

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	9
1.3	Strongly Connected Components	12
2	Directed Graphs	17
2.1	Directed Graphs with Explicit Node Set and Set of Initial Nodes	17
3	Automata	22
3.1	Generalized Buchi Graphs	22
3.2	Generalized Buchi Automata	25
3.3	Buchi Graphs	27
3.4	Buchi Automata	28
3.5	Indexed acceptance classes	29
3.5.1	Indexing Conversion	32
3.5.2	Degeneralization	37
3.6	System Automata	47
3.6.1	Product Construction	48
4	Lassos	51
4.1	Implementing runs by lassos	61
5	Simulation	62
6	Simulation	62
6.1	Functional Relations	62
6.2	Relation between Runs	63
6.3	Simulation	63
6.4	Bisimulation	66
7	Implementing Graphs	74
7.1	Directed Graphs by Successor Function	75
7.1.1	Restricting Edges	76
7.2	Rooted Graphs	78
7.2.1	Operation Identification Setup	78
7.2.2	Generic Implementation	78
7.2.3	Implementation with list-set for Nodes	79
7.2.4	Implementation with Cfun for Nodes	80
7.2.5	Renaming	81
7.3	Graphs from Lists	85

8	Implementing Automata	86
8.1	Indexed Generalized Buchi Graphs	86
8.1.1	Implementation with bit-set	87
8.2	Indexed Generalized Buchi Automata	89
8.2.1	Implementation as function	90
8.3	Generalized Buchi Graphs	92
8.3.1	Implementation with list of lists	93
8.4	GBAs	94
8.4.1	Implementation as function	96
8.5	Buchi Graphs	97
8.5.1	Implementation with Characteristic Functions	98
8.6	System Automata	99
8.6.1	Implementation with Function	101
8.7	Index Conversion	102
8.8	Degeneralization	103
8.9	Product Construction	105

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:  $(u,v) \in E \implies \text{path } E \ u \ [u] \ v$ 
by (auto intro: path.intros)
```

```
lemma path-empty-conv[simp]:
  path E u [] v  $\longleftrightarrow u=v$ 
by (auto intro: path0 elim: path.cases)
```

```
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  $\longleftrightarrow (u=v \wedge p=[])$ 
by (cases p) (auto simp: path-cons-conv)
```

```
lemma path-conc:
  assumes P1: path E u la v
  assumes P2: path E v lb w
  shows path E u (la@lb) w
  using P1 P2 apply induct
```

by (auto intro: path.intros)

lemma *path-append*:

$\llbracket \text{path } E \ u \ l \ v; (v,w) \in E \rrbracket \implies \text{path } E \ u \ (l@[v]) \ w$
using *path-conc[OF - path1]* .

lemma *path-unconc*:

assumes *path E u (la@lb) w*
obtains *v* **where** *path E u la v* **and** *path E v lb w*
using *assms*
thm *path.induct*
apply (induct u *la@lb w arbitrary: la lb rule: path.induct*)
apply (auto intro: *path.intros elim!: list-Cons-eq-append-cases*)
done

lemma *path-conc-conv*:

$\text{path } E \ u \ (la@lb) \ w \longleftrightarrow (\exists v. \text{path } E \ u \ la \ v \wedge \text{path } E \ v \ lb \ w)$
by (auto intro: *path-conc elim: path-unconc*)

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

by (*simp add: path-cons-conv path-conc-conv*)

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*: $\llbracket (u,v) \in E; (v,w) \in E \rrbracket \implies \text{path } E \ u \ [u] \ v$

by (auto *simp: path-simps*)

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

assumes *path (insert (u,v) E) w p x*
obtains
 $\text{path } E \ w \ p \ x$
| *p1 p2* **where** *path E w p1 u* $\text{path (insert (u,v) E) v p2 x}$
using *assms*
apply *induction*
apply *simp*
apply (*clarsimp*)
apply (*metis path-simps path-cons-conv*)
done

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

assumes *path (insert (u,v) E) w p x*
obtains
 $\text{path } E \ w \ p \ x$
| *p1 p2* **where** *path (insert (u,v) E) w p1 u* $\text{path } E \ v \ p2 \ x$
using *assms*
apply (*induction p arbitrary: x rule: rev-induct*)

```

apply simp
apply (clarsimp simp: path-cons-conv path-conc-conv)
apply (metis path-simps path-append-conv)
done

```

```

lemma path-mono:
  assumes  $S: E \subseteq E'$ 
  assumes  $P: \text{path } E \ u \ p \ v$ 
  shows  $\text{path } E' \ u \ p \ v$ 
  using  $P$ 
  apply induction
  apply simp
  using  $S$ 
  apply (auto simp: path-cons-conv)
  done

```

```

lemma path-is-rtrancl:
  assumes  $\text{path } E \ u \ l \ v$ 
  shows  $(u, v) \in E^*$ 
  using assms
  by induct auto

```

```

lemma rtrancl-is-path:
  assumes  $(u, v) \in E^*$ 
  obtains  $l$  where  $\text{path } E \ u \ l \ v$ 
  using assms
  by induct (auto intro: path0 path-append)

```

```

lemma path-is-trancl:
  assumes  $\text{path } E \ u \ l \ v$ 
  and  $l \neq []$ 
  shows  $(u, v) \in E^+$ 
  using assms
  apply induct
  apply auto  $[]$ 
  apply (case-tac l)
  apply auto
  done

```

```

lemma trancl-is-path:
  assumes  $(u, v) \in E^+$ 
  obtains  $l$  where  $l \neq []$  and  $\text{path } E \ u \ l \ v$ 
  using assms
  by induct (auto intro: path0 path-append)

```

```

lemma path-nth-conv:  $\text{path } E \ u \ p \ v \longleftrightarrow (\text{let } p' = p@[v] \text{ in}$ 
   $u = p'^!0 \wedge$ 
   $(\forall i < \text{length } p' - 1. (p'^!i, p'^!Suc\ i) \in E))$ 

```

```

apply (induct p arbitrary: v rule: rev-induct)
apply (auto simp: path-conc-conv path-cons-conv nth-append)
done

```

```

lemma path-mapI:
  assumes path E u p v
  shows path (pairself f ' E) (f u) (map f p) (f v)
  using assms
  apply induction
  apply (simp)
  apply (force simp: path-cons-conv)
done

```

```

lemma path-restrict:
  assumes path E u p v
  shows path (E ∩ set p × insert v (set (tl p))) u p v
  using assms
proof induction
  print-cases
  case (path-prepend u v p w)
  from path-prepend.IH have path (E ∩ set (u#p) × insert w (set p)) v p w
    apply (rule path-mono[rotated])
    by (cases p) auto
  thus ?case using ⟨(u,v)∈E⟩
    by (cases p) (auto simp add: path-cons-conv)
qed auto

```

```

lemma path-restrict-closed:
  assumes CLOSED: E''D ⊆ D
  assumes I: v∈D and P: path E v p v'
  shows path (E∩D×D) v p v'
  using P CLOSED I
  by induction (auto simp: path-cons-conv)

```

```

lemma path-set-induct:
  assumes path E u p v and u∈I and E''I ⊆ I
  shows set p ⊆ I
  using assms
  by (induction rule: path.induct) auto

```

```

lemma path-nodes-reachable: path E u p v ⇒ insert v (set p) ⊆ E*''{u}
  apply (auto simp: in-set-conv-decomp path-cons-conv path-conc-conv)
  apply (auto dest!: path-is-rtrancl)
done

```

```

lemma path-nodes-edges: path E u p v ⇒ set p ⊆ fst'E
  by (induction rule: path.induct) auto

```

lemma *path-tl-nodes-edges*:
assumes *path E u p v*
shows $set (tl p) \subseteq fst'E \cap snd'E$
proof –
from *path-nodes-edges[OF assms]* **have** $set (tl p) \subseteq fst'E$
by (*cases p*) *auto*

moreover **have** $set (tl p) \subseteq snd'E$
using *assms*
apply (*cases*)
apply *simp*
apply *simp*
apply (*erule path-set-induct[where I = snd'E]*)
apply *auto*
done

ultimately show *?thesis*
by *auto*

qed

lemma *path-loop-shift*:
assumes *P: path E u p u*
assumes *S: v ∈ set p*
obtains *p'* **where** $set p' = set p$ *path E v p' v*
proof –
from *S* **obtain** *p1 p2* **where** [*simp*]: $p = p1 @ v \# p2$ **by** (*auto simp: in-set-conv-decomp*)
from *P* **obtain** *v'* **where** *A: path E u p1 v* $(v, v') \in E$ *path E v' p2 u*
by (*auto simp: path-simps*)
hence *path E v (v \# p2 @ p1) v* **by** (*auto simp: path-simps*)
thus *?thesis* **using** *that[of v \# p2 @ p1]* **by** *auto*

qed

lemma *path-hd*:
assumes $p \neq []$ *path E v p w*
shows $hd p = v$
using *assms*
by (*auto simp: path-cons-conv neq-Nil-conv*)

lemma *path-last-is-edge*:
assumes *path E x p y*
and $p \neq []$
shows $(last p, y) \in E$
using *assms*
by (*auto simp: neq-Nil-rev-conv path-simps*)

lemma *path-member-reach-end*:
assumes *P: path E x p y*
and $v: v \in set p$
shows $(v, y) \in E^+$

using *assms*
by (*auto intro!*: *path-is-trancl simp: in-set-conv-decomp path-simps*)

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 $P x y \wedge (\forall v \in \text{set } (tl\ p). P x v)$

proof –
from $P\ NE$ **have** $p \neq []$ **by** *auto*
thus *?thesis* **using** P
proof (*induction p arbitrary: y rule: rev-nonempty-induct*)
case (*single u*) **hence** $(x,y) \in E$ **by** (*simp add: path-cons-conv*)
with S **show** *?case* **by** *simp*
next
case (*snoc u us*) **hence** *path* E x us u **by** (*simp add: path-append-conv*)
with *snoc path-is-trancl* **have**
 $P x u \quad (x,u) \in E^+ \quad \forall v \in \text{set } (tl\ us). P x v$
by *simp-all*
moreover with *snoc* **have** $\forall v \in \text{set } (tl\ (us@[u])). P x v$ **by** *simp*
moreover from *snoc* **have** $(u,y) \in E$ **by** (*simp add: path-append-conv*)
ultimately show *?case* **by** (*auto intro: ST*)
qed
qed

lemma *path-restrict-tl*:
 $[[\ u \notin R; \text{path } (E \cap UNIV \times -R) \ u \ p \ v \]] \implies \text{path } (\text{rel-restrict } E \ R) \ u \ p \ v$
apply (*induction p arbitrary: u*)
apply (*auto simp: path-simps rel-restrict-def*)
done

lemma *path1-restr-conv*: *path* $(E \cap UNIV \times -R) \ u \ (x \# xs) \ v$
 $\longleftrightarrow (\exists w. w \notin R \wedge x = u \wedge (u,w) \in E \wedge \text{path } (\text{rel-restrict } E \ R) \ w \ xs \ v)$
proof –
have 1 : *rel-restrict* $E \ R \subseteq E \cap UNIV \times -R$ **by** (*auto simp: rel-restrict-def*)

show *?thesis*
by (*auto simp: path-simps intro: path-restrict-tl path-mono[OF 1]*)
qed

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

```

shows path E v c x
using assms
proof (induction arbitrary: w c rule: list-nonempty-induct)
  case (single p) thus ?case by (auto simp add: path-simps)
next
  case (cons p ps) hence [simp]: w = p by (simp add: path-cons-conv)
  show ?case
  proof (cases p=v)
    case True with cons show ?thesis by simp
  next
    case False with cons have c = dropWhile ((≠) v) ps by simp
    moreover from cons.prem1 obtain y where path E y ps x
      using path-uncons by metis
    moreover from cons.prem2 False have v ∈ set ps by simp
    ultimately show ?thesis using cons.IH by metis
  qed
qed

```

```

lemma takeWhileNot-path:
  assumes p ≠ []
  and path E w p x
  and v ∈ set p
  and takeWhile ((≠) v) p = c
  shows path E w c v
  using assms
proof (induction arbitrary: w c rule: list-nonempty-induct)
  case (single p) thus ?case by (auto simp add: path-simps)
next
  case (cons p ps) hence [simp]: w = p by (simp add: path-cons-conv)
  show ?case
  proof (cases p=v)
    case True with cons show ?thesis by simp
  next
    case False with cons obtain c' where
      c' = takeWhile ((≠) v) ps and
      [simp]: c = p#c'
      by simp-all
    moreover from cons.prem1 obtain y where
      path E y ps x and (w,y) ∈ E
      using path-uncons by metis+
    moreover from cons.prem2 False have v ∈ set ps by simp
    ultimately have path E y c' v using cons.IH by metis
    with ⟨(w,y) ∈ E⟩ show ?thesis by (auto simp add: path-cons-conv)
  qed
qed

```

1.2 Infinite Paths

```

definition ipath :: 'q digraph ⇒ 'q word ⇒ bool

```

— Predicate for an infinite path in a digraph
where $ipath\ E\ r \equiv \forall i. (r\ i, r\ (Suc\ i)) \in E$

lemma *ipath-conc-conv*:

$ipath\ E\ (u \frown v) \longleftrightarrow (\exists a. path\ E\ a\ u\ (v\ 0) \wedge ipath\ E\ v)$
apply (*auto simp: conc-def ipath-def path-nth-conv nth-append*)
apply (*metis add-Suc-right diff-add-inverse not-add-less1*)
by (*metis Suc-diff-Suc diff-Suc-Suc not-less-eq*)

lemma *ipath-iter-conv*:

assumes $p \neq []$
shows $ipath\ E\ (p^\omega) \longleftrightarrow (path\ E\ (hd\ p)\ p\ (hd\ p))$
proof (*cases p*)
case Nil thus ?thesis using assms by simp
next
case (Cons u p[^]) hence PLEN: length p > 0 by simp
show ?thesis proof
assume $ipath\ E\ (iter\ (p))$
hence $\forall i. (iter\ (p)\ i, iter\ (p)\ (Suc\ i)) \in E$
unfolding ipath-def by simp
hence $(\forall i < length\ p. (p!\ i, (p@[hd\ p])!\ Suc\ i) \in E)$
apply (*simp add: assms*)
apply safe
apply (*drule-tac x=i in spec*)
apply simp
apply (*case-tac Suc i = length p*)
apply (*simp add: Cons*)
apply (*simp add: nth-append*)
done
thus $path\ E\ (hd\ p)\ p\ (hd\ p)$
by (*auto simp: path-nth-conv Cons nth-append nth-Cons'*)
next
assume $path\ E\ (hd\ p)\ p\ (hd\ p)$
thus $ipath\ E\ (iter\ p)$
apply (*auto simp: path-nth-conv ipath-def assms Let-def*)
apply (*drule-tac x=i mod length p in spec*)
apply (*auto simp: nth-append assms split: if-split-asm*)
apply (*metis less-not-refl mod-Suc*)
by (*metis PLEN diff-self-eq-0 mod-Suc nth-Cons-0 mod-less-divisor*)
qed
qed

lemma *ipath-to-rtrancl*:

assumes $R: ipath\ E\ r$
assumes $I: i1 \leq i2$
shows $(r\ i1, r\ i2) \in E^*$
using I
proof (*induction i2*)

```

case (Suc i2)
show ?case proof (cases i1=Suc i2)
  assume i1≠Suc i2
  with Suc have (r i1,r i2)∈E* by auto
  also from R have (r i2,r (Suc i2))∈E unfolding ipath-def by auto
  finally show ?thesis .
qed simp
qed simp

```

```

lemma ipath-to-trancl:
  assumes R: ipath E r
  assumes I: i1<i2
  shows (r i1,r i2)∈E+
proof -
  from R have (r i1,r (Suc i1))∈E
    by (auto simp: ipath-def)
  also have (r (Suc i1),r i2)∈E*
    using ipath-to-rtrancl[OF R,of Suc i1 i2] I by auto
  finally (rtrancl-into-trancl2) show ?thesis .
qed

```

```

lemma run-limit-two-connectedI:
  assumes A: ipath E r
  assumes B: a ∈ limit r    b∈limit r
  shows (a,b)∈E+
proof -
  from B have {a,b} ⊆ limit r by simp
  with A show ?thesis
    by (metis ipath-to-trancl two-in-limit-iff)
qed

```

```

lemma ipath-subpath:
  assumes P: ipath E r
  assumes LE: l≤u
  shows path E (r l) (map r [l..<u]) (r u)
  using LE
proof (induction u-l arbitrary: u l)
  case (Suc n)
  note IH=Suc.hyps(1)
  from ⟨Suc n = u-l⟩ ⟨l≤u⟩ obtain u' where [simp]: u=Suc u'
    and A: n=u'-l    l ≤ u'
    by (cases u) auto

  note IH[OF A]
  also from P have (r u',r u)∈E
    by (auto simp: ipath-def)
  finally show ?case using ⟨l ≤ u'⟩ by (simp add: upt-Suc-append)
qed auto

```

lemma *ipath-restrict-eq*: $ipath (E \cap (E^* \{r\} \times E^* \{r\})) r \longleftrightarrow ipath E r$
unfolding *ipath-def*
by (*auto simp: relpow-fun-conv rtrancl-power*)
lemma *ipath-restrict*: $ipath E r \implies ipath (E \cap (E^* \{r\} \times E^* \{r\})) r$
by (*simp add: ipath-restrict-eq*)

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

lemma *ipathD*: $ipath E r \implies (r i, r (Suc i)) \in E$
unfolding *ipath-def* **by** *auto*

lemma *ipath-in-Domain*: $ipath E r \implies r i \in Domain E$
unfolding *ipath-def* **by** *auto*

lemma *ipath-in-Range*: $[ipath E r; i \neq 0] \implies r i \in Range E$
unfolding *ipath-def* **by** (*cases i*) *auto*

lemma *ipath-suffix*: $ipath E r \implies ipath E (suffix i r)$
unfolding *suffix-def ipath-def* **by** *auto*

1.3 Strongly Connected Components

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

definition *is-scc* :: $'q \text{ digraph} \Rightarrow 'q \text{ set} \Rightarrow \text{bool}$
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]*: $\neg is-scc E \{ \}$ **unfolding** *is-scc-def* **by** *auto*

lemma *scc-non-empty'[simp]*: $is-scc E U \implies U \neq \{ \}$ **unfolding** *is-scc-def* **by** *auto*

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 $y \in U$

proof –

from *SCC MEM P* **have** $insert y U \times insert y U \subseteq E^*$
unfolding *is-scc-def*
apply *clarsimp*
apply *rule*
apply *clarsimp-all*
apply (*erule disjE1*)
apply *clarsimp*
apply (*metis in-mono mem-Sigma-iff rtrancl-trans*)
apply *auto* \square

apply (*metis in-mono mem-Sigma-iff rtrancl-trans*)
done
with *SCC* **show** *?thesis unfolding is-scc-def by blast*
qed

lemma *is-scc-connected*:
assumes *SCC: is-scc E U*
assumes *MEM: x ∈ U y ∈ U*
shows $(x, y) \in E^*$
using *assms unfolding is-scc-def by auto*

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*:
 $mconn\ E = \{(u, v). (u, v) \in E^* \wedge (v, u) \in E^*\}$
by (*auto simp add: mconn-def rtrancl-converse*)

mconn is an equivalence relation:

lemma *mconn-refl[simp]*: $Id \subseteq mconn\ E$
by (*auto simp add: mconn-def*)

lemma *mconn-sym*: $mconn\ E = (mconn\ E)^{-1}$
by (*auto simp add: mconn-pointwise*)

lemma *mconn-trans*: $mconn\ E\ O\ mconn\ E = mconn\ E$
by (*auto simp add: mconn-def*)

lemma *mconn-refl'*: $refl\ (mconn\ E)$
by (*auto intro: refl-onI simp: mconn-pointwise*)

lemma *mconn-sym'*: $sym\ (mconn\ E)$
by (*auto intro: symI simp: mconn-pointwise*)

lemma *mconn-trans'*: $trans\ (mconn\ E)$
by (*metis mconn-def trans-Int trans-rtrancl*)

lemma *mconn-equiv*: $equiv\ UNIV\ (mconn\ E)$
using *mconn-refl' mconn-sym' mconn-trans'*
by (*rule equivI*)

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

assume $A: is\text{-}scc\ E\ U$
then obtain x **where** $x \in U$ **unfolding** $is\text{-}scc\text{-}def$ **by** $auto$
hence $U = mconn\ E\ \{x\}$ **using** A
unfolding $mconn\text{-}pointwise\ is\text{-}scc\text{-}def$
apply $clarsimp$
apply $rule$
apply $auto\ []$
apply $clarsimp$
by $(metis\ A\ is\text{-}scc\text{-}closed)$
thus $U \in UNIV\ //\ mconn\ E$
by $(auto\ simp:\ quotient\text{-}def)$
next
assume $U \in UNIV\ //\ mconn\ E$
thus $is\text{-}scc\ E\ U$
by $(auto\ simp:\ is\text{-}scc\text{-}def\ mconn\text{-}pointwise\ quotient\text{-}def)$
qed

lemma $is\text{-}scc\ E\ U \iff U \in UNIV\ //\ (E^* \cap (E^{-1})^*)$
unfolding $is\text{-}scc\text{-}mconn\text{-}eqclasses\ mconn\text{-}def$ **by** $simp$

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

lemma $find\text{-}outside\text{-}node$:

assumes $(u, v) \in E^*$
assumes $(u, v) \notin (E \cap U \times U)^*$
assumes $u \in U\ v \in U$
shows $\exists u'. u' \notin U \wedge (u, u') \in E^* \wedge (u', v) \in E^*$
using $assms$
apply $(induction)$
apply $auto\ []$
apply $clarsimp$
by $(metis\ IntI\ mem\text{-}Sigma\text{-}iff\ rtrancl.\text{simps})$

lemma $is\text{-}scc\text{-}restrict1$:

assumes $SCC: is\text{-}scc\ E\ U$
shows $U \times U \subseteq (E \cap U \times U)^*$
using $assms$
unfolding $is\text{-}scc\text{-}def$
apply $clarsimp$
apply $(rule\ ccontr)$
apply $(drule\ (2)\ find\text{-}outside\text{-}node[rotated])$
apply $auto\ []$
by $(metis\ is\text{-}scc\text{-}closed[OF\ SCC]\ mem\text{-}Sigma\text{-}iff\ rtrancl\text{-}trans\ subsetD)$

lemma $is\text{-}scc\text{-}restrict2$:

assumes $SCC: is\text{-}scc\ E\ U$

assumes $V \supset U$
shows $\neg (V \times V \subseteq (E \cap V \times V)^*)$
using *assms*
unfolding *is-scc-def*
apply *clarsimp*
using *rtrancl-mono*[of $E \cap V \times V$ E]
apply *clarsimp*
apply *blast*
done

lemma *is-scc-restrict3*:
assumes *SCC*: *is-scc* E U
shows $((E^* \text{“} ((E^* \text{“} U) - U)) \cap U = \{\})$
apply *auto*
by (*metis assms is-scc-closed is-scc-connected rtrancl-trans*)

lemma *is-scc-alt-restrict-path*:
is-scc E $U \iff U \neq \{\} \wedge$
 $(U \times U \subseteq (E \cap U \times U)^*) \wedge ((E^* \text{“} ((E^* \text{“} U) - U)) \cap U = \{\})$
apply *rule*
apply (*intro conjI*)
apply *simp*
apply (*blast dest: is-scc-restrict1*)
apply (*blast dest: is-scc-restrict3*)

unfolding *is-scc-def*
apply *rule*
apply *clarsimp*
apply (*metis (full-types) Int-lower1 in-mono mem-Sigma-iff rtrancl-mono-mp*)
apply *blast*
done

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
unfolding *is-scc-alt-restrict-path*
by *blast*

lemma *is-scc-unique*:
assumes *SCC*: *is-scc* E *scc* $is-scc$ E *scc'*
and $v \in scc \quad v \in scc'$
shows $scc = scc'$
proof —
from *SCC* **have** $scc = scc' \vee scc \cap scc' = \{\}$
using *quotient-disj*[OF *mconn-equiv*]
by (*simp add: is-scc-mconn-eqclasses*)

with v **show** $?thesis$ **by** $auto$
qed

lemma $is-scc-ex1$:

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

proof ($rule\ ex1I, rule\ conjI$)

let $?scc = mconn\ E\ \{\! \{v\}$

have $?scc \in UNIV // mconn\ E$ **by** ($auto\ intro: quotientI$)

thus $is-scc\ E\ ?scc$ **by** ($simp\ add: is-scc-mconn-eclasses$)

moreover show $v \in ?scc$ **by** ($blast\ intro: refl-onD[OF\ mconn-refl]$)

ultimately show $\bigwedge scc. is-scc\ E\ scc \wedge v \in scc \implies scc = ?scc$

by ($metis\ is-scc-unique$)

qed

lemma $is-scc-ex$:

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

by ($metis\ is-scc-ex1$)

lemma $is-scc-connected'$:

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

unfolding $is-scc-pointwise$

by $blast$

definition $scc-of :: ('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ set$

where

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

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

$is-scc\ E\ (scc-of\ E\ v)$

using $is-scc-ex1[of\ E\ v]$

by ($auto\ dest!: theI'\ simp: scc-of-def$)

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

$v \in scc-of\ E\ v$

using $is-scc-ex1[of\ E\ v]$

by ($auto\ dest!: theI'\ simp: scc-of-def$)

lemma $scc-of-unique$:

assumes $w \in scc-of\ E\ v$

shows $scc-of\ E\ v = scc-of\ E\ w$

proof –

have $is-scc\ E\ (scc-of\ E\ v)$ **by** $simp$

moreover note $assms$

moreover have $is-scc\ E\ (scc-of\ E\ w)$ **by** $simp$

moreover have $w \in scc-of\ E\ w$ **by** $simp$

ultimately show $?thesis$ **using** $is-scc-unique$ **by** $metis$

qed

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  $\Rightarrow$  'v set  $\Rightarrow$  ('v, 'more)
graph-rec-scheme
  where graph-restrict G R  $\equiv$ 
  (
    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
unfolding graph-restrict-def by auto

```

```

lemma graph-restrict-trivial[simp]: graph-restrict G {} = G by simp

```

```

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 by (auto intro: finite-subset)

```

```

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 $run-ipath: is-run\ r \implies ipath\ E\ r$ **unfolding** $is-run-def$ **by** $auto$
lemma $run-V0: is-run\ r \implies r\ 0 \in V0$ **unfolding** $is-run-def$ **by** $auto$

lemma $run-reachable: is-run\ r \implies range\ r \subseteq reachable$
unfolding $is-run-def$ **using** $ipath-to-rtrancl$ **by** $blast$

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 $reachable-V: reachable \subseteq V$ **using** $V0-ss\ E-ss$ **by** $(auto\ elim: rtrancl-induct)$

lemma $finite-E: finite\ V \implies finite\ E$ **using** $finite-subset\ E-ss$ **by** $auto$

end

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

lemma $fb-graph-subset:$
assumes $g-V\ G' = V$
assumes $g-E\ G' \subseteq E$
assumes $finite\ (g-V0\ G')$
assumes $g-V0\ G' \subseteq reachable$
shows $fb-graph\ G'$

proof
show $g-V0\ G' \subseteq g-V\ G'$ **using** $reachable-V\ assms(1, 4)$ **by** $simp$
show $g-E\ G' \subseteq g-V\ G' \times g-V\ G'$ **using** $E-ss\ assms(1, 2)$ **by** $simp$
show $finite\ (g-V0\ G')$ **using** $assms(3)$ **by** $this$

next
fix v
assume $1: v \in (g-E\ G')^*$ “ $g-V0\ G'$
obtain u **where** $2: u \in g-V0\ G'$ $(u, v) \in (g-E\ G')^*$ **using** 1 **by** $rule$
have $3: u \in reachable$ $(u, v) \in E^*$ **using** $rtrancl-mono\ assms(2, 4)$ 2 **by**

auto

have 4: $v \in \text{reachable}$ **using** *rtrancl-image-advance-rtrancl 3* **by** *metis*
have 5: $\text{finite } (E \text{ `` } \{v\})$ **using** 4 **by** *rule*
have 6: $g\text{-}E \ G' \text{ `` } \{v\} \subseteq E \text{ `` } \{v\}$ **using** *assms(2)* **by** *auto*
show $\text{finite } (g\text{-}E \ G' \text{ `` } \{v\})$ **using** *finite-subset 5 6* **by** *auto*
qed

lemma *fb-graph-restrict*: $\text{fb-graph } (\text{graph-restrict } G \ R)$
by (*rule fb-graph-subset, auto simp: rel-restrict-sub*)

end

lemma (*in graph*) *fb-graphI-fr*:

assumes *finite reachable*

shows *fb-graph G*

proof

from *assms* **show** *finite V0* **by** (*rule finite-subset[rotated]*) *auto*

fix v

assume $v \in \text{reachable}$

hence $\text{succ } v \subseteq \text{reachable}$ **by** (*metis Image-singleton-iff rtrancl-image-advance subsetI*)

thus $\text{finite } (\text{succ } v)$ **using** *assms* **by** (*rule finite-subset*)

qed

abbreviation *rename-E f E* $\equiv (\lambda(u,v). (f \ u, f \ v)) \text{ `} E$

definition *fr-rename-ext ecnv f G* $\equiv \langle$

$g\text{-}V = f \text{ `} (g\text{-}V \ G),$

$g\text{-}E = \text{rename-E } f \ (g\text{-}E \ G),$

$g\text{-}V0 = (f \text{ `} g\text{-}V0 \ G),$

$\dots = \text{ecnv } G$

\rangle

locale *g-rename-precond* =

graph G

for $G :: ('u, 'more) \text{ graph-rec-scheme}$

+

fixes $f :: 'u \Rightarrow 'v$

fixes $\text{ecnv} :: ('u, 'more) \text{ graph-rec-scheme} \Rightarrow 'more'$

assumes *INJ: inj-on f V*

begin

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

lemma *G'-fields*:

$g\text{-}V \ G' = f \text{ `} V$

$g\text{-}V0 \ G' = f \text{ `} V0$

$g\text{-}E \ G' = \text{rename-E } f \ E$

unfolding *fr-rename-ext-def* **by** *simp-all*

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

lemma

$fi\text{-}f: x \in V \implies fi (f x) = x$ **and**

$f\text{-}fi: y \in f'V \implies f (fi y) = y$ **and**

$fi\text{-}f\text{-}eq: \llbracket f x = y; x \in V \rrbracket \implies fi y = x$

unfolding $fi\text{-}def$

by (*auto*)

simp: the-inv-into-f-f f-the-inv-into-f the-inv-into-f-eq INJ)

lemma $E'\text{-to-}E: (u, v) \in g\text{-}E G' \implies (fi u, fi v) \in E$

using $E\text{-ss}$

by (*auto simp: fi-f G'-fields*)

lemma $V0'\text{-to-}V0: v \in g\text{-}V0 G' \implies fi v \in V0$

using $V0\text{-ss}$

by (*auto simp: fi-f G'-fields*)

lemma $rtrancl\text{-}E'\text{-sim}$:

assumes $(f u, v') \in (g\text{-}E G')^*$

assumes $u \in V$

shows $\exists v. v' = f v \wedge v \in V \wedge (u, v) \in E^*$

using $assms$

proof (*induction f u v' arbitrary: u*)

case ($rtrancl\text{-into-}rtrancl v' w' u$)

then obtain $v w$ **where** $v' = f v \quad w' = f w \quad (v, w) \in E$

by (*auto simp: G'-fields*)

hence $v \in V \quad w \in V$ **using** $E\text{-ss}$ **by** *auto*

from $rtrancl\text{-into-}rtrancl$ **obtain** vv **where** $v' = f vv \quad vv \in V \quad (u, vv) \in E^*$

by *blast*

from $\langle v' = f v \rangle \langle v \in V \rangle \langle v' = f vv \rangle \langle vv \in V \rangle$ **have** $[simp]: vv = v$

using INJ **by** (*metis inj-on-contrad*)

note $\langle (u, vv) \in E^* \rangle [simplified]$

also note $\langle (v, w) \in E \rangle$

finally show $?case$ **using** $\langle w' = f w \rangle \langle w \in V \rangle$ **by** *blast*

qed *auto*

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

using $assms$ **apply** *induction*

by (*fastforce intro: E'-to-E rtrancl-into-rtrancl*) $+$

lemma $G'\text{-invar}$: *graph* G'

apply *unfold-locales*

proof $-$

show $g\text{-}V0 G' \subseteq g\text{-}V G'$

using $V0\text{-ss}$ **by** (*auto simp: G'-fields*) \square

```

show  $g\text{-}E\ G' \subseteq g\text{-}V\ G' \times g\text{-}V\ G'$ 
  using  $E\text{-}ss$  by (auto simp: G'-fields) []
qed

```

```

sublocale  $G'$ : graph  $G'$  using  $G'\text{-}invar$  .

```

```

lemma  $G'\text{-}finite\text{-}reachable$ :
  assumes  $finite\ ((g\text{-}E\ G')^* \text{ `` } g\text{-}V0\ G')$ 
  shows  $finite\ ((g\text{-}E\ G')^* \text{ `` } g\text{-}V0\ G')$ 
proof -
  have  $(g\text{-}E\ G')^* \text{ `` } g\text{-}V0\ G' \subseteq f\text{ ' } (E^* \text{ `` } V0)$ 
    apply (clarsimp-all simp: G'-fields(2))
    apply (drule rtrancl-E'-sim)
    using  $V0\text{-}ss$  apply auto []
    apply auto
    done
  thus ?thesis using finite-subset assms by blast
qed

```

```

lemma  $V'\text{-}to\text{-}V$ :  $v \in G'.V \implies fi\ v \in V$ 
  by (auto simp: fi-f G'-fields)

```

```

lemma  $ipath\text{-}sim1$ :  $ipath\ E\ r \implies ipath\ G'.E\ (f\ o\ r)$ 
  unfolding  $ipath\text{-}def$  by (auto simp: G'-fields)

```

```

lemma  $ipath\text{-}sim2$ :  $ipath\ G'.E\ r \implies ipath\ E\ (fi\ o\ r)$ 
  unfolding  $ipath\text{-}def$ 
  apply (clarsimp simp: G'-fields)
  apply (drule-tac x=i in spec)
  using  $E\text{-}ss$ 
  by (auto simp: fi-f)

```

```

lemma  $run\text{-}sim1$ :  $is\text{-}run\ r \implies G'.is\text{-}run\ (f\ o\ r)$ 
  unfolding  $is\text{-}run\text{-}def\ G'.is\text{-}run\text{-}def$ 
  apply (intro conjI)
  apply (auto simp: G'-fields) []
  apply (auto simp: ipath-sim1)
  done

```

```

lemma  $run\text{-}sim2$ :  $G'.is\text{-}run\ r \implies is\text{-}run\ (fi\ o\ r)$ 
  unfolding  $is\text{-}run\text{-}def\ G'.is\text{-}run\text{-}def$ 
  by (auto simp: ipath-sim2 V0'-to-V0)

```

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 +
  gbq-F :: 'Q set set
```

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

```
lemma is-gb-graph: gb-graph G by unfold-locales
```

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

unfolding $is-acc-run-def\ is-acc-def$.

lemma $acc-run-run: is-acc-run\ r \implies is-run\ r$

unfolding $is-acc-run-def$ by simp

lemmas $acc-run-reachable = run-reachable[OF\ acc-run-run]$

lemma $acc-eq-limit:$

assumes $FIN: \text{finite } (\text{range } r)$
shows $\text{is-acc } r \longleftrightarrow (\forall A \in F. \text{limit } r \cap A \neq \{\})$
proof
assume $\forall A \in F. \text{limit } r \cap A \neq \{\}$
thus $\text{is-acc } r$
unfolding is-acc-def
by $(\text{metis limit-inter-INF})$
next
from FIN **have** $FIN': \bigwedge A. \text{finite } (A \cap \text{range } r)$
by simp

assume $\text{is-acc } r$
hence $AUX: \forall A \in F. \exists_{\infty} i. r \ i \in (A \cap \text{range } r)$
unfolding is-acc-def **by** auto
have $\forall A \in F. \text{limit } r \cap (A \cap \text{range } r) \neq \{\}$
apply (rule ballI)
apply $(\text{drule bspec}[OF AUX])$
apply $(\text{subst } (\text{asm}) \text{fin-ex-inf-eq-limit}[OF FIN'])$

thus $\forall A \in F. \text{limit } r \cap A \neq \{\}$
by auto
qed

lemma $\text{is-acc-run-limit-alt}$:
assumes $\text{finite } (E^* \text{ “ } V0)$
shows $\text{is-acc-run } r \longleftrightarrow \text{is-run } r \wedge (\forall A \in F. \text{limit } r \cap A \neq \{\})$
using $\text{assms acc-eq-limit[symmetric]}$ **unfolding** is-acc-run-def
by $(\text{auto dest: run-reachable finite-subset})$

lemma $\text{is-acc-suffix[simp]}$: $\text{is-acc } (\text{suffix } i \ r) \longleftrightarrow \text{is-acc } r$
unfolding $\text{is-acc-def suffix-def}$
apply $(\text{clarsimp simp: INFM-nat})$
apply (rule iffI)
apply $(\text{metis trans-less-add2})$
by $(\text{metis add-lessD1 less-imp-add-positive nat-add-left-cancel-less})$

lemma finite-V-Fe :
assumes $\text{finite } V \quad A \in F$
shows $\text{finite } A$
using assms **by** $(\text{metis Pow-iff infinite-super rev-subsetD F-ss})$

end

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

abbreviation $gb\text{-}rename\text{-}ext\ ecnv\ f \equiv fr\text{-}rename\text{-}ext\ (gb\text{-}rename\text{-}ecnv\ ecnv\ f)\ f$

locale $gb\text{-}rename\text{-}precond =$
 $gb\text{-}graph\ G +$
 $g\text{-}rename\text{-}precond\ G\ f\ gb\text{-}rename\text{-}ecnv\ ecnv\ f$
for $G :: ('u, 'more)\ gb\text{-}graph\text{-}rec\text{-}scheme$
and $f :: 'u \Rightarrow 'v$ **and** $ecnv$
begin
lemma $G'\text{-}gb\text{-}fields: gb\text{-}F\ G' = \{ f'A \mid A. A \in F \}$
unfolding $gb\text{-}rename\text{-}ecnv\text{-}def\ fr\text{-}rename\text{-}ext\text{-}def$
by $simp$

sublocale $G': gb\text{-}graph\ G'$
apply $unfold\text{-}locales$
apply $(simp\text{-}all\ add: G'\text{-}fields\ G'\text{-}gb\text{-}fields)$
using $F\text{-}ss$
by $auto$

lemma $acc\text{-}sim1: is\text{-}acc\ r \Longrightarrow G'.is\text{-}acc\ (f\ o\ r)$
unfolding $is\text{-}acc\text{-}def\ G'.is\text{-}acc\text{-}def\ G'\text{-}gb\text{-}fields$
by $(fastforce\ intro: imageI\ simp: INFM\text{-}nat)$

lemma $acc\text{-}sim2:$
assumes $G'.is\text{-}acc\ r$ **shows** $is\text{-}acc\ (fi\ o\ r)$
proof $-$
from $assms$ **have** $1: \bigwedge A\ m. A \in gb\text{-}F\ G \Longrightarrow \exists i > m. r\ i \in f'A$
unfolding $G'.is\text{-}acc\text{-}def\ G'\text{-}gb\text{-}fields$
by $(auto\ simp: INFM\text{-}nat)$

{ **fix** $A\ m$
assume $2: A \in gb\text{-}F\ G$
from $1[OF\ this, of\ m]$ **have** $\exists i > m. fi\ (r\ i) \in A$
using $F\text{-}ss$
apply $clarsimp$
by $(metis\ Pow\text{-}iff\ 2\ fi\text{-}f\ in\text{-}mono)$
} **thus** $?thesis$
unfolding $is\text{-}acc\text{-}def$
by $(auto\ simp: INFM\text{-}nat)$

qed

lemma $acc\text{-}run\text{-}sim1: is\text{-}acc\text{-}run\ r \Longrightarrow G'.is\text{-}acc\text{-}run\ (f\ o\ r)$
using $acc\text{-}sim1\ run\text{-}sim1$ **unfolding** $G'.is\text{-}acc\text{-}run\text{-}def\ is\text{-}acc\text{-}run\text{-}def$
by $auto$

lemma $acc\text{-}run\text{-}sim2: G'.is\text{-}acc\text{-}run\ r \Longrightarrow is\text{-}acc\text{-}run\ (fi\ o\ r)$
using $acc\text{-}sim2\ run\text{-}sim2$ **unfolding** $G'.is\text{-}acc\text{-}run\text{-}def\ is\text{-}acc\text{-}run\text{-}def$
by $auto$

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

locale *gba* =
gb-graph *G*
for *G* :: ('Q,'L,'more) *gba-rec-scheme* +
assumes *L-ss*: *gba-L* *G* *q* *l* \Longrightarrow *q* \in *V*
begin
abbreviation *L* \equiv *gba-L* *G*

lemma *is-gba*: *gba* *G* **by** *unfold-locales*

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

lemma *acceptI*[*intro?*]: $[[\text{is-acc-run } r; \bigwedge i. L (r\ i) (w\ i)]] \Longrightarrow \text{accept } w$
by (*auto simp*: *accept-def*)

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

lemma *langI*[*intro?*]: *accept* *w* \Longrightarrow *w* \in *lang* **by** (*auto simp*: *lang-def*)

end

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

if *q* \in *f* \cdot *V* then *L* (*f* *q*) *l* else *False*)

unfolding *gb-rename-ecnv-def* *gba-rename-ecnv-def* *fr-rename-ext-def* *fi-def*

by *simp*

sublocale *G'*: *gba* *G'*

apply *unfold-locales*
apply (*auto simp add: G'-gba-fields G'-fields split: if-split-asm*)
done

lemma *L-sim1*: $\llbracket \text{range } r \subseteq V; L (r \ i) \ l \rrbracket \Longrightarrow G'.L (f (r \ i)) \ l$
by (*auto simp: G'-gba-fields fi-def[symmetric] fi-f*
dest: inj-onD[OF INJ]
dest!: rev-subsetD[OF rangeI[of - i]])

lemma *L-sim2*: $\llbracket \text{range } r \subseteq f'V; G'.L (r \ i) \ l \rrbracket \Longrightarrow L (fi (r \ i)) \ l$
by (*auto*
simp: G'-gba-fields fi-def[symmetric] f-fi
dest!: rev-subsetD[OF rangeI[of - i]])

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

apply (*rule ext*)

unfolding *accept-def G'.accept-def*

proof *safe*

fix $w \ r$

assume $R: G'.is\text{-acc-run } r$

assume $L: \forall i. G'.L (r \ i) (w \ i)$

from R **have** $RAN: \text{range } r \subseteq f'V$

using $G'.run\text{-reachable}[OF G'.acc\text{-run-run}[OF R]]$ $G'.reachable\text{-}V$

unfolding $G'\text{-fields}$

by *simp*

from L **show** $\exists r. is\text{-acc-run } r \wedge (\forall i. L (r \ i) (w \ i))$

using $acc\text{-run-sim2}[OF R]$ $L\text{-sim2}[OF RAN]$

by *auto*

next

fix $w \ r$

assume $R: is\text{-acc-run } r$

assume $L: \forall i. L (r \ i) (w \ i)$

from R **have** $RAN: \text{range } r \subseteq V$

using $run\text{-reachable}[OF acc\text{-run-run}[OF R]]$ $reachable\text{-}V$ **by** *simp*

from L **show** $\exists r.$

$G'.is\text{-acc-run } r$

$\wedge (\forall i. G'.L (r \ i) (w \ i))$

using $acc\text{-run-sim1}[OF R]$ $L\text{-sim1}[OF RAN]$

by *auto*

qed

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

unfolding $G'.lang\text{-def lang-def}$ **by** *simp*

lemma *finite-G'-V*:

assumes *finite V*

shows *finite G'.V*

using *assms* **by** (*auto simp add: G'-gba-fields G'-fields split: if-split-asm*)

end

abbreviation *gba-rename* \equiv *gba-rename-ext* (λ -. ())

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) \implies *finite* (g - V G')

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

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

unfolding G' -*def*

proof –

let $?G' =$ *gba-rename* f G

interpret *gba* G **by** *fact*

from *INJ* **interpret** *gba-rename-precond* G f λ -. ()

by *unfold-locales simp-all*

show *gba* $?G'$ **by** (*rule* G' .*is-gba*)

show *finite* (g - V G) \implies *finite* (g - V $?G'$) **by** (*fact* *finite-G'-V*)

show G' .*accept* = *accept* **by** *simp*

show G' .*lang* = *lang* **by** *simp*

qed

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 **by** *unfold-locales*

definition *to-gbg-ext* m

\equiv (\lfloor g - $V = V$,

g - $E = E$,

```

    g-V0=V0,
    gbg-F = if F=UNIV then {} else {F},
    ... = m )
abbreviation to-gbg ≡ to-gbg-ext ()

sublocale gbg: gb-graph to-gbg-ext m
apply unfold-locales
using V0-ss E-ss F-ss
apply (auto simp: to-gbg-ext-def split: if-split-asm)
done

definition is-acc :: 'Q word ⇒ bool where is-acc r ≡ (∃∞i. r i ∈ F)
definition is-acc-run where is-acc-run r ≡ is-run r ∧ 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
unfolding is-run-def[abs-def] gbg.is-run-def[abs-def]
  is-acc-def[abs-def] gbg.is-acc-def[abs-def]
  is-acc-run-def[abs-def] gbg.is-acc-run-def[abs-def]
by (auto simp: to-gbg-ext-def)

end

### 3.4 Buchi Automata



Buchi automata are labeled Buchi graphs
record ('Q,'L) ba-rec = 'Q b-graph-rec +
  ba-L :: 'Q ⇒ 'L ⇒ bool

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

  lemma is-ba: ba G by unfold-locales

  abbreviation to-gba-ext m ≡ to-gbg-ext (| gba-L = L, ...=m |)
  abbreviation to-gba ≡ to-gba-ext ()

```

```

sublocale gba: gba to-gba-ext m
  apply unfold-locales
  unfolding to-gbg-ext-def
  using L-ss apply auto []
done

```

```

lemma ba-acc-simps[simp]: gba.L T m = L
  by (simp add: to-gbg-ext-def)

```

```

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

```

```

definition lang  $\equiv \text{Collect } \text{accept}$ 

```

```

lemma to-gba-alt-accept:
  gba.accept T m = accept
  apply (intro ext)
  unfolding accept-def gba.accept-def
  apply (simp-all add: to-gbg-alt)
done

```

```

lemma to-gba-alt-lang:
  gba.lang T m = lang
  unfolding lang-def gba.lang-def
  apply (simp-all add: to-gba-alt-accept)
done

```

```

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:  $\bigcup (\text{range } (\text{igbg-acc } G)) \subseteq \{0..<(\text{igbg-num-acc } G)\}$ 
  assumes acc-ss: igbg-acc G q  $\neq \{\}$   $\implies q \in V$ 
begin
  abbreviation num-acc where num-acc  $\equiv \text{igbg-num-acc } G$ 
  abbreviation acc where acc  $\equiv \text{igbg-acc } G$ 

```

```

lemma is-igb-graph: igb-graph G by unfold-locales

```

```

lemma acc-boundI[simp, intro]: x  $\in \text{acc } q \implies x < \text{num-acc}$ 
  using acc-bound by fastforce

```

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

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

definition $to-gbg-ext\ m$

$\equiv (\mid g-V = V, g-E = E, g-V0 = V0, gbg-F = F, \dots = m \mid)$

sublocale $gbg: gb-graph\ to-gbg-ext\ m$

apply $unfold-locales$

using $V0-ss\ E-ss\ acc-ss$

apply $(auto\ simp: to-gbg-ext-def\ F-def\ accn-def)$

done

lemma $to-gbg-alt1:$

$gbg.E\ T\ m = E$

$gbg.V0\ T\ m = V0$

$gbg.F\ T\ m = F$

by $(simp-all\ add: to-gbg-ext-def)$

lemma $F-fin[simp,intro!]: finite\ F$

unfolding $F-def$

by $auto$

definition $is-acc :: 'Q\ word \Rightarrow bool$

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

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

lemma $is-run-gbg:$

$gbg.is-run\ T\ m = is-run$

unfolding $is-run-def[abs-def]\ is-acc-run-def[abs-def]$

$gbg.is-run-def[abs-def]\ gbg.is-acc-run-def[abs-def]$

by $(simp-all\ add: to-gbg-ext-def)$

lemma $is-acc-gbg:$

$gbg.is-acc\ T\ m = is-acc$

apply $(intro\ ext)$

unfolding $gbg.is-acc-def\ is-acc-def$

apply $(simp\ add: to-gbg-alt1\ is-run-gbg)$

unfolding $F-def\ accn-def$

apply $(blast\ intro: INFM-mono)$

done

lemma $is-acc-run-gbg:$

$gbg.is-acc-run\ T\ m = is-acc-run$

apply $(intro\ ext)$

unfolding $gbg.is-acc-run-def\ is-acc-run-def$

by $(simp-all\ add: to-gbg-alt1\ is-run-gbg\ is-acc-gbg)$

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  $\longleftrightarrow$  ( $\forall n < \text{num-acc. } \text{limit } r \cap \text{accn } n \neq \{\}$ )
proof
  assume  $\forall n < \text{num-acc. } \text{limit } r \cap \text{accn } n \neq \{\}$ 
  thus is-acc r
    unfolding is-acc-def accn-def
    by (auto dest!: limit-inter-INF)
next
  from FIN have FIN':  $\bigwedge A. \text{finite } (A \cap \text{range } r)$  by simp
  assume is-acc r
  hence  $\forall n < \text{num-acc. } \text{limit } r \cap (\text{accn } n \cap \text{range } r) \neq \{\}$ 
  unfolding is-acc-def accn-def
  by (auto simp: fin-ex-inf-eq-limit[OF FIN', symmetric])
  thus  $\forall n < \text{num-acc. } \text{limit } r \cap \text{accn } n \neq \{\}$  by auto
qed

lemma acc-limit-alt':
  finite (range r)  $\implies$  is-acc r  $\longleftrightarrow$  ( $\bigcup (\text{acc } ' \text{limit } r) = \{0..<\text{num-acc}\}$ )
  unfolding acc-limit-alt
  by (auto simp: accn-def)

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  $L \equiv \text{igba-L } G$ 

  lemma is-igba: igba G by unfold-locales

  abbreviation to-gba-ext m  $\equiv$  to-gbg-ext ( $\big| \text{gba-L} = \text{igba-L } G, \dots = m \big|$ )

  sublocale gba: gba to-gba-ext m
  apply unfold-locales
  unfolding to-gbg-ext-def
  using L-ss
  apply auto
  done

  lemma to-gba-alt-L:

```


gba.L T m = L
by (*auto simp: to-gbg-ext-def*)

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

definition *lang* $\equiv \text{Collect } \text{accept}$

lemma *accept-gba-alt: gba.accept T m = accept*

apply (*intro ext*)
unfolding *accept-def gba.accept-def*
apply (*simp add: to-gbg-alt to-gba-alt-L*)
done

lemma *lang-gba-alt: gba.lang T m = lang*

unfolding *lang-def gba.lang-def*
apply (*simp add: accept-gba-alt*)
done

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* $:: 'Q \text{ set set} \Rightarrow (\text{nat} \times ('Q \Rightarrow \text{nat set})) \text{ nres}$ **where**

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

lemma *F-to-idx-correct:*

shows *F-to-idx F* $\leq \text{SPEC } (\lambda(\text{num-acc}, \text{acc}). F = \{ \{q. i \in \text{acc } q\} \mid i. i < \text{num-acc}$

$\} \wedge \bigcup (\text{range } \text{acc}) \subseteq \{0..<\text{num-acc}\})$

unfolding *F-to-idx-def*

apply (*refine-rcg refine-vcg*)
apply (*clarsimp dest!: sym[where t=F]*)
apply (*intro equalityI subsetI*)
apply (*auto simp: in-set-conv-nth*) [2]

apply *auto* []

done

definition *mk-acc-impl Flist* $\equiv \text{do}$ {

let acc = Map.empty;

$(-, \text{acc}) \leftarrow \text{nfoldli } \text{Flist } (\lambda-. \text{True}) (\lambda A (i, \text{acc}). \text{do}$ {
acc $\leftarrow \text{FOREACHi } (\lambda it \text{acc}'.$

```

    acc' = (λv.
      if v∈A-it then
        Some (insert i (the-default {} (acc v)))
      else
        acc v
    )
  )
  A (λv acc. RETURN (acc(v→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 $FIN: \forall A \in set\ Flist. finite\ A$

shows $mk-acc-impl\ Flist' \leq \Downarrow Id\ (RETURN\ (\lambda v. \{i . i < length\ Flist \wedge v \in Flist!i\}))$

using F **apply** *simp*

unfolding *mk-acc-impl-def*

apply (*refine-rcg*

nfoldli-rule [**where**

$I = \lambda l1\ l2\ (i, res). i = length\ l1$

$\wedge the-default\ \{\}\ o\ res = (\lambda v. \{j . j < i \wedge v \in Flist!j\})$

]

refine-vcg

)

using FIN **apply** (*simp-all*)

apply (*rule ext*) **apply** *auto* []

apply (*rule ext*) **apply** (*auto split: if-split-asm simp: nth-append nth-Cons'*) []

apply (*rule ext*) **apply** (*auto split: if-split-asm simp: nth-append nth-Cons'*
fun-comp-eq-conv) []

apply (*rule ext*) **apply** (*auto simp: fun-comp-eq-conv*) []

done

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

$F-to-idx-impl\ F \equiv do\ \{$

$Flist \leftarrow SPEC\ (\lambda Flist. distinct\ Flist \wedge set\ Flist = F);$

$let\ num-acc = length\ Flist;$

$acc \leftarrow mk-acc-impl\ Flist;$

$RETURN\ (num-acc, acc)$

$\}$

lemma *F-to-idx-refine*:

assumes $FIN: \forall A \in F. finite\ A$

shows $F-to-idx-impl\ F \leq \Downarrow Id\ (F-to-idx\ F)$

using *assms*

unfolding $F\text{-to-idx-impl-def } F\text{-to-idx-def}$

apply ($\text{refine-rcg bind-Let-refine2}[OF \text{ mk-acc-impl-correct}]$)

apply *auto*

done

definition $gbg\text{-to-idx-ext}$

```
:: -  $\Rightarrow$  ('a, 'more)  $gb\text{-graph-rec-scheme} \Rightarrow$  ('a, 'more')  $igb\text{-graph-rec-scheme nres}$ 
where  $gbg\text{-to-idx-ext ecnv } A = \text{do}$  {
  ( $num\text{-acc}, acc$ )  $\leftarrow F\text{-to-idx-impl } (gbg\text{-}F A)$ ;
  RETURN (|
     $g\text{-}V = g\text{-}V A$ ,
     $g\text{-}E = g\text{-}E A$ ,
     $g\text{-}V0 = g\text{-}V0 A$ ,
     $igb\text{-}num\text{-acc} = num\text{-acc}$ ,
     $igb\text{-}acc = acc$ ,
    ... =  $ecnv A$ 
  |)
}
```

lemma (**in** $gb\text{-graph}$) $gbg\text{-to-idx-ext-correct}$:

assumes [$simp, intro$]: $\bigwedge A. A \in F \implies \text{finite } A$

shows $gbg\text{-to-idx-ext ecnv } G \leq SPEC (\lambda G'.$

$igb\text{-graph.is-acc-run } G' = is\text{-acc-run}$

$\wedge g\text{-}V G' = V$

$\wedge g\text{-}E G' = E$

$\wedge g\text{-}V0 G' = V0$

$\wedge igb\text{-graph-rec.more } G' = ecnv G$

$\wedge igb\text{-graph } G'$

)

proof –

note $F\text{-to-idx-refine}[of F]$

also note $F\text{-to-idx-correct}$

finally have $R: F\text{-to-idx-impl } F$

$\leq SPEC (\lambda(num\text{-acc}, acc). F = \{\{q. i \in acc\} \mid i. i < num\text{-acc}\})$

$\wedge \bigcup (range\ acc) \subseteq \{0..<num\text{-acc}\}$) **by** $simp$

have $eq\text{-conj}I: \bigwedge a\ b\ c. (b \leftarrow c) \implies (a \& b \longleftrightarrow a \& c)$ **by** $simp$

```
{
  fix  $acc :: 'Q \Rightarrow nat\ set$  and  $num\text{-acc } r$ 
  have ( $\forall A. (\exists i. A = \{q. i \in acc\} \wedge i < num\text{-acc}) \longrightarrow (limit\ r \cap A \neq \{\})$ )
     $\longleftrightarrow (\forall i < num\text{-acc}. \exists q \in limit\ r. i \in acc\ q)$ 
    by  $blast$ 
} note  $aux1 = this$ 
```

```
{
  fix  $acc :: 'Q \Rightarrow nat\ set$  and  $num\text{-acc } i$ 
```

```

assume FE:  $F = \{\{q. i \in \text{acc } q\} \mid i. i < \text{num-acc}\}$ 
assume INR:  $(\bigcup x. \text{acc } x) \subseteq \{0..<\text{num-acc}\}$ 
have finite  $\{q. i \in \text{acc } q\}$ 
proof (cases  $i < \text{num-acc}$ )
  case True thus ?thesis using FE by auto
next
  case False hence  $\{q. i \in \text{acc } q\} = \{\}$  using INR by force
  thus ?thesis by simp
qed
} note aux2=this

{
fix acc :: 'Q  $\Rightarrow$  nat set and num-acc q
assume FE:  $F = \{\{q. i \in \text{acc } q\} \mid i. i < \text{num-acc}\}$ 
  and INR:  $(\bigcup x. \text{acc } x) \subseteq \{0..<\text{num-acc}\}$ 
  and acc  $q \neq \{\}$ 
then obtain i where  $i \in \text{acc } q$  by auto
moreover with INR have  $i < \text{num-acc}$  by force
ultimately have  $q \in \bigcup F$  by (auto simp: FE)
with F-ss have  $q \in V$  by auto
} note aux3=this

show ?thesis
  unfolding gbg-to-idx-ext-def
  apply (refine-rcg order-trans[OF R] refine-vcg)
proof clarsimp-all
  fix acc and num-acc :: nat
  assume FE[simp]:  $F = \{\{q. i \in \text{acc } q\} \mid i. i < \text{num-acc}\}$ 
  and BOUND:  $(\bigcup x. \text{acc } x) \subseteq \{0..<\text{num-acc}\}$ 
  let ?G' = ( $\mid$ 
    g-V = V,
    g-E = E,
    g-V0 = V0,
    igbg-num-acc = num-acc,
    igbg-acc = acc,
    ... = ecnv G)

interpret G': igb-graph ?G'
  apply unfold-locales
  using V0-ss E-ss
  apply (auto simp add: aux2 aux3 BOUND)
  done

show igb-graph ?G' by unfold-locales

show G'.is-acc-run = is-acc-run
  unfolding G'.is-acc-run-def[abs-def] is-acc-run-def[abs-def]
  G'.is-run-def[abs-def] is-run-def[abs-def]
  G'.is-acc-def[abs-def] is-acc-def[abs-def]

```

```

    apply (clarsimp intro!: ext eq-conjI)
    apply auto []
    apply (metis (lifting, no-types) INFM-mono mem-Collect-eq)
  done
qed
qed

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-Lcnv where ti-Lcnv ecnv A  $\equiv$  ( $\mid$  igba-L = gba-L A, ...=ecnv A  $\mid$ )

abbreviation gba-to-idx-ext ecnv  $\equiv$  gbg-to-idx-ext (ti-Lcnv 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')
  apply (rule order-trans[OF gbg-to-idx-ext-correct])
  apply (rule, assumption)
  apply (rule SPEC-rule)
  apply (elim conjE, intro conjI)
proof -
  fix G'
  assume
    ARUN: igb-graph.is-acc-run G' = is-acc-run
    and MORE: igb-graph-rec.more G' = ti-Lcnv ecnv G
    and LOC: igb-graph G'
    and FIELDS: g-V G' = V    g-E G' = E    g-V0 G' = V0

  from LOC interpret igb: igb-graph G' .

interpret igb: igba G'
  apply unfold-locales
  using MORE FIELDS L-ss
  unfolding ti-Lcnv-def
  apply (cases G')
  apply simp
  done

show igba.accept G' = accept and igba-rec.more G' = ecnv G

```

```

using ARUN MORE
unfolding accept-def[abs-def] igb.accept-def[abs-def] ti-Lcnv-def
apply (cases G', (auto) []) +
done

```

```

show igba G' by unfold-locales
qed

```

```

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 ( $\lambda G'. \text{igba.lang } G' = \text{lang} \wedge \text{igba-rec.more } G' = \text{ecnv } G \wedge \text{igba } G'$ )
apply (rule order-trans[OF gba-to-idx-ext-correct])
apply (rule, assumption)
apply (rule SPEC-rule)
unfolding lang-def[abs-def]
apply (subst igba.lang-def)
apply auto
done

```

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

```

abbreviation degeneralize **where** degeneralize \equiv degeneralize-ext ($\lambda \cdot. ()$)

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

by *auto*

lemma *degen-invar*: *b-graph (degeneralize-ext ecnv)*

proof

let $?G' = \text{degeneralize-ext ecnv}$

show $g-V0 ?G' \subseteq g-V ?G'$

unfolding *degeneralize-ext-def*

using *V0-ss*

by *auto*

show $g-E ?G' \subseteq g-V ?G' \times g-V ?G'$

unfolding *degeneralize-ext-def*

using *E-ss*

by *auto*

show $bg-F ?G' \subseteq g-V ?G'$

unfolding *degeneralize-ext-def*

using *acc-ss*

by *auto*

qed

sublocale *degen*: *b-graph degeneralize-ext m using degen-invar* .

lemma *degen-finite-reachable*:

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

shows *finite ((g-E (degeneralize-ext ecnv))* “ g-V0 (degeneralize-ext ecnv))*

proof –

let $?G' = \text{degeneralize-ext ecnv}$

have $((g-E ?G')^* “ g-V0 ?G')$

$\subseteq E^* “ V0 \times \{0..num-acc\}$

proof –

{

fix $q\ n\ q'\ n'$

assume $((q,n),(q',n')) \in (g-E ?G')^*$

and $0: (q,n) \in g-V0 ?G'$

hence $G1: (q,q') \in E^* \wedge n' \leq num-acc$

apply (*induction rule: rtrancl-induct2*)

by (*auto simp: degeneralize-ext-def split: if-split-asm*)

from 0 **have** $G2: q \in V0 \wedge n \leq num-acc$

by (*auto simp: degeneralize-ext-def split: if-split-asm*)

note $G1\ G2$

} **thus** *?thesis* **by** *fastforce*

qed

also have *finite ...* **by** *auto*

finally (*finite-subset*) **show** *finite ((g-E ?G')^* “ g-V0 ?G')* .

qed

```

lemma degen-is-run-sound:
  degen.is-run  $T$   $m$   $r \implies is-run$  ( $fst$   $o$   $r$ )
  unfolding degen.is-run-def is-run-def
  unfolding degeneralize-ext-def
  apply (clarsimp split: if-split-asm simp: ipath-def)
  apply (metis fst-conv)+
  done

lemma degen-path-sound:
  assumes path (degen.E  $T$   $m$ )  $u$   $p$   $v$ 
  shows path  $E$  ( $fst$   $u$ ) ( $map$   $fst$   $p$ ) ( $fst$   $v$ )
  using assms
  by induction (auto simp: degeneralize-ext-def path-simps split: if-split-asm)

lemma degen-V0-sound:
  assumes  $u \in degen.V0$   $T$   $m$ 
  shows  $fst$   $u \in V0$ 
  using assms
  by (auto simp: degeneralize-ext-def path-simps split: if-split-asm)

lemma degen-visit-acc:
  assumes path (degen.E  $T$   $m$ ) ( $q,n$ )  $p$  ( $q',n'$ )
  assumes  $n \neq n'$ 
  shows  $\exists qa. (qa,n) \in set$   $p \wedge n \in acc$   $qa$ 
  using assms
proof (induction - ( $q,n$ )  $p$  ( $q',n'$ ) arbitrary: q rule: path.induct)
  case (path-prepend qnh p)
  then obtain  $qh$   $nh$  where [simp]:  $qnh=(qh,nh)$  by (cases qnh)
  from  $\langle ((q,n),qnh) \in degen.E$   $T$   $m \rangle$ 
  have  $nh=n \vee (nh=(n+1) \bmod num-acc \wedge n \in acc$   $q)$ 
  by (auto simp: degeneralize-ext-def split: if-split-asm)
  thus ?case proof
    assume [simp]:  $nh=n$ 
    from path-prepend obtain  $qa$  where  $(qa, n) \in set$   $p$  and  $n \in acc$   $qa$ 
    by auto
    thus ?case by auto
  next
    assume  $(nh=(n+1) \bmod num-acc \wedge n \in acc$   $q)$  thus ?case by auto
  qed
qed simp

lemma degen-run-complete0:
  assumes [simp]:  $num-acc = 0$ 
  assumes  $R$ : is-run  $r$ 
  shows degen.is-run  $T$   $m$  ( $\lambda i. (r$   $i,0)$ )
  using  $R$ 
  unfolding degen.is-run-def is-run-def

```


unfolding *ipath-def degeneralize-ext-def*
by *auto*

lemma *degen-acc-run-complete0*:

assumes [*simp*]: *num-acc = 0*

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

shows *degen.is-acc-run T m (λi. (r i,0))*

using *R*

unfolding *degen.is-acc-run-def is-acc-run-def is-acc-def degen.is-acc-def*

apply (*simp add: degen-run-complete0*)

unfolding *degeneralize-ext-def*

using *run-reachable[of r] reachable-V*

by (*auto simp: INFM-nat*)

lemma *degen-run-complete*:

assumes [*simp*]: *num-acc ≠ 0*

assumes *R*: *is-run r*

shows $\exists r'. \text{degen.is-run } T \ m \ r' \wedge r = \text{fst } o \ r'$

using *R*

unfolding *degen.is-run-def is-run-def ipath-def*

apply (*elim conjE*)

proof –

assume *R0*: $r \ 0 \in V0$ **and** *RS*: $\forall i. (r \ i, r \ (\text{Suc } i)) \in E$

define *r'* **where** $r' = \text{rec-nat}$

$(r \ 0, 0)$

$(\lambda i \ (q, n). (r \ (\text{Suc } i), \text{if } n \in \text{acc } q \text{ then } (n+1) \ \text{mod } \text{num-acc} \ \text{else } n))$

have [*simp*]:

$r' \ 0 = (r \ 0, 0)$

$\bigwedge i. r' \ (\text{Suc } i) = ($

let

$(q, n) = r' \ i$

in

$(r \ (\text{Suc } i), \text{if } n \in \text{acc } q \text{ then } (n+1) \ \text{mod } \text{num-acc} \ \text{else } n)$

)

unfolding *r'-def*

by *auto*

have *R0'*: $r' \ 0 \in \text{degen.V0 } T \ m$ **using** *R0*

unfolding *degeneralize-ext-def* **by** *auto*

have *MAP*: $r = \text{fst } o \ r'$

proof (*rule ext*)

fix *i*

show $r \ i = (\text{fst } o \ r') \ i$

by (*cases i*) (*auto simp: split: prod.split*)

qed

```

have [simp]: 0 < num-acc by (cases num-acc) auto

have SND-LESS:  $\bigwedge i. \text{snd } (r' i) < \text{num-acc}$ 
proof -
  fix i show  $\text{snd } (r' i) < \text{num-acc}$  by (induction i) (auto split: prod.split)
qed

have RS':  $\forall i. (r' i, r' (\text{Suc } i)) \in \text{degen.E } T m$ 
proof
  fix i
  obtain n where [simp]:  $r' i = (r i, n)$ 
  apply (cases i)
  apply (force)
  apply (force split: prod.split)
  done
  from SND-LESS[of i] have [simp]:  $n < \text{num-acc}$  by simp

  show  $(r' i, r' (\text{Suc } i)) \in \text{degen.E } T m$  using RS
  by (auto simp: degeneralize-ext-def)
qed

from R0' RS' MAP show
 $\exists r'. (r' 0 \in \text{degen.V0 } T m$ 
 $\wedge (\forall i. (r' i, r' (\text{Suc } i)) \in \text{degen.E } T m))$ 
 $\wedge r = \text{fst} \circ r'$  by blast
qed

lemma degenerate-run-bound:
  assumes [simp]:  $\text{num-acc} \neq 0$ 
  assumes R:  $\text{degen.is-run } T m r$ 
  shows  $\text{snd } (r i) < \text{num-acc}$ 
  apply (induction i)
  using R
  unfolding degenerate.is-run-def is-run-def
  unfolding degeneralize-ext-def ipath-def
  apply -
  apply auto []
  apply clarsimp
  by (metis snd-conv)

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

```

from *RI* **have** $n < \text{num-acc}$ **using** *degen-run-bound*[*OF NN0 R, of i*] **by** *auto*
from *EXJ* **have**

$j \geq i$
 $n \in \text{acc} (\text{fst} (r j))$
 $\forall k \geq i. n \in \text{acc} (\text{fst} (r k)) \longrightarrow j \leq k$
using *LeastI-ex*[*OF EXJ*]
unfolding *j-def*
apply (*auto*) [2]
apply (*metis* (*lifting*) *Least-le*)
done

hence $\forall k \geq i. k < j \longrightarrow n \notin \text{acc} (\text{fst} (r k))$ **by** *auto*

have $\forall k. k \geq i \wedge k \leq j \longrightarrow (\text{snd} (r k) = n)$

proof (*clarify*)

fix *k*

assume $i \leq k \quad k \leq j$

thus $\text{snd} (r k) = n$

proof (*induction k rule: less-induct*)

case (*less k*)

show *?case proof* (*cases k=i*)

case *True* **thus** *?thesis using RI by simp*

next

case *False* **with** *less.prem*s **have** $k - 1 < k \quad i \leq k - 1 \quad k - 1 \leq j$

by *auto*

from *less.IH*[*OF this*] **have** $\text{snd} (r (k - 1)) = n$.

moreover from *R* **have**

$(r (k - 1), r k) \in \text{degen.E T m}$

unfolding *degen.is-run-def is-run-def ipath-def*

by *clarsimp* (*metis One-nat-def Suc-diff-1* $\langle k - 1 < k \rangle$

less-nat-zero-code neq0-conv)

moreover have $n \notin \text{acc} (\text{fst} (r (k - 1)))$

using $\langle \forall k \geq i. k < j \longrightarrow n \notin \text{acc} (\text{fst} (r k)) \rangle \langle i \leq k - 1 \rangle \langle k - 1 < k \rangle$
*dual-order.strict-trans1 less.prem*s(2)

by *blast*

ultimately show *?thesis*

by (*auto simp: degeneralize-ext-def*)

qed

qed

qed

thus *?thesis*

by (*metis* $\langle i \leq j \rangle \langle n \in \text{local.acc} (\text{fst} (r j)) \rangle$
order-refl surjective-pairing)

qed

lemma *degen-acc-run-complete-aux1'*:

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

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

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

assumes $RI: r\ i = (q, n)$
shows $\exists j \geq i. \exists q'. r\ j = (q', n) \wedge n \in \text{acc } q'$
proof –
from RI **have** $n < \text{num-acc}$ **using** $\text{degen-run-bound}[OF\ NN0\ R, \text{ of } i]$ **by** auto
with ACC **have** $EXJ: \exists j \geq i. n \in \text{acc } (\text{fst } (r\ j))$
unfolding $INFM\text{-nat-le}$ **by** blast

from $\text{degen-acc-run-complete-aux1}[OF\ NN0\ R\ EXJ\ RI]$ **show** $?thesis$.
qed

lemma $\text{degen-acc-run-complete-aux2}$:
assumes $NN0[\text{simp}]: \text{num-acc} \neq 0$
assumes $R: \text{degen.is-run } T\ m\ r$
assumes $ACC: \forall n < \text{num-acc}. \exists_{\infty} i. n \in \text{acc } (\text{fst } (r\ i))$
assumes $RI: r\ i = (q, n)$ **and** $OFS: \text{ofs} < \text{num-acc}$
shows $\exists j \geq i. \exists q'.$
 $r\ j = (q', (n + \text{ofs}) \bmod \text{num-acc}) \wedge (n + \text{ofs}) \bmod \text{num-acc} \in \text{acc } q'$
using $RI\ OFS$
proof ($\text{induction ofs arbitrary: } q\ n\ i$)
case 0
from $\text{degen-run-bound}[OF\ NN0\ R, \text{ of } i]$ $\langle r\ i = (q, n) \rangle$
have $NLE: n < \text{num-acc}$
by simp

with $\text{degen-acc-run-complete-aux1}'[OF\ NN0\ R\ ACC\ \langle r\ i = (q, n) \rangle]$ **show** $?case$
by auto
next
case $(\text{Suc } \text{ofs})$
from $\text{Suc.IH}[OF\ \text{Suc.prem}(1)]\ \text{Suc.prem}(2)$
obtain $j\ q'$ **where** $j \geq i$ **and** $RJ: r\ j = (q', (n + \text{ofs}) \bmod \text{num-acc})$
and $A: (n + \text{ofs}) \bmod \text{num-acc} \in \text{acc } q'$
by auto
from R **have** $(r\ j, r\ (\text{Suc } j)) \in \text{degen.E } T\ m$
by ($\text{auto simp: degen.is-run-def is-run-def ipath-def}$)
with $RJ\ A$ **obtain** $q2$ **where** $RSJ: r\ (\text{Suc } j) = (q2, (n + \text{Suc } \text{ofs}) \bmod \text{num-acc})$

by ($\text{auto simp: degeneralize-ext-def mod-simps}$)

have $\text{aux: } \bigwedge j'. i \leq j \implies \text{Suc } j \leq j' \implies i \leq j'$ **by** auto
from $\text{degen-acc-run-complete-aux1}'[OF\ NN0\ R\ ACC\ RSJ]$ $\langle j \geq i \rangle$
show $?case$
by (auto dest: aux)
qed

lemma $\text{degen-acc-run-complete}$:
assumes $AR: \text{is-acc-run } r$
obtains r'
where $\text{degen.is-acc-run } T\ m\ r'$ **and** $r = \text{fst } o\ r'$
proof ($\text{cases num-acc} = 0$)

```

case True
with AR degen-acc-run-complete0
show ?thesis by (auto intro!: that[of ( $\lambda i. (r\ i, 0)$ )])
next
case False
assume NN0[simp]: num-acc  $\neq 0$ 

from AR have R: is-run r and ACC:  $\forall n < \text{num-acc}. \exists_{\infty} i. n \in \text{acc}\ (r\ i)$ 
unfolding is-acc-run-def is-acc-def by auto

from degen-run-complete[OF NN0 R] obtain r' where
  R': degen.is-run T m r'
  and [simp]: r = fst  $\circ$  r'
  by auto

from ACC have ACC':  $\forall n < \text{num-acc}. \exists_{\infty} i. n \in \text{acc}\ (\text{fst}\ (r'\ i))$  by simp

have  $\forall i. \exists j > i. r'\ j \in \text{degen.F T m}$ 
proof
  fix i
  obtain q n where RI: r' (Suc i) = (q, n) by (cases r' (Suc i))
  have  $(n + (\text{num-acc} - n \bmod \text{num-acc})) \bmod \text{num-acc} = 0$ 
    apply (rule dvd-imp-mod-0)
    apply (metis (mono-tags, lifting) NN0 add-diff-inverse mod-0-imp-dvd
      mod-add-left-eq mod-less-divisor mod-self nat-diff-split not-gr-zero zero-less-diff)
    done
  then obtain ofs where
    OFS-LESS: ofs  $<$  num-acc
    and [simp]:  $(n + \text{ofs}) \bmod \text{num-acc} = 0$ 
    by (metis NN0 Nat.add-0-right diff-less neq0-conv)
  with degen-acc-run-complete-aux2[OF NN0 R' ACC' RI OFS-LESS]
  obtain j q' where
    j  $>$  i r' j = (q', 0) and  $0 \in \text{acc}\ q'$ 
    by (auto simp: less-eq-Suc-le)
  thus  $\exists j > i. r'\ j \in \text{degen.F T m}$ 
    by (auto simp: degeneralize-ext-def)
qed
hence  $\exists_{\infty} i. r'\ i \in \text{degen.F T m}$  by (auto simp: INFM-nat)

have degen.is-acc-run T m r'
unfolding degen.is-acc-run-def degen.is-acc-def
by rule fact+
thus ?thesis by (auto intro: that)
qed

lemma degen-run-find-change:
assumes NN0[simp]: num-acc  $\neq 0$ 
assumes R: degen.is-run T m r
assumes A:  $i \leq j \quad r\ i = (q, n) \quad r\ j = (q', n') \quad n \neq n'$ 

```

obtains $k \ qk$ **where** $i \leq k \quad k < j \quad r \ k = (qk, n) \quad n \in \text{acc } qk$
proof –
from *degen-run-bound*[*OF NNO R*] *A* **have** $n < \text{num-acc} \quad n' < \text{num-acc}$
by (*metis snd-conv*)+

define k **where** $k = (\text{LEAST } k. i < k \wedge \text{snd } (r \ k) \neq n)$

have $i < k \quad \text{snd } (r \ k) \neq n$
by (*metis (lifting, mono-tags) LeastI-ex A k-def leD less-linear snd-conv*)+

from *Least-le*[**where** $P = \lambda k. i < k \wedge \text{snd } (r \ k) \neq n$, *folded k-def*]
have *LEK-EQN*: $\forall k'. i \leq k' \wedge k' < k \longrightarrow \text{snd } (r \ k') = n$
using $\langle r \ i = (q, n) \rangle$
by *clarsimp (metis le-neg-implies-less not-le snd-conv)*
hence *SND-RKMO*: $\text{snd } (r \ (k - 1)) = n$ **using** $\langle i < k \rangle$ **by** *auto*
moreover from *R* **have** $(r \ (k - 1), r \ k) \in \text{degen.E } T \ m$
unfolding *degen.is-run-def ipath-def* **using** $\langle i < k \rangle$
by *clarsimp (metis Suc-pred gr-implies-not0 neq0-conv)*
moreover note $\langle \text{snd } (r \ k) \neq n \rangle$
ultimately have $n \in \text{acc } (\text{fst } (r \ (k - 1)))$
by (*auto simp: degeneralize-ext-def split: if-split-asm*)
moreover have $k - 1 < j$ **using** *A LEK-EQN*
apply (*rule-tac ccontr*)
apply *clarsimp*
by (*metis One-nat-def* $\langle \text{snd } (r \ (k - 1)) = n \rangle$ *less-Suc-eq*
less-imp-diff-less not-less-eq snd-conv)
ultimately show *thesis*
apply –
apply (*rule that*[*of* $k - 1 \quad \text{fst } (r \ (k - 1))$])
using $\langle i < k \rangle$ *SND-RKMO* **by** *auto*
qed

lemma *degen-run-find-acc-aux*:
assumes *NN0*[*simp*]: $\text{num-acc} \neq 0$
assumes *AR*: *degen.is-acc-run* *T m r*
assumes *A*: $r \ i = (q, 0) \quad 0 \in \text{acc } q \quad n < \text{num-acc}$
shows $\exists j \ qj. i \leq j \wedge r \ j = (qj, n) \wedge n \in \text{acc } qj$
proof –
from *AR* **have** *R*: *degen.is-run* *T m r*
and *ACC*: $\exists_{\infty} i. r \ i \in \text{degen.F } T \ m$

unfolding *degen.is-acc-run-def degen.is-acc-def* **by** *auto*
from *ACC* **have** *ACC'*: $\forall i. \exists j > i. r \ j \in \text{degen.F } T \ m$
by (*auto simp: INFM-nat*)

show *?thesis* **using** $\langle n < \text{num-acc} \rangle$
proof (*induction n*)
case *0* **thus** *?case* **using** *A* **by** *auto*

next
case $(\text{Suc } n)$
then obtain $j \ qj$ **where** $i \leq j$ $r \ j = (qj, n)$ $n \in \text{acc } qj$ **by** *auto*
moreover from R **have** $(r \ j, r \ (\text{Suc } j)) \in \text{degen}.E \ T \ m$
unfolding *degen.is-run-def ipath-def*
by *auto*
ultimately obtain qsj **where** $RSJ: r \ (\text{Suc } j) = (qsj, \text{Suc } n)$
unfolding *degeneralize-ext-def* **using** $\langle \text{Suc } n < \text{num-acc} \rangle$ **by** *auto*

from ACC' **obtain** $k \ q0$ **where** $\text{Suc } j \leq k$ $r \ k = (q0, 0)$
unfolding *degeneralize-ext-def* **apply** *auto*
by *(metis less-imp-le-nat)*
from *degen-run-find-change*[$OF \ NNO \ R \ \langle \text{Suc } j \leq k \rangle \ RSJ \ \langle r \ k = (q0, 0) \rangle$]
obtain $l \ ql$ **where**
 $\text{Suc } j \leq l$ $l < k$ $r \ l = (ql, \text{Suc } n)$ $\text{Suc } n \in \text{acc } ql$
by *blast*
thus *?case* **using** $\langle i \leq j \rangle$
by *(intro exI[where x=l] exI[where x=ql]) auto*
qed
qed

lemma *degen-acc-run-sound*:
assumes $A: \text{degen}.is\text{-acc}\text{-run } T \ m \ r$
shows *is-acc-run* $(fst \ o \ r)$
proof $-$
from A **have** $R: \text{degen}.is\text{-run } T \ m \ r$
and $ACC: \exists_{\infty} i. r \ i \in \text{degen}.F \ T \ m$
unfolding *degen.is-acc-run-def degen.is-acc-def* **by** *auto*
from *degen-is-run-sound*[$OF \ R$] **have** $R': is\text{-run } (fst \ o \ r)$.

show *?thesis*
proof $(cases \ \text{num-acc} = 0)$
case $NNO[simp]: \text{False}$

from ACC **have** $ACC': \forall i. \exists j > i. r \ j \in \text{degen}.F \ T \ m$
by *(auto simp: INFM-nat)*

have $\forall n < \text{num-acc}. \forall i. \exists j > i. n \in \text{acc } (fst \ (r \ j))$
proof *(intro allI impI)*
fix $n \ i$

obtain $j \ qj$ **where** $j > i$ **and** $RJ: r \ j = (qj, 0)$ **and** $ACCJ: 0 \in \text{acc } (qj)$
using ACC' **unfolding** *degeneralize-ext-def* **by** *fastforce*

assume $NLESS: n < \text{num-acc}$
show $\exists j > i. n \in \text{acc } (fst \ (r \ j))$
proof $(cases \ n)$
case 0 **thus** *?thesis* **using** $\langle j > i \rangle \ RJ \ ACCJ$ **by** *auto*
next

```

      case [simp]: (Suc n')
      from degen-run-find-acc-aux[OF NN0 A RJ ACCJ NLESS] obtain k qk
where
  j ≤ k   r k = (qk,n)   n ∈ acc qk by auto
  thus ?thesis
  by (metis ⟨i < j⟩ dual-order.strict-trans1 fst-conv)
qed
qed
hence ∀ n < num-acc. ∃ ∞ i. n ∈ acc (fst (r i))
  by (auto simp: INFM-nat)
with R' show ?thesis
  unfolding is-acc-run-def is-acc-def by auto
next
case [simp]: True
with R' show ?thesis
  unfolding is-acc-run-def is-acc-def
  by auto
qed
qed

lemma degen-acc-run-iff:
  is-acc-run r ⟷ (∃ r'. fst o r' = r ∧ degen.is-acc-run T m r')
  using degen-acc-run-complete degen-acc-run-sound
  by blast

```

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 ⇒ 'L

```

```

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

```

```

  abbreviation L where L ≡ sa-L G

```

```

  definition accept w ≡ ∃ r. is-run r ∧ w = L o r

```

```

  lemma acceptI[intro?]: [[is-run r; w = L o r]] ⇒ accept w by (auto simp:
accept-def)

```

```

  definition lang ≡ Collect accept

```

```

  lemma langI[intro?]: accept w ⇒ w ∈ lang by (auto simp: lang-def)

```


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* ≡ (|
g-V = *igba.V* × *sa.V*,
g-E = { ((*q*, *s*), (*q'*, *s'*)).
igba.L q (sa.L s) ∧ (q, q') ∈ igba.E ∧ (s, s') ∈ sa.E },
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 {}*) |)

lemma *prod-invar*: *igb-graph prod*
apply *unfold-locales*

using *igba.V0-ss sa.V0-ss*
apply (*auto simp: prod-def*) []

using *igba.E-ss sa.E-ss*
apply (*auto simp: prod-def*) []

using *igba.acc-bound*
apply (*auto simp: prod-def split: if-split-asm*) []

using *igba.acc-ss*
apply (*fastforce simp: prod-def split: if-split-asm*) []
done

sublocale *prod*: *igb-graph prod using prod-invar* .

lemma *prod-finite-reachable*:
assumes *finite (igba.E* “ igba.V0) finite (sa.E* “ sa.V0)*
shows *finite ((g-E prod)* “ g-V0 prod)*

proof –
{
fix *q s q' s'*
assume ((*q*, *s*), (*q'*, *s'*)) ∈ (*g-E prod*)*
hence (*q*, *q'*) ∈ (*igba.E*)* ∧ (*s*, *s'*) ∈ (*sa.E*)*

```

    apply (induction rule: rtrancl-induct2)
    apply (auto simp: prod-def)
  done
} note gsp-reach=this

have [simp]:  $\bigwedge q s. (q,s) \in g-V0 \text{ prod} \iff q \in igba.V0 \wedge s \in sa.V0$ 
  by (auto simp: prod-def)

have reachSS:
  ((g-E prod)* “ g-V0 prod)
   $\subseteq ((igba.E)* “ igba.V0) \times (sa.E* “ sa.V0)$ 
  by (auto dest: gsp-reach)
show ?thesis
  apply (rule finite-subset[OF reachSS])
  using assms
  by simp
qed

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 = ( $\lambda(q,s). \text{if } s \in sa.V \text{ then } igba.acc \ q \text{ else } \{\}$ )
  unfolding prod-def
  apply simp-all
  done

lemma prod-run: prod.is-run r  $\iff$ 
  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)
  apply rule
  unfolding igba.is-run-def sa.is-run-def prod.is-run-def
  unfolding prod-def ipath-def
  apply (auto split: prod.split-asm intro: in-prod-fst-sndI)
  apply (metis surjective-pairing)
  apply (metis surjective-pairing)
  apply (metis fst-conv snd-conv)
  apply (metis fst-conv snd-conv)
  apply (metis fst-conv snd-conv)
  done

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

```

```

fix  $i$ 
from  $A$  have  $prod.acc (r i) = igba.acc (fst (r i))$ 
  unfolding  $prod\text{-}fields$ 
  apply  $safe$ 
  apply ( $clarsimp\text{-}all\ split: if\text{-}split\text{-}asm$ )
  by ( $metis UNIV\text{-}I comp\text{-}apply imageI snd\text{-}conv subsetD$ )
} note [ $simp$ ] =  $this$ 
show  $?thesis$ 
  unfolding  $prod.is\text{-}acc\text{-}def igba.is\text{-}acc\text{-}def$ 
  by ( $simp add: prod\text{-}fields(4)$ )
qed

lemma  $gsp\text{-}correct1$ :
  assumes  $A: prod.is\text{-}acc\text{-}run r$ 
  shows  $sa.is\text{-}run (snd o r) \wedge (sa.L o snd o r \in igba.lang)$ 
proof –
  from  $A$  have  $PR: prod.is\text{-}run r$  and  $PA: prod.is\text{-}acc r$ 
  unfolding  $prod.is\text{-}acc\text{-}run\text{-}def$  by  $auto$ 

  have  $PRR: range r \subseteq prod.V$  using  $prod.run\text{-}reachable prod.reachable\text{-}V PR$ 
by  $auto$ 

  have  $RSR: range (snd o r) \subseteq sa.V$  using  $PRR$  unfolding  $prod\text{-}fields$  by  $auto$ 

  show  $?thesis$ 
  using  $PR PA$ 
  unfolding  $igba.is\text{-}acc\text{-}run\text{-}def$ 
   $igba.lang\text{-}def igba.accept\text{-}def[abs\text{-}def]$ 
  apply ( $auto simp: prod\text{-}run prod\text{-}acc[OF RSR]$ )
  done
qed

lemma  $gsp\text{-}correct2$ :
  assumes  $A: sa.is\text{-}run r \quad sa.L o r \in igba.lang$ 
  shows  $\exists r'. r = snd o r' \wedge prod.is\text{-}acc\text{-}run r'$ 
proof –
  have [ $simp$ ]:  $\bigwedge r r'. fst o (\lambda i. (r i, r' i)) = r$ 
   $\bigwedge r r'. snd o (\lambda i. (r i, r' i)) = r'$ 
  by  $auto$ 

from  $A$  show  $?thesis$ 
  unfolding  $prod.is\text{-}acc\text{-}run\text{-}def$ 
   $igba.lang\text{-}def igba.accept\text{-}def[abs\text{-}def] igba.is\text{-}acc\text{-}run\text{-}def$ 
  apply ( $clarsimp simp: prod\text{-}run$ )
  apply ( $rename\text{-}tac ra$ )
  apply ( $rule\text{-}tac x=\lambda i. (ra i, r i)$  in  $exI$ )
  apply  $clarsimp$ 

  apply ( $subst prod\text{-}acc$ )

```

```

    using order-trans[OF sa.run-reachable sa.reachable-V]
    apply auto []

    apply auto []
    done
qed

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 $lasso-of-prpl\ prpl \equiv case\ prpl\ of\ (pr,pl) \Rightarrow (\mid$
 $lasso-reach = pr,$
 $lasso-va = hd\ pl,$
 $lasso-cysfx = tl\ pl\ \mid)$

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

lemma $prpl-of-lasso-simps[simp]:$
 $fst\ (prpl-of-lasso\ L) = lasso-reach\ L$
 $snd\ (prpl-of-lasso\ L) = lasso-va\ L \# lasso-cysfx\ L$
unfolding $prpl-of-lasso-def$ **by** $auto$

lemma $lasso-of-prpl-simps[simp]:$
 $lasso-reach\ (lasso-of-prpl\ prpl) = fst\ prpl$
 $snd\ prpl \neq [] \Longrightarrow lasso-cycle\ (lasso-of-prpl\ prpl) = snd\ prpl$
unfolding $lasso-of-prpl-def\ lasso-cycle-def$ **by** $(auto\ split:\ prod.split)$

definition $run-of-lasso :: 'q\ lasso \Rightarrow 'q\ word$
— Run described by a lasso
where $run-of-lasso\ L \equiv lasso-reach\ L \frown (lasso-cycle\ L)^\omega$

lemma $run-of-lasso-of-prpl:$

$pl \neq [] \implies \text{run-of-lasso } (\text{lasso-of-prpl } (pr, pl)) = pr \frown pl^\omega$
unfolding $\text{run-of-lasso-def } \text{lasso-of-prpl-def } \text{lasso-cycle-def}$
by *auto*

definition $\text{map-lasso } f L \equiv ()$
 $\text{lasso-reach} = \text{map } f (\text{lasso-reach } L),$
 $\text{lasso-va} = f (\text{lasso-va } L),$
 $\text{lasso-cysfx} = \text{map } f (\text{lasso-cysfx } L)$
 \rangle

lemma $\text{map-lasso-simps}[simp]:$
 $\text{lasso-reach } (\text{map-lasso } f L) = \text{map } f (\text{lasso-reach } L)$
 $\text{lasso-va } (\text{map-lasso } f L) = f (\text{lasso-va } L)$
 $\text{lasso-cysfx } (\text{map-lasso } f L) = \text{map } f (\text{lasso-cysfx } L)$
 $\text{lasso-v0 } (\text{map-lasso } f L) = f (\text{lasso-v0 } L)$
 $\text{lasso-cycle } (\text{map-lasso } f L) = \text{map } f (\text{lasso-cycle } L)$
unfolding $\text{map-lasso-def } \text{lasso-v0-def } \text{lasso-cycle-def}$
by (*auto split: list.split*)

lemma $\text{map-lasso-run}[simp]:$
shows $\text{run-of-lasso } (\text{map-lasso } f L) = f \circ (\text{run-of-lasso } L)$
by (*auto simp add: map-lasso-def run-of-lasso-def conc-def iter-def*
 $\text{lasso-cycle-def } \text{lasso-v0-def } \text{fun-eq-iff not-less nth-Cons'}$
 nz-le-conv-less)

context *graph begin*

definition $\text{is-lasso-pre} :: 'v \text{lasso} \implies \text{bool}$
where $\text{is-lasso-pre } L \equiv$
 $\text{lasso-v0 } L \in V0$
 $\wedge \text{path } E (\text{lasso-v0 } L) (\text{lasso-reach } L) (\text{lasso-va } L)$
 $\wedge \text{path } E (\text{lasso-va } L) (\text{lasso-cycle } L) (\text{lasso-va } L)$

definition $\text{is-lasso-prpl-pre } prpl \equiv \text{case } prpl \text{ of } (pr, pl) \implies \exists v0 va.$
 $v0 \in V0$
 $\wedge pl \neq []$
 $\wedge \text{path } E v0 pr va$
 $\wedge \text{path } E va pl va$

lemma $\text{is-lasso-pre-prpl-of-lasso}[simp]:$
 $\text{is-lasso-prpl-pre } (prpl\text{-of-lasso } L) \iff \text{is-lasso-pre } L$
unfolding $\text{is-lasso-pre-def } prpl\text{-of-lasso-def } \text{is-lasso-prpl-pre-def}$
unfolding $\text{lasso-v0-def } \text{lasso-cycle-def}$
by (*auto simp: path-simps split: list.split*)

lemma $\text{is-lasso-prpl-pre-conv}:$
 $\text{is-lasso-prpl-pre } prpl$
 $\iff (\text{snd } prpl \neq [] \wedge \text{is-lasso-pre } (\text{lasso-of-prpl } prpl))$

```

unfolding is-lasso-pre-def lasso-of-prpl-def is-lasso-prpl-pre-def
unfolding lasso-v0-def lasso-cycle-def
apply (cases prpl)
apply (rename-tac a b)
apply (case-tac b)
apply (auto simp: path-simps split: list.splits)
done

```

```

lemma is-lasso-pre-empty[simp]:  $V0 = \{\}$   $\implies \neg is-lasso-pre L$ 
unfolding is-lasso-pre-def by auto

```

```

lemma run-of-lasso-pre:
assumes is-lasso-pre L
shows is-run (run-of-lasso L)
and run-of-lasso L 0 ∈ V0
using assms
unfolding is-lasso-pre-def is-run-def run-of-lasso-def
lasso-cycle-def lasso-v0-def
by (auto simp: ipath-conc-conv ipath-iter-conv path-cons-conv conc-fst
split: list.splits)

```

end

context *gb-graph* **begin**

```

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

```

```

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

```

```

lemma is-lasso-prpl-of-lasso[simp]:
is-lasso-prpl (prpl-of-lasso L)  $\longleftrightarrow$  is-lasso L
unfolding is-lasso-def is-lasso-prpl-def
unfolding lasso-v0-def lasso-cycle-def
by auto

```

```

lemma is-lasso-prpl-conv:
is-lasso-prpl prpl  $\longleftrightarrow$  (snd prpl  $\neq []$   $\wedge$  is-lasso (lasso-of-prpl prpl))
unfolding is-lasso-def is-lasso-prpl-def is-lasso-prpl-pre-conv
apply safe
apply simp-all
done

```

lemma *is-lasso-empty*[simp]: $V0 = \{\} \implies \neg is-lasso\ L$
unfolding *is-lasso-def* **by** *auto*

lemma *lasso-accepted*:

assumes $L: is-lasso\ L$

shows *is-acc-run* (*run-of-lasso* L)

proof –

obtain $pr\ va\ pls$ **where**

[simp]: $L = (\!|lasso-reach = pr, lasso-va = va, lasso-cysfx = pls\!|)$

by (*cases* L)

from L **have** *is-run* (*run-of-lasso* L)

unfolding *is-lasso-def* **by** (*auto simp: run-of-lasso-pre*)

moreover from L **have** $(\forall A \in F. set\ (va\#\!pls) \cap A \neq \{\})$

by (*auto simp: is-lasso-def lasso-cycle-def*)

moreover from L **have** (*run-of-lasso* L) $0 \in V0$

unfolding *is-lasso-def* **by** (*auto simp: run-of-lasso-pre*)

ultimately show *is-acc-run* (*run-of-lasso* L)

unfolding *is-acc-run-def is-acc-def run-of-lasso-def*

lasso-cycle-def lasso-v0-def

by (*fastforce intro: limit-inter-INF*)

qed

lemma *lasso-prpl-acc-run*:

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

apply (*clarsimp simp: is-lasso-prpl-conv*)

apply (*drule lasso-accepted*)

apply (*simp add: run-of-lasso-of-prpl*)

done

end

context *gb-graph*

begin

lemma *accepted-lasso*:

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

assumes $A: is-acc-run\ r$

shows $\exists L. is-lasso\ L$

proof –

from A **have**

$RUN: is-run\ r$

and $ACC: \forall A \in F. limit\ r \cap A \neq \{\}$

by (*auto simp: is-acc-run-limit-alt*)

from RUN **have** [*simp*]: $r\ 0 \in V0$ **and** $RUN': ipath\ E\ r$

by (*simp-all add: is-run-def*)

Choose a node that is visited infinitely often

from RUN **have** $RAN-REACH: range\ r \subseteq E^* \text{ “ } V0$

by (auto simp: is-run-def dest: ipath-to-rtrancl)
 hence finite (range r) by (auto intro: finite-subset)
 then obtain u where u ∈ limit r using limit-nonempty by blast
 hence U-REACH: u ∈ E* “ V0 using RAN-REACH limit-in-range by force
 then obtain v0 pr where PR: v0 ∈ V0 path E v0 pr u
 by (auto intro: rtrancl-is-path)
 moreover

Build a path from u to u, that contains nodes from each acceptance set

have $\exists pl. pl \neq [] \wedge \text{path } E \ u \ pl \ u \wedge (\forall A \in F. \text{set } pl \cap A \neq \{\})$
 using finite-F ACC
 proof (induction rule: finite-induct)
 case empty
 from run-limit-two-connectedI[OF RUN' $\langle u \in \text{limit } r \rangle \langle u \in \text{limit } r \rangle$]
 obtain p where [simp]: p ≠ [] and P: path E u p u
 by (rule trancl-is-path)
 thus ?case by blast
 next
 case (insert A F)
 from insert.IH insert.prem obtain pl where
 [simp]: pl ≠ []
 and P: path E u pl u
 and ACC: $(\forall A' \in F. \text{set } pl \cap A' \neq \{\})$
 by auto
 from insert.prem obtain v where VACC: v ∈ A v ∈ limit r by auto
 from run-limit-two-connectedI[OF RUN' $\langle u \in \text{limit } r \rangle \langle v \in \text{limit } r \rangle$]
 obtain p1 where [simp]: p1 ≠ []
 and P1: path E u p1 v
 by (rule trancl-is-path)

 from run-limit-two-connectedI[OF RUN' $\langle v \in \text{limit } r \rangle \langle u \in \text{limit } r \rangle$]
 obtain p2 where [simp]: p2 ≠ []
 and P2: path E v p2 u
 by (rule trancl-is-path)

 note P also note P1 also (path-conc) note P2 finally (path-conc)
 have path E u (pl@p1@p2) u by simp
 moreover from P2 have v ∈ set (p1@p2)
 by (cases p2) (auto simp: path-cons-conv)
 with ACC VACC have $(\forall A' \in \text{insert } A \ F. \text{set } (pl@p1@p2) \cap A' \neq \{\})$ by
 auto
 moreover have pl@p1@p2 ≠ [] by auto
 ultimately show ?case by (intro exI conjI)
 qed
 then obtain pl where pl ≠ [] path E u pl u $(\forall A \in F. \text{set } pl \cap A \neq \{\})$
 by blast
 then obtain pls where path E u (u#pls) u $\forall A \in F. \text{set } (u\#pls) \cap A \neq \{\}$
 by (cases pl) (auto simp add: path-simps)
 ultimately show ?thesis


```

apply –
apply (rule
  exI[where  $x=(\text{lasso-reach} = pr, \text{lasso-va} = u, \text{lasso-cysfx} = pls)$ ])
unfolding is-lasso-def lasso-v0-def lasso-cycle-def is-lasso-pre-def
apply (cases pr)
apply (simp-all add: path-simps)
done
qed
end

```

context *b-graph*

begin

```

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

```

```

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

```

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

```

gbg.is-lasso-pre  $T m = \text{is-lasso-pre}$ 
unfolding gbg.is-lasso-pre-def[abs-def] is-lasso-pre-def[abs-def]
unfolding to-gbg-ext-def
by auto

```

lemma *is-lasso-gbg*:

```

gbg.is-lasso  $T m = \text{is-lasso}$ 
unfolding is-lasso-def[abs-def] gbg.is-lasso-def[abs-def]
apply simp
apply (auto simp: to-gbg-ext-def lasso-cycle-def)
done

```

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  $L$ )  $\longleftrightarrow \text{is-lasso } L$ 
unfolding is-lasso-def is-lasso-prpl-def
unfolding lasso-v0-def lasso-cycle-def
by auto

```

lemma *is-lasso-prpl-conv*:

```

is-lasso-prpl  $prpl \longleftrightarrow (\text{snd } prpl \neq [] \wedge \text{is-lasso } (\text{lasso-of-prpl } prpl))$ 
unfolding is-lasso-def is-lasso-prpl-def is-lasso-prpl-pre-conv
apply safe
apply auto
done

```

lemma *lasso-prpl-acc-run*:
is-lasso-prpl (*pr*, *pl*) \implies *is-acc-run* (*pr* \frown *iter pl*)
apply (*clarsimp simp: is-lasso-prpl-conv*)
apply (*drule lasso-accepted*)
apply (*simp add: run-of-lasso-of-prpl*)
done

end

context *igb-graph* **begin**

definition *is-lasso* *L* \equiv
is-lasso-pre *L*
 $\wedge (\forall i < \text{num-acc}. \exists q \in \text{set} (\text{lasso-cycle } L). i \in \text{acc } q)$

definition *is-lasso-prpl* *L* \equiv
is-lasso-prpl-pre *L*
 $\wedge (\forall i < \text{num-acc}. \exists q \in \text{set} (\text{snd } L). i \in \text{acc } q)$

lemma *is-lasso-prpl-of-lasso*[*simp*]:
is-lasso-prpl (*prpl-of-lasso* *L*) \longleftrightarrow *is-lasso* *L*
unfolding *is-lasso-def is-lasso-prpl-def*
unfolding *lasso-v0-def lasso-cycle-def*
by *auto*

lemma *is-lasso-prpl-conv*:
is-lasso-prpl *prpl* \longleftrightarrow (*snd prpl* $\neq []$) \wedge *is-lasso* (*lasso-of-prpl prpl*)
unfolding *is-lasso-def is-lasso-prpl-def is-lasso-prpl-pre-conv*
apply *safe*
apply *auto*
done

lemma *is-lasso-pre-ext*[*simp*]:
gbg.is-lasso-pre *T m* = *is-lasso-pre*
unfolding *gbg.is-lasso-pre-def[abs-def] is-lasso-pre-def[abs-def]*
unfolding *to-gbg-ext-def*
by *auto*

lemma *is-lasso-gbg*: *gbg.is-lasso* *T m* = *is-lasso*
unfolding *is-lasso-def[abs-def] gbg.is-lasso-def[abs-def]*
apply *simp*
apply (*simp-all add: to-gbg-ext-def*)
apply (*intro ext*)
apply (*fo-rule arg-cong | intro ext*)
apply (*auto simp: F-def accn-def intro!: ext*)
done

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*)
apply (*clarsimp simp: is-lasso-prpl-conv*)
apply (*drule lasso-accepted*)
apply (*simp add: run-of-lasso-of-prpl*)
done

lemma *degen-lasso-sound*:
assumes *A*: *degen.is-lasso* *T m L*
shows *is-lasso* (*map-lasso fst L*)
proof –

from *A* **have**
V0: *lasso-v0 L* \in *degen.V0 T m* **and**
REACH: *path* (*degen.E T m*)
(*lasso-v0 L*) (*lasso-reach L*) (*lasso-va L*) **and**
LOOP: *path* (*degen.E T m*)
(*lasso-va L*) (*lasso-cycle L*) (*lasso-va L*) **and**
ACC: (*set* (*lasso-cycle L*)) \cap *degen.F T m* \neq {}
unfolding *degen.is-lasso-def degen.is-lasso-pre-def* **by** *auto*

{
fix *i*
assume *i < num-acc*
hence $\exists q \in \text{set}(\text{lasso-cycle } L). i \in \text{local.}acc(\text{fst } q) \wedge \text{snd } q = i$
proof (*induction i*)
case 0
thus ?*case* **using** *ACC unfolding degeneralize-ext-def* **by** *fastforce*
next
case (*Suc i*)
then obtain *q* **where** (*q,i*) \in *set* (*lasso-cycle L*) **and** *i* $\in acc$ *q* **by** *auto*
with *LOOP* **obtain** *pl'* **where** *SPL*: *set* (*lasso-cycle L*) = *set pl'*
and *PS*: *path* (*degen.E T m*) (*q,i*) *pl'* (*q,i*)
by (*blast elim: path-loop-shift*)
from *SPL* **have** *pl' \neq []* **by** (*auto simp: lasso-cycle-def*)
then obtain *pl''* **where** [*simp*]: *pl' = (q,i) # pl''*
using *PS* **by** (*cases pl'*) (*auto simp: path-simps*)
then obtain *q2 pl'''* **where**
[*simp*]: *pl'' = (q2,(i + 1) mod num-acc) # pl'''*
using *PS* $\langle i \in acc \ q \rangle$ $\langle Suc \ i < num-acc \rangle$
apply (*cases pl''*)
apply (*auto*
simp: path-simps degeneralize-ext-def
split: if-split-asm)
done
from *PS* **have**
path (*degen.E T m*) (*q2,Suc i*) *pl''* (*q,i*)
using $\langle Suc \ i < num-acc \rangle$

```

    by (auto simp: path-simps)
  from degen-visit-acc[OF this] obtain qa
    where (qa,Suc i)∈set pl''   Suc i ∈ acc qa
    by auto
  thus ?case
    by (rule-tac bexI[where x=(qa,Suc i)]) (auto simp: SPL)
qed
} note aux=this

```

```

from degen-V0-sound[OF V0]
  degen-path-sound[OF REACH]
  degen-path-sound[OF LOOP] aux
show ?thesis
  unfolding is-lasso-def is-lasso-pre-def
  by auto
qed

```

end

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

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

lemma *lasso-rel-ext-sv[relator-props]*:
 $\bigwedge Re R. \llbracket \text{single-valued } Re; \text{single-valued } R \rrbracket \implies \text{single-valued } (\langle Re,R \rangle \text{lasso-rel-ext})$
unfolding *lasso-rel-ext-def*
apply (*rule single-valuedI*)
apply *safe*
apply (*blast dest: single-valuedD list-rel-sv[THEN single-valuedD]*)+
done

lemma *lasso-rel-ext-id*[*relator-props*]:
 $\bigwedge Re R. \llbracket Re=Id; R=Id \rrbracket \implies \langle Re, R \rangle \text{lasso-rel-ext} = Id$
unfolding *lasso-rel-ext-def*
apply *simp*
apply *safe*
by (*metis lasso.surjective*)

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*]:
 $\bigwedge Re R. (\text{lasso-reach}, \text{lasso-reach}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle R \rangle \text{list-rel}$
 $\bigwedge Re R. (\text{lasso-va}, \text{lasso-va}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow R$
 $\bigwedge Re R. (\text{lasso-cysfx}, \text{lasso-cysfx}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle R \rangle \text{list-rel}$
 $\bigwedge Re R. (\text{lasso-ext}, \text{lasso-ext})$
 $\in \langle R \rangle \text{list-rel} \rightarrow R \rightarrow \langle R \rangle \text{list-rel} \rightarrow Re \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso-reach-update}, \text{lasso-reach-update})$
 $\in (\langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso-va-update}, \text{lasso-va-update})$
 $\in (R \rightarrow R) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso-cysfx-update}, \text{lasso-cysfx-update})$
 $\in (\langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
 $\bigwedge Re R. (\text{lasso.more-update}, \text{lasso.more-update})$
 $\in (Re \rightarrow Re) \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle Re, R \rangle \text{lasso-rel-ext}$
unfolding *lasso-rel-ext-def*
by (*auto dest: fun-relD*)

lemma *lasso-param2*[*param, autoref-rules*]:
 $\bigwedge Re R. (\text{lasso-v0}, \text{lasso-v0}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow R$
 $\bigwedge Re R. (\text{lasso-cycle}, \text{lasso-cycle}) \in \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle R \rangle \text{list-rel}$
 $\bigwedge Re R. (\text{map-lasso}, \text{map-lasso})$
 $\in (R \rightarrow R') \rightarrow \langle Re, R \rangle \text{lasso-rel-ext} \rightarrow \langle \text{unit-rel}, R' \rangle \text{lasso-rel-ext}$
unfolding *lasso-v0-def*[*abs-def*] *lasso-cycle-def*[*abs-def*] *map-lasso-def*[*abs-def*]
by *parametricity+*

lemma *lasso-of-prpl-param*: $\llbracket (l', l) \in \langle R \rangle \text{list-rel} \times_r \langle R \rangle \text{list-rel}; \text{snd } l \neq \llbracket \rrbracket$
 $\implies (\text{lasso-of-prpl } l', \text{lasso-of-prpl } l) \in \langle \text{unit-rel}, R \rangle \text{lasso-rel-ext}$
unfolding *lasso-of-prpl-def*
apply (*cases l, cases l', clarsimp*)
apply (*case-tac b, simp, case-tac ba, clarsimp-all*)
apply *parametricity*
done

context begin interpretation *autoref-syn* .

lemma *lasso-of-prpl-autoref*[*autoref-rules*]:
assumes *SIDE-PRECOND* (*snd l* \neq [])
assumes $(l', l) \in \langle R \rangle \text{list-rel} \times_r \langle R \rangle \text{list-rel}$
shows (*lasso-of-prpl* *l'*,
OP *lasso-of-prpl*
 $\text{:: } \langle R \rangle \text{list-rel} \times_r \langle R \rangle \text{list-rel} \rightarrow \langle \text{unit-rel}, R \rangle \text{lasso-rel-ext} \l)
 $\in \langle \text{unit-rel}, R \rangle \text{lasso-rel-ext}$
using *assms*
apply (*simp add: lasso-of-prpl-param*)
done

end

4.1 Implementing runs by lassos

definition *lasso-run-rel-def-internal*:
 $\text{lasso-run-rel } R \equiv \text{br run-of-lasso } (\lambda-. \text{True}) \text{ } O \text{ } (\text{nat-rel} \rightarrow R)$
lemma *lasso-run-rel-def*:
 $\langle R \rangle \text{lasso-run-rel} = \text{br run-of-lasso } (\lambda-. \text{True}) \text{ } O \text{ } (\text{nat-rel} \rightarrow R)$
unfolding *lasso-run-rel-def-internal* *relAPP-def* **by** *simp*

lemma *lasso-run-rel-sv*[*relator-props*]:
 $\text{single-valued } R \implies \text{single-valued } (\langle R \rangle \text{lasso-run-rel})$
unfolding *lasso-run-rel-def*
by *tagged-solver*

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-map-run* **by** *simp*

lemma *map-lasso-run-refine*[*autoref-rules*]:
shows $(\text{map-lasso}, \text{op-map-run}) \in (R \rightarrow R') \rightarrow \langle R \rangle \text{lasso-run-rel} \rightarrow \langle R' \rangle \text{lasso-run-rel}$
unfolding *lasso-run-rel-def* *op-map-run-def*
proof (*intro fun-relI*)
fix *f f' l r*
assume [*param*]: $(f, f') \in R \rightarrow R'$ **and**
 $(l, r) \in \text{br run-of-lasso } (\lambda-. \text{True}) \text{ } O \text{ } (\text{nat-rel} \rightarrow R)$
then obtain *r'* **where** [*param*]: $(r', r) \in \text{nat-rel} \rightarrow R$
and [*simp*]: $r' = \text{run-of-lasso } l$
by (*auto simp: br-def*)

have $(\text{map-lasso } f \ l, f \ o \ r') \in \text{br run-of-lasso } (\lambda-. \text{True})$

```

    by (simp add: br-def)
  also have  $(f \circ r', f' \circ r) \in \text{nat-rel} \rightarrow R'$  by parametricity
  finally (relcompI) show
     $(\text{map-lasso } f \ l, f' \circ r) \in \text{br run-of-lasso } (\lambda-. \text{ True}) \ O \ (\text{nat-rel} \rightarrow R')$ 
  .
qed
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  $(R''A)$ 
proof -
  note [[simproc add: finite-Collect]]
  have  $R''A = \bigcup \{R''\{a\} \mid a. a:A\}$ 
    by auto
  also have finite  $(\bigcup \{R''\{a\} \mid a. a:A\})$ 
    apply (rule finite-Union)
    apply (simp add: assms)
    apply (clarsimp simp: assms)
    done
  finally show ?thesis .
qed

```

6 Simulation

6.1 Functional Relations

```

definition the-br- $\alpha$  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-br- $\alpha$  R) (the-br-invar R) = R
  unfolding build-rel-def the-br- $\alpha$ -def
  apply auto
  apply (metis someI-ex)
  apply (metis assms someI-ex single-valuedD)
  done

```

```

lemma the-br-br[simp]:

```

$I x \implies \text{the-br-}\alpha (br \ \alpha \ I) x = \alpha x$
 $\text{the-br-invar } (br \ \alpha \ I) = I$
unfolding $\text{the-br-}\alpha\text{-def build-rel-def[abs-def]}$
by *auto*

6.2 Relation between Runs

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

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

unfolding run-rel-def **by** *auto*

lemma $\text{run-rel-single-valued: single-valued } R$
 $\implies (ra, rb) \in \text{run-rel } R \iff ((\forall i. \text{the-br-invar } R (ra \ i)) \wedge rb = \text{the-br-}\alpha \ R \ o \ ra)$

unfolding $\text{run-rel-def the-br-}\alpha\text{-def}$

apply (*auto intro!: ext*)

apply (*metis single-valuedD someI-ex*)

apply (*metis DomainE someI-ex*)

done

6.3 Simulation

locale $\text{simulation} =$

$a: \text{graph } A +$

$b: \text{graph } B$

for $R :: ('a \times 'b) \text{ set}$

and $A :: ('a, -) \text{ graph-rec-scheme}$

and $B :: ('b, -) \text{ graph-rec-scheme}$

$+$

assumes $\text{nodes-sim}: a \in a.V \implies (a, b) \in R \implies b \in b.V$

assumes $\text{init-sim}: a0 \in a.V0 \implies \exists b0. b0 \in b.V0 \wedge (a0, b0) \in R$

assumes $\text{step-sim}: (a, a') \in a.E \implies (a, b) \in R \implies \exists b'. (b, b') \in b.E \wedge (a', b') \in R$

begin

lemma $\text{simulation-this: simulation } R \ A \ B$ **by** *unfold-locales*

lemma run-sim:

assumes $\text{arun}: a.\text{is-run } ra$

obtains rb **where** $b.\text{is-run } rb \quad (ra, rb) \in \text{run-rel } R$

proof $-$

from arun **have** $\text{ainit}: ra \ 0 \in a.V0$

and $\text{astep}: \forall i. (ra \ i, ra \ (\text{Suc } i)) \in a.E$

using $a.\text{run-V0 } a.\text{run-ipath } ipathD$ **by** *blast+*

from init-sim **obtain** $rb0$ **where** $\text{rel0}: (ra \ 0, rb0) \in R$ **and** $\text{binit}: rb0 \in b.V0$

by (*auto intro: ainit*)


```

define rb
  where rb = rec-nat rb0 ( $\lambda i$  rbi. SOME rbsi. (rbi, rbsi)  $\in$  b.E  $\wedge$  (ra (Suc
i), rbsi)  $\in$  R)

  have [simp]:
    rb 0 = rb0
     $\bigwedge i$ . rb (Suc i) = (SOME rbsi. (rb i, rbsi)  $\in$  b.E  $\wedge$  (ra (Suc i), rbsi)  $\in$  R)
    unfolding rb-def by auto

  {
    fix i
    have (rb i, rb (Suc i))  $\in$  b.E  $\wedge$  (ra (Suc i), rb (Suc i))  $\in$  R
    proof (induction i)
      case 0
      from step-sim astep rel0 obtain rb1 where (rb 0, rb1)  $\in$  b.E and (ra 1,
rb1)  $\in$  R
      by fastforce
      thus ?case by (auto intro!: someI)
    next
      case (Suc i)
      with step-sim astep obtain rbss where (rb (Suc i), rbss)  $\in$  b.E and
        (ra (Suc (Suc i)), rbss)  $\in$  R
      by fastforce
      thus ?case by (auto intro!: someI)
    qed
  } note aux=this

from aux binit have b.is-run rb
  unfolding b.is-run-def ipath-def by simp
moreover from aux rel0 have (ra, rb)  $\in$  run-rel R
  unfolding run-rel-def
  apply safe
  apply (case-tac i)
  by auto
ultimately show ?thesis by rule
qed

```

```

lemma stuck-sim:
  assumes (a, b)  $\in$  R
  assumes b  $\notin$  Domain b.E
  shows a  $\notin$  Domain a.E
  using assms
  by (auto dest: step-sim)

```

```

lemma run-Domain: a.is-run r  $\implies$  r  $\in$  Domain R
  by (erule run-sim) (auto simp: run-rel-def)

```

```

lemma br-run-sim:
  assumes R = br  $\alpha$  I

```

```

assumes a.is-run r
shows b.is-run ( $\alpha$  o r)
using assms
apply –
apply (erule run-sim)
apply (auto simp: run-rel-def build-rel-def a.is-run-def b.is-run-def ipath-def)
done

```

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

```

apply safe
apply (erule rtrancl-induct)
apply (metis ImageI init-sim rtrancl.rtrancl-refl)
apply (metis rtrancl-image-advance step-sim)
done

```

lemma *reachable-sim*: $a.E^* \text{ `` } a.V0 \subseteq R^{-1} \text{ `` } b.E^* \text{ `` } b.V0$
using *is-reachable-sim* **by** *blast*

lemma *reachable-finite-sim*:
assumes *finite* ($b.E^* \text{ `` } b.V0$)
assumes $\bigwedge b. b \in b.E^* \text{ `` } b.V0 \implies \text{finite } (R^{-1} \text{ `` } \{b\})$
shows *finite* ($a.E^* \text{ `` } a.V0$)
apply (*rule finite-subset[OF reachable-sim]*)
apply (*rule finite-ImageI*)
apply *fact+*
done

end

lemma *simulation-trans*[*trans*]:
assumes *simulation* *R1* *A* *B*
assumes *simulation* *R2* *B* *C*
shows *simulation* (*R1* *O* *R2*) *A* *C*
proof –
interpret *s1: simulation* *R1* *A* *B* **by** *fact*
interpret *s2: simulation* *R2* *B* *C* **by** *fact*
show *?thesis*
apply *unfold-locales*
using *s1.nodes-sim s2.nodes-sim* **apply** *blast*
using *s1.init-sim s2.init-sim* **apply** *blast*
using *s1.step-sim s2.step-sim* **apply** *blast*
done

qed

lemma (**in** *graph*) *simulation-refl*[*simp*]: *simulation* *Id* *G* *G* **by** *unfold-locales*
auto

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 ⇒ a.L a = b.L b
begin

lemma lsimulation-this: lsimulation R A B by unfold-locales

lemma run-rel-consistent: (ra, rb) ∈ run-rel R ⇒ a.L o ra = b.L o rb
  using labeling-consistent unfolding run-rel-def
  by auto

lemma accept-sim: a.accept w ⇒ b.accept w
  unfolding a.accept-def b.accept-def
  apply clarsimp
  apply (erule run-sim)
  apply (auto simp: run-rel-consistent)
  done

end

lemma lsimulation-trans[trans]:
  assumes lsimulation R1 A B
  assumes lsimulation R2 B C
  shows lsimulation (R1 O R2) A C
proof -
  interpret s1: lsimulation R1 A B by fact
  interpret s2: lsimulation R2 B C by fact
  interpret simulation R1 O R2 A C
  using simulation-trans s1.simulation-this s2.simulation-this by this
  show ?thesis
  apply unfold-locales
  using s1.labeling-consistent s2.labeling-consistent
  by auto
qed

lemma (in sa) lsimulation-refl[simp]: lsimulation Id G G by unfold-locales auto

```

6.4 Bisimulation

```

locale bisimulation =
  a: graph A +
  b: graph B +
  s1: simulation R A B +
  s2: simulation R-1 B A

```

```

for  $R :: ('a \times 'b)$  set
and  $A :: ('a, -)$  graph-rec-scheme
and  $B :: ('b, -)$  graph-rec-scheme
begin

  lemma bisimulation-this: bisimulation  $R A B$  by unfold-locales

  lemma converse: bisimulation  $(R^{-1}) B A$ 
  proof –
    interpret simulation  $(R^{-1})^{-1} A B$  by simp unfold-locales
    show ?thesis by unfold-locales
  qed

  lemma br-run-conv:
    assumes  $R = br \ \alpha \ I$ 
    shows  $b.is-run \ rb \longleftrightarrow (\exists \ ra. \ rb = \alpha \ o \ ra \ \wedge \ a.is-run \ ra)$ 
    using assms
    apply safe
    apply (erule s2.run-sim, auto
      intro!: ext
      simp: run-rel-def build-rel-def) []
    apply (erule s1.br-run-sim, assumption)
    done

  lemma bri-run-conv:
    assumes  $R = (br \ \gamma \ I)^{-1}$ 
    shows  $a.is-run \ ra \longleftrightarrow (\exists \ rb. \ ra = \gamma \ o \ rb \ \wedge \ b.is-run \ rb)$ 
    using assms
    apply safe
    apply (erule s1.run-sim, auto simp: run-rel-def build-rel-def intro!: ext) []

    apply (erule s2.run-sim, auto simp: run-rel-def build-rel-def)
    by (metis (no-types, opaque-lifting) fun-comp-eq-conv)

  lemma inj-map-run-eq:
    assumes inj  $\alpha$ 
    assumes  $E: \alpha \ o \ r1 = \alpha \ o \ r2$ 
    shows  $r1 = r2$ 
  proof (rule ext)
    fix  $i$ 
    from  $E$  have  $\alpha \ (r1 \ i) = \alpha \ (r2 \ i)$ 
      by (simp add: comp-def) metis
    with  $\langle inj \ \alpha \rangle$  show  $r1 \ i = r2 \ i$ 
      by (auto dest: injD)
  qed

  lemma br-inj-run-conv:
    assumes INJ: inj  $\alpha$ 
    assumes [simp]:  $R = br \ \alpha \ I$ 

```

shows $b.is-run (\alpha o ra) \longleftrightarrow a.is-run ra$
apply (*subst br-run-conv*[*OF assms*(2)])
apply (*auto dest: inj-map-run-eq*[*OF INJ*])
done

lemma *single-valued-run-conv*:
assumes *single-valued R*
shows $b.is-run rb$
 $\longleftrightarrow (\exists ra. rb=the-br-\alpha R o ra \wedge a.is-run ra)$
using *assms*
apply *safe*
apply (*erule s2.run-sim*)
apply (*auto simp add: run-rel-single-valued*)
apply (*erule s1.run-sim*)
apply (*auto simp add: run-rel-single-valued*)
done

lemma *stuck-bisim*:
assumes $A: (a, b) \in R$
shows $a \in Domain a.E \longleftrightarrow b \in Domain b.E$
using *s1.stuck-sim*[*OF A*]
using *s2.stuck-sim*[*OF A*[*THEN converseI*[*of - - R*]]]
by *blast*

end

lemma *bisimulation-trans*[*trans*]:
assumes *bisimulation R1 A B*
assumes *bisimulation R2 B C*
shows *bisimulation (R1 O R2) A C*
proof –
interpret *s1: bisimulation R1 A B by fact*
interpret *s2: bisimulation R2 B C by fact*
interpret *t1: simulation (R1 O R2) A C*
using *simulation-trans s1.s1.simulation-this s2.s1.simulation-this by this*
interpret *t2: simulation (R1 O R2)⁻¹ C A*
using *simulation-trans s2.s2.simulation-this s1.s2.simulation-this*
unfolding *converse-relcomp*
by *this*
show *?thesis by unfold-locales*
qed

lemma (*in graph*) *bisimulation-refl*[*simp*]: *bisimulation Id G G by unfold-locales*
auto

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 × 'b) set
and A :: ('a, 'l, -) sa-rec-scheme
and B :: ('b, 'l, -) sa-rec-scheme
begin

lemma lsimulation-this: lbisimulation R A B by unfold-locales

lemma accept-bisim: a.accept = b.accept
  apply (rule ext)
  using s1.accept-sim s2.accept-sim by blast

end

lemma lsimulation-trans[trans]:
  assumes lbisimulation R1 A B
  assumes lbisimulation R2 B C
  shows lbisimulation (R1 O R2) A C
proof -
  interpret s1: lbisimulation R1 A B by fact
  interpret s2: lbisimulation R2 B C by fact

  from lsimulation-trans[OF s1.s1.lsimulation-this s2.s1.lsimulation-this]
  interpret t1: lsimulation (R1 O R2) A C .

  from lsimulation-trans[OF s2.s2.lsimulation-this s1.s2.lsimulation-this, folded
converse-relcomp]
  interpret t2: lsimulation (R1 O R2)-1 C A .

  show ?thesis by unfold-locales
qed

lemma (in sa) lbisimulation-refl[simp]: lbisimulation Id G G by unfold-locales
auto

end
theory Step-Conv
imports Main
begin

definition rel-of-pred s ≡ {(a,b). s a b}
definition rel-of-succ s ≡ {(a,b). b ∈ s a}

definition pred-of-rel s ≡ λa b. (a,b) ∈ s
definition pred-of-succ s ≡ λa b. b ∈ s a

definition succ-of-rel s ≡ λa. {b. (a,b) ∈ s}

```

definition $\text{succ-of-pred } s \equiv \lambda a. \{b. s a b\}$

lemma $\text{rps-expand}[simp]$:

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

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

$b \in \text{succ-of-rel } r a \longleftrightarrow (a,b) \in r$
 $b \in \text{succ-of-pred } p a \longleftrightarrow p a b$
unfolding $\text{rel-of-pred-def pred-of-rel-def}$
unfolding $\text{rel-of-succ-def succ-of-rel-def}$
unfolding $\text{pred-of-succ-def succ-of-pred-def}$
by *auto*

lemma $\text{rps-conv}[simp]$:

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

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

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

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

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

$\text{succ-of-pred } (\text{pred-of-succ } s) = s$
 $\text{succ-of-pred } (\text{pred-of-rel } r) = \text{succ-of-rel } r$
unfolding $\text{rel-of-pred-def pred-of-rel-def}$
unfolding $\text{rel-of-succ-def succ-of-rel-def}$
unfolding $\text{pred-of-succ-def succ-of-pred-def}$
by *auto*

definition $\text{m2r-rel} :: ('a \times 'a \text{ option}) \text{ set} \Rightarrow 'a \text{ option rel}$
where $\text{m2r-rel } r \equiv \{(Some a,b) \mid a b. (a,b) \in r\}$

definition $\text{m2r-pred} :: ('a \Rightarrow 'a \text{ option} \Rightarrow \text{bool}) \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow \text{bool}$
where $\text{m2r-pred } p \equiv \lambda \text{None} \Rightarrow \lambda -. \text{False} \mid \text{Some } a \Rightarrow p a$

definition $\text{m2r-succ} :: ('a \Rightarrow 'a \text{ option set}) \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option set}$
where $\text{m2r-succ } s \equiv \lambda \text{None} \Rightarrow \{\} \mid \text{Some } a \Rightarrow s a$

lemma *m2r-expand[simp]*:
 $(a,b) \in m2r\text{-rel } r \iff (\exists a'. a = \text{Some } a' \wedge (a',b) \in r)$
 $m2r\text{-pred } p \ a \ b \iff (\exists a'. a = \text{Some } a' \wedge p \ a' \ b)$
 $b \in m2r\text{-succ } s \ a \iff (\exists a'. a = \text{Some } a' \wedge b \in s \ a')$
unfolding *m2r-rel-def m2r-succ-def m2r-pred-def*
by (*auto split: option.splits*)

lemma *m2r-conv[simp]*:
 $m2r\text{-rel } (\text{rel-of-succ } s) = \text{rel-of-succ } (m2r\text{-succ } s)$
 $m2r\text{-rel } (\text{rel-of-pred } p) = \text{rel-of-pred } (m2r\text{-pred } p)$

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

 $m2r\text{-succ } (\text{succ-of-pred } p) = \text{succ-of-pred } (m2r\text{-pred } p)$
 $m2r\text{-succ } (\text{succ-of-rel } r) = \text{succ-of-rel } (m2r\text{-rel } r)$
unfolding *rel-of-pred-def pred-of-rel-def*
unfolding *rel-of-succ-def succ-of-rel-def*
unfolding *pred-of-succ-def succ-of-pred-def*
unfolding *m2r-rel-def m2r-succ-def m2r-pred-def*
by (*auto split: option.splits*)

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

lemma *x-of-enex-expand[simp]*:
 $(s, s') \in \text{rel-of-enex } (en, ex) \iff (\exists a \in en \ s. s' = ex \ a \ s)$
 $\text{pred-of-enex } (en, ex) \ s \ s' \iff (\exists a \in en \ s. s' = ex \ a \ s)$
 $s' \in \text{succ-of-enex } (en, ex) \ s \iff (\exists a \in en \ s. s' = ex \ a \ s)$
unfolding *rel-of-enex-def pred-of-enex-def succ-of-enex-def* **by** *auto*

lemma *x-of-enex-conv[simp]*:
 $\text{rel-of-pred } (\text{pred-of-enex } enex) = \text{rel-of-enex } enex$
 $\text{rel-of-succ } (\text{succ-of-enex } enex) = \text{rel-of-enex } enex$
 $\text{pred-of-rel } (\text{rel-of-enex } enex) = \text{pred-of-enex } enex$
 $\text{pred-of-succ } (\text{succ-of-enex } enex) = \text{pred-of-enex } enex$
 $\text{succ-of-rel } (\text{rel-of-enex } enex) = \text{succ-of-enex } enex$
 $\text{succ-of-pred } (\text{pred-of-enex } enex) = \text{succ-of-enex } enex$
unfolding *rel-of-enex-def pred-of-enex-def succ-of-enex-def*
unfolding *rel-of-pred-def rel-of-succ-def*
unfolding *pred-of-rel-def pred-of-succ-def*
unfolding *succ-of-rel-def succ-of-pred-def*
by *auto*

end
theory *Stuttering-Extension*

imports *Simulation Step-Conv*
begin

definition *stutter-extend-edges* :: 'v set \Rightarrow 'v digraph \Rightarrow 'v digraph
where *stutter-extend-edges* V E \equiv E \cup {(v, v) | v. v \in V \wedge v \notin Domain E}

lemma *stutter-extend-edgesI-edge*:
assumes (u, v) \in E
shows (u, v) \in *stutter-extend-edges* V E
using *assms unfolding stutter-extend-edges-def* **by** *auto*

lemma *stutter-extend-edgesI-stutter*:
assumes v \in V v \notin Domain E
shows (v, v) \in *stutter-extend-edges* V E
using *assms unfolding stutter-extend-edges-def* **by** *auto*

lemma *stutter-extend-edgesE*:
assumes (u, v) \in *stutter-extend-edges* V E
obtains (edge) (u, v) \in E | (stutter) u \in V u \notin Domain E u = v
using *assms unfolding stutter-extend-edges-def* **by** *auto*

lemma *stutter-extend-wf*: E \subseteq V \times V \Longrightarrow *stutter-extend-edges* V E \subseteq V \times V
unfolding *stutter-extend-edges-def* **by** *auto*

lemma *stutter-extend-edges-rtrancl[simp]*: (*stutter-extend-edges* V E)* = E*
unfolding *stutter-extend-edges-def* **by** (*auto intro: in-rtrancl-UnI elim: rtrancl-induct*)

lemma *stutter-extend-domain*: V \subseteq Domain (*stutter-extend-edges* V E)
unfolding *stutter-extend-edges-def* **by** *auto*

definition *stutter-extend* :: ('v, -) graph-rec-scheme \Rightarrow ('v, -) graph-rec-scheme
where *stutter-extend* G \equiv
 (

 g-V = g-V G,
 g-E = *stutter-extend-edges* (g-V G) (g-E G),
 g-V0 = g-V0 G,
 ... = *graph-rec.more* G
)

lemma *stutter-extend-simps[simp]*:
 g-V (*stutter-extend* G) = g-V G
 g-E (*stutter-extend* G) = *stutter-extend-edges* (g-V G) (g-E G)
 g-V0 (*stutter-extend* G) = g-V0 G
unfolding *stutter-extend-def* **by** *auto*

lemma *stutter-extend-simps-sa[simp]*:
 sa-L (*stutter-extend* G) = sa-L G
unfolding *stutter-extend-def*
by (*metis graph-rec.select-convs(4) sa-rec.select-convs(1) sa-rec.surjective*)

lemma (**in** graph) *stutter-extend-graph*: graph (*stutter-extend* G)

```

    using V0-ss E-ss by (unfold-locales, auto intro!: stutter-extend-wf)
lemma (in sa) stutter-extend-sa: sa (stutter-extend G)
proof -
  interpret graph stutter-extend G using stutter-extend-graph by this
  show ?thesis by unfold-locales
qed

lemma (in bisimulation) stutter-extend: bisimulation R (stutter-extend A) (stutter-extend
B)
proof -
  interpret ea: graph stutter-extend A by (rule a.stutter-extend-graph)
  interpret eb: graph stutter-extend B by (rule b.stutter-extend-graph)
  show ?thesis
  proof
    fix a b
    assume a ∈ g-V (stutter-extend A) (a, b) ∈ R
    thus b ∈ g-V (stutter-extend B) using s1.nodes-sim by simp
  next
    fix a
    assume a ∈ g-V0 (stutter-extend A)
    thus ∃ b. b ∈ g-V0 (stutter-extend B) ∧ (a, b) ∈ R using s1.init-sim by
simp
  next
    fix a a' b
    assume (a, a') ∈ g-E (stutter-extend A) (a, b) ∈ R
    thus ∃ b'. (b, b') ∈ g-E (stutter-extend B) ∧ (a', b') ∈ R
    apply simp
    using s1.nodes-sim s1.step-sim s2.stuck-sim
    by (blast
      intro: stutter-extend-edgesI-edge stutter-extend-edgesI-stutter
      elim: stutter-extend-edgesE)
  next
    fix b a
    assume b ∈ g-V (stutter-extend B) (b, a) ∈ R-1
    thus a ∈ g-V (stutter-extend A) using s2.nodes-sim by simp
  next
    fix b
    assume b ∈ g-V0 (stutter-extend B)
    thus ∃ a. a ∈ g-V0 (stutter-extend A) ∧ (b, a) ∈ R-1 using s2.init-sim by
simp
  next
    fix b b' a
    assume (b, b') ∈ g-E (stutter-extend B) (b, a) ∈ R-1
    thus ∃ a'. (a, a') ∈ g-E (stutter-extend A) ∧ (b', a') ∈ R-1
    apply simp
    using s2.nodes-sim s2.step-sim s1.stuck-sim
    by (blast
      intro: stutter-extend-edgesI-edge stutter-extend-edgesI-stutter
      elim: stutter-extend-edgesE)
  
```

qed
qed

lemma (in *lbisimulation*) *lstutter-extend*: *lbisimulation* *R* (*stutter-extend* *A*)
(*stutter-extend* *B*)

proof –

interpret *se*: *bisimulation* *R* *stutter-extend* *A* *stutter-extend* *B* **by** (rule
stutter-extend)

show *?thesis* **by** (*unfold-locales*, *auto simp*: *s1.labeling-consistent*)

qed

definition *stutter-extend-en* :: ('s ⇒ 'a set) ⇒ ('s ⇒ 'a option set) **where**
stutter-extend-en *en* ≡ λ*s*. let *as* = *en s* in if *as* = {} then {None} else Some 'as

definition *stutter-extend-ex* :: ('a ⇒ 's ⇒ 's) ⇒ ('a option ⇒ 's ⇒ 's) **where**
stutter-extend-ex *ex* ≡ λNone ⇒ id | Some *a* ⇒ *ex a*

abbreviation *stutter-extend-enex*

:: ('s ⇒ 'a set) × ('a ⇒ 's ⇒ 's) ⇒ ('s ⇒ 'a option set) × ('a option ⇒ 's ⇒ 's)

where

stutter-extend-enex ≡ *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)

unfolding *rel-of-enex-def* *stutter-extend-edges-def*

apply (*auto simp*: *stutter-extend-en-def* *stutter-extend-ex-def* *split*: *prod.splits*)

apply *force*

apply *blast*

done

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

Some a ∈ *stutter-extend-en* *en* *gc* ↔ *a* ∈ *en* *gc*

stutter-extend-ex *ex* (*Some a*) *gc* = *ex a* *gc*

unfolding *stutter-extend-en-def* *stutter-extend-ex-def* **by** *auto*

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

stutter-extend-ex *ex* None = *id*

unfolding *stutter-extend-ex-def* **by** *auto*

end

7 Implementing Graphs

theory *Digraph-Impl*

imports *Digraph*

begin

7.1 Directed Graphs by Successor Function

type-synonym 'a slg = 'a ⇒ 'a list

definition *slg-rel* :: ('a × 'b) set ⇒ ('a slg × 'b digraph) set **where**
slg-rel-def-internal: *slg-rel* R ≡
 (R → ⟨R⟩list-set-rel) O br (λsuccs. {(u,v). v ∈ succs u}) (λ-. True)

lemma *slg-rel-def*: ⟨R⟩slg-rel =
 (R → ⟨R⟩list-set-rel) O br (λsuccs. {(u,v). v ∈ succs u}) (λ-. True)
unfolding *slg-rel-def-internal* *relAPP-def* **by** *simp*

lemma *slg-rel-sv*[*relator-props*]:
 [single-valued R; Range R = UNIV] ⇒ single-valued (⟨R⟩slg-rel)
unfolding *slg-rel-def*
by (*tagged-solver*)

consts *i-slg* :: interface ⇒ interface

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

definition [*simp*]: *op-slg-succs* E v ≡ E^{“{v}}

lemma [*autoref-itype*]: *op-slg-succs* ::_i ⟨I⟩_ii-slg →_i I →_i ⟨I⟩_ii-set **by** *simp*

context begin interpretation *autoref-syn* .

lemma [*autoref-op-pat*]: E^{“{v}} ≡ *op-slg-succs* \$E\$ v **by** *simp*
end

lemma *refine-slg-succs*[*autoref-rules-raw*]:
 (λsuccs v. succs v, *op-slg-succs*) ∈ ⟨R⟩slg-rel → R → ⟨R⟩list-set-rel
apply (*intro fun-relI*)
apply (*auto simp add: slg-rel-def br-def dest: fun-relD*)
done

definition *E-of-succ* succ ≡ { (u,v). v ∈ succ u }

definition *succ-of-E* E ≡ (λu. {v . (u,v) ∈ E})

lemma *E-of-succ-of-E*[*simp*]: *E-of-succ* (*succ-of-E* E) = E
unfolding *E-of-succ-def succ-of-E-def*
by *auto*

lemma *succ-of-E-of-succ*[*simp*]: *succ-of-E* (*E-of-succ* E) = E
unfolding *E-of-succ-def succ-of-E-def*
by *auto*

context begin interpretation *autoref-syn* .

lemma [*autoref-itype*]: *E-of-succ* ::_i (I →_i ⟨I⟩_ii-set) →_i ⟨I⟩_ii-slg **by** *simp*

lemma [*autoref-itype*]: *succ-of-E* ::_i ⟨I⟩_ii-slg →_i I →_i ⟨I⟩_ii-set **by** *simp*

end

lemma *E-of-succ-refine*[*autoref-rules*]:

$(\lambda x. x, E\text{-of-succ}) \in (R \rightarrow \langle R \rangle \text{list-set-rel}) \rightarrow \langle R \rangle \text{slg-rel}$

$(\lambda x. x, \text{succ-of-}E) \in \langle R \rangle \text{slg-rel} \rightarrow (R \rightarrow \langle R \rangle \text{list-set-rel})$

unfolding *E-of-succ-def*[*abs-def*] *succ-of-E-def*[*abs-def*] *slg-rel-def* *br-def*

apply *auto* []

apply *clarsimp*

apply (*blast dest: fun-relD*)

done

7.1.1 Restricting Edges

definition *op-graph-restrict* :: $'v \text{ set} \Rightarrow 'v \text{ set} \Rightarrow ('v \times 'v) \text{ set} \Rightarrow ('v \times 'v) \text{ set}$

where [*simp*]: $op\text{-graph-restrict } Vl \ Vr \ E \equiv E \cap Vl \times Vr$

definition *op-graph-restrict-left* :: $'v \text{ set} \Rightarrow ('v \times 'v) \text{ set} \Rightarrow ('v \times 'v) \text{ set}$

where [*simp*]: $op\text{-graph-restrict-left } Vl \ E \equiv E \cap Vl \times UNIV$

definition *op-graph-restrict-right* :: $'v \text{ set} \Rightarrow ('v \times 'v) \text{ set} \Rightarrow ('v \times 'v) \text{ set}$

where [*simp*]: $op\text{-graph-restrict-right } Vr \ E \equiv E \cap UNIV \times Vr$

lemma [*autoref-op-pat*]:

$E \cap (Vl \times Vr) \equiv op\text{-graph-restrict } Vl \ Vr \ E$

$E \cap (Vl \times UNIV) \equiv op\text{-graph-restrict-left } Vl \ E$

$E \cap (UNIV \times Vr) \equiv op\text{-graph-restrict-right } Vr \ E$

by *simp-all*

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

$E\text{-of-succ } (\lambda v. \text{if } v \in Vl \text{ then } \{x \in E''\{v\}. x \in Vr\} \text{ else } \{\})$

by (*auto simp: E-of-succ-def succ-of-E-def split: if-split-asm*)

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

$E\text{-of-succ } (\lambda v. \text{if } v \in Vl \text{ then } E''\{v\} \text{ else } \{\})$

by (*auto simp: E-of-succ-def succ-of-E-def split: if-split-asm*)

lemma *graph-restrict-right-aimpl*: $op\text{-graph-restrict-right } Vr \ E =$

$E\text{-of-succ } (\lambda v. \{x \in E''\{v\}. x \in Vr\})$

by (*auto simp: E-of-succ-def succ-of-E-def split: if-split-asm*)

schematic-goal *graph-restrict-impl-aux*:

fixes *Rsl Rsr*

notes [*autoref-rel-intf*] = $REL\text{-INTFI}[of \ Rsl \ i\text{-set}] \ REL\text{-INTFI}[of \ Rsr \ i\text{-set}]$

assumes [*autoref-rules*]: $(\text{meml}, (\in)) \in R \rightarrow \langle R \rangle Rsl \rightarrow \text{bool-rel}$

assumes [*autoref-rules*]: $(\text{memr}, (\in)) \in R \rightarrow \langle R \rangle Rsr \rightarrow \text{bool-rel}$

shows (*?c*, *op-graph-restrict*) $\in \langle R \rangle Rsl \rightarrow \langle R \rangle Rsr \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \langle R \rangle \text{slg-rel}$

unfolding *graph-restrict-aimpl*[*abs-def*]

apply (*autoref (keep-goal)*)

done

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 $[autoref-rules]: (meml, (\in)) \in R \rightarrow \langle R \rangle Rsl \rightarrow bool-rel$
shows $(?c, op-graph-restrict-left) \in \langle R \rangle Rsl \rightarrow \langle R \rangle slg-rel \rightarrow \langle R \rangle slg-rel$
unfolding $graph-restrict-left-aimpl[abs-def]$
apply $(autoref\ (keep-goal, trace))$
done

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 $[autoref-rules]: (memr, (\in)) \in R \rightarrow \langle R \rangle Rsr \rightarrow bool-rel$
shows $(?c, op-graph-restrict-right) \in \langle R \rangle Rsr \rightarrow \langle R \rangle slg-rel \rightarrow \langle R \rangle slg-rel$
unfolding $graph-restrict-right-aimpl[abs-def]$
apply $(autoref\ (keep-goal, trace))$
done

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 $autoref-syn$.

lemma $[autoref-itype]:$
 $op-graph-restrict ::_i \langle I \rangle_i i-set \rightarrow_i \langle I \rangle_i i-set \rightarrow_i \langle I \rangle_i i-slg \rightarrow_i \langle I \rangle_i i-slg$
 $op-graph-restrict-right ::_i \langle I \rangle_i i-set \rightarrow_i \langle I \rangle_i i-slg \rightarrow_i \langle I \rangle_i i-slg$
 $op-graph-restrict-left ::_i \langle I \rangle_i i-set \rightarrow_i \langle I \rangle_i i-slg \rightarrow_i \langle I \rangle_i i-slg$
by $auto$

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 $(?c::?c, \lambda(E::nat\ digraph)\ x.\ E'\{x\}) \in ?R$

apply $(autoref\ (keep-goal))$
done

lemma $graph-minus-aimpl:$

fixes $E1\ E2 :: 'a\ rel$
shows $E1 - E2 = E-of-succ\ (\lambda x.\ E1'\{x\} - E2'\{x\})$
by $(auto\ simp:\ E-of-succ-def)$

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

fixes $R :: ('vi \times 'v)\ set$
assumes $[autoref-rules]: (eq, (=)) \in R \rightarrow R \rightarrow bool-rel$
shows $(?c, (-)) \in \langle R \rangle slg-rel \rightarrow \langle R \rangle slg-rel \rightarrow \langle R \rangle slg-rel$
apply $(subst\ graph-minus-aimpl[abs-def])$
apply $(autoref\ (keep-goal, trace))$

done

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

lemma graph-minus-set-aimpl:

fixes $E1\ E2 :: 'a\ rel$

shows $E1 - E2 = E\text{-of-succ}\ (\lambda u. \{v \in E1 \mid \{u\}. (u, v) \notin E2\})$

by (auto simp: E-of-succ-def)

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

fixes $R :: ('v \times 'v)\ set$

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

assumes [autoref-rules]: $(mem, (\in)) \in R \times_r R \rightarrow \langle R \times_r R \rangle Rs \rightarrow bool\text{-rel}$

shows $(?c, (-)) \in \langle R \rangle slg\text{-rel} \rightarrow \langle R \times_r R \rangle Rs \rightarrow \langle R \rangle slg\text{-rel}$

apply (subst graph-minus-set-aimpl[abs-def])

apply (autoref (keep-goal, trace))

done

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 .

lemma g-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

$::_i \langle I \rangle_i i\text{-set} \rightarrow_i \langle I \rangle_i i\text{-slg} \rightarrow_i \langle I \rangle_i i\text{-set} \rightarrow_i iE \rightarrow_i \langle Ie, I \rangle_i i\text{-g-ext}$

by simp-all

end

7.2.2 Generic Implementation

record ($'vi, 'ei, 'v0i$) gen-g-impl =

$gi\text{-}V :: 'vi$

$gi\text{-}E :: 'ei$

$gi\text{-}V0 :: 'v0i$

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

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

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

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

unfolding *gen-g-impl-rel-ext-internal-def relAPP-def* **by** *simp*

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

$\bigwedge Rm Rv Re Rv0. \llbracket \text{single-valued } Rv; \text{single-valued } Re; \text{single-valued } Rv0; \text{single-valued } Rm \rrbracket \implies$

single-valued $(\langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext})$

unfolding *gen-g-impl-rel-ext-def*

apply *auto*

intro!: *single-valuedI*

dest: *single-valuedD slg-rel-sv list-set-rel-sv*

done

lemma *gen-g-refine*:

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

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

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

$\bigwedge Rm Rv Re Rv0. (\text{gen-g-impl-ext}, \text{graph-rec-ext})$

$\in Rv \rightarrow Re \rightarrow Rv0 \rightarrow Rm \rightarrow \langle Rm, Rv, Re, Rv0 \rangle \text{gen-g-impl-rel-ext}$

unfolding *gen-g-impl-rel-ext-def*

by *auto*

7.2.3 Implementation with list-set for Nodes

type-synonym $(v, m) \text{frgv-impl-scheme} =$

$(v \text{ list}, v \Rightarrow v \text{ list}, v \text{ list}, m) \text{gen-g-impl-scheme}$

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

frgv-impl-rel-ext $Rm Rv$

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

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

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

unfolding *frgv-impl-rel-ext-internal-def relAPP-def* **by** *simp*

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

by *(rule REL-INTFI)*

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

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 \implies bool, 'v \implies 'v list, 'v list, 'm) gen-g-impl-scheme

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

lemma [autoref-rel-intf]: REL-INTF g-impl-rel-ext i-g-ext
by (rule REL-INTFI)

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

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-V-update, g-V-update) \in ($\langle Rv \rangle \text{fun-set-rel} \rightarrow \langle Rv \rangle \text{fun-set-rel}$)
 \rightarrow

$\langle Rm, Rv \rangle \text{g-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle \text{g-impl-rel-ext}$
unfolding g-impl-rel-ext-def gen-g-impl-rel-ext-def
by (auto, metis (full-types) tagged-fun-relD-both)

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

$\langle Rm, Rv \rangle \text{g-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle \text{g-impl-rel-ext}$
unfolding g-impl-rel-ext-def gen-g-impl-rel-ext-def
by (auto, metis (full-types) tagged-fun-relD-both)

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

$\langle Rm, Rv \rangle \text{g-impl-rel-ext} \rightarrow \langle Rm, Rv \rangle \text{g-impl-rel-ext}$

unfolding *g-impl-rel-ext-def gen-g-impl-rel-ext-def*
by (*auto, metis (full-types) tagged-fun-relD-both*)

lemma [*autoref-hom*]:

CONSTRAINT graph-rec-ext ($\langle Rv \rangle Rvs \rightarrow \langle Rv \rangle Res \rightarrow \langle Rv \rangle Rv0s \rightarrow Rm \rightarrow \langle Rm, Rv \rangle Rg$)
by *simp*

schematic-goal ($?c::?'c, \lambda G x. g-E G \{x\} \in ?R$)
apply (*autoref (keep-goal)*)
done

schematic-goal ($?c, \lambda V0 E.$
 $(\mid g-V = UNIV, g-E = E, g-V0 = V0 \mid)$
 $\in \langle R \rangle list-set-rel \rightarrow \langle R \rangle slg-rel \rightarrow \langle unit-rel, R \rangle g-impl-rel-ext$
apply (*autoref (keep-goal)*)
done

schematic-goal ($?c, \lambda V V0 E.$
 $(\mid g-V = V, g-E = E, g-V0 = V0 \mid)$
 $\in \langle R \rangle list-set-rel \rightarrow \langle R \rangle list-set-rel \rightarrow \langle R \rangle slg-rel \rightarrow \langle unit-rel, R \rangle frgv-impl-rel-ext$
apply (*autoref (keep-goal)*)
done

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*]:
 $the-inv-into-map\ V\ f\ x = None \iff x \notin f'V$
unfolding *the-inv-into-map-def* **by** *auto*

lemma *the-inv-into-map-Some'*:
 $the-inv-into-map\ V\ f\ x = Some\ y \iff x \in f'V \wedge y = the-inv-into\ V\ f\ x$
unfolding *the-inv-into-map-def* **by** *auto*

lemma *the-inv-into-map-Some*[*simp*]:
 $inj-on\ f\ V \implies the-inv-into-map\ V\ f\ x = Some\ y \iff y \in V \wedge x = f\ y$
by (*auto simp: the-inv-into-map-Some' the-inv-into-f-f*)

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 $the-inv-into-map-impl\ V\ f \leq SPEC\ (\lambda r.\ r = the-inv-into-map\ V\ f)$

unfolding *the-inv-into-map-impl-def*
apply (*refine-rcg*
 FOREACH-rule[**where** $I = \lambda it m. m = \text{the-inv-into-map } (V - it) f$]
refine-vcg
)

apply (*vc-solve*
simp: the-inv-into-map-def[abs-def] it-step-insert-iff
intro!: ext)

apply (*intro allI impI conjI*)

apply (*subst the-inv-into-f-f[OF subset-inj-on[OF INJ]], auto*) []

apply (*subst the-inv-into-f-f[OF subset-inj-on[OF INJ]], auto*) []

apply *safe* []
apply (*subst the-inv-into-f-f[OF subset-inj-on[OF INJ]], (auto) [2]*)+
apply *simp*
done

schematic-goal *the-inv-into-map-code-aux*:
fixes $Rv' :: ('vti \times 'vt) \text{ set}$
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $Rv' \text{ eq } bhc$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('vti) \text{ (def-size)}$
assumes [*autoref-rules*]: $(Vi, V) \in \langle Rv \rangle \text{ list-set-rel}$
assumes [*autoref-rules*]: $(fi, f) \in Rv \rightarrow Rv'$
shows (*RETURN ?c, the-inv-into-map-impl V f*) $\in \langle \langle Rv', Rv \rangle \text{ ahm-rel } bhc \rangle \text{ nres-rel}$
unfolding *the-inv-into-map-impl-def[abs-def]*
apply (*autoref-monadic (plain)*)
done

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* .
lemma *autoref-the-inv-into-map[autoref-rules]*:
fixes $Rv' :: ('vti \times 'vt) \text{ set}$
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* $Rv' \text{ eq } bhc$)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE('vti) \text{ def-size}$)
assumes *INJ: SIDE-PRECOND* (*inj-on* $f V$)
assumes $V: (Vi, V) \in \langle Rv \rangle \text{ list-set-rel}$
assumes $F: (fi, f) \in Rv \rightarrow Rv'$
shows (*the-inv-into-map-code eq bhc def-size Vi fi,*
(OP the-inv-into-map
 $\dots \langle Rv \rangle \text{ list-set-rel} \rightarrow (Rv \rightarrow Rv') \rightarrow \langle Rv', Rv \rangle \text{ Impl-Array-Hash-Map.ahm-rel}$
bhc)

```

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

from V have FIN: finite V using list-set-rel-range by auto

note the-inv-into-map-code.refine[
  OF assms(1-2,4-5)[unfolded autoref-tag-defs], THEN nres-relD
]
also note the-inv-into-map-impl-correct[OF FIN INJ[unfolded autoref-tag-defs]
finally show (the-inv-into-map-code eq bhc def-size Vi fi, the-inv-into-map V f)
  ∈ ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel bhc
  by (simp add: refine-pw-simps pw-le-iff)
qed

end

schematic-goal (?c::?'c, do {
  let s = {1,2,3::nat};
  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 = (if x ∈ f'V then Some (fi x) else None)
lemma fi-map-alt: fi-map = the-inv-into-map V f
apply (rule ext)
unfolding fi-map-def the-inv-into-map-def fi-def
by simp

lemma fi-map-Some: (fi-map u = Some v)  $\longleftrightarrow$  u ∈ f'V ∧ fi u = v
unfolding fi-map-def by (auto split: if-split-asm)

```

lemma *fi-map-None*: $(fi\text{-map } u = None) \longleftrightarrow u \notin f \cdot V$
unfolding *fi-map-def* **by** (*auto split: if-split-asm*)

lemma *rename-E-aimpl-alt*: $rename\text{-E } f E = E\text{-of-succ } (\lambda v. \text{case } fi\text{-map } v \text{ of } Some\ u \Rightarrow f \cdot (succ\text{-of-E } E\ u) \mid None \Rightarrow \{\})$
unfolding *E-of-succ-def succ-of-E-def*
using *E-ss*
by (*force*
simp: fi-f fi-fi fi-map-Some fi-map-None
split: if-split-asm option.splits)

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 –

show *?thesis*
unfolding *fr-rename-ext-def fr-rename-ext-aimpl-def*
apply (*refine-rcg*
order-trans[OF ECNV[unfolded fi-map-alt]]
refine-vcg)
using *subset-inj-on[OF - V0-ss]*
apply (*auto intro: INJ simp: rename-E-aimpl-alt fi-map-alt*)
done

qed
end

term *frv-rename-ext-aimpl*
schematic-goal *fr-rename-ext-impl-aux*:
fixes *Re* and *Rv'* :: $('vti \times 'vt)$ *set*
assumes [*autoref-rules*]: $(eq, (=)) \in Rv' \rightarrow Rv' \rightarrow bool\text{-rel}$
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* Rv' *eq* *bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE('vti)$ *def-size*
shows $(?c.fr\text{-rename-ext-aimpl}) \in$
 $(\langle \langle Rv', Rv \rangle ahm\text{-rel } bhc \rangle \rightarrow \langle Re, Rv \rangle frgv\text{-impl-rel-ext} \rightarrow \langle Re' \rangle nres\text{-rel}) \rightarrow$
 $(Rv \rightarrow Rv') \rightarrow$
 $\langle Re, Rv \rangle frgv\text{-impl-rel-ext} \rightarrow$
 $\langle \langle Re', Rv' \rangle frgv\text{-impl-rel-ext} \rangle nres\text{-rel}$
unfolding *fr-rename-ext-impl-def[abs-def]*
apply (*autoref (keep-goal)*)
done

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×nat) list ⇒ nat ⇒ nat set

where

```

succ-of-list l ≡ let
  m = fold (λ(u,v) g.
    case g u of
      None ⇒ g(u→{v})
    | Some s ⇒ g(u→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 –

term *the-default*

```

{ fix m
  have fold (λ(u,v) g.
    case g u of
      None ⇒ g(u→{v})
    | Some s ⇒ g(u→insert v s)
  ) l m
  = (λu. let s=set l “ {u} in
    if s={} then m u else Some (the-default {} (m u) ∪ s))
  apply (induction l arbitrary: m)
  apply (auto
    split: option.split if-split
    simp: Let-def Image-def
    intro!: ext)
  done
} note aux=this

```

show *?thesis*

unfolding *succ-of-list-def aux*

by (auto simp: br-def Let-def split: option.splits if-split-asm)

qed

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

unfolding *succ-of-list-def[abs-def]*

apply (autoref (keep-goal))

done

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*: $(succ-of-list-impl, set) \in Id \rightarrow \langle Id \rangle slg-rel$
apply *rule*
unfolding *slg-rel-def*
apply *rule*
apply (*rule succ-of-list-impl.refine*[*THEN fun-relD*])
apply *simp*
apply (*rule succ-of-list-correct-aux*)
done

end

8 Implementing Automata

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

8.1 Indexed Generalized Buchi Graphs

consts

i-igbg-eext :: *interface* \Rightarrow *interface* \Rightarrow *interface*

abbreviation *i-igbg Ie Iv* $\equiv \langle \langle Ie, Iv \rangle_i i-igbg-eext, Iv \rangle_i i-g-ext$

context begin interpretation *autoref-syn* .

lemma *igbg-type*[*autoref-itype*]:

igbg-num-acc ::_{*i*} *i-igbg Ie Iv* \rightarrow_i *i-nat*

igbg-acc ::_{*i*} *i-igbg Ie Iv* \rightarrow_i *Iv* \rightarrow_i $\langle i-nat \rangle_i i-set$

igbg-graph-rec-ext

::_{*i*} *i-nat* \rightarrow_i (*Iv* \rightarrow_i $\langle i-nat \rangle_i i-set$) \rightarrow_i *Ie* \rightarrow_i $\langle Ie, Iv \rangle_i i-igbg-eext$

by *simp-all*

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 $\equiv \{$ (
 \langle *igbgi-num-acc* = *num-acci*, *igbgi-acc* = *acci*, ... = *mi* \rangle ,
 \langle *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc*, ... = *m* \rangle)
 $|$ *num-acci acci mi num-acc acc m.
 $(num-acci, num-acc) \in nat-rel$*

$\wedge (acci, acc) \in Racc$
 $\wedge (mi, m) \in Rm$
 $\}$

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

$\langle Rm, Racc \rangle \text{gen-igbg-impl-rel-eext} = \{ ($
 $\langle \text{igbgi-num-acc} = \text{num-acci}, \text{igbgi-acc} = \text{acci}, \dots = mi \rangle,$
 $\langle \text{igbg-num-acc} = \text{num-acc}, \text{igbg-acc} = \text{acc}, \dots = m \rangle)$
 $| \text{num-acci } acci \text{ } mi \text{ } \text{num-acc } acc \text{ } m.$
 $(\text{num-acci}, \text{num-acc}) \in \text{nat-rel}$
 $\wedge (acci, acc) \in Racc$
 $\wedge (mi, m) \in Rm$
 $\}$

unfolding *gen-igbg-impl-rel-eext-def-internal relAPP-def* by *simp*

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

$\llbracket \text{single-valued } Racc; \text{single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Racc \rangle \text{gen-igbg-impl-rel-eext})$
unfolding *gen-igbg-impl-rel-eext-def*
apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD[rotated], assumption+*)
apply (*rule single-valuedD[rotated], assumption+*)
done

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

$:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times (-, -) \text{igbg-graph-rec-scheme}) \text{ set}$
where *gen-igbg-impl-rel-ext* $Rm \ Racc$
 $\equiv \langle \langle Rm, Racc \rangle \text{gen-igbg-impl-rel-eext} \rangle \text{gen-g-impl-rel-ext}$

lemma *gen-igbg-refine*:

fixes $Rv \ Re \ Rv0 \ Racc$
assumes $TERM \ (Rv, Re, Rv0)$
assumes $TERM \ (Racc)$
shows
 $(\text{igbgi-num-acc}, \text{igbg-num-acc})$
 $\in \langle Rv, Re, Rv0 \rangle \text{gen-igbg-impl-rel-ext } Rm \ Racc \rightarrow \text{nat-rel}$
 $(\text{igbgi-acc}, \text{igbg-acc})$
 $\in \langle Rv, Re, Rv0 \rangle \text{gen-igbg-impl-rel-ext } Rm \ Racc \rightarrow Racc$
 $(\text{gen-igbg-impl-ext}, \text{igbg-graph-rec-ext})$
 $\in \text{nat-rel} \rightarrow Racc \rightarrow Rm \rightarrow \langle Rm, Racc \rangle \text{gen-igbg-impl-rel-eext}$
unfolding *gen-igbg-impl-rel-eext-def gen-g-impl-rel-ext-def*
by *auto*

8.1.1 Implementation with bit-set

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

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

lemma *igbg-impl-rel-eext-def*:
 $\langle Rm, Rv \rangle \text{igbg-impl-rel-eext} \equiv \langle Rm, Rv \rightarrow \langle \text{nat-rel} \rangle \text{bs-set-rel} \rangle \text{gen-igbg-impl-rel-eext}$
unfolding *igbg-impl-rel-eext-internal-def relAPP-def* **by** *simp*

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

lemma [*relator-props, simp*]:
 $\llbracket \text{Range } Rv = \text{UNIV}; \text{single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv \rangle \text{igbg-impl-rel-eext})$
unfolding *igbg-impl-rel-eext-def* **by** *tagged-solver*

lemma *g-tag*: *TERM* ($\langle Rv \rangle \text{fun-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel}$) .
lemma *frgv-tag*: *TERM* ($\langle Rv \rangle \text{list-set-rel}, \langle Rv \rangle \text{slg-rel}, \langle Rv \rangle \text{list-set-rel}$) .
lemma *igbg-bs-tag*: *TERM* ($Rv \rightarrow \langle \text{nat-rel} \rangle \text{bs-set-rel}$) .

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

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

type-synonym (*'v, 'm*) *igbgv-impl-scheme* =
(*'v*, ($\mid \text{igbgi-num-acc}::\text{nat}, \text{igbgi-acc}::'v \Rightarrow \text{integer}, \dots::'m$ \mid))
frgv-impl-scheme
type-synonym (*'v, 'm*) *igbg-impl-scheme* =
(*'v*, ($\mid \text{igbgi-num-acc}::\text{nat}, \text{igbgi-acc}::'v \Rightarrow \text{integer}, \dots::'m$ \mid))
g-impl-scheme

context fixes *Rv* :: (*'vi* \times *'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*,
 $\lambda G x. \text{if } \text{igbg-num-acc } G = 0 \wedge 1 \in \text{igbg-acc } G \text{ then } (g\text{-E } G \text{ “ } \{x\} \text{) else } \{\}$
 $\in ?R$
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*,

$\lambda V0 E \text{ num-acc } \text{acc}.$
 $(\mid g-V = UNIV, g-E = E, g-V0 = V0, \text{igbg-num-acc} = \text{num-acc}, \text{igbg-acc} = \text{acc} \mid)$
 $\in \langle R \rangle \text{list-set-rel} \rightarrow \langle R \rangle \text{slg-rel} \rightarrow \text{nat-rel} \rightarrow (R \rightarrow \langle \text{nat-rel} \rangle \text{bs-set-rel})$
 $\rightarrow \text{igbg-impl-rel-ext unit-rel } R$
apply (*autoref* (*keep-goal*))
done

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

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* .

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

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

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

by *simp-all*

end

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

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

$igbai-L :: 'Li$

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

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

$(\mid \text{igbai-L} = Li, \dots = mi \mid),$

$(\mid \text{igba-L} = L, \dots = m \mid)$

$\mid Li mi L m.$

$(Li, L) \in Rl$

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

$\}$

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

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

$(\mid \text{igbai-L} = Li, \dots = mi \mid),$

$(\mid \text{igba-L} = L, \dots = m \mid)$

$| Li\ mi\ L\ m.$
 $(Li, L) \in Rl$
 $\wedge (mi, m) \in Rm$
 $\}$
unfolding *gen-igba-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-igba-impl-rel-sv[relator-props]*:
 $\llbracket \text{single-valued } Rl; \text{ single-valued } Rm \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rl \rangle \text{gen-igba-impl-rel-eext})$
unfolding *gen-igba-impl-rel-eext-def*
apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD[rotated], assumption+*)
apply (*rule single-valuedD[rotated], assumption+*)
done

abbreviation *gen-igba-impl-rel-ext*
 $:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('a, 'b, 'c) \text{igba-rec-scheme}) \text{ set}$
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
 $(\text{igbai-L}, \text{igba-L})$
 $\in \langle Rv, Re, Rv0 \rangle \text{gen-igba-impl-rel-ext } Rm\ Rl\ Racc \rightarrow Rl$
 $(\text{gen-igba-impl-ext}, \text{igba-rec-ext})$
 $\in Rl \rightarrow Rm \rightarrow \langle Rm, Rl \rangle \text{gen-igba-impl-rel-eext}$
unfolding *gen-igba-impl-rel-eext-def gen-igbg-impl-rel-eext-def*
 $\text{gen-g-impl-rel-ext-def}$
by *auto*

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

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

lemma [*relator-props, simp*]:
 $\llbracket \text{Range } Rv = \text{UNIV}; \text{ single-valued } Rm; \text{ Range } Rl = \text{UNIV} \rrbracket$

\Rightarrow *single-valued* ($\langle Rm, Rv, Rl \rangle$ *igba-impl-rel-eext*)
unfolding *igba-impl-rel-eext-def* **by** *tagged-solver*

lemma *igba-f-tag*: *TERM* ($Rv \rightarrow Rl \rightarrow$ *bool-rel*) .

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

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

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

type-synonym ($'v, 'l, 'm$) *igba-impl-scheme* =
 $('v, (\mid$ *igbai-L* $:: 'v \Rightarrow 'l \Rightarrow$ *bool* $, \dots :: '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-eext-def igba-impl-rel-eext-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-eext-def igba-impl-rel-eext-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* $\{\}$ $) \in ?R$

apply (*autoref* (*keep-goal*))

done

schematic-goal

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

shows $(?c::?'c, \lambda E\ (V0::'a)$ *set*) *num-acc* $acc\ L.$

$(\mid$ $g-V = UNIV, g-E = E, g-V0 = V0,$

$igbg-num-acc = num-acc, igbg-acc = acc, igba-L = L$ $)$

$) \in ?R$

apply (*autoref* (*keep-goal*))

done

schematic-goal

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

8.3 Generalized Buchi Graphs

consts

$i-gbg-eext :: interface \Rightarrow interface \Rightarrow interface$

abbreviation $i-gbg Ie Iv \equiv \langle \langle Ie, Iv \rangle_i i-gbg-eext, Iv \rangle_i i-g-ext$

context begin interpretation $autoref-syn$.

lemma $gbg-type[autoref-itype]:$

$gbg-F ::_i i-gbg Ie Iv \rightarrow_i \langle \langle Iv \rangle_i i-set \rangle_i i-set$
 $gb-graph-rec-ext ::_i \langle \langle Iv \rangle_i i-set \rangle_i i-set \rightarrow_i Ie \rightarrow_i \langle Ie, Iv \rangle_i i-gbg-eext$
by $simp-all$

end

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

definition $gen-gbg-impl-rel-eext-def-internal:$

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

lemma $gen-gbg-impl-rel-eext-def:$

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

unfolding $gen-gbg-impl-rel-eext-def-internal relAPP-def$ **by** $simp$

lemma $gen-gbg-impl-rel-sv[relator-props]:$

$\llbracket single-valued Rm; single-valued Rf \rrbracket$
 $\implies single-valued (\langle Rm, Rf \rangle gen-gbg-impl-rel-eext)$
unfolding $gen-gbg-impl-rel-eext-def$

apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD[rotated]*, *assumption+*)
apply (*rule single-valuedD[rotated]*, *assumption+*)
done

abbreviation *gen-gbg-impl-rel-ext*
 $:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('q,-) \text{ gb-graph-rec-scheme}) \text{ set}$
where *gen-gbg-impl-rel-ext Rm Rf*
 $\equiv \langle\langle Rm, Rf \rangle \text{ gen-gbg-impl-rel-ext} \rangle \text{ gen-g-impl-rel-ext}$

lemma *gen-gbg-refine*:
fixes *Rv Re Rv0 Rf*
assumes *TERM (Rv, Re, Rv0)*
assumes *TERM (Rf)*
shows
(*gbgi-F, gbg-F*)
 $\in \langle Rv, Re, Rv0 \rangle \text{ gen-gbg-impl-rel-ext } Rm \ Rf \rightarrow Rf$
(*gen-gbg-impl-ext, gb-graph-rec-ext*)
 $\in Rf \rightarrow Rm \rightarrow \langle Rm, Rf \rangle \text{ gen-gbg-impl-rel-ext}$
unfolding *gen-gbg-impl-rel-ext-def gen-g-impl-rel-ext-def*
by *auto*

8.3.1 Implementation with list of lists

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

lemma *gbg-impl-rel-ext-def*:
 $\langle Rm, Rv \rangle \text{ gbg-impl-rel-ext}$
 $\equiv \langle Rm, \langle \langle Rv \rangle \text{ list-set-rel} \rangle \text{ list-set-rel} \rangle \text{ gen-gbg-impl-rel-ext}$
unfolding *gbg-impl-rel-ext-internal-def relAPP-def* **by** *simp*

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

lemma [*relator-props, simp*]:
 $\llbracket \text{single-valued } Rm; \text{ single-valued } Rv \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rv \rangle \text{ gbg-impl-rel-ext})$
unfolding *gbg-impl-rel-ext-def* **by** *tagged-solver*

lemma *gbg-ls-tag*: *TERM* ($\langle \langle Rv \rangle \text{ list-set-rel} \rangle \text{ list-set-rel}$) .

abbreviation *gbgv-impl-rel-ext Rm Rv*
 $\equiv \langle \langle Rm, Rv \rangle \text{ gbg-impl-rel-ext}, Rv \rangle \text{ frgv-impl-rel-ext}$

abbreviation *gbg-impl-rel-ext Rm Rv*
 $\equiv \langle \langle Rm, Rv \rangle \text{ gbg-impl-rel-ext}, Rv \rangle \text{ g-impl-rel-ext}$

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-eext-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-eext-def]
end

schematic-goal ($?c::?'c,$
 $\lambda G x.$ *if gbg-F G = {} then (g-E G “ {x}) else {}*
 $) \in ?R$
apply (*autoref (keep-goal)*)
done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI[of Id :: ('a \times 'a) set]*
shows ($?c::?'c, \lambda E (V0::'a \text{ set}) F.$
 $(\mid g-V = \{\}, g-E = E, g-V0 = V0, gbg-F = F \mid) \in ?R$
apply (*autoref (keep-goal)*)
done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI[of Id :: ('a \times 'a) set]*
shows ($?c::?'c, \lambda E (V0::'a \text{ set}) F.$
 $(\mid g-V = UNIV, g-E = E, g-V0 = V0, gbg-F = insert \{\} F \mid) \in ?R$
apply (*autoref (keep-goal)*)
done

schematic-goal ($?c::?'c, it-to-sorted-list (\lambda -. True) \{1,2::nat\} \in ?R$
apply (*autoref (keep-goal)*)
done

8.4 GBAs

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

abbreviation $i\text{-gba } Ie Iv Il$
 $\equiv \langle \langle \langle Ie, Iv, Il \rangle_i i\text{-gba-eext}, Iv \rangle_i i\text{-gbg-eext}, Iv \rangle_i i\text{-g-ext}$
context begin interpretation *autoref-syn* .

lemma $gba\text{-type}[autoref\text{-itype}]$:
 $gba-L ::_i i\text{-gba } Ie Iv Il \rightarrow_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool})$
 $gba\text{-rec-ext} ::_i (Iv \rightarrow_i Il \rightarrow_i i\text{-bool}) \rightarrow_i Ie \rightarrow_i \langle Ie, Iv, Il \rangle_i i\text{-gba-eext}$
by *simp-all*
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-eeext-def-internal*:

gen-gba-impl-rel-eeext *Rm* *Rl* \equiv { (
 (\Downarrow *gbai-L* = *Li*, ...=*mi* \Downarrow),
 (\Downarrow *gba-L* = *L*, ...=*m* \Downarrow))
 | *Li* *mi* *L* *m*.
 (*Li,L*) \in *Rl*
 \wedge (*mi,m*) \in *Rm*
 }

lemma *gen-gba-impl-rel-eeext-def*:

\langle *Rm,Rl* \rangle *gen-gba-impl-rel-eeext* = { (
 (\Downarrow *gbai-L* = *Li*, ...=*mi* \Downarrow),
 (\Downarrow *gba-L* = *L*, ...=*m* \Downarrow))
 | *Li* *mi* *L* *m*.
 (*Li,L*) \in *Rl*
 \wedge (*mi,m*) \in *Rm*
 }

unfolding *gen-gba-impl-rel-eeext-def-internal* *relAPP-def* **by** *simp*

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

\llbracket *single-valued* *Rl*; *single-valued* *Rm* \rrbracket
 \implies *single-valued* (\langle *Rm,Rl* \rangle *gen-gba-impl-rel-eeext*)
unfolding *gen-gba-impl-rel-eeext-def*
apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD[rotated]*, *assumption+*)
apply (*rule single-valuedD[rotated]*, *assumption+*)
done

abbreviation *gen-gba-impl-rel-ext*

:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times (*'a,'b,'c*) *gba-rec-scheme*) *set*
where *gen-gba-impl-rel-ext* *Rm* *Rl*
 \equiv *gen-gbg-impl-rel-ext* (\langle *Rm,Rl* \rangle *gen-gba-impl-rel-eeext*)

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*)

$\in Rl \rightarrow Rm \rightarrow \langle Rm, Rl \rangle \text{gen-gba-impl-rel-eext}$
unfolding $\text{gen-gba-impl-rel-eext-def}$ $\text{gen-gbg-impl-rel-eext-def}$
 $\text{gen-g-impl-rel-eext-def}$
by *auto*

8.4.1 Implementation as function

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

lemma $\text{gba-impl-rel-eext-def}$:
 $\langle Rm, Rv, Rl \rangle \text{gba-impl-rel-eext} \equiv \langle Rm, Rv \rightarrow Rl \rightarrow \text{bool-rel} \rangle \text{gen-gba-impl-rel-eext}$
unfolding $\text{gba-impl-rel-eext-internal-def}$ relAPP-def **by** *simp*

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

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

lemma gba-f-tag : $\text{TERM } (Rv \rightarrow Rl \rightarrow \text{bool-rel})$.

abbreviation $\text{gbav-impl-rel-ext } Rm \ Rv \ Rl$
 $\equiv \text{gbg-impl-rel-ext } (\langle Rm, Rv, Rl \rangle \text{gba-impl-rel-eext}) \ Rv$

abbreviation $\text{gba-impl-rel-ext } Rm \ Rv \ Rl$
 $\equiv \text{gbg-impl-rel-ext } (\langle Rm, Rv, Rl \rangle \text{gba-impl-rel-eext}) \ Rv$

context

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

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

begin

lemmas $[\text{autoref-rules}] = \text{gen-gba-refine}$
 $[\text{OF } \text{frgv-tag}[\text{of } Rv] \ \text{gbg-ls-tag}[\text{of } Rv] \ \text{gba-f-tag}[\text{of } Rv \ Rl],$
 $\text{folded } \text{frgv-impl-rel-ext-def} \ \text{gbg-impl-rel-eext-def} \ \text{gba-impl-rel-eext-def}]$

lemmas $[\text{autoref-rules}] = \text{gen-gba-refine}$
 $[\text{OF } \text{g-tag}[\text{of } Rv] \ \text{gbg-ls-tag}[\text{of } Rv] \ \text{gba-f-tag}[\text{of } Rv \ Rl],$
 $\text{folded } \text{g-impl-rel-ext-def} \ \text{gbg-impl-rel-eext-def} \ \text{gba-impl-rel-eext-def}]$

end

thm autoref-itype

schematic-goal

$(?c :: ?'c, \lambda G \ x \ l. \text{if } \text{gba-L } G \ x \ l \text{ then } (\text{g-E } G \ \{x\}) \ \text{else } \{\}) \in ?R$

apply $(\text{autoref } (\text{keep-goal}))$

done

schematic-goal

notes $[autoref-tyrel] = TYRELI[of\ Id :: ('a \times 'a)\ set]$
shows $(?c :: ?'c, \lambda E (V0 :: 'a\ set)\ F\ L.$
 $(\ | g-V = UNIV, g-E = E, g-V0 = V0,$
 $\quad gbg-F = F, gba-L = L \ |)$
 $) \in ?R$
apply $(autoref\ (keep-goal))$
done

schematic-goal

notes $[autoref-tyrel] = TYRELI[of\ Id :: ('a \times 'a)\ set]$
shows $(?c :: ?'c, \lambda E (V0 :: 'a\ set)\ F\ L.$
 $(\ | g-V = V0, g-E = E, g-V0 = V0,$
 $\quad gbg-F = F, gba-L = L \ |)$
 $) \in ?R$
apply $(autoref\ (keep-goal))$
done

8.5 Buchi Graphs

consts

$i\text{-bg-ecxt} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}$

abbreviation $i\text{-bg}\ Ie\ Iv \equiv \langle \langle Ie, Iv \rangle_i i\text{-bg-ecxt}, Iv \rangle_i i\text{-g-ecxt}$

context begin interpretation $autoref\text{-syn}$.

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

$bg-F ::_i i\text{-bg}\ Ie\ Iv \rightarrow_i \langle Iv \rangle_i i\text{-set}$
 $gb\text{-graph-rec-ecxt} ::_i \langle \langle Iv \rangle_i i\text{-set} \rangle_i i\text{-set} \rightarrow_i Ie \rightarrow_i \langle Ie, Iv \rangle_i i\text{-bg-ecxt}$
by $simp\text{-all}$

end

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

definition $gen\text{-bg-impl-rel-ecxt-def-internal}$:

$gen\text{-bg-impl-rel-ecxt}\ Rm\ Rf \equiv \{ (\$
 $\ | bg\text{-}F = Fi, \dots = mi \ |),$
 $\ | bg\text{-}F = F, \dots = m \ |)$
 $\ | Fi\ mi\ F\ m.$
 $\quad (Fi, F) \in Rf$
 $\quad \wedge (mi, m) \in Rm$
 $\}$

lemma $gen\text{-bg-impl-rel-ecxt-def}$:

$\langle Rm, Rf \rangle gen\text{-bg-impl-rel-ecxt} = \{ (\$
 $\ | bg\text{-}F = Fi, \dots = mi \ |),$
 $\ | bg\text{-}F = F, \dots = m \ |)$
 $\ | Fi\ mi\ F\ m.$

$(Fi, F) \in Rf$
 $\wedge (mi, m) \in Rm$
 $\}$
unfolding *gen-bg-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-bg-impl-rel-sv[relator-props]*:
 $\llbracket \text{single-valued } Rm; \text{single-valued } Rf \rrbracket$
 $\implies \text{single-valued } (\langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext})$
unfolding *gen-bg-impl-rel-eext-def*
apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD[rotated], assumption+*)
apply (*rule single-valuedD[rotated], assumption+*)
done

abbreviation *gen-bg-impl-rel-ext*
 $:: - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow - \Rightarrow (- \times ('g, -) \text{b-graph-rec-scheme}) \text{set}$
where *gen-bg-impl-rel-ext Rm Rf*
 $\equiv \langle \langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext} \rangle \text{gen-g-impl-rel-ext}$

lemma *gen-bg-refine*:
fixes *Rv Re Rv0 Rf*
assumes *TERM (Rv, Re, Rv0)*
assumes *TERM (Rf)*
shows
 $(\text{bgi-}F, \text{bg-}F)$
 $\in \langle Rv, Re, Rv0 \rangle \text{gen-bg-impl-rel-ext } Rm \text{ } Rf \rightarrow Rf$
 $(\text{gen-bg-impl-ext}, \text{b-graph-rec-ext})$
 $\in Rf \rightarrow Rm \rightarrow \langle Rm, Rf \rangle \text{gen-bg-impl-rel-eext}$
unfolding *gen-bg-impl-rel-eext-def gen-g-impl-rel-ext-def*
by *auto*

8.5.1 Implementation with Characteristic Functions

definition *bg-impl-rel-eext-internal-def*:
 $\text{bg-impl-rel-eext } Rm \text{ } Rv$
 $\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel} \rangle \text{gen-bg-impl-rel-eext}$

lemma *bg-impl-rel-eext-def*:
 $\langle Rm, Rv \rangle \text{bg-impl-rel-eext}$
 $\equiv \langle Rm, \langle Rv \rangle \text{fun-set-rel} \rangle \text{gen-bg-impl-rel-eext}$
unfolding *bg-impl-rel-eext-internal-def relAPP-def* **by** *simp*

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

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

unfolding *bg-impl-rel-eext-def* **by** *tagged-solver*

lemma *bg-fs-tag*: *TERM* ($\langle Rv \rangle$ *fun-set-rel*) .

abbreviation *bgv-impl-rel-ext* *Rm Rv*
 $\equiv \langle \langle Rm, Rv \rangle$ *bg-impl-rel-eext*, *Rv* \rangle *frgv-impl-rel-ext*

abbreviation *bg-impl-rel-ext* *Rm Rv*
 $\equiv \langle \langle Rm, Rv \rangle$ *bg-impl-rel-eext*, *Rv* \rangle *g-impl-rel-ext*

context fixes *Rv* :: ($'vi \times 'v$) **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 ($?c::?'c$,
 $\lambda G x.$ *if* $x \in$ *bg-F* *G* *then* (*g-E* *G* “ {*x*} *else* {
 }) $\in ?R$
 apply (*autoref* (*keep-goal*))
 done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: ($'a \times 'a$) *set*]
shows ($?c::?'c$, λE ($V0::'a$ *set*) *F*.
 (\langle *g-V* = { $\}$, *g-E* = *E*, *g-V0* = *V0*, *bg-F* = *F* \rangle) $\in ?R$
 apply (*autoref* (*keep-goal*))
 done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: ($'a \times 'a$) *set*]
shows ($?c::?'c$, λE ($V0::'a$ *set*) *F*.
 (\langle *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*, *bg-F* = *F* \rangle) $\in ?R$
 apply (*autoref* (*keep-goal*))
 done

8.6 System Automata

consts
 i-sa-eext :: *interface* \Rightarrow *interface* \Rightarrow *interface* \Rightarrow *interface*

abbreviation *i-sa* *Ie Iv Il* $\equiv \langle \langle Ie, Iv, Il \rangle_i$ *i-sa-eext*, *Iv* \rangle_i *i-g-ext*

context begin interpretation *autoref-syn* .
term *sa-L*

lemma *sa-type*[*autoref-itype*]:
sa-L ::_i *i-sa* *Ie Iv Il* →_i *Iv* →_i *Il*
sa-rec-ext ::_i (*Iv* →_i *Il*) →_i *Ie* →_i ⟨*Ie,Iv,Il*⟩_i *i-sa-ext*
by *simp-all*
end

record (*'vi,'ei,'v0i,'li*) *gen-sa-impl* = (*'vi,'ei,'v0i*) *gen-g-impl* +
sai-L :: *'li*

definition *gen-sa-impl-rel-ext-def-internal*:
gen-sa-impl-rel-ext *Rm Rl* ≡ { (
 (| *sai-L* = *Li*, ... = *mi* |),
 (| *sa-L* = *L*, ... = *m* |))
 | *Li mi L m*.
 (*Li,L*) ∈ *Rl*
 ∧ (*mi,m*) ∈ *Rm*
 }

lemma *gen-sa-impl-rel-ext-def*:
 ⟨*Rm,Rl*⟩ *gen-sa-impl-rel-ext* = { (
 (| *sai-L* = *Li*, ... = *mi* |),
 (| *sa-L* = *L*, ... = *m* |))
 | *Li mi L m*.
 (*Li,L*) ∈ *Rl*
 ∧ (*mi,m*) ∈ *Rm*
 }

unfolding *gen-sa-impl-rel-ext-def-internal* *relAPP-def* **by** *simp*

lemma *gen-sa-impl-rel-sv*[*relator-props*]:
 [single-valued *Rm*; single-valued *Rf*]
 ⇒ *single-valued* (⟨*Rm,Rf*⟩ *gen-sa-impl-rel-ext*)
unfolding *gen-sa-impl-rel-ext-def*
apply (*rule single-valuedI*)
apply (*clarsimp*)
apply (*intro conjI*)
apply (*rule single-valuedD*[*rotated*], *assumption+*)
apply (*rule single-valuedD*[*rotated*], *assumption+*)
done

abbreviation *gen-sa-impl-rel-ext*
 :: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × (*'q,'l,-*) *sa-rec-scheme*) *set*
where *gen-sa-impl-rel-ext* *Rm Rf*
 ≡ ⟨⟨*Rm,Rf*⟩ *gen-sa-impl-rel-ext*⟩ *gen-g-impl-rel-ext*

lemma *gen-sa-refine*:
fixes *Rv Re Rv0*
assumes *TERM* (*Rv,Re,Rv0*)
assumes *TERM* (*Rl*)
shows

$(sai-L, sa-L)$
 $\in \langle Rv, Re, Rv0 \rangle gen-sa-impl-rel-ext \ Rm \ Rl \rightarrow Rl$
 $(gen-sa-impl-ext, sa-rec-ext)$
 $\in Rl \rightarrow Rm \rightarrow \langle Rm, Rl \rangle gen-sa-impl-rel-ext$
unfolding $gen-sa-impl-rel-ext-def \ gen-g-impl-rel-ext-def$
by $auto$

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 $sa-impl-rel-ext-def$:

$\langle Rm, Rv, Rl \rangle sa-impl-rel-ext$
 $\equiv \langle Rm, Rv \rightarrow Rl \rangle gen-sa-impl-rel-ext$
unfolding $sa-impl-rel-ext-internal-def \ relAPP-def$ **by** $simp$

lemmas $[autoref-rel-intf] = REL-INTFI[of \ sa-impl-rel-ext \ i-sa-ext]$

lemma $[relator-props, simp]$:

$\llbracket single-valued \ Rm; \ single-valued \ Rl; \ Range \ Rv = \ UNIV \rrbracket$
 $\implies single-valued \ (\langle Rm, Rv, Rl \rangle sa-impl-rel-ext)$
unfolding $sa-impl-rel-ext-def$ **by** $tagged-solver$

lemma $sa-f-tag: \ TERM \ (Rv \rightarrow Rl) \ .$

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 $(v, l, m) \ sav-impl-scheme =$

$(v, (\ | \ sai-L :: v \Rightarrow l, \dots :: m \ |)) \ frgv-impl-scheme$

type-synonym $(v, l, m) \ sa-impl-scheme =$

$(v, (\ | \ sai-L :: v \Rightarrow l, \dots :: m \ |)) \ g-impl-scheme$

context fixes $Rv :: (v \times v) \ set$ **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 {}
)∈?R
apply (autoref (keep-goal))
done

schematic-goal
notes [autoref-tyrel] = TYRELI[of Id :: ('a×'a) set]
shows (?c::?'c, λE (V0::'a set) L.
($g-V = \{\}, g-E = E, g-V0 = V0, sa-L = L$))∈?R
apply (autoref (keep-goal))
done

schematic-goal
notes [autoref-tyrel] = TYRELI[of Id :: ('a×'a) set]
shows (?c::?'c, λE (V0::'a set) L.
($g-V = UNIV, g-E = E, g-V0 = V0, sa-L = L$))∈?R
apply (autoref (keep-goal))
done

8.7 Index Conversion

schematic-goal gbg-to-idx-ext-impl-aux:
fixes Re and Rv :: ('qi × '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 :: - ⇒ ('q, -) gb-graph-rec-scheme ⇒ -)
∈ (gbgv-impl-rel-ext Re Rv → Ri)
→ gbgv-impl-rel-ext Re Rv
→ (igbgv-impl-rel-ext Ri Rv)nres-rel
unfolding gbg-to-idx-ext-def[abs-def] F-to-idx-impl-def mk-acc-impl-def
using [[autoref-trace-failed-id]]
apply (autoref (keep-goal))
done

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 ≤ gbg-to-idx-ext-impl eq bhc def-size ecnv G
unfolding gbg-to-idx-ext-impl-def
by (refine-transfer)

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* .

lemma [*autoref-op-pat*]: *gba-to-idx-ext ecnv* \equiv *OP gba-to-idx-ext \$ ecnv* **by** *simp*
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 (*igbav-impl-rel-ext Ri Rv Rl*) *nres-rel*

using [[*autoref-trace-failed-id*]] **unfolding** *ti-Lcnev-def[abs-def]*

apply (*autoref (keep-goal)*)

done

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*

unfolding *gba-to-idx-ext-impl-def*

by (*refine-transfer*)

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* ($\lambda(q,x). \text{if } x=0 \text{ then } (\lambda q'. (q',0)) \text{'succ-of-E } E \text{ } q \text{ else } \{\}$),

g-V0 = ($\lambda q'. (q',0)$) *'V0*,

bg-F = *Collect* ($\lambda(q,x). x=0 \wedge q \in V$),

\dots = *ecnv G*

)

else (

g-V = *Collect* ($\lambda(q,x). x < \text{num-acc} \wedge q \in V$),

g-E = *E-of-succ* ($\lambda(q,i).$

if $i < \text{num-acc}$ *then*

let

$i' = \text{if } i \in \text{acc } q \text{ then } (i + 1) \text{ mod } \text{num-acc} \text{ else } i$

in ($\lambda q'. (q',i')$) *'succ-of-E } E } q*

else $\{\}$

),


```

    g-V0 = (λq'. (q',0))'V0,
    bg-F = Collect (λ(q,x). x=0 ∧ 0∈acc q),
    ... = ecnv G
  )))
unfolding degeneralize-ext-def
apply (cases num-acc = 0)
apply simp-all
apply (auto simp: E-of-succ-def succ-of-E-def split: if-split-asm) []
apply (fastforce simp: E-of-succ-def succ-of-E-def split: if-split-asm) []
done

schematic-goal degeneralize-ext-impl-aux:
  fixes Re Rv
  assumes [autoref-rules]: (Gi,G) ∈ igbg-impl-rel-ext Re Rv
  shows (?c, degeneralize-ext)
  ∈ (igbg-impl-rel-ext Re Rv → Re') → bg-impl-rel-ext Re' (Rv ×r nat-rel)
  unfolding degen-impl-aux-alt[abs-def]
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done

end

definition [simp]:
  op-igbg-graph-degeneralize-ext ecnv G ≡ igbg-graph.degeneralize-ext G ecnv

lemma [autoref-op-pat]:
  igbg-graph.degeneralize-ext ≡ λG ecnv. op-igbg-graph-degeneralize-ext ecnv G
  by simp

thm igbg-graph.degeneralize-ext-impl-aux[param-fo]
concrete-definition degeneralize-ext-impl
  uses igbg-graph.degeneralize-ext-impl-aux[param-fo]

thm degeneralize-ext-impl.refine

context begin interpretation autoref-syn .
lemma [autoref-rules]:
  fixes Re
  assumes SIDE-PRECOND (igbg-graph G)
  assumes CNVR: (ecnvi,ecnv) ∈ (igbg-impl-rel-ext Re Rv → Re')
  assumes GR: (Gi,G)∈igbg-impl-rel-ext Re Rv
  shows (degeneralize-ext-impl Gi ecnvi,
    (OP op-igbg-graph-degeneralize-ext
     ∷: (igbg-impl-rel-ext Re Rv → Re') → igbg-impl-rel-ext Re Rv
     → bg-impl-rel-ext Re' (Rv ×r nat-rel) )$ecnv$G )
  ∈ bg-impl-rel-ext Re' (Rv ×r nat-rel)
proof –
  from assms have A: igbg-graph G by simp

```

```

show ?thesis
  apply simp
  using degeneralize-ext-impl.refine[OF A GR CNVR]
.
qed

end
thm autoref-itype(1)

```

```

schematic-goal
  assumes [simp]: igb-graph G
  assumes [autoref-rules]: (Gi,G) ∈ igbg-impl-rel-ext unit-rel nat-rel
  shows (?c::?'c, igb-graph.degeneralize-ext G (λ-. ())) ∈ ?R
  apply (autoref (keep-goal))
done

```

8.9 Product Construction

```

context igba-sys-prod-precond begin

```

```

lemma prod-impl-aux-alt:
  prod = (|
    g-V = Collect (λ(q,s). q ∈ igba.V ∧ s ∈ sa.V),
    g-E = E-of-succ (λ(q,s).
      if igba.L q (sa.L s) then
        succ-of-E (igba.E) q × succ-of-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 {}
  |)
  unfolding prod-def
  apply (auto simp: succ-of-E-def E-of-succ-def split: if-split-asm)
done

```

```

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)
  unfolding prod-impl-aux-alt[abs-def]
  apply (autoref (keep-goal))
done

```

```

end

```

definition [*simp*]: $op\text{-}igba\text{-}sys\text{-}prod \equiv igba\text{-}sys\text{-}prod\text{-}precond.prod$

lemma [*autoref-op-pat*]:
 $igba\text{-}sys\text{-}prod\text{-}precond.prod \equiv op\text{-}igba\text{-}sys\text{-}prod$
by *simp*

thm $igba\text{-}sys\text{-}prod\text{-}precond.prod\text{-}impl\text{-}aux[param\text{-}fo]$
concrete-definition $igba\text{-}sys\text{-}prod\text{-}impl$
uses $igba\text{-}sys\text{-}prod\text{-}precond.prod\text{-}impl\text{-}aux[param\text{-}fo]$

thm $igba\text{-}sys\text{-}prod\text{-}impl.refine$

context begin interpretation *autoref-syn* .

lemma [*autoref-rules*]:
fixes Re
assumes $SIDE\text{-}PRECOND (igba\ G)$
assumes $SIDE\text{-}PRECOND (sa\ S)$
assumes $GR: (Gi, G) \in igba\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rq\ Rl$
assumes $SR: (Si, S) \in sa\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rs\ Rl$
shows $(igba\text{-}sys\text{-}prod\text{-}impl\ Gi\ Si,$
 $(OP\ op\text{-}igba\text{-}sys\text{-}prod$
 $::: igba\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rq\ Rl$
 $\rightarrow sa\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rs\ Rl$
 $\rightarrow igbg\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ (Rq \times_r\ Rs)) \$G\$S)$
 $\in igbg\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ (Rq \times_r\ Rs)$

proof –

from *assms* **interpret** $igba: igba\ G + sa: sa\ S$ **by** *simp-all*
have $A: igba\text{-}sys\text{-}prod\text{-}precond\ G\ S$ **by** *unfold-locales*

show *?thesis*
apply *simp*
using $igba\text{-}sys\text{-}prod\text{-}impl.refine[OF\ A\ GR\ SR]$

qed

end

schematic-goal

assumes [*simp*]: $igba\ G\ sa\ S$
assumes [*autoref-rules*]: $(Gi, G) \in igba\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rq\ Rl$
assumes [*autoref-rules*]: $(Si, S) \in sa\text{-}impl\text{-}rel\text{-}ext\ unit\text{-}rel\ Rs\ Rl$
shows $(?c:: ?'c, igba\text{-}sys\text{-}prod\text{-}precond.prod\ G\ S) \in ?R$
apply (*autoref (keep-goal)*)
done

end