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* = $('v \times 'v)$ *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* $\Rightarrow$ $'v \Rightarrow$ $'v$ *list* $\Rightarrow$ $'v \Rightarrow$ *bool* **for** E **where**
path0: *path* E u [] u
| *path-prepend*: $[\![(u,v) \in E;$ *path* E v l $w]\!] \Longrightarrow$ *path* E u $(u \# l)$ w

lemma *path1*: $(u,v) \in E \Longrightarrow$ *path* E u $[u]$ v
⟨*proof*⟩

lemma *path-empty-conv*[*simp*]:
path E u [] $v \longleftrightarrow u = v$
⟨*proof*⟩

inductive-cases *path-uncons*: *path* E u $(u' \# l)$ w
inductive-simps *path-cons-conv*: *path* E u $(u' \# l)$ w

lemma *path-no-edges*[*simp*]: *path* $\{\}$ u p $v \longleftrightarrow (u = v \wedge p = [])$
⟨*proof*⟩

lemma *path-conc*:
assumes *P1*: *path* E u la v
assumes *P2*: *path* E v lb w
shows *path* E u $(la @ lb)$ w
⟨*proof*⟩

lemma *path-append*:
⟦ *path E u l v*; $(v,w)\in E$ ⟧ ⟹ *path E u* $(l@[v])$ *w*
⟨*proof*⟩

lemma *path-unconc*:
assumes *path E u* $(la@lb)$ *w*
obtains *v* **where** *path E u la v* **and** *path E v lb w*
⟨*proof*⟩

lemma *path-conc-conv*:
path E u $(la@lb)$ *w* ⟷ $(\exists\, v.$ *path E u la v* ∧ *path E v lb w*$)$
⟨*proof*⟩

lemma (**in** −) *path-append-conv*: *path E u* $(p@[v])$ *w* ⟷ (*path E u p v* ∧ $(v,w)\in E$)
⟨*proof*⟩

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

lemmas *path-trans*[*trans*] = *path-prepend path-conc path-append*
lemma *path-from-edges*: ⟦$(u,v)\in E$; $(v,w)\in E$⟧ ⟹ *path E u* $[u]$ *v*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *path-mono*:
assumes *S*: $E\subseteq E'$
assumes *P*: *path E u p v*
shows *path* E' *u p v*
⟨*proof*⟩

lemma *path-is-rtrancl*:
assumes *path E u l v*
shows $(u,v)\in E^*$

⟨*proof*⟩

lemma *rtrancl-is-path*:
assumes $(u,v)\in E^*$
obtains l **where** *path* E u l v
⟨*proof*⟩

lemma *path-is-trancl*:
assumes *path* E u l v
and $l\neq[]$
shows $(u,v)\in E^+$
⟨*proof*⟩

lemma *trancl-is-path*:
assumes $(u,v)\in E^+$
obtains l **where** $l\neq[]$ **and** *path* E u l v
⟨*proof*⟩

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

lemma *path-mapI*:
assumes *path* E u p v
shows *path* (*pairself f* ‘ E) (f u) (*map f p*) (f v)
⟨*proof*⟩

lemma *path-restrict*:
assumes *path* E u p v
shows *path* ($E \cap$ *set* $p \times$ *insert* v (*set* (*tl* p))) u p v
⟨*proof*⟩

lemma *path-restrict-closed*:
assumes *CLOSED*: $E``D \subseteq D$
assumes I: $v\in D$ **and** P: *path* E v p v'
shows *path* ($E\cap D\times D$) v p v'
⟨*proof*⟩

lemma *path-set-induct*:
assumes *path* E u p v **and** $u\in I$ **and** $E``I \subseteq I$
shows *set* $p \subseteq I$
⟨*proof*⟩

lemma *path-nodes-reachable*: *path* E u p $v \Longrightarrow$ *insert* v (*set* p) $\subseteq E^*``\{u\}$
⟨*proof*⟩

lemma *path-nodes-edges*: *path* E u p $v \Longrightarrow$ *set* $p \subseteq$ *fst*‘E

⟨*proof*⟩

lemma *path-tl-nodes-edges*:
assumes *path E u p v*
shows *set (tl p)* ⊆ *fst‘E* ∩ *snd‘E*
⟨*proof*⟩

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*⟩

lemma *path-hd*:
assumes *p* ≠ [] *path E v p w*
shows *hd p* = *v*
⟨*proof*⟩

lemma *path-last-is-edge*:
assumes *path E x p y*
and *p* ≠ []
shows (*last p*, *y*) ∈ *E*
⟨*proof*⟩

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

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*⟩

lemma *path-restrict-tl*:
⟦ *u*∉*R*; *path* (*E* ∩ *UNIV* × −*R*) *u p v* ⟧ ⟹ *path* (*rel-restrict E R*) *u p v*
⟨*proof*⟩

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*⟩

lemma *dropWhileNot-path*:
assumes $p \neq []$
and *path E w p x*
and $v \in$ *set p*
and *dropWhile* $((\neq)\ v)\ p = c$
shows *path E v c x*
⟨*proof*⟩

lemma *takeWhileNot-path*:
assumes $p \neq []$
and *path E w p x*
and $v \in$ *set p*
and *takeWhile* $((\neq)\ v)\ p = c$
shows *path E w c v*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ipath-iter-conv*:
assumes *p*≠[]
shows *ipath E* (p^ω) ⟷ (*path E* (*hd p*) *p* (*hd p*))
⟨*proof*⟩

lemma *ipath-to-rtrancl*:
assumes *R*: *ipath E r*
assumes *I*: *i1*≤*i2*
shows (*r i1*,*r i2*)∈E^*
⟨*proof*⟩

lemma *ipath-to-trancl*:
assumes *R*: *ipath E r*
assumes *I*: *i1*<*i2*
shows (*r i1*,*r i2*)∈E^+
⟨*proof*⟩

lemma *run-limit-two-connectedI*:
assumes *A*: *ipath E r*
assumes *B*: *a* ∈ *limit r* *b*∈*limit r*
shows (*a*,*b*)∈E^+

⟨*proof*⟩

lemma *ipath-subpath*:
assumes *P*: *ipath E r*
assumes *LE*: $l \leq u$
shows *path E* (*r l*) (*map r* [*l*..<*u*]) (*r u*)
⟨*proof*⟩

lemma *ipath-restrict-eq*: *ipath* $(E \cap (E^* \text{``} \{r\ 0\} \times E^* \text{``} \{r\ 0\}))\ r \longleftrightarrow$ *ipath E r*
⟨*proof*⟩
lemma *ipath-restrict*: *ipath E r* $\Longrightarrow$ *ipath* $(E \cap (E^* \text{``} \{r\ 0\} \times E^* \text{``} \{r\ 0\}))\ r$
⟨*proof*⟩

lemma *ipathI*[*intro?*]: $[\![\bigwedge i.\ (r\ i, r\ (Suc\ i)) \in E]\!] \Longrightarrow$ *ipath E r*
⟨*proof*⟩

lemma *ipathD*: *ipath E r* $\Longrightarrow (r\ i, r\ (Suc\ i)) \in E$
⟨*proof*⟩

lemma *ipath-in-Domain*: *ipath E r* $\Longrightarrow r\ i \in$ *Domain E*
⟨*proof*⟩

lemma *ipath-in-Range*: $[\![$*ipath E r*; $i \neq 0]\!] \Longrightarrow r\ i \in$ *Range E*
⟨*proof*⟩

lemma *ipath-suffix*: *ipath E r* $\Longrightarrow$ *ipath E* (*suffix i r*)
⟨*proof*⟩

1.3 Strongly Connected Components

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

definition *is-scc* :: *'q digraph* $\Rightarrow$ *'q set* $\Rightarrow$ *bool*
where *is-scc E U* $\longleftrightarrow U \times U \subseteq E^* \wedge (\forall V.\ V \supset U \longrightarrow \neg\ (V \times V \subseteq E^*))$

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

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

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*⟩

lemma *is-scc-connected*:

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

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

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

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

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

mconn is an equivalence relation:

lemma *mconn-refl*[*simp*]: $Id \subseteq mconn\ E$
⟨*proof*⟩

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

lemma *mconn-trans*: $mconn\ E\ O\ mconn\ E = mconn\ E$
⟨*proof*⟩

lemma *mconn-refl'*: *refl* $(mconn\ E)$
⟨*proof*⟩

lemma *mconn-sym'*: *sym* $(mconn\ E)$
⟨*proof*⟩

lemma *mconn-trans'*: *trans* $(mconn\ E)$
⟨*proof*⟩

lemma *mconn-equiv*: *equiv UNIV* $(mconn\ E)$
⟨*proof*⟩

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

lemma *is-scc E U* ⟷ $U \in UNIV\ //\ (E^* \cap (E^{-1})^*)$
⟨*proof*⟩

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

lemma *find-outside-node*:
assumes $(u,v)\in E^*$
assumes $(u,v)\notin(E\cap U\times U)^*$
assumes $u\in U \quad v\in U$
shows $\exists\, u'.\ u'\notin U \wedge (u,u')\in E^* \wedge (u',v)\in E^*$
⟨*proof*⟩

lemma *is-scc-restrict1*:
assumes *SCC*: *is-scc E U*
shows $U\times U\subseteq(E\cap U\times U)^*$
⟨*proof*⟩

lemma *is-scc-restrict2*:
assumes *SCC*: *is-scc E U*
assumes $V\supset U$
shows $\neg\ (V\times V\subseteq(E\cap V\times V)^*)$
⟨*proof*⟩

lemma *is-scc-restrict3*:
assumes *SCC*: *is-scc E U*
shows $((E^*\ ``((E^*\ ``U)\ -\ U))\cap U = \{\})$
⟨*proof*⟩

lemma *is-scc-alt-restrict-path*:
is-scc E U $\longleftrightarrow$ $U\neq\{\}\ \wedge$
$(U\times U\subseteq(E\cap U\times U)^*)\ \wedge\ ((E^*\ ``((E^*\ ``U)\ -\ U))\cap U = \{\})$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *is-scc-unique*:
assumes *SCC*: *is-scc E scc* *is-scc E scc'*
and *v*: $v\in scc \quad v\in scc'$
shows $scc = scc'$
⟨*proof*⟩

lemma *is-scc-ex1*:
$\exists!scc.$ *is-scc E scc* $\wedge\ v\in scc$
⟨*proof*⟩

lemma *is-scc-ex*:
$\exists\, scc.$ *is-scc E scc* $\wedge\ v\in scc$

⟨*proof*⟩

lemma *is-scc-connected'*:
⟦*is-scc E scc*; $x \in scc$; $y \in scc$⟧ ⟹ $(x,y) \in (Restr\ E\ scc)^*$
⟨*proof*⟩

definition *scc-of* :: $('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ set$
where
scc-of E v = (*THE scc*. *is-scc E scc* ∧ $v \in scc$)

lemma *scc-of-is-scc*[*simp*]:
is-scc E (*scc-of E v*)
⟨*proof*⟩

lemma *node-in-scc-of-node*[*simp*]:
$v \in$ *scc-of E v*
⟨*proof*⟩

lemma *scc-of-unique*:
assumes $w \in$ *scc-of E v*
shows *scc-of E v* = *scc-of E w*
⟨*proof*⟩

end

2 Directed Graphs

theory *Digraph*
imports
CAVA-Base.CAVA-Base
Digraph-Basic
begin

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

record $'v$ *graph-rec* =
g-V :: $'v\ set$
g-E :: $'v\ digraph$
g-V0 :: $'v\ set$

definition *graph-restrict* :: ($'v$, $'more$) *graph-rec-scheme* ⇒ $'v\ set$ ⇒ ($'v$, $'more$) *graph-rec-scheme*
where *graph-restrict G R* ≡
(|
g-V = *g-V G*,
g-E = *rel-restrict* (*g-E G*) *R*,
g-V0 = *g-V0 G* − *R*,
... = *graph-rec.more G*

⟫

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

lemma *graph-restrict-trivial*[*simp*]: *graph-restrict G* {} = *G* ⟨*proof*⟩

locale *graph-defs* =
fixes *G* :: ('*v*, '*more*) *graph-rec-scheme*
begin

abbreviation *V* ≡ *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* ⟨*proof*⟩

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* ⟨*proof*⟩
lemma *run-V0*: *is-run r* ⟹ *r 0* ∈ *V0* ⟨*proof*⟩

lemma *run-reachable*: *is-run r* ⟹ *range r* ⊆ *reachable*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *finite-E*: *finite V* ⟹ *finite E* ⟨*proof*⟩

end

locale *fb-graph* =
  graph G
  for *G* :: (*'v*, *'more*) *graph-rec-scheme*
  +
  assumes *finite-V0*[*simp*, *intro*!]: *finite V0*
  assumes *finitely-branching*[*simp*, *intro*]: *v* ∈ *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*⟩

  lemma *fb-graph-restrict*: *fb-graph* (*graph-restrict G R*)
    ⟨*proof*⟩

end

lemma (**in** *graph*) *fb-graphI-fr*:
  assumes *finite reachable*
  shows *fb-graph G*
⟨*proof*⟩

abbreviation *rename-E f E* ≡ (λ(*u*,*v*). (*f u*, *f v*))‘*E*

definition *fr-rename-ext ecnv f G* ≡ ⦇
    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'* ≡ *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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *E′-to-E*: (*u*,*v*) ∈ *g-E G′* ⟹ (*fi u*, *fi v*)∈*E*
⟨*proof*⟩

lemma *V0′-to-V0*: *v*∈*g-V0 G′* ⟹ *fi v* ∈ *V0*
⟨*proof*⟩

lemma *rtrancl-E′-sim*:
assumes (*f u*,*v′*)∈(*g-E G′*)*
assumes *u*∈*V*
shows ∃ *v*. *v′* = *f v* ∧ *v*∈*V* ∧ (*u*,*v*)∈*E**
⟨*proof*⟩

lemma *rtrancl-E′-to-E*: **assumes** (*u*,*v*)∈(*g-E G′*)* **shows** (*fi u*, *fi v*)∈*E**
⟨*proof*⟩

lemma *G′-invar*: *graph G′*
⟨*proof*⟩

sublocale *G′*: *graph G′* ⟨*proof*⟩

lemma *G′-finite-reachable*:
assumes *finite* ((*g-E G*)* `` *g-V0 G*)
shows *finite* ((*g-E G′*)* `` *g-V0 G′*)
⟨*proof*⟩

lemma *V′-to-V*: *v* ∈ *G′.V* ⟹ *fi v* ∈ *V*
⟨*proof*⟩

lemma *ipath-sim1*: *ipath E r* ⟹ *ipath G′.E* (*f o r*)
⟨*proof*⟩

lemma *ipath-sim2*: *ipath G′.E r* ⟹ *ipath E* (*fi o r*)
⟨*proof*⟩

lemma *run-sim1*: *is-run r* ⟹ *G′.is-run* (*f o r*)

⟨*proof*⟩

lemma *run-sim2*: *G'.is-run r* ⟹ *is-run* (*fi o r*)
⟨*proof*⟩

end

end

3 Automata

theory *Automata*
imports *Digraph*
begin

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

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

3.1 Generalized Buchi Graphs

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

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

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

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

lemma *is-gb-graph*: *gb-graph G* ⟨*proof*⟩

definition
is-acc :: *'Q word* ⇒ *bool* **where** *is-acc r* ≡ (∀ *A*∈*F*. ∃$_\infty$*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*. ∃$_\infty$*i*. *r i* ∈ *A*)
⟨*proof*⟩

lemma *acc-run-run*: *is-acc-run r* ⟹ *is-run r*
⟨*proof*⟩

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

lemma *acc-eq-limit*:
assumes *FIN*: *finite* (*range r*)
shows *is-acc r* ⟷ (∀ *A*∈*F*. *limit r* ∩ *A* ≠ {})
⟨*proof*⟩

lemma *is-acc-run-limit-alt*:
assumes *finite* (*E** `` *V0*)
shows *is-acc-run r* ⟷ *is-run r* ∧ (∀ *A*∈*F*. *limit r* ∩ *A* ≠ {})
⟨*proof*⟩

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

lemma *finite-V-Fe*:
assumes *finite V* *A* ∈ *F*
shows *finite A*
⟨*proof*⟩

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* }
⟨*proof*⟩

sublocale *G*': *gb-graph G*'

⟨*proof*⟩

lemma *acc-sim1*: *is-acc r* ⟹ *G'.is-acc* (*f o r*)
⟨*proof*⟩

lemma *acc-sim2*:
assumes *G'.is-acc r* **shows** *is-acc* (*fi o r*)
⟨*proof*⟩

lemma *acc-run-sim1*: *is-acc-run r* ⟹ *G'.is-acc-run* (*f o r*)
⟨*proof*⟩

lemma *acc-run-sim2*: *G'.is-acc-run r* ⟹ *is-acc-run* (*fi o r*)
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

definition *lang* ≡ *Collect* (*accept*)
lemma *langI*[*intro?*]: *accept w* ⟹ *w*∈*lang* ⟨*proof*⟩
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*)
    ⟨*proof*⟩

  sublocale *G'*: *gba G'*
    ⟨*proof*⟩

  lemma *L-sim1*: ⟦*range r* ⊆ *V*; *L* (*r i*) *l*⟧ ⟹ *G'.L* (*f* (*r i*)) *l*
    ⟨*proof*⟩

  lemma *L-sim2*: ⟦ *range r* ⊆ *f*‘*V*; *G'.L* (*r i*) *l* ⟧ ⟹ *L* (*fi* (*r i*)) *l*
    ⟨*proof*⟩

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

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

  lemma *finite-G'-V*:
    assumes *finite V*
    shows *finite G'.V*
    ⟨*proof*⟩

end

abbreviation *gba-rename* ≡ *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*
  ⟨*proof*⟩

3.3 Buchi Graphs

A Buchi graph has exactly one acceptance class

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

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

  lemma *is-b-graph: b-graph G ⟨proof⟩*

  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*
    ⟨proof⟩

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

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

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* ⟨*proof*⟩

  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*
    ⟨*proof*⟩

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

  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*
    ⟨*proof*⟩

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

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

3.5 Indexed acceptance classes

record *'Q igb-graph-rec* = *'Q graph-rec* +
  igbg-num-acc :: *nat*
  igbg-acc :: *'Q* ⇒ *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* ⟨*proof*⟩

lemma *acc-boundI*[*simp*, *intro*]: *x*∈*acc q* ⟹ *x*<*num-acc*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *to-gbg-alt1*:
gbg.E T m = *E*
gbg.V0 T m = *V0*
gbg.F T m = *F*
⟨*proof*⟩

lemma *F-fin*[*simp*,*intro*!]: *finite F*
⟨*proof*⟩

definition *is-acc* :: *'Q word* ⇒ *bool*
where *is-acc r* ≡ (∀ *n*<*num-acc*. ∃$_\infty$*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*
⟨*proof*⟩

lemma *is-acc-gbg*:
gbg.is-acc T m = *is-acc*
⟨*proof*⟩

lemma *is-acc-run-gbg*:
gbg.is-acc-run T m = *is-acc-run*
⟨*proof*⟩

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*⟩

lemma *acc-limit-alt'*:

finite (*range r*) $\Longrightarrow$ *is-acc r* $\longleftrightarrow$ ($\bigcup$(*acc* ‘ *limit r*) = {*0*..<*num-acc*})
⟨*proof*⟩

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* $\Longrightarrow$ *q* ∈ *V*
begin
abbreviation *L* **where** *L* ≡ *igba-L G*

lemma *is-igba*: *igba G* ⟨*proof*⟩

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

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

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

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*
⟨*proof*⟩

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

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

end

3.5.1 Indexing Conversion

definition *F-to-idx* :: '*Q set set* ⇒ (*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})
  ⟨proof⟩

definition mk-acc-impl Flist ≡ do {
  let acc = Map.empty;

  (-,acc) ← nfoldli Flist (λ-. True) (λA (i,acc). do {
    acc ← FOREACHi (λit 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}))
  ⟨proof⟩

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)
  ⟨proof⟩

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]: ⋀ A. A ∈ F ⟹ finite A
  shows gbg-to-idx-ext ecnv G ≤ SPEC (λG'.
    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⟩

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 (λG'.
    igba.accept G' = accept
  ∧ g-V G' = V
  ∧ g-E G' = E
  ∧ g-V0 G' = V0
  ∧ igba-rec.more G' = ecnv G
  ∧ igba G')
  ⟨proof⟩

corollary (in gba) gba-to-idx-ext-lang-correct:
  assumes [simp, intro]: ⋀ A. A ∈ F ⟹ finite A
  shows gba-to-idx-ext ecnv G ≤
    SPEC (λG'. igba.lang G' = lang ∧ igba-rec.more G' = ecnv G ∧ igba G')
  ⟨proof⟩
```

3.5.2 Degeneralization

context *igb-graph*
begin

definition *degeneralize-ext* :: - ⇒ ('*Q* × *nat*, -) *b-graph-rec-scheme* **where**
degeneralize-ext ecnv ≡
if num-acc = *0 then* (|
g-V = *V* × {*0*},
g-E = {((*q*,*0*),(*q*′,*0*)) | *q q*′. (*q*,*q*′)∈*E*},
g-V0 = *V0*×{*0*},
bg-F = *V* × {*0*},
... = *ecnv G*
|)
else (|
g-V = *V* × {*0*..<*num-acc*},
g-E = { ((*q*,*i*),(*q*′,*i*′)) | *i i*′ *q q*′.
i<*num-acc*
∧ (*q*,*q*′)∈*E*
∧ *i*′ = (*if i*∈*acc q then* (*i*+*1*) *mod num-acc else i*) },
g-V0 = *V0* × {*0*},
bg-F = {(*q*,*0*)| *q*. *0*∈*acc q*},
... = *ecnv G*
|)

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

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

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

sublocale *degen*: *b-graph degeneralize-ext m* ⟨*proof*⟩

lemma *degen-finite-reachable*:
assumes [*simp*, *intro*]: *finite* (*E** `` *V0*)
shows *finite* ((*g-E* (*degeneralize-ext ecnv*))* `` *g-V0* (*degeneralize-ext ecnv*))
⟨*proof*⟩

lemma *degen-is-run-sound*:
degen.is-run T m r ⟹ *is-run* (*fst o r*)
⟨*proof*⟩

lemma *degen-path-sound*:
assumes *path* (*degen.E T m*) *u p v*
shows *path E* (*fst u*) (*map fst p*) (*fst v*)
⟨*proof*⟩

lemma *degen-V0-sound*:

assumes $u \in degen.V0\ T\ m$
shows $fst\ u \in V0$
⟨*proof*⟩

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

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

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

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

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

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

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

⟨*proof*⟩

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*. ∃$_\infty$*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'*
⟨*proof*⟩

lemma *degen-acc-run-complete*:
assumes *AR*: *is-acc-run r*
obtains *r'*
where *degen.is-acc-run T m r'* **and** *r* = *fst o r'*
⟨*proof*⟩

lemma *degen-run-find-change*:
assumes *NN0*[*simp*]: *num-acc* ≠ *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*⟩

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*⟩

lemma *degen-acc-run-sound*:
assumes *A*: *degen.is-acc-run T m r*
shows *is-acc-run* (*fst o r*)
⟨*proof*⟩

lemma *degen-acc-run-iff*:
is-acc-run r ⟷ (∃ *r'*. *fst o r'* = *r* ∧ *degen.is-acc-run T m r'*)
⟨*proof*⟩

end

3.6 System Automata

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

record ('*Q*,'*L*) *sa-rec* = '*Q graph-rec* +
sa-L :: '*Q* ⇒ '*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* ⟨*proof*⟩

  definition *lang* ≡ *Collect accept*

  lemma *langI*[*intro?*]: *accept w* ⟹ *w*∈*lang* ⟨*proof*⟩

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*
    ⟨*proof*⟩

  sublocale *prod*: *igb-graph prod* ⟨*proof*⟩

  lemma *prod-finite-reachable*:
    assumes *finite* (*igba.E** `` *igba.V0*)  *finite* (*sa.E*\* `` *sa.V0*)
    shows *finite* ((*g-E prod*)* `` *g-V0 prod*)
  ⟨*proof*⟩

  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* {})
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *prod-acc*:
assumes *A*: *range* (*snd o r*) ⊆ *sa.V*
shows *prod.is-acc r* ⟷ *igba.is-acc* (*fst o r*)
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *lasso-of-prpl-simps*[*simp*]:
lasso-reach (*lasso-of-prpl prpl*) = *fst prpl*
snd prpl ≠ [] ⟹ *lasso-cycle* (*lasso-of-prpl prpl*) = *snd prpl*
⟨*proof*⟩

definition *run-of-lasso* :: *'q lasso* ⇒ *'q word*
— Run described by a lasso
where *run-of-lasso L* ≡ *lasso-reach L* ⌢ (*lasso-cycle L*)$^\omega$

lemma *run-of-lasso-of-prpl*:
pl ≠ [] ⟹ *run-of-lasso* (*lasso-of-prpl* (*pr*, *pl*)) = *pr* ⌢ *pl*$^\omega$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-lasso-run*[*simp*]:
shows *run-of-lasso* (*map-lasso f L*) = *f o* (*run-of-lasso L*)
⟨*proof*⟩

context *graph* **begin**
definition *is-lasso-pre* :: *'v lasso* ⇒ *bool*
where *is-lasso-pre L* ≡
lasso-v0 L ∈ *V0*

$\wedge$ *path E* (*lasso-v0 L*) (*lasso-reach L*) (*lasso-va L*)
$\wedge$ *path E* (*lasso-va L*) (*lasso-cycle L*) (*lasso-va L*)

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

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

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

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *is-lasso-empty*[*simp*]: *V0* = {} ⟹ ¬*is-lasso L*
⟨*proof*⟩

lemma *lasso-accepted*:
assumes *L*: *is-lasso L*
shows *is-acc-run* (*run-of-lasso L*)
⟨*proof*⟩

lemma *lasso-prpl-acc-run*:
is-lasso-prpl (*pr*, *pl*) ⟹ *is-acc-run* (*pr* ⌢ *iter pl*)
⟨*proof*⟩

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*⟩
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*
⟨*proof*⟩

lemma *is-lasso-gbg*:
gbg.is-lasso T m = *is-lasso*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *is-lasso-prpl-conv*:
is-lasso-prpl prpl ⟷ (*snd prpl*≠[] ∧ *is-lasso* (*lasso-of-prpl prpl*))
⟨*proof*⟩

lemma *lasso-prpl-acc-run*:
is-lasso-prpl (*pr*, *pl*) ⟹ *is-acc-run* (*pr* ⌢ *iter pl*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-lasso-prpl-conv*:
is-lasso-prpl prpl ⟷ (*snd prpl*≠[] ∧ *is-lasso* (*lasso-of-prpl prpl*))
⟨*proof*⟩

lemma *is-lasso-pre-ext*[*simp*]:
gbg.is-lasso-pre T m = *is-lasso-pre*
⟨*proof*⟩

lemma *is-lasso-gbg*: *gbg.is-lasso T m* = *is-lasso*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

end

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

lemma *lasso-rel-ext-def*: $\bigwedge$ $Re\ R.$ $\langle Re,R\rangle$*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.
$(r',r) \in \langle R\rangle$*list-rel*
$\wedge$ $(va',va)\in R$
$\wedge$ $(cysfx',\ cysfx) \in \langle R\rangle$*list-rel*
$\wedge$ $(m',m) \in Re$
}
⟨*proof*⟩

lemma *lasso-rel-ext-sv*[*relator-props*]:
$\bigwedge$ $Re\ R.$ $[\![$ *single-valued Re*; *single-valued R* $]\!] \Longrightarrow$ *single-valued* ($\langle Re,R\rangle$*lasso-rel-ext*)
⟨*proof*⟩

lemma *lasso-rel-ext-id*[*relator-props*]:
$\bigwedge Re\ R.$ $[\![$ $Re{=}Id$; $R{=}Id$ $]\!] \Longrightarrow$ $\langle Re,R\rangle$*lasso-rel-ext* = *Id*
⟨*proof*⟩

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

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

find-consts (-,-) *lasso-scheme*
term *lasso-reach-update*

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

$\bigwedge Re\ R.\ (lasso\text{-}cysfx\text{-}update,\ lasso\text{-}cysfx\text{-}update)$
$\in (\langle R\rangle list\text{-}rel \to \langle R\rangle list\text{-}rel) \to \langle Re,R\rangle lasso\text{-}rel\text{-}ext \to \langle Re,R\rangle lasso\text{-}rel\text{-}ext$
$\bigwedge Re\ R.\ (lasso.more\text{-}update,\ lasso.more\text{-}update)$
$\in (Re \to Re) \to \langle Re,R\rangle lasso\text{-}rel\text{-}ext \to \langle Re,R\rangle lasso\text{-}rel\text{-}ext$
⟨*proof*⟩

lemma *lasso-param2*[*param*, *autoref-rules*]:
$\bigwedge Re\ R.\ (lasso\text{-}v0,\ lasso\text{-}v0) \in \langle Re,R\rangle lasso\text{-}rel\text{-}ext \to R$
$\bigwedge Re\ R.\ (lasso\text{-}cycle,\ lasso\text{-}cycle) \in \langle Re,R\rangle lasso\text{-}rel\text{-}ext \to \langle R\rangle list\text{-}rel$
$\bigwedge Re\ R.\ (map\text{-}lasso,\ map\text{-}lasso)$
$\in (R \to R') \to \langle Re,R\rangle lasso\text{-}rel\text{-}ext \to \langle unit\text{-}rel,R'\rangle lasso\text{-}rel\text{-}ext$
⟨*proof*⟩

lemma *lasso-of-prpl-param*: $[\![(l',l) \in \langle R\rangle list\text{-}rel \times_r \langle R\rangle list\text{-}rel;\ snd\ l \neq []]\!]$
$\Longrightarrow (lasso\text{-}of\text{-}prpl\ l',\ lasso\text{-}of\text{-}prpl\ l) \in \langle unit\text{-}rel,R\rangle lasso\text{-}rel\text{-}ext$
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *lasso-of-prpl-autoref*[*autoref-rules*]:
assumes *SIDE-PRECOND* $(snd\ l \neq [])$
assumes $(l',l) \in \langle R\rangle list\text{-}rel \times_r \langle R\rangle list\text{-}rel$
shows (*lasso-of-prpl l'*,
(*OP lasso-of-prpl*
::: $\langle R\rangle list\text{-}rel \times_r \langle R\rangle list\text{-}rel \to \langle unit\text{-}rel,R\rangle lasso\text{-}rel\text{-}ext)\$l$)
$\in \langle unit\text{-}rel,R\rangle lasso\text{-}rel\text{-}ext$
⟨*proof*⟩

end

4.1 Implementing runs by lassos

definition *lasso-run-rel-def-internal*:
$lasso\text{-}run\text{-}rel\ R \equiv br\ run\text{-}of\text{-}lasso\ (\lambda\text{-}.\ True)\ O\ (nat\text{-}rel \to R)$
lemma *lasso-run-rel-def*:
$\langle R\rangle lasso\text{-}run\text{-}rel = br\ run\text{-}of\text{-}lasso\ (\lambda\text{-}.\ True)\ O\ (nat\text{-}rel \to R)$
⟨*proof*⟩

lemma *lasso-run-rel-sv*[*relator-props*]:
$single\text{-}valued\ R \Longrightarrow single\text{-}valued\ (\langle R\rangle lasso\text{-}run\text{-}rel)$
⟨*proof*⟩

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*]: $(o) \equiv$ *op-map-run* ⟨*proof*⟩

lemma *map-lasso-run-refine*[*autoref-rules*]:
shows (*map-lasso*,*op-map-run*) ∈ $(R \to R') \to \langle R \rangle$*lasso-run-rel* → $\langle R' \rangle$*lasso-run-rel*
⟨*proof*⟩

end

5 Simulation

theory *Simulation*
imports *Automata*
begin

lemma *finite-ImageI*:
assumes *finite A*
assumes $\bigwedge a.\ a \in A \Longrightarrow$ *finite* $(R``\{a\})$
shows *finite* $(R``A)$
⟨*proof*⟩

6 Simulation

6.1 Functional Relations

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

lemma *the-br*[*simp*]:
assumes *single-valued R*
shows *br* (*the-br-α R*) (*the-br-invar R*) = *R*
⟨*proof*⟩

lemma *the-br-br*[*simp*]:
I x ⟹ *the-br-α* (*br α I*) *x* = *α x*
the-br-invar (*br α I*) = *I*
⟨*proof*⟩

6.2 Relation between Runs

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

lemma *run-rel-converse*[*simp*]: $(ra,\ rb) \in$ *run-rel* $(R^{-1}) \longleftrightarrow (rb,\ ra) \in$ *run-rel R*
⟨*proof*⟩

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

$\Longrightarrow$ $(ra,\ rb) \in$ *run-rel* $R \longleftrightarrow ((\forall\, i.$ *the-br-invar* $R\ (ra\ i)) \wedge rb =$ *the-br-*α $R\ o$ $ra)$
⟨*proof*⟩

6.3 Simulation

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

lemma *simulation-this*: *simulation* $R\ A\ B$ ⟨*proof*⟩

lemma *run-sim*:
assumes *arun*: *a.is-run ra*
obtains *rb* **where** *b.is-run rb* $(ra,\ rb) \in$ *run-rel* R
⟨*proof*⟩

lemma *stuck-sim*:
assumes $(a,\ b) \in R$
assumes $b \notin$ *Domain b.E*
shows $a \notin$ *Domain a.E*
⟨*proof*⟩

lemma *run-Domain*: *a.is-run r* $\Longrightarrow$ $r\ i \in$ *Domain* R
⟨*proof*⟩

lemma *br-run-sim*:
assumes $R =$ *br* $\alpha\ I$
assumes *a.is-run r*
shows *b.is-run* $(\alpha\ o\ r)$
⟨*proof*⟩

lemma *is-reachable-sim*: $a \in a.E^*$ `` $a.V0 \Longrightarrow \exists\ b.\ (a,\ b) \in R \wedge b \in b.E^*$ `` $b.V0$
⟨*proof*⟩

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

lemma *reachable-finite-sim*:

assumes *finite* $(b.E^*\ ``\ b.V0)$
assumes $\bigwedge b.\ b \in b.E^*\ ``\ b.V0 \Longrightarrow$ *finite* $(R^{-1}\ ``\ \{b\})$
shows *finite* $(a.E^*\ ``\ a.V0)$
⟨*proof*⟩

end

lemma *simulation-trans*[*trans*]:
assumes *simulation R1 A B*
assumes *simulation R2 B C*
shows *simulation* $(R1\ O\ R2)\ A\ C$
⟨*proof*⟩

lemma (**in** *graph*) *simulation-refl*[*simp*]: *simulation Id G G* ⟨*proof*⟩

locale *lsimulation* =
a: *sa A* +
b: *sa B* +
simulation R A B
for $R :: ('a \times 'b)\ set$
and $A :: ('a, 'l, -)\ sa\text{-}rec\text{-}scheme$
and $B :: ('b, 'l, -)\ sa\text{-}rec\text{-}scheme$
+
assumes *labeling-consistent*: $(a, b) \in R \Longrightarrow a.L\ a = b.L\ b$
begin

lemma *lsimulation-this*: *lsimulation R A B* ⟨*proof*⟩

lemma *run-rel-consistent*: $(ra, rb) \in$ *run-rel* $R \Longrightarrow a.L\ o\ ra = b.L\ o\ rb$
⟨*proof*⟩

lemma *accept-sim*: $a.accept\ w \Longrightarrow b.accept\ w$
⟨*proof*⟩

end

lemma *lsimulation-trans*[*trans*]:
assumes *lsimulation R1 A B*
assumes *lsimulation R2 B C*
shows *lsimulation* $(R1\ O\ R2)\ A\ C$
⟨*proof*⟩

lemma (**in** *sa*) *lsimulation-refl*[*simp*]: *lsimulation Id G G* ⟨*proof*⟩

6.4 Bisimulation

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

s1: *simulation* R A B +
s2: *simulation* R^{-1} B A
for R :: $('a \times 'b)$ *set*
and A :: $('a,$ -$)$ *graph-rec-scheme*
and B :: $('b,$ -$)$ *graph-rec-scheme*
begin

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

lemma *converse*: *bisimulation* (R^{-1}) B A
⟨*proof*⟩

lemma *br-run-conv*:
assumes $R = br\ \alpha\ I$
shows *b.is-run rb* ⟷ $(\exists\, ra.\ rb{=}\alpha\ o\ ra \wedge$ *a.is-run ra*$)$
⟨*proof*⟩

lemma *bri-run-conv*:
assumes $R = (br\ \gamma\ I)^{-1}$
shows *a.is-run ra* ⟷ $(\exists\, rb.\ ra{=}\gamma\ o\ rb \wedge$ *b.is-run rb*$)$
⟨*proof*⟩

lemma *inj-map-run-eq*:
assumes *inj* α
assumes *E*: $\alpha\ o\ r1 = \alpha\ o\ r2$
shows $r1 = r2$
⟨*proof*⟩

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

lemma *single-valued-run-conv*:
assumes *single-valued R*
shows *b.is-run rb*
⟷ $(\exists\, ra.\ rb{=}$*the-br-*$\alpha\ R\ o\ ra \wedge$ *a.is-run ra*$)$
⟨*proof*⟩

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

end

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

assumes *bisimulation R2 B C*
shows *bisimulation (R1 O R2) A C*
⟨*proof*⟩

lemma (**in** *graph*) *bisimulation-refl*[*simp*]: *bisimulation Id G G* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma *accept-bisim*: *a.accept* = *b.accept*
⟨*proof*⟩

end

lemma *lbisimulation-trans*[*trans*]:
assumes *lbisimulation R1 A B*
assumes *lbisimulation R2 B C*
shows *lbisimulation (R1 O R2) A C*
⟨*proof*⟩

lemma (**in** *sa*) *lbisimulation-refl*[*simp*]: *lbisimulation Id G G* ⟨*proof*⟩

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*

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

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

$b \in succ\text{-}of\text{-}rel\ r\ a \longleftrightarrow (a,b) \in r$
$b \in succ\text{-}of\text{-}pred\ p\ a \longleftrightarrow p\ a\ b$
⟨*proof*⟩

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

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

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

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

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

$succ\text{-}of\text{-}pred\ (pred\text{-}of\text{-}succ\ s) = s$
$succ\text{-}of\text{-}pred\ (pred\text{-}of\text{-}rel\ r) = succ\text{-}of\text{-}rel\ r$
⟨*proof*⟩

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

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

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

lemma *m2r-expand*[*simp*]:
$(a,b) \in m2r\text{-}rel\ r \longleftrightarrow (\exists\, a'.\ a = Some\ a' \wedge (a',b) \in r)$
$m2r\text{-}pred\ p\ a\ b \longleftrightarrow (\exists\, a'.\ a = Some\ a' \wedge p\ a'\ b)$
$b \in m2r\text{-}succ\ s\ a \longleftrightarrow (\exists\, a'.\ a = Some\ a' \wedge b \in s\ a')$
⟨*proof*⟩

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-pred (*pred-of-succ s*) = *pred-of-succ* (*m2r-succ s*)
m2r-pred (*pred-of-rel r*) = *pred-of-rel* (*m2r-rel r*)

m2r-succ (*succ-of-pred p*) = *succ-of-pred* (*m2r-pred p*)
m2r-succ (*succ-of-rel r*) = *succ-of-rel* (*m2r-rel r*)
⟨*proof*⟩

definition *rel-of-enex enex* ≡ *let* (*en*, *ex*) = *enex in* {(*s*, *ex a s*) |*s a*. *a* ∈ *en s*}
definition *pred-of-enex enex* ≡ λ*s s*′. *let* (*en*,*ex*) = *enex in* ∃ *a*∈*en s*. *s*′=*ex a s*
definition *succ-of-enex enex* ≡ λ*s*. *let* (*en*,*ex*) = *enex in* {*s*′. ∃ *a*∈*en s*. *s*′=*ex a s*}

lemma *x-of-enex-expand*[*simp*]:
(*s*, *s*′) ∈ *rel-of-enex* (*en*, *ex*) ⟷ (∃ *a* ∈ *en s*. *s*′ = *ex a s*)
pred-of-enex (*en*,*ex*) *s s*′ ⟷ (∃ *a*∈*en s*. *s*′=*ex a s*)
s′∈*succ-of-enex* (*en*,*ex*) *s* ⟷ (∃ *a*∈*en s*. *s*′=*ex a s*)
⟨*proof*⟩

lemma *x-of-enex-conv*[*simp*]:
rel-of-pred (*pred-of-enex enex*) = *rel-of-enex enex*
rel-of-succ (*succ-of-enex enex*) = *rel-of-enex enex*
pred-of-rel (*rel-of-enex enex*) = *pred-of-enex enex*
pred-of-succ (*succ-of-enex enex*) = *pred-of-enex enex*
succ-of-rel (*rel-of-enex enex*) = *succ-of-enex enex*
succ-of-pred (*pred-of-enex enex*) = *succ-of-enex enex*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *stutter-extend-edgesI-stutter*:
assumes *v* ∈ *V* *v* ∉ *Domain E*
shows (*v*, *v*) ∈ *stutter-extend-edges V E*
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *stutter-extend-wf*: $E \subseteq V \times V \Longrightarrow$ *stutter-extend-edges* $V\ E \subseteq V \times V$
⟨*proof*⟩

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

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

definition *stutter-extend* :: ('*v*, -) *graph-rec-scheme* ⇒ ('*v*, -) *graph-rec-scheme*
where *stutter-extend* $G \equiv$
(|
g-V = *g-V* G,
g-E = *stutter-extend-edges* (*g-V* G) (*g-E* G),
g-V0 = *g-V0* G,
... = *graph-rec.more* G
|)

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

lemma *stutter-extend-simps-sa*[*simp*]:
sa-L (*stutter-extend* G) = *sa-L* G
⟨*proof*⟩

lemma (**in** *graph*) *stutter-extend-graph*: *graph* (*stutter-extend* G)
⟨*proof*⟩
lemma (**in** *sa*) *stutter-extend-sa*: *sa* (*stutter-extend* G)
⟨*proof*⟩

lemma (**in** *bisimulation*) *stutter-extend*: *bisimulation* R (*stutter-extend* A) (*stutter-extend* B)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stutter-extend-ex-None-eq*[*simp*]:
stutter-extend-ex ex None = *id*
⟨*proof*⟩

end

7 Implementing Graphs

theory *Digraph-Impl*
imports *Digraph*
begin

7.1 Directed Graphs by Successor Function

type-synonym *'a slg* = *'a* ⇒ *'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*)
⟨*proof*⟩

lemma *slg-rel-sv*[*relator-props*]:
⟦*single-valued R*; *Range R* = *UNIV*⟧ ⟹ *single-valued* (⟨*R*⟩*slg-rel*)
⟨*proof*⟩

consts *i-slg* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of slg-rel i-slg*]

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

lemma [*autoref-itype*]: *op-slg-succs* ::$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ *I* →$_i$ ⟨*I*⟩$_i$*i-set* ⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-op-pat*]: $E``\{v\} \equiv$ *op-slg-succs*Ev ⟨*proof*⟩
end

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

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

lemma *E-of-succ-of-E*[*simp*]: *E-of-succ* (*succ-of-E E*) = *E*
⟨*proof*⟩

lemma *succ-of-E-of-succ*[*simp*]: *succ-of-E* (*E-of-succ E*) = *E*
⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]: *E-of-succ* ::$_i$ (*I* →$_i$ ⟨*I*⟩$_i$*i-set*) →$_i$ ⟨*I*⟩$_i$*i-slg* ⟨*proof*⟩
lemma [*autoref-itype*]: *succ-of-E* ::$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ *I* →$_i$ ⟨*I*⟩$_i$*i-set* ⟨*proof*⟩
end

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

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* ≡ $E \cap Vl \times Vr$

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* ≡ $E \cap Vl \times UNIV$

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* ≡ $E \cap UNIV \times Vr$

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*
⟨*proof*⟩

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* $\{\}$)
⟨*proof*⟩
lemma *graph-restrict-left-aimpl*: *op-graph-restrict-left Vl E* =

E-of-succ (λ*v*. *if v*∈*Vl then E*‘‘{*v*} *else* {})
⟨*proof*⟩
lemma *graph-restrict-right-aimpl*: *op-graph-restrict-right Vr E* =
E-of-succ (λ*v*. {*x* ∈ *E*‘‘{*v*}. *x*∈*Vr*})
⟨*proof*⟩

schematic-goal *graph-restrict-impl-aux*:
fixes *Rsl Rsr*
notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
assumes [*autoref-rules*]: (*meml*, (∈)) ∈ *R* → ⟨*R*⟩*Rsl* → *bool-rel*
assumes [*autoref-rules*]: (*memr*, (∈)) ∈ *R* → ⟨*R*⟩*Rsr* → *bool-rel*
shows (*?c*, *op-graph-restrict*) ∈ ⟨*R*⟩*Rsl* → ⟨*R*⟩*Rsr* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*slg-rel*
⟨*proof*⟩

schematic-goal *graph-restrict-left-impl-aux*:
fixes *Rsl Rsr*
notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
assumes [*autoref-rules*]: (*meml*, (∈)) ∈ *R* → ⟨*R*⟩*Rsl* → *bool-rel*
shows (*?c*, *op-graph-restrict-left*) ∈ ⟨*R*⟩*Rsl* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*slg-rel*
⟨*proof*⟩

schematic-goal *graph-restrict-right-impl-aux*:
fixes *Rsl Rsr*
notes [*autoref-rel-intf*] = *REL-INTFI*[*of Rsl i-set*] *REL-INTFI*[*of Rsr i-set*]
assumes [*autoref-rules*]: (*memr*, (∈)) ∈ *R* → ⟨*R*⟩*Rsr* → *bool-rel*
shows (*?c*, *op-graph-restrict-right*) ∈ ⟨*R*⟩*Rsr* → ⟨*R*⟩*slg-rel* → ⟨*R*⟩*slg-rel*
⟨*proof*⟩

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]:
op-graph-restrict ::$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ ⟨*I*⟩$_i$*i-slg*
op-graph-restrict-right ::$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ ⟨*I*⟩$_i$*i-slg*
op-graph-restrict-left ::$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ ⟨*I*⟩$_i$*i-slg*
⟨*proof*⟩
end

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

schematic-goal (*?c*::*?*′*c*, λ(*E*::*nat digraph*) *x*. *E*‘‘{*x*}) ∈ *?R*
⟨*proof*⟩

lemma *graph-minus-aimpl*:

fixes *E1 E2* :: *'a rel*
shows *E1−E2 = E-of-succ* (λ*x*. *E1''*{*x*} − *E2''*{*x*})
⟨*proof*⟩

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*
⟨*proof*⟩

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*})
⟨*proof*⟩

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* ×$_r$ *R* → ⟨*R* ×$_r$ *R*⟩*Rs* → *bool-rel*
shows (*?c*, (−)) ∈ ⟨*R*⟩*slg-rel* → ⟨*R*×$_r$*R*⟩*Rs* → ⟨*R*⟩*slg-rel*
⟨*proof*⟩

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* ≡ ⟨*i-unit*⟩$_i$*i-g-ext*

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *g-type*[*autoref-itype*]:
g-V ::$_i$ ⟨*Ie*,*I*⟩$_i$*i-g-ext* →$_i$ ⟨*I*⟩$_i$*i-set*
g-E ::$_i$ ⟨*Ie*,*I*⟩$_i$*i-g-ext* →$_i$ ⟨*I*⟩$_i$*i-slg*
g-V0 ::$_i$ ⟨*Ie*,*I*⟩$_i$*i-g-ext* →$_i$ ⟨*I*⟩$_i$*i-set*
graph-rec-ext
::$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ *iE* →$_i$ ⟨*Ie*,*I*⟩$_i$*i-g-ext*
⟨*proof*⟩

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*: ⋀*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*
  }
  ⟨*proof*⟩

lemma *gen-g-impl-rel-sv*[*relator-props*]:
  ⋀*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*)
  ⟨*proof*⟩

lemma *gen-g-refine*:
  ⋀*Rm Rv Re Rv0*. (*gi-V*,*g-V*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Rv*
  ⋀*Rm Rv Re Rv0*. (*gi-E*,*g-E*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Re*
  ⋀*Rm Rv Re Rv0*. (*gi-V0*,*g-V0*) ∈ ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext* → *Rv0*
  ⋀*Rm Rv Re Rv0*. (*gen-g-impl-ext*, *graph-rec-ext*)
    ∈ *Rv* → *Re* → *Rv0* → *Rm* → ⟨*Rm*,*Rv*,*Re*,*Rv0*⟩*gen-g-impl-rel-ext*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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

lemma [*relator-props*, *simp*]:
⟦*single-valued Rv*; *Range Rv* = *UNIV*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*frgv-impl-rel-ext*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*autoref-rel-intf*]: *REL-INTF g-impl-rel-ext i-g-ext*
⟨*proof*⟩

lemma [*relator-props*, *simp*]:
⟦*single-valued Rv*; *Range Rv* = *UNIV*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*⟩*g-impl-rel-ext*)
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩
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*
⟨*proof*⟩

lemma [*autoref-hom*]:
CONSTRAINT graph-rec-ext (⟨*Rv*⟩*Rvs* → ⟨*Rv*⟩*Res* → ⟨*Rv*⟩*Rv0s* → *Rm* → ⟨*Rm*,*Rv*⟩*Rg*)
⟨*proof*⟩

schematic-goal (*?c*::*?'c*, λ*G x*. *g-E G* ‘‘ {*x*})∈*?R*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *autoref-the-inv-into-map*[*autoref-rules*]:
fixes *Rv' :: ('vti × 'vt) set*
assumes *SIDE-GEN-ALGO (is-bounded-hashcode Rv' eq bhc)*
assumes *SIDE-GEN-ALGO (is-valid-def-hm-size TYPE('vti) def-size)*
assumes *INJ: SIDE-PRECOND (inj-on f V)*
assumes *V: (Vi,V)∈⟨Rv⟩list-set-rel*
assumes *F: (fi,f)∈Rv→Rv'*
shows *(the-inv-into-map-code eq bhc def-size Vi fi,*
(OP the-inv-into-map
::: ⟨Rv⟩list-set-rel → (Rv→Rv') → ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel bhc)
Vf) ∈ ⟨Rv', Rv⟩Impl-Array-Hash-Map.ahm-rel bhc
⟨*proof*⟩

end

schematic-goal *(?c::?'c, do {*
let s = {1,2,3::nat};
ASSERT (inj-on Suc s);
RETURN (the-inv-into-map s Suc) }) ∈ ?R
⟨*proof*⟩

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

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

definition *fi-map x* = (*if x* ∈ *f*‘*V then Some* (*fi x*) *else None*)
lemma *fi-map-alt*: *fi-map* = *the-inv-into-map V f*
⟨*proof*⟩

lemma *fi-map-Some*: (*fi-map u* = *Some v*) ⟷ *u*∈*f*‘*V* ∧ *fi u* = *v*
⟨*proof*⟩

lemma *fi-map-None*: (*fi-map u* = *None*) ⟷ *u*∉*f*‘*V*
⟨*proof*⟩

lemma *rename-E-aimpl-alt*: *rename-E f E* = *E-of-succ* (λ*v*. *case fi-map v of*
Some u ⇒ *f* ‘ (*succ-of-E E u*) | *None* ⇒ {})
⟨*proof*⟩

lemma *frv-rename-ext-aimpl-alt*:
assumes *ECNV*: *ecnv*′ *fi-map G* ≤ *SPEC* (λ*r*. *r* = *ecnv G*)
shows *fr-rename-ext-aimpl ecnv*′ *f G*
≤ *SPEC* (λ*r*. *r* = *fr-rename-ext ecnv f G*)
⟨*proof*⟩
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*
⟨*proof*⟩

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

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

7.3 Graphs from Lists

definition *succ-of-list* :: (*nat*×*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⟩


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

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

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

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

end
```

8 Implementing Automata

```
theory Automata-Impl
imports Digraph-Impl Automata
begin
```

8.1 Indexed Generalized Buchi Graphs

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

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

context begin interpretation autoref-syn ⟨proof⟩

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

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

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

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

lemma *gen-igbg-impl-rel-sv*[*relator-props*]:
  ⟦*single-valued Racc*; *single-valued Rm*⟧
  ⟹ *single-valued* (⟨*Rm*,*Racc*⟩*gen-igbg-impl-rel-eext*)
  ⟨*proof*⟩

abbreviation *gen-igbg-impl-rel-ext*
  :: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (-×(-,-)*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*
  ⟨*proof*⟩

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*:
⟨*Rm*,*Rv*⟩*igbg-impl-rel-eext* ≡ ⟨*Rm*, *Rv* → ⟨*nat-rel*⟩*bs-set-rel*⟩*gen-igbg-impl-rel-eext*
⟨*proof*⟩

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* (⟨*Rm*,*Rv*⟩*igbg-impl-rel-eext*)
⟨*proof*⟩

lemma *g-tag*: *TERM* (⟨*Rv*⟩*fun-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*) ⟨*proof*⟩
lemma *frgv-tag*: *TERM* (⟨*Rv*⟩*list-set-rel*,⟨*Rv*⟩*slg-rel*,⟨*Rv*⟩*list-set-rel*) ⟨*proof*⟩
lemma *igbg-bs-tag*: *TERM* (*Rv* → ⟨*nat-rel*⟩*bs-set-rel*) ⟨*proof*⟩

abbreviation *igbgv-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*igbg-impl-rel-eext*, *Rv*⟩*frgv-impl-rel-ext*

abbreviation *igbg-impl-rel-ext Rm Rv*
≡ ⟨⟨*Rm*, *Rv*⟩*igbg-impl-rel-eext*, *Rv*⟩*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*,
λG x. *if igbg-num-acc G* = *0* ∧ *1*∈*igbg-acc G x then* (*g-E G* `` {*x*}) *else* {}
)∈*?R*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

8.2 Indexed Generalized Buchi Automata

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

abbreviation *i-igba Ie Iv Il*
≡ ⟨⟨*Ie*,*Iv*,*Il*⟩*i i-igba-eext*,*Iv*⟩*i i-igbg-eext*,*Iv*⟩*i i-g-ext*
context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *igba-type*[*autoref-itype*]:
igba-L ::*i i-igba Ie Iv Il* →*i* (*Iv* →*i Il* →*i i-bool*)
igba-rec-ext ::*i* (*Iv* →*i Il* →*i i-bool*) →*i Ie* →*i* ⟨*Ie*,*Iv*,*Il*⟩*i i-igba-eext*
⟨*proof*⟩
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*
}
⟨*proof*⟩

lemma *gen-igba-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rl*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm,Rl*⟩*gen-igba-impl-rel-eext*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟧
⟹ *single-valued* (⟨*Rm,Rv,Rl*⟩*igba-impl-rel-eext*)
⟨*proof*⟩

lemma *igba-f-tag*: *TERM* (*Rv* → *Rl* → *bool-rel*) ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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* = *V0*, *g-E* = *E*, *g-V0* = *V0*,
    igbg-num-acc = *num-acc*, *igbg-acc* = *acc*, *igba-L* = *L* |)
)∈*?R*
  ⟨*proof*⟩

8.3 Generalized Buchi Graphs

consts

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

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

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *gbg-type*[*autoref-itype*]:
gbg-F ::$_i$ *i-gbg Ie Iv* →$_i$ ⟨⟨*Iv*⟩$_i$*i-set*⟩$_i$*i-set*
gb-graph-rec-ext ::$_i$ ⟨⟨*Iv*⟩$_i$*i-set*⟩$_i$*i-set* →$_i$ *Ie* →$_i$ ⟨*Ie*,*Iv*⟩$_i$*i-gbg-eext*
⟨*proof*⟩
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*
}
⟨*proof*⟩

lemma *gen-gbg-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rf*⟧
⟹ *single-valued* (⟨*Rm*,*Rf*⟩*gen-gbg-impl-rel-eext*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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*,
λ*G x*. *if gbg-F G* = {} *then* (*g-E G* ‘‘ {*x*}) *else* {}
)∈*?R*
⟨*proof*⟩

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*, *gbg-F* = *F* |)∈*?R*
⟨*proof*⟩

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*, *gbg-F* = *insert* {} *F* |)∈*?R*
⟨*proof*⟩

schematic-goal (*?c*::*?'c*, *it-to-sorted-list* (λ- -. *True*) {*1*,*2*::*nat*})∈*?R*
⟨*proof*⟩

8.4 GBAs

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

abbreviation *i-gba Ie Iv Il*
≡ ⟨⟨*Ie*,*Iv*,*Il*⟩*i* *i-gba-eext*,*Iv*⟩*i* *i-gbg-eext*,*Iv*⟩*i* *i-g-ext*
context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *gba-type*[*autoref-itype*]:
gba-L ::*i* *i-gba Ie Iv Il* →*i* (*Iv* →*i* *Il* →*i* *i-bool*)
gba-rec-ext ::*i* (*Iv* →*i* *Il* →*i* *i-bool*) →*i* *Ie* →*i* ⟨*Ie*,*Iv*,*Il*⟩*i* *i-gba-eext*
⟨*proof*⟩
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*
}

⟨*proof*⟩

lemma *gen-gba-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rl*; *single-valued Rm*⟧
⟹ *single-valued* (⟨*Rm*,*Rl*⟩*gen-gba-impl-rel-eext*)
⟨*proof*⟩

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*)
∈ *Rl* → *Rm* → ⟨*Rm*,*Rl*⟩*gen-gba-impl-rel-eext*
⟨*proof*⟩

8.4.1 Implementation as function

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

lemma *gba-impl-rel-eext-def*:
⟨*Rm*,*Rv*,*Rl*⟩*gba-impl-rel-eext* ≡ ⟨*Rm*, *Rv* → *Rl* → *bool-rel*⟩*gen-gba-impl-rel-eext*
⟨*proof*⟩

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

lemma [*relator-props*, *simp*]:
⟦*Range Rv* = *UNIV*; *single-valued Rm*; *Range Rl* = *UNIV*⟧
⟹ *single-valued* (⟨*Rm*,*Rv*,*Rl*⟩*gba-impl-rel-eext*)
⟨*proof*⟩

lemma *gba-f-tag*: *TERM* (*Rv* → *Rl* → *bool-rel*) ⟨*proof*⟩

abbreviation *gbav-impl-rel-ext Rm Rv Rl*
≡ *gbgv-impl-rel-ext* (⟨*Rm*, *Rv*, *Rl*⟩*gba-impl-rel-eext*) *Rv*

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

context

fixes *Rv* :: (*'vi*×*'v*) *set*
fixes *Rl* :: (*'Li*×*'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* {})∈*?R*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

8.5 Buchi Graphs

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

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *bg-type*[*autoref-itype*]:
bg-F ::$_i$ *i-bg Ie Iv* →$_i$ ⟨*Iv*⟩$_i$*i-set*
gb-graph-rec-ext ::$_i$ ⟨⟨*Iv*⟩$_i$*i-set*⟩$_i$*i-set* →$_i$ *Ie* →$_i$ ⟨*Ie*,*Iv*⟩$_i$*i-bg-eext*
⟨*proof*⟩
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*
}
⟨*proof*⟩

lemma *gen-bg-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rf*⟧
⟹ *single-valued* (⟨*Rm*,*Rf*⟩*gen-bg-impl-rel-eext*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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*,
λG x. *if x* ∈ *bg-F G then* (*g-E G* `` {*x*}) *else* {}
)∈*?R*
⟨*proof*⟩

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*, *bg-F* = *F* |)∈*?R*
⟨*proof*⟩

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*, *bg-F* = *F* |)∈*?R*
⟨*proof*⟩

8.6 System Automata

consts

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

abbreviation *i-sa Ie Iv Il* ≡ ⟨⟨*Ie*,*Iv*,*Il*⟩$_i$*i-sa-eext*,*Iv*⟩$_i$*i-g-ext*

context begin interpretation *autoref-syn* ⟨*proof*⟩
term *sa-L*
lemma *sa-type*[*autoref-itype*]:
sa-L ::$_i$ *i-sa Ie Iv Il* →$_i$ *Iv* →$_i$ *Il*
sa-rec-ext ::$_i$ (*Iv* →$_i$ *Il*) →$_i$ *Ie* →$_i$ ⟨*Ie*,*Iv*,*Il*⟩$_i$*i-sa-eext*
⟨*proof*⟩
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*
}
⟨*proof*⟩

lemma *gen-sa-impl-rel-sv*[*relator-props*]:
⟦*single-valued Rm*; *single-valued Rf*⟧
⟹ *single-valued* (⟨*Rm*,*Rf*⟩*gen-sa-impl-rel-eext*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sa-f-tag*: *TERM* (*Rv*→*Rl*) ⟨*proof*⟩

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*,

$\lambda G\ x\ l.$ *if sa-L* $G\ x = l$ *then* (*g-E* G ‘‘ $\{x\}$) *else* $\{\}$
)∈*?R*
⟨*proof*⟩

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: ($'a \times 'a$) *set*]
shows (*?c*::*?'c*, λE (*V0*::$'a$ *set*) *L*.
(‖ *g-V* = $\{\}$, *g-E* = *E*, *g-V0* = *V0*, *sa-L* = *L* ‖)∈*?R*
⟨*proof*⟩

schematic-goal
notes [*autoref-tyrel*] = *TYRELI*[*of Id* :: ($'a \times 'a$) *set*]
shows (*?c*::*?'c*, λE (*V0*::$'a$ *set*) *L*.
(‖ *g-V* = *UNIV*, *g-E* = *E*, *g-V0* = *V0*, *sa-L* = *L* ‖)∈*?R*
⟨*proof*⟩

8.7 Index Conversion

schematic-goal *gbg-to-idx-ext-impl-aux*:
fixes *Re* **and** *Rv* :: ($'qi \times 'q$) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode Rv eq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE*($'qi$) (*def-size*)
shows (*?c*, *gbg-to-idx-ext* :: - ⇒ ($'q$, -) *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*
⟨*proof*⟩
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*
⟨*proof*⟩
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* ⟨*proof*⟩
lemma [*autoref-op-pat*]: *gba-to-idx-ext ecnv* ≡ *OP gba-to-idx-ext* $ *ecnv* ⟨*proof*⟩
end

schematic-goal *gba-to-idx-ext-impl-aux*:
fixes *Re* **and** *Rv* :: ($'qi \times 'q$) *set*

assumes [*autoref-ga-rules*]: *is-bounded-hashcode Rv eq bhc*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE('qi) (def-size)*
shows (*?c*, *gba-to-idx-ext* :: - ⇒ ('*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*
⟨*proof*⟩
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*
⟨*proof*⟩
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*
|))
⟨*proof*⟩

schematic-goal *degeneralize-ext-impl-aux*:
fixes *Re Rv*
assumes [*autoref-rules*]: (*Gi*,*G*) ∈ *igbg-impl-rel-ext Re Rv*
shows (*?c*, *degeneralize-ext*)
∈ (*igbg-impl-rel-ext Re Rv* → *Re'*) → *bg-impl-rel-ext Re'* (*Rv* ×$_r$ *nat-rel*)

⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩
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* ×$_r$ *nat-rel*))$*ecnv*$*G*)
∈ *bg-impl-rel-ext Re′* (*Rv* ×$_r$ *nat-rel*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* {}
⦈)
⟨*proof*⟩

schematic-goal *prod-impl-aux*:
fixes *Re*

assumes [*autoref-rules*]: (*Gi*,*G*) ∈ *igba-impl-rel-ext* *Re* *Rq* *Rl*
assumes [*autoref-rules*]: (*Si*,*S*) ∈ *sa-impl-rel-ext* *Re2* *Rs* *Rl*
shows (*?c*, *prod*) ∈ *igbg-impl-rel-ext* *unit-rel* (*Rq* ×$_r$ *Rs*)
⟨*proof*⟩

end

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

lemma [*autoref-op-pat*]:
igba-sys-prod-precond.prod ≡ *op-igba-sys-prod*
⟨*proof*⟩

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

thm *igba-sys-prod-impl.refine*

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-rules*]:
fixes *Re*
assumes *SIDE-PRECOND* (*igba* *G*)
assumes *SIDE-PRECOND* (*sa* *S*)
assumes *GR*: (*Gi*,*G*)∈*igba-impl-rel-ext* *unit-rel* *Rq* *Rl*
assumes *SR*: (*Si*,*S*)∈*sa-impl-rel-ext* *unit-rel* *Rs* *Rl*
shows (*igba-sys-prod-impl* *Gi* *Si*,
(*OP* *op-igba-sys-prod*
::: *igba-impl-rel-ext* *unit-rel* *Rq* *Rl*
→ *sa-impl-rel-ext* *unit-rel* *Rs* *Rl*
→ *igbg-impl-rel-ext* *unit-rel* (*Rq* ×$_r$ *Rs*))$*G*$*S*)
∈ *igbg-impl-rel-ext* *unit-rel* (*Rq* ×$_r$ *Rs*)
⟨*proof*⟩

end

schematic-goal
assumes [*simp*]: *igba G sa S*
assumes [*autoref-rules*]: (*Gi*,*G*)∈*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*
⟨*proof*⟩

end