The CAVA Automata Library

Peter Lammich

May 26, 2024

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

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*: ⟦ (*u*,*v*)∈*E*; *path E v l w* ⟧ ⟹ *path E u* (*u*#*l*) *w*

lemma *path1*: (*u*,*v*)∈*E* ⟹ *path E u* [*u*] *v*
  by (*auto intro*: *path.intros*)

lemma *path-empty-conv*[*simp*]:
  path E u [] *v* ⟷ *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* ⟷ (*u*=*v* ∧ *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*:
⟦ *path E u l v*; (*v*,*w*)∈*E* ⟧ ⟹ *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*:
path E u (*la*@*lb*) *w* ⟷ (∃ *v*. *path E u la v* ∧ *path E v lb w*)
by (*auto intro*: *path-conc elim*: *path-unconc*)

lemma (**in** −) *path-append-conv*: *path E u* (*p*@[*v*]) *w* ⟷ (*path E u p v* ∧ (*v*,*w*)∈*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*: ⟦(*u*,*v*)∈*E*; (*v*,*w*)∈*E*⟧ ⟹ *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
path E w p x
| *p1 p2* **where** *path E w p1 u*   *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
path E w p x
| *p1 p2* **where** *path* (*insert* (*u*,*v*) *E*) *w p1 u*   *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*: *path E u p v*
shows *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 *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** *path E u l v*
using *assms*
by *induct* (*auto intro*: *path0 path-append*)

lemma *path-is-trancl*:
assumes *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** *path E u l v*
using *assms*
by *induct* (*auto intro*: *path0 path-append*)

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

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* ($E \cap$ *set p* $\times$ *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* ($E \cap$ *set* (*u#p*) $\times$ *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*) ⊆ *fst*‘*E* ∩ *snd*‘*E*
proof −
  from *path-nodes-edges*[*OF assms*] **have** *set* (*tl p*) ⊆ *fst*‘*E*
    by (*cases p*) *auto*

  moreover have *set* (*tl p*) ⊆ *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*′) ∈ *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* ≠ []   *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* ≠ []
  shows (*last p*, *y*) ∈ *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* ∈ *set p*
  shows (*v*,*y*) ∈ 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 ≠ y
  and S: ⋀u. (x,u) ∈ E ⟹ P x u
  and ST: ⋀u v. ⟦(x,u) ∈ E⁺; (u,v) ∈ E; P x u⟧ ⟹ P x v
  shows P x y ∧ (∀ v ∈ set (tl p). P x v)
proof –
  from P NE have p ≠ [] by auto
  thus ?thesis using P
  proof (induction p arbitrary: y rule: rev-nonempty-induct)
    case (single u) hence (x,y) ∈ 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    (x,u) ∈ E⁺    ∀ v ∈ set (tl us). P x v
      by simp-all
    moreover with snoc have ∀ v ∈ set (tl (us@[u])). P x v by simp
    moreover from snoc have (u,y) ∈ E by (simp add: path-append-conv)
    ultimately show ?case by (auto intro: ST)
  qed
qed


lemma path-restrict-tl:
  ⟦ u∉R; path (E ∩ UNIV × −R) u p v ⟧ ⟹ path (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∩UNIV × −R) u (x#xs) v
  ⟷ (∃ w. w∉R ∧ x=u ∧ (u,w)∈E ∧ path (rel-restrict E R) w xs v)
proof –
  have 1: rel-restrict E R ⊆ E ∩ UNIV × − 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 ≠ []
  and path E w p x
  and v ∈ set p
  and dropWhile ((≠) 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.prems obtain y where path E y ps x
      using path-uncons by metis
    moreover from cons.prems 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.prems obtain y where
      path E y ps x and (w,y) ∈ E
      using path-uncons by metis+
    moreover from cons.prems 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 ≡ ∀ i. (r i, r (Suc i))∈E


lemma ipath-conc-conv:
  ipath E (u ⌢ v) ⟷ (∃ a. path E a u (v 0) ∧ 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≠[]
  shows ipath E (p^ω) ⟷ (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 ∀ i. (iter (p) i, iter (p) (Suc i)) ∈ E
      unfolding ipath-def by simp
    hence (∀ i<length p. (p!i,(p@[hd p])!Suc i)∈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≤i2
  shows (r i1,r i2)∈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* $(E \cap (E^* `` \{r\ 0\} \times E^* `` \{r\ 0\}))\ r \longleftrightarrow ipath\ E\ r$
  unfolding *ipath-def*
  by (*auto simp*: *relpow-fun-conv rtrancl-power*)
lemma *ipath-restrict*: *ipath* $E\ r \Longrightarrow ipath\ (E \cap (E^* `` \{r\ 0\} \times E^* `` \{r\ 0\}))\ r$
  by (*simp add*: *ipath-restrict-eq*)

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

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

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

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

lemma *ipath-suffix*: $ipath\ E\ r \Longrightarrow ipath\ E\ (suffix\ i\ r)$
  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* :: $'q\ digraph \Rightarrow\ 'q\ set \Rightarrow bool$
  where $is\text{-}scc\ E\ U \longleftrightarrow U \times U \subseteq E^* \wedge (\forall\ V.\ V \supset U \longrightarrow \neg\ (V \times V \subseteq E^*))$

lemma *scc-non-empty*[*simp*]: $\neg is\text{-}scc\ E\ \{\}$ **unfolding** *is-scc-def* **by** *auto*

lemma *scc-non-empty'*[*simp*]: $is\text{-}scc\ E\ U \Longrightarrow U \neq \{\}$ **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* []

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*: $x \in U$ $y \in U$
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*⊆*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* $\longleftrightarrow$ $U \in$ *UNIV* // *mconn E*

— The strongly connected components are the equivalence classes of the mutually-connected relation on nodes

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

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

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

lemma *find-outside-node*:
  assumes $(u,v) \in E^*$
  assumes $(u,v) \notin (E \cap U \times U)^*$
  assumes $u \in U$ $v \in U$
  shows $\exists u'.\ u' \notin U \land (u,u') \in E^* \land (u',v) \in E^*$
  using *assms*
  apply (*induction*)
  apply *auto* []
  apply *clarsimp*
  by (*metis IntI mem-Sigma-iff rtrancl.simps*)

lemma *is-scc-restrict1*:
  assumes *SCC*: *is-scc E U*
  shows $U \times U \subseteq (E \cap U \times U)^*$
  using *assms*
  unfolding *is-scc-def*
  apply *clarsimp*
  apply (*rule ccontr*)
  apply (*drule* (*2*) *find-outside-node*[*rotated*])
  apply *auto* []
  by (*metis is-scc-closed*[*OF SCC*] *mem-Sigma-iff rtrancl-trans subsetD*)

lemma *is-scc-restrict2*:
  assumes *SCC*: *is-scc E U*

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

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

lemma *is-scc-alt-restrict-path*:
is-scc E U $\longleftrightarrow U \neq \{\} \wedge$
$(U \times U \subseteq (E \cap U \times U)^*) \wedge ((E^* `` ((E^* `` U) - U)) \cap U = \{\})$
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 $\longleftrightarrow$
$U \neq \{\}$
$\wedge\ (\forall u \in U.\ \forall v \in U.\ (u,v) \in (E \cap U \times U)^*)$
$\wedge\ (\forall u \in U.\ \forall v.\ (v \notin U \wedge (u,v) \in E^*) \longrightarrow (\forall u' \in U.\ (v,u') \notin E^*))$
— Alternative, pointwise characterization
unfolding *is-scc-alt-restrict-path*
by *blast*

lemma *is-scc-unique*:
assumes *SCC*: *is-scc E scc* $\quad$ *is-scc E scc'*
and *v*: $v \in scc \quad v \in scc'$
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-eqclasses*)
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* $\Longrightarrow$ *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'*:
$[\![$*is-scc E scc*; *x* $\in$ *scc*; *y* $\in$ *scc*$]\!]$ $\Longrightarrow$ $(x,y) \in (Restr\ E\ scc)^*$
unfolding *is-scc-pointwise*
by *blast*

definition *scc-of* :: ('*v*×'*v*) *set* ⇒ '*v* ⇒ '*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* ⇒ *'v set* ⇒ (*'v*, *'more*) *graph-rec-scheme*
  where *graph-restrict G R* ≡
  ⦇
    g-V = *g-V G*,
    g-E = *rel-restrict* (*g-E G*) *R*,
    g-V0 = *g-V0 G* − *R*,
    ... = *graph-rec.more G*
  ⦈

lemma *graph-restrict-simps*[*simp*]:
  g-V (*graph-restrict G R*) = *g-V G*
  g-E (*graph-restrict G R*) = *rel-restrict* (*g-E G*) *R*
  g-V0 (*graph-restrict G R*) = *g-V0 G* − *R*
  graph-rec.more (*graph-restrict G R*) = *graph-rec.more G*
  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* ≡ *g-V G*
  abbreviation *E* ≡ *g-E G*
  abbreviation *V0* ≡ *g-V0 G*

  abbreviation *reachable* ≡ E^* ‘‘ *V0*
  abbreviation *succ v* ≡ *E* ‘‘ {*v*}

  lemma *finite-V0*: *finite reachable* ⟹ *finite V0* **by** (*auto intro*: *finite-subset*)

  definition *is-run*

— Infinite run, i.e., a rooted infinite path
where *is-run r* ≡ *r 0* ∈ *V0* ∧ *ipath E r*

lemma *run-ipath*: *is-run r* ⟹ *ipath E r* **unfolding** *is-run-def* **by** *auto*
lemma *run-V0*: *is-run r* ⟹ *r 0* ∈ *V0* **unfolding** *is-run-def* **by** *auto*

lemma *run-reachable*: *is-run r* ⟹ *range r* ⊆ *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* ⊆ *V*
assumes *E-ss*: *E* ⊆ *V* × *V*
begin

lemma *reachable-V*: *reachable* ⊆ *V* **using** *V0-ss E-ss* **by** (*auto elim*: *rtrancl-induct*)

lemma *finite-E*: *finite V* ⟹ *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* ∈ *reachable* ⟹ *finite* (*succ v*)
begin

lemma *fb-graph-subset*:
assumes *g-V G*′ = *V*
assumes *g-E G*′ ⊆ *E*
assumes *finite* (*g-V0 G*′)
assumes *g-V0 G*′ ⊆ *reachable*
shows *fb-graph G*′
proof
show *g-V0 G*′ ⊆ *g-V G*′ **using** *reachable-V assms*(*1*, *4*) **by** *simp*
show *g-E G*′ ⊆ *g-V G*′ × *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* ∈ (*g-E G*′)* “ *g-V0 G*′
obtain *u* **where** *2*: *u* ∈ *g-V0 G*′ (*u*, *v*) ∈ (*g-E G*′)* **using** *1* **by** *rule*
have *3*: *u* ∈ *reachable* (*u*, *v*) ∈ *E** **using** *rtrancl-mono assms*(*2*, *4*) *2* **by**

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

 lemma *fb-graph-restrict*: *fb-graph* (*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 $v \in$ *reachable*
 hence *succ v* $\subseteq$ *reachable* **by** (*metis Image-singleton-iff rtrancl-image-advance subsetI*)
 thus *finite* (*succ v*) **using** *assms* **by** (*rule finite-subset*)
qed

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

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

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

 abbreviation G' $\equiv$ *fr-rename-ext ecnv f G*

 lemma G'*-fields*:
  g-V G' = *f*`*V*
  g-V0 G' = *f*`*V0*
  g-E G' = *rename-E f E*
  unfolding *fr-rename-ext-def* **by** *simp-all*

```
definition fi ≡ the-inv-into V f

lemma
  fi-f: x∈V ⟹ fi (f x) = x and
  f-fi: y∈f‘V ⟹ f (fi y) = y and
  fi-f-eq: ⟦f x = y; x∈V⟧ ⟹ fi y = x
  unfolding fi-def
  by (auto
    simp: the-inv-into-f-f f-the-inv-into-f the-inv-into-f-eq INJ)

lemma E'-to-E: (u,v) ∈ g-E G' ⟹ (fi u, fi v)∈E
  using E-ss
  by (auto simp: fi-f G'-fields)

lemma V0'-to-V0: v∈g-V0 G' ⟹ fi v ∈ V0
  using V0-ss
  by (auto simp: fi-f G'-fields)


lemma rtrancl-E'-sim:
  assumes (f u,v')∈(g-E G')*
  assumes u∈V
  shows ∃ v. v' = f v ∧ v∈V ∧ (u,v)∈E*
  using assms
proof (induction f u v' arbitrary: u)
  case (rtrancl-into-rtrancl v' w' u)
  then obtain v w where v' = f v   w' = f w   (v,w)∈E
    by (auto simp: G'-fields)
  hence v∈V   w∈V using E-ss by auto
  from rtrancl-into-rtrancl obtain vv where v' = f vv   vv∈V   (u,vv)∈E*
    by blast
  from ‹v' = f v› ‹v∈V› ‹v' = f vv› ‹vv∈V› have [simp]: vv = v
    using INJ by (metis inj-on-contraD)

  note ‹(u,vv)∈E*›[simplified]
  also note ‹(v,w)∈E›
  finally show ?case using ‹w' = f w› ‹w∈V› by blast
qed auto

lemma rtrancl-E'-to-E: assumes (u,v)∈(g-E G')* shows (fi u, fi v)∈E*
  using assms apply induction
  by (fastforce intro: E'-to-E rtrancl-into-rtrancl)+

lemma G'-invar: graph G'
  apply unfold-locales
proof −
  show g-V0 G' ⊆ g-V G'
    using V0-ss by (auto simp: G'-fields) []
```

```
    show g-E G′ ⊆ g-V G′ × g-V G′
      using E-ss by (auto simp: G′-fields) []
  qed

  sublocale G′: graph G′ using G′-invar .

  lemma G′-finite-reachable:
    assumes finite ((g-E G)* `` g-V0 G)
    shows finite ((g-E G′)* `` g-V0 G′)
  proof −
    have (g-E G′)* `` g-V0 G′ ⊆ f ` (E*``V0)
      apply (clarsimp-all simp: G′-fields(2))
      apply (drule rtrancl-E′-sim)
      using V0-ss apply auto []
      apply auto
      done
    thus ?thesis using finite-subset assms by blast
  qed

  lemma V′-to-V: v ∈ G′.V ⟹ fi v ∈ V
    by (auto simp: fi-f G′-fields)

  lemma ipath-sim1: ipath E r ⟹ ipath G′.E (f o r)
    unfolding ipath-def by (auto simp: G′-fields)

  lemma ipath-sim2: ipath G′.E r ⟹ ipath E (fi o r)
    unfolding ipath-def
    apply (clarsimp simp: G′-fields)
    apply (drule-tac x=i in spec)
    using E-ss
    by (auto simp: fi-f)

  lemma run-sim1: is-run r ⟹ G′.is-run (f o r)
    unfolding is-run-def G′.is-run-def
    apply (intro conjI)
    apply (auto simp: G′-fields) []
    apply (auto simp: ipath-sim1)
    done

  lemma run-sim2: G′.is-run r ⟹ is-run (fi o r)
    unfolding is-run-def G′.is-run-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 +
  gbg-F :: 'Q set set


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

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


  definition
    is-acc :: 'Q word ⇒ bool where is-acc r ≡ (∀ A∈F. ∃∞i. r i ∈ A)

  definition is-acc-run r ≡ is-run r ∧ is-acc r


  lemma is-acc-run r ≡ is-run r ∧ (∀ A∈F. ∃∞i. r i ∈ A)
    unfolding is-acc-run-def is-acc-def .

  lemma acc-run-run: is-acc-run r ⟹ 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*: *finite* (*range r*)
shows *is-acc r* ⟷ (∀ *A*∈*F*. *limit r* ∩ *A* ≠ {})
proof
assume ∀ *A*∈*F*. *limit r* ∩ *A* ≠ {}
thus *is-acc r*
unfolding *is-acc-def*
by (*metis limit-inter-INF*)
next
from *FIN* **have** *FIN′*: ⋀*A*. *finite* (*A* ∩ *range r*)
by *simp*

assume *is-acc r*
hence *AUX*: ∀ *A*∈*F*. $\exists_\infty$*i*. *r i* ∈ (*A* ∩ *range r*)
unfolding *is-acc-def* **by** *auto*
have ∀ *A*∈*F*. *limit r* ∩ (*A* ∩ *range r*) ≠ {}
apply (*rule ballI*)
apply (*drule bspec*[*OF AUX*])
apply (*subst* (*asm*) *fin-ex-inf-eq-limit*[*OF FIN′*])
.
thus ∀ *A*∈*F*. *limit r* ∩ *A* ≠ {}
by *auto*
qed

lemma *is-acc-run-limit-alt*:
assumes *finite* (*E** ‘‘ *V0*)
shows *is-acc-run r* ⟷ *is-run r* ∧ (∀ *A*∈*F*. *limit r* ∩ *A* ≠ {})
using *assms acc-eq-limit*[*symmetric*] **unfolding** *is-acc-run-def*
by (*auto dest*: *run-reachable finite-subset*)

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

lemma *finite-V-Fe*:
assumes *finite V*   *A* ∈ *F*
shows *finite A*
using *assms* **by** (*metis Pow-iff infinite-super rev-subsetD F-ss*)

end

definition *gb-rename-ecnv ecnv f G* ≡ ⦇
gbg-F = { *f*‘*A* | *A*. *A*∈*gbg-F G* }, . . . = *ecnv G*
⦈

```
abbreviation gb-rename-ext ecnv f ≡ fr-rename-ext (gb-rename-ecnv ecnv f) f

locale gb-rename-precond =
  gb-graph G +
  g-rename-precond G f gb-rename-ecnv ecnv f
  for G :: ('u,'more) gb-graph-rec-scheme
  and f :: 'u ⇒ 'v and ecnv
begin
  lemma G'-gb-fields: gbg-F G' = { f`A | A. A∈F }
    unfolding gb-rename-ecnv-def fr-rename-ext-def
    by simp

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

  lemma acc-sim1: is-acc r ⟹ G'.is-acc (f o r)
    unfolding is-acc-def G'.is-acc-def G'-gb-fields
    by (fastforce intro: imageI simp: INFM-nat)

  lemma acc-sim2:
    assumes G'.is-acc r shows is-acc (fi o r)
  proof -
    from assms have 1: ⋀A m. A ∈ gbg-F G ⟹ ∃ i>m. r i ∈ f`A
      unfolding G'.is-acc-def G'-gb-fields
      by (auto simp: INFM-nat)

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

  lemma acc-run-sim1: is-acc-run r ⟹ G'.is-acc-run (f o r)
    using acc-sim1 run-sim1 unfolding G'.is-acc-run-def is-acc-run-def
    by auto

  lemma acc-run-sim2: G'.is-acc-run r ⟹ is-acc-run (fi o r)
    using acc-sim2 run-sim2 unfolding G'.is-acc-run-def is-acc-run-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 labeld with this word.

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

locale *gba* =
  gb-graph G
  for *G* :: ('Q,'L,'more) *gba-rec-scheme* +
  assumes *L-ss*: *gba-L G q l* ⟹ *q* ∈ *V*
begin
  abbreviation *L* ≡ *gba-L G*

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

  definition *accept w* ≡ ∃ *r*. *is-acc-run r* ∧ (∀ *i*. *L* (*r i*) (*w i*))
  lemma *acceptI*[*intro?*]: ⟦*is-acc-run r*; ⋀*i*. *L* (*r i*) (*w i*)⟧ ⟹ *accept w*
    by (*auto simp*: *accept-def*)

  definition *lang* ≡ *Collect* (*accept*)
  lemma *langI*[*intro?*]: *accept w* ⟹ *w*∈*lang* **by** (*auto simp*: *lang-def*)
end

definition *gba-rename-ecnv ecnv f G* ≡ ⦇
  gba-L = λ*q l*.
    if q∈*f*‘*g-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* ≡ *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 ⇒ 'v **and** *ecnv*
begin
  lemma *G'-gba-fields*: *gba-L G'* = (λ*q l*.
    if q∈*f*‘*V then L* (*fi 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: ⟦range r ⊆ V; L (r i) l⟧ ⟹ 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: ⟦ range r ⊆ f‘V; G′.L (r i) l ⟧ ⟹ 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-acc-run r
  assume L: ∀ i. G′.L (r i) (w i)
  from R have RAN: range r ⊆ f‘V
    using G′.run-reachable[OF G′.acc-run-run[OF R]] G′.reachable-V
    unfolding G′-fields
    by simp
  from L show ∃ r. is-acc-run r ∧ (∀ i. L (r i) (w i))
    using acc-run-sim2[OF R] L-sim2[OF RAN]
    by auto
next
  fix w r
  assume R: is-acc-run r
  assume L: ∀ i. L (r i) (w i)

  from R have RAN: range r ⊆ V
    using run-reachable[OF acc-run-run[OF R]] reachable-V by simp

  from L show ∃ r.
      G′.is-acc-run r
    ∧ (∀ i. G′.L (r i) (w i))
    using acc-run-sim1[OF R] L-sim1[OF RAN]
    by auto
qed

lemma lang-eq[simp]: G′.lang = lang
  unfolding G′.lang-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 ≡ 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′ ≡ gba-rename f G
 shows gba G′
 and finite (g-V G) ⟹ 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) ⟹ 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 ⊆ V
begin
 abbreviation F where F ≡ bg-F G

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

 definition to-gbg-ext m
  ≡ (| 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* ≡ ($\exists_\infty$*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 ≡ (∃ r. is-acc-run r ∧ (∀ i. L (r i) (w i)))
  definition lang ≡ Collect 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 ⇒ nat set

locale igb-graph =
  graph G
  for G :: ('Q,'more) igb-graph-rec-scheme
  +
  assumes acc-bound: ⋃(range (igbg-acc G)) ⊆ {0..<(igbg-num-acc G)}
  assumes acc-ss: igbg-acc G q ≠ {} ⟹ q∈V
begin
  abbreviation num-acc where num-acc ≡ igbg-num-acc G
  abbreviation acc where acc ≡ igbg-acc G

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


  lemma acc-boundI[simp, intro]: x∈acc q ⟹ x<num-acc
    using acc-bound by fastforce
```

definition *accn i* ≡ {*q* . *i*∈*acc q*}
definition *F* ≡ { *accn i* | *i*. *i*<*num-acc* }

definition *to-gbg-ext m*
  ≡ (| *g-V* = *V*, *g-E* = *E*, *g-V0* = *V0*, *gbg-F* = *F*, ...=*m* |)

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* ⇒ *bool*
  where *is-acc r* ≡ (∀ *n*<*num-acc*. $\exists_\infty$*i*. *n* ∈ *acc* (*r i*))
definition *is-acc-run r* ≡ *is-run r* ∧ *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 ⟷ (∀ n<num-acc. limit r ∩ accn n ≠ {})
proof
  assume ∀ n<num-acc. limit r ∩ accn n ≠ {}
  thus is-acc r
    unfolding is-acc-def accn-def
    by (auto dest!: limit-inter-INF)
next
  from FIN have FIN': ⋀A. finite (A ∩ range r) by simp
  assume is-acc r
  hence ∀ n<num-acc. limit r ∩ (accn n ∩ range r) ≠ {}
    unfolding is-acc-def accn-def
    by (auto simp: fin-ex-inf-eq-limit[OF FIN', symmetric])
  thus ∀ n<num-acc. limit r ∩ accn n ≠ {} by auto
qed

lemma acc-limit-alt':
  finite (range r) ⟹ is-acc r ⟷ (⋃(acc ` limit r) = {0..<num-acc})
  unfolding acc-limit-alt
  by (auto simp: accn-def)

end

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

locale igba =
  igbg?: igb-graph G
  for G :: ('Q,'L,'more) igba-rec-scheme
  +
  assumes L-ss: igba-L G q l ⟹ q ∈ V
begin
  abbreviation L where L ≡ igba-L G

  lemma is-igba: igba G by unfold-locales


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

  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* ≡ ∃ *r*. *is-acc-run r* ∧ (∀ *i*. *L* (*r i*) (*w i*))
definition *lang* ≡ *Collect 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 set set* ⇒ (*nat* × (*'Q* ⇒ *nat set*)) *nres* **where**
F-to-idx F ≡ *do* {
Flist ← *SPEC* (λ*Flist*. *distinct Flist* ∧ *set Flist = F*);
let num-acc = length Flist;
let acc = (λ*v*. {*i* . *i*<*num-acc* ∧ *v*∈*Flist*!*i*});
RETURN (*num-acc*,*acc*)
}

lemma *F-to-idx-correct*:
shows *F-to-idx F* ≤ *SPEC* (λ(*num-acc*,*acc*). *F* = { {*q*. *i*∈*acc q*} | *i*. *i*<*num-acc*
}
∧ ⋃(*range acc*) ⊆ {*0*..<*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* ≡ *do* {
let acc = Map.empty;

(-,*acc*) ← *nfoldli Flist* (λ-. *True*) (λ*A* (*i*,*acc*). *do* {
acc ← *FOREACHi* (λ*it 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)∈Id
  assumes FIN: ∀A∈set Flist. finite A
 shows mk-acc-impl Flist′ ≤ ⇓Id (RETURN (λv. {i . i<length Flist ∧ v∈Flist!i}))
  using F apply simp
  unfolding mk-acc-impl-def

  apply (refine-rcg
    nfoldli-rule[where
      I=λl1 l2 (i,res). i=length l1
        ∧ the-default {} o res = (λv. {j . j<i ∧ v∈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* ⇒ (*nat* × (*′Q* ⇒ *nat set*)) *nres* **where**

```
  F-to-idx-impl F ≡ do {
    Flist ← SPEC (λFlist. distinct Flist ∧ set Flist = F);
    let num-acc = length Flist;
    acc ← mk-acc-impl Flist;
    RETURN (num-acc,acc)
  }
```

lemma *F-to-idx-refine*:

```
  assumes FIN: ∀A∈F. finite A
  shows F-to-idx-impl F ≤ ⇓Id (F-to-idx F)
  using assms
```

unfolding *F-to-idx-impl-def F-to-idx-def*

apply (*refine-rcg bind-Let-refine2*[*OF mk-acc-impl-correct*])

apply *auto*
done

definition *gbg-to-idx-ext*
:: - ⇒ (*'a*, *'more*) *gb-graph-rec-scheme* ⇒ (*'a*, *'more'*) *igb-graph-rec-scheme nres*
where *gbg-to-idx-ext ecnv A* = *do* {
(*num-acc*,*acc*) ← *F-to-idx-impl* (*gbg-F A*);
RETURN (|
g-V = *g-V A*,
g-E = *g-E A*,
g-V0=*g-V0 A*,
igbg-num-acc = *num-acc*,
igbg-acc = *acc*,
... = *ecnv A*
|)
}

lemma (**in** *gb-graph*) *gbg-to-idx-ext-correct*:
assumes [*simp*, *intro*]: $\bigwedge$ *A*. *A* ∈ *F* ⟹ *finite A*
shows *gbg-to-idx-ext ecnv G* ≤ *SPEC* (λ*G'*.
igb-graph.is-acc-run G' = *is-acc-run*
∧ *g-V G'* = *V*
∧ *g-E G'* = *E*
∧ *g-V0 G'* = *V0*
∧ *igb-graph-rec.more G'* = *ecnv G*
∧ *igb-graph G'*
)
proof −
note *F-to-idx-refine*[*of F*]
also note *F-to-idx-correct*
finally have *R*: *F-to-idx-impl F*
≤ *SPEC* (λ(*num-acc*, *acc*). *F* = {{*q*. *i* ∈ *acc q*} |*i*. *i* < *num-acc*}
∧ $\bigcup$(*range acc*) ⊆ {*0*..<*num-acc*}) **by** *simp*

have *eq-conjI*: $\bigwedge$*a b c*. (*b*⟷*c*) ⟹ (*a*&*b* ⟷ *a*&*c*) **by** *simp*

{
fix *acc* :: *'Q* ⇒ *nat set* **and** *num-acc r*
have (∀ *A*. (∃ *i*. *A* = {*q*. *i* ∈ *acc q*} ∧ *i* < *num-acc*) ⟶ (*limit r* ∩ *A* ≠ {}))
⟷ (∀ *i*<*num-acc*. ∃ *q*∈*limit r*. *i*∈*acc q*)
by *blast*
} **note** *aux1*=*this*

{
fix *acc* :: *'Q* ⇒ *nat set* **and** *num-acc i*

```
  assume FE: F = {{q. i ∈ acc q} |i. i < num-acc}
  assume INR: (⋃x. acc x) ⊆ {0..<num-acc}
  have finite {q. i ∈ acc q}
  proof (cases i<num-acc)
    case True thus ?thesis using FE by auto
  next
    case False hence {q. i ∈ acc q} = {} using INR by force
    thus ?thesis by simp
  qed
} note aux2=this

{
  fix acc :: 'Q ⇒ nat set and num-acc q
  assume FE: F = {{q. i ∈ acc q} |i. i < num-acc}
    and INR: (⋃x. acc x) ⊆ {0..<num-acc}
    and acc q ≠ {}
  then obtain i where i∈acc q by auto
  moreover with INR have i<num-acc by force
  ultimately have q∈⋃F by (auto simp: FE)
  with F-ss have q∈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 ∈ acc q} |i. i < num-acc}
    and BOUND: (⋃x. acc x) ⊆ {0..<num-acc}
  let ?G' = (|
    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* ⇒ '*q igb-graph-rec nres*
where *gbg-to-idx* ≡ *gbg-to-idx-ext* (λ-. ())

definition *ti-Lcnv* **where** *ti-Lcnv ecnv A* ≡ (| *igba-L* = *gba-L A*, ...=*ecnv A* |)

abbreviation *gba-to-idx-ext ecnv* ≡ *gbg-to-idx-ext* (*ti-Lcnv ecnv*)
abbreviation *gba-to-idx* ≡ *gba-to-idx-ext* (λ-. ())

lemma (**in** *gba*) *gba-to-idx-ext-correct*:
assumes [*simp*, *intro*]: ⋀ *A*. *A* ∈ *F* ⟹ *finite A*
shows *gba-to-idx-ext ecnv G* ≤
SPEC (λ*G*′.
igba.accept G′ = *accept*
∧ *g-V G*′ = *V*
∧ *g-E G*′ = *E*
∧ *g-V0 G*′ = *V0*
∧ *igba-rec.more G*′ = *ecnv G*
∧ *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*]: $\bigwedge A.\ A \in F \Longrightarrow finite\ A$
shows *gba-to-idx-ext ecnv G* $\leq$
$SPEC\ (\lambda G'.\ igba.lang\ G' = lang \wedge igba\text{-}rec.more\ G' = ecnv\ G \wedge igba\ G')$
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* :: - ⇒ ($'Q \times nat$, -) *b-graph-rec-scheme* **where**
degeneralize-ext ecnv ≡
if num-acc = 0 then (|
$g\text{-}V = V \times \{0\}$,
$g\text{-}E = \{((q,0),(q',0)) \mid q\ q'.\ (q,q') \in E\}$,
$g\text{-}V0 = V0 \times \{0\}$,
$bg\text{-}F = V \times \{0\}$,
$\ldots = ecnv\ G$
|)
else (|
$g\text{-}V = V \times \{0..<num\text{-}acc\}$,
$g\text{-}E = \{ ((q,i),(q',i')) \mid i\ i'\ q\ q'.$
$i<num\text{-}acc$
$\wedge (q,q') \in E$
$\wedge i' = (if\ i \in acc\ q\ then\ (i+1)\ mod\ num\text{-}acc\ else\ i) \}$,
$g\text{-}V0 = V0 \times \{0\}$,
$bg\text{-}F = \{(q,0) \mid q.\ 0 \in acc\ q\}$,
$\ldots = ecnv\ G$
|)

abbreviation *degeneralize* **where** *degeneralize* ≡ *degeneralize-ext* (λ-. ())

lemma *degen-more*[*simp*]: *b-graph-rec.more* (*degeneralize-ext ecnv*) = *ecnv G*
unfolding *degeneralize-ext-def*

```
  by auto

lemma degen-invar: b-graph (degeneralize-ext ecnv)
proof
  let ?G′ = degeneralize-ext ecnv

  show g-V0 ?G′ ⊆ g-V ?G′
    unfolding degeneralize-ext-def
    using V0-ss
    by auto

  show g-E ?G′ ⊆ g-V ?G′ × g-V ?G′
    unfolding degeneralize-ext-def
    using E-ss
    by auto

  show bg-F ?G′ ⊆ 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′ = degeneralize-ext ecnv
  have ((g-E ?G′)* `` g-V0 ?G′)
    ⊆ E* ``V0 × {0..num-acc}
  proof −
    {
      fix q n q′ n′
      assume ((q,n),(q′,n′))∈(g-E ?G′)*
        and 0: (q,n)∈g-V0 ?G′
      hence G1: (q,q′)∈E* ∧ n′≤num-acc
        apply (induction rule: rtrancl-induct2)
        by (auto simp: degeneralize-ext-def split: if-split-asm)

      from 0 have G2: q∈V0 ∧ n≤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 ⟹ *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* ∈ *degen.V0 T m*
  shows *fst u* ∈ *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*≠*n′*
  shows ∃ *qa*. (*qa*,*n*)∈*set p* ∧ *n*∈*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 ‹((*q*,*n*),*qnh*) ∈ *degen.E T m*›
  have *nh*=*n* ∨ (*nh*=(*n*+*1*) *mod num-acc* ∧ *n*∈*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*) ∈ *set p* **and** *n* ∈ *acc qa*
      by *auto*
    thus *?case* **by** *auto*
  next
    assume (*nh*=(*n*+*1*) *mod num-acc* ∧ *n*∈*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* (λ*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 ∃r'. degen.is-run T m r' ∧ r = fst o r'
  using R
  unfolding degen.is-run-def is-run-def ipath-def
  apply (elim conjE)
proof −
  assume R0: r 0 ∈ V0 and RS: ∀i. (r i, r (Suc i)) ∈ E

  define r' where r' = rec-nat
    (r 0,0)
    (λi (q,n). (r (Suc i), if n ∈ acc q then (n+1) mod num-acc else n))

  have [simp]:
    r' 0 = (r 0,0)
    ⋀i. r' (Suc i) = (
      let
        (q,n)=r' i
      in
        (r (Suc i), if n ∈ acc q then (n+1) mod num-acc else n)
    )
    unfolding r'-def
    by auto

  have R0': r' 0 ∈ degen.V0 T m using R0
    unfolding degeneralize-ext-def by auto

  have MAP: r = fst o r'
  proof (rule ext)
    fix i
    show r i = (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: ⋀i. snd (r′ i) < num-acc
  proof −
    fix i show snd (r′ i) < num-acc by (induction i) (auto split: prod.split)
  qed

  have RS′: ∀ i. (r′ i, r′ (Suc i)) ∈ 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<num-acc by simp

    show (r′ i, r′ (Suc i)) ∈ degen.E T m using RS
      by (auto simp: degeneralize-ext-def)
  qed

  from R0′ RS′ MAP show
    ∃ r′. (r′ 0 ∈ degen.V0 T m
    ∧ (∀ i. (r′ i, r′ (Suc i)) ∈ degen.E T m))
    ∧ r = fst ∘ r′ by blast
qed

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

lemma degen-acc-run-complete-aux1:
  assumes NN0[simp]: num-acc ≠ 0
  assumes R: degen.is-run T m r
  assumes EXJ: ∃ j≥i. n ∈ acc (fst (r j))
  assumes RI: r i = (q,n)
  shows ∃ j≥i. ∃ q′. r j = (q′,n) ∧ n ∈ acc q′
proof −
  define j where j = (LEAST j. j≥i ∧ n ∈ acc (fst (r j)))
```

```
  from RI have n<num-acc using degen-run-bound[OF NN0 R, of i] by auto
  from EXJ have
    j≥i
    n ∈ acc (fst (r j))
    ∀k≥i. n ∈ acc (fst (r k)) ⟶ j≤k
    using LeastI-ex[OF EXJ]
    unfolding j-def
    apply (auto) [2]
    apply (metis (lifting) Least-le)
    done
  hence ∀k≥i. k<j ⟶ n ∉ acc (fst (r k)) by auto

  have ∀k. k≥i ∧ k≤j ⟶ (snd (r k) = n)
  proof (clarify)
    fix k
    assume i≤k   k≤j
    thus 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.prems have k − 1 < k   i ≤ k − 1   k − 1≤j
          by auto
        from less.IH[OF this] have snd (r (k − 1)) = n .
        moreover from R have
          (r (k − 1), r k) ∈ degen.E T m
          unfolding degen.is-run-def is-run-def ipath-def
          by clarsimp (metis One-nat-def Suc-diff-1 ‹k − 1 < k›
            less-nat-zero-code neq0-conv)
        moreover have n ∉ acc (fst (r (k − 1)))
          using ‹∀k≥i. k < j ⟶ n ∉ acc (fst (r k))› ‹i ≤ k − 1› ‹k − 1 < k›
            dual-order.strict-trans1 less.prems(2)
            by blast
        ultimately show ?thesis
          by (auto simp: degeneralize-ext-def)
      qed
    qed
  qed

  thus ?thesis
    by (metis ‹i ≤ j› ‹n ∈ local.acc (fst (r j))›
      order-refl surjective-pairing)
qed

lemma degen-acc-run-complete-aux1':
  assumes NN0[simp]: num-acc ≠ 0
  assumes R: degen.is-run T m r
  assumes ACC: ∀n<num-acc. ∃∞i. n ∈ acc (fst (r i))
```

assumes *RI*: *r i* = (*q*,*n*)
shows ∃ *j*≥*i*. ∃ *q*′. *r j* = (*q*′,*n*) ∧ *n* ∈ *acc q*′
proof −
from *RI* **have** *n*<*num-acc* **using** *degen-run-bound*[*OF NN0 R*, *of i*] **by** *auto*
with *ACC* **have** *EXJ*: ∃ *j*≥*i*. *n* ∈ *acc* (*fst* (*r j*))
unfolding *INFM-nat-le* **by** *blast*

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

lemma *degen-acc-run-complete-aux2*:
assumes *NN0*[*simp*]: *num-acc* ≠ *0*
assumes *R*: *degen.is-run T m r*
assumes *ACC*: ∀ *n*<*num-acc*. $\exists_\infty$*i*. *n* ∈ *acc* (*fst* (*r i*))
assumes *RI*: *r i* = (*q*,*n*) **and** *OFS*: *ofs*<*num-acc*
shows ∃ *j*≥*i*. ∃ *q*′.
r j = (*q*′,(*n* + *ofs*) *mod num-acc*) ∧ (*n* + *ofs*) *mod num-acc* ∈ *acc q*′
using *RI OFS*
proof (*induction ofs arbitrary*: *q n i*)
case *0*
from *degen-run-bound*[*OF NN0 R*, *of i*] ‹*r i* = (*q*, *n*)›
have *NLE*: *n*<*num-acc*
by *simp*

with *degen-acc-run-complete-aux1*′[*OF NN0 R ACC* ‹*r i* = (*q*, *n*)›] **show** *?case*
by *auto*
next
case (*Suc ofs*)
from *Suc.IH*[*OF Suc.prems*(*1*)] *Suc.prems*(*2*)
obtain *j q*′ **where** *j*≥*i* **and** *RJ*: *r j* = (*q*′,(*n*+*ofs*) *mod num-acc*)
and *A*: (*n*+*ofs*) *mod num-acc* ∈ *acc q*′
by *auto*
from *R* **have** (*r j*, *r* (*Suc j*)) ∈ *degen.E T m*
by (*auto simp*: *degen.is-run-def is-run-def ipath-def*)
with *RJ A* **obtain** *q2* **where** *RSJ*: *r* (*Suc j*) = (*q2*,(*n*+*Suc ofs*) *mod num-acc*)

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

have *aux*: ⋀*j*′. *i*≤*j* ⟹ *Suc j* ≤ *j*′ ⟹ *i*≤*j*′ **by** *auto*
from *degen-acc-run-complete-aux1*′[*OF NN0 R ACC RSJ*] ‹*j*≥*i*›
show *?case*
by (*auto dest*: *aux*)
qed

lemma *degen-acc-run-complete*:
assumes *AR*: *is-acc-run r*
obtains *r*′
where *degen.is-acc-run T m r*′ **and** *r* = *fst o r*′
proof (*cases num-acc* = *0*)

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

  from AR have R: is-run r and ACC: ∀n<num-acc. ∃∞i. n ∈ 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 ∘ r'
    by auto

  from ACC have ACC': ∀n<num-acc. ∃∞i. n ∈ acc (fst (r' i)) by simp

  have ∀i. ∃j>i. r' j ∈ degen.F T m
  proof
    fix i
    obtain q n where RI: r' (Suc i) = (q,n) by (cases r' (Suc i))
    have (n + (num-acc − n mod num-acc)) mod 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 + ofs) mod 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∈acc q'
      by (auto simp: less-eq-Suc-le)
    thus ∃j>i. r' j ∈ degen.F T m
      by (auto simp: degeneralize-ext-def)
  qed
  hence ∃∞i. r' i ∈ 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 ≠ 0
  assumes R: degen.is-run T m r
  assumes A: i≤j    r i = (q,n)    r j = (q',n')    n≠n'
```

```
  obtains k qk where i≤k   k<j   r k = (qk,n)   n ∈ acc qk
proof −
  from degen-run-bound[OF NN0 R] A have n<num-acc   n'<num-acc
    by (metis snd-conv)+

  define k where k = (LEAST k. i<k ∧ snd (r k) ≠ n)

  have i<k   snd (r k) ≠ n
    by (metis (lifting, mono-tags) LeastI-ex A k-def leD less-linear snd-conv)+

  from Least-le[where P=λk. i<k ∧ snd (r k) ≠ n, folded k-def]
  have LEK-EQN: ∀k'. i≤k' ∧ k'<k ⟶ snd (r k') = n
    using ‹r i = (q,n)›
    by clarsimp (metis le-neq-implies-less not-le snd-conv)
  hence SND-RKMO: snd (r (k − 1)) = n using ‹i<k› by auto
  moreover from R have (r (k − 1), r k) ∈ degen.E T m
    unfolding degen.is-run-def ipath-def using ‹i<k›
    by clarsimp (metis Suc-pred gr-implies-not0 neq0-conv)
  moreover note ‹snd (r k) ≠ n›
  ultimately have n ∈ acc (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 ‹snd (r (k − 1)) = n› less-Suc-eq
      less-imp-diff-less not-less-eq snd-conv)
  ultimately show thesis
    apply −
    apply (rule that[of k − 1   fst (r (k − 1))])
    using ‹i<k› SND-RKMO by auto
qed


lemma degen-run-find-acc-aux:
  assumes NN0[simp]: num-acc ≠ 0
  assumes AR: degen.is-acc-run T m r
  assumes A: r i = (q,0)   0 ∈ acc q   n<num-acc
  shows ∃j qj. i≤j ∧ r j = (qj,n) ∧ n ∈ acc qj
proof −
  from AR have R: degen.is-run T m r
    and ACC: ∃∞i. r i ∈ degen.F T m

    unfolding degen.is-acc-run-def degen.is-acc-def by auto
  from ACC have ACC': ∀i. ∃j>i. r j ∈ degen.F T m
    by (auto simp: INFM-nat)

  show ?thesis using ‹n<num-acc›
  proof (induction n)
    case 0 thus ?case using A by auto
```

```
  next
    case (Suc n)
    then obtain j qj where i≤j   r j = (qj,n)   n∈acc qj by auto
    moreover from R have (r j, r (Suc j)) ∈ degen.E T m
      unfolding degen.is-run-def ipath-def
      by auto
    ultimately obtain qsj where RSJ: r (Suc j) = (qsj,Suc n)
      unfolding degeneralize-ext-def using ‹Suc n<num-acc› by auto

    from ACC' obtain k q0 where Suc j ≤ k   r k = (q0, 0)
      unfolding degeneralize-ext-def apply auto
      by (metis less-imp-le-nat)
    from degen-run-find-change[OF NN0 R ‹Suc j ≤ k› RSJ ‹r k = (q0, 0)›]
    obtain l ql where
      Suc j ≤ l   l < k   r l = (ql, Suc n)   Suc n ∈ acc ql
      by blast
    thus ?case using ‹i ≤ j›
      by (intro exI[where x=l] exI[where x=ql]) auto
  qed
qed

lemma degen-acc-run-sound:
  assumes A: degen.is-acc-run T m r
  shows is-acc-run (fst o r)
proof -
  from A have R: degen.is-run T m r
    and ACC: ∃∞i. r i ∈ 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-run (fst o r) .

  show ?thesis
  proof (cases num-acc = 0)
    case NN0[simp]: False

    from ACC have ACC': ∀ i. ∃j>i. r j ∈ degen.F T m
      by (auto simp: INFM-nat)

    have ∀ n<num-acc. ∀ i. ∃j>i. n ∈ 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 ∈ acc (qj)
        using ACC' unfolding degeneralize-ext-def by fastforce

      assume NLESS: n<num-acc
      show ∃j>i. n ∈ acc (fst (r j))
      proof (cases n)
        case 0 thus ?thesis using ‹j>i› 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*. $\exists_\infty$*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]: ⋀q s. (q,s) ∈ g-V0 prod ⟷ q ∈ igba.V0 ∧ s ∈ sa.V0
    by (auto simp: prod-def)

  have reachSS:
    ((g-E prod)* `` g-V0 prod)
    ⊆ ((igba.E)* `` igba.V0) × (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 × sa.V
  prod.E = { ((q,s),(q′,s′)).
    igba.L q (sa.L s) ∧ (q,q′) ∈ igba.E ∧ (s,s′) ∈ sa.E }
  prod.V0 = igba.V0 × sa.V0
  prod.num-acc = igba.num-acc
  prod.acc = (λ(q,s). if s∈sa.V then igba.acc q else {} )
  unfolding prod-def
  apply simp-all
  done

lemma prod-run: prod.is-run r ⟷
    igba.is-run (fst o r)
  ∧ sa.is-run (snd o r)
  ∧ (∀ 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) ⊆ sa.V
  shows prod.is-acc r ⟷ igba.is-acc (fst o r)
proof –
  {
```

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

lemma gsp-correct1:
  assumes A: prod.is-acc-run r
  shows sa.is-run (snd o r) ∧ (sa.L o snd o r ∈ igba.lang)
proof −
  from A have PR: prod.is-run r and PA: prod.is-acc r
    unfolding prod.is-acc-run-def by auto

   have PRR: range r ⊆ prod.V using prod.run-reachable prod.reachable-V PR
by auto

  have RSR: range (snd o r) ⊆ sa.V using PRR unfolding prod-fields by auto

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

lemma gsp-correct2:
  assumes A: sa.is-run r    sa.L o r ∈ igba.lang
  shows ∃ r'. r = snd o r' ∧ prod.is-acc-run r'
proof −
  have [simp]: ⋀r r'. fst o (λi. (r i, r' i)) = r
    ⋀r r'. snd o (λi. (r i, r' i)) = r'
    by auto

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

    apply (subst prod-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 ≡ case lasso-reach L of [] ⇒ lasso-va L | (v0#-) ⇒ v0
  definition lasso-cycle where lasso-cycle L = lasso-va L # lasso-cysfx L


  definition lasso-of-prpl prpl ≡ case prpl of (pr,pl) ⇒ (|
    lasso-reach = pr,
    lasso-va = hd pl,
    lasso-cysfx = tl pl |)

  definition prpl-of-lasso L ≡ (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 ≠ [] ⟹ 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 ⇒ 'q word
    — Run described by a lasso
    where run-of-lasso L ≡ lasso-reach L ⌢ (lasso-cycle L)^ω

  lemma run-of-lasso-of-prpl:
```

pl ≠ [] ⟹ *run-of-lasso* (*lasso-of-prpl* (*pr*, *pl*)) = *pr* ⌢ pl^ω
unfolding *run-of-lasso-def lasso-of-prpl-def lasso-cycle-def*
by *auto*

definition *map-lasso f L* ≡ ⦇
lasso-reach = *map f* (*lasso-reach L*),
lasso-va = *f* (*lasso-va L*),
lasso-cysfx = *map f* (*lasso-cysfx L*)
⦈

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

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

context *graph* **begin**
definition *is-lasso-pre* :: *'v lasso* ⇒ *bool*
where *is-lasso-pre L* ≡
lasso-v0 L ∈ *V0*
∧ *path E* (*lasso-v0 L*) (*lasso-reach L*) (*lasso-va L*)
∧ *path E* (*lasso-va L*) (*lasso-cycle L*) (*lasso-va L*)

definition *is-lasso-prpl-pre prpl* ≡ *case prpl of* (*pr*, *pl*) ⇒ ∃ *v0 va*.
v0∈*V0*
∧ *pl* ≠ []
∧ *path E v0 pr va*
∧ *path E va pl va*

lemma *is-lasso-pre-prpl-of-lasso*[*simp*]:
is-lasso-prpl-pre (*prpl-of-lasso L*) ⟷ *is-lasso-pre L*
unfolding *is-lasso-pre-def prpl-of-lasso-def is-lasso-prpl-pre-def*
unfolding *lasso-v0-def lasso-cycle-def*
by (*auto simp*: *path-simps split*: *list.split*)

lemma *is-lasso-prpl-pre-conv*:
is-lasso-prpl-pre prpl
⟷ (*snd prpl*≠[] ∧ *is-lasso-pre* (*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*]: $V0 = \{\} \Longrightarrow \neg$*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* $\in$ *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* $\Rightarrow$ *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* = {} ⟹ ¬*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** (∀ *A*∈*F*. *set* (*va*#*pls*) ∩ *A* ≠ {})
by (*auto simp*: *is-lasso-def lasso-cycle-def*)
moreover from *L* **have** (*run-of-lasso L*) *0* ∈ *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*) ⟹ *is-acc-run* (*pr* ⌢ *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^* `` *V0*)
assumes *A*: *is-acc-run r*
shows ∃ *L*. *is-lasso L*
proof −
from *A* **have**
RUN: *is-run r*
and *ACC*: ∀ *A*∈*F*. *limit r* ∩ *A* ≠ {}
by (*auto simp*: *is-acc-run-limit-alt*)
from *RUN* **have** [*simp*]: *r 0* ∈ *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* ⊆ E^* ``*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 ∃pl. pl≠[] ∧ path E u pl u ∧ (∀A∈F. set pl ∩ A ≠ {})
    using finite-F ACC
  proof (induction rule: finite-induct)
    case empty
    from run-limit-two-connectedI[OF RUN' ‹u∈limit r› ‹u∈limit r›]
    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.prems obtain pl where
      [simp]: pl≠[]
        and P: path E u pl u
        and ACC: (∀A'∈F. set pl ∩ A' ≠ {})
      by auto
    from insert.prems obtain v where VACC: v∈A   v∈limit r by auto
    from run-limit-two-connectedI[OF RUN' ‹u∈limit r› ‹v∈limit r›]
    obtain p1 where [simp]: p1≠[]
      and P1: path E u p1 v
      by (rule trancl-is-path)

    from run-limit-two-connectedI[OF RUN' ‹v∈limit r› ‹u∈limit r›]
    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 (∀A'∈insert A F. set (pl@p1@p2) ∩ A' ≠ {}) 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   (∀A∈F. set pl ∩ A ≠ {})
    by blast
  then obtain pls where path E u (u#pls) u   ∀A∈F. set (u#pls) ∩ A ≠ {}
    by (cases pl) (auto simp add: path-simps)
  ultimately show ?thesis
```

```
    apply −
    apply (rule
      exI[where x=(|lasso-reach = pr,lasso-va = u,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 ≡
    is-lasso-pre L
    ∧ (set (lasso-cycle L)) ∩ F ≠ {}

  definition is-lasso-prpl where is-lasso-prpl L ≡
    is-lasso-prpl-pre L
    ∧ (set (snd L)) ∩ F ≠ {}

  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 (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) ⟷ 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 ⟷ (snd prpl≠[] ∧ 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 *lasso-prpl-acc-run*:
is-lasso-prpl (*pr*, *pl*) $\Longrightarrow$ *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* ≡
is-lasso-pre L
∧ (∀ *i*<*num-acc*. ∃ *q*∈*set* (*lasso-cycle L*). *i*∈*acc q*)

definition *is-lasso-prpl L* ≡
is-lasso-prpl-pre L
∧ (∀ *i*<*num-acc*. ∃ *q*∈*set* (*snd L*). *i*∈*acc q*)

lemma *is-lasso-prpl-of-lasso*[*simp*]:
is-lasso-prpl (*prpl-of-lasso L*) ⟷ *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 ⟷ (*snd prpl*≠[] ∧ *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*) ⟹ *is-acc-run* (*pr* ⌢ *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* ∈ *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*)) ∩ *degen.F T m* ≠ {}
    unfolding *degen.is-lasso-def degen.is-lasso-pre-def* **by** *auto*

  {
    fix *i*
    assume *i*<*num-acc*
    hence ∃ *q*∈*set* (*lasso-cycle L*). *i* ∈ *local.acc* (*fst q*) ∧ *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*)∈*set* (*lasso-cycle L*) **and** *i*∈*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′*≠[] **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* ‹*i*∈*acc q*› ‹*Suc i* < *num-acc*›
        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 ‹*Suc i* < *num-acc*›

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*: ⋀*Re R*. *lasso-rel-ext Re R* ≡ {
((| *lasso-reach* = *r'*, *lasso-va* = *va'*, *lasso-cysfx* = *cysfx'*, ...=*m'* |),
(| *lasso-reach* = *r*, *lasso-va* = *va*, *lasso-cysfx* = *cysfx*, ...=*m* |)) |
r' r va' va cysfx' cysfx m' m.
(*r'*,*r*) ∈ ⟨*R*⟩*list-rel*
∧ (*va'*,*va*)∈*R*
∧ (*cysfx'*, *cysfx*) ∈ ⟨*R*⟩*list-rel*
∧ (*m'*,*m*) ∈ *Re*
}

lemma *lasso-rel-ext-def*: ⋀ *Re R*. ⟨*Re*,*R*⟩*lasso-rel-ext* = {
((| *lasso-reach* = *r'*, *lasso-va* = *va'*, *lasso-cysfx* = *cysfx'*, ...=*m'* |),
(| *lasso-reach* = *r*, *lasso-va* = *va*, *lasso-cysfx* = *cysfx*, ...=*m* |)) |
r' r va' va cysfx' cysfx m' m.
(*r'*,*r*) ∈ ⟨*R*⟩*list-rel*
∧ (*va'*,*va*)∈*R*
∧ (*cysfx'*, *cysfx*) ∈ ⟨*R*⟩*list-rel*
∧ (*m'*,*m*) ∈ *Re*
}
unfolding *lasso-rel-ext-internal-def relAPP-def* **by** *auto*

lemma *lasso-rel-ext-sv*[*relator-props*]:
⋀ *Re R*. ⟦ *single-valued Re*; *single-valued R* ⟧ ⟹ *single-valued* (⟨*Re*,*R*⟩*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*. ⟦ *Re*=*Id*; *R*=*Id* ⟧ ⟹ ⟨*Re*,*R*⟩*lasso-rel-ext* = *Id*
  unfolding *lasso-rel-ext-def*
  apply *simp*
  apply *safe*
  by (*metis lasso.surjective*)

consts *i-lasso-ext* :: *interface* ⇒ *interface* ⇒ *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*. (*lasso-reach*, *lasso-reach*) ∈ ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*R*⟩*list-rel*
  $\bigwedge$*Re R*. (*lasso-va*, *lasso-va*) ∈ ⟨*Re*,*R*⟩*lasso-rel-ext* → *R*
  $\bigwedge$*Re R*. (*lasso-cysfx*, *lasso-cysfx*) ∈ ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*R*⟩*list-rel*
  $\bigwedge$*Re R*. (*lasso-ext*, *lasso-ext*)
    ∈ ⟨*R*⟩*list-rel* → *R* → ⟨*R*⟩*list-rel* → *Re* → ⟨*Re*,*R*⟩*lasso-rel-ext*
  $\bigwedge$*Re R*. (*lasso-reach-update*, *lasso-reach-update*)
    ∈ (⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*) → ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*Re*,*R*⟩*lasso-rel-ext*
  $\bigwedge$*Re R*. (*lasso-va-update*, *lasso-va-update*)
    ∈ (*R*→*R*) → ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*Re*,*R*⟩*lasso-rel-ext*
  $\bigwedge$*Re R*. (*lasso-cysfx-update*, *lasso-cysfx-update*)
    ∈ (⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*) → ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*Re*,*R*⟩*lasso-rel-ext*
  $\bigwedge$*Re R*. (*lasso.more-update*, *lasso.more-update*)
    ∈ (*Re*→*Re*) → ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*Re*,*R*⟩*lasso-rel-ext*
  unfolding *lasso-rel-ext-def*
  by (*auto dest*: *fun-relD*)

lemma *lasso-param2*[*param*, *autoref-rules*]:
  $\bigwedge$*Re R*. (*lasso-v0*, *lasso-v0*) ∈ ⟨*Re*,*R*⟩*lasso-rel-ext* → *R*
  $\bigwedge$*Re R*. (*lasso-cycle*, *lasso-cycle*) ∈ ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*R*⟩*list-rel*
  $\bigwedge$*Re R*. (*map-lasso*, *map-lasso*)
    ∈ (*R*→*R'*) → ⟨*Re*,*R*⟩*lasso-rel-ext* → ⟨*unit-rel*,*R'*⟩*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*: ⟦(*l'*,*l*)∈⟨*R*⟩*list-rel* $\times_r$ ⟨*R*⟩*list-rel*; *snd l* ≠ []⟧
  ⟹ (*lasso-of-prpl l'*, *lasso-of-prpl l*) ∈ ⟨*unit-rel*,*R*⟩*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* ≠ [])
  assumes $(l',l) \in \langle R \rangle$*list-rel* $\times_r$ $\langle R \rangle$*list-rel*
  shows (*lasso-of-prpl l'*,
    (*OP lasso-of-prpl*
      ::: $\langle R \rangle$*list-rel* $\times_r$ $\langle R \rangle$*list-rel* → ⟨*unit-rel*,*R*⟩*lasso-rel-ext*)$*l*)
    ∈ ⟨*unit-rel*,*R*⟩*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*:
  lasso-run-rel R ≡ *br run-of-lasso* (λ-. *True*) *O* (*nat-rel* → *R*)
lemma *lasso-run-rel-def*:
  ⟨*R*⟩*lasso-run-rel* = *br run-of-lasso* (λ-. *True*) *O* (*nat-rel* → *R*)
  unfolding *lasso-run-rel-def-internal relAPP-def* **by** *simp*

lemma *lasso-run-rel-sv*[*relator-props*]:
  single-valued R ⟹ *single-valued* (⟨*R*⟩*lasso-run-rel*)
  unfolding *lasso-run-rel-def*
  by *tagged-solver*

consts *i-run* :: *interface* ⇒ *interface*

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

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

lemma [*autoref-op-pat*]: (*o*) ≡ *op-map-run* **by** *simp*

lemma *map-lasso-run-refine*[*autoref-rules*]:
 shows (*map-lasso*,*op-map-run*) ∈ (*R*→*R'*) → ⟨*R*⟩*lasso-run-rel* → ⟨*R'*⟩*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'*)∈*R*→*R'* **and**
    (*l*, *r*) ∈ *br run-of-lasso* (λ-. *True*) *O* (*nat-rel* → *R*)
  then obtain *r'* **where** [*param*]: (*r'*,*r*)∈*nat-rel* → *R*
    and [*simp*]: *r'* = *run-of-lasso l*
    by (*auto simp*: *br-def*)

  have (*map-lasso f l*, *f o r'*) ∈ *br run-of-lasso* (λ-. *True*)

```
    by (simp add: br-def)
  also have (f o r′, f′ o r) ∈ nat-rel → R′ by parametricity
  finally (relcompI) show
    (map-lasso f l, f′ o r) ∈ br run-of-lasso (λ-. True) O (nat-rel → R′)
    .
qed

end
```

5 Simulation

```
theory Simulation
imports Automata
begin

  lemma finite-ImageI:
    assumes finite A
    assumes ⋀a. a∈A ⟹ finite (R‘‘{a})
    shows finite (R‘‘A)
  proof −
    note [[simproc add: finite-Collect]]
    have R‘‘A = ⋃{R‘‘{a} | a. a:A}
      by auto
    also have finite (⋃{R‘‘{a} | 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-α R ≡ λ x. SOME y. (x, y) ∈ R
  abbreviation (input) the-br-invar R ≡ λ x. x ∈ Domain R

  lemma the-br[simp]:
    assumes single-valued R
    shows br (the-br-α R) (the-br-invar R) = R
    unfolding build-rel-def the-br-α-def
    apply auto
    apply (metis someI-ex)
    apply (metis assms someI-ex single-valuedD)
    done

  lemma the-br-br[simp]:
```

$I\ x \Longrightarrow$ *the-br-α* $(br\ \alpha\ I)\ x = \alpha\ x$
the-br-invar $(br\ \alpha\ I) = I$
unfolding *the-br-α-def build-rel-def*[*abs-def*]
by *auto*

6.2 Relation between Runs

definition *run-rel* :: ('*a* × '*b*) *set* ⇒ ('*a word* × '*b word*) *set* **where**
run-rel R ≡ {(*ra*, *rb*). ∀ *i*. (*ra i*, *rb i*) ∈ *R*}

lemma *run-rel-converse*[*simp*]: (*ra*, *rb*) ∈ *run-rel* (R^{-1}) ⟷ (*rb*, *ra*) ∈ *run-rel* *R*
unfolding *run-rel-def* **by** *auto*

lemma *run-rel-single-valued*: *single-valued R*
⟹ (*ra*, *rb*) ∈ *run-rel R* ⟷ ((∀ *i*. *the-br-invar R* (*ra i*)) ∧ *rb* = *the-br-α R o* *ra*)
unfolding *run-rel-def the-br-α-def*
apply (*auto intro*!: *ext*)
apply (*metis single-valuedD someI-ex*)
apply (*metis DomainE someI-ex*)
done

6.3 Simulation

locale *simulation* =
a: *graph A* +
b: *graph B*
for *R* :: ('*a* × '*b*) *set*
and *A* :: ('*a*, -) *graph-rec-scheme*
and *B* :: ('*b*, -) *graph-rec-scheme*
\+
assumes *nodes-sim*: *a* ∈ *a.V* ⟹ (*a*, *b*) ∈ *R* ⟹ *b* ∈ *b.V*
assumes *init-sim*: *a0* ∈ *a.V0* ⟹ ∃ *b0*. *b0* ∈ *b.V0* ∧ (*a0*, *b0*) ∈ *R*
assumes *step-sim*: (*a*, *a'*) ∈ *a.E* ⟹ (*a*, *b*) ∈ *R* ⟹ ∃ *b'*. (*b*, *b'*) ∈ *b.E* ∧ (*a'*, *b'*) ∈ *R*
begin

lemma *simulation-this*: *simulation R A B* **by** *unfold-locales*

lemma *run-sim*:
assumes *arun*: *a.is-run ra*
obtains *rb* **where** *b.is-run rb*   (*ra*, *rb*) ∈ *run-rel R*
proof −
from *arun* **have** *ainit*: *ra 0* ∈ *a.V0*
and *astep*: ∀ *i*. (*ra i*, *ra* (*Suc i*)) ∈ *a.E*
using *a.run-V0 a.run-ipath ipathD* **by** *blast*+
from *init-sim* **obtain** *rb0* **where** *rel0*: (*ra 0*, *rb0*) ∈ *R* **and** *binit*: *rb0* ∈ *b.V0*
by (*auto intro*: *ainit*)

```
define rb
  where rb = rec-nat rb0 (λi rbi. SOME rbsi. (rbi, rbsi) ∈ b.E ∧ (ra (Suc
i), rbsi) ∈ R)

have [simp]:
  rb 0 = rb0
  ⋀i. rb (Suc i) = (SOME rbsi. (rb i, rbsi) ∈ b.E ∧ (ra (Suc i), rbsi) ∈ R)
  unfolding rb-def by auto

{
  fix i
  have (rb i, rb (Suc i)) ∈ b.E ∧ (ra (Suc i), rb (Suc i)) ∈ R
  proof (induction i)
    case 0
    from step-sim astep rel0 obtain rb1 where (rb 0, rb1) ∈ b.E and (ra 1,
rb1) ∈ R
      by fastforce
    thus ?case by (auto intro!: someI)
  next
    case (Suc i)
    with step-sim astep obtain rbss where (rb (Suc i), rbss) ∈ b.E and
      (ra (Suc (Suc i)), rbss) ∈ 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) ∈ 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) ∈ R
  assumes b ∉ Domain b.E
  shows a ∉ Domain a.E
  using assms
  by (auto dest: step-sim)

lemma run-Domain: a.is-run r ⟹ r i ∈ Domain R
  by (erule run-sim) (auto simp: run-rel-def)

lemma br-run-sim:
  assumes R = br α I
```

assumes *a.is-run r*
shows *b.is-run* (α *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*: $a \in a.E^* \text{ `` } a.V0 \Longrightarrow \exists\ b.\ (a,\ b) \in R \wedge b \in b.E^* \text{ `` } b.V0$
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 $\bigwedge b.\ b \in b.E^* \text{ `` } b.V0 \Longrightarrow$ *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 × '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 α I
    shows b.is-run rb ⟷ (∃ra. rb=α o ra ∧ 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 γ I)⁻¹
    shows a.is-run ra ⟷ (∃rb. ra=γ o rb ∧ 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 α
    assumes E: α o r1 = α o r2
    shows r1 = r2
  proof (rule ext)
    fix i
    from E have α (r1 i) = α (r2 i)
      by (simp add: comp-def) metis
    with ‹inj α› show r1 i = r2 i
      by (auto dest: injD)
  qed

  lemma br-inj-run-conv:
    assumes INJ: inj α
    assumes [simp]: R = br α I
```

```
    shows b.is-run (α o ra) ⟷ a.is-run ra
    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
      ⟷ (∃ ra. rb=the-br-α R o ra ∧ 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) ∈ R
    shows a ∈ Domain a.E ⟷ b ∈ 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 *lbisimulation-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 *lbisimulation-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* ≡ λ*a b*. (*a*,*b*)∈*s*
definition *pred-of-succ s* ≡ λ*a b*. *b*∈*s a*

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

definition *succ-of-pred s* ≡ λ*a*. {*b*. *s a b*}

lemma *rps-expand*[*simp*]:
(*a*,*b*)∈*rel-of-pred p* ⟷ *p a b*
(*a*,*b*)∈*rel-of-succ s* ⟷ *b* ∈ *s a*

pred-of-rel r a b ⟷ (*a*,*b*)∈*r*
pred-of-succ s a b ⟷ *b* ∈ *s a*

b∈*succ-of-rel r a* ⟷ (*a*,*b*)∈*r*
b∈*succ-of-pred p a* ⟷ *p a b*
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*
by *auto*

lemma *rps-conv*[*simp*]:
rel-of-pred (*pred-of-rel r*) = *r*
rel-of-pred (*pred-of-succ s*) = *rel-of-succ s*

rel-of-succ (*succ-of-rel r*) = *r*
rel-of-succ (*succ-of-pred p*) = *rel-of-pred p*

pred-of-rel (*rel-of-pred p*) = *p*
pred-of-rel (*rel-of-succ s*) = *pred-of-succ s*

pred-of-succ (*succ-of-pred p*) = *p*
pred-of-succ (*succ-of-rel r*) = *pred-of-rel r*

succ-of-rel (*rel-of-succ s*) = *s*
succ-of-rel (*rel-of-pred p*) = *succ-of-pred p*

succ-of-pred (*pred-of-succ s*) = *s*
succ-of-pred (*pred-of-rel r*) = *succ-of-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*
by *auto*

definition *m2r-rel* :: (*'a* × *'a option*) *set* ⇒ *'a option rel*
where *m2r-rel r* ≡ {(*Some a*,*b*)|*a b*. (*a*,*b*)∈*r*}

definition *m2r-pred* :: (*'a* ⇒ *'a option* ⇒ *bool*) ⇒ *'a option* ⇒ *'a option* ⇒ *bool*
where *m2r-pred p* ≡ λ*None* ⇒ λ-. *False* | *Some a* ⇒ *p a*

definition *m2r-succ* :: (*'a* ⇒ *'a option set*) ⇒ *'a option* ⇒ *'a option set*
where *m2r-succ s* ≡ λ*None* ⇒ {} | *Some a* ⇒ *s a*

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

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

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

  $m2r\text{-}succ\ (succ\text{-}of\text{-}pred\ p) = succ\text{-}of\text{-}pred\ (m2r\text{-}pred\ p)$
  $m2r\text{-}succ\ (succ\text{-}of\text{-}rel\ r) = succ\text{-}of\text{-}rel\ (m2r\text{-}rel\ r)$
  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 $rel\text{-}of\text{-}enex\ enex \equiv let\ (en,\ ex) = enex\ in\ \{(s,\ ex\ a\ s)\ |s\ a.\ a \in en\ s\}$
definition $pred\text{-}of\text{-}enex\ enex \equiv \lambda s\ s'.\ let\ (en,ex) = enex\ in\ \exists\, a{\in}en\ s.\ s'{=}ex\ a\ s$
definition $succ\text{-}of\text{-}enex\ enex \equiv \lambda s.\ let\ (en,ex) = enex\ in\ \{s'.\ \exists\, a{\in}en\ s.\ s'{=}ex\ a\ s\}$

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

lemma *x-of-enex-conv*[*simp*]:
  $rel\text{-}of\text{-}pred\ (pred\text{-}of\text{-}enex\ enex) = rel\text{-}of\text{-}enex\ enex$
  $rel\text{-}of\text{-}succ\ (succ\text{-}of\text{-}enex\ enex) = rel\text{-}of\text{-}enex\ enex$
  $pred\text{-}of\text{-}rel\ (rel\text{-}of\text{-}enex\ enex) = pred\text{-}of\text{-}enex\ enex$
  $pred\text{-}of\text{-}succ\ (succ\text{-}of\text{-}enex\ enex) = pred\text{-}of\text{-}enex\ enex$
  $succ\text{-}of\text{-}rel\ (rel\text{-}of\text{-}enex\ enex) = succ\text{-}of\text{-}enex\ enex$
  $succ\text{-}of\text{-}pred\ (pred\text{-}of\text{-}enex\ enex) = succ\text{-}of\text{-}enex\ enex$
  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 ⇒ 'v digraph ⇒ 'v digraph
    where stutter-extend-edges V E ≡ E ∪ {(v, v) |v. v ∈ V ∧ v ∉ Domain E}

  lemma stutter-extend-edgesI-edge:
    assumes (u, v) ∈ E
    shows (u, v) ∈ stutter-extend-edges V E
    using assms unfolding stutter-extend-edges-def by auto
  lemma stutter-extend-edgesI-stutter:
    assumes v ∈ V    v ∉ Domain E
    shows (v, v) ∈ stutter-extend-edges V E
    using assms unfolding stutter-extend-edges-def by auto
  lemma stutter-extend-edgesE:
    assumes (u, v) ∈ stutter-extend-edges V E
    obtains (edge) (u, v) ∈ E | (stutter)    u ∈ V    u ∉ Domain E    u = v
    using assms unfolding stutter-extend-edges-def by auto

  lemma stutter-extend-wf: E ⊆ V × V ⟹ stutter-extend-edges V E ⊆ V × 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 ⊆ Domain (stutter-extend-edges V E)
    unfolding stutter-extend-edges-def by auto

  definition stutter-extend :: ('v, -) graph-rec-scheme ⇒ ('v, -) graph-rec-scheme
    where stutter-extend G ≡
    ⦇
      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 ≡ λ*None* ⇒ *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* (λ*succs*. {(*u*,*v*). *v*∈*succs u*}) (λ-. *True*)

lemma *slg-rel-def*: ⟨*R*⟩*slg-rel* =
(*R* → ⟨*R*⟩*list-set-rel*) *O br* (λ*succs*. {(*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* $::_i$ $\langle I\rangle_i$*i-slg* $\rightarrow_i$ *I* $\rightarrow_i$ $\langle I\rangle_i$*i-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*]:
(λ*succs 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* ≡ (λ*u*. {*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* $\rightarrow_i$ $\langle I\rangle_i$*i-set*) $\rightarrow_i$ $\langle I\rangle_i$*i-slg* **by** *simp*
lemma [*autoref-itype*]: *succ-of-E* $::_i$ $\langle I\rangle_i$*i-slg* $\rightarrow_i$ *I* $\rightarrow_i$ $\langle I\rangle_i$*i-set* **by** *simp*

end

lemma *E-of-succ-refine*[*autoref-rules*]:
$(\lambda x.\ x,\ E\text{-}of\text{-}succ) \in (R \to \langle R\rangle list\text{-}set\text{-}rel) \to \langle R\rangle slg\text{-}rel$
$(\lambda x.\ x,\ succ\text{-}of\text{-}E) \in \langle R\rangle slg\text{-}rel \to (R \to \langle R\rangle list\text{-}set\text{-}rel)$
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* :: $'v\ set \Rightarrow 'v\ set \Rightarrow ('v \times 'v)\ set \Rightarrow ('v \times 'v)\ set$
where [*simp*]: *op-graph-restrict Vl Vr E* $\equiv E \cap Vl \times Vr$

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

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

lemma [*autoref-op-pat*]:
$E \cap (Vl \times Vr) \equiv$ *op-graph-restrict Vl Vr E*
$E \cap (Vl \times UNIV) \equiv$ *op-graph-restrict-left Vl E*
$E \cap (UNIV \times Vr) \equiv$ *op-graph-restrict-right Vr E*
by *simp-all*

lemma *graph-restrict-aimpl*: *op-graph-restrict Vl Vr E* =
E-of-succ ($\lambda v.$ *if* $v \in Vl$ *then* $\{x \in E``\{v\}.\ x \in Vr\}$ *else* $\{\}$)
by (*auto simp*: *E-of-succ-def succ-of-E-def split*: *if-split-asm*)
lemma *graph-restrict-left-aimpl*: *op-graph-restrict-left Vl E* =
E-of-succ ($\lambda v.$ *if* $v \in Vl$ *then* $E``\{v\}$ *else* $\{\}$)
by (*auto simp*: *E-of-succ-def succ-of-E-def split*: *if-split-asm*)
lemma *graph-restrict-right-aimpl*: *op-graph-restrict-right Vr E* =
E-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-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
assumes [*autoref-rules*]: $(meml, (\in)) \in R \to \langle R\rangle Rsl \to bool\text{-}rel$
assumes [*autoref-rules*]: $(memr, (\in)) \in R \to \langle R\rangle Rsr \to bool\text{-}rel$
shows $(?c,\ op\text{-}graph\text{-}restrict) \in \langle R\rangle Rsl \to \langle R\rangle Rsr \to \langle R\rangle slg\text{-}rel \to \langle R\rangle slg\text{-}rel$
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*, (∈)) ∈ *R* → ⟨*R*⟩*Rsl* → *bool-rel*
shows (*?c*, *op-graph-restrict-left*) ∈ ⟨*R*⟩*Rsl* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*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*, (∈)) ∈ *R* → ⟨*R*⟩*Rsr* → *bool-rel*
shows (*?c*, *op-graph-restrict-right*) ∈ ⟨*R*⟩*Rsr* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*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*, λ(*E*::*nat digraph*) *x*. *E*‘‘{*x*}) ∈ *?R*
apply (*autoref* (*keep-goal*))
done

lemma *graph-minus-aimpl*:
fixes *E1 E2* :: *'a rel*
shows *E1*−*E2* = *E-of-succ* (λ*x*. *E1*‘‘{*x*} − *E2*‘‘{*x*})
by (*auto simp*: *E-of-succ-def*)

schematic-goal *graph-minus-impl-aux*:
fixes *R* :: (*'vi*×*'v*) *set*
assumes [*autoref-rules*]: (*eq*,(=))∈*R*→*R*→*bool-rel*
shows (*?c*, (−)) ∈ ⟨*R*⟩*slg-rel* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*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-of-succ* (λ*u*. {*v*∈*E1*``{*u*}. (*u*,*v*)∉*E2*})
by (*auto simp*: *E-of-succ-def*)

schematic-goal *graph-minus-set-impl-aux*:
fixes *R* :: (*'vi*×*'v*) *set*
assumes [*autoref-rules*]: (*eq*,(=))∈*R*→*R*→*bool-rel*
assumes [*autoref-rules*]: (*mem*,(∈)) ∈ $R \times_r R$ → ⟨$R \times_r R$⟩*Rs* → *bool-rel*
shows (*?c*, (−)) ∈ ⟨*R*⟩*slg-rel* → ⟨$R{\times_r}R$⟩*Rs* → ⟨*R*⟩*slg-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-g-ext :: *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-frg* ≡ $\langle$*i-unit*$\rangle_i$*i-g-ext*

context begin interpretation *autoref-syn* **.**

lemma *g-type*[*autoref-itype*]:
g-V $::_i$ $\langle Ie,I\rangle_i$*i-g-ext* $\to_i$ $\langle I\rangle_i$*i-set*
g-E $::_i$ $\langle Ie,I\rangle_i$*i-g-ext* $\to_i$ $\langle I\rangle_i$*i-slg*
g-V0 $::_i$ $\langle Ie,I\rangle_i$*i-g-ext* $\to_i$ $\langle I\rangle_i$*i-set*
graph-rec-ext
$::_i$ $\langle I\rangle_i$*i-set* $\to_i$ $\langle I\rangle_i$*i-slg* $\to_i$ $\langle I\rangle_i$*i-set* $\to_i$ *iE* $\to_i$ $\langle Ie,I\rangle_i$*i-g-ext*
by *simp-all*

end

7.2.2 Generic Implementation

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

definition *gen-g-impl-rel-ext-internal-def*: $\bigwedge$ *Rm Rv Re Rv0*. *gen-g-impl-rel-ext Rm Rv Re Rv0*
≡ { (*gen-g-impl-ext Vi Ei V0i mi*, *graph-rec-ext V E V0 m*)
| *Vi Ei V0i mi V E V0 m*.
(*Vi*,*V*)∈*Rv* ∧ (*Ei*,*E*)∈*Re* ∧ (*V0i*,*V0*)∈*Rv0* ∧ (*mi*,*m*)∈*Rm*
}

lemma *gen-g-impl-rel-ext-def*: $\bigwedge$*Rm Rv Re Rv0*. ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext*
≡ { (*gen-g-impl-ext Vi Ei V0i mi*, *graph-rec-ext V E V0 m*)
| *Vi Ei V0i mi V E V0 m*.
(*Vi*,*V*)∈*Rv* ∧ (*Ei*,*E*)∈*Re* ∧ (*V0i*,*V0*)∈*Rv0* ∧ (*mi*,*m*)∈*Rm*
}
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*. ⟦*single-valued Rv*; *single-valued Re*; *single-valued Rv0*; *single-valued Rm* ⟧ ⟹
single-valued (⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*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*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Rv*
$\bigwedge$*Rm Rv Re Rv0*. (*gi-E*,*g-E*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Re*
$\bigwedge$*Rm Rv Re Rv0*. (*gi-V0*,*g-V0*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Rv0*
$\bigwedge$*Rm Rv Re Rv0*. (*gen-g-impl-ext*, *graph-rec-ext*)
∈ *Rv* → *Re* → *Rv0* → *Rm* → ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*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*) *frgv-impl-scheme* =
(′*v list*, ′*v* ⇒ ′*v list*, ′*v list*, ′*m*) *gen-g-impl-scheme*

definition *frgv-impl-rel-ext-internal-def*:
frgv-impl-rel-ext Rm Rv
≡ ⟨*Rm*,⟨*Rv*⟩*list-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*⟩*gen-g-impl-rel-ext*

lemma *frgv-impl-rel-ext-def*: ⟨*Rm*,*Rv*⟩*frgv-impl-rel-ext*
≡ ⟨*Rm*,⟨*Rv*⟩*list-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*⟩*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*]:
⟦*single-valued Rv*; *Range Rv* = *UNIV*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*frgv-impl-rel-ext*)
unfolding *frgv-impl-rel-ext-def* **by** *tagged-solver*

lemmas [*param*, *autoref-rules*] = *gen-g-refine*[**where**
Rv = ⟨*Rv*⟩*list-set-rel* **and** *Re* = ⟨*Rv*⟩*slg-rel* **and** *?Rv0.0* = ⟨*Rv*⟩*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* ⇒ *bool*, ′*v* ⇒ ′*v list*, ′*v list*, ′*m*) *gen-g-impl-scheme*

definition *g-impl-rel-ext-internal-def*:
g-impl-rel-ext Rm Rv
≡ ⟨*Rm*,⟨*Rv*⟩*fun-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*⟩*gen-g-impl-rel-ext*

lemma *g-impl-rel-ext-def*: ⟨*Rm*,*Rv*⟩*g-impl-rel-ext*
≡ ⟨*Rm*,⟨*Rv*⟩*fun-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*⟩*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*]:
⟦*single-valued Rv*; *Range Rv* = *UNIV*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*g-impl-rel-ext*)
unfolding *g-impl-rel-ext-def* **by** *tagged-solver*

lemmas [*param*, *autoref-rules*] = *gen-g-refine*[**where**
Rv = ⟨*Rv*⟩*fun-set-rel*
and *Re* = ⟨*Rv*⟩*slg-rel*
and *?Rv0.0* = ⟨*Rv*⟩*list-set-rel*
for *Rv*, *folded g-impl-rel-ext-def*]

lemma [*autoref-rules*]: (*gi-V-update*, *g-V-update*) ∈ (⟨*Rv*⟩*fun-set-rel* → ⟨*Rv*⟩*fun-set-rel*)
→
⟨*Rm*, *Rv*⟩*g-impl-rel-ext* → ⟨*Rm*, *Rv*⟩*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*) ∈ (⟨*Rv*⟩*slg-rel* → ⟨*Rv*⟩*slg-rel*) →
⟨*Rm*, *Rv*⟩*g-impl-rel-ext* → ⟨*Rm*, *Rv*⟩*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*) ∈ (⟨*Rv*⟩*list-set-rel* → ⟨*Rv*⟩*list-set-rel*)
→
⟨*Rm*, *Rv*⟩*g-impl-rel-ext* → ⟨*Rm*, *Rv*⟩*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 (⟨*Rv*⟩*Rvs* → ⟨*Rv*⟩*Res* → ⟨*Rv*⟩*Rv0s* → *Rm* → ⟨*Rm*,*Rv*⟩*Rg*)
by *simp*

schematic-goal (*?c*::*?′c*, λ*G x*. *g-E G* ‘‘ {*x*})∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*,λ*V0 E*.
(| *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0* |))
∈⟨*R*⟩*list-set-rel* → ⟨*R*⟩*slg-rel* → ⟨*unit-rel*,*R*⟩*g-impl-rel-ext*
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*,λ*V V0 E*.
(| *g-V* = *V*, *g-E* = *E*, *g-V0* = *V0* |))
∈⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*slg-rel* → ⟨*unit-rel*,*R*⟩*frgv-impl-rel-ext*
apply (*autoref* (*keep-goal*))
done

7.2.5 Renaming

definition *the-inv-into-map V f x*
= (*if x* ∈ *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* ⟷ *x* ∉ *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* ⟷ *x* ∈ *f*‘*V* ∧ *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 ⟹ *the-inv-into-map V f x* = *Some y* ⟷ *y*∈*V* ∧ *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 (λ*x m*. *RETURN* (*m*(*f x* ↦ *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* ≤ *SPEC* (λ*r*. *r* = *the-inv-into-map V f*)

unfolding *the-inv-into-map-impl-def*
apply (*refine-rcg*
FOREACH-rule[**where** $I=\lambda it\ m.\ m=$*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* × *′vt*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode Rv′ eq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*(*′vti*) (*def-size*)
assumes [*autoref-rules*]: (*Vi*, *V*)∈⟨*Rv*⟩*list-set-rel*
assumes [*autoref-rules*]: (*fi*,*f*)∈*Rv*→*Rv′*
shows (*RETURN ?c*, *the-inv-into-map-impl V f*) ∈ ⟨⟨*Rv′*,*Rv*⟩*ahm-rel bhc*⟩*nres-rel*
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* × *′vt*) *set*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode Rv′ eq bhc*)
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*′vti*) *def-size*)
assumes *INJ*: *SIDE-PRECOND* (*inj-on f V*)
assumes *V*: (*Vi*, *V*)∈⟨*Rv*⟩*list-set-rel*
assumes *F*: (*fi*,*f*)∈*Rv*→*Rv′*
shows (*the-inv-into-map-code eq bhc def-size Vi fi*,
(*OP the-inv-into-map*
::: ⟨*Rv*⟩*list-set-rel* → (*Rv*→*Rv′*) → ⟨*Rv′*, *Rv*⟩*Impl-Array-Hash-Map.ahm-rel bhc*)

```
    $V$f) ∈ ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel bhc
proof 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 Suc s);
  RETURN (the-inv-into-map s Suc) }) ∈ ?R
  apply (autoref (keep-goal))
  done


definition fr-rename-ext-aimpl ecnv f G ≡ do {
    ASSERT (inj-on f (g-V G));
    ASSERT (inj-on f (g-V0 G));
    let fi-map = the-inv-into-map (g-V G) f;
    e ← ecnv fi-map G;
    RETURN (|
      g-V = f`(g-V G),
      g-E = (E-of-succ (λv. case fi-map v of
          Some u ⇒ f ` (succ-of-E (g-E G) u) | None ⇒ {})),
      g-V0 = (f`g-V0 G),
      ... = e
    |)
  }

context g-rename-precond begin

  definition fi-map x = (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) ⟷ u∈f`V ∧ fi u = v
    unfolding fi-map-def by (auto split: if-split-asm)
```

```
  lemma fi-map-None: (fi-map u = None) ⟷ u∉f‘V
    unfolding fi-map-def by (auto split: if-split-asm)

  lemma rename-E-aimpl-alt: rename-E f E = E-of-succ (λv. case fi-map v of
    Some u ⇒ f ‘ (succ-of-E E u) | None ⇒ {})
    unfolding E-of-succ-def succ-of-E-def
    using E-ss
    by (force
      simp: fi-f f-fi fi-map-Some fi-map-None
      split: if-split-asm option.splits)


  lemma frv-rename-ext-aimpl-alt:
    assumes ECNV: ecnv' fi-map G ≤ SPEC (λr. r = ecnv G)
    shows fr-rename-ext-aimpl ecnv' f G
      ≤ SPEC (λr. r = fr-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 × 'vt) set
  assumes [autoref-rules]: (eq, (=)) ∈ Rv' → Rv' → bool-rel
  assumes [autoref-ga-rules]: is-bounded-hashcode Rv' eq bhc
  assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE('vti) def-size
  shows (?c,fr-rename-ext-aimpl) ∈
    ((⟨Rv',Rv⟩ahm-rel bhc) → ⟨Re,Rv⟩frgv-impl-rel-ext → ⟨Re'⟩nres-rel) →
    (Rv→Rv') →
    ⟨Re,Rv⟩frgv-impl-rel-ext →
    ⟨⟨Re',Rv'⟩frgv-impl-rel-ext⟩nres-rel
  unfolding fr-rename-ext-aimpl-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*) ∈ *Id* → ⟨*Id*⟩*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* ⇒ *interface* ⇒ *interface*

abbreviation *i-igbg Ie Iv* ≡ $\langle\langle Ie,Iv\rangle_i$*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* $\to_i$ *i-nat*
  igbg-acc $::_i$ *i-igbg Ie Iv* $\to_i$ *Iv* $\to_i$ ⟨*i-nat*⟩$_i$*i-set*
  igb-graph-rec-ext
    $::_i$ *i-nat* $\to_i$ (*Iv* $\to_i$ ⟨*i-nat*⟩$_i$*i-set*) $\to_i$ *Ie* $\to_i$ ⟨*Ie*,*Iv*⟩$_i$*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 ≡ { (
  ⦇ *igbgi-num-acc* = *num-acci*, *igbgi-acc* = *acci*, ...=*mi* ⦈,
  ⦇ *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc*, ...=*m* ⦈)
  | *num-acci acci mi num-acc acc m*.
    (*num-acci*,*num-acc*)∈*nat-rel*

∧ (*acci*,*acc*)∈*Racc*
∧ (*mi*,*m*)∈*Rm*
}

lemma *gen-igbg-impl-rel-eext-def*:
⟨*Rm*,*Racc*⟩*gen-igbg-impl-rel-eext* = { (
⦇ *igbgi-num-acc* = *num-acci*, *igbgi-acc* = *acci*, . . .=*mi* ⦈,
⦇ *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc*, . . .=*m* ⦈)
| *num-acci acci mi num-acc acc m*.
(*num-acci*,*num-acc*)∈*nat-rel*
∧ (*acci*,*acc*)∈*Racc*
∧ (*mi*,*m*)∈*Rm*
}
unfolding *gen-igbg-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-igbg-impl-rel-sv*[*relator-props*]:
⟦*single-valued Racc*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Racc*⟩*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*
:: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (-×(-,-)*igb-graph-rec-scheme*) *set*
where *gen-igbg-impl-rel-ext Rm Racc*
≡ ⟨⟨*Rm*,*Racc*⟩*gen-igbg-impl-rel-eext*⟩*gen-g-impl-rel-ext*

lemma *gen-igbg-refine*:
fixes *Rv Re Rv0 Racc*
assumes *TERM* (*Rv*,*Re*,*Rv0*)
assumes *TERM* (*Racc*)
shows
(*igbgi-num-acc*,*igbg-num-acc*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-igbg-impl-rel-ext Rm Racc* → *nat-rel*
(*igbgi-acc*,*igbg-acc*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-igbg-impl-rel-ext Rm Racc* → *Racc*
(*gen-igbg-impl-ext*, *igb-graph-rec-ext*)
∈ *nat-rel* → *Racc* → *Rm* → ⟨*Rm*,*Racc*⟩*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*:
igbg-impl-rel-eext Rm Rv ≡ ⟨*Rm*, *Rv* → ⟨*nat-rel*⟩*bs-set-rel*⟩*gen-igbg-impl-rel-eext*

lemma *igbg-impl-rel-eext-def*:
$\langle Rm,Rv\rangle$*igbg-impl-rel-eext* ≡ $\langle Rm,\ Rv \rightarrow \langle$*nat-rel*$\rangle$*bs-set-rel*$\rangle$*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*]:
⟦*Range Rv* = *UNIV*; *single-valued Rm*⟧
⟹ *single-valued* ($\langle Rm,Rv\rangle$*igbg-impl-rel-eext*)
unfolding *igbg-impl-rel-eext-def* **by** *tagged-solver*

lemma *g-tag*: *TERM* ($\langle Rv\rangle$*fun-set-rel*,$\langle Rv\rangle$*slg-rel*,$\langle Rv\rangle$*list-set-rel*) **.**
lemma *frgv-tag*: *TERM* ($\langle Rv\rangle$*list-set-rel*,$\langle Rv\rangle$*slg-rel*,$\langle Rv\rangle$*list-set-rel*) **.**
lemma *igbg-bs-tag*: *TERM* ($Rv \rightarrow \langle$*nat-rel*$\rangle$*bs-set-rel*) **.**

abbreviation *igbgv-impl-rel-ext Rm Rv*
≡ $\langle\langle Rm,\ Rv\rangle$*igbg-impl-rel-eext*, $Rv\rangle$*frgv-impl-rel-ext*

abbreviation *igbg-impl-rel-ext Rm Rv*
≡ $\langle\langle Rm,\ Rv\rangle$*igbg-impl-rel-eext*, $Rv\rangle$*g-impl-rel-ext*

type-synonym ('*v*,'*m*) *igbgv-impl-scheme* =
('*v*, ⦇ *igbgi-num-acc*::*nat*, *igbgi-acc*::'*v*⇒*integer*, . . .::'*m* ⦈)
frgv-impl-scheme
type-synonym ('*v*,'*m*) *igbg-impl-scheme* =
('*v*, ⦇ *igbgi-num-acc*::*nat*, *igbgi-acc*::'*v*⇒*integer*, . . .::'*m* ⦈)
g-impl-scheme

context fixes *Rv* :: ('*vi*×'*v*) *set* **begin**
lemmas [*autoref-rules*] = *gen-igbg-refine*[
OF frgv-tag[*of Rv*] *igbg-bs-tag*[*of Rv*],
folded frgv-impl-rel-ext-def igbg-impl-rel-eext-def]

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

schematic-goal (*?c*::*?'c*,
λ*G x*. *if igbg-num-acc G* = *0* ∧ *1*∈*igbg-acc G x then* (*g-E G* ‘‘ {*x*}) *else* {}
)∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*,

λV0 E num-acc acc.
(| *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*, *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc* |)
)∈⟨*R*⟩*list-set-rel* → ⟨*R*⟩*slg-rel* → *nat-rel* → (*R* → ⟨*nat-rel*⟩*bs-set-rel*)
→ *igbg-impl-rel-ext unit-rel R*
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*,
λV0 E num-acc acc.
(| *g-V* = {}, *g-E* = *E*, *g-V0* = *V0*, *igbg-num-acc* = *num-acc*, *igbg-acc* = *acc* |)
)∈⟨*R*⟩*list-set-rel* → ⟨*R*⟩*slg-rel* → *nat-rel* → (*R* → ⟨*nat-rel*⟩*bs-set-rel*)
→ *igbgv-impl-rel-ext unit-rel R*
apply (*autoref* (*keep-goal*))
done

8.2 Indexed Generalized Buchi Automata

consts
i-igba-eext :: *interface* ⇒ *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-igba Ie Iv Il*
≡ $\langle\langle\langle Ie,Iv,Il\rangle_i$ *i-igba-eext*,*Iv*$\rangle_i$ *i-igbg-eext*,*Iv*$\rangle_i$ *i-g-ext*
context begin interpretation *autoref-syn* **.**

lemma *igba-type*[*autoref-itype*]:
igba-L $::_i$ *i-igba Ie Iv Il* $\rightarrow_i$ (*Iv* $\rightarrow_i$ *Il* $\rightarrow_i$ *i-bool*)
igba-rec-ext $::_i$ (*Iv* $\rightarrow_i$ *Il* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *Ie* $\rightarrow_i$ $\langle Ie,Iv,Il\rangle_i$*i-igba-eext*
by *simp-all*
end

record (*'vi*,*'ei*,*'v0i*,*'acci*,*'Li*) *gen-igba-impl* =
(*'vi*,*'ei*,*'v0i*,*'acci*)*gen-igbg-impl* +
igbai-L :: *'Li*

definition *gen-igba-impl-rel-eext-def-internal*:
gen-igba-impl-rel-eext Rm Rl ≡ { (
(| *igbai-L* = *Li*, ...=*mi* |),
(| *igba-L* = *L*, ...=*m* |))
| *Li mi L m.*
(*Li*,*L*)∈*Rl*
∧ (*mi*,*m*)∈*Rm*
}

lemma *gen-igba-impl-rel-eext-def*:
⟨*Rm*,*Rl*⟩*gen-igba-impl-rel-eext* = { (
(| *igbai-L* = *Li*, ...=*mi* |),
(| *igba-L* = *L*, ...=*m* |))

| *Li mi L m.*
  (Li,L)∈*Rl*
∧ *(mi,m)*∈*Rm*
}
unfolding *gen-igba-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-igba-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rl*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rl*⟩*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*
:: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × (*'a*,*'b*,*'c*) *igba-rec-scheme*) *set*
where *gen-igba-impl-rel-ext Rm Rl*
≡ *gen-igbg-impl-rel-ext* (⟨*Rm*,*Rl*⟩*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
(*igbai-L*,*igba-L*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-igba-impl-rel-ext Rm Rl Racc* → *Rl*
(*gen-igba-impl-ext*, *igba-rec-ext*)
∈ *Rl* → *Rm* → ⟨*Rm*,*Rl*⟩*gen-igba-impl-rel-eext*
unfolding *gen-igba-impl-rel-eext-def gen-igbg-impl-rel-eext-def*
gen-g-impl-rel-ext-def
by *auto*

8.2.1 Implementation as function

definition *igba-impl-rel-eext-internal-def*:
igba-impl-rel-eext Rm Rv Rl ≡ ⟨*Rm*, *Rv* → *Rl* → *bool-rel*⟩*gen-igba-impl-rel-eext*

lemma *igba-impl-rel-eext-def*:
⟨*Rm*,*Rv*,*Rl*⟩*igba-impl-rel-eext* ≡ ⟨*Rm*, *Rv* → *Rl* → *bool-rel*⟩*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*]:
⟦*Range Rv* = *UNIV*; *single-valued Rm*; *Range Rl* = *UNIV*⟧

$\Longrightarrow$ *single-valued* (⟨*Rm*,*Rv*,*Rl*⟩*igba-impl-rel-eext*)
unfolding *igba-impl-rel-eext-def* **by** *tagged-solver*

lemma *igba-f-tag*: *TERM* (*Rv* → *Rl* → *bool-rel*) **.**

abbreviation *igbav-impl-rel-ext Rm Rv Rl*
≡ *igbgv-impl-rel-ext* (⟨*Rm*, *Rv*, *Rl*⟩*igba-impl-rel-eext*) *Rv*

abbreviation *igba-impl-rel-ext Rm Rv Rl*
≡ *igbg-impl-rel-ext* (⟨*Rm*, *Rv*, *Rl*⟩*igba-impl-rel-eext*) *Rv*

type-synonym (*'v*,*'l*,*'m*) *igbav-impl-scheme* =
(*'v*, ⦇ *igbai-L* :: *'v* ⇒ *'l* ⇒ *bool* , . . .::*'m* ⦈)
igbgv-impl-scheme

type-synonym (*'v*,*'l*,*'m*) *igba-impl-scheme* =
(*'v*, ⦇ *igbai-L* :: *'v* ⇒ *'l* ⇒ *bool* , . . .::*'m* ⦈)
igbg-impl-scheme

context
fixes *Rv* :: (*'vi*×*'v*) *set*
fixes *Rl* :: (*'Li*×*'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*, λ*G x l*. *if igba-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*, λ*E* (*V0*::*'a set*) *num-acc acc L*.
⦇ *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*,
igbg-num-acc = *num-acc*, *igbg-acc* = *acc*, *igba-L* = *L* ⦈
)∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal

notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: (*'a*×*'a*) *set*]
shows (*?c*::*?'c*, *λE* (*V0*::*'a set*) *num-acc acc L*.
(| *g-V* = *V0*, *g-E* = *E*, *g-V0* = *V0*,
igbg-num-acc = *num-acc*, *igbg-acc* = *acc*, *igba-L* = *L* |)
)∈*?R*
apply (*autoref* (*keep-goal*))
done

8.3 Generalized Buchi Graphs

consts
i-gbg-eext :: *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-gbg Ie Iv* ≡ $\langle\langle Ie,Iv\rangle_i$ *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 ≡ { (
(| *gbgi-F* = *Fi*, . . .=*mi* |),
(| *gbg-F* = *F*, . . .=*m* |))
| *Fi mi F m*.
(*Fi*,*F*)∈*Rf*
∧ (*mi*,*m*)∈*Rm*
}

lemma *gen-gbg-impl-rel-eext-def*:
⟨*Rm*,*Rf*⟩*gen-gbg-impl-rel-eext* = { (
(| *gbgi-F* = *Fi*, . . .=*mi* |),
(| *gbg-F* = *F*, . . .=*m* |))
| *Fi mi F m*.
(*Fi*,*F*)∈*Rf*
∧ (*mi*,*m*)∈*Rm*
}
unfolding *gen-gbg-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-gbg-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rf*⟧
⟹ *single-valued* (⟨*Rm*,*Rf*⟩*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*
:: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × ('*q*,-) *gb-graph-rec-scheme*) *set*
where *gen-gbg-impl-rel-ext Rm Rf*
≡ ⟨⟨*Rm*,*Rf*⟩*gen-gbg-impl-rel-eext*⟩*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*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-gbg-impl-rel-ext Rm Rf* → *Rf*
(*gen-gbg-impl-ext*, *gb-graph-rec-ext*)
∈ *Rf* → *Rm* → ⟨*Rm*,*Rf*⟩*gen-gbg-impl-rel-eext*
unfolding *gen-gbg-impl-rel-eext-def gen-g-impl-rel-ext-def*
by *auto*

8.3.1 Implementation with list of lists

definition *gbg-impl-rel-eext-internal-def*:
gbg-impl-rel-eext Rm Rv
≡ ⟨*Rm*, ⟨⟨*Rv*⟩*list-set-rel*⟩*list-set-rel*⟩*gen-gbg-impl-rel-eext*

lemma *gbg-impl-rel-eext-def*:
⟨*Rm*,*Rv*⟩*gbg-impl-rel-eext*
≡ ⟨*Rm*, ⟨⟨*Rv*⟩*list-set-rel*⟩*list-set-rel*⟩*gen-gbg-impl-rel-eext*
unfolding *gbg-impl-rel-eext-internal-def relAPP-def* **by** *simp*

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

lemma [*relator-props*, *simp*]:
⟦*single-valued Rm*; *single-valued Rv*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*gbg-impl-rel-eext*)
unfolding *gbg-impl-rel-eext-def* **by** *tagged-solver*

lemma *gbg-ls-tag*: *TERM* (⟨⟨*Rv*⟩*list-set-rel*⟩*list-set-rel*) **.**

abbreviation *gbgv-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*gbg-impl-rel-eext*, *Rv*⟩*frgv-impl-rel-ext*

abbreviation *gbg-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*gbg-impl-rel-eext*, *Rv*⟩*g-impl-rel-ext*

context fixes *Rv* :: (*'vi*×*'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*,
λG x. if gbg-F G = {} *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*) *F*.
(| *g-V* = {}, *g-E* = *E*, *g-V0* = *V0*, *gbg-F* = *F* |))∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: (*'a*×*'a*) *set*]
shows (*?c*::*?'c*, *λE* (*V0*::*'a set*) *F*.
(| *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*, *gbg-F* = *insert* {} *F* |))∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal (*?c*::*?'c*, *it-to-sorted-list* (λ- -. *True*) {*1*,*2*::*nat*})∈*?R*
apply (*autoref* (*keep-goal*))
done

8.4 GBAs

consts
i-gba-eext :: *interface* ⇒ *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-gba Ie Iv Il*
≡ $\langle\langle\langle Ie,Iv,Il\rangle_i$ *i-gba-eext*,$Iv\rangle_i$ *i-gbg-eext*,$Iv\rangle_i$ *i-g-ext*
context begin interpretation *autoref-syn* **.**

lemma *gba-type*[*autoref-itype*]:
gba-L $::_i$ *i-gba Ie Iv Il* $\rightarrow_i$ (*Iv* $\rightarrow_i$ *Il* $\rightarrow_i$ *i-bool*)
gba-rec-ext $::_i$ (*Iv* $\rightarrow_i$ *Il* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *Ie* $\rightarrow_i$ $\langle Ie,Iv,Il\rangle_i$ *i-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-eext-def-internal*:
gen-gba-impl-rel-eext Rm Rl ≡ { (
⦇ *gbai-L* = *Li*, ...=*mi* ⦈,
⦇ *gba-L* = *L*, ...=*m* ⦈)
| *Li mi L m*.
(*Li*,*L*)∈*Rl*
∧ (*mi*,*m*)∈*Rm*
}

lemma *gen-gba-impl-rel-eext-def*:
⟨*Rm*,*Rl*⟩*gen-gba-impl-rel-eext* = { (
⦇ *gbai-L* = *Li*, ...=*mi* ⦈,
⦇ *gba-L* = *L*, ...=*m* ⦈)
| *Li mi L m*.
(*Li*,*L*)∈*Rl*
∧ (*mi*,*m*)∈*Rm*
}
unfolding *gen-gba-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-gba-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rl*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rl*⟩*gen-gba-impl-rel-eext*)
unfolding *gen-gba-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-gba-impl-rel-ext*
:: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × ('a,'b,'c) *gba-rec-scheme*) *set*
where *gen-gba-impl-rel-ext Rm Rl*
≡ *gen-gbg-impl-rel-ext* (⟨*Rm*,*Rl*⟩*gen-gba-impl-rel-eext*)

lemma *gen-gba-refine*:
fixes *Rv Re Rv0 Racc Rl*
assumes *TERM* (*Rv*,*Re*,*Rv0*)
assumes *TERM* (*Racc*)
assumes *TERM* (*Rl*)
shows
(*gbai-L*,*gba-L*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-gba-impl-rel-ext Rm Rl Racc* → *Rl*
(*gen-gba-impl-ext*, *gba-rec-ext*)

$\in$ *Rl* $\rightarrow$ *Rm* $\rightarrow$ $\langle$*Rm*,*Rl*$\rangle$*gen-gba-impl-rel-eext*
unfolding *gen-gba-impl-rel-eext-def gen-gbg-impl-rel-eext-def*
gen-g-impl-rel-ext-def
by *auto*

8.4.1 Implementation as function

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

lemma *gba-impl-rel-eext-def*:
$\langle$*Rm*,*Rv*,*Rl*$\rangle$*gba-impl-rel-eext* $\equiv$ $\langle$*Rm*, *Rv* $\rightarrow$ *Rl* $\rightarrow$ *bool-rel*$\rangle$*gen-gba-impl-rel-eext*
unfolding *gba-impl-rel-eext-internal-def relAPP-def* **by** *simp*

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

lemma [*relator-props*, *simp*]:
$\llbracket$*Range Rv* = *UNIV*; *single-valued Rm*; *Range Rl* = *UNIV*$\rrbracket$
$\Longrightarrow$ *single-valued* ($\langle$*Rm*,*Rv*,*Rl*$\rangle$*gba-impl-rel-eext*)
unfolding *gba-impl-rel-eext-def* **by** *tagged-solver*

lemma *gba-f-tag*: *TERM* (*Rv* $\rightarrow$ *Rl* $\rightarrow$ *bool-rel*) **.**

abbreviation *gbav-impl-rel-ext Rm Rv Rl*
$\equiv$ *gbgv-impl-rel-ext* ($\langle$*Rm*, *Rv*, *Rl*$\rangle$*gba-impl-rel-eext*) *Rv*

abbreviation *gba-impl-rel-ext Rm Rv Rl*
$\equiv$ *gbg-impl-rel-ext* ($\langle$*Rm*, *Rv*, *Rl*$\rangle$*gba-impl-rel-eext*) *Rv*

context
fixes *Rv* :: ('*vi*$\times$'*v*) *set*
fixes *Rl* :: ('*Li*$\times$'*l*) *set*
begin
lemmas [*autoref-rules*] = *gen-gba-refine*[
OF frgv-tag[*of Rv*] *gbg-ls-tag*[*of Rv*] *gba-f-tag*[*of Rv Rl*],
folded frgv-impl-rel-ext-def gbg-impl-rel-eext-def gba-impl-rel-eext-def]

lemmas [*autoref-rules*] = *gen-gba-refine*[
OF g-tag[*of Rv*] *gbg-ls-tag*[*of Rv*] *gba-f-tag*[*of Rv Rl*],
folded g-impl-rel-ext-def gbg-impl-rel-eext-def gba-impl-rel-eext-def]
end

thm *autoref-itype*

schematic-goal
(*?c*::*?'c*, λ*G x l*. *if gba-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*×*'a*) *set*]
  shows (*?c*::*?'c*, λ*E* (*V0*::*'a set*) *F L*.
  (| *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*,
    gbg-F = *F*, *gba-L* = *L* |)
)∈*?R*
  apply (*autoref* (*keep-goal*))
  done

schematic-goal
  notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: (*'a*×*'a*) *set*]
  shows (*?c*::*?'c*, λ*E* (*V0*::*'a set*) *F L*.
  (| *g-V* = *V0*, *g-E* = *E*, *g-V0* = *V0*,
    gbg-F = *F*, *gba-L* = *L* |)
)∈*?R*
  apply (*autoref* (*keep-goal*))
  done

8.5 Buchi Graphs

consts
  i-bg-eext :: *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-bg Ie Iv* ≡ $\langle\langle Ie,Iv\rangle_i$*i-bg-eext*,$Iv\rangle_i$*i-g-ext*

context begin interpretation *autoref-syn* **.**
lemma *bg-type*[*autoref-itype*]:
  bg-F $::_i$ *i-bg Ie Iv* $\rightarrow_i$ $\langle Iv\rangle_i$*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-bg-eext*
  by *simp-all*
end

record (*'vi*,*'ei*,*'v0i*,*'fi*) *gen-bg-impl* = (*'vi*,*'ei*,*'v0i*) *gen-g-impl* +
  bgi-F :: *'fi*

definition *gen-bg-impl-rel-eext-def-internal*:
  gen-bg-impl-rel-eext Rm Rf ≡ { (
  (| *bgi-F* = *Fi*, . . .=*mi* |),
  (| *bg-F* = *F*, . . .=*m* |))
  | *Fi mi F m*.
    (*Fi*,*F*)∈*Rf*
  ∧ (*mi*,*m*)∈*Rm*
  }

lemma *gen-bg-impl-rel-eext-def*:
  ⟨*Rm*,*Rf*⟩*gen-bg-impl-rel-eext* = { (
  (| *bgi-F* = *Fi*, . . .=*mi* |),
  (| *bg-F* = *F*, . . .=*m* |))
  | *Fi mi F m*.

(*Fi*,*F*)∈*Rf*
∧ (*mi*,*m*)∈*Rm*
}
unfolding *gen-bg-impl-rel-eext-def-internal relAPP-def* **by** *simp*

lemma *gen-bg-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rf*⟧
⟹ *single-valued* (⟨*Rm*,*Rf*⟩*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*
:: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × (′*q*,-) *b-graph-rec-scheme*) *set*
where *gen-bg-impl-rel-ext Rm Rf*
≡ ⟨⟨*Rm*,*Rf*⟩*gen-bg-impl-rel-eext*⟩*gen-g-impl-rel-ext*

lemma *gen-bg-refine*:
fixes *Rv Re Rv0 Rf*
assumes *TERM* (*Rv*,*Re*,*Rv0*)
assumes *TERM* (*Rf*)
shows
(*bgi-F*,*bg-F*)
∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-bg-impl-rel-ext Rm Rf* → *Rf*
(*gen-bg-impl-ext*, *b-graph-rec-ext*)
∈ *Rf* → *Rm* → ⟨*Rm*,*Rf*⟩*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*:
bg-impl-rel-eext Rm Rv
≡ ⟨*Rm*, ⟨*Rv*⟩*fun-set-rel*⟩*gen-bg-impl-rel-eext*

lemma *bg-impl-rel-eext-def*:
⟨*Rm*,*Rv*⟩*bg-impl-rel-eext*
≡ ⟨*Rm*, ⟨*Rv*⟩*fun-set-rel*⟩*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*]:
⟦*single-valued Rm*; *single-valued Rv*; *Range Rv* = *UNIV*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*bg-impl-rel-eext*)

unfolding *bg-impl-rel-eext-def* **by** *tagged-solver*

lemma *bg-fs-tag*: *TERM* (⟨*Rv*⟩*fun-set-rel*) **.**

abbreviation *bgv-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*bg-impl-rel-eext*, *Rv*⟩*frgv-impl-rel-ext*

abbreviation *bg-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*bg-impl-rel-eext*, *Rv*⟩*g-impl-rel-ext*

context fixes *Rv* :: (*'vi*×*'v*) *set* **begin**
lemmas [*autoref-rules*] = *gen-bg-refine*[
OF frgv-tag[*of Rv*] *bg-fs-tag*[*of Rv*],
folded frgv-impl-rel-ext-def bg-impl-rel-eext-def]

lemmas [*autoref-rules*] = *gen-bg-refine*[
OF g-tag[*of Rv*] *bg-fs-tag*[*of Rv*],
folded g-impl-rel-ext-def bg-impl-rel-eext-def]
end

schematic-goal (*?c*::*?'c*,
λ*G x*. *if x* ∈ *bg-F G 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*, λ*E* (*V0*::*'a set*) *F*.
(| *g-V* = {}, *g-E* = *E*, *g-V0* = *V0*, *bg-F* = *F* |))∈*?R*
apply (*autoref* (*keep-goal*))
done

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: (*'a*×*'a*) *set*]
shows (*?c*::*?'c*, λ*E* (*V0*::*'a set*) *F*.
(| *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*, *bg-F* = *F* |))∈*?R*
apply (*autoref* (*keep-goal*))
done

8.6 System Automata

consts
i-sa-eext :: *interface* ⇒ *interface* ⇒ *interface* ⇒ *interface*

abbreviation *i-sa Ie Iv Il* ≡ $\langle\langle Ie,Iv,Il\rangle_i$*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* $\to_i$ *Iv* $\to_i$ *Il*
  sa-rec-ext $::_i$ (*Iv* $\to_i$ *Il*) $\to_i$ *Ie* $\to_i$ ⟨*Ie*,*Iv*,*Il*⟩$_i$*i-sa-eext*
  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-eext-def-internal*:
  gen-sa-impl-rel-eext 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-eext-def*:
  ⟨*Rm*,*Rl*⟩*gen-sa-impl-rel-eext* = { (
  ⦇ *sai-L* = *Li*, ...=*mi* ⦈,
  ⦇ *sa-L* = *L*, ...=*m* ⦈)
  | *Li mi L m*.
    (*Li*,*L*)∈*Rl*
  ∧ (*mi*,*m*)∈*Rm*
  }
  unfolding *gen-sa-impl-rel-eext-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-eext*)
  unfolding *gen-sa-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-sa-impl-rel-ext*
  :: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (- × (*'q*,*'l*,-) *sa-rec-scheme*) *set*
  where *gen-sa-impl-rel-ext Rm Rf*
  ≡ ⟨⟨*Rm*,*Rf*⟩*gen-sa-impl-rel-eext*⟩*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*)
  ∈ ⟨*Rv*,*Re*,*Rv0*⟩*gen-sa-impl-rel-ext Rm Rl* → *Rl*
(*gen-sa-impl-ext*, *sa-rec-ext*)
  ∈ *Rl* → *Rm* → ⟨*Rm*,*Rl*⟩*gen-sa-impl-rel-eext*
unfolding *gen-sa-impl-rel-eext-def gen-g-impl-rel-ext-def*
by *auto*

8.6.1 Implementation with Function

definition *sa-impl-rel-eext-internal-def*:
sa-impl-rel-eext Rm Rv Rl
≡ ⟨*Rm*, *Rv*→*Rl*⟩*gen-sa-impl-rel-eext*

lemma *sa-impl-rel-eext-def*:
⟨*Rm*,*Rv*,*Rl*⟩*sa-impl-rel-eext*
  ≡ ⟨*Rm*, *Rv*→*Rl*⟩*gen-sa-impl-rel-eext*
unfolding *sa-impl-rel-eext-internal-def relAPP-def* **by** *simp*

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

lemma [*relator-props*, *simp*]:
⟦*single-valued Rm*; *single-valued Rl*; *Range Rv* = *UNIV*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*,*Rl*⟩*sa-impl-rel-eext*)
unfolding *sa-impl-rel-eext-def* **by** *tagged-solver*

lemma *sa-f-tag*: *TERM* (*Rv*→*Rl*) **.**

abbreviation *sav-impl-rel-ext Rm Rv Rl*
≡ ⟨⟨*Rm*, *Rv*, *Rl*⟩*sa-impl-rel-eext*, *Rv*⟩*frgv-impl-rel-ext*

abbreviation *sa-impl-rel-ext Rm Rv Rl*
≡ ⟨⟨*Rm*, *Rv*, *Rl*⟩*sa-impl-rel-eext*, *Rv*⟩*g-impl-rel-ext*

type-synonym (′*v*,′*l*,′*m*) *sav-impl-scheme* =
(′*v*, ⦇ *sai-L* :: ′*v* ⇒ ′*l* , . . .::′*m* ⦈) *frgv-impl-scheme*

type-synonym (′*v*,′*l*,′*m*) *sa-impl-scheme* =
(′*v*, ⦇ *sai-L* :: ′*v* ⇒ ′*l* , . . .::′*m* ⦈) *g-impl-scheme*

context fixes *Rv* :: (′*vi*×′*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-eext-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-eext-def]
end

schematic-goal (*?c*::*?'c*,
  λG 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* ≡ *OP gba-to-idx-ext* \$ *ecnv* **by** *simp*
end

schematic-goal *gba-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*, *gba-to-idx-ext* :: - ⇒ (*'q*, *'l*, -) *gba-rec-scheme* ⇒ -)
∈ (*gbav-impl-rel-ext Re Rv Rl*→*Ri*)
→ *gbav-impl-rel-ext Re Rv Rl*
→ ⟨*igbav-impl-rel-ext Ri Rv Rl*⟩*nres-rel*
using [[*autoref-trace-failed-id*]] **unfolding** *ti-Lcnv-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 ≤ *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* (λ(*q*,*x*). *x*=*0* ∧ *q*∈*V*),
g-E= *E-of-succ* (λ(*q*,*x*). *if x*=*0 then* (λ*q'*. (*q'*,*0*))‘*succ-of-E E q else* {}),
g-V0 = (λ*q'*. (*q'*,*0*))‘*V0*,
bg-F = *Collect* (λ(*q*,*x*). *x*=*0* ∧ *q*∈*V*),
. . . = *ecnv G*
|)
else (|
g-V = *Collect* (λ(*q*,*x*). *x*<*num-acc* ∧ *q*∈*V*),
g-E = *E-of-succ* (λ(*q*,*i*).
if i<*num-acc then*
let
i' = *if i* ∈ *acc q then* (*i* + *1*) *mod num-acc else i*
in (λ*q'*. (*q'*,*i'*))‘*succ-of-E E q*
else {}
),

g-V0 = (λ*q'*. (*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* $\times_r$ *nat-rel*)
unfolding *degen-impl-aux-alt*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*))
done

end

definition [*simp*]:
op-igb-graph-degeneralize-ext ecnv G ≡ *igb-graph.degeneralize-ext G ecnv*

lemma [*autoref-op-pat*]:
igb-graph.degeneralize-ext ≡ λ*G ecnv*. *op-igb-graph-degeneralize-ext ecnv G*
by *simp*

thm *igb-graph.degeneralize-ext-impl-aux*[*param-fo*]
concrete-definition *degeneralize-ext-impl*
uses *igb-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* (*igb-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-igb-graph-degeneralize-ext*
::: (*igbg-impl-rel-ext Re Rv* → *Re'*) → *igbg-impl-rel-ext Re Rv*
→ *bg-impl-rel-ext Re'* (*Rv* $\times_r$ *nat-rel*))$*ecnv*$*G*)
∈ *bg-impl-rel-ext Re'* (*Rv* $\times_r$ *nat-rel*)
proof −
from *assms* **have** *A*: *igb-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* $\times_r$ *Rs*)
  unfolding *prod-impl-aux-alt*[*abs-def*]
  apply (*autoref* (*keep-goal*))
  done

end

```
definition [simp]: op-igba-sys-prod ≡ igba-sys-prod-precond.prod

lemma [autoref-op-pat]:
  igba-sys-prod-precond.prod ≡ op-igba-sys-prod
  by simp

thm igba-sys-prod-precond.prod-impl-aux[param-fo]
concrete-definition igba-sys-prod-impl
  uses igba-sys-prod-precond.prod-impl-aux[param-fo]

thm igba-sys-prod-impl.refine

context begin interpretation autoref-syn .
lemma [autoref-rules]:
  fixes Re
  assumes SIDE-PRECOND (igba G)
  assumes SIDE-PRECOND (sa S)
  assumes GR: (Gi,G)∈igba-impl-rel-ext unit-rel Rq Rl
  assumes SR: (Si,S)∈sa-impl-rel-ext unit-rel Rs Rl
  shows (igba-sys-prod-impl Gi Si,
    (OP op-igba-sys-prod
      ::: igba-impl-rel-ext unit-rel Rq Rl
       → sa-impl-rel-ext unit-rel Rs Rl
       → igbg-impl-rel-ext unit-rel (Rq ×r Rs) )$G$S )
  ∈ igbg-impl-rel-ext unit-rel (Rq ×r Rs)
proof −
  from assms interpret igba: igba G + sa: sa S by simp-all
  have A: igba-sys-prod-precond G S by unfold-locales

  show ?thesis
    apply simp
    using igba-sys-prod-impl.refine[OF A GR SR]
    .
qed

end

schematic-goal
  assumes [simp]: igba G   sa S
  assumes [autoref-rules]: (Gi,G)∈igba-impl-rel-ext unit-rel Rq Rl
  assumes [autoref-rules]: (Si,S)∈sa-impl-rel-ext unit-rel Rs Rl
  shows (?c::?'c,igba-sys-prod-precond.prod G S)∈?R
  apply (autoref (keep-goal))
  done

end
```