\text{-of-enex-expand}[simp]$:

$(s, s') \in \text{rel-of-enex } (en, ex) \longleftrightarrow (\exists a \in en \ s. \ s' = ex \ a \ s)$
 $\text{pred-of-enex } (en, ex) \ s \ s' \longleftrightarrow (\exists a \in en \ s. \ s' = ex \ a \ s)$
 $s' \in \text{succ-of-enex } (en, ex) \ s \longleftrightarrow (\exists a \in en \ s. \ s' = ex \ a \ s)$
 $\langle \text{proof} \rangle$

lemma $x\text{-of-enex-conv}[simp]$:

$\text{rel-of-pred } (\text{pred-of-enex } enex) = \text{rel-of-enex } enex$
 $\text{rel-of-succ } (\text{succ-of-enex } enex) = \text{rel-of-enex } enex$
 $\text{pred-of-rel } (\text{rel-of-enex } enex) = \text{pred-of-enex } enex$
 $\text{pred-of-succ } (\text{succ-of-enex } enex) = \text{pred-of-enex } enex$
 $\text{succ-of-rel } (\text{rel-of-enex } enex) = \text{succ-of-enex } enex$
 $\text{succ-of-pred } (\text{pred-of-enex } enex) = \text{succ-of-enex } enex$
 $\langle \text{proof} \rangle$

end

theory *Stuttering-Extension*

imports *Simulation Step-Conv*

begin

definition $\text{stutter-extend-edges} :: 'v \ \text{set} \Rightarrow 'v \ \text{digraph} \Rightarrow 'v \ \text{digraph}$

where $\text{stutter-extend-edges } V \ E \equiv E \cup \{(v, v) \mid v. \ v \in V \wedge v \notin \text{Domain } E\}$

lemma $\text{stutter-extend-edgesI-edge}$:

assumes $(u, v) \in E$
shows $(u, v) \in \text{stutter-extend-edges } V \ E$
 $\langle \text{proof} \rangle$

lemma $\text{stutter-extend-edgesI-stutter}$:

assumes $v \in V \quad v \notin \text{Domain } E$
shows $(v, v) \in \text{stutter-extend-edges } V \ E$
 $\langle \text{proof} \rangle$

lemma $\text{stutter-extend-edgesE}$:

assumes $(u, v) \in \text{stutter-extend-edges } V \ E$
obtains $(\text{edge}) \ (u, v) \in E \mid (\text{stutter}) \quad u \in V \quad u \notin \text{Domain } E \quad u = v$
 $\langle \text{proof} \rangle$

lemma *stutter-extend-wf*: $E \subseteq V \times V \implies \text{stutter-extend-edges } V E \subseteq V \times V$
 ⟨proof⟩

lemma *stutter-extend-edges-rtrancl[simp]*: $(\text{stutter-extend-edges } V E)^* = E^*$
 ⟨proof⟩

lemma *stutter-extend-domain*: $V \subseteq \text{Domain } (\text{stutter-extend-edges } V E)$
 ⟨proof⟩

definition *stutter-extend* :: $('v, -)$ graph-rec-scheme $\Rightarrow ('v, -)$ graph-rec-scheme
where *stutter-extend* $G \equiv$
 (
 $g\text{-}V = g\text{-}V G,$
 $g\text{-}E = \text{stutter-extend-edges } (g\text{-}V G) (g\text{-}E G),$
 $g\text{-}V0 = g\text{-}V0 G,$
 $\dots = \text{graph-rec.more } G$
)

lemma *stutter-extend-simps[simp]*:
 $g\text{-}V (\text{stutter-extend } G) = g\text{-}V G$
 $g\text{-}E (\text{stutter-extend } G) = \text{stutter-extend-edges } (g\text{-}V G) (g\text{-}E G)$
 $g\text{-}V0 (\text{stutter-extend } G) = g\text{-}V0 G$
 ⟨proof⟩

lemma *stutter-extend-simps-sa[simp]*:
 $sa\text{-}L (\text{stutter-extend } G) = sa\text{-}L G$
 ⟨proof⟩

lemma (in graph) *stutter-extend-graph*: graph $(\text{stutter-extend } G)$
 ⟨proof⟩

lemma (in sa) *stutter-extend-sa*: sa $(\text{stutter-extend } G)$
 ⟨proof⟩

lemma (in bisimulation) *stutter-extend*: bisimulation $R (\text{stutter-extend } A) (\text{stutter-extend } B)$
 ⟨proof⟩

lemma (in lbisimulation) *lstutter-extend*: lbisimulation $R (\text{stutter-extend } A) (\text{stutter-extend } B)$
 ⟨proof⟩

definition *stutter-extend-en* :: $('s \Rightarrow 'a \text{ set}) \Rightarrow ('s \Rightarrow 'a \text{ option set})$ **where**
 $\text{stutter-extend-en } en \equiv \lambda s. \text{ let } as = en \ s \text{ in if } as = \{\} \text{ then } \{None\} \text{ else } Some\ 'as$

definition *stutter-extend-ex* :: $('a \Rightarrow 's \Rightarrow 's) \Rightarrow ('a \text{ option} \Rightarrow 's \Rightarrow 's)$ **where**
 $\text{stutter-extend-ex } ex \equiv \lambda None \Rightarrow id \mid Some \ a \Rightarrow ex \ a$

abbreviation *stutter-extend-enex*
 :: $('s \Rightarrow 'a \text{ set}) \times ('a \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 'a \text{ option set}) \times ('a \text{ option} \Rightarrow 's \Rightarrow 's)$

where

stutter-extend-enex \equiv *map-prod* *stutter-extend-en* *stutter-extend-ex*

lemma *stutter-extend-pred-of-enex-conv*:

stutter-extend-edges UNIV (rel-of-enex enex) = rel-of-enex (stutter-extend-enex enex)
<proof>

lemma *stutter-extend-en-Some-eq[simp]*:

Some a \in stutter-extend-en en gc \longleftrightarrow a \in en gc
stutter-extend-ex ex (Some a) gc = ex a gc
<proof>

lemma *stutter-extend-ex-None-eq[simp]*:

stutter-extend-ex ex None = id
<proof>

end

7 Implementing Graphs

theory *Digraph-Impl*

imports *Digraph*

begin

7.1 Directed Graphs by Successor Function

type-synonym *'a slg* = *'a \Rightarrow 'a list*

definition *slg-rel* :: *('a \times 'b) set \Rightarrow ('a slg \times 'b digraph) set **where***

slg-rel-def-internal: slg-rel R \equiv
(R \rightarrow $\langle R \rangle$ list-set-rel) O br (λ succs. $\{(u,v). v \in \text{succs } u\}$) (λ -. True)

lemma *slg-rel-def: $\langle R \rangle$ slg-rel =*

(R \rightarrow $\langle R \rangle$ list-set-rel) O br (λ succs. $\{(u,v). v \in \text{succs } u\}$) (λ -. True)
<proof>

lemma *slg-rel-sv[relator-props]*:

[[single-valued R; Range R = UNIV]] \implies single-valued ($\langle R \rangle$ slg-rel)
<proof>

consts *i-slg* :: *interface \Rightarrow interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI[of slg-rel i-slg]*

definition [*simp*]: *op-slg-succs E v \equiv E^{''}{v}*

lemma [*autoref-itype*]: *op-slg-succs ::_i $\langle I \rangle_i$ i-slg \rightarrow_i I \rightarrow_i $\langle I \rangle_i$ i-set* *<proof>*

context begin interpretation *autoref-syn* $\langle proof \rangle$
lemma [*autoref-op-pat*]: $E^{\{v\}} \equiv op\text{-slg}\text{-succs}\$E\$v \langle proof \rangle$
end

lemma *refine-slg-succs*[*autoref-rules-raw*]:
 $(\lambda succs\ v.\ succs\ v, op\text{-slg}\text{-succs}) \in \langle R \rangle slg\text{-rel} \rightarrow R \rightarrow \langle R \rangle list\text{-set}\text{-rel}$
 $\langle proof \rangle$

definition *E-of-succ succ* $\equiv \{ (u, v). v \in succ\ u \}$

definition *succ-of-E E* $\equiv (\lambda u.\ \{v.\ (u, v) \in E\})$

lemma *E-of-succ-of-E*[*simp*]: $E\text{-of}\text{-succ}\ (succ\text{-of}\text{-E}\ E) = E$
 $\langle proof \rangle$

lemma *succ-of-E-of-succ*[*simp*]: $succ\text{-of}\text{-E}\ (E\text{-of}\text{-succ}\ E) = E$
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$
lemma [*autoref-itype*]: $E\text{-of}\text{-succ} ::_i (I \rightarrow_i \langle I \rangle_i i\text{-set}) \rightarrow_i \langle I \rangle_i i\text{-slg} \langle proof \rangle$
lemma [*autoref-itype*]: $succ\text{-of}\text{-E} ::_i \langle I \rangle_i i\text{-slg} \rightarrow_i I \rightarrow_i \langle I \rangle_i i\text{-set} \langle proof \rangle$
end

lemma *E-of-succ-refine*[*autoref-rules*]:
 $(\lambda x.\ x, E\text{-of}\text{-succ}) \in (R \rightarrow \langle R \rangle list\text{-set}\text{-rel}) \rightarrow \langle R \rangle slg\text{-rel}$
 $(\lambda x.\ x, succ\text{-of}\text{-E}) \in \langle R \rangle slg\text{-rel} \rightarrow (R \rightarrow \langle R \rangle list\text{-set}\text{-rel})$
 $\langle proof \rangle$

7.1.1 Restricting Edges

definition *op-graph-restrict* :: $'v\ set \Rightarrow 'v\ set \Rightarrow ('v \times 'v)\ set \Rightarrow ('v \times 'v)\ set$
where [*simp*]: $op\text{-graph}\text{-restrict}\ Vl\ Vr\ E \equiv E \cap Vl \times Vr$

definition *op-graph-restrict-left* :: $'v\ set \Rightarrow ('v \times 'v)\ set \Rightarrow ('v \times 'v)\ set$
where [*simp*]: $op\text{-graph}\text{-restrict}\text{-left}\ Vl\ E \equiv E \cap Vl \times UNIV$

definition *op-graph-restrict-right* :: $'v\ set \Rightarrow ('v \times 'v)\ set \Rightarrow ('v \times 'v)\ set$
where [*simp*]: $op\text{-graph}\text{-restrict}\text{-right}\ Vr\ E \equiv E \cap UNIV \times Vr$

lemma [*autoref-op-pat*]:
 $E \cap (Vl \times Vr) \equiv op\text{-graph}\text{-restrict}\ Vl\ Vr\ E$
 $E \cap (Vl \times UNIV) \equiv op\text{-graph}\text{-restrict}\text{-left}\ Vl\ E$
 $E \cap (UNIV \times Vr) \equiv op\text{-graph}\text{-restrict}\text{-right}\ Vr\ E$
 $\langle proof \rangle$

lemma *graph-restrict-aimpl*: $op\text{-graph}\text{-restrict}\ Vl\ Vr\ E =$
 $E\text{-of}\text{-succ}\ (\lambda v.\ \text{if } v \in Vl \text{ then } \{x \in E^{\{v\}}.\ x \in Vr\} \text{ else } \{\})$
 $\langle proof \rangle$

lemma *graph-restrict-left-aimpl*: $op\text{-graph}\text{-restrict}\text{-left}\ Vl\ E =$

E-of-succ ($\lambda v. \text{if } v \in V \text{ then } E^{\{v\}} \text{ else } \{\}$)
 <proof>

lemma *graph-restrict-right-aimpl*: *op-graph-restrict-right* $\forall r E =$
E-of-succ ($\lambda v. \{x \in E^{\{v\}}, x \in Vr\}$)
 <proof>

schematic-goal *graph-restrict-impl-aux*:
 fixes *Rsl Rsr*
 notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
 assumes [*autoref-rules*]: (*meml*, (\in)) $\in R \rightarrow \langle R \rangle Rsl \rightarrow \text{bool-rel}$
 assumes [*autoref-rules*]: (*memr*, (\in)) $\in R \rightarrow \langle R \rangle Rsr \rightarrow \text{bool-rel}$
 shows ($?c, \text{op-graph-restrict}$) $\in \langle R \rangle Rsl \rightarrow \langle R \rangle Rsr \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \langle R \rangle \text{slg-rel}$
 <proof>

schematic-goal *graph-restrict-left-impl-aux*:
 fixes *Rsl Rsr*
 notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
 assumes [*autoref-rules*]: (*meml*, (\in)) $\in R \rightarrow \langle R \rangle Rsl \rightarrow \text{bool-rel}$
 shows ($?c, \text{op-graph-restrict-left}$) $\in \langle R \rangle Rsl \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \langle R \rangle \text{slg-rel}$
 <proof>

schematic-goal *graph-restrict-right-impl-aux*:
 fixes *Rsl Rsr*
 notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
 assumes [*autoref-rules*]: (*memr*, (\in)) $\in R \rightarrow \langle R \rangle Rsr \rightarrow \text{bool-rel}$
 shows ($?c, \text{op-graph-restrict-right}$) $\in \langle R \rangle Rsr \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \langle R \rangle \text{slg-rel}$
 <proof>

concrete-definition *graph-restrict-impl* **uses** *graph-restrict-impl-aux*
concrete-definition *graph-restrict-left-impl* **uses** *graph-restrict-left-impl-aux*
concrete-definition *graph-restrict-right-impl* **uses** *graph-restrict-right-impl-aux*

context begin interpretation *autoref-syn* <proof>
lemma [*autoref-itype*]:
op-graph-restrict $::_i \langle I \rangle_i \text{i-set} \rightarrow_i \langle I \rangle_i \text{i-set} \rightarrow_i \langle I \rangle_i \text{i-slg} \rightarrow_i \langle I \rangle_i \text{i-slg}$
op-graph-restrict-right $::_i \langle I \rangle_i \text{i-set} \rightarrow_i \langle I \rangle_i \text{i-slg} \rightarrow_i \langle I \rangle_i \text{i-slg}$
op-graph-restrict-left $::_i \langle I \rangle_i \text{i-set} \rightarrow_i \langle I \rangle_i \text{i-slg} \rightarrow_i \langle I \rangle_i \text{i-slg}$
 <proof>

end

lemmas [*autoref-rules-raw*] =
graph-restrict-impl.refine[*OF GEN-OP-D GEN-OP-D*]
graph-restrict-left-impl.refine[*OF GEN-OP-D*]
graph-restrict-right-impl.refine[*OF GEN-OP-D*]

schematic-goal ($?c::?c, \lambda(E::\text{nat digraph}) x. E^{\{x\}} \in ?R$)
 <proof>

lemma *graph-minus-aimpl*:

fixes $E1\ E2 :: 'a\ rel$
shows $E1 - E2 = E\text{-of-succ}\ (\lambda x. E1\ \{\{x\}\} - E2\ \{\{x\}\})$
 $\langle proof \rangle$

schematic-goal $graph\ minus\ impl\ aux:$

fixes $R :: ('v_i \times 'v)\ set$
assumes $[autoref\ rules]: (eq, (=)) \in R \rightarrow R \rightarrow bool\ rel$
shows $(?c, (-)) \in \langle R \rangle slg\ rel \rightarrow \langle R \rangle slg\ rel \rightarrow \langle R \rangle slg\ rel$
 $\langle proof \rangle$

lemmas $[autoref\ rules] = graph\ minus\ impl\ aux[OF\ GEN\ OP\ D]$

lemma $graph\ minus\ set\ aimpl:$

fixes $E1\ E2 :: 'a\ rel$
shows $E1 - E2 = E\text{-of-succ}\ (\lambda u. \{v \in E1\ \{\{u\}\}. (u, v) \notin E2\})$
 $\langle proof \rangle$

schematic-goal $graph\ minus\ set\ impl\ aux:$

fixes $R :: ('v_i \times 'v)\ set$
assumes $[autoref\ rules]: (eq, (=)) \in R \rightarrow R \rightarrow bool\ rel$
assumes $[autoref\ rules]: (mem, (\in)) \in R \times_r R \rightarrow \langle R \times_r R \rangle Rs \rightarrow bool\ rel$
shows $(?c, (-)) \in \langle R \rangle slg\ rel \rightarrow \langle R \times_r R \rangle Rs \rightarrow \langle R \rangle slg\ rel$
 $\langle proof \rangle$

lemmas $[autoref\ rules\ (overloaded)] = graph\ minus\ set\ impl\ aux[OF\ GEN\ OP\ D\ GEN\ OP\ D]$

7.2 Rooted Graphs

7.2.1 Operation Identification Setup

consts

$i\text{-g-ext} :: interface \Rightarrow interface \Rightarrow interface$

abbreviation $i\text{-frg} \equiv \langle i\text{-unit} \rangle_i i\text{-g-ext}$

context begin interpretation $autoref\ syn\ \langle proof \rangle$

lemma $g\text{-type}[autoref\ itype]:$

$g\text{-V} ::_i \langle Ie, I \rangle_i i\text{-g-ext} \rightarrow_i \langle I \rangle_i i\text{-set}$
 $g\text{-E} ::_i \langle Ie, I \rangle_i i\text{-g-ext} \rightarrow_i \langle I \rangle_i i\text{-slg}$
 $g\text{-V0} ::_i \langle Ie, I \rangle_i i\text{-g-ext} \rightarrow_i \langle I \rangle_i i\text{-set}$
 $graph\ rec\ ext$
 $::_i \langle I \rangle_i i\text{-set} \rightarrow_i \langle I \rangle_i i\text{-slg} \rightarrow_i \langle I \rangle_i i\text{-set} \rightarrow_i iE \rightarrow_i \langle Ie, I \rangle_i i\text{-g-ext}$
 $\langle proof \rangle$

end

7.2.2 Generic Implementation

record ($'vi, 'ei, 'v0i$) *gen-g-impl* =
 $gi-V :: 'vi$
 $gi-E :: 'ei$
 $gi-V0 :: 'v0i$

definition *gen-g-impl-rel-ext-internal-def*: $\bigwedge Rm Rv Re Rv0. \text{gen-g-impl-rel-ext } Rm Rv Re Rv0$

$\equiv \{$ (*gen-g-impl-ext* $Vi Ei V0i mi, \text{graph-rec-ext } V E V0 m$)
 $| Vi Ei V0i mi V E V0 m.$
 $(Vi, V) \in Rv \wedge (Ei, E) \in Re \wedge (V0i, V0) \in Rv0 \wedge (mi, m) \in Rm$
 $\}$

lemma *gen-g-impl-rel-ext-def*: $\bigwedge Rm Rv Re Rv0. \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext}$

$\equiv \{$ (*gen-g-impl-ext* $Vi Ei V0i mi, \text{graph-rec-ext } V E V0 m$)
 $| Vi Ei V0i mi V E V0 m.$
 $(Vi, V) \in Rv \wedge (Ei, E) \in Re \wedge (V0i, V0) \in Rv0 \wedge (mi, m) \in Rm$
 $\}$
 $\langle \text{proof} \rangle$

lemma *gen-g-impl-rel-sv[relator-props]*:

$\bigwedge Rm Rv Re Rv0. \llbracket \text{single-valued } Rv; \text{single-valued } Re; \text{single-valued } Rv0; \text{single-valued } Rm \rrbracket \implies$
 $\text{single-valued } (\langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext})$
 $\langle \text{proof} \rangle$

lemma *gen-g-refine*:

$\bigwedge Rm Rv Re Rv0. (gi-V, g-V) \in \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext} \rightarrow Rv$
 $\bigwedge Rm Rv Re Rv0. (gi-E, g-E) \in \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext} \rightarrow Re$
 $\bigwedge Rm Rv Re Rv0. (gi-V0, g-V0) \in \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext} \rightarrow Rv0$
 $\bigwedge Rm Rv Re Rv0. (\text{gen-g-impl-ext}, \text{graph-rec-ext})$
 $\in Rv \rightarrow Re \rightarrow Rv0 \rightarrow Rm \rightarrow \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext}$
 $\langle \text{proof} \rangle$

7.2.3 Implementation with list-set for Nodes

type-synonym ($'v, 'm$) *frgv-impl-scheme* =
 $('v \text{ list}, 'v \Rightarrow 'v \text{ list}, 'v \text{ list}, 'm) \text{gen-g-impl-scheme}$

definition *frgv-impl-rel-ext-internal-def*:

frgv-impl-rel-ext $Rm Rv$
 $\equiv \langle Rm, \langle Rv \rangle \text{list-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel} \rangle \text{gen-g-impl-rel-ext}$

lemma *frgv-impl-rel-ext-def*: $\langle Rm, Rv \rangle \text{frgv-impl-rel-ext}$

$\equiv \langle Rm, \langle Rv \rangle \text{list-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel} \rangle \text{gen-g-impl-rel-ext}$
 $\langle \text{proof} \rangle$

lemma [*autoref-rel-intf*]: *REL-INTF* *frgv-impl-rel-ext i-g-ext*

$\langle \text{proof} \rangle$

lemma [*relator-props, simp*]:
 $\llbracket \text{single-valued } Rv; \text{Range } Rv = \text{UNIV}; \text{single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv \rangle \text{frgv-impl-rel-ext})$
 $\langle \text{proof} \rangle$

lemmas [*param, autoref-rules*] = *gen-g-refine*[**where**
 $Rv = \langle Rv \rangle \text{list-set-rel}$ **and** $Re = \langle Rv \rangle \text{slg-rel}$ **and** $?Rv0.0 = \langle Rv \rangle \text{list-set-rel}$
for Rv , *folded frgv-impl-rel-ext-def*]

7.2.4 Implementation with Cfun for Nodes

This implementation allows for the universal node set.

type-synonym ($'v, 'm$) *g-impl-scheme* =
 $('v \Rightarrow \text{bool}, 'v \Rightarrow 'v \text{ list}, 'v \text{ list}, 'm) \text{gen-g-impl-scheme}$

definition *g-impl-rel-ext-internal-def*:
 $g\text{-impl-rel-ext } Rm \ Rv$
 $\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel} \rangle \text{gen-g-impl-rel-ext}$

lemma *g-impl-rel-ext-def*: $\langle Rm, Rv \rangle g\text{-impl-rel-ext}$
 $\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel} \rangle \text{gen-g-impl-rel-ext}$
 $\langle \text{proof} \rangle$

lemma [*autoref-rel-intf*]: *REL-INTF g-impl-rel-ext i-g-ext*
 $\langle \text{proof} \rangle$

lemma [*relator-props, simp*]:
 $\llbracket \text{single-valued } Rv; \text{Range } Rv = \text{UNIV}; \text{single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv \rangle g\text{-impl-rel-ext})$
 $\langle \text{proof} \rangle$

lemmas [*param, autoref-rules*] = *gen-g-refine*[**where**
 $Rv = \langle Rv \rangle \text{fun-set-rel}$
and $Re = \langle Rv \rangle \text{slg-rel}$
and $?Rv0.0 = \langle Rv \rangle \text{list-set-rel}$
for Rv , *folded g-impl-rel-ext-def*]

lemma [*autoref-rules*]: $(gi\text{-V-update}, g\text{-V-update}) \in (\langle Rv \rangle \text{fun-set-rel} \rightarrow \langle Rv \rangle \text{fun-set-rel})$
 \rightarrow
 $\langle Rm, Rv \rangle g\text{-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle g\text{-impl-rel-ext}$
 $\langle \text{proof} \rangle$

lemma [*autoref-rules*]: $(gi\text{-E-update}, g\text{-E-update}) \in (\langle Rv \rangle \text{slg-rel} \rightarrow \langle Rv \rangle \text{slg-rel}) \rightarrow$
 $\langle Rm, Rv \rangle g\text{-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle g\text{-impl-rel-ext}$
 $\langle \text{proof} \rangle$

lemma [*autoref-rules*]: $(gi\text{-V0-update}, g\text{-V0-update}) \in (\langle Rv \rangle \text{list-set-rel} \rightarrow \langle Rv \rangle \text{list-set-rel})$
 \rightarrow
 $\langle Rm, Rv \rangle g\text{-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle g\text{-impl-rel-ext}$
 $\langle \text{proof} \rangle$

lemma [autoref-hom]:

CONSTRAINT graph-rec-ext ($\langle Rv \rangle Rvs \rightarrow \langle Rv \rangle Res \rightarrow \langle Rv \rangle Rv0s \rightarrow Rm \rightarrow \langle Rm, Rv \rangle Rg$)
 <proof>

schematic-goal (?c::?'c, $\lambda G x. g-E G \{x\} \in ?R$)
 <proof>

schematic-goal (?c, $\lambda V0 E.$
 ($\lfloor g-V = UNIV, g-E = E, g-V0 = V0 \rfloor$)
 $\in \langle R \rangle list-set-rel \rightarrow \langle R \rangle slg-rel \rightarrow \langle unit-rel, R \rangle g-impl-rel-ext$
 <proof>

schematic-goal (?c, $\lambda V V0 E.$
 ($\lfloor g-V = V, g-E = E, g-V0 = V0 \rfloor$)
 $\in \langle R \rangle list-set-rel \rightarrow \langle R \rangle list-set-rel \rightarrow \langle R \rangle slg-rel \rightarrow \langle unit-rel, R \rangle frgv-impl-rel-ext$
 <proof>

7.2.5 Renaming

definition the-inv-into-map $V f x$
 = (if $x \in f'V$ then Some (the-inv-into $V f x$) else None)

lemma the-inv-into-map-None[simp]:
 the-inv-into-map $V f x = None \iff x \notin f'V$
 <proof>

lemma the-inv-into-map-Some':
 the-inv-into-map $V f x = Some y \iff x \in f'V \wedge y = the-inv-into V f x$
 <proof>

lemma the-inv-into-map-Some[simp]:
 inj-on $f V \implies the-inv-into-map V f x = Some y \iff y \in V \wedge x = f y$
 <proof>

definition the-inv-into-map-impl $V f =$
 FOREACH $V (\lambda x m. RETURN (m(f x \mapsto x))) Map.empty$

lemma the-inv-into-map-impl-correct:
 assumes [simp]: finite V
 assumes INJ: inj-on $f V$
 shows the-inv-into-map-impl $V f \leq SPEC (\lambda r. r = the-inv-into-map V f)$
 <proof>

schematic-goal the-inv-into-map-code-aux:
 fixes $Rv' :: ('vti \times 'vt) set$
 assumes [autoref-ga-rules]: is-bounded-hashcode Rv' eq bhc

```

assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE('vti) (def-size)
assumes [autoref-rules]: (Vi, V) ∈ ⟨Rv⟩list-set-rel
assumes [autoref-rules]: (fi, f) ∈ Rv → Rv'
shows (RETURN ?c, the-inv-into-map-impl V f) ∈ ⟨⟨Rv', Rv⟩ahm-rel bhc⟩nres-rel
⟨proof⟩

```

concrete-definition *the-inv-into-map-code* **uses** *the-inv-into-map-code-aux*
export-code *the-inv-into-map-code* **checking** *SML*

thm *the-inv-into-map-code.refine*

context begin interpretation *autoref-syn* ⟨proof⟩

lemma *autoref-the-inv-into-map*[autoref-rules]:

fixes *Rv'* :: ('vti × 'vt) set

assumes *SIDE-GEN-ALGO* (is-bounded-hashcode *Rv'* eq bhc)

assumes *SIDE-GEN-ALGO* (is-valid-def-hm-size TYPE('vti) def-size)

assumes *INJ*: *SIDE-PRECOND* (inj-on f V)

assumes *V*: (Vi, V) ∈ ⟨Rv⟩list-set-rel

assumes *F*: (fi, f) ∈ Rv → Rv'

shows (the-inv-into-map-code eq bhc def-size Vi fi,

(OP the-inv-into-map

∴ ⟨Rv⟩list-set-rel → (Rv → Rv') → ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel

bhc)

\$V\$f) ∈ ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel bhc

⟨proof⟩

end

schematic-goal (?c::?'c, do {

let s = {1, 2, 3::nat};

~~ASSERT (inj-on f (g-V G));~~

RETURN (the-inv-into-map s Suc) }) ∈ ?R

⟨proof⟩

definition *fr-rename-ext-aimpl ecnv f G* ≡ do {

ASSERT (inj-on f (g-V G));

ASSERT (inj-on f (g-V0 G));

let fi-map = the-inv-into-map (g-V G) f;

e ← ecnv fi-map G;

RETURN (

g-V = f'(g-V G),

g-E = (E-of-succ (λv. case fi-map v of

Some u ⇒ f '(succ-of-E (g-E G) u) | None ⇒ {})),

g-V0 = (f'g-V0 G),

... = e

)

}

context *g-rename-precond* **begin**

definition *fi-map* $x = (\text{if } x \in f^{\cdot}V \text{ then } \text{Some } (fi\ x) \text{ else } \text{None})$

lemma *fi-map-alt*: *fi-map* = *the-inv-into-map* $V\ f$
<proof>

lemma *fi-map-Some*: $(fi\text{-map } u = \text{Some } v) \longleftrightarrow u \in f^{\cdot}V \wedge fi\ u = v$
<proof>

lemma *fi-map-None*: $(fi\text{-map } u = \text{None}) \longleftrightarrow u \notin f^{\cdot}V$
<proof>

lemma *rename-E-aimpl-alt*: $rename\text{-E } f\ E = E\text{-of-succ } (\lambda v. \text{case } fi\text{-map } v \text{ of } \text{Some } u \Rightarrow f^{\cdot} (\text{succ-of-E } E\ u) \mid \text{None} \Rightarrow \{\})$
<proof>

lemma *frv-rename-ext-aimpl-alt*:
assumes *ECNV*: $ecnv' fi\text{-map } G \leq SPEC (\lambda r. r = ecnv\ G)$
shows *fr-rename-ext-aimpl* $ecnv' f\ G$
 $\leq SPEC (\lambda r. r = fr\text{-rename-ext } ecnv\ f\ G)$
<proof>

end

term *frv-rename-ext-aimpl*

schematic-goal *fr-rename-ext-impl-aux*:

fixes *Re* and *Rv'* :: $('vti \times 'vt)$ *set*

assumes [*autoref-rules*]: $(eq, (=)) \in Rv' \rightarrow Rv' \rightarrow \text{bool-rel}$

assumes [*autoref-ga-rules*]: *is-bounded-hashcode* Rv' *eq* *bhc*

assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('vti)$ *def-size*

shows $(?c, fr\text{-rename-ext-aimpl}) \in$

$((\langle Rv', Rv \rangle ahm\text{-rel } bhc) \rightarrow \langle Re, Rv \rangle frgv\text{-impl-rel-ext} \rightarrow \langle Re' \rangle nres\text{-rel}) \rightarrow$

$(Rv \rightarrow Rv') \rightarrow$

$\langle Re, Rv \rangle frgv\text{-impl-rel-ext} \rightarrow$

$\langle \langle Re', Rv' \rangle frgv\text{-impl-rel-ext} \rangle nres\text{-rel}$

<proof>

concrete-definition *fr-rename-ext-impl* **uses** *fr-rename-ext-impl-aux*

thm *fr-rename-ext-impl.refine*[*OF GEN-OP-D SIDE-GEN-ALGO-D SIDE-GEN-ALGO-D*]

7.3 Graphs from Lists

definition *succ-of-list* :: $(nat \times nat)$ *list* $\Rightarrow nat \Rightarrow nat$ *set*

where

succ-of-list $l \equiv let$

$m = fold (\lambda(u, v) g.$

$\text{case } g\ u \text{ of}$

$\text{None} \Rightarrow g(u \rightarrow \{v\})$

```

      | Some s ⇒ g(w→insert v s)
    ) l Map.empty
in
  (λu. case m u of None ⇒ {} | Some s ⇒ s)

```

lemma *succ-of-list-correct-aux*:
 (succ-of-list l, set l) ∈ br (λsuccs. {(u,v). v∈succs u}) (λ-. True)
 ⟨proof⟩

schematic-goal *succ-of-list-impl*:
notes [autoref-tyrel] =
 ty-REL[where 'a=nat→nat set and R=⟨nat-rel,R⟩iam-map-rel for R]
 ty-REL[where 'a=nat set and R=⟨nat-rel⟩list-set-rel]

shows (?f::?'c,succ-of-list) ∈ ?R
 ⟨proof⟩

concrete-definition *succ-of-list-impl uses succ-of-list-impl*
export-code *succ-of-list-impl in SML*

lemma *succ-of-list-impl-correct*: (succ-of-list-impl,set) ∈ Id → ⟨Id⟩slg-rel
 ⟨proof⟩

end

8 Implementing Automata

theory *Automata-Impl*
imports *Digraph-Impl Automata*
begin

8.1 Indexed Generalized Buchi Graphs

consts
i-igbg-eext :: interface ⇒ interface ⇒ interface

abbreviation *i-igbg* Ie Iv ≡ ⟨⟨Ie,Iv⟩_i,i-igbg-eext,Iv⟩_i,i-g-ext

context begin interpretation *autoref-syn* ⟨proof⟩

lemma *igbg-type[autoref-itype]*:
igbg-num-acc ::_i i-igbg Ie Iv →_i i-nat
igbg-acc ::_i i-igbg Ie Iv →_i Iv →_i ⟨i-nat⟩_i,i-set
igbg-graph-rec-ext
 ::_i i-nat →_i (Iv →_i ⟨i-nat⟩_i,i-set) →_i Ie →_i ⟨Ie,Iv⟩_i,i-igbg-eext
 ⟨proof⟩

end

record (*'vi,'ei,'v0i,'acci*) *gen-igbg-impl* = (*'vi,'ei,'v0i*) *gen-g-impl* +
igbgi-num-acc :: *nat*
igbgi-acc :: *'acci*

definition *gen-igbg-impl-rel-eext-def-internal*:

gen-igbg-impl-rel-eext *Rm Racc* ≡ { (
 ⟨ *igbgi-num-acc* = *num-acci*, *igbgi-acc* = *acci*, ... = *mi* ⟩,
 ⟨ *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc*, ... = *m* ⟩)
 | *num-acci acci mi num-acc acc m*.
 (*num-acci,num-acc*) ∈ *nat-rel*
 ∧ (*acci,acc*) ∈ *Racc*
 ∧ (*mi,m*) ∈ *Rm*
 }

lemma *gen-igbg-impl-rel-eext-def*:

⟨ *Rm,Racc* ⟩ *gen-igbg-impl-rel-eext* = { (
 ⟨ *igbgi-num-acc* = *num-acci*, *igbgi-acc* = *acci*, ... = *mi* ⟩,
 ⟨ *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc*, ... = *m* ⟩)
 | *num-acci acci mi num-acc acc m*.
 (*num-acci,num-acc*) ∈ *nat-rel*
 ∧ (*acci,acc*) ∈ *Racc*
 ∧ (*mi,m*) ∈ *Rm*
 }
 ⟨ *proof* ⟩

lemma *gen-igbg-impl-rel-sv[relator-props]*:

[[*single-valued Racc*; *single-valued Rm*]]
 ⇒ *single-valued* (⟨ *Rm,Racc* ⟩ *gen-igbg-impl-rel-eext*)
 ⟨ *proof* ⟩

abbreviation *gen-igbg-impl-rel-ext*

:: - ⇒ - ⇒ - ⇒ - ⇒ (-×(-, -)) *igbg-graph-rec-scheme* *set*
where *gen-igbg-impl-rel-ext* *Rm Racc*
 ≡ ⟨⟨ *Rm,Racc* ⟩ *gen-igbg-impl-rel-eext*⟩ *gen-g-impl-rel-ext*

lemma *gen-igbg-refine*:

fixes *Rv Re Rv0 Racc*
assumes *TERM* (*Rv,Re,Rv0*)
assumes *TERM* (*Racc*)
shows
 (*igbgi-num-acc,igbg-num-acc*)
 ∈ ⟨ *Rv,Re,Rv0* ⟩ *gen-igbg-impl-rel-ext* *Rm Racc* → *nat-rel*
 (*igbgi-acc,igbg-acc*)
 ∈ ⟨ *Rv,Re,Rv0* ⟩ *gen-igbg-impl-rel-ext* *Rm Racc* → *Racc*
 (*gen-igbg-impl-ext, igbg-graph-rec-ext*)
 ∈ *nat-rel* → *Racc* → *Rm* → ⟨ *Rm,Racc* ⟩ *gen-igbg-impl-rel-eext*
 ⟨ *proof* ⟩

8.1.1 Implementation with bit-set

definition *igbg-impl-rel-eext-internal-def*:

$igbg\text{-}impl\text{-}rel\text{-}eext\ Rm\ Rv \equiv \langle Rm, Rv \rightarrow \langle nat\text{-}rel \rangle bs\text{-}set\text{-}rel \rangle gen\text{-}igbg\text{-}impl\text{-}rel\text{-}eext$

lemma *igbg-impl-rel-eext-def*:

$\langle Rm, Rv \rangle igbg\text{-}impl\text{-}rel\text{-}eext \equiv \langle Rm, Rv \rightarrow \langle nat\text{-}rel \rangle bs\text{-}set\text{-}rel \rangle gen\text{-}igbg\text{-}impl\text{-}rel\text{-}eext$
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of igbg-impl-rel-eext i-igbg-eext*]

lemma [*relator-props, simp*]:

$\llbracket Range\ Rv = UNIV; single\text{-}valued\ Rm \rrbracket$
 $\implies single\text{-}valued\ (\langle Rm, Rv \rangle igbg\text{-}impl\text{-}rel\text{-}eext)$
 $\langle proof \rangle$

lemma *g-tag*: *TERM* ($\langle \langle Rv \rangle fun\text{-}set\text{-}rel, \langle Rv \rangle slg\text{-}rel, \langle Rv \rangle list\text{-}set\text{-}rel \rangle \langle proof \rangle$)

lemma *frgv-tag*: *TERM* ($\langle \langle Rv \rangle list\text{-}set\text{-}rel, \langle Rv \rangle slg\text{-}rel, \langle Rv \rangle list\text{-}set\text{-}rel \rangle \langle proof \rangle$)

lemma *igbg-bs-tag*: *TERM* ($Rv \rightarrow \langle nat\text{-}rel \rangle bs\text{-}set\text{-}rel$) $\langle proof \rangle$

abbreviation *igbgv-impl-rel-ext Rm Rv*

$\equiv \langle \langle Rm, Rv \rangle igbg\text{-}impl\text{-}rel\text{-}eext, Rv \rangle frgv\text{-}impl\text{-}rel\text{-}ext$

abbreviation *igbg-impl-rel-ext Rm Rv*

$\equiv \langle \langle Rm, Rv \rangle igbg\text{-}impl\text{-}rel\text{-}eext, Rv \rangle g\text{-}impl\text{-}rel\text{-}ext$

type-synonym (*'v, 'm*) *igbgv-impl-scheme* =

$('v, (\ \langle igbg\text{-}num\text{-}acc::nat, igbg\text{-}acc::'v \Rightarrow integer, \dots::'m \ \rangle))$
 $frgv\text{-}impl\text{-}scheme$

type-synonym (*'v, 'm*) *igbg-impl-scheme* =

$('v, (\ \langle igbg\text{-}num\text{-}acc::nat, igbg\text{-}acc::'v \Rightarrow integer, \dots::'m \ \rangle))$
 $g\text{-}impl\text{-}scheme$

context fixes *Rv :: ('vi × 'v) set begin*

lemmas [*autoref-rules*] = *gen-igbg-refine*[

OF frgv-tag[of Rv] igbg-bs-tag[of Rv],
folded frgv-impl-rel-ext-def igbg-impl-rel-eext-def]

lemmas [*autoref-rules*] = *gen-igbg-refine*[

OF g-tag[of Rv] igbg-bs-tag[of Rv],
folded g-impl-rel-ext-def igbg-impl-rel-eext-def]

end

schematic-goal (*?c::?'c,*

$\lambda G\ x.\ if\ igbg\text{-}num\text{-}acc\ G = 0 \wedge 1 \in igbg\text{-}acc\ G\ x\ then\ (g\text{-}E\ G\ \{\{x\}\})\ else\ \{\}$
 $\}) \in ?R$
 $\langle proof \rangle$

schematic-goal (?c,
 $\lambda V0 E \text{ num-acc } \text{acc}.$
 $\langle \langle g-V = UNIV, g-E = E, g-V0 = V0, \text{igbg-num-acc} = \text{num-acc}, \text{igbg-acc} = \text{acc} \rangle \rangle$
 $\rangle \in \langle R \rangle \text{list-set-rel} \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \text{nat-rel} \rightarrow (R \rightarrow \langle \text{nat-rel} \rangle \text{bs-set-rel})$
 $\rightarrow \text{igbg-impl-rel-ext unit-rel } R$
 $\langle \text{proof} \rangle$

schematic-goal (?c,
 $\lambda V0 E \text{ num-acc } \text{acc}.$
 $\langle \langle g-V = \{\}, g-E = E, g-V0 = V0, \text{igbg-num-acc} = \text{num-acc}, \text{igbg-acc} = \text{acc} \rangle \rangle$
 $\rangle \in \langle R \rangle \text{list-set-rel} \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \text{nat-rel} \rightarrow (R \rightarrow \langle \text{nat-rel} \rangle \text{bs-set-rel})$
 $\rightarrow \text{igbgv-impl-rel-ext unit-rel } R$
 $\langle \text{proof} \rangle$

8.2 Indexed Generalized Buchi Automata

consts

$i\text{-igba-eext} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

abbreviation $i\text{-igba } Ie Iv Il$

$\equiv \langle \langle \langle Ie, Iv, Il \rangle_i i\text{-igba-eext}, Iv \rangle_i i\text{-igbg-eext}, Iv \rangle_i i\text{-g-ext}$

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

lemma $igba\text{-type}[\text{autoref-itype}]$:

$igba-L ::_i i\text{-igba } Ie Iv Il \rightarrow_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool})$

$igba\text{-rec-ext} ::_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool}) \rightarrow_i Ie \rightarrow_i \langle Ie, Iv, Il \rangle_i i\text{-igba-eext}$

$\langle \text{proof} \rangle$

end

record $(\text{'vi}, \text{'ei}, \text{'v0i}, \text{'acci}, \text{'Li}) \text{ gen-igba-impl} =$

$(\text{'vi}, \text{'ei}, \text{'v0i}, \text{'acci}) \text{ gen-igbg-impl} +$

$igbai-L :: \text{'Li}$

definition $\text{gen-igba-impl-rel-eext-def-internal}$:

$\text{gen-igba-impl-rel-eext } Rm Rl \equiv \{ ($

$\langle \langle \text{igbai-L} = Li, \dots = mi \rangle \rangle,$

$\langle \langle \text{igba-L} = L, \dots = m \rangle \rangle)$

$| Li mi L m.$

$(Li, L) \in Rl$

$\wedge (mi, m) \in Rm$

$\}$

lemma $\text{gen-igba-impl-rel-eext-def}$:

$\langle Rm, Rl \rangle \text{ gen-igba-impl-rel-eext} = \{ ($

$\langle \langle \text{igbai-L} = Li, \dots = mi \rangle \rangle,$

$\langle \langle \text{igba-L} = L, \dots = m \rangle \rangle)$

| $Li\ mi\ L\ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$
 $\langle proof \rangle$

lemma *gen-igba-impl-rel-sv*[*relator-props*]:
 $\llbracket \text{single-valued } Rl; \text{ single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rl \rangle \text{gen-igba-impl-rel-eext})$
 $\langle proof \rangle$

abbreviation *gen-igba-impl-rel-ext*
 $:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('a, 'b, 'c) \text{igba-rec-scheme}) \text{ set}$
where *gen-igba-impl-rel-ext* $Rm\ Rl$
 $\equiv \text{gen-igbg-impl-rel-ext } (\langle Rm, Rl \rangle \text{gen-igba-impl-rel-eext})$

lemma *gen-igba-refine*:
fixes $Rv\ Re\ Rv0\ Racc\ Rl$
assumes $TERM\ (Rv, Re, Rv0)$
assumes $TERM\ (Racc)$
assumes $TERM\ (Rl)$
shows
 $(\text{igbai-L}, \text{igba-L})$
 $\in \langle Rv, Re, Rv0 \rangle \text{gen-igba-impl-rel-ext } Rm\ Rl\ Racc \rightarrow Rl$
 $(\text{gen-igba-impl-ext}, \text{igba-rec-ext})$
 $\in Rl \rightarrow Rm \rightarrow \langle Rm, Rl \rangle \text{gen-igba-impl-rel-eext}$
 $\langle proof \rangle$

8.2.1 Implementation as function

definition *igba-impl-rel-eext-internal-def*:
 $\text{igba-impl-rel-eext } Rm\ Rv\ Rl \equiv \langle Rm, Rv \rightarrow Rl \rightarrow \text{bool-rel} \rangle \text{gen-igba-impl-rel-eext}$

lemma *igba-impl-rel-eext-def*:
 $\langle Rm, Rv, Rl \rangle \text{igba-impl-rel-eext} \equiv \langle Rm, Rv \rightarrow Rl \rightarrow \text{bool-rel} \rangle \text{gen-igba-impl-rel-eext}$
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = $REL-INTFI[\text{of } \text{igba-impl-rel-eext } i\text{-igba-eext}]$

lemma [*relator-props, simp*]:
 $\llbracket \text{Range } Rv = UNIV; \text{ single-valued } Rm; \text{ Range } Rl = UNIV \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv, Rl \rangle \text{igba-impl-rel-eext})$
 $\langle proof \rangle$

lemma *igba-f-tag*: $TERM\ (Rv \rightarrow Rl \rightarrow \text{bool-rel}) \langle proof \rangle$

abbreviation *igbav-impl-rel-ext* $Rm\ Rv\ Rl$
 $\equiv \text{igbgv-impl-rel-ext } (\langle Rm, Rv, Rl \rangle \text{igba-impl-rel-eext})\ Rv$

abbreviation *igba-impl-rel-ext* *Rm Rv Rl*
 $\equiv \text{igbg-impl-rel-ext } (\langle Rm, Rv, Rl \rangle \text{igba-impl-rel-ext}) Rv$

type-synonym (*'v, 'l, 'm*) *igbav-impl-scheme* =
 (*'v, (| igbai-L :: 'v \Rightarrow 'l \Rightarrow bool , ...::'m)*)
igbgv-impl-scheme

type-synonym (*'v, 'l, 'm*) *igba-impl-scheme* =
 (*'v, (| igbai-L :: 'v \Rightarrow 'l \Rightarrow bool , ...::'m)*)
igbg-impl-scheme

context

fixes *Rv* :: (*'vi \times 'v*) *set*

fixes *Rl* :: (*'Li \times 'l*) *set*

begin

lemmas [*autoref-rules*] = *gen-igba-refine*[
OF frgv-tag[of Rv] igbg-bs-tag[of Rv] igba-f-tag[of Rv Rl],
folded frgv-impl-rel-ext-def igbg-impl-rel-ext-def igba-impl-rel-ext-def]

lemmas [*autoref-rules*] = *gen-igba-refine*[
OF g-tag[of Rv] igbg-bs-tag[of Rv] igba-f-tag[of Rv Rl],
folded g-impl-rel-ext-def igbg-impl-rel-ext-def igba-impl-rel-ext-def]

end

thm *autoref-itype*

schematic-goal

(*?c::?'c, $\lambda G x l$. if igba-L $G x l$ then (g-E $G \{x\}$) else {}*) $\in ?R$
<proof>

schematic-goal

notes [*autoref-tyrel*] = *TYRELI[of Id :: ('a \times 'a) set]*
shows (*?c::?'c, $\lambda E (V0::'a set)$ num-acc acc L.*
 (*| g-V = UNIV, g-E = E, g-V0 = V0,*
igbg-num-acc = num-acc, igbg-acc = acc, igba-L = L)
) $\in ?R$
<proof>

schematic-goal

notes [*autoref-tyrel*] = *TYRELI[of Id :: ('a \times 'a) set]*
shows (*?c::?'c, $\lambda E (V0::'a set)$ num-acc acc L.*
 (*| g-V = V0, g-E = E, g-V0 = V0,*
igbg-num-acc = num-acc, igbg-acc = acc, igba-L = L)
) $\in ?R$
<proof>

8.3 Generalized Buchi Graphs

consts

$i\text{-gbg-eeext} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

abbreviation $i\text{-gbg } Ie \ Iv \equiv \langle \langle Ie, Iv \rangle_i i\text{-gbg-eeext}, Iv \rangle_i i\text{-g-ext}$

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

lemma $\text{gbg-type}[\text{autoref-itype}]$:

$\text{gbg-F} ::_i i\text{-gbg } Ie \ Iv \rightarrow_i \langle \langle Iv \rangle_i i\text{-set} \rangle_i i\text{-set}$

$\text{gb-graph-rec-ext} ::_i \langle \langle Iv \rangle_i i\text{-set} \rangle_i i\text{-set} \rightarrow_i Ie \rightarrow_i \langle Ie, Iv \rangle_i i\text{-gbg-eeext}$
 $\langle \text{proof} \rangle$

end

record $(\text{'vi}, \text{'ei}, \text{'v0i}, \text{'fi}) \text{ gen-gbg-impl} = (\text{'vi}, \text{'ei}, \text{'v0i}) \text{ gen-g-impl} +$
 $\text{gbgi-F} :: \text{'fi}$

definition $\text{gen-gbg-impl-rel-eeext-def-internal}$:

$\text{gen-gbg-impl-rel-eeext } Rm \ Rf \equiv \{ ($
 $\langle \text{gbgi-F} = Fi, \dots = mi \rangle,$
 $\langle \text{gbg-F} = F, \dots = m \rangle)$
 $| \text{Fi } mi \ F \ m.$
 $(Fi, F) \in Rf$
 $\wedge (mi, m) \in Rm$
 $\}$

lemma $\text{gen-gbg-impl-rel-eeext-def}$:

$\langle Rm, Rf \rangle \text{ gen-gbg-impl-rel-eeext} = \{ ($
 $\langle \text{gbgi-F} = Fi, \dots = mi \rangle,$
 $\langle \text{gbg-F} = F, \dots = m \rangle)$
 $| \text{Fi } mi \ F \ m.$
 $(Fi, F) \in Rf$
 $\wedge (mi, m) \in Rm$
 $\}$
 $\langle \text{proof} \rangle$

lemma $\text{gen-gbg-impl-rel-sv}[\text{relator-props}]$:

$\llbracket \text{single-valued } Rm; \text{ single-valued } Rf \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rf \rangle \text{ gen-gbg-impl-rel-eeext})$
 $\langle \text{proof} \rangle$

abbreviation $\text{gen-gbg-impl-rel-ext}$

$:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times (\text{'q}, -) \text{ gb-graph-rec-scheme}) \text{ set}$

where $\text{gen-gbg-impl-rel-ext } Rm \ Rf$

$\equiv \langle \langle Rm, Rf \rangle \text{ gen-gbg-impl-rel-eeext} \rangle \text{ gen-g-impl-rel-ext}$

lemma gen-gbg-refine :

fixes $Rv \ Re \ Rv0 \ Rf$

assumes $\text{TERM } (Rv, Re, Rv0)$

assumes $\text{TERM } (Rf)$

shows

$(gbgi-F, gbgi-F)$
 $\in \langle Rv, Re, Rv0 \rangle gen-gbg-impl-rel-ext Rm Rf \rightarrow Rf$
 $(gen-gbg-impl-ext, gb-graph-rec-ext)$
 $\in Rf \rightarrow Rm \rightarrow \langle Rm, Rf \rangle gen-gbg-impl-rel-ext$
 $\langle proof \rangle$

8.3.1 Implementation with list of lists

definition *gbg-impl-rel-ext-internal-def*:

$gbg-impl-rel-ext Rm Rv$
 $\equiv \langle Rm, \langle \langle Rv \rangle list-set-rel \rangle list-set-rel \rangle gen-gbg-impl-rel-ext$

lemma *gbg-impl-rel-ext-def*:

$\langle Rm, Rv \rangle gbg-impl-rel-ext$
 $\equiv \langle Rm, \langle \langle Rv \rangle list-set-rel \rangle list-set-rel \rangle gen-gbg-impl-rel-ext$
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[of *gbg-impl-rel-ext i-gbg-ext*]

lemma [*relator-props, simp*]:

$\llbracket single-valued Rm; single-valued Rv \rrbracket$
 $\implies single-valued (\langle Rm, Rv \rangle gbg-impl-rel-ext)$
 $\langle proof \rangle$

lemma *gbg-ls-tag*: *TERM* ($\langle \langle Rv \rangle list-set-rel \rangle list-set-rel$) $\langle proof \rangle$

abbreviation *gbgv-impl-rel-ext Rm Rv*

$\equiv \langle \langle Rm, Rv \rangle gbg-impl-rel-ext, Rv \rangle frgv-impl-rel-ext$

abbreviation *gbg-impl-rel-ext Rm Rv*

$\equiv \langle \langle Rm, Rv \rangle gbg-impl-rel-ext, Rv \rangle g-impl-rel-ext$

context fixes *Rv* :: ($'vi \times 'v$) *set begin*

lemmas [*autoref-rules*] = *gen-gbg-refine*[

$OF frgv-tag[of Rv] gbg-ls-tag[of Rv],$
 $folded frgv-impl-rel-ext-def gbg-impl-rel-ext-def]$

lemmas [*autoref-rules*] = *gen-gbg-refine*[

$OF g-tag[of Rv] gbg-ls-tag[of Rv],$
 $folded g-impl-rel-ext-def gbg-impl-rel-ext-def]$

end

schematic-goal ($?c::?'c,$

$\lambda G x. if gbg-F G = \{\}$ then $(g-E G \text{ “ } \{x\}$ else $\{\}$
 $) \in ?R$
 $\langle proof \rangle$

schematic-goal

notes [*autoref-tyrel*] = *TYRELI*[of *Id* :: ($'a \times 'a$) *set*]

shows ($?c::?'c, \lambda E (V0::'a \text{ set}) F.$
 $(\Downarrow g-V = \{\}, g-E = E, g-V0 = V0, gbg-F = F \Downarrow) \in ?R$
 $\langle \text{proof} \rangle$

schematic-goal

notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: ('a × 'a) set]
shows ($?c::?'c, \lambda E (V0::'a \text{ set}) F.$
 $(\Downarrow g-V = UNIV, g-E = E, g-V0 = V0, gbg-F = \text{insert } \{\} F \Downarrow) \in ?R$
 $\langle \text{proof} \rangle$

schematic-goal ($?c::?'c, \text{it-to-sorted-list } (\lambda - . \text{True}) \{1,2::\text{nat}\}) \in ?R$
 $\langle \text{proof} \rangle$

8.4 GBAs

consts

$i\text{-gba}\text{-eext} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

abbreviation *i-gba* *Ie Iv Il*

$\equiv \langle \langle \langle Ie, Iv, Il \rangle_i i\text{-gba}\text{-eext}, Iv \rangle_i i\text{-gbg}\text{-eext}, Iv \rangle_i i\text{-g}\text{-ext}$

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

lemma *gba-type*[*autoref-itype*]:

$gba-L ::_i i\text{-gba } Ie Iv Il \rightarrow_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool})$
 $gba\text{-rec}\text{-ext} ::_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool}) \rightarrow_i Ie \rightarrow_i \langle Ie, Iv, Il \rangle_i i\text{-gba}\text{-eext}$
 $\langle \text{proof} \rangle$

end

record ('*vi*, '*ei*, '*v0i*, '*acci*, '*Li*) *gen-gba-impl* =
 ('*vi*, '*ei*, '*v0i*, '*acci*) *gen-gbg-impl* +
gbai-L :: '*Li*

definition *gen-gba-impl-rel-eext-def-internal*:

$gen\text{-gba}\text{-impl}\text{-rel}\text{-eext } Rm \ Rl \equiv \{ ($
 $(\Downarrow gbai-L = Li, \dots = mi \Downarrow),$
 $(\Downarrow gba-L = L, \dots = m \Downarrow)$
 $| Li \ mi \ L \ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$

lemma *gen-gba-impl-rel-eext-def*:

$\langle Rm, Rl \rangle gen\text{-gba}\text{-impl}\text{-rel}\text{-eext} = \{ ($
 $(\Downarrow gbai-L = Li, \dots = mi \Downarrow),$
 $(\Downarrow gba-L = L, \dots = m \Downarrow)$
 $| Li \ mi \ L \ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$

$\langle proof \rangle$

lemma *gen-gba-impl-rel-sv*[*relator-props*]:
[[*single-valued* *Rl*; *single-valued* *Rm*]]
 \implies *single-valued* ($\langle Rm, Rl \rangle$ *gen-gba-impl-rel-eext*)
 $\langle proof \rangle$

abbreviation *gen-gba-impl-rel-ext*
 $:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('a, 'b, 'c) \textit{gba-rec-scheme}) \textit{set}$
where *gen-gba-impl-rel-ext* *Rm* *Rl*
 \equiv *gen-gbg-impl-rel-ext* ($\langle Rm, Rl \rangle$ *gen-gba-impl-rel-eext*)

lemma *gen-gba-refine*:
fixes *Rv* *Re* *Rv0* *Racc* *Rl*
assumes *TERM* (*Rv, Re, Rv0*)
assumes *TERM* (*Racc*)
assumes *TERM* (*Rl*)
shows
(*gbai-L, gba-L*)
 $\in \langle Rv, Re, Rv0 \rangle$ *gen-gba-impl-rel-ext* *Rm* *Rl* *Racc* \rightarrow *Rl*
(*gen-gba-impl-ext, gba-rec-ext*)
 \in *Rl* \rightarrow *Rm* \rightarrow $\langle Rm, Rl \rangle$ *gen-gba-impl-rel-eext*
 $\langle proof \rangle$

8.4.1 Implementation as function

definition *gba-impl-rel-eext-internal-def*:
gba-impl-rel-eext *Rm* *Rv* *Rl* \equiv $\langle Rm, Rv \rightarrow Rl \rightarrow \textit{bool-rel} \rangle$ *gen-gba-impl-rel-eext*

lemma *gba-impl-rel-eext-def*:
 $\langle Rm, Rv, Rl \rangle$ *gba-impl-rel-eext* \equiv $\langle Rm, Rv \rightarrow Rl \rightarrow \textit{bool-rel} \rangle$ *gen-gba-impl-rel-eext*
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of* *gba-impl-rel-eext* *i-gba-eext*]

lemma [*relator-props, simp*]:
[[*Range* *Rv* = *UNIV*; *single-valued* *Rm*; *Range* *Rl* = *UNIV*]]
 \implies *single-valued* ($\langle Rm, Rv, Rl \rangle$ *gba-impl-rel-eext*)
 $\langle proof \rangle$

lemma *gba-f-tag*: *TERM* (*Rv* \rightarrow *Rl* \rightarrow *bool-rel*) $\langle proof \rangle$

abbreviation *gbav-impl-rel-ext* *Rm* *Rv* *Rl*
 \equiv *gbgv-impl-rel-ext* ($\langle Rm, Rv, Rl \rangle$ *gba-impl-rel-eext*) *Rv*

abbreviation *gba-impl-rel-ext* *Rm* *Rv* *Rl*
 \equiv *gbg-impl-rel-ext* ($\langle Rm, Rv, Rl \rangle$ *gba-impl-rel-eext*) *Rv*

context

```

fixes  $Rv :: ('vi \times 'v) \text{ set}$ 
fixes  $Rl :: ('Li \times 'l) \text{ set}$ 
begin
lemmas [autoref-rules] = gen-gba-refine[
   $OF \text{ frgv-tag[of } Rv] \text{ gbg-ls-tag[of } Rv] \text{ gba-f-tag[of } Rv \text{ } Rl],$ 
   $\text{folded frgv-impl-rel-ext-def gbg-impl-rel-eext-def gba-impl-rel-eext-def}$ ]

lemmas [autoref-rules] = gen-gba-refine[
   $OF \text{ g-tag[of } Rv] \text{ gbg-ls-tag[of } Rv] \text{ gba-f-tag[of } Rv \text{ } Rl],$ 
   $\text{folded g-impl-rel-ext-def gbg-impl-rel-eext-def gba-impl-rel-eext-def}$ ]
end

thm autoref-itype

schematic-goal
   $(?c::?'c, \lambda G \ x \ l. \text{ if gba-L } G \ x \ l \text{ then } (g-E \ G \ \{\{x\}\} \text{ else } \{\}) \in ?R$ 
   $\langle \text{proof} \rangle$ 

schematic-goal
notes [autoref-tyrel] = TYRELI[of  $Id :: ('a \times 'a) \text{ set}$ ]
shows  $(?c::?'c, \lambda E \ (V0::'a \text{ set}) \ F \ L.$ 
   $(\mid g-V = UNIV, g-E = E, g-V0 = V0,$ 
   $\text{gbg-F} = F, \text{gba-L} = L \mid)$ 
   $\rangle \in ?R$ 
   $\langle \text{proof} \rangle$ 

schematic-goal
notes [autoref-tyrel] = TYRELI[of  $Id :: ('a \times 'a) \text{ set}$ ]
shows  $(?c::?'c, \lambda E \ (V0::'a \text{ set}) \ F \ L.$ 
   $(\mid g-V = V0, g-E = E, g-V0 = V0,$ 
   $\text{gbg-F} = F, \text{gba-L} = L \mid)$ 
   $\rangle \in ?R$ 
   $\langle \text{proof} \rangle$ 

## 8.5 Buchi Graphs

consts
   $i\text{-bg-eext} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$ 

abbreviation  $i\text{-bg } Ie \ Iv \equiv \langle \langle Ie, Iv \rangle_i i\text{-bg-eext}, Iv \rangle_i i\text{-g-ext}$ 

context begin interpretation autoref-syn  $\langle \text{proof} \rangle$ 
lemma bg-type[autoref-itype]:
   $bg-F ::_i i\text{-bg } Ie \ Iv \rightarrow_i \langle Iv \rangle_i i\text{-set}$ 
   $gb\text{-graph-rec-ext} ::_i \langle \langle Iv \rangle_i i\text{-set} \rangle_i i\text{-set} \rightarrow_i Ie \rightarrow_i \langle Ie, Iv \rangle_i i\text{-bg-eext}$ 
   $\langle \text{proof} \rangle$ 
end

record  $( 'vi, 'ei, 'v0i, 'fi) \text{ gen-bg-impl} = ( 'vi, 'ei, 'v0i) \text{ gen-g-impl} +$ 

```


$bg\text{-}F :: 'f\text{i}$

definition *gen-bg-impl-rel-eext-def-internal*:

$$\begin{aligned} \text{gen-bg-impl-rel-eext } Rm \text{ } Rf &\equiv \{ (\\ &\langle \langle bg\text{-}F = Fi, \dots = mi \rangle \rangle, \\ &\langle \langle bg\text{-}F = F, \dots = m \rangle \rangle) \\ &| \text{ } Fi \text{ } mi \text{ } F \text{ } m. \\ &\quad (Fi, F) \in Rf \\ &\quad \wedge (mi, m) \in Rm \\ &\} \end{aligned}$$

lemma *gen-bg-impl-rel-eext-def*:

$$\begin{aligned} \langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext} &= \{ (\\ &\langle \langle bg\text{-}F = Fi, \dots = mi \rangle \rangle, \\ &\langle \langle bg\text{-}F = F, \dots = m \rangle \rangle) \\ &| \text{ } Fi \text{ } mi \text{ } F \text{ } m. \\ &\quad (Fi, F) \in Rf \\ &\quad \wedge (mi, m) \in Rm \\ &\} \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *gen-bg-impl-rel-sv[relator-props]*:

$$\begin{aligned} &\llbracket \text{single-valued } Rm; \text{ single-valued } Rf \rrbracket \\ &\implies \text{single-valued } (\langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext}) \\ &\langle \text{proof} \rangle \end{aligned}$$

abbreviation *gen-bg-impl-rel-ext*

$$\begin{aligned} &:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('q, -) \text{ b-graph-rec-scheme}) \text{ set} \\ &\text{where } \text{gen-bg-impl-rel-ext } Rm \text{ } Rf \\ &\equiv \langle \langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext} \rangle \text{gen-g-impl-rel-ext} \end{aligned}$$

lemma *gen-bg-refine*:

$$\begin{aligned} &\text{fixes } Rv \text{ } Re \text{ } Rv0 \text{ } Rf \\ &\text{assumes } \text{TERM } (Rv, Re, Rv0) \\ &\text{assumes } \text{TERM } (Rf) \\ &\text{shows} \\ &\quad (bg\text{-}F, bg\text{-}F) \\ &\quad \in \langle Rv, Re, Rv0 \rangle \text{gen-bg-impl-rel-ext } Rm \text{ } Rf \rightarrow Rf \\ &\quad (\text{gen-bg-impl-ext}, \text{ b-graph-rec-ext}) \\ &\quad \in Rf \rightarrow Rm \rightarrow \langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext} \\ &\langle \text{proof} \rangle \end{aligned}$$

8.5.1 Implementation with Characteristic Functions

definition *bg-impl-rel-eext-internal-def*:

$$\begin{aligned} &bg\text{-impl-rel-eext } Rm \text{ } Rv \\ &\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel} \rangle \text{gen-bg-impl-rel-eext} \end{aligned}$$

lemma *bg-impl-rel-eext-def*:

$\langle Rm, Rv \rangle \text{bg-impl-rel-eext}$
 $\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel} \rangle \text{gen-bg-impl-rel-eext}$
 $\langle \text{proof} \rangle$

lemmas [autoref-rel-intf] = REL-INTFI[of bg-impl-rel-eext i-bg-eext]

lemma [relator-props, simp]:
 $\llbracket \text{single-valued } Rm; \text{single-valued } Rv; \text{Range } Rv = UNIV \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv \rangle \text{bg-impl-rel-eext})$
 $\langle \text{proof} \rangle$

lemma bg-fs-tag: TERM ($\langle Rv \rangle \text{fun-set-rel}$) $\langle \text{proof} \rangle$

abbreviation bgv-impl-rel-ext Rm Rv
 $\equiv \langle \langle Rm, Rv \rangle \text{bg-impl-rel-eext}, Rv \rangle \text{frgv-impl-rel-ext}$

abbreviation bg-impl-rel-ext Rm Rv
 $\equiv \langle \langle Rm, Rv \rangle \text{bg-impl-rel-eext}, Rv \rangle \text{g-impl-rel-ext}$

context fixes Rv :: ('vi × 'v) set **begin**
lemmas [autoref-rules] = gen-bg-refine[
OF frgv-tag[of Rv] bg-fs-tag[of Rv],
folded frgv-impl-rel-ext-def bg-impl-rel-eext-def]

lemmas [autoref-rules] = gen-bg-refine[
OF g-tag[of Rv] bg-fs-tag[of Rv],
folded g-impl-rel-ext-def bg-impl-rel-eext-def]
end

schematic-goal (?c::?'c,
 $\lambda G x. \text{if } x \in \text{bg-F } G \text{ then } (g-E \ G \ \{\!\! \{ x \}\!\! \}) \text{ else } \{\}$
) $\in ?R$
 $\langle \text{proof} \rangle$

schematic-goal
notes [autoref-tyrel] = TYRELI[of Id :: ('a × 'a) set]
shows (?c::?'c, $\lambda E (V0::'a \text{ set}) F.$
 $(\ () \ g-V = \{\}, g-E = E, g-V0 = V0, \text{bg-F} = F \)) \in ?R$
 $\langle \text{proof} \rangle$

schematic-goal
notes [autoref-tyrel] = TYRELI[of Id :: ('a × 'a) set]
shows (?c::?'c, $\lambda E (V0::'a \text{ set}) F.$
 $(\ () \ g-V = UNIV, g-E = E, g-V0 = V0, \text{bg-F} = F \)) \in ?R$
 $\langle \text{proof} \rangle$

8.6 System Automata

consts

$i\text{-sa-eext} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

abbreviation $i\text{-sa } Ie \ Iv \ Il \equiv \langle \langle Ie, Iv, Il \rangle_i i\text{-sa-eext}, Iv \rangle_i i\text{-g-ext}$

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

term $sa\text{-}L$

lemma $sa\text{-}type[autoref\text{-}itype]$:

$sa\text{-}L ::_i i\text{-sa } Ie \ Iv \ Il \rightarrow_i Iv \rightarrow_i Il$

$sa\text{-}rec\text{-}ext ::_i (Iv \rightarrow_i Il) \rightarrow_i Ie \rightarrow_i \langle Ie, Iv, Il \rangle_i i\text{-sa-eext}$

$\langle \text{proof} \rangle$

end

record $(\prime vi, \prime ei, \prime v0i, \prime li)$ $gen\text{-}sa\text{-}impl = (\prime vi, \prime ei, \prime v0i)$ $gen\text{-}g\text{-}impl +$
 $sa\text{-}L :: \prime li$

definition $gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext\text{-}def\text{-}internal$:

$gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext \ Rm \ Rl \equiv \{ ($
 $\langle \langle sa\text{-}L = Li, \dots = mi \rangle \rangle,$
 $\langle \langle sa\text{-}L = L, \dots = m \rangle \rangle$
 $| Li \ mi \ L \ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$

lemma $gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext\text{-}def$:

$\langle Rm, Rl \rangle gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext = \{ ($
 $\langle \langle sa\text{-}L = Li, \dots = mi \rangle \rangle,$
 $\langle \langle sa\text{-}L = L, \dots = m \rangle \rangle$
 $| Li \ mi \ L \ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$
 $\langle \text{proof} \rangle$

lemma $gen\text{-}sa\text{-}impl\text{-}rel\text{-}sv[relator\text{-}props]$:

$\llbracket \text{single-valued } Rm; \text{ single-valued } Rf \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rf \rangle gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext)$
 $\langle \text{proof} \rangle$

abbreviation $gen\text{-}sa\text{-}impl\text{-}rel\text{-}ext$

$:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times (\prime q, \prime l, -)) \text{ sa-rec-scheme} \ \text{set}$

where $gen\text{-}sa\text{-}impl\text{-}rel\text{-}ext \ Rm \ Rf$

$\equiv \langle \langle Rm, Rf \rangle gen\text{-}sa\text{-}impl\text{-}rel\text{-}eext \rangle gen\text{-}g\text{-}impl\text{-}rel\text{-}ext$

lemma $gen\text{-}sa\text{-}refine$:

fixes $Rv \ Re \ Rv0$

assumes $TERM \ (Rv, Re, Rv0)$

assumes $TERM \ (Rl)$

shows

$(sai-L, sa-L)$
 $\in \langle Rv, Re, Rv0 \rangle gen-sa-impl-rel-ext \ Rm \ Rl \rightarrow Rl$
 $(gen-sa-impl-ext, sa-rec-ext)$
 $\in Rl \rightarrow Rm \rightarrow \langle Rm, Rl \rangle gen-sa-impl-rel-ext$
 $\langle proof \rangle$

8.6.1 Implementation with Function

definition *sa-impl-rel-ext-internal-def*:

$sa-impl-rel-ext \ Rm \ Rv \ Rl$
 $\equiv \langle Rm, Rv \rightarrow Rl \rangle gen-sa-impl-rel-ext$

lemma *sa-impl-rel-ext-def*:

$\langle Rm, Rv, Rl \rangle sa-impl-rel-ext$
 $\equiv \langle Rm, Rv \rightarrow Rl \rangle gen-sa-impl-rel-ext$
 $\langle proof \rangle$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of sa-impl-rel-ext i-sa-ext*]

lemma [*relator-props, simp*]:

$\llbracket single-valued \ Rm; \ single-valued \ Rl; \ Range \ Rv = \ UNIV \rrbracket$
 $\implies \ single-valued \ (\langle Rm, Rv, Rl \rangle sa-impl-rel-ext)$
 $\langle proof \rangle$

lemma *sa-f-tag*: *TERM* $(Rv \rightarrow Rl)$ $\langle proof \rangle$

abbreviation *sav-impl-rel-ext* $Rm \ Rv \ Rl$

$\equiv \langle \langle Rm, Rv, Rl \rangle sa-impl-rel-ext, Rv \rangle frgv-impl-rel-ext$

abbreviation *sa-impl-rel-ext* $Rm \ Rv \ Rl$

$\equiv \langle \langle Rm, Rv, Rl \rangle sa-impl-rel-ext, Rv \rangle g-impl-rel-ext$

type-synonym (v, l, m) *sav-impl-scheme* =

$(v, (\ \lvert \ sai-L \ :: \ v \Rightarrow \ l \ , \ \dots \ :: \ m \ \rvert)) \ frgv-impl-scheme$

type-synonym (v, l, m) *sa-impl-scheme* =

$(v, (\ \lvert \ sai-L \ :: \ v \Rightarrow \ l \ , \ \dots \ :: \ m \ \rvert)) \ g-impl-scheme$

context fixes $Rv \ :: \ (v_i \times v)$ *set* **begin**

lemmas [*autoref-rules*] = *gen-sa-refine*[

$OF \ frgv-tag[of \ Rv] \ sa-f-tag[of \ Rv],$
 $folded \ frgv-impl-rel-ext-def \ sa-impl-rel-ext-def]$

lemmas [*autoref-rules*] = *gen-sa-refine*[

$OF \ g-tag[of \ Rv] \ sa-f-tag[of \ Rv],$
 $folded \ g-impl-rel-ext-def \ sa-impl-rel-ext-def]$

end

schematic-goal $(?c::?c,$

$\lambda G x l.$ if $sa-L\ G\ x = l$ then $(g-E\ G\ \{\ x\})$ else $\{\}$
 $\in ?R$
 $\langle proof \rangle$

schematic-goal

notes $[autoref-tyrel] = TYRELI[of\ Id :: ('a \times 'a)\ set]$
shows $(?c :: ?'c, \lambda E (V0 :: 'a\ set)\ L.$
 $(\ | g-V = \{\}, g-E = E, g-V0 = V0, sa-L = L\ |)) \in ?R$
 $\langle proof \rangle$

schematic-goal

notes $[autoref-tyrel] = TYRELI[of\ Id :: ('a \times 'a)\ set]$
shows $(?c :: ?'c, \lambda E (V0 :: 'a\ set)\ L.$
 $(\ | g-V = UNIV, g-E = E, g-V0 = V0, sa-L = L\ |)) \in ?R$
 $\langle proof \rangle$

8.7 Index Conversion

schematic-goal $gbg-to-idx-ext-impl-aux:$

fixes Re and $Rv :: ('qi \times 'q)\ set$
assumes $[autoref-ga-rules]: is-bounded-hashcode\ Rv\ eq\ bhc$
assumes $[autoref-ga-rules]: is-valid-def-hm-size\ TYPE('qi)\ (def-size)$
shows $(?c, gbg-to-idx-ext :: - \Rightarrow ('q, -)\ gb-graph-rec-scheme \Rightarrow -)$
 $\in (gbgv-impl-rel-ext\ Re\ Rv \rightarrow Ri)$
 $\rightarrow gbgv-impl-rel-ext\ Re\ Rv$
 $\rightarrow \langle igbgv-impl-rel-ext\ Ri\ Rv \rangle nres-rel$
 $\langle proof \rangle$

concrete-definition $gbg-to-idx-ext-impl$

for $eq\ bhc\ def-size$ **uses** $gbg-to-idx-ext-impl-aux$

lemmas $[autoref-rules] =$

$gbg-to-idx-ext-impl.refine[$
 $OF\ SIDE-GEN-ALGO-D\ SIDE-GEN-ALGO-D]$

schematic-goal $gbg-to-idx-ext-code-aux:$

$RETURN\ ?c \leq gbg-to-idx-ext-impl\ eq\ bhc\ def-size\ ecnv\ G$
 $\langle proof \rangle$

concrete-definition $gbg-to-idx-ext-code$

for $eq\ bhc\ ecnv\ G$ **uses** $gbg-to-idx-ext-code-aux$

lemmas $[refine-transfer] = gbg-to-idx-ext-code.refine$

term $ahm-rel$

context begin interpretation $autoref-syn\ \langle proof \rangle$

lemma $[autoref-op-pat]: gba-to-idx-ext\ ecnv \equiv OP\ gba-to-idx-ext\ \$\ ecnv\ \langle proof \rangle$

end

schematic-goal $gba-to-idx-ext-impl-aux:$

fixes Re and $Rv :: ('qi \times 'q)\ set$

assumes [autoref-ga-rules]: *is-bounded-hashcode* Rv *eq* bhc
assumes [autoref-ga-rules]: *is-valid-def-hm-size* $TYPE('qi)$ (*def-size*)
shows ($?c$, *gba-to-idx-ext* :: $- \Rightarrow ('q, 'l, -)$ *gba-rec-scheme* $\Rightarrow -$)
 \in (*gbav-impl-rel-ext* Re Rv $Rl \rightarrow Ri$)
 \rightarrow *gbav-impl-rel-ext* Re Rv Rl
 \rightarrow $\langle igbav-impl-rel-ext$ Ri Rv $Rl \rangle nres-rel$
 $\langle proof \rangle$
concrete-definition *gba-to-idx-ext-impl* **for** eq bhc **uses** *gba-to-idx-ext-impl-aux*
lemmas [autoref-rules] =
gba-to-idx-ext-impl.refine[*OF SIDE-GEN-ALGO-D SIDE-GEN-ALGO-D*]

schematic-goal *gba-to-idx-ext-code-aux*:
 $RETURN$ $?c \leq$ *gba-to-idx-ext-impl* eq bhc *def-size* $ecnv$ G
 $\langle proof \rangle$
concrete-definition *gba-to-idx-ext-code* **for** $ecnv$ G **uses** *gba-to-idx-ext-code-aux*
lemmas [*refine-transfer*] = *gba-to-idx-ext-code.refine*

8.8 Degeneralization

context *igb-graph* **begin**

lemma *degen-impl-aux-alt*: *degeneralize-ext* $ecnv$ = (
 if *num-acc* = 0 then (
 $g-V$ = *Collect* ($\lambda(q,x).$ $x=0 \wedge q \in V$),
 $g-E$ = *E-of-succ* ($\lambda(q,x).$ if $x=0$ then $(\lambda q'. (q',0))$ '*succ-of-E* E q else $\{\}$),
 $g-V0$ = $(\lambda q'. (q',0))$ ' $V0$,
 $bg-F$ = *Collect* ($\lambda(q,x).$ $x=0 \wedge q \in V$),
 ... = $ecnv$ G
)
 else (
 $g-V$ = *Collect* ($\lambda(q,x).$ $x < num-acc \wedge q \in V$),
 $g-E$ = *E-of-succ* ($\lambda(q,i).$
 if $i < num-acc$ then
 let
 $i' =$ if $i \in acc$ q then $(i + 1) \bmod num-acc$ else i
 in $(\lambda q'. (q',i'))$ '*succ-of-E* E q
 else $\{\}$
),
 $g-V0$ = $(\lambda q'. (q',0))$ ' $V0$,
 $bg-F$ = *Collect* ($\lambda(q,x).$ $x=0 \wedge 0 \in acc$ q),
 ... = $ecnv$ G
)
 $\langle proof \rangle$

schematic-goal *degeneralize-ext-impl-aux*:
fixes Re Rv
assumes [autoref-rules]: $(Gi, G) \in$ *igbg-impl-rel-ext* Re Rv
shows ($?c$, *degeneralize-ext*)
 \in (*igbg-impl-rel-ext* Re $Rv \rightarrow Re'$) \rightarrow *bg-impl-rel-ext* Re' ($Rv \times_r$ *nat-rel*)

$\langle proof \rangle$
end
definition [*simp*]:
 $op\text{-}igb\text{-}graph\text{-}degeneralize\text{-}ext\ ecnv\ G \equiv igb\text{-}graph.degeneralize\text{-}ext\ G\ ecnv$
lemma [*autoref-op-pat*]:
 $igb\text{-}graph.degeneralize\text{-}ext \equiv \lambda G\ ecnv. op\text{-}igb\text{-}graph\text{-}degeneralize\text{-}ext\ ecnv\ G$
 $\langle proof \rangle$
thm $igb\text{-}graph.degeneralize\text{-}ext\text{-}impl\text{-}aux[param\text{-}fo]$
concrete-definition $degeneralize\text{-}ext\text{-}impl$
uses $igb\text{-}graph.degeneralize\text{-}ext\text{-}impl\text{-}aux[param\text{-}fo]$
thm $degeneralize\text{-}ext\text{-}impl.refine$
context begin interpretation $autoref\text{-}syn\ \langle proof \rangle$
lemma [*autoref-rules*]:
fixes Re
assumes $SIDE\text{-}PRECOND\ (igb\text{-}graph\ G)$
assumes $CNVR: (ecnvi, ecnv) \in (igbg\text{-}impl\text{-}rel\text{-}ext\ Re\ Rv \rightarrow Re')$
assumes $GR: (Gi, G) \in igbg\text{-}impl\text{-}rel\text{-}ext\ Re\ Rv$
shows $(degeneralize\text{-}ext\text{-}impl\ Gi\ ecnvi,$
 $(OP\ op\text{-}igb\text{-}graph\text{-}degeneralize\text{-}ext$
 $::: (igbg\text{-}impl\text{-}rel\text{-}ext\ Re\ Rv \rightarrow Re') \rightarrow igbg\text{-}impl\text{-}rel\text{-}ext\ Re\ Rv$
 $\rightarrow bg\text{-}impl\text{-}rel\text{-}ext\ Re' (Rv \times_r\ nat\text{-}rel)) \$ecnv\$G)$
 $\in bg\text{-}impl\text{-}rel\text{-}ext\ Re' (Rv \times_r\ nat\text{-}rel)$
 $\langle proof \rangle$
end
thm $autoref\text{-}itype(1)$
schematic-goal
assumes [*simp*]: $igb\text{-}graph\ G$
assumes [*autoref-rules*]: $(Gi, G) \in igbg\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ nat\text{-}rel$
shows $(?c::?c', igb\text{-}graph.degeneralize\text{-}ext\ G\ (\lambda\text{-}.\ ())) \in ?R$
 $\langle proof \rangle$

8.9 Product Construction

context $igba\text{-}sys\text{-}prod\text{-}precond\ \mathbf{begin}$

lemma $prod\text{-}impl\text{-}aux\text{-}alt$:

$prod = (\{\}$
 $g\text{-}V = Collect\ (\lambda(q, s). q \in igba.V \wedge s \in sa.V),$
 $g\text{-}E = E\text{-of}\text{-}succ\ (\lambda(q, s).$
 $\text{if } igba.L\ q\ (sa.L\ s)\ \text{then}$
 $\quad succ\text{-of}\text{-}E\ (igba.E)\ q \times succ\text{-of}\text{-}E\ sa.E\ s$

```

      else
        {}
      ),
      g-V0 = igba.V0 × sa.V0,
      igbg-num-acc = igba.num-acc,
      igbg-acc = λ(q,s). if s∈sa.V then igba.acc q else {}
    ))
  ⟨proof⟩

```

schematic-goal *prod-impl-aux*:

fixes *Re*

```

assumes [autoref-rules]: (Gi,G) ∈ igba-impl-rel-ext Re Rq Rl
assumes [autoref-rules]: (Si,S) ∈ sa-impl-rel-ext Re2 Rs Rl
shows (?c, prod) ∈ igbg-impl-rel-ext unit-rel (Rq ×r Rs)
  ⟨proof⟩

```

end

definition [*simp*]: *op-igba-sys-prod* ≡ *igba-sys-prod-precond.prod*

lemma [*autoref-op-pat*]:

```

igba-sys-prod-precond.prod ≡ op-igba-sys-prod
  ⟨proof⟩

```

thm *igba-sys-prod-precond.prod-impl-aux*[*param-fo*]

concrete-definition *igba-sys-prod-impl*

uses *igba-sys-prod-precond.prod-impl-aux*[*param-fo*]

thm *igba-sys-prod-impl.refine*

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma [*autoref-rules*]:

fixes *Re*

assumes *SIDE-PRECOND* (*igba G*)

assumes *SIDE-PRECOND* (*sa S*)

assumes *GR*: (Gi,G)∈*igba-impl-rel-ext unit-rel Rq Rl*

assumes *SR*: (Si,S)∈*sa-impl-rel-ext unit-rel Rs Rl*

shows (*igba-sys-prod-impl Gi Si*,

(*OP op-igba-sys-prod*

∴ *igba-impl-rel-ext unit-rel Rq Rl*

→ *sa-impl-rel-ext unit-rel Rs Rl*

→ *igbg-impl-rel-ext unit-rel (Rq ×_r Rs))\$G\$S)*

∈ *igbg-impl-rel-ext unit-rel (Rq ×_r Rs)*

⟨*proof*⟩

end


```

schematic-goal
  assumes [simp]: igba G sa S
  assumes [autoref-rules]: (Gi,G) $\in$ igba-impl-rel-ext unit-rel Rq Rl
  assumes [autoref-rules]: (Si,S) $\in$ sa-impl-rel-ext unit-rel Rs Rl
  shows (?c::?c',igba-sys-prod-precond.prod G S) $\in$ ?R
  <proof>

end

```