Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm

Peter Lammich

March 17, 2025

Abstract

We present an Isabelle/HOL formalization of Gabow's algorithm for finding the strongly connected components of a directed graph. Using data refinement techniques, we extract efficient code that performs comparable to a reference implementation in Java. Our style of formalization allows for re-using large parts of the proofs when defining variants of the algorithm. We demonstrate this by verifying an algorithm for the emptiness check of generalized Büchi automata, re-using most of the existing proofs.

Contents

1 Introduction

A strongly connected component (SCC) of a directed graph is a maximal subset of mutually reachable nodes. Finding the SCCs is a standard problem from graph theory with applications in many fields ([19, Chap. 4.2]).

This formalization accompanies our conference paper [11], where we describe the used formalization techniques.

There are several algorithms to partition the nodes of a graph into SCCs, the main ones being the Kosaraju-Sharir algorithm[20], Tarjan's algorithm[21], and the class of path-based algorithms[17, 15, 3, 1, 5].

In this formalization, we present the verification of Gabow's path-based SCC-algorithm[5] within the theorem prover Isabelle/HOL[16]. Using refinement techniques and a collection of efficient verified data structures, we extract Standard ML (SML)[14] code from the formalization. Our verified algorithm has a performance comparable to a reference implementation in Java, taken from Sedgewick and Wayne's textbook on algorithms[19, Chap. 4.2].

Our main interest in SCC-algorithms stems from the fact that they can be used for the emptiness check of generalized Büchi automata (GBA), a problem that arises in LTL model checking[22, 6, 2]. Towards this end, we extend the algorithm to check the emptiness of generalized Büchi automata, re-using many of the proofs from the original verification.

Contributions and Related Work Up to our knowledge, we present the first mechanically verified SCC-algorithm, as well as the first mechanically verified SCC-based emptiness check for GBA. Path-based algorithms have already been regarded for the emptiness check of GBAs[18]. However, we are the first to use the data structure proposed by Gabow[5].[1] Finally, our development is a case study for using the Isabelle/HOL Monadic Refinement and Collection Frameworks[12, 13, 9, 10] to engineer a verified, efficient implementation of a quite complex algorithm, while keeping proofs modular and re-usable.

This development is part of the CAVA project[4] to produce a fully verified LTL model checker.

Outline The rest of this formalization is organized as follows: In Section 1.1, we define a skeleton algorithm and show preservation of some general-purpose invariants. In Section 1.5, we define and prove correct an algorithm that takes a directed graph and computes a list of SCCs in topological order. In Section 1.12 we provide a simple safety property mod-

[1] Although called Gabow-based algorithm in [18], a union-find data structure is used to implement collapsing of nodes, while Gabow proposes a different data structure[5, pg. 109]

elchecker, which tries to find a path to a node violating a given property in a graph. This is used in Section 1.14, where we define an algorithm that checks the language of a given generalized Büchi graph[2] (GBG) for emptiness, and returns a counterexample in case of non-emptiness. In the next three sections (1.23, 1.27, 1.31,1.36) we use the Autoref Tool[10] to refine the above algorithms to efficient data structures, and extract SML code using Isabelle/HOL's code generator[7, 8].

1.1 Skeleton for Gabow's SCC Algorithm

theory *Gabow-Skeleton*
imports *CAVA-Automata.Digraph*
begin

locale *fr-graph* =
  graph G
  for *G* :: ('*v*, '*more*) *graph-rec-scheme*
  +
  assumes *finite-reachableE-V0*[*simp*, *intro*!]: *finite* (*E** `` *V0*)

In this theory, we formalize a skeleton of Gabow's SCC algorithm. The skeleton serves as a starting point to develop concrete algorithms, like enumerating the SCCs or checking emptiness of a generalized Büchi automaton.

1.2 Statistics Setup

We define some dummy-constants that are included into the generated code, and may be mapped to side-effecting ML-code that records statistics and debug information about the execution. In the skeleton algorithm, we count the number of visited nodes, and include a timing for the whole algorithm.

definition *stat-newnode* :: *unit* => *unit* — Invoked if new node is visited
  where [*code*]: *stat-newnode* ≡ λ-. ()

definition *stat-start* :: *unit* => *unit* — Invoked once if algorithm starts
  where [*code*]: *stat-start* ≡ λ-. ()

definition *stat-stop* :: *unit* => *unit* — Invoked once if algorithm stops
  where [*code*]: *stat-stop* ≡ λ-. ()

lemma [*autoref-rules*]:
  (*stat-newnode*,*stat-newnode*) ∈ *unit-rel* → *unit-rel*
  (*stat-start*,*stat-start*) ∈ *unit-rel* → *unit-rel*
  (*stat-stop*,*stat-stop*) ∈ *unit-rel* → *unit-rel*
  ⟨*proof*⟩

[2] GBGs are generalized Büchi automata without labels.

abbreviation *stat-newnode-nres* ≡ *RETURN* (*stat-newnode* ())
abbreviation *stat-start-nres* ≡ *RETURN* (*stat-start* ())
abbreviation *stat-stop-nres* ≡ *RETURN* (*stat-stop* ())

lemma *discard-stat-refine*[*refine*]:
m1≤*m2* ⟹ *stat-newnode-nres* ≫ *m1* ≤ *m2*
m1≤*m2* ⟹ *stat-start-nres* ≫ *m1* ≤ *m2*
m1≤*m2* ⟹ *stat-stop-nres* ≫ *m1* ≤ *m2*
⟨*proof*⟩

1.3 Abstract Algorithm

In this section, we formalize an abstract version of a path-based SCC algorithm. Later, this algorithm will be refined to use Gabow's data structure.

1.3.1 Preliminaries

definition *path-seg* :: *'a set list* ⇒ *nat* ⇒ *nat* ⇒ *'a set*
— Set of nodes in a segment of the path
where *path-seg p i j* ≡ ⋃{*p*!*k*|*k*. *i*≤*k* ∧ *k*<*j*}

lemma *path-seg-simps*[*simp*]:
j≤*i* ⟹ *path-seg p i j* = {}
path-seg p i (*Suc i*) = *p*!*i*
⟨*proof*⟩

lemma *path-seg-drop*:
⋃(*set* (*drop i p*)) = *path-seg p i* (*length p*)
⟨*proof*⟩

lemma *path-seg-butlast*:
p≠[] ⟹ *path-seg p 0* (*length p* − *Suc 0*) = ⋃(*set* (*butlast p*))
⟨*proof*⟩

definition *idx-of* :: *'a set list* ⇒ *'a* ⇒ *nat*
— Index of path segment that contains a node
where *idx-of p v* ≡ *THE i*. *i*<*length p* ∧ *v*∈*p*!*i*

lemma *idx-of-props*:
assumes
p-disjoint-sym: ∀ *i j v*. *i*<*length p* ∧ *j*<*length p* ∧ *v*∈*p*!*i* ∧ *v*∈*p*!*j* ⟶ *i*=*j*
assumes *ON-STACK*: *v*∈⋃(*set p*)
shows
idx-of p v < *length p* **and**
v ∈ *p* ! *idx-of p v*
⟨*proof*⟩

lemma *idx-of-uniq*:

assumes
p-disjoint-sym: $\forall i\ j\ v.\ i<length\ p \wedge j<length\ p \wedge v\in p!i \wedge v\in p!j \longrightarrow i=j$
assumes *A*: $i<length\ p \quad v\in p!i$
shows *idx-of p v* = *i*
⟨*proof*⟩

1.3.2 Invariants

The state of the inner loop consists of the path p of collapsed nodes, the set D of finished (done) nodes, and the set pE of pending edges.

type-synonym *'v abs-state* = *'v set list* × *'v set* × (*'v*×*'v*) *set*

context *fr-graph*
begin
definition *touched* :: *'v set list* ⇒ *'v set* ⇒ *'v set*
— Touched: Nodes that are done or on path
where *touched p D* ≡ $D \cup \bigcup (set\ p)$

definition *vE* :: *'v set list* ⇒ *'v set* ⇒ (*'v* × *'v*) *set* ⇒ (*'v* × *'v*) *set*
— Visited edges: No longer pending edges from touched nodes
where *vE p D pE* ≡ (*E* ∩ (*touched p D* × *UNIV*)) − *pE*

lemma *vE-ss-E*: *vE p D pE* ⊆ *E* — Visited edges are edges
⟨*proof*⟩

end

locale *outer-invar-loc* — Invariant of the outer loop
= *fr-graph G* **for** *G* :: (*'v*,*'more*) *graph-rec-scheme* +
fixes *it* :: *'v set* — Remaining nodes to iterate over
fixes *D* :: *'v set* — Finished nodes

assumes *it-initial*: *it*⊆*V0* — Only start nodes to iterate over

assumes *it-done*: *V0* − *it* ⊆ *D* — Nodes already iterated over are visited
assumes *D-reachable*: *D*⊆*E** ‘‘*V0* — Done nodes are reachable
assumes *D-closed*: *E*‘‘*D* ⊆ *D* — Done is closed under transitions
begin

lemma *locale-this*: *outer-invar-loc G it D* ⟨*proof*⟩

definition (**in** *fr-graph*) *outer-invar* ≡ λ*it D*. *outer-invar-loc G it D*

lemma *outer-invar-this*[*simp*, *intro*!]: *outer-invar it D*
⟨*proof*⟩
end

locale *invar-loc* — Invariant of the inner loop
= *fr-graph G*

for $G :: ('v, 'more)$ *graph-rec-scheme* +
fixes $v0 :: 'v$
fixes $D0 :: 'v\ set$
fixes $p :: 'v\ set\ list$
fixes $D :: 'v\ set$
fixes $pE :: ('v \times 'v)\ set$

assumes *v0-initial*[*simp*, *intro*!]: $v0 \in V0$
assumes *D-incr*: $D0 \subseteq D$

assumes *pE-E-from-p*: $pE \subseteq E \cap (\bigcup (set\ p)) \times UNIV$
— Pending edges are edges from path
assumes *E-from-p-touched*: $E \cap (\bigcup (set\ p) \times UNIV) \subseteq pE \cup UNIV \times touched\ p\ D$
— Edges from path are pending or touched
assumes *D-reachable*: $D \subseteq E^* ``V0$ — Done nodes are reachable
assumes *p-connected*: $Suc\ i < length\ p \Longrightarrow p!i \times p!Suc\ i \cap (E - pE) \neq \{\}$
— CNodes on path are connected by non-pending edges

assumes *p-disjoint*: $[\![i<j; j<length\ p]\!] \Longrightarrow p!i \cap p!j = \{\}$
— CNodes on path are disjoint
assumes *p-sc*: $U \in set\ p \Longrightarrow U \times U \subseteq (vE\ p\ D\ pE \cap U \times U)^*$
— Nodes in CNodes are mutually reachable by visited edges

assumes *root-v0*: $p \neq [] \Longrightarrow v0 \in hd\ p$ — Root CNode contains start node
assumes *p-empty-v0*: $p = [] \Longrightarrow v0 \in D$ — Start node is done if path empty

assumes *D-closed*: $E ``D \subseteq D$ — Done is closed under transitions

assumes *vE-no-back*: $[\![i<j; j<length\ p]\!] \Longrightarrow vE\ p\ D\ pE \cap p!j \times p!i = \{\}$
— Visited edges do not go back on path
assumes *p-not-D*: $\bigcup (set\ p) \cap D = \{\}$ — Path does not contain done nodes
begin
abbreviation *ltouched* **where** $ltouched \equiv touched\ p\ D$
abbreviation *lvE* **where** $lvE \equiv vE\ p\ D\ pE$

lemma *locale-this*: *invar-loc G v0 D0 p D pE* ⟨*proof*⟩

definition (**in** *fr-graph*)
$invar \equiv \lambda v0\ D0\ (p,D,pE).\ invar\text{-}loc\ G\ v0\ D0\ p\ D\ pE$

lemma *invar-this*[*simp*, *intro*!]: *invar v0 D0* (p,D,pE)
⟨*proof*⟩

lemma *finite-reachableE-v0*[*simp*, *intro*!]: *finite* $(E^* ``\{v0\})$
⟨*proof*⟩

lemma *D-vis*: $E \cap D \times UNIV \subseteq lvE$ — All edges from done nodes are visited

⟨*proof*⟩

lemma *vE-touched*: $lvE \subseteq ltouched \times ltouched$
— Visited edges only between touched nodes
⟨*proof*⟩

lemma *lvE-ss-E*: $lvE \subseteq E$ — Visited edges are edges
⟨*proof*⟩

lemma *path-touched*: $\bigcup(set\ p) \subseteq ltouched$ ⟨*proof*⟩
lemma *D-touched*: $D \subseteq ltouched$ ⟨*proof*⟩

lemma *pE-by-vE*: $pE = (E \cap \bigcup(set\ p) \times UNIV) - lvE$
— Pending edges are edges from path not yet visited
⟨*proof*⟩

lemma *pick-pending*: $p \neq [] \Longrightarrow pE \cap last\ p \times UNIV = (E - lvE) \cap last\ p \times UNIV$
— Pending edges from end of path are non-visited edges from end of path
⟨*proof*⟩

lemma *p-connected'*:
assumes *A*: $Suc\ i < length\ p$
shows $p!i \times p!Suc\ i \cap lvE \neq \{\}$
⟨*proof*⟩

end

Termination **context** *fr-graph*
begin

The termination argument is based on unprocessed edges: Reachable edges from untouched nodes and pending edges.

definition *unproc-edges v0 p D pE* $\equiv (E \cap (E^* `` \{v0\} - (D \cup \bigcup(set\ p))) \times UNIV) \cup pE$

In each iteration of the loop, either the number of unprocessed edges decreases, or the path length decreases.

definition *abs-wf-rel v0* $\equiv$ *inv-image* (*finite-psubset* <∗*lex*∗> *measure length*) (λ(*p*,*D*,*pE*). (*unproc-edges v0 p D pE*, *p*))

lemma *abs-wf-rel-wf*[*simp*, *intro*!]: *wf* (*abs-wf-rel v0*)
⟨*proof*⟩
end

1.3.3 Abstract Skeleton Algorithm

context *fr-graph*
begin

```
definition (in fr-graph) initial :: 'v ⇒ 'v set ⇒ 'v abs-state
  where initial v0 D ≡ ([{v0}], D, (E ∩ {v0}×UNIV))

definition (in −) collapse-aux :: 'a set list ⇒ nat ⇒ 'a set list
  where collapse-aux p i ≡ take i p @ [⋃(set (drop i p))]

definition (in −) collapse :: 'a ⇒ 'a abs-state ⇒ 'a abs-state
  where collapse v PDPE ≡
  let
    (p,D,pE)=PDPE;
    i=idx-of p v;
    p = collapse-aux p i
  in (p,D,pE)

definition (in −)
  select-edge :: 'a abs-state ⇒ ('a option × 'a abs-state) nres
  where
  select-edge PDPE ≡ do {
    let (p,D,pE) = PDPE;
    e ← SELECT (λe. e ∈ pE ∩ last p × UNIV);
    case e of
      None ⇒ RETURN (None,(p,D,pE))
    | Some (u,v) ⇒ RETURN (Some v, (p,D,pE − {(u,v)}))
  }

definition (in fr-graph) push :: 'v ⇒ 'v abs-state ⇒ 'v abs-state
  where push v PDPE ≡
  let
    (p,D,pE) = PDPE;
    p = p@[{v}];
    pE = pE ∪ (E∩{v}×UNIV)
  in
    (p,D,pE)

definition (in −) pop :: 'v abs-state ⇒ 'v abs-state
  where pop PDPE ≡ let
    (p,D,pE) = PDPE;
    (p,V) = (butlast p, last p);
    D = V ∪ D
  in
    (p,D,pE)
```

The following lemmas match the definitions presented in the paper:

```
lemma select-edge (p,D,pE) ≡ do {
    e ← SELECT (λe. e ∈ pE ∩ last p × UNIV);
    case e of
      None ⇒ RETURN (None,(p,D,pE))
    | Some (u,v) ⇒ RETURN (Some v, (p,D,pE − {(u,v)}))
```

```
  }
  ⟨proof⟩

lemma collapse v (p,D,pE)
  ≡ let i=idx-of p v in (take i p @ [⋃(set (drop i p))],D,pE)
  ⟨proof⟩

lemma push v (p, D, pE) ≡ (p @ [{v}], D, pE ∪ E ∩ {v} × UNIV)
  ⟨proof⟩

lemma pop (p, D, pE) ≡ (butlast p, last p ∪ D, pE)
  ⟨proof⟩

thm pop-def[unfolded Let-def, no-vars]

thm select-edge-def[unfolded Let-def]


definition skeleton :: 'v set nres
  — Abstract Skeleton Algorithm
  where
  skeleton ≡ do {
    let D = {};
    r ← FOREACHi outer-invar V0 (λv0 D0. do {
      if v0∉D0 then do {
        let s = initial v0 D0;

        (p,D,pE) ← WHILEIT (invar v0 D0)
          (λ(p,D,pE). p ≠ []) (λ(p,D,pE).
        do {
          — Select edge from end of path
          (vo,(p,D,pE)) ← select-edge (p,D,pE);

          ASSERT (p≠[]);
          case vo of
            Some v ⇒ do { — Found outgoing edge to node v
              if v ∈ ⋃(set p) then do {
                — Back edge: Collapse path
                RETURN (collapse v (p,D,pE))
              } else if v∉D then do {
                — Edge to new node. Append to path
                RETURN (push v (p,D,pE))
              } else do {
                — Edge to done node. Skip
                RETURN (p,D,pE)
              }
            }
          | None ⇒ do {
              ASSERT (pE ∩ last p × UNIV = {});
```

```
            — No more outgoing edges from current node on path
            RETURN (pop (p,D,pE))
          }
        }) s;
        ASSERT (p=[] ∧ pE={});
        RETURN D
      } else
        RETURN D0
    }) D;
    RETURN r
  }
```

end

1.3.4 Invariant Preservation

context *fr-graph* **begin**

lemma *set-collapse-aux*[*simp*]: $\bigcup(set\ (collapse\text{-}aux\ p\ i)) = \bigcup(set\ p)$
⟨*proof*⟩

lemma *touched-collapse*[*simp*]: *touched* (*collapse-aux p i*) *D* = *touched p D*
⟨*proof*⟩

lemma *vE-collapse-aux*[*simp*]: *vE* (*collapse-aux p i*) *D pE* = *vE p D pE*
⟨*proof*⟩

lemma *touched-push*[*simp*]: *touched* (*p* @ [*V*]) *D* = *touched p D* ∪ *V*
⟨*proof*⟩

end

Corollaries of the invariant In this section, we prove some more corollaries of the invariant, which are helpful to show invariant preservation

context *invar-loc*
begin
lemma *cnode-connectedI*:
$[\![i<length\ p;\ u\in p!i;\ v\in p!i]\!] \Longrightarrow (u,v)\in(lvE \cap p!i\times p!i)^*$
⟨*proof*⟩

lemma *cnode-connectedI′*: $[\![i<length\ p;\ u\in p!i;\ v\in p!i]\!] \Longrightarrow (u,v)\in(lvE)^*$
⟨*proof*⟩

lemma *p-no-empty*: $\{\} \notin set\ p$
⟨*proof*⟩

corollary *p-no-empty-idx*: $i<length\ p \Longrightarrow p!i\neq\{\}$
⟨*proof*⟩

lemma *p-disjoint-sym*: $[\![i<length\ p;\ j<length\ p;\ v\in p!i;\ v\in p!j]\!] \Longrightarrow i=j$
⟨*proof*⟩

lemma *pi-ss-path-seg-eq*[*simp*]:
assumes *A*: $i<length\ p \quad u\leq length\ p$
shows $p!i\subseteq path\text{-}seg\ p\ l\ u \longleftrightarrow l\leq i \wedge i<u$
⟨*proof*⟩

lemma *path-seg-ss-eq*[*simp*]:
assumes *A*: $l1<u1 \quad u1\leq length\ p \quad l2<u2 \quad u2\leq length\ p$
shows $path\text{-}seg\ p\ l1\ u1 \subseteq path\text{-}seg\ p\ l2\ u2 \longleftrightarrow l2\leq l1 \wedge u1\leq u2$
⟨*proof*⟩

lemma *pathI*:
assumes $x\in p!i \quad y\in p!j$
assumes $i\leq j \quad j<length\ p$
defines $seg \equiv path\text{-}seg\ p\ i\ (Suc\ j)$
shows $(x,y)\in(lvE \cap seg\times seg)^*$
— We can obtain a path between cnodes on path
⟨*proof*⟩

lemma *p-reachable*: $\bigcup(set\ p) \subseteq E^*``\{v0\}$ — Nodes on path are reachable
⟨*proof*⟩

lemma *touched-reachable*: $ltouched \subseteq E^*``V0$ — Touched nodes are reachable
⟨*proof*⟩

lemma *vE-reachable*: $lvE \subseteq E^*``V0 \times E^*``V0$
⟨*proof*⟩

lemma *pE-reachable*: $pE \subseteq E^*``\{v0\} \times E^*``\{v0\}$
⟨*proof*⟩

lemma *D-closed-vE-rtrancl*: $lvE^*``D \subseteq D$
⟨*proof*⟩

lemma *D-closed-path*: $[\![path\ E\ u\ q\ w;\ u\in D]\!] \Longrightarrow set\ q \subseteq D$
⟨*proof*⟩

lemma *D-closed-path-vE*: $[\![path\ lvE\ u\ q\ w;\ u\in D]\!] \Longrightarrow set\ q \subseteq D$
⟨*proof*⟩

lemma *path-in-lastnode*:
assumes *P*: *path lvE u q v*
assumes [*simp*]: $p\neq[]$
assumes *ND*: $u\in last\ p \quad v\in last\ p$
shows $set\ q \subseteq last\ p$
— A path from the last Cnode to the last Cnode remains in the last Cnode

⟨*proof*⟩

lemma *loop-in-lastnode*:
assumes *P*: *path lvE u q u*
assumes [*simp*]: *p*≠[]
assumes *ND*: *set q* ∩ *last p* ≠ {}
shows *u*∈*last p* **and** *set q* ⊆ *last p*
— A loop that touches the last node is completely inside the last node
⟨*proof*⟩

lemma *no-D-p-edges*: *E* ∩ *D* × ⋃(*set p*) = {}
⟨*proof*⟩

lemma *idx-of-props*:
assumes *ON-STACK*: *v*∈⋃(*set p*)
shows
idx-of p v < *length p* **and**
v ∈ *p* ! *idx-of p v*
⟨*proof*⟩

end

Auxiliary Lemmas Regarding the Operations **lemma** (**in** *fr-graph*) *vE-initial*[*simp*]: *vE* [{*v0*}] {} (*E* ∩ {*v0*} × *UNIV*) = {}
⟨*proof*⟩

context *invar-loc*
begin
lemma *vE-push*: ⟦ (*u*,*v*)∈*pE*; *u*∈*last p*; *v*∉⋃(*set p*); *v*∉*D* ⟧
⟹ *vE* (*p* @ [{*v*}]) *D* ((*pE* − {(*u*,*v*)}) ∪ *E*∩{*v*}×*UNIV*) = *insert* (*u*,*v*) *lvE*
⟨*proof*⟩

lemma *vE-remove*[*simp*]:
⟦*p*≠[]; (*u*,*v*)∈*pE*⟧ ⟹ *vE p D* (*pE* − {(*u*,*v*)}) = *insert* (*u*,*v*) *lvE*
⟨*proof*⟩

lemma *vE-pop*[*simp*]: *p*≠[] ⟹ *vE* (*butlast p*) (*last p* ∪ *D*) *pE* = *lvE*
⟨*proof*⟩

lemma *pE-fin*: *p*=[] ⟹ *pE*={}
⟨*proof*⟩

lemma (**in** *invar-loc*) *lastp-un-D-closed*:
assumes *NE*: *p* ≠ []
assumes *NO'*: *pE* ∩ (*last p* × *UNIV*) = {}
shows *E*‘‘(*last p* ∪ *D*) ⊆ (*last p* ∪ *D*)
— On pop, the popped CNode and D are closed under transitions

⟨*proof*⟩

end

Preservation of Invariant by Operations **context** *fr-graph*
begin
lemma (**in** *outer-invar-loc*) *invar-initial-aux*:
assumes *v0*∈*it* − *D*
shows *invar v0 D* (*initial v0 D*)
⟨*proof*⟩

lemma *invar-initial*:
⟦*outer-invar it D0*; *v0*∈*it*; *v0*∉*D0*⟧ ⟹ *invar v0 D0* (*initial v0 D0*)
⟨*proof*⟩

lemma *outer-invar-initial*[*simp*, *intro*!]: *outer-invar V0* {}
⟨*proof*⟩

lemma *invar-pop*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *NO'*: *pE* ∩ (*last p* × *UNIV*) = {}
shows *invar v0 D0* (*pop* (*p*,*D*,*pE*))
⟨*proof*⟩

thm *invar-pop*[*of v-0 D-0*, *no-vars*]

lemma *invar-collapse*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* **and** *u*∈*last p*
assumes *BACK*: *v*∈⋃(*set p*)
defines *i* ≡ *idx-of p v*
defines *p'* ≡ *collapse-aux p i*
shows *invar v0 D0* (*collapse v* (*p*,*D*,*pE* − {(*u*,*v*)}))
⟨*proof*⟩

lemma *invar-push*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* **and** *UIL*: *u*∈*last p*
assumes *VNE*: *v*∉⋃(*set p*) *v*∉*D*
shows *invar v0 D0* (*push v* (*p*,*D*,*pE* − {(*u*,*v*)}))
⟨*proof*⟩

lemma *invar-skip*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)

assumes *NE*[*simp*]: $p \neq []$
assumes *E*: $(u,v) \in pE$ **and** *UIL*: $u \in last\ p$
assumes *VNP*: $v \notin \bigcup (set\ p)$ **and** *VD*: $v \in D$
shows *invar v0 D0* $(p, D, pE - \{(u, v)\})$
⟨*proof*⟩

lemma *fin-D-is-reachable*:
— When inner loop terminates, all nodes reachable from start node are finished
assumes *INV*: *invar v0 D0* ([], *D*, *pE*)
shows $D \supseteq E^* ``\{v0\}$
⟨*proof*⟩

lemma *fin-reachable-path*:
— When inner loop terminates, nodes reachable from start node are reachable over visited edges
assumes *INV*: *invar v0 D0* ([], *D*, *pE*)
assumes *UR*: $u \in E^* ``\{v0\}$
shows *path* (*vE* [] *D pE*) *u q v* $\longleftrightarrow$ *path E u q v*
⟨*proof*⟩

lemma *invar-outer-newnode*:
assumes *A*: $v0 \notin D0 \quad v0 \in it$
assumes *OINV*: *outer-invar it D0*
assumes *INV*: *invar v0 D0* ([], D', *pE*)
shows *outer-invar* $(it - \{v0\})$ D'
⟨*proof*⟩

lemma *invar-outer-Dnode*:
assumes *A*: $v0 \in D0 \quad v0 \in it$
assumes *OINV*: *outer-invar it D0*
shows *outer-invar* $(it - \{v0\})$ *D0*
⟨*proof*⟩

lemma *pE-fin'*: *invar x σ* ([], *D*, *pE*) $\Longrightarrow$ $pE = \{\}$
⟨*proof*⟩

end

Termination **context** *invar-loc*
begin
lemma *unproc-finite*[*simp*, *intro*!]: *finite* (*unproc-edges v0 p D pE*)
— The set of unprocessed edges is finite
⟨*proof*⟩

lemma *unproc-decreasing*:
— As effect of selecting a pending edge, the set of unprocessed edges decreases
assumes [*simp*]: $p \neq []$ **and** *A*: $(u,v) \in pE \quad u \in last\ p$
shows *unproc-edges v0 p D* $(pE - \{(u,v)\}) \subset$ *unproc-edges v0 p D pE*

⟨*proof*⟩
end

context *fr-graph*
begin

lemma *abs-wf-pop*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *NO*: *pE* ∩ *last aba* × *UNIV* = {}
shows (*pop* (*p*,*D*,*pE*), (*p*, *D*, *pE*)) ∈ *abs-wf-rel v0*
⟨*proof*⟩

lemma *abs-wf-collapse*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* *u*∈*last p*
shows (*collapse v* (*p*,*D*,*pE*−{(*u*,*v*)}), (*p*, *D*, *pE*))∈ *abs-wf-rel v0*
⟨*proof*⟩

lemma *abs-wf-push*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* *u*∈*last p* **and** *A*: *v*∉*D* *v*∉⋃(*set p*)
shows (*push v* (*p*,*D*,*pE*−{(*u*,*v*)}), (*p*, *D*, *pE*)) ∈ *abs-wf-rel v0*
⟨*proof*⟩

lemma *abs-wf-skip*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* *u*∈*last p*
shows ((*p*, *D*, *pE*−{(*u*,*v*)}), (*p*, *D*, *pE*)) ∈ *abs-wf-rel v0*
⟨*proof*⟩
end

Main Correctness Theorem **context** *fr-graph*
begin
lemmas *invar-preserve* =
invar-initial
invar-pop invar-push invar-skip invar-collapse
abs-wf-pop abs-wf-collapse abs-wf-push abs-wf-skip
outer-invar-initial invar-outer-newnode invar-outer-Dnode

The main correctness theorem for the dummy-algorithm just states that it satisfies the invariant when finished, and the path is empty.

theorem *skeleton-spec*: *skeleton* ≤ *SPEC* (λ*D*. *outer-invar* {} *D*)
⟨*proof*⟩

Short proof, as presented in the paper

context
notes [*refine*] = *refine-vcg*
begin
theorem *skeleton* ≤ *SPEC* (λ*D*. *outer-invar* {} *D*)
⟨*proof*⟩
end

end

1.3.5 Consequences of Invariant when Finished

context *fr-graph*
begin
lemma *fin-outer-D-is-reachable*:
— When outer loop terminates, exactly the reachable nodes are finished
assumes *INV*: *outer-invar* {} *D*
shows *D* = *E** ``*V0*
⟨*proof*⟩

end

1.4 Refinement to Gabow's Data Structure

The implementation due to Gabow [5] represents a path as a stack S of single nodes, and a stack B that contains the boundaries of the collapsed segments. Moreover, a map I maps nodes to their stack indices.

As we use a tail-recursive formulation, we use another stack $P :: (nat \times 'v\ set)\ list$ to represent the pending edges. The entries in P are sorted by ascending first component, and P only contains entries with non-empty second component. An entry (i,l) means that the edges from the node at $S[i]$ to the nodes stored in l are pending.

1.4.1 Preliminaries

primrec *find-max-nat* :: *nat* ⇒ (*nat*⇒*bool*) ⇒ *nat*
— Find the maximum number below an upper bound for which a predicate holds
where
find-max-nat 0 - = 0
| *find-max-nat* (*Suc n*) *P* = (*if* (*P n*) *then n else find-max-nat n P*)

lemma *find-max-nat-correct*:
⟦*P 0*; *0*<*u*⟧ ⟹ *find-max-nat u P* = *Max* {*i*. *i*<*u* ∧ *P i*}
⟨*proof*⟩

lemma *find-max-nat-param*[*param*]:
assumes (*n*,*n'*)∈*nat-rel*
assumes ⋀*j j'*. ⟦(*j*,*j'*)∈*nat-rel*; *j'*<*n'*⟧ ⟹ (*P j*,*P' j'*)∈*bool-rel*
shows (*find-max-nat n P*,*find-max-nat n' P'*) ∈ *nat-rel*

⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *find-max-nat-autoref*[*autoref-rules*]:
assumes $(n,n')\in$*nat-rel*
assumes $\bigwedge j\ j'.\ [\![(j,j')\in$*nat-rel*; $j'<n']\!] \Longrightarrow (P\ j, P'\$j')\in$*bool-rel*
shows (*find-max-nat n P*,
(*OP find-max-nat* ::: *nat-rel* → (*nat-rel*→*bool-rel*) → *nat-rel*) n'P'
) ∈ *nat-rel*
⟨*proof*⟩

end

1.4.2 Gabow's Datastructure

Definition and Invariant **datatype** *node-state* = *STACK nat* | *DONE*

type-synonym *'v oGS* = *'v* ⇀ *node-state*

definition *oGS-α* :: *'v oGS* ⇒ *'v set* **where** *oGS-α I* ≡ {*v*. *I v* = *Some DONE*}

locale *oGS-invar* =
fixes *I* :: *'v oGS*
assumes *I-no-stack*: *I v* ≠ *Some* (*STACK j*)

type-synonym *'a GS*
= *'a list* × *nat list* × (*'a* ⇀ *node-state*) × (*nat* × *'a set*) *list*
locale *GS* =
fixes *SBIP* :: *'a GS*
begin
definition *S* ≡ (λ(*S*,*B*,*I*,*P*). *S*) *SBIP*
definition *B* ≡ (λ(*S*,*B*,*I*,*P*). *B*) *SBIP*
definition *I* ≡ (λ(*S*,*B*,*I*,*P*). *I*) *SBIP*
definition *P* ≡ (λ(*S*,*B*,*I*,*P*). *P*) *SBIP*

definition *seg-start* :: *nat* ⇒ *nat* — Start index of segment, inclusive
where *seg-start i* ≡ *B*!*i*

definition *seg-end* :: *nat* ⇒ *nat* — End index of segment, exclusive
where *seg-end i* ≡ *if i*+*1* = *length B then length S else B*!(*i*+*1*)

definition *seg* :: *nat* ⇒ *'a set* — Collapsed set at index
where *seg i* ≡ {*S*!*j* | *j*. *seg-start i* ≤ *j* ∧ *j* < *seg-end i* }

definition *p-α* ≡ *map seg* [*0*..<*length B*] — Collapsed path

definition *D-α* ≡ {*v*. *I v* = *Some DONE*} — Done nodes

definition *pE-α* ≡ { (*u*,*v*) . ∃ *j I*. (*j*,*I*)∈*set P* ∧ *u* = *S*!*j* ∧ *v*∈*I* }
— Pending edges

definition *α* ≡ (*p-α*,*D-α*,*pE-α*) — Abstract state

end

lemma *GS-sel-simps*[*simp*]:
GS.S (*S*,*B*,*I*,*P*) = *S*
GS.B (*S*,*B*,*I*,*P*) = *B*
GS.I (*S*,*B*,*I*,*P*) = *I*
GS.P (*S*,*B*,*I*,*P*) = *P*
⟨*proof*⟩

context *GS* **begin**
lemma *seg-start-indep*[*simp*]: *GS.seg-start* (*S*′,*B*,*I*′,*P*′) = *seg-start*
⟨*proof*⟩
lemma *seg-end-indep*[*simp*]: *GS.seg-end* (*S*,*B*,*I*′,*P*′) = *seg-end*
⟨*proof*⟩
lemma *seg-indep*[*simp*]: *GS.seg* (*S*,*B*,*I*′,*P*′) = *seg*
⟨*proof*⟩
lemma *p-α-indep*[*simp*]: *GS.p-α* (*S*,*B*,*I*′,*P*′) = *p-α*
⟨*proof*⟩

lemma *D-α-indep*[*simp*]: *GS.D-α* (*S*′,*B*′,*I*,*P*′) = *D-α*
⟨*proof*⟩

lemma *pE-α-indep*[*simp*]: *GS.pE-α* (*S*,*B*′,*I*′,*P*) = *pE-α*
⟨*proof*⟩

definition *find-seg* — Abs-path index for stack index
where *find-seg j* ≡ *Max* {*i*. *i*<*length B* ∧ *B*!*i*≤*j*}

definition *S-idx-of* — Stack index for node
where *S-idx-of v* ≡ *case I v of Some* (*STACK i*) ⇒ *i*

end

locale *GS-invar* = *GS* +
assumes *B-in-bound*: *set B* ⊆ {*0*..<*length S*}
assumes *B-sorted*: *sorted B*
assumes *B-distinct*: *distinct B*
assumes *B0*: *S*≠[] ⟹ *B*≠[] ∧ *B*!*0*=*0*
assumes *S-distinct*: *distinct S*

assumes *I-consistent*: (*I v* = *Some* (*STACK j*)) ⟷ (*j*<*length S* ∧ *v* = *S*!*j*)

assumes *P-sorted*: *sorted* (*map fst P*)
assumes *P-distinct*: *distinct* (*map fst P*)

assumes *P-bound*: *set P* $\subseteq \{0..<length\ S\} \times Collect\ ((\neq)\ \{\})$
begin
lemma *locale-this*: *GS-invar SBIP* ⟨*proof*⟩

end

definition *oGS-rel* ≡ *br oGS-α oGS-invar*
lemma *oGS-rel-sv*[*intro*!,*simp*,*relator-props*]: *single-valued oGS-rel*
⟨*proof*⟩

definition *GS-rel* ≡ *br GS.α GS-invar*
lemma *GS-rel-sv*[*intro*!,*simp*,*relator-props*]: *single-valued GS-rel*
⟨*proof*⟩

context *GS-invar*
begin
lemma *empty-eq*: *S*=[] ⟷ *B*=[]
⟨*proof*⟩

lemma *B-in-bound'*: *i*<*length B* ⟹ *B*!*i* < *length S*
⟨*proof*⟩

lemma *seg-start-bound*:
assumes *A*: *i*<*length B* **shows** *seg-start i* < *length S*
⟨*proof*⟩

lemma *seg-end-bound*:
assumes *A*: *i*<*length B* **shows** *seg-end i* ≤ *length S*
⟨*proof*⟩

lemma *seg-start-less-end*: *i*<*length B* ⟹ *seg-start i* < *seg-end i*
⟨*proof*⟩

lemma *seg-end-less-start*: ⟦*i*<*j*; *j*<*length B*⟧ ⟹ *seg-end i* ≤ *seg-start j*
⟨*proof*⟩

lemma *find-seg-bounds*:
assumes *A*: *j*<*length S*
shows *seg-start* (*find-seg j*) ≤ *j*
and *j* < *seg-end* (*find-seg j*)
and *find-seg j* < *length B*
⟨*proof*⟩

lemma *find-seg-correct*:
assumes *A*: *j*<*length S*
shows *S*!*j* ∈ *seg* (*find-seg j*) **and** *find-seg j* < *length B*
⟨*proof*⟩

lemma *set-p-α-is-set-S*:

$\bigcup(set\ p\text{-}\alpha) = set\ S$
⟨*proof*⟩

lemma *S-idx-uniq*:
$[\![i<length\ S;\ j<length\ S]\!] \Longrightarrow S!i=S!j \longleftrightarrow i=j$
⟨*proof*⟩

lemma *S-idx-of-correct*:
assumes *A*: $v\in\bigcup(set\ p\text{-}\alpha)$
shows *S-idx-of* $v < length\ S$ **and** $S!S\text{-}idx\text{-}of\ v = v$
⟨*proof*⟩

lemma *p-α-disjoint-sym*:
shows $\forall\, i\ j\ v.\ i<length\ p\text{-}\alpha \wedge j<length\ p\text{-}\alpha \wedge v\in p\text{-}\alpha!i \wedge v\in p\text{-}\alpha!j \longrightarrow i=j$
⟨*proof*⟩

end

1.4.3 Refinement of the Operations

definition *GS-initial-impl* :: *'a oGS* ⇒ *'a* ⇒ *'a set* ⇒ *'a GS* **where**
GS-initial-impl I v0 succs ≡ (
[*v0*],
[*0*],
$I(v0\mapsto(STACK\ 0))$,
if succs={} *then* [] *else* [(*0*,*succs*)])

context *GS*
begin
definition *push-impl v succs* ≡
let
- = *stat-newnode* ();
$j = length\ S$;
$S = S@[v]$;
$B = B@[j]$;
$I = I(v \mapsto STACK\ j)$;
$P = if\ succs=\{\}\ then\ P\ else\ P@[(j,succs)]$
in
(*S*,*B*,*I*,*P*)

definition *mark-as-done*
where $\bigwedge l\ u\ I.\ mark\text{-}as\text{-}done\ l\ u\ I \equiv do\ \{$
(-,*I*)←*WHILET*
$(\lambda(l,I).\ l<u)$
$(\lambda(l,I).\ do\ \{\ ASSERT\ (l<length\ S);\ RETURN\ (Suc\ l,I(S!l \mapsto DONE))\})$
(*l*,*I*);
RETURN I
}

definition *mark-as-done-abs* **where**
⋀*l u I*. *mark-as-done-abs l u I*
≡ (λ*v*. *if v*∈{*S*!*j* | *j*. *l*≤*j* ∧ *j*<*u*} *then Some DONE else I v*)

lemma *mark-as-done-aux*:
fixes *l u I*
shows ⟦*l*<*u*; *u*≤*length S*⟧ ⟹ *mark-as-done l u I*
≤ *SPEC* (λ*r*. *r* = *mark-as-done-abs l u I*)
⟨*proof*⟩

definition *pop-impl* ≡
do {
let lsi = *length B* − *1*;
ASSERT (*lsi*<*length B*);
I ← *mark-as-done* (*seg-start lsi*) (*seg-end lsi*) *I*;
ASSERT (*B*≠[]);
let S = *take* (*last B*) *S*;
ASSERT (*B*≠[]);
let B = *butlast B*;
RETURN (*S*,*B*,*I*,*P*)
}

definition *sel-rem-last* ≡
if P=[] *then*
RETURN (*None*,(*S*,*B*,*I*,*P*))
else do {
let (*j*,*succs*) = *last P*;
ASSERT (*length B* − *1* < *length B*);
if j ≥ *seg-start* (*length B* − *1*) *then do* {
ASSERT (*succs*≠{});
v ← *SPEC* (λ*x*. *x*∈*succs*);
let succs = *succs* − {*v*};
ASSERT (*P*≠[] ∧ *length P* − *1* < *length P*);
let P = (*if succs*={} *then butlast P else P*[*length P* − *1* := (*j*,*succs*)]);
RETURN (*Some v*,(*S*,*B*,*I*,*P*))
} *else RETURN* (*None*,(*S*,*B*,*I*,*P*))
}

definition *find-seg-impl j* ≡ *find-max-nat* (*length B*) (λ*i*. *B*!*i*≤*j*)

lemma (**in** *GS-invar*) *find-seg-impl*:
j<*length S* ⟹ *find-seg-impl j* = *find-seg j*
⟨*proof*⟩

definition *idx-of-impl v* ≡ *do* {
ASSERT (∃ *i*. *I v* = *Some* (*STACK i*));

```
    let j = S-idx-of v;
    ASSERT (j<length S);
    let i = find-seg-impl j;
    RETURN i
  }

 definition collapse-impl v ≡
  do {
    i←idx-of-impl v;
    ASSERT (i+1 ≤ length B);
    let B = take (i+1) B;
    RETURN (S,B,I,P)
  }

end

lemma (in −) GS-initial-correct:
 assumes REL: (I,D)∈oGS-rel
 assumes A: v0∉D
 shows GS.α (GS-initial-impl I v0 succs) = ([{v0}],D,{v0}×succs) (is ?G1)
 and GS-invar (GS-initial-impl I v0 succs) (is ?G2)
⟨proof⟩

context GS-invar
begin
 lemma push-correct:
  assumes A: v∉⋃(set p-α) and B: v∉D-α
  shows GS.α (push-impl v succs) = (p-α@[{v}],D-α,pE-α ∪ {v}×succs)
   (is ?G1)
  and GS-invar (push-impl v succs) (is ?G2)
 ⟨proof⟩

 lemma no-last-out-P-aux:
  assumes NE: p-α≠[] and NS: pE-α ∩ last p-α × UNIV = {}
  shows set P ⊆ {0..<last B} × UNIV
 ⟨proof⟩

 lemma pop-correct:
  assumes NE: p-α≠[] and NS: pE-α ∩ last p-α × UNIV = {}
  shows pop-impl
   ≤ ⇓GS-rel (SPEC (λr. r=(butlast p-α, D-α ∪ last p-α, pE-α)))
 ⟨proof⟩


 lemma sel-rem-last-correct:
  assumes NE: p-α≠[]
  shows
  sel-rem-last ≤ ⇓(Id ×r GS-rel) (select-edge (p-α,D-α,pE-α))
 ⟨proof⟩
```

lemma *find-seg-idx-of-correct*:
assumes *A*: $v \in \bigcup (set\ p\text{-}\alpha)$
shows $(find\text{-}seg\ (S\text{-}idx\text{-}of\ v)) = idx\text{-}of\ p\text{-}\alpha\ v$
⟨*proof*⟩

lemma *idx-of-correct*:
assumes *A*: $v \in \bigcup (set\ p\text{-}\alpha)$
shows $idx\text{-}of\text{-}impl\ v \leq SPEC\ (\lambda x.\ x = idx\text{-}of\ p\text{-}\alpha\ v \wedge x < length\ B)$
⟨*proof*⟩

lemma *collapse-correct*:
assumes *A*: $v \in \bigcup (set\ p\text{-}\alpha)$
shows $collapse\text{-}impl\ v \leq \Downarrow GS\text{-}rel\ (SPEC\ (\lambda r.\ r = collapse\ v\ \alpha))$
⟨*proof*⟩

end

Technical adjustment for avoiding case-splits for definitions extracted from GS-locale

lemma *opt-GSdef*: $f \equiv g \Longrightarrow f\ s \equiv case\ s\ of\ (S,B,I,P) \Rightarrow g\ (S,B,I,P)$ ⟨*proof*⟩

lemma *ext-def*: $f \equiv g \Longrightarrow f\ x \equiv g\ x$ ⟨*proof*⟩

context *fr-graph* **begin**
definition *push-impl v s* ≡ *GS.push-impl s v* (*E*‘‘{*v*})
lemmas *push-impl-def-opt* =
push-impl-def[*abs-def*,
THEN ext-def, *THEN opt-GSdef*, *unfolded GS.push-impl-def GS-sel-simps*]

Definition for presentation

lemma *push-impl v* (*S*,*B*,*I*,*P*) ≡ (*S*@[*v*], *B*@[*length S*], *I*(*v*↦*STACK* (*length S*)),
if E‘‘{*v*}={} *then P else P*@[(*length S*,*E*‘‘{*v*})])
⟨*proof*⟩

lemma *GS-α-split*:
$GS.\alpha\ s = (p,D,pE) \longleftrightarrow (p = GS.p\text{-}\alpha\ s \wedge D = GS.D\text{-}\alpha\ s \wedge pE = GS.pE\text{-}\alpha\ s)$
$(p,D,pE) = GS.\alpha\ s \longleftrightarrow (p = GS.p\text{-}\alpha\ s \wedge D = GS.D\text{-}\alpha\ s \wedge pE = GS.pE\text{-}\alpha\ s)$
⟨*proof*⟩

lemma *push-refine*:
assumes *A*: $(s,(p,D,pE)) \in GS\text{-}rel \quad (v,v') \in Id$
assumes *B*: $v \notin \bigcup (set\ p) \quad v \notin D$
shows $(push\text{-}impl\ v\ s,\ push\ v'\ (p,D,pE)) \in GS\text{-}rel$
⟨*proof*⟩

definition *pop-impl s* ≡ *GS.pop-impl s*
lemmas *pop-impl-def-opt* =

pop-impl-def[*abs-def*, *THEN opt-GSdef*, *unfolded GS.pop-impl-def GS.mark-as-done-def GS.seg-start-def GS.seg-end-def GS-sel-simps*]

lemma *pop-refine*:
assumes *A*: $(s,(p,D,pE))\in GS\text{-}rel$
assumes *B*: $p \neq [] \quad pE \cap last\ p \times UNIV = \{\}$
shows $pop\text{-}impl\ s \leq \Downarrow GS\text{-}rel\ (RETURN\ (pop\ (p,D,pE)))$
⟨*proof*⟩

thm *pop-refine*[*no-vars*]

definition $collapse\text{-}impl\ v\ s \equiv GS.collapse\text{-}impl\ s\ v$
lemmas *collapse-impl-def-opt* =
collapse-impl-def[*abs-def*,
THEN ext-def, *THEN opt-GSdef*, *unfolded GS.collapse-impl-def GS-sel-simps*]

lemma *collapse-refine*:
assumes *A*: $(s,(p,D,pE))\in GS\text{-}rel \quad (v,v')\in Id$
assumes *B*: $v'\in\bigcup(set\ p)$
shows $collapse\text{-}impl\ v\ s \leq \Downarrow GS\text{-}rel\ (RETURN\ (collapse\ v'\ (p,D,pE)))$
⟨*proof*⟩

definition $select\text{-}edge\text{-}impl\ s \equiv GS.sel\text{-}rem\text{-}last\ s$
lemmas *select-edge-impl-def-opt* =
select-edge-impl-def[*abs-def*,
THEN opt-GSdef,
unfolded GS.sel-rem-last-def GS.seg-start-def GS-sel-simps]

lemma *select-edge-refine*:
assumes *A*: $(s,(p,D,pE))\in GS\text{-}rel$
assumes *NE*: $p \neq []$
shows $select\text{-}edge\text{-}impl\ s \leq \Downarrow(Id \times_r GS\text{-}rel)\ (select\text{-}edge\ (p,D,pE))$
⟨*proof*⟩

definition $initial\text{-}impl\ v0\ I \equiv GS\text{-}initial\text{-}impl\ I\ v0\ (E``\{v0\})$

lemma *initial-refine*:
$[\![v0\notin D0;\ (I,D0)\in oGS\text{-}rel;\ (v0i,v0)\in Id]\!]$
$\Longrightarrow (initial\text{-}impl\ v0i\ I, initial\ v0\ D0)\in GS\text{-}rel$
⟨*proof*⟩

definition $path\text{-}is\text{-}empty\text{-}impl\ s \equiv GS.S\ s = []$
lemma *path-is-empty-refine*:
$GS\text{-}invar\ s \Longrightarrow path\text{-}is\text{-}empty\text{-}impl\ s \longleftrightarrow GS.p\text{-}\alpha\ s=[]$
⟨*proof*⟩

definition (**in** *GS*) *is-on-stack-impl v*

$\equiv$ *case I v of Some (STACK -)* $\Rightarrow$ *True* | - $\Rightarrow$ *False*

lemma (**in** *GS-invar*) *is-on-stack-impl-correct*:
shows *is-on-stack-impl v* $\longleftrightarrow$ *v*$\in\bigcup$(*set p-*α)
⟨*proof*⟩

definition *is-on-stack-impl v s* $\equiv$ *GS.is-on-stack-impl s v*
lemmas *is-on-stack-impl-def-opt* =
is-on-stack-impl-def[*abs-def*, *THEN ext-def*, *THEN opt-GSdef*,
unfolded GS.is-on-stack-impl-def GS-sel-simps]

lemma *is-on-stack-refine*:
⟦ *GS-invar s* ⟧ $\Longrightarrow$ *is-on-stack-impl v s* $\longleftrightarrow$ *v*$\in\bigcup$(*set* (*GS.p-*α *s*))
⟨*proof*⟩

definition (**in** *GS*) *is-done-impl v*
$\equiv$ *case I v of Some DONE* $\Rightarrow$ *True* | - $\Rightarrow$ *False*

lemma (**in** *GS-invar*) *is-done-impl-correct*:
shows *is-done-impl v* $\longleftrightarrow$ *v*$\in$*D-*α
⟨*proof*⟩

definition *is-done-oimpl v I* $\equiv$ *case I v of Some DONE* $\Rightarrow$ *True* | - $\Rightarrow$ *False*

definition *is-done-impl v s* $\equiv$ *GS.is-done-impl s v*

lemma *is-done-orefine*:
⟦ *oGS-invar s* ⟧ $\Longrightarrow$ *is-done-oimpl v s* $\longleftrightarrow$ *v*$\in$*oGS-*α *s*
⟨*proof*⟩

lemma *is-done-refine*:
⟦ *GS-invar s* ⟧ $\Longrightarrow$ *is-done-impl v s* $\longleftrightarrow$ *v*$\in$*GS.D-*α *s*
⟨*proof*⟩

lemma *oinitial-refine*: (*Map.empty*, {}) $\in$ *oGS-rel*
⟨*proof*⟩

end

1.4.4 Refined Skeleton Algorithm

context *fr-graph* **begin**

lemma *I-to-outer*:
assumes ((*S*, *B*, *I*, *P*), ([], *D*, {})) $\in$ *GS-rel*
shows (*I*,*D*)$\in$*oGS-rel*
⟨*proof*⟩

```
definition skeleton-impl :: 'v oGS nres where
  skeleton-impl ≡ do {
    stat-start-nres;
    let I=Map.empty;
    r ← FOREACHi (λit I. outer-invar it (oGS-α I)) V0 (λv0 I0. do {
      if ¬is-done-oimpl v0 I0 then do {
        let s = initial-impl v0 I0;

        (S,B,I,P)←WHILEIT (invar v0 (oGS-α I0) o GS.α)
          (λs. ¬path-is-empty-impl s) (λs.
        do {
          — Select edge from end of path
          (vo,s) ← select-edge-impl s;

          case vo of
            Some v ⇒ do {
              if is-on-stack-impl v s then do {
                collapse-impl v s
              } else if ¬is-done-impl v s then do {
                — Edge to new node. Append to path
                RETURN (push-impl v s)
              } else do {
                — Edge to done node. Skip
                RETURN s
              }
            }
          | None ⇒ do {
              — No more outgoing edges from current node on path
              pop-impl s
            }
        }) s;
        RETURN I
      } else
        RETURN I0
    }) I;
    stat-stop-nres;
    RETURN r
  }
```

Correctness Theorem **lemma** *skeleton-impl* $\leq \Downarrow$*oGS-rel skeleton*
⟨*proof*⟩

lemmas *skeleton-refines*
= *select-edge-refine push-refine pop-refine collapse-refine*
initial-refine oinitial-refine
lemmas *skeleton-refine-simps*
= *GS-rel-def br-def GS.α-def oGS-rel-def oGS-α-def*
is-on-stack-refine path-is-empty-refine is-done-refine is-done-orefine

Short proof, for presentation

context
notes [[*goals-limit* = *1*]]
notes [*refine*] = *inj-on-id bind-refine′*
begin
lemma *skeleton-impl* ≤ ⇓*oGS-rel skeleton*
⟨*proof*⟩

end

end

end

1.5 Enumerating the SCCs of a Graph

theory *Gabow-SCC*
imports *Gabow-Skeleton*
begin

As a first variant, we implement an algorithm that computes a list of SCCs of a graph, in topological order. This is the standard variant described by Gabow [5].

1.6 Specification

context *fr-graph*
begin

We specify a distinct list that covers all reachable nodes and contains SCCs in topological order

definition *compute-SCC-spec* ≡ *SPEC* (λ*l*.
distinct l ∧ ⋃(*set l*) = *E***‘‘*V0* ∧ (∀ *U*∈*set l*. *is-scc E U*)
∧ (∀ *i j*. *i*<*j* ∧ *j*<*length l* ⟶ *l*!*j* × *l*!*i* ∩ *E** = {}))
end

1.7 Extended Invariant

locale *cscc-invar-ext* = *fr-graph G*
for *G* :: (′*v*,′*more*) *graph-rec-scheme* +
fixes *l* :: ′*v set list* **and** *D* :: ′*v set*
assumes *l-is-D*: ⋃(*set l*) = *D* — The output contains all done CNodes
assumes *l-scc*: *set l* ⊆ *Collect* (*is-scc E*) — The output contains only SCCs
assumes *l-no-fwd*: ⋀*i j*. ⟦*i*<*j*; *j*<*length l*⟧ ⟹ *l*!*j* × *l*!*i* ∩ *E** = {}
— The output contains no forward edges
begin
lemma *l-no-empty*: {}∉*set l* ⟨*proof*⟩
end

```
locale cscc-outer-invar-loc = outer-invar-loc G it D + cscc-invar-ext G l D
  for G :: ('v,'more) graph-rec-scheme and it l D
begin
  lemma locale-this: cscc-outer-invar-loc G it l D ⟨proof⟩
  lemma abs-outer-this: outer-invar-loc G it D ⟨proof⟩
end

locale cscc-invar-loc = invar-loc G v0 D0 p D pE + cscc-invar-ext G l D
  for G :: ('v,'more) graph-rec-scheme and v0 D0 and l :: 'v set list
  and p D pE
begin
  lemma locale-this: cscc-invar-loc G v0 D0 l p D pE ⟨proof⟩
  lemma invar-this: invar-loc G v0 D0 p D pE ⟨proof⟩
end

context fr-graph
begin
  definition cscc-outer-invar ≡ λit (l,D). cscc-outer-invar-loc G it l D
  definition cscc-invar ≡ λv0 D0 (l,p,D,pE). cscc-invar-loc G v0 D0 l p D pE
end
```

1.8 Definition of the SCC-Algorithm

```
context fr-graph
begin
  definition compute-SCC :: 'v set list nres where
    compute-SCC ≡ do {
      let so = ([],{});
      (l,D) ← FOREACHi cscc-outer-invar V0 (λv0 (l,D0). do {
        if v0∉D0 then do {
          let s = (l,initial v0 D0);

          (l,p,D,pE) ←
          WHILEIT (cscc-invar v0 D0)
            (λ(l,p,D,pE). p ≠ []) (λ(l,p,D,pE).
          do {
            — Select edge from end of path
            (vo,(p,D,pE)) ← select-edge (p,D,pE);

            ASSERT (p≠[]);
            case vo of
              Some v ⇒ do {
                if v ∈ ⋃(set p) then do {
                  — Collapse
                  RETURN (l,collapse v (p,D,pE))
                } else if v∉D then do {
                  — Edge to new node. Append to path
                  RETURN (l,push v (p,D,pE))
```

```
            } else RETURN (l,p,D,pE)
          }
        | None ⇒ do {
            — No more outgoing edges from current node on path
            ASSERT (pE ∩ last p × UNIV = {});
            let V = last p;
            let (p,D,pE) = pop (p,D,pE);
            let l = V#l;
            RETURN (l,p,D,pE)
          }
        }) s;
        ASSERT (p=[] ∧ pE={});
        RETURN (l,D)
      } else
        RETURN (l,D0)
    }) so;
    RETURN l
  }
end
```

1.9 Preservation of Invariant Extension

```
context cscc-invar-ext
begin
  lemma l-disjoint:
    assumes A: i<j    j<length l
    shows l!i ∩ l!j = {}
  ⟨proof⟩

  corollary l-distinct: distinct l
    ⟨proof⟩
end

context fr-graph
begin
  definition cscc-invar-part ≡ λ(l,p,D,pE). cscc-invar-ext G l D

  lemma cscc-invarI[intro?]:
    assumes invar v0 D0 PDPE
    assumes invar v0 D0 PDPE ⟹ cscc-invar-part (l,PDPE)
    shows cscc-invar v0 D0 (l,PDPE)
    ⟨proof⟩

  thm cscc-invarI[of v-0 D-0 s l]

  lemma cscc-outer-invarI[intro?]:
    assumes outer-invar it D
    assumes outer-invar it D ⟹ cscc-invar-ext G l D
    shows cscc-outer-invar it (l,D)
```

⟨*proof*⟩

lemma *cscc-invar-initial*[*simp*, *intro*!]:
assumes *A*: *v0*∈*it* *v0*∉*D0*
assumes *INV*: *cscc-outer-invar it* (*l*,*D0*)
shows *cscc-invar-part* (*l*,*initial v0 D0*)
⟨*proof*⟩

lemma *cscc-invar-pop*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
assumes *invar v0 D0* (*pop* (*p*,*D*,*pE*))
assumes *NE*[*simp*]: *p*≠[]
assumes *NO'*: *pE* ∩ (*last p* × *UNIV*) = {}
shows *cscc-invar-part* (*last p* # *l*, *pop* (*p*,*D*,*pE*))
⟨*proof*⟩

thm *cscc-invar-pop*[*of v-0 D-0 l p D pE*]

lemma *cscc-invar-unchanged*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*p'*,*D*,*pE'*)
⟨*proof*⟩

corollary *cscc-invar-collapse*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*collapse v* (*p'*,*D*,*pE'*))
⟨*proof*⟩

corollary *cscc-invar-push*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*push v* (*p'*,*D*,*pE'*))
⟨*proof*⟩

lemma *cscc-outer-invar-initial*: *cscc-invar-ext G* [] {}
⟨*proof*⟩

lemma *cscc-invar-outer-newnode*:
assumes *A*: *v0*∉*D0* *v0*∈*it*
assumes *OINV*: *cscc-outer-invar it* (*l*,*D0*)
assumes *INV*: *cscc-invar v0 D0* (*l'*,[],*D'*,*pE*)
shows *cscc-invar-ext G l' D'*
⟨*proof*⟩

lemma *cscc-invar-outer-Dnode*:
assumes *cscc-outer-invar it* (*l*, *D*)
shows *cscc-invar-ext G l D*
⟨*proof*⟩

lemmas *cscc-invar-preserve = invar-preserve*
cscc-invar-initial
cscc-invar-pop cscc-invar-collapse cscc-invar-push cscc-invar-unchanged
cscc-outer-invar-initial cscc-invar-outer-newnode cscc-invar-outer-Dnode

On termination, the invariant implies the specification

lemma *cscc-finI*:
assumes *INV*: *cscc-outer-invar* {} (*l*,*D*)
shows *fin-l-is-scc*: ⟦*U*∈*set l*⟧ ⟹ *is-scc E U*
and *fin-l-distinct*: *distinct l*
and *fin-l-is-reachable*: ⋃(*set l*) = E^* `` *V0*
and *fin-l-no-fwd*: ⟦*i*<*j*; *j*<*length l*⟧ ⟹ *l*!*j* ×*l*!*i* ∩ E^* = {}
⟨*proof*⟩

end

1.10 Main Correctness Proof

context *fr-graph*
begin
lemma *invar-from-cscc-invarI*: *cscc-invar v0 D0* (*L*,*PDPE*) ⟹ *invar v0 D0 PDPE*
⟨*proof*⟩

lemma *outer-invar-from-cscc-invarI*:
cscc-outer-invar it (*L*,*D*) ⟹*outer-invar it D*
⟨*proof*⟩

With the extended invariant and the auxiliary lemmas, the actual correctness proof is straightforward:

theorem *compute-SCC-correct*: *compute-SCC* ≤ *compute-SCC-spec*
⟨*proof*⟩

Simple proof, for presentation

context
notes [*refine*]=*refine-vcg*
notes [[*goals-limit = 1*]]
begin
theorem *compute-SCC* ≤ *compute-SCC-spec*
⟨*proof*⟩
end

end

1.11 Refinement to Gabow's Data Structure

context *GS* **begin**
definition *seg-set-impl l u* ≡ *do* {

```
  (-,res) ← WHILET
    (λ(l,-). l<u)
    (λ(l,res). do {
      ASSERT (l<length S);
      let x = S!l;
      ASSERT (x∉res);
      RETURN (Suc l,insert x res)
    })
    (l,{});

  RETURN res
}

lemma seg-set-impl-aux:
  fixes l u
  shows ⟦l<u; u≤length S; distinct S⟧ ⟹ seg-set-impl l u
  ≤ SPEC (λr. r = {S!j | j. l≤j ∧ j<u})
  ⟨proof⟩

lemma (in GS-invar) seg-set-impl-correct:
  assumes i<length B
  shows seg-set-impl (seg-start i) (seg-end i) ≤ SPEC (λr. r=p-α!i)
  ⟨proof⟩

definition last-seg-impl
  ≡ do {
    ASSERT (length B − 1 < length B);
    seg-set-impl (seg-start (length B − 1)) (seg-end (length B − 1))
  }

lemma (in GS-invar) last-seg-impl-correct:
  assumes p-α ≠ []
  shows last-seg-impl ≤ SPEC (λr. r=last p-α)
  ⟨proof⟩

end

context fr-graph
begin

  definition last-seg-impl s ≡ GS.last-seg-impl s
  lemmas last-seg-impl-def-opt =
    last-seg-impl-def[abs-def, THEN opt-GSdef,
      unfolded GS.last-seg-impl-def GS.seg-set-impl-def
    GS.seg-start-def GS.seg-end-def GS-sel-simps]


  lemma last-seg-impl-refine:
    assumes A: (s,(p,D,pE))∈GS-rel
```

```
  assumes NE: p≠[]
  shows last-seg-impl s ≤ ⇓Id (RETURN (last p))
⟨proof⟩

definition compute-SCC-impl :: 'v set list nres where
  compute-SCC-impl ≡ do {
    stat-start-nres;
    let so = ([],Map.empty);
    (l,D) ← FOREACHi (λit (l,s). cscc-outer-invar it (l,oGS-α s))
      V0 (λv0 (l,I0). do {
        if ¬is-done-oimpl v0 I0 then do {
          let ls = (l,initial-impl v0 I0);

          (l,(S,B,I,P))←WHILEIT (λ(l,s). cscc-invar v0 (oGS-α I0) (l,GS.α s))
            (λ(l,s). ¬path-is-empty-impl s) (λ(l,s).
          do {
            — Select edge from end of path
            (vo,s) ← select-edge-impl s;

            case vo of
              Some v ⇒ do {
                if is-on-stack-impl v s then do {
                  s←collapse-impl v s;
                  RETURN (l,s)
                } else if ¬is-done-impl v s then do {
                  — Edge to new node. Append to path
                  RETURN (l,push-impl v s)
                } else do {
                  — Edge to done node. Skip
                  RETURN (l,s)
                }
              }
            | None ⇒ do {
                — No more outgoing edges from current node on path
                scc ← last-seg-impl s;
                s←pop-impl s;
                let l = scc#l;
                RETURN (l,s)
              }
          }) (ls);
          RETURN (l,I)
        } else RETURN (l,I0)
    }) so;
    stat-stop-nres;
    RETURN l
  }

lemma compute-SCC-impl-refine: compute-SCC-impl ≤ ⇓Id compute-SCC
⟨proof⟩
```

end

end

1.12 Safety-Property Model-Checker

theory *Find-Path*
imports
CAVA-Automata.Digraph
CAVA-Base.CAVA-Code-Target
begin

1.13 Finding Path to Error

This function searches a graph and a set of start nodes for a reachable node that satisfies some property, and returns a path to such a node iff it exists.

definition *find-path E U0 P* ≡ *do* {
ASSERT (*finite U0*);
ASSERT (*finite* (E^* ‘‘ *U0*));
SPEC (λ*p*. *case p of*
Some (*p*,*v*) ⇒ ∃ *u0*∈*U0*. *path E u0 p v* ∧ *P v* ∧ (∀ *v*∈*set p*. ¬ *P v*)
| *None* ⇒ ∀ *u0*∈*U0*. ∀ *v*∈E^* ‘‘{*u0*}. ¬*P v*)
}

lemma *find-path-ex-rule*:
assumes *finite U0*
assumes *finite* (E^* ‘‘*U0*)
assumes ∃ *v*∈E^* ‘‘*U0*. *P v*
shows *find-path E U0 P* ≤ *SPEC* (λ*r*.
∃ *p v*. *r* = *Some* (*p*,*v*) ∧ *P v* ∧ (∀ *v*∈*set p*. ¬*P v*) ∧ (∃ *u0*∈*U0*. *path E u0 p v*))
⟨*proof*⟩

1.13.1 Nontrivial Paths

definition *find-path1 E u0 P* ≡ *do* {
ASSERT (*finite* (E^* ‘‘{*u0*}));
SPEC (λ*p*. *case p of*
Some (*p*,*v*) ⇒ *path E u0 p v* ∧ *P v* ∧ *p*≠[]
| *None* ⇒ ∀ *v*∈E^+ ‘‘{*u0*}. ¬*P v*)}

lemma (**in** −) *find-path1-ex-rule*:
assumes *finite* (E^* ‘‘{*u0*})
assumes ∃ *v*∈E^+ ‘‘{*u0*}. *P v*
shows *find-path1 E u0 P* ≤ *SPEC* (λ*r*.
∃ *p v*. *r* = *Some* (*p*,*v*) ∧ *p*≠[] ∧ *P v* ∧ *path E u0 p v*)
⟨*proof*⟩

end

1.14 Lasso Finding Algorithm for Generalized Büchi Graphs

theory *Gabow-GBG*
imports
Gabow-Skeleton
CAVA-Automata.Lasso
Find-Path
begin

locale *igb-fr-graph* =
igb-graph G + *fr-graph G*
for *G* :: (*'Q*,*'more*) *igb-graph-rec-scheme*

lemma *igb-fr-graphI*:
assumes *igb-graph G*
assumes *finite* ((*g-E G*)* `` *g-V0 G*)
shows *igb-fr-graph G*
⟨*proof*⟩

We implement an algorithm that computes witnesses for the non-emptiness of Generalized Büchi Graphs (GBG).

1.15 Specification

context *igb-graph*
begin
definition *ce-correct*
— Specifies a correct counter-example
where
ce-correct Vr Vl ≡ (∃ *pr pl*.
Vr ⊆ *E** ``*V0* ∧ *Vl* ⊆ *E*\* ``*V0* — Only reachable nodes are covered
∧ *set pr*⊆*Vr* ∧ *set pl*⊆*Vl* — The paths are inside the specified sets
∧ *Vl*×*Vl* ⊆ (*E* ∩ *Vl*×*Vl*)* — *Vl* is mutually connected
∧ *Vl*×*Vl* ∩ *E* ≠ {} — *Vl* is non-trivial
∧ *is-lasso-prpl* (*pr*,*pl*)) — Paths form a lasso

definition *find-ce-spec* :: (*'Q set* × *'Q set*) *option nres* **where**
find-ce-spec ≡ *SPEC* (λ*r*. *case r of*
None ⇒ (∀ *prpl*. ¬*is-lasso-prpl prpl*)
| *Some* (*Vr*,*Vl*) ⇒ *ce-correct Vr Vl*
)

definition *find-lasso-spec* :: (*'Q list* × *'Q list*) *option nres* **where**
find-lasso-spec ≡ *SPEC* (λ*r*. *case r of*
None ⇒ (∀ *prpl*. ¬*is-lasso-prpl prpl*)
| *Some prpl* ⇒ *is-lasso-prpl prpl*
)

end

1.16 Invariant Extension

Extension of the outer invariant:

context *igb-fr-graph*
begin
  definition *no-acc-over*
    — Specifies that there is no accepting cycle touching a set of nodes
    where
    no-acc-over D ≡ ¬(∃ *v*∈*D*. ∃ *pl*. *pl*≠[] ∧ *path E v pl v* ∧
    (∀ *i*<*num-acc*. ∃ *q*∈*set pl*. *i*∈*acc q*))

  definition *fgl-outer-invar-ext* ≡ λ*it* (*brk*,*D*).
    case brk of None ⇒ *no-acc-over D* | *Some* (*Vr*, *Vl*) ⇒ *ce-correct Vr Vl*

  definition *fgl-outer-invar* ≡ λ*it* (*brk*,*D*). *case brk of*
    None ⇒ *outer-invar it D* ∧ *no-acc-over D*
  | *Some* (*Vr*, *Vl*) ⇒ *ce-correct Vr Vl*

end

Extension of the inner invariant:

locale *fgl-invar-loc* =
  invar-loc G v0 D0 p D pE
  + *igb-graph G*
  for *G* :: ('*Q*, '*more*) *igb-graph-rec-scheme*
  and *v0 D0* **and** *brk* :: ('*Q set* × '*Q set*) *option* **and** *p D pE* +
  assumes *no-acc*: *brk*=*None* ⟹ ¬(∃ *v pl*. *pl*≠[] ∧ *path lvE v pl v* ∧
    (∀ *i*<*num-acc*. ∃ *q*∈*set pl*. *i*∈*acc q*)) — No accepting cycle over visited edges
  assumes *acc*: *brk*=*Some* (*Vr*, *Vl*) ⟹ *ce-correct Vr Vl*
begin
  lemma *locale-this*: *fgl-invar-loc G v0 D0 brk p D pE*
    ⟨*proof*⟩
  lemma *invar-loc-this*: *invar-loc G v0 D0 p D pE* ⟨*proof*⟩
  lemma *eas-gba-graph-this*: *igb-graph G* ⟨*proof*⟩
end

definition (**in** *igb-graph*) *fgl-invar v0 D0* ≡
  λ(*brk*, *p*, *D*, *pE*). *fgl-invar-loc G v0 D0 brk p D pE*

1.17 Definition of the Lasso-Finding Algorithm

context *igb-fr-graph*
begin
  definition *find-ce* :: ('*Q set* × '*Q set*) *option nres* **where**
    find-ce ≡ *do* {
      let D = {};

```
    (brk,-)←FOREACHci fgl-outer-invar V0
      (λ(brk,-). brk=None)
      (λv0 (brk,D0). do {
        if v0∉D0 then do {
          let s = (None,initial v0 D0);

          (brk,p,D,pE) ← WHILEIT (fgl-invar v0 D0)
            (λ(brk,p,D,pE). brk=None ∧ p ≠ []) (λ(-,p,D,pE).
          do {
            — Select edge from end of path
            (vo,(p,D,pE)) ← select-edge (p,D,pE);

            ASSERT (p≠[]);
            case vo of
              Some v ⇒ do {
                if v ∈ ⋃(set p) then do {
                  — Collapse
                  let (p,D,pE) = collapse v (p,D,pE);

                  ASSERT (p≠[]);

                  if ∀ i<num-acc. ∃ q∈last p. i∈acc q then
                    RETURN (Some (⋃(set (butlast p)),last p),p,D,pE)
                  else
                    RETURN (None,p,D,pE)
                } else if v∉D then do {
                  — Edge to new node. Append to path
                  RETURN (None,push v (p,D,pE))
                } else RETURN (None,p,D,pE)
              }
            | None ⇒ do {
                — No more outgoing edges from current node on path
                ASSERT (pE ∩ last p × UNIV = {});
                RETURN (None,pop (p,D,pE))
              }
          }) s;
          ASSERT (brk=None ⟶ (p=[] ∧ pE={}));
          RETURN (brk,D)
        } else
          RETURN (brk,D0)
      }) (None,D);
      RETURN brk
    }
end
```

1.18 Invariant Preservation

context *igb-fr-graph*
begin

definition *fgl-invar-part* ≡ λ(*brk*, *p*, *D*, *pE*).
fgl-invar-loc-axioms G brk p D pE

lemma *fgl-outer-invarI*[*intro?*]:
⟦
brk=*None* ⟹ *outer-invar it D*;
⟦*brk*=*None* ⟹ *outer-invar it D*⟧ ⟹ *fgl-outer-invar-ext it* (*brk*,*D*)⟧
⟹ *fgl-outer-invar it* (*brk*,*D*)
⟨*proof*⟩

lemma *fgl-invarI*[*intro?*]:
⟦ *invar v0 D0 PDPE*;
invar v0 D0 PDPE ⟹ *fgl-invar-part* (*B*,*PDPE*)⟧
⟹ *fgl-invar v0 D0* (*B*,*PDPE*)
⟨*proof*⟩

lemma *fgl-invar-initial*:
assumes *OINV*: *fgl-outer-invar it* (*None*,*D0*)
assumes *A*: *v0*∈*it* *v0*∉*D0*
shows *fgl-invar-part* (*None*, *initial v0 D0*)
⟨*proof*⟩

lemma *fgl-invar-pop*:
assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
assumes *INV'*: *invar v0 D0* (*pop* (*p*,*D*,*pE*))
assumes *NE*[*simp*]: *p*≠[]
assumes *NO'*: *pE* ∩ *last p* × *UNIV* = {}
shows *fgl-invar-part* (*None*, *pop* (*p*,*D*,*pE*))
⟨*proof*⟩

lemma *fgl-invar-collapse-ce-aux*:
assumes *INV*: *invar v0 D0* (*p*, *D*, *pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *NONTRIV*: *vE p D pE* ∩ (*last p* × *last p*) ≠ {}
assumes *ACC*: ∀ *i*<*num-acc*. ∃ *q*∈*last p*. *i*∈*acc q*
shows *fgl-invar-part* (*Some* (⋃(*set* (*butlast p*)), *last p*), *p*, *D*, *pE*)
⟨*proof*⟩

lemma *fgl-invar-collapse-ce*:
fixes *u v*
assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
defines *pE'* ≡ *pE* − {(*u*,*v*)}
assumes *CFMT*: (*p'*,*D'*,*pE''*) = *collapse v* (*p*,*D*,*pE'*)
assumes *INV'*: *invar v0 D0* (*p'*,*D'*,*pE''*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* **and** *u*∈*last p*
assumes *BACK*: *v*∈⋃(*set p*)

assumes *ACC*: $\forall i{<}num\text{-}acc.\ \exists q{\in}last\ p'.\ i{\in}acc\ q$
defines *i-def*: $i \equiv idx\text{-}of\ p\ v$
shows *fgl-invar-part* (
$Some\ (\bigcup(set\ (butlast\ p')),\ last\ p'),$
$collapse\ v\ (p,D,pE')$)
⟨*proof*⟩

lemma *fgl-invar-collapse-nce*:
fixes $u\ v$
assumes *INV*: *fgl-invar v0 D0* $(None,p,D,pE)$
defines $pE' \equiv pE - \{(u,v)\}$
assumes *CFMT*: $(p',D',pE'') = collapse\ v\ (p,D,pE')$
assumes *INV'*: *invar v0 D0* (p',D',pE'')
assumes *NE*[*simp*]: $p{\neq}[]$
assumes *E*: $(u,v){\in}pE$ **and** $u{\in}last\ p$
assumes *BACK*: $v{\in}\bigcup(set\ p)$
assumes *NACC*: $j{<}num\text{-}acc \quad \forall q{\in}last\ p'.\ j{\notin}acc\ q$
defines $i \equiv idx\text{-}of\ p\ v$
shows *fgl-invar-part* $(None,\ collapse\ v\ (p,D,pE'))$
⟨*proof*⟩

lemma *collapse-ne*: $([],D',pE') \neq collapse\ v\ (p,D,pE)$
⟨*proof*⟩

lemma *fgl-invar-push*:
assumes *INV*: *fgl-invar v0 D0* $(None,p,D,pE)$
assumes *BRK*[*simp*]: *brk=None*
assumes *NE*[*simp*]: $p{\neq}[]$
assumes *E*: $(u,v){\in}pE$ **and** *UIL*: $u{\in}last\ p$
assumes *VNE*: $v{\notin}\bigcup(set\ p) \quad v{\notin}D$
assumes *INV'*: *invar v0 D0* $(push\ v\ (p,D,pE - \{(u,v)\}))$
shows *fgl-invar-part* $(None,\ push\ v\ (p,D,pE - \{(u,v)\}))$
⟨*proof*⟩

lemma *fgl-invar-skip*:
assumes *INV*: *fgl-invar v0 D0* $(None,p,D,pE)$
assumes *BRK*[*simp*]: *brk=None*
assumes *NE*[*simp*]: $p{\neq}[]$
assumes *E*: $(u,v){\in}pE$ **and** *UIL*: $u{\in}last\ p$
assumes *VID*: $v{\in}D$
assumes *INV'*: *invar v0 D0* $(p,\ D,\ (pE - \{(u,v)\}))$
shows *fgl-invar-part* $(None,\ p,\ D,\ (pE - \{(u,v)\}))$
⟨*proof*⟩

lemma *fgl-outer-invar-initial*:
outer-invar V0 $\{\} \Longrightarrow$ *fgl-outer-invar-ext V0* $(None,\ \{\})$
⟨*proof*⟩

lemma *fgl-outer-invar-brk*:
assumes *INV*: *fgl-invar v0 D0* (*Some* (*Vr*, *Vl*),*p*,*D*,*pE*)
shows *fgl-outer-invar-ext anyIt* (*Some* (*Vr*, *Vl*), *anyD*)
⟨*proof*⟩

lemma *fgl-outer-invar-newnode-nobrk*:
assumes *A*: *v0*∉*D0* *v0*∈*it*
assumes *OINV*: *fgl-outer-invar it* (*None*,*D0*)
assumes *INV*: *fgl-invar v0 D0* (*None*,[],*D′*,*pE*)
shows *fgl-outer-invar-ext* (*it*−{*v0*}) (*None*,*D′*)
⟨*proof*⟩

lemma *fgl-outer-invar-newnode*:
assumes *A*: *v0*∉*D0* *v0*∈*it*
assumes *OINV*: *fgl-outer-invar it* (*None*,*D0*)
assumes *INV*: *fgl-invar v0 D0* (*brk*,*p*,*D′*,*pE*)
assumes *CASES*: (∃ *Vr Vl*. *brk* = *Some* (*Vr*, *Vl*)) ∨ *p* = []
shows *fgl-outer-invar-ext* (*it*−{*v0*}) (*brk*,*D′*)
⟨*proof*⟩

lemma *fgl-outer-invar-Dnode*:
assumes *fgl-outer-invar it* (*None*, *D*) *v*∈*D*
shows *fgl-outer-invar-ext* (*it* − {*v*}) (*None*, *D*)
⟨*proof*⟩

lemma *fgl-fin-no-lasso*:
assumes *A*: *fgl-outer-invar* {} (*None*, *D*)
assumes *B*: *is-lasso-prpl prpl*
shows *False*
⟨*proof*⟩

lemma *fgl-fin-lasso*:
assumes *A*: *fgl-outer-invar it* (*Some* (*Vr*, *Vl*), *D*)
shows *ce-correct Vr Vl*
⟨*proof*⟩

lemmas *fgl-invar-preserve* =
fgl-invar-initial fgl-invar-push fgl-invar-pop
fgl-invar-collapse-ce fgl-invar-collapse-nce fgl-invar-skip
fgl-outer-invar-newnode fgl-outer-invar-Dnode
invar-initial outer-invar-initial fgl-invar-initial fgl-outer-invar-initial
fgl-fin-no-lasso fgl-fin-lasso

end

1.19 Main Correctness Proof

context *igb-fr-graph*
begin
lemma *outer-invar-from-fgl-invarI*:
fgl-outer-invar it (*None*,*D*) $\Longrightarrow$ *outer-invar it D*
⟨*proof*⟩

lemma *invar-from-fgl-invarI*: *fgl-invar v0 D0* (*B*,*PDPE*) $\Longrightarrow$ *invar v0 D0 PDPE*
⟨*proof*⟩

theorem *find-ce-correct*: *find-ce* $\leq$ *find-ce-spec*
⟨*proof*⟩
end

1.20 Emptiness Check

Using the lasso-finding algorithm, we can define an emptiness check

context *igb-fr-graph*
begin
definition *abs-is-empty* $\equiv$ *do* {
ce $\leftarrow$ *find-ce*;
RETURN (*ce* = *None*)
}

theorem *abs-is-empty-correct*:
abs-is-empty $\leq$ *SPEC* (λ*res*. *res* $\longleftrightarrow$ ($\forall$ *r*. $\neg$*is-acc-run r*))
⟨*proof*⟩

definition *abs-is-empty-ce* $\equiv$ *do* {
ce $\leftarrow$ *find-ce*;
case ce of
None $\Rightarrow$ *RETURN None*
| *Some* (*Vr*, *Vl*) $\Rightarrow$ *do* {
ASSERT ($\exists$ *pr pl*. *set pr* $\subseteq$ *Vr* $\wedge$ *set pl* $\subseteq$ *Vl* $\wedge$ *Vl* $\times$ *Vl* $\subseteq$ (*E* $\cap$ *Vl*$\times$*Vl*)*
$\wedge$ *is-lasso-prpl* (*pr*,*pl*));
(*pr*,*pl*) $\leftarrow$ *SPEC* (λ(*pr*,*pl*).
set pr $\subseteq$ *Vr*
$\wedge$ *set pl* $\subseteq$ *Vl*
$\wedge$ *Vl* $\times$ *Vl* $\subseteq$ (*E* $\cap$ *Vl*$\times$*Vl*)*
$\wedge$ *is-lasso-prpl* (*pr*,*pl*));
RETURN (*Some* (*pr*,*pl*))
}
}

theorem *abs-is-empty-ce-correct*: *abs-is-empty-ce* $\leq$ *SPEC* (λ*res*. *case res of*
None $\Rightarrow$ ($\forall$ *r*. $\neg$*is-acc-run r*)
| *Some* (*pr*,*pl*) $\Rightarrow$ *is-acc-run* (*pr*$\frown$*pl*$^\omega$)
)

⟨*proof*⟩

end

1.21 Refinement

In this section, we refine the lasso finding algorithm to use efficient data structures. First, we explicitely keep track of the set of acceptance classes for every c-node on the path. Second, we use Gabow's data structure to represent the path.

1.21.1 Addition of Explicit Accepting Sets

In a first step, we explicitely keep track of the current set of acceptance classes for every c-node on the path.

type-synonym *'a abs-gstate = nat set list × 'a abs-state*
type-synonym *'a ce = ('a set × 'a set) option*
type-synonym *'a abs-gostate = 'a ce × 'a set*

context *igb-fr-graph*
begin

definition *gstate-invar :: 'Q abs-gstate ⇒ bool* **where**
gstate-invar ≡ λ(a,p,D,pE). a = map (λV. ⋃(acc‘V)) p

definition *gstate-rel ≡ br snd gstate-invar*

lemma *gstate-rel-sv*[*relator-props*,*simp*,*intro*!]: *single-valued gstate-rel*
⟨*proof*⟩

definition (**in** −) *gcollapse-aux*
:: *nat set list ⇒ 'a set list ⇒ nat ⇒ nat set list × 'a set list*
where *gcollapse-aux a p i ≡*
(take i a @ [⋃(set (drop i a))],take i p @ [⋃(set (drop i p))])

definition (**in** −) *gcollapse :: 'a ⇒ 'a abs-gstate ⇒ 'a abs-gstate*
where *gcollapse v APDPE ≡*
let
(a,p,D,pE)=APDPE;
i=idx-of p v;
(a,p) = gcollapse-aux a p i
in (a,p,D,pE)

definition *gpush v s ≡*
let
(a,s) = s
in
(a@[acc v],push v s)

```
definition gpop s ≡
  let (a,s) = s in (butlast a,pop s)

definition ginitial :: 'Q ⇒ 'Q abs-gostate ⇒ 'Q abs-gstate
  where ginitial v0 s0 ≡ ([acc v0], initial v0 (snd s0))

definition goinitial :: 'Q abs-gostate where goinitial ≡ (None,{})
definition go-is-no-brk :: 'Q abs-gostate ⇒ bool
  where go-is-no-brk s ≡ fst s = None
definition goD :: 'Q abs-gostate ⇒ 'Q set where goD s ≡ snd s
definition goBrk :: 'Q abs-gostate ⇒ 'Q ce where goBrk s ≡ fst s
definition gto-outer :: 'Q ce ⇒ 'Q abs-gstate ⇒ 'Q abs-gostate
  where gto-outer brk s ≡ let (A,p,D,pE)=s in (brk,D)

definition gselect-edge s ≡ do {
  let (a,s)=s;
  (r,s)←select-edge s;
  RETURN (r,a,s)
}

definition gfind-ce :: ('Q set × 'Q set) option nres where
  gfind-ce ≡ do {
    let os = goinitial;
    os←FOREACHci fgl-outer-invar V0 (go-is-no-brk) (λv0 s0. do {
      if v0∉goD s0 then do {
        let s = (None,ginitial v0 s0);

        (brk,a,p,D,pE) ← WHILEIT (λ(brk,a,s). fgl-invar v0 (goD s0) (brk,s))
          (λ(brk,a,p,D,pE). brk=None ∧ p ≠ []) (λ(-,a,p,D,pE).
        do {
          — Select edge from end of path
          (vo,(a,p,D,pE)) ← gselect-edge (a,p,D,pE);

          ASSERT (p≠[]);
          case vo of
            Some v ⇒ do {
              if v ∈ ⋃(set p) then do {
                — Collapse
                let (a,p,D,pE) = gcollapse v (a,p,D,pE);

                ASSERT (p≠[]);
                ASSERT (a≠[]);

                if last a = {0..<num-acc} then
                  RETURN (Some (⋃(set (butlast p)),last p),a,p,D,pE)
                else
                  RETURN (None,a,p,D,pE)
              } else if v∉D then do {
```

```
            — Edge to new node. Append to path
            RETURN (None,gpush v (a,p,D,pE))
          } else RETURN (None,a,p,D,pE)
        }
      | None ⇒ do {
          — No more outgoing edges from current node on path
          ASSERT (pE ∩ last p × UNIV = {});
          RETURN (None,gpop (a,p,D,pE))
        }
    }) s;
    ASSERT (brk=None ⟶ (p=[] ∧ pE={}));
    RETURN (gto-outer brk (a,p,D,pE))
  } else RETURN s0
}) os;
RETURN (goBrk os)
}

lemma gcollapse-refine:
  ⟦(v',v)∈Id; (s',s)∈gstate-rel⟧
    ⟹ (gcollapse v' s',collapse v s)∈gstate-rel
  ⟨proof⟩

lemma gpush-refine:
  ⟦(v',v)∈Id; (s',s)∈gstate-rel⟧ ⟹ (gpush v' s',push v s)∈gstate-rel
  ⟨proof⟩

lemma gpop-refine:
  ⟦(s',s)∈gstate-rel⟧ ⟹ (gpop s',pop s)∈gstate-rel
  ⟨proof⟩

lemma ginitial-refine:
  (ginitial x (None, b), initial x b) ∈ gstate-rel
  ⟨proof⟩

lemma oinitial-b-refine: ((None,{}),(None,{}))∈Id×ᵣId ⟨proof⟩

lemma gselect-edge-refine: ⟦(s',s)∈gstate-rel⟧ ⟹ gselect-edge s'
  ≤⇓(⟨Id⟩option-rel ×ᵣ gstate-rel) (select-edge s)
  ⟨proof⟩

lemma last-acc-impl:
  assumes p≠[]
  assumes ((a',p',D',pE'),(p,D,pE))∈gstate-rel
  shows (last a' = {0..<num-acc}) = (∀ i<num-acc. ∃ q∈last p. i∈acc q)
  ⟨proof⟩

lemma fglr-aux1:
  assumes V: (v',v)∈Id and S: (s',s)∈gstate-rel
    and P: ⋀a' p' D' pE' p D pE. ((a',p',D',pE'),(p,D,pE))∈gstate-rel
```

$\Longrightarrow f'\ a'\ p'\ D'\ pE' \leq \Downarrow R\ (f\ p\ D\ pE)$
shows $(let\ (a',p',D',pE') = gcollapse\ v'\ s'\ in\ f'\ a'\ p'\ D'\ pE')$
$\leq \Downarrow R\ (let\ (p,D,pE) = collapse\ v\ s\ in\ f\ p\ D\ pE)$
⟨*proof*⟩

lemma *gstate-invar-empty*:
$gstate\text{-}invar\ (a,[],D,pE) \Longrightarrow a=[]$
$gstate\text{-}invar\ ([],p,D,pE) \Longrightarrow p=[]$
⟨*proof*⟩

lemma *find-ce-refine*: $gfind\text{-}ce \leq \Downarrow Id\ find\text{-}ce$
⟨*proof*⟩
end

1.21.2 Refinement to Gabow's Data Structure

Preliminaries **definition** $Un\text{-}set\text{-}drop\text{-}impl :: nat \Rightarrow 'a\ set\ list \Rightarrow 'a\ set\ nres$
— Executable version of $\bigcup set\ (drop\ i\ A)$, using indexing to access A
where $Un\text{-}set\text{-}drop\text{-}impl\ i\ A \equiv$
do {
$(\text{-},res) \leftarrow WHILET\ (\lambda(i,res).\ i < length\ A)\ (\lambda(i,res).\ do\ \{$
$ASSERT\ (i<length\ A)$;
$let\ res = A!i \cup res$;
$let\ i = i + 1$;
$RETURN\ (i,res)$
}) $(i,\{\})$;
$RETURN\ res$
}

lemma *Un-set-drop-impl-correct*:
$Un\text{-}set\text{-}drop\text{-}impl\ i\ A \leq SPEC\ (\lambda r.\ r=\bigcup(set\ (drop\ i\ A)))$
⟨*proof*⟩

schematic-goal *Un-set-drop-code-aux*:
assumes [*autoref-rules*]: $(es\text{-}impl,\{\})\in\langle R\rangle Rs$
assumes [*autoref-rules*]: $(un\text{-}impl,(\cup))\in\langle R\rangle Rs\rightarrow\langle R\rangle Rs\rightarrow\langle R\rangle Rs$
shows $(?c, Un\text{-}set\text{-}drop\text{-}impl)\in nat\text{-}rel \rightarrow \langle\langle R\rangle Rs\rangle as\text{-}rel \rightarrow \langle\langle R\rangle Rs\rangle nres\text{-}rel$
⟨*proof*⟩
concrete-definition *Un-set-drop-code* **uses** *Un-set-drop-code-aux*

schematic-goal *Un-set-drop-tr-aux*:
$RETURN\ ?c \leq Un\text{-}set\text{-}drop\text{-}code\ es\text{-}impl\ un\text{-}impl\ i\ A$
⟨*proof*⟩
concrete-definition *Un-set-drop-tr* **for** *es-impl un-impl i A*
uses *Un-set-drop-tr-aux*

lemma *Un-set-drop-autoref*[*autoref-rules*]:
assumes $GEN\text{-}OP\ es\text{-}impl\ \{\}\ (\langle R\rangle Rs)$
assumes $GEN\text{-}OP\ un\text{-}impl\ (\cup)\ (\langle R\rangle Rs\rightarrow\langle R\rangle Rs\rightarrow\langle R\rangle Rs)$

shows $(\lambda i\ A.\ RETURN\ (Un\text{-}set\text{-}drop\text{-}tr\ es\text{-}impl\ un\text{-}impl\ i\ A), Un\text{-}set\text{-}drop\text{-}impl)$
$\in nat\text{-}rel \to \langle\langle R\rangle Rs\rangle as\text{-}rel \to \langle\langle R\rangle Rs\rangle nres\text{-}rel$
⟨*proof*⟩

Actual Refinement **type-synonym** $'Q\ gGS = nat\ set\ list \times 'Q\ GS$

type-synonym $'Q\ goGS = 'Q\ ce \times 'Q\ oGS$

context *igb-graph*
begin

definition $gGS\text{-}invar :: 'Q\ gGS \Rightarrow bool$
where $gGS\text{-}invar\ s \equiv$
let $(a,S,B,I,P) = s$ *in*
$GS\text{-}invar\ (S,B,I,P)$
$\wedge\ length\ a = length\ B$
$\wedge\ \bigcup(set\ a) \subseteq \{0..<num\text{-}acc\}$

definition $gGS\text{-}\alpha :: 'Q\ gGS \Rightarrow 'Q\ abs\text{-}gstate$
where $gGS\text{-}\alpha\ s \equiv let\ (a,s)=s\ in\ (a,GS.\alpha\ s)$

definition $gGS\text{-}rel \equiv br\ gGS\text{-}\alpha\ gGS\text{-}invar$

lemma *gGS-rel-sv*[*relator-props*,*intro*!,*simp*]: *single-valued gGS-rel*
⟨*proof*⟩

definition $goGS\text{-}invar :: 'Q\ goGS \Rightarrow bool$ **where**
$goGS\text{-}invar\ s \equiv let\ (brk,ogs)=s\ in\ brk=None \longrightarrow oGS\text{-}invar\ ogs$

definition $goGS\text{-}\alpha\ s \equiv let\ (brk,ogs)=s\ in\ (brk,oGS\text{-}\alpha\ ogs)$

definition $goGS\text{-}rel \equiv br\ goGS\text{-}\alpha\ goGS\text{-}invar$

lemma *goGS-rel-sv*[*relator-props*,*intro*!,*simp*]: *single-valued goGS-rel*
⟨*proof*⟩

end

context *igb-fr-graph*
begin
lemma *gGS-relE*:
assumes $(s',(a,p,D,pE))\in gGS\text{-}rel$
obtains $S'\ B'\ I'\ P'$ **where** $s'=(a,S',B',I',P')$
and $((S',B',I',P'),(p,D,pE))\in GS\text{-}rel$
and $length\ a = length\ B'$
and $\bigcup(set\ a) \subseteq \{0..<num\text{-}acc\}$

⟨*proof*⟩

definition *goinitial-impl* :: *'Q goGS*
where *goinitial-impl* ≡ (*None*,*Map.empty*)
lemma *goinitial-impl-refine*: (*goinitial-impl*,*goinitial*)∈*goGS-rel*
⟨*proof*⟩

definition *gto-outer-impl* :: *'Q ce* ⇒ *'Q gGS* ⇒ *'Q goGS*
where *gto-outer-impl brk s* ≡ *let* (*A*,*S*,*B*,*I*,*P*)=*s in* (*brk*,*I*)

lemma *gto-outer-refine*:
assumes *A*: *brk* = *None* ⟶ (*p*=[] ∧ *pE*={})
assumes *B*: (*s*, (*A*,*p*, *D*, *pE*)) ∈ *gGS-rel*
assumes *C*: (*brk'*,*brk*)∈*Id*
shows (*gto-outer-impl brk' s*,*gto-outer brk* (*A*,*p*,*D*,*pE*))∈*goGS-rel*
⟨*proof*⟩

definition *gpush-impl v s* ≡ *let* (*a*,*s*)=*s in* (*a*@[*acc v*], *push-impl v s*)

lemma *gpush-impl-refine*:
assumes *B*: (*s'*,(*a*,*p*,*D*,*pE*))∈*gGS-rel*
assumes *A*: (*v'*,*v*)∈*Id*
assumes *PRE*: *v'* ∉ ⋃(*set p*) *v'* ∉ *D*
shows (*gpush-impl v' s'*, *gpush v* (*a*,*p*,*D*,*pE*))∈*gGS-rel*
⟨*proof*⟩

definition *gpop-impl* :: *'Q gGS* ⇒ *'Q gGS nres*
where *gpop-impl s* ≡ *do* {
let (*a*,*s*)=*s*;
s←*pop-impl s*;
ASSERT (*a*≠[]);
let a = *butlast a*;
RETURN (*a*,*s*)
}

lemma *gpop-impl-refine*:
assumes *A*: (*s'*,(*a*,*p*,*D*,*pE*))∈*gGS-rel*
assumes *PRE*: *p* ≠ [] *pE* ∩ *last p* × *UNIV* = {}
shows *gpop-impl s'* ≤ ⇓*gGS-rel* (*RETURN* (*gpop* (*a*,*p*,*D*,*pE*)))
⟨*proof*⟩

definition *gselect-edge-impl* :: *'Q gGS* ⇒ (*'Q option* × *'Q gGS*) *nres*
where *gselect-edge-impl s* ≡
do {
let (*a*,*s*)=*s*;
(*vo*,*s*)←*select-edge-impl s*;
RETURN (*vo*,*a*,*s*)

}

thm *select-edge-refine*
lemma *gselect-edge-impl-refine*:
assumes *A*: $(s', a, p, D, pE) \in gGS\text{-}rel$
assumes *PRE*: $p \neq []$
shows $gselect\text{-}edge\text{-}impl\ s' \leq \Downarrow(Id \times_r gGS\text{-}rel)\ (gselect\text{-}edge\ (a, p, D, pE))$
⟨*proof*⟩

term *GS.idx-of-impl*

thm *GS-invar.idx-of-correct*

definition $gcollapse\text{-}impl\text{-}aux :: 'Q \Rightarrow 'Q\ gGS \Rightarrow 'Q\ gGS\ nres$ **where**
gcollapse-impl-aux v s ≡
do {
let (*A*,*s*)=*s*;
~~*ASSERT* (*v* ∈ ⋃ (*set* (*fst* (*GS.dest s*))));~~
i ← *GS.idx-of-impl s v*;
s ← *collapse-impl v s*;
ASSERT (*i* < *length A*);
us ← *Un-set-drop-impl i A*;
let A = *take i A* @ [*us*];
RETURN (*A*,*s*)
}

term *collapse*
lemma *gcollapse-alt*:
gcollapse v APDPE = (
let
(*a*,*p*,*D*,*pE*)=*APDPE*;
i=*idx-of p v*;
s=*collapse v* (*p*,*D*,*pE*);
us=⋃(*set* (*drop i a*));
a = *take i a* @ [*us*]
in (*a*,*s*))
⟨*proof*⟩

thm *collapse-refine*
lemma *gcollapse-impl-aux-refine*:
assumes *A*: $(s', a, p, D, pE) \in gGS\text{-}rel$
assumes *B*: $(v', v) \in Id$
assumes *PRE*: $v \in \bigcup(set\ p)$
shows $gcollapse\text{-}impl\text{-}aux\ v'\ s'$
$\leq \Downarrow\ gGS\text{-}rel\ (RETURN\ (gcollapse\ v\ (a, p, D, pE)))$
⟨*proof*⟩

definition *gcollapse-impl* :: *'Q ⇒ 'Q gGS ⇒ 'Q gGS nres*
where *gcollapse-impl v s ≡*
do {
let (A,S,B,I,P)=s;
i ← GS.idx-of-impl (S,B,I,P) v;
ASSERT (i+1 ≤ length B);
let B = take (i+1) B;
ASSERT (i < length A);
us←Un-set-drop-impl i A;
let A = take i A @ [us];
RETURN (A,S,B,I,P)
}

lemma *gcollapse-impl-aux-opt-refine*:
gcollapse-impl v s ≤ gcollapse-impl-aux v s
⟨*proof*⟩

lemma *gcollapse-impl-refine*:
assumes *A*: *(s', a, p, D, pE) ∈ gGS-rel*
assumes *B*: *(v',v)∈Id*
assumes *PRE*: *v∈⋃(set p)*
shows *gcollapse-impl v' s'*
≤ ⇓ gGS-rel (RETURN (gcollapse v (a, p, D, pE)))
⟨*proof*⟩

definition *ginitial-impl* :: *'Q ⇒ 'Q goGS ⇒ 'Q gGS*
where *ginitial-impl v0 s0 ≡ ([acc v0],initial-impl v0 (snd s0))*
lemma *ginitial-impl-refine*:
assumes *A*: *v0∉goD s0 go-is-no-brk s0*
assumes *REL*: *(s0i,s0)∈goGS-rel (v0i,v0)∈Id*
shows *(ginitial-impl v0i s0i,ginitial v0 s0)∈gGS-rel*
⟨*proof*⟩

definition *gpath-is-empty-impl* :: *'Q gGS ⇒ bool*
where *gpath-is-empty-impl s = path-is-empty-impl (snd s)*

lemma *gpath-is-empty-refine*:
(s,(a,p,D,pE))∈gGS-rel ⟹ gpath-is-empty-impl s ⟷ p=[]
⟨*proof*⟩

definition *gis-on-stack-impl* :: *'Q ⇒ 'Q gGS ⇒ bool*
where *gis-on-stack-impl v s = is-on-stack-impl v (snd s)*

lemma *gis-on-stack-refine*:
⟦(s,(a,p,D,pE))∈gGS-rel⟧ ⟹ gis-on-stack-impl v s ⟷ v∈⋃(set p)
⟨*proof*⟩

definition *gis-done-impl* :: *'Q ⇒ 'Q gGS ⇒ bool*
where *gis-done-impl v s ≡ is-done-impl v (snd s)*

thm *is-done-refine*
lemma *gis-done-refine*: $(s,(a,p,D,pE))\in gGS\text{-}rel$
$\Longrightarrow$ *gis-done-impl v s* $\longleftrightarrow$ $(v \in D)$
⟨*proof*⟩

definition (**in** −) *on-stack-less I u v* ≡
case I v of
Some (*STACK j*) ⇒ $j<u$
| - ⇒ *False*

definition (**in** −) *on-stack-ge I l v* ≡
case I v of
Some (*STACK j*) ⇒ $l\leq j$
| - ⇒ *False*

lemma (**in** *GS-invar*) *set-butlast-p-refine*:
assumes *PRE*: $p\text{-}\alpha\neq[]$
shows *Collect* (*on-stack-less I* (*last B*)) = $\bigcup$(*set* (*butlast p-α*)) (**is** *?L=?R*)
⟨*proof*⟩

lemma (**in** *GS-invar*) *set-last-p-refine*:
assumes *PRE*: $p\text{-}\alpha\neq[]$
shows *Collect* (*on-stack-ge I* (*last B*)) = *last p-α* (**is** *?L=?R*)
⟨*proof*⟩

definition *ce-impl* :: *'Q gGS* ⇒ ((*'Q set* × *'Q set*) *option* × *'Q gGS*) *nres*
where *ce-impl s* ≡
do {
let (*a*,*S*,*B*,*I*,*P*) = *s*;
ASSERT (*B*≠[]);
let bls = *Collect* (*on-stack-less I* (*last B*));
let ls = *Collect* (*on-stack-ge I* (*last B*));
RETURN (*Some* (*bls*, *ls*),*a*,*S*,*B*,*I*,*P*)
}

lemma *ce-impl-refine*:
assumes *A*: $(s,(a,p,D,pE))\in gGS\text{-}rel$
assumes *PRE*: $p\neq[]$
shows *ce-impl s* ≤ ⇓($Id\times_r gGS\text{-}rel$)
(*RETURN* (*Some* ($\bigcup$(*set* (*butlast p*)),*last p*),*a*,*p*,*D*,*pE*))
⟨*proof*⟩

definition *last-is-acc-impl s* ≡
do {
let (*a*,-)=*s*;
ASSERT (*a*≠[]);
RETURN (∀ *i*<*num-acc*. *i*∈*last a*)

}

lemma *last-is-acc-impl-refine*:
assumes *A*: $(s,(a,p,D,pE))\in gGS\text{-}rel$
assumes *PRE*: $a\neq[]$
shows *last-is-acc-impl* $s \leq$ *RETURN* $(last\ a = \{0..<num\text{-}acc\})$
⟨*proof*⟩

definition *go-is-no-brk-impl* :: $'Q\ goGS \Rightarrow bool$
where *go-is-no-brk-impl* $s \equiv fst\ s = None$
lemma *go-is-no-brk-refine*:
$(s,s')\in goGS\text{-}rel \Longrightarrow go\text{-}is\text{-}no\text{-}brk\text{-}impl\ s \longleftrightarrow go\text{-}is\text{-}no\text{-}brk\ s'$
⟨*proof*⟩

definition *goD-impl* :: $'Q\ goGS \Rightarrow 'Q\ oGS$ **where** *goD-impl* $s \equiv snd\ s$
lemma *goD-refine*:
go-is-no-brk $s' \Longrightarrow (s,s')\in goGS\text{-}rel \Longrightarrow (goD\text{-}impl\ s,\ goD\ s')\in oGS\text{-}rel$
⟨*proof*⟩

definition *go-is-done-impl* :: $'Q \Rightarrow 'Q\ goGS \Rightarrow bool$
where *go-is-done-impl* $v\ s \equiv is\text{-}done\text{-}oimpl\ v\ (snd\ s)$
thm *is-done-orefine*
lemma *go-is-done-impl-refine*: $[\![go\text{-}is\text{-}no\text{-}brk\ s';\ (s,s')\in goGS\text{-}rel;\ (v,v')\in Id]\!]$
$\Longrightarrow go\text{-}is\text{-}done\text{-}impl\ v\ s \longleftrightarrow (v'\in goD\ s')$
⟨*proof*⟩

definition *goBrk-impl* :: $'Q\ goGS \Rightarrow 'Q\ ce$ **where** *goBrk-impl* $\equiv fst$

lemma *goBrk-refine*: $(s,s')\in goGS\text{-}rel \Longrightarrow (goBrk\text{-}impl\ s,\ goBrk\ s')\in Id$
⟨*proof*⟩

definition *find-ce-impl* :: $('Q\ set \times 'Q\ set)\ option\ nres$ **where**
find-ce-impl $\equiv$ *do* {
stat-start-nres;
let os=goinitial-impl;
os←*FOREACHci* $(\lambda it\ os.\ fgl\text{-}outer\text{-}invar\ it\ (goGS\text{-}\alpha\ os))\ V0$
(*go-is-no-brk-impl*) $(\lambda v0\ s0.$
do {
if ¬*go-is-done-impl v0 s0 then do* {

let s = (*None*,*ginitial-impl v0 s0*);

(*brk*,*s*)←*WHILEIT*
$(\lambda(brk,s).\ fgl\text{-}invar\ v0\ (oGS\text{-}\alpha\ (goD\text{-}impl\ s0))\ (brk,snd\ (gGS\text{-}\alpha\ s)))$
$(\lambda(brk,s).\ brk=None \wedge \neg gpath\text{-}is\text{-}empty\text{-}impl\ s)\ (\lambda(l,s).$
do {
— Select edge from end of path
(*vo*,*s*) ← *gselect-edge-impl s*;

```
        case vo of
          Some v ⇒ do {
            if gis-on-stack-impl v s then do {
              s←gcollapse-impl v s;
              b←last-is-acc-impl s;
              if b then
                ce-impl s
              else
                RETURN (None,s)
            } else if ¬gis-done-impl v s then do {
              — Edge to new node. Append to path
              RETURN (None,gpush-impl v s)
            } else do {
              — Edge to done node. Skip
              RETURN (None,s)
            }
          }
        | None ⇒ do {
            — No more outgoing edges from current node on path
            s←gpop-impl s;
            RETURN (None,s)
          }
      }) (s);
      RETURN (gto-outer-impl brk s)
    } else RETURN s0
  }) os;
  stat-stop-nres;
  RETURN (goBrk-impl os)
}
```

lemma *find-ce-impl-refine*: *find-ce-impl* ≤ ⇓*Id gfind-ce*
⟨*proof*⟩

end

1.22 Constructing a Lasso from Counterexample

1.22.1 Lassos in GBAs

context *igb-fr-graph* **begin**

```
definition reconstruct-reach :: 'Q set ⇒ 'Q set ⇒ ('Q list × 'Q) nres
  — Reconstruct the reaching path of a lasso
  where reconstruct-reach Vr Vl ≡ do {
    res ← find-path (E∩Vr×UNIV) V0 (λv. v∈Vl);
    ASSERT (res ≠ None);
    RETURN (the res)
  }
```

lemma *reconstruct-reach-correct*:
assumes *CEC*: *ce-correct Vr Vl*
shows *reconstruct-reach Vr Vl*
$\leq$ *SPEC* (λ(*pr*,*va*). $\exists$ *v0*$\in$*V0*. *path E v0 pr va* $\wedge$ *va*$\in$*Vl*)
⟨*proof*⟩

definition *rec-loop-invar Vl va s* $\equiv$ *let* (*v*,*p*,*cS*) = *s in*
va $\in$ E^{*} ‘‘*V0* $\wedge$
path E va p v $\wedge$
cS = *acc v* $\cup$ ($\bigcup$(*acc*‘*set p*)) $\wedge$
va $\in$ *Vl* $\wedge$ *v* $\in$ *Vl* $\wedge$ *set p* $\subseteq$ *Vl*

definition *reconstruct-lasso* :: *'Q set* $\Rightarrow$ *'Q set* $\Rightarrow$ (*'Q list* $\times$ *'Q list*) *nres*
— Reconstruct lasso
where *reconstruct-lasso Vr Vl* $\equiv$ *do* {
(*pr*,*va*) $\leftarrow$ *reconstruct-reach Vr Vl*;

let cS-full = {*0*..<*num-acc*};
let E = *E* $\cap$ *UNIV*$\times$*Vl*;

(*vd*,*p*,-) $\leftarrow$ *WHILEIT* (*rec-loop-invar Vl va*)
(λ(-,-,*cS*). *cS* $\neq$ *cS-full*)
(λ(*v*,*p*,*cS*). *do* {
ASSERT ($\exists$ *v'*. (*v*,*v'*)$\in$$E^{*}$ $\wedge$ $\neg$ (*acc v'* $\subseteq$ *cS*));
sr $\leftarrow$ *find-path E* {*v*} (λ*v*. $\neg$ (*acc v* $\subseteq$ *cS*));
ASSERT (*sr* $\neq$ *None*);
let (*p-seg*,*v*) = *the sr*;
RETURN (*v*,*p*@*p-seg*,*cS* $\cup$ *acc v*)
}) (*va*,[],*acc va*);

p-close-r $\leftarrow$ (*if p*=[] *then*
find-path1 E vd ((=) *va*)
else
find-path E {*vd*} ((=) *va*));

ASSERT (*p-close-r* $\neq$ *None*);
let (*p-close*,-) = *the p-close-r*;

RETURN (*pr*, *p*@*p-close*)
}

lemma (**in** *igb-fr-graph*) *reconstruct-lasso-correct*:
assumes *CEC*: *ce-correct Vr Vl*
shows *reconstruct-lasso Vr Vl* $\leq$ *SPEC* (*is-lasso-prpl*)
⟨*proof*⟩

definition *find-lasso* **where** *find-lasso* $\equiv$ *do* {
ce $\leftarrow$ *find-ce-spec*;

```
  case ce of
    None ⇒ RETURN None
  | Some (Vr, Vl) ⇒ do {
      l ← reconstruct-lasso Vr Vl;
      RETURN (Some l)
    }
}

lemma (in igb-fr-graph) find-lasso-correct: find-lasso ≤ find-lasso-spec
  ⟨proof⟩

end

end
```

1.23 Code Generation for the Skeleton Algorithm

```
theory Gabow-Skeleton-Code
imports
  Gabow-Skeleton
  CAVA-Automata.Digraph-Impl
  CAVA-Base.CAVA-Code-Target
begin
```

1.24 Statistics

In this section, we do the ML setup that gathers statistics about the algorithm's execution.

```
code-printing
  code-module Gabow-Skeleton-Statistics ⇀ (SML) ‹
    structure Gabow-Skeleton-Statistics = struct
      val active = Unsynchronized.ref false
      val num-vis = Unsynchronized.ref 0

      val time = Unsynchronized.ref Time.zeroTime

      fun is-active () = !active
      fun newnode () =
      (
        num-vis := !num-vis + 1;
        if !num-vis mod 10000 = 0 then tracing (IntInf.toString (!num-vis) ^ "\n")
else ()
      )

      fun start () = (active := true; time := Time.now ())
      fun stop () = (time := Time.- (Time.now (), !time))

      fun to-string () = let
        val t = Time.toMilliseconds (!time)
```

```
    val states-per-ms = real (!num-vis) / real t
    val realStr = Real.fmt (StringCvt.FIX (SOME 2))
  in
    Required time:  ^ IntInf.toString (t) ^ ms\n
  ^ States per ms:  ^ realStr states-per-ms ^ \n
  ^ # states:  ^ IntInf.toString (!num-vis) ^ \n
  end

  val - = Statistics.register-stat (Gabow−Skeleton,is-active,to-string)

end
›
code-reserved (SML) Gabow-Skeleton-Statistics

code-printing
  constant stat-newnode ⇀ (SML) Gabow′-Skeleton′-Statistics.newnode
| constant stat-start ⇀ (SML) Gabow′-Skeleton′-Statistics.start
| constant stat-stop ⇀ (SML) Gabow′-Skeleton′-Statistics.stop
```

1.25 Automatic Refinement Setup

```
consts i-node-state :: interface

definition node-state-rel ≡ {(−1::int,DONE)} ∪ {(int k,STACK k) | k. True }
lemma node-state-rel-simps[simp]:
  (i,DONE)∈node-state-rel ⟷ i=−1
  (i,STACK n)∈node-state-rel ⟷ i = int n
  ⟨proof⟩

lemma node-state-rel-sv[simp,intro!,relator-props]:
  single-valued node-state-rel
  ⟨proof⟩

lemmas [autoref-rel-intf] = REL-INTFI[of node-state-rel i-node-state]

primrec is-DONE where
  is-DONE DONE = True
| is-DONE (STACK -) = False

lemma node-state-rel-refine[autoref-rules]:
  (−1,DONE)∈node-state-rel
  (int,STACK)∈nat-rel→node-state-rel
  (λi. i<0,is-DONE)∈node-state-rel→bool-rel
  ((λf g i. if i≥0 then f (nat i) else g),case-node-state)
    ∈(nat-rel → R) → R → node-state-rel → R
  ⟨proof⟩

lemma [autoref-op-pat]:
  (x=DONE) ≡ is-DONE x
```

$(DONE{=}x) \equiv is\text{-}DONE\ x$
⟨*proof*⟩

consts *i-node* :: *interface*

locale *fr-graph-impl-loc* = *fr-graph G*
for *mrel* **and** *node-rel* :: $('vi \times 'v)\ set$
and *node-eq-impl* :: $'vi \Rightarrow 'vi \Rightarrow bool$
and *node-hash-impl* :: $nat \Rightarrow 'vi \Rightarrow nat$
and *node-def-hash-size* :: *nat*
and *G-impl* **and** *G* :: $('v,'more)$ *graph-rec-scheme*

+
assumes *G-refine*: $(G\text{-}impl,G) \in \langle mrel,node\text{-}rel\rangle g\text{-}impl\text{-}rel\text{-}ext$
and *node-eq-refine*: $(node\text{-}eq\text{-}impl, (=)) \in node\text{-}rel \to node\text{-}rel \to bool\text{-}rel$
and *node-hash*: *is-bounded-hashcode node-rel node-eq-impl node-hash-impl*
and *node-hash-def-size*: $(is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE('vi)\ node\text{-}def\text{-}hash\text{-}size)$
begin

lemmas $[autoref\text{-}rel\text{-}intf] = REL\text{-}INTFI[of\ node\text{-}rel\ i\text{-}node]$

lemmas $[autoref\text{-}rules] = G\text{-}refine\ node\text{-}eq\text{-}refine$

lemmas $[autoref\text{-}ga\text{-}rules] = node\text{-}hash\ node\text{-}hash\text{-}def\text{-}size$

lemma *locale-this*: *fr-graph-impl-loc mrel node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
⟨*proof*⟩

abbreviation $oGSi\text{-}rel \equiv \langle node\text{-}rel,node\text{-}state\text{-}rel\rangle(ahm\text{-}rel\ node\text{-}hash\text{-}impl)$

abbreviation *GSi-rel* ≡
$\langle node\text{-}rel\rangle as\text{-}rel$
$\times_r\ \langle nat\text{-}rel\rangle as\text{-}rel$
$\times_r\ oGSi\text{-}rel$
$\times_r\ \langle nat\text{-}rel \times_r \langle node\text{-}rel\rangle list\text{-}set\text{-}rel\rangle as\text{-}rel$

lemmas $[autoref\text{-}op\text{-}pat] = GS.S\text{-}def\ GS.B\text{-}def\ GS.I\text{-}def\ GS.P\text{-}def$

end

1.26 Generating the Code

thm *autoref-ga-rules*

context *fr-graph-impl-loc*
begin
schematic-goal *push-code-aux*: (*?c*,*push-impl*)∈*node-rel* → *GSi-rel* → *GSi-rel*
⟨*proof*⟩
concrete-definition (**in** −) *push-code* **uses** *fr-graph-impl-loc.push-code-aux*
lemmas [*autoref-rules*] = *push-code.refine*[*OF locale-this*]

schematic-goal *pop-code-aux*: (*?c*,*pop-impl*)∈*GSi-rel* → ⟨*GSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *pop-code* **uses** *fr-graph-impl-loc.pop-code-aux*
lemmas [*autoref-rules*] = *pop-code.refine*[*OF locale-this*]

schematic-goal *S-idx-of-code-aux*:
notes [*autoref-rules*] = *IdI*[*of undefined*::*nat*]
shows (*?c*,*GS.S-idx-of*)∈*GSi-rel* → *node-rel* → *nat-rel*
⟨*proof*⟩
concrete-definition (**in** −) *S-idx-of-code*
uses *fr-graph-impl-loc.S-idx-of-code-aux*
lemmas [*autoref-rules*] = *S-idx-of-code.refine*[*OF locale-this*]

schematic-goal *idx-of-code-aux*:
notes [*autoref-rules*] = *IdI*[*of undefined*::*nat*]
shows (*?c*,*GS.idx-of-impl*)∈ *GSi-rel* → *node-rel* → ⟨*nat-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *idx-of-code* **uses** *fr-graph-impl-loc.idx-of-code-aux*
lemmas [*autoref-rules*] = *idx-of-code.refine*[*OF locale-this*]

schematic-goal *collapse-code-aux*:
(*?c*,*collapse-impl*)∈*node-rel* → *GSi-rel* → ⟨*GSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *collapse-code*
uses *fr-graph-impl-loc.collapse-code-aux*
lemmas [*autoref-rules*] = *collapse-code.refine*[*OF locale-this*]

term *select-edge-impl*
schematic-goal *select-edge-code-aux*:
(*?c*,*select-edge-impl*)
∈ *GSi-rel* → ⟨⟨*node-rel*⟩*option-rel* ×$_r$ *GSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *select-edge-code*
uses *fr-graph-impl-loc.select-edge-code-aux*
lemmas [*autoref-rules*] = *select-edge-code.refine*[*OF locale-this*]

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

term *fr-graph.pop-impl*
lemma [*autoref-op-pat*]:
push-impl ≡ *OP push-impl*
collapse-impl ≡ *OP collapse-impl*

select-edge-impl ≡ *OP select-edge-impl*
pop-impl ≡ *OP pop-impl*
⟨*proof*⟩

end

schematic-goal *skeleton-code-aux*:
(*?c*,*skeleton-impl*) ∈ ⟨*oGSi-rel*⟩*nres-rel*
⟨*proof*⟩

concrete-definition (**in** −) *skeleton-code*
for *node-eq-impl G-impl*
uses *fr-graph-impl-loc.skeleton-code-aux*

thm *skeleton-code.refine*

lemmas [*autoref-rules*] = *skeleton-code.refine*[*OF locale-this*]

schematic-goal *pop-tr-aux*: *RETURN ?c* ≤ *pop-code node-eq-impl node-hash-impl s*
⟨*proof*⟩
concrete-definition (**in** −) *pop-tr* **uses** *fr-graph-impl-loc.pop-tr-aux*
lemmas [*refine-transfer*] = *pop-tr.refine*[*OF locale-this*]

schematic-goal *select-edge-tr-aux*: *RETURN ?c* ≤ *select-edge-code node-eq-impl s*
⟨*proof*⟩
concrete-definition (**in** −) *select-edge-tr*
uses *fr-graph-impl-loc.select-edge-tr-aux*
lemmas [*refine-transfer*] = *select-edge-tr.refine*[*OF locale-this*]

schematic-goal *idx-of-tr-aux*: *RETURN ?c* ≤ *idx-of-code node-eq-impl node-hash-impl v s*
⟨*proof*⟩
concrete-definition (**in** −) *idx-of-tr* **uses** *fr-graph-impl-loc.idx-of-tr-aux*
lemmas [*refine-transfer*] = *idx-of-tr.refine*[*OF locale-this*]

schematic-goal *collapse-tr-aux*: *RETURN ?c* ≤ *collapse-code node-eq-impl node-hash-impl v s*
⟨*proof*⟩
concrete-definition (**in** −) *collapse-tr* **uses** *fr-graph-impl-loc.collapse-tr-aux*
lemmas [*refine-transfer*] = *collapse-tr.refine*[*OF locale-this*]

schematic-goal *skeleton-tr-aux*: *RETURN ?c* ≤ *skeleton-code node-hash-impl node-def-hash-size node-eq-impl g*
⟨*proof*⟩

concrete-definition (**in** −) *skeleton-tr* **uses** *fr-graph-impl-loc.skeleton-tr-aux*
lemmas [*refine-transfer*] = *skeleton-tr.refine*[*OF locale-this*]

end

term *skeleton-tr*

export-code *skeleton-tr* **checking** *SML*

end

1.27 Code Generation for SCC-Computation

theory *Gabow-SCC-Code*
imports
Gabow-SCC
Gabow-Skeleton-Code
CAVA-Base.CAVA-Code-Target
begin

1.28 Automatic Refinement to Efficient Data Structures

context *fr-graph-impl-loc*
begin
schematic-goal *last-seg-code-aux*:
(*?c*,*last-seg-impl*)∈*GSi-rel* → ⟨⟨*node-rel*⟩*list-set-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *last-seg-code*
uses *fr-graph-impl-loc.last-seg-code-aux*
lemmas [*autoref-rules*] = *last-seg-code.refine*[*OF locale-this*]

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

lemma [*autoref-op-pat*]:
last-seg-impl ≡ *OP last-seg-impl*
⟨*proof*⟩
end

schematic-goal *compute-SCC-code-aux*:
(*?c*,*compute-SCC-impl*) ∈ ⟨⟨⟨*node-rel*⟩*list-set-rel*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩

concrete-definition (**in** −) *compute-SCC-code*
uses *fr-graph-impl-loc.compute-SCC-code-aux*
lemmas [*autoref-rules*] = *compute-SCC-code.refine*[*OF locale-this*]

schematic-goal *last-seg-tr-aux*: *RETURN ?c* ≤ *last-seg-code s*
⟨*proof*⟩
concrete-definition (**in** −) *last-seg-tr* **uses** *fr-graph-impl-loc.last-seg-tr-aux*
lemmas [*refine-transfer*] = *last-seg-tr.refine*[*OF locale-this*]

schematic-goal *compute-SCC-tr-aux*: *RETURN ?c* ≤ *compute-SCC-code node-eq-impl node-hash-impl node-def-hash-size g*
⟨*proof*⟩
concrete-definition (**in** −) *compute-SCC-tr*
uses *fr-graph-impl-loc.compute-SCC-tr-aux*
lemmas [*refine-transfer*] = *compute-SCC-tr.refine*[*OF locale-this*]
end

export-code *compute-SCC-tr* **checking** *SML*

1.29 Correctness Theorem

theorem *compute-SCC-tr-correct*:
— Correctness theorem for the constant we extracted to SML
fixes *Re* **and** *node-rel* :: (′*vi* × ′*v*) *set*
fixes *G* :: (′*v*,′*more*) *graph-rec-scheme*
assumes *A*:
(*G-impl*,*G*)∈⟨*Re*,*node-rel*⟩*g-impl-rel-ext*
(*node-eq-impl*, (=)) ∈ *node-rel* → *node-rel* → *bool-rel*
is-bounded-hashcode node-rel node-eq-impl node-hash-impl
(*is-valid-def-hm-size TYPE*(′*vi*) *node-def-hash-size*)

assumes *C*: *fr-graph G*
shows *RETURN* (*compute-SCC-tr node-eq-impl node-hash-impl node-def-hash-size G-impl*)
≤ ⇓(⟨⟨*node-rel*⟩*list-set-rel*⟩*list-rel*) (*fr-graph.compute-SCC-spec G*)
⟨*proof*⟩

1.30 Extraction of Benchmark Code

schematic-goal *list-set-of-list-aux*:
(*?c*,*set*)∈⟨*nat-rel*⟩*list-rel* → ⟨*nat-rel*⟩*list-set-rel*
⟨*proof*⟩
concrete-definition *list-set-of-list* **uses** *list-set-of-list-aux*

term *compute-SCC-tr*

definition *compute-SCC-tr-nat* :: - ⇒ - ⇒ - ⇒ - ⇒ *nat list list*
where *compute-SCC-tr-nat* ≡ *compute-SCC-tr*

end

1.31 Implementation of Safety Property Model Checker

theory *Find-Path-Impl*
imports
Find-Path

CAVA-Automata.Digraph-Impl
begin

1.32 Workset Algorithm

A simple implementation is by a workset algorithm.

definition *ws-update E u p V ws* ≡ *RETURN* (
  V ∪ *E*``{*u*},
  ws ++ (λ*v*. *if* (*u*,*v*)∈*E* ∧ *v*∉*V then Some* (*u*#*p*) *else None*))

definition *s-init U0* ≡ *RETURN* (*None*,*U0*,λ*u*. *if u*∈*U0 then Some* [] *else None*)

definition *wset-find-path E U0 P* ≡ *do* {
  ASSERT (*finite U0*);
  s0 ← *s-init U0*;
  (*res*,-,-) ← *WHILET*
    (λ(*res*,*V*,*ws*). *res*=*None* ∧ *ws*≠*Map.empty*)
    (λ(*res*,*V*,*ws*). *do* {
      ASSERT (*ws*≠*Map.empty*);
      (*u*,*p*) ← *SPEC* (λ(*u*,*p*). *ws u* = *Some p*);
      let ws=*ws* |` (−{*u*});

      if P u then
        RETURN (*Some* (*rev p*,*u*),*V*,*ws*)
      else do {
        ASSERT (*finite* (*E*``{*u*}));
        ASSERT (*dom ws* ⊆ *V*);
        (*V*,*ws*) ← *ws-update E u p V ws*;
        RETURN (*None*,*V*,*ws*)
      }
    }) *s0*;
    RETURN res
  }

lemma *wset-find-path-correct*:
  fixes *E* :: ('*v*×'*v*) *set*
  shows *wset-find-path E U0 P* ≤ *find-path E U0 P*
⟨*proof*⟩

We refine the algorithm to use a foreach-loop

definition *ws-update-foreach E u p V ws* ≡
  FOREACH (*LIST-SET-REV-TAG* (*E*``{*u*})) (λ*v* (*V*,*ws*).
    if v∈*V then*
      RETURN (*V*,*ws*)
    else do {
      ASSERT (*v*∉*dom ws*);
      RETURN (*insert v V*,*ws*(*v* ↦ *u*#*p*))
    }
) (*V*,*ws*)

lemma *ws-update-foreach-refine*[*refine*]:
assumes *FIN*: *finite* (E``{u})
assumes *WSS*: *dom ws* ⊆ V
assumes *ID*: (E',E)∈*Id* (u',u)∈*Id* (p',p)∈*Id* (V',V)∈*Id* (ws',*ws*)∈*Id*
shows *ws-update-foreach* E' u' p' V' ws' ≤ ⇓*Id* (*ws-update* E u p V *ws*)
⟨*proof*⟩

definition *s-init-foreach U0* ≡ *do* {
($U0$,*ws*) ← *FOREACH U0* (λx ($U0$,*ws*).
RETURN (*insert x U0*,*ws*(x↦[]))) ({},*Map.empty*);
RETURN (*None*,$U0$,*ws*)
}

lemma *s-init-foreach-refine*[*refine*]:
assumes *FIN*: *finite U0*
assumes *ID*: ($U0'$,$U0$)∈*Id*
shows *s-init-foreach* $U0'$ ≤⇓*Id* (*s-init U0*)
⟨*proof*⟩

definition *wset-find-path'* E *U0* P ≡ *do* {
ASSERT (*finite U0*);
s0←*s-init-foreach U0*;
(*res*,-,-) ← *WHILET*
(λ(*res*,V,*ws*). *res*=*None* ∧ *ws*≠*Map.empty*)
(λ(*res*,V,*ws*). *do* {
ASSERT (*ws*≠*Map.empty*);
((*u*,*p*),*ws*) ← *op-map-pick-remove ws*;

if P *u then*
RETURN (*Some* (*rev p*,*u*),V,*ws*)
else do {
(V,*ws*) ← *ws-update-foreach* E *u* p V *ws*;
RETURN (*None*,V,*ws*)
}
})
s0;
RETURN res
}

lemma *wset-find-path'-refine*:
wset-find-path' E *U0* P ≤ ⇓*Id* (*wset-find-path* E *U0* P)
⟨*proof*⟩

1.33 Refinement to efficient data structures

schematic-goal *wset-find-path'-refine-aux*:
fixes *U0*::'*a set* **and** P::'*a* ⇒ *bool* **and** E::('*a*×'*a*) *set*
and *Pimpl* :: '*ai* ⇒ *bool*

and *node-rel* :: $('ai \times 'a)$ *set*
and *node-eq-impl* :: $'ai \Rightarrow 'ai \Rightarrow$ *bool*
and *node-hash-impl*
and *node-def-hash-size*

assumes [*autoref-rules*]:
$(succi,E)\in\langle node\text{-}rel\rangle slg\text{-}rel$
$(Pimpl,P)\in node\text{-}rel \to bool\text{-}rel$
$(node\text{-}eq\text{-}impl, (=)) \in node\text{-}rel \to node\text{-}rel \to bool\text{-}rel$
$(U0',U0)\in\langle node\text{-}rel\rangle list\text{-}set\text{-}rel$
assumes [*autoref-ga-rules*]:
is-bounded-hashcode node-rel node-eq-impl node-hash-impl
is-valid-def-hm-size $TYPE('ai)$ *node-def-hash-size*
notes [*autoref-tyrel*] =
$TYRELI$[**where**
$R=\langle node\text{-}rel,\langle node\text{-}rel\rangle list\text{-}rel\rangle list\text{-}map\text{-}rel$]
$TYRELI$[**where** $R=\langle node\text{-}rel\rangle map2set\text{-}rel$ (*ahm-rel node-hash-impl*)]

shows $(?c::?'c, wset\text{-}find\text{-}path'\ E\ U0\ P) \in ?R$
⟨*proof*⟩

concrete-definition *wset-find-path-impl* **for** *node-eq-impl succi U0' Pimpl*
uses *wset-find-path'-refine-aux*

1.34 Autoref Setup

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-itype*]:
$find\text{-}path ::_i \langle I\rangle_i i\text{-}slg \to_i \langle I\rangle_i i\text{-}set \to_i (I\to_i i\text{-}bool)$
$\to_i \langle\langle\langle I\rangle_i i\text{-}list, I\rangle_i i\text{-}prod\rangle_i i\text{-}option\rangle_i i\text{-}nres$ ⟨*proof*⟩

lemma *wset-find-path-autoref*[*autoref-rules*]:
fixes *node-rel* :: $('ai \times 'a)$ *set*
assumes *eq*: *GEN-OP node-eq-impl* $(=)$ $(node\text{-}rel\to node\text{-}rel\to bool\text{-}rel)$
assumes *hash*: *SIDE-GEN-ALGO* (*is-bounded-hashcode node-rel node-eq-impl node-hash-impl*)
assumes *hash-dsz*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE('ai)$ *node-def-hash-size*)
shows (
wset-find-path-impl node-hash-impl node-def-hash-size node-eq-impl,
find-path)
$\in \langle node\text{-}rel\rangle slg\text{-}rel \to \langle node\text{-}rel\rangle list\text{-}set\text{-}rel \to (node\text{-}rel\to bool\text{-}rel)$
$\to \langle\langle\langle node\text{-}rel\rangle list\text{-}rel\times_r node\text{-}rel\rangle option\text{-}rel\rangle nres\text{-}rel$
⟨*proof*⟩
end

schematic-goal *wset-find-path-transfer-aux*:
$RETURN\ ?c \le$ *wset-find-path-impl hashi dszi eqi E U0 P*

⟨*proof*⟩
concrete-definition *wset-find-path-code*
for *E ?U0.0 P* **uses** *wset-find-path-transfer-aux*
lemmas [*refine-transfer*] = *wset-find-path-code.refine*

export-code *wset-find-path-code* **checking** *SML*

1.35 Nontrivial paths

definition *find-path1-gen E u0 P* ≡ *do* {
res ← *find-path E* (*E*‘‘{*u0*}) *P*;
case res of None ⇒ *RETURN None*
| *Some* (*p*,*v*) ⇒ *RETURN* (*Some* (*u0*#*p*,*v*))
}

lemma *find-path1-gen-correct*: *find-path1-gen E u0 P* ≤ *find-path1 E u0 P*
⟨*proof*⟩

schematic-goal *find-path1-impl-aux*:
fixes *node-rel* :: (*'ai* × *'a*) *set*
assumes [*autoref-rules*]: (*node-eq-impl*, (=)) ∈ *node-rel* → *node-rel* → *bool-rel*
assumes [*autoref-ga-rules*]:
is-bounded-hashcode node-rel node-eq-impl node-hash-impl
is-valid-def-hm-size TYPE(*'ai*) *node-def-hash-size*

shows (*?c*,*find-path1-gen*::(-×-) *set* ⇒ -) ∈ ⟨*node-rel*⟩*slg-rel* → *node-rel* → (*node-rel* → *bool-rel*) → ⟨⟨⟨*node-rel*⟩*list-rel* ×$_r$ *node-rel*⟩*option-rel*⟩*nres-rel*
⟨*proof*⟩

lemma [*autoref-itype*]:
find-path1 ::$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ *I* →$_i$ (*I*→$_i$*i-bool*)
→$_i$ ⟨⟨⟨⟨*I*⟩$_i$*i-list*, *I*⟩$_i$*i-prod*⟩$_i$*i-option*⟩$_i$*i-nres* ⟨*proof*⟩

concrete-definition *find-path1-impl* **uses** *find-path1-impl-aux*

lemma *find-path1-autoref*[*autoref-rules*]:
fixes *node-rel* :: (*'ai* × *'a*) *set*
assumes *eq*: *GEN-OP node-eq-impl* (=) (*node-rel*→*node-rel*→*bool-rel*)
assumes *hash*: *SIDE-GEN-ALGO* (*is-bounded-hashcode node-rel node-eq-impl node-hash-impl*)
assumes *hash-dsz*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*(*'ai*) *node-def-hash-size*)

shows (*find-path1-impl node-eq-impl node-hash-impl node-def-hash-size*,*find-path1*)

∈ ⟨*node-rel*⟩*slg-rel* →*node-rel* → (*node-rel* → *bool-rel*) →
⟨⟨⟨*node-rel*⟩*list-rel* ×$_r$ *node-rel*⟩*Relators.option-rel*⟩*nres-rel*
⟨*proof*⟩

schematic-goal *find-path1-transfer-aux*:

RETURN ?c ≤ find-path1-impl eqi hashi dszi E u P
⟨*proof*⟩
concrete-definition *find-path1-code* **for** *E u P* **uses** *find-path1-transfer-aux*
lemmas [*refine-transfer*] = *find-path1-code.refine*

end

1.36 Code Generation for GBG Lasso Finding Algorithm

theory *Gabow-GBG-Code*
imports
Gabow-GBG
Gabow-Skeleton-Code
CAVA-Automata.Automata-Impl
Find-Path-Impl
CAVA-Base.CAVA-Code-Target
begin

1.37 Autoref Setup

locale *impl-lasso-loc* = *igb-fr-graph G*
+ *fr-graph-impl-loc* ⟨*mrel*,*node-rel*⟩*igbg-impl-rel-eext node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
for *mrel* **and** *node-rel* **and** *node-eq-impl node-hash-impl node-def-hash-size* **and** *G-impl* **and** *G* :: ('*q*,'*more*) *igb-graph-rec-scheme*
begin

lemma *locale-this*: *impl-lasso-loc mrel node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
⟨*proof*⟩

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

lemma [*autoref-op-pat*]:
goinitial-impl ≡ *OP goinitial-impl*
ginitial-impl ≡ *OP ginitial-impl*
gpath-is-empty-impl ≡ *OP gpath-is-empty-impl*
gselect-edge-impl ≡ *OP gselect-edge-impl*
gis-on-stack-impl ≡ *OP gis-on-stack-impl*
gcollapse-impl ≡ *OP gcollapse-impl*
last-is-acc-impl ≡ *OP last-is-acc-impl*
ce-impl ≡ *OP ce-impl*
gis-done-impl ≡ *OP gis-done-impl*
gpush-impl ≡ *OP gpush-impl*
gpop-impl ≡ *OP gpop-impl*
goBrk-impl ≡ *OP goBrk-impl*
gto-outer-impl ≡ *OP gto-outer-impl*
go-is-done-impl ≡ *OP go-is-done-impl*
is-done-oimpl ≡ *OP is-done-oimpl*
go-is-no-brk-impl ≡ *OP go-is-no-brk-impl*

⟨*proof*⟩
end

abbreviation *gGSi-rel* ≡ ⟨⟨*nat-rel*⟩*bs-set-rel*⟩*as-rel* ×$_r$ *GSi-rel*
abbreviation (**in** −) *ce-rel node-rel* ≡ ⟨⟨*node-rel*⟩*fun-set-rel* ×$_r$ ⟨*node-rel*⟩*fun-set-rel*⟩*option-rel*
abbreviation *goGSi-rel* ≡ *ce-rel node-rel* ×$_r$ *oGSi-rel*
end

1.38 Automatic Refinement

context *impl-lasso-loc*
begin

schematic-goal *goinitial-code-aux*: (*?c*,*goinitial-impl*)∈*goGSi-rel*
⟨*proof*⟩
concrete-definition (**in** −) *goinitial-code*
uses *impl-lasso-loc.goinitial-code-aux*
lemmas [*autoref-rules*] = *goinitial-code.refine*[*OF locale-this*]

term *ginitial-impl*
schematic-goal *ginitial-code-aux*:
(*?c*,*ginitial-impl*)∈*node-rel* → *goGSi-rel* → *gGSi-rel*
⟨*proof*⟩
concrete-definition (**in** −) *ginitial-code* **uses** *impl-lasso-loc.ginitial-code-aux*
lemmas [*autoref-rules*] = *ginitial-code.refine*[*OF locale-this*]

schematic-goal *gpath-is-empty-code-aux*:
(*?c*,*gpath-is-empty-impl*)∈*gGSi-rel*→*bool-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gpath-is-empty-code*
uses *impl-lasso-loc.gpath-is-empty-code-aux*
lemmas [*autoref-rules*] = *gpath-is-empty-code.refine*[*OF locale-this*]

term *goBrk*
schematic-goal *goBrk-code-aux*: (*?c*,*goBrk-impl*)∈*goGSi-rel*→*ce-rel node-rel*
⟨*proof*⟩
concrete-definition (**in** −) *goBrk-code* **uses** *impl-lasso-loc.goBrk-code-aux*
lemmas [*autoref-rules*] = *goBrk-code.refine*[*OF locale-this*]
thm *autoref-itype*(*1*)

term *gto-outer-impl*
schematic-goal *gto-outer-code-aux*:
(*?c*,*gto-outer-impl*)∈*ce-rel node-rel* → *gGSi-rel*→*goGSi-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gto-outer-code*
uses *impl-lasso-loc.gto-outer-code-aux*
lemmas [*autoref-rules*] = *gto-outer-code.refine*[*OF locale-this*]

term *go-is-done-impl*
schematic-goal *go-is-done-code-aux*:
(*?c*,*go-is-done-impl*)∈*node-rel* → *goGSi-rel* → *bool-rel*
⟨*proof*⟩
concrete-definition (**in** −) *go-is-done-code*
uses *impl-lasso-loc.go-is-done-code-aux*
lemmas [*autoref-rules*] = *go-is-done-code.refine*[*OF locale-this*]

schematic-goal *go-is-no-brk-code-aux*:
(*?c*,*go-is-no-brk-impl*)∈*goGSi-rel*→*bool-rel*
⟨*proof*⟩
concrete-definition (**in** −) *go-is-no-brk-code*
uses *impl-lasso-loc.go-is-no-brk-code-aux*
lemmas [*autoref-rules*] = *go-is-no-brk-code.refine*[*OF locale-this*]

schematic-goal *gselect-edge-code-aux*: (*?c*,*gselect-edge-impl*)
∈ *gGSi-rel* → ⟨⟨*node-rel*⟩*option-rel* ×$_r$ *gGSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gselect-edge-code*
uses *impl-lasso-loc.gselect-edge-code-aux*
lemmas [*autoref-rules*] = *gselect-edge-code.refine*[*OF locale-this*]

term *gis-on-stack-impl*
schematic-goal *gis-on-stack-code-aux*:
(*?c*,*gis-on-stack-impl*)∈*node-rel* → *gGSi-rel* → *bool-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gis-on-stack-code*
uses *impl-lasso-loc.gis-on-stack-code-aux*
lemmas [*autoref-rules*] = *gis-on-stack-code.refine*[*OF locale-this*]

term *gcollapse-impl*
schematic-goal *gcollapse-code-aux*: (*?c*,*gcollapse-impl*)∈*node-rel* → *gGSi-rel*
→ ⟨*gGSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gcollapse-code*
uses *impl-lasso-loc.gcollapse-code-aux*
lemmas [*autoref-rules*] = *gcollapse-code.refine*[*OF locale-this*]

schematic-goal *last-is-acc-code-aux*:
(*?c*,*last-is-acc-impl*)∈*gGSi-rel*→⟨*bool-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *last-is-acc-code*
uses *impl-lasso-loc.last-is-acc-code-aux*
lemmas [*autoref-rules*] = *last-is-acc-code.refine*[*OF locale-this*]

schematic-goal *ce-code-aux*: (*?c*,*ce-impl*)
∈*gGSi-rel*→⟨*ce-rel node-rel* ×$_r$ *gGSi-rel*⟩*nres-rel*

⟨*proof*⟩
concrete-definition (**in** −) *ce-code* **uses** *impl-lasso-loc.ce-code-aux*
lemmas [*autoref-rules*] = *ce-code.refine*[*OF locale-this*]

schematic-goal *gis-done-code-aux*:
(*?c*,*gis-done-impl*)∈*node-rel*→*gGSi-rel*→*bool-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gis-done-code* **uses** *impl-lasso-loc.gis-done-code-aux*
lemmas [*autoref-rules*] = *gis-done-code.refine*[*OF locale-this*]

schematic-goal *gpush-code-aux*:
(*?c*,*gpush-impl*)∈*node-rel* → *gGSi-rel*→*gGSi-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gpush-code* **uses** *impl-lasso-loc.gpush-code-aux*
lemmas [*autoref-rules*] = *gpush-code.refine*[*OF locale-this*]

schematic-goal *gpop-code-aux*: (*?c*,*gpop-impl*)∈*gGSi-rel*→⟨*gGSi-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *gpop-code* **uses** *impl-lasso-loc.gpop-code-aux*
lemmas [*autoref-rules*] = *gpop-code.refine*[*OF locale-this*]

schematic-goal *find-ce-code-aux*: (*?c*,*find-ce-impl*)∈⟨*ce-rel node-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *find-ce-code*
uses *impl-lasso-loc.find-ce-code-aux*
lemmas [*autoref-rules*] = *find-ce-code.refine*[*OF locale-this*]

schematic-goal *find-ce-tr-aux*: *RETURN ?c* ≤ *find-ce-code node-eq-impl node-hash-impl node-def-hash-size G-impl*
⟨*proof*⟩
concrete-definition (**in** −) *find-ce-tr* **for** *G-impl*
uses *impl-lasso-loc.find-ce-tr-aux*
lemmas [*refine-transfer*] = *find-ce-tr.refine*[*OF locale-this*]

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-op-pat*]:
find-ce-spec ≡ *OP find-ce-spec*
⟨*proof*⟩
end

theorem *find-ce-autoref*[*autoref-rules*]:
— Main Correctness theorem (inside locale)
shows (*find-ce-code node-eq-impl node-hash-impl node-def-hash-size G-impl*, *find-ce-spec*) ∈ ⟨*ce-rel node-rel*⟩*nres-rel*
⟨*proof*⟩

end

context *impl-lasso-loc*
begin

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

lemma [*autoref-op-pat*]:
reconstruct-reach ≡ *OP reconstruct-reach*
reconstruct-lasso ≡ *OP reconstruct-lasso*
⟨*proof*⟩
end

schematic-goal *reconstruct-reach-code-aux*:
shows (*?c*, *reconstruct-reach*)∈⟨*node-rel*⟩*fun-set-rel* →
⟨*node-rel*⟩*fun-set-rel* →
⟨⟨*node-rel*⟩*list-rel* ×$_r$ *node-rel*⟩*nres-rel*
⟨*proof*⟩

concrete-definition (**in** −) *reconstruct-reach-code*
uses *impl-lasso-loc.reconstruct-reach-code-aux*
lemmas [*autoref-rules*] = *reconstruct-reach-code.refine*[*OF locale-this*]

schematic-goal *reconstruct-lasso-code-aux*:
shows (*?c*, *reconstruct-lasso*)∈⟨*node-rel*⟩*fun-set-rel* →
⟨*node-rel*⟩*fun-set-rel* →
⟨⟨*node-rel*⟩*list-rel* ×$_r$ ⟨*node-rel*⟩*list-rel*⟩*nres-rel*
⟨*proof*⟩
concrete-definition (**in** −) *reconstruct-lasso-code*
uses *impl-lasso-loc.reconstruct-lasso-code-aux*
lemmas [*autoref-rules*] = *reconstruct-lasso-code.refine*[*OF locale-this*]

schematic-goal *reconstruct-lasso-tr-aux*:
RETURN ?c ≤ *reconstruct-lasso-code eqi hi dszi G-impl Vr Vl*
⟨*proof*⟩
concrete-definition (**in** −) *reconstruct-lasso-tr* **for** *G-impl*
uses *impl-lasso-loc.reconstruct-lasso-tr-aux*
lemmas [*refine-transfer*] = *reconstruct-lasso-tr.refine*[*OF locale-this*]

schematic-goal *find-lasso-code-aux*:
shows (*?c*::*?'c*, *find-lasso*)∈*?R*
⟨*proof*⟩
concrete-definition (**in** −) *find-lasso-code*
uses *impl-lasso-loc.find-lasso-code-aux*
lemmas [*autoref-rules*] = *find-lasso-code.refine*[*OF locale-this*]

schematic-goal *find-lasso-tr-aux*:
RETURN ?c ≤ *find-lasso-code node-eq-impl node-hash-impl node-def-hash-size*

G-impl
  ⟨*proof*⟩
 concrete-definition (**in** −) *find-lasso-tr* **for** *G-impl*
  uses *impl-lasso-loc.find-lasso-tr-aux*
 lemmas [*refine-transfer*] = *find-lasso-tr.refine*[*OF locale-this*]

end

export-code *find-lasso-tr* **checking** *SML*

1.39 Main Correctness Theorem

abbreviation *fl-rel* :: - ⇒ (- × (*'a list* × *'a list*) *option*) *set* **where**
 fl-rel node-rel ≡ ⟨⟨*node-rel*⟩*list-rel* ×$_r$ ⟨*node-rel*⟩*list-rel*⟩*Relators.option-rel*

theorem *find-lasso-tr-correct*:
 — Correctness theorem for the constant we extracted to SML
 fixes *Re* **and** *node-rel* :: (*'vi* × *'v*) *set*
 assumes *A*: (*G-impl*,*G*)∈*igbg-impl-rel-ext Re node-rel*
  and *node-eq-refine*: (*node-eq-impl*, (=)) ∈ *node-rel* → *node-rel* → *bool-rel*
  and *node-hash*: *is-bounded-hashcode node-rel node-eq-impl node-hash-impl*
  and *node-hash-def-size*: (*is-valid-def-hm-size TYPE*(*'vi*) *node-def-hash-size*)

 assumes *B*: *igb-fr-graph G*
 shows *RETURN* (*find-lasso-tr node-eq-impl node-hash-impl node-def-hash-size G-impl*)
 ≤ ⇓(*fl-rel node-rel*) (*igb-graph.find-lasso-spec G*)
⟨*proof*⟩

1.40 Autoref Setup for *igb-graph.find-lasso-spec*

Setup for Autoref, such that *igb-graph.find-lasso-spec* can be used

definition [*simp*]: *op-find-lasso-spec* ≡ *igb-graph.find-lasso-spec*

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

lemma [*autoref-op-pat*]: *igb-graph.find-lasso-spec* ≡ *op-find-lasso-spec*
 ⟨*proof*⟩

term *op-find-lasso-spec*

lemma [*autoref-itype*]:
 op-find-lasso-spec
  ::$_i$ *i-igbg Ie I* →$_i$ ⟨⟨⟨⟨*I*⟩$_i$*i-list*, ⟨*I*⟩$_i$*i-list*⟩$_i$*i-prod*⟩$_i$*i-option*⟩$_i$*i-nres*
 ⟨*proof*⟩

lemma *find-lasso-spec-autoref*[*autoref-rules-raw*]:
 fixes *Re* **and** *node-rel* :: (*'vi* × *'v*) *set*
 assumes *GR*: *SIDE-PRECOND* (*igb-fr-graph G*)

assumes *eq*: *GEN-OP node-eq-impl* (=) (*node-rel*→*node-rel*→*bool-rel*)
assumes *hash*: *SIDE-GEN-ALGO* (*is-bounded-hashcode node-rel node-eq-impl node-hash-impl*)
assumes *hash-dsz*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*('*vi*) *node-def-hash-size*)
assumes *Gi*: (*G-impl*,*G*)∈*igbg-impl-rel-ext Re node-rel*
shows (*RETURN* (*find-lasso-tr node-eq-impl node-hash-impl node-def-hash-size G-impl*),
(*OP op-find-lasso-spec*
::: *igbg-impl-rel-ext Re node-rel* → ⟨*fl-rel node-rel*⟩*nres-rel*)$*G*) ∈ ⟨*fl-rel node-rel*⟩*nres-rel*
⟨*proof*⟩

end

end

2 Conclusion

We have presented a verification of two variants of Gabow's algorithm: Computation of the strongly connected components of a graph, and emptiness check of a generalized Büchi automaton. We have extracted efficient code with a performance comparable to a reference implementation in Java.

We have modularized the formalization in two directions: First, we share most of the proofs between the two variants of the algorithm. Second, we use a stepwise refinement approach to separate the algorithmic ideas and the correctness proof from implementation details. Sharing of the proofs reduced the overall effort of developing both algorithms. Using a stepwise refinement approach allowed us to formalize an efficient implementation, without making the correctness proof complex and unmanageable by cluttering it with implementation details.

Our development approach is independent of Gabow's algorithm, and can be re-used for the verification of other algorithms.

Current and Future Work An important direction of future work is to fine-tune the implementation of the emptiness check algorithm for speed, as speed of the checking algorithm directely influences the performance of the modelchecker.

References

[1] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[2] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized büchi automata. In *Proc. of SPIN*, pages 169–184. Springer, 2005.

[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. Ch. 25.

[4] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

[5] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, 2000.

[6] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theor. Comput. Sci.*, 345(1):60–82, Nov. 2005.

[7] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

[8] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.

[9] P. Lammich. Refinement for monadic programs. In *Archive of Formal Proofs*. http://isa-afp.org/entries/Refine_Monadic.shtml, 2012. Formal proof development.

[10] P. Lammich. Automatic data refinement. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 84–99. Springer Berlin Heidelberg, 2013.

[11] P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *Proc. of ITP*, 2014. to appear.

[12] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *Proc. of ITP*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.

[13] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *Proc. of ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[15] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56 – 58, 1971.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[17] J. Purdom, Paul. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[18] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *LNCS*, pages 668–682. Springer, 2013.

[19] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 2011. 4th edition.

[20] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, Jan. 1981.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[22] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.