

Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm

Peter Lammich

May 26, 2024

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	4
1.1	Skeleton for Gabow's SCC Algorithm	5
1.2	Statistics Setup	5
1.3	Abstract Algorithm	6
1.3.1	Preliminaries	6
1.3.2	Invariants	7
1.3.3	Abstract Skeleton Algorithm	9
1.3.4	Invariant Preservation	12
1.3.5	Consequences of Invariant when Finished	18
1.4	Refinement to Gabow's Data Structure	18
1.4.1	Preliminaries	18
1.4.2	Gabow's Datastructure	19
1.4.3	Refinement of the Operations	22
1.4.4	Refined Skeleton Algorithm	27
1.5	Enumerating the SCCs of a Graph	29
1.6	Specification	29
1.7	Extended Invariant	29
1.8	Definition of the SCC-Algorithm	30
1.9	Preservation of Invariant Extension	31
1.10	Main Correctness Proof	33
1.11	Refinement to Gabow's Data Structure	33
1.12	Safety-Property Model-Checker	36
1.13	Finding Path to Error	36
1.13.1	Nontrivial Paths	36
1.14	Lasso Finding Algorithm for Generalized Büchi Graphs	37
1.15	Specification	37
1.16	Invariant Extension	38
1.17	Definition of the Lasso-Finding Algorithm	38
1.18	Invariant Preservation	39
1.19	Main Correctness Proof	43
1.20	Emptiness Check	43
1.21	Refinement	44
1.21.1	Addition of Explicit Accepting Sets	44
1.21.2	Refinement to Gabow's Data Structure	47
1.22	Constructing a Lasso from Counterexample	54
1.22.1	Lassos in GBAs	54
1.23	Code Generation for the Skeleton Algorithm	56
1.24	Statistics	56
1.25	Automatic Refinement Setup	57
1.26	Generating the Code	58
1.27	Code Generation for SCC-Computation	61
1.28	Automatic Refinement to Efficient Data Structures	61

1.29	Correctness Theorem	62
1.30	Extraction of Benchmark Code	62
1.31	Implementation of Safety Property Model Checker	62
1.32	Workset Algorithm	63
1.33	Refinement to efficient data structures	64
1.34	Autoref Setup	65
1.35	Nontrivial paths	66
1.36	Code Generation for GBG Lasso Finding Algorithm	67
1.37	Autoref Setup	67
1.38	Automatic Refinement	68
1.39	Main Correctness Theorem	72
1.40	Autoref Setup for <i>igb-graph.find-lasso-spec</i>	72
2	Conclusion	73

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-

¹Although 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 graph² (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⟩
```

²GBGs are generalized Büchi automata without labels.

abbreviation $stat\text{-}newnode\text{-}nres \equiv RETURN (stat\text{-}newnode ())$

abbreviation $stat\text{-}start\text{-}nres \equiv RETURN (stat\text{-}start ())$

abbreviation $stat\text{-}stop\text{-}nres \equiv RETURN (stat\text{-}stop ())$

lemma $discard\text{-}stat\text{-}refine[refine]$:

$m1 \leq m2 \implies stat\text{-}newnode\text{-}nres \gg m1 \leq m2$

$m1 \leq m2 \implies stat\text{-}start\text{-}nres \gg m1 \leq m2$

$m1 \leq m2 \implies stat\text{-}stop\text{-}nres \gg m1 \leq m2$

$\langle proof \rangle$

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 $path\text{-}seg :: 'a\ set\ list \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ set$

— Set of nodes in a segment of the path

where $path\text{-}seg\ p\ i\ j \equiv \bigcup \{p!k \mid k. i \leq k \wedge k < j\}$

lemma $path\text{-}seg\text{-}simps[simp]$:

$j \leq i \implies path\text{-}seg\ p\ i\ j = \{\}$

$path\text{-}seg\ p\ i\ (Suc\ i) = p!i$

$\langle proof \rangle$

lemma $path\text{-}seg\text{-}drop$:

$\bigcup (set (drop\ i\ p)) = path\text{-}seg\ p\ i\ (length\ p)$

$\langle proof \rangle$

lemma $path\text{-}seg\text{-}butlast$:

$p \neq [] \implies path\text{-}seg\ p\ 0\ (length\ p - Suc\ 0) = \bigcup (set (butlast\ p))$

$\langle proof \rangle$

definition $idx\text{-}of :: 'a\ set\ list \Rightarrow 'a \Rightarrow nat$

— Index of path segment that contains a node

where $idx\text{-}of\ p\ v \equiv THE\ i. i < length\ p \wedge v \in p!i$

lemma $idx\text{-}of\text{-}props$:

assumes

$p\text{-}disjoint\text{-}sym: \forall i\ j\ v. i < length\ p \wedge j < length\ p \wedge v \in p!i \wedge v \in p!j \longrightarrow i = j$

assumes $ON\text{-}STACK: v \in \bigcup (set\ p)$

shows

$idx\text{-}of\ p\ v < length\ p$ **and**

$v \in p ! idx\text{-}of\ p\ v$

$\langle proof \rangle$

lemma $idx\text{-}of\text{-}uniq$:

assumes

p-disjoint-sym: $\forall i j v. i < \text{length } p \wedge j < \text{length } p \wedge v \in p!i \wedge v \in p!j \longrightarrow i=j$

assumes *A*: $i < \text{length } p \quad v \in p!i$

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* \times *'v set* \times (*'v* \times *'v*) *set*

context *fr-graph*

begin

definition *touched* :: *'v set list* \Rightarrow *'v set* \Rightarrow *'v set*

— Touched: Nodes that are done or on path

where *touched p D* $\equiv D \cup \bigcup(\text{set } p)$

definition *vE* :: *'v set list* \Rightarrow *'v set* \Rightarrow (*'v* \times *'v*) *set* \Rightarrow (*'v* \times *'v*) *set*

— Visited edges: No longer pending edges from touched nodes

where *vE p D pE* $\equiv (E \cap (\text{touched } p D \times \text{UNIV})) - pE$

lemma *vE-ss-E*: *vE p D pE* $\subseteq E$ — 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* $\subseteq V0$ — Only start nodes to iterate over

assumes *it-done*: *V0 - it* $\subseteq D$ — Nodes already iterated over are visited

assumes *D-reachable*: *D* $\subseteq E^* V0$ — Done nodes are reachable

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

begin

lemma *locale-this*: *outer-invar-loc G it D* *<proof>*

definition (**in** *fr-graph*) *outer-invar* $\equiv \lambda it D. \text{outer-invar-loc } G \text{ it } D$

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

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

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

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

assumes vE -*no-back*: $\llbracket i < j; j < \text{length } p \rrbracket \implies vE \ p \ D \ pE \cap p!j \times p!i = \{\}$
— Visited edges do not go back on path
assumes p -*not-D*: $\bigcup(\text{set } p) \cap D = \{\}$ — Path does not contain done nodes

begin

abbreviation $ltouched$ **where** $ltouched \equiv touched \ p \ D$
abbreviation lwE **where** $lwE \equiv vE \ p \ D \ pE$

lemma $locale$ -*this*: *invar-loc* $G \ v0 \ D0 \ p \ D \ pE$ $\langle proof \rangle$

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

lemma $invar$ -*this*[*simp*, *intro!*]: $invar \ v0 \ D0 \ (p, D, pE)$
 $\langle proof \rangle$

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

lemma D -*vis*: $E \cap D \times UNIV \subseteq lwE$ — 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*: $p \neq [] \implies pE \cap last\ p \times UNIV = (E - lvE) \cap last\ p \times UNIV$
— 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* $v0 \equiv inv\ image\ (finite\ psubset\ < *lex* >\ measure\ length)$
 $(\lambda(p,D,pE). (unproc\ edges\ v0\ p\ D\ pE, p))$

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* $v (p, D, pE)$
 $\equiv \text{let } i = \text{idx-of } p \text{ } v \text{ in } (\text{take } i \text{ } p @ [\bigcup (\text{set } (\text{drop } i \text{ } p))], D, pE)$
 ⟨proof⟩

lemma *push* $v (p, D, pE) \equiv (p @ [\{v\}], D, pE \cup E \cap \{v\} \times UNIV)$
 ⟨proof⟩

lemma *pop* $(p, D, pE) \equiv (\text{butlast } p, \text{last } p \cup D, pE)$
 ⟨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 $\equiv \text{do } \{$
 $\text{let } D = \{\};$
 $r \leftarrow \text{FOREACHi outer-invar } V0 (\lambda v0 \text{ } D0. \text{do } \{$
 $\text{if } v0 \notin D0 \text{ then do } \{$
 $\text{let } s = \text{initial } v0 \text{ } D0;$

$(p, D, pE) \leftarrow \text{WHILEIT } (\text{invar } v0 \text{ } D0)$
 $(\lambda(p, D, pE). p \neq []) (\lambda(p, D, pE).$
 $\text{do } \{$
 — Select edge from end of path
 $(vo, (p, D, pE)) \leftarrow \text{select-edge } (p, D, pE);$

$\text{ASSERT } (p \neq []);$

$\text{case } vo \text{ of}$

$\text{Some } v \Rightarrow \text{do } \{$ — Found outgoing edge to node v
 $\text{if } v \in \bigcup (\text{set } p) \text{ then do } \{$
 — Back edge: Collapse path
 $\text{RETURN } (\text{collapse } v (p, D, pE))$
 $\} \text{ else if } v \notin D \text{ then do } \{$
 — Edge to new node. Append to path
 $\text{RETURN } (\text{push } v (p, D, pE))$
 $\} \text{ else do } \{$
 — Edge to done node. Skip
 $\text{RETURN } (p, D, pE)$
 $\}$

$\} \text{ None } \Rightarrow \text{do } \{$
 $\text{ASSERT } (pE \cap \text{last } p \times UNIV = \{\});$

```

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

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

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

lemma *touched-push[simp]*: $\text{touched } (p \ @ \ [V]) \ D = \text{touched } p \ D \cup V$
 ⟨*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*:
 $\llbracket i < \text{length } p; u \in p!i; v \in p!i \rrbracket \implies (u,v) \in (lvE \cap p!i \times p!i)^*$
 ⟨*proof*⟩

lemma *cnode-connectedI'*: $\llbracket i < \text{length } p; u \in p!i; v \in p!i \rrbracket \implies (u,v) \in (lvE)^*$
 ⟨*proof*⟩

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

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

lemma *p-disjoint-sym*: $\llbracket i < \text{length } p; j < \text{length } p; v \in p!i; v \in p!j \rrbracket \implies i=j$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *touched-reachable*: $ltouched \subseteq E^* \{v0\}$ — Touched nodes are reachable
 $\langle \text{proof} \rangle$

lemma *vE-reachable*: $lvE \subseteq E^* \{v0\} \times E^* \{v0\}$
 $\langle \text{proof} \rangle$

lemma *pE-reachable*: $pE \subseteq E^* \{v0\} \times E^* \{v0\}$
 $\langle \text{proof} \rangle$

lemma *D-closed-vE-rtrancl*: $lvE^* \{D\} \subseteq D$
 $\langle \text{proof} \rangle$

lemma *D-closed-path*: $\llbracket \text{path } E \ u \ q \ w; u \in D \rrbracket \implies \text{set } q \subseteq D$
 $\langle \text{proof} \rangle$

lemma *D-closed-path-vE*: $\llbracket \text{path } lvE \ u \ q \ w; u \in D \rrbracket \implies \text{set } q \subseteq D$
 $\langle \text{proof} \rangle$

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

$\langle proof \rangle$

lemma *loop-in-lastnode*:

assumes P : $path\ lvE\ u\ q\ u$

assumes $[simp]$: $p \neq []$

assumes ND : $set\ q \cap last\ p \neq \{\}$

shows $u \in last\ p$ **and** $set\ q \subseteq last\ p$

— A loop that touches the last node is completely inside the last node

$\langle proof \rangle$

lemma *no-D-p-edges*: $E \cap D \times \bigcup (set\ p) = \{\}$

$\langle proof \rangle$

lemma *idx-of-props*:

assumes $ON-STACK$: $v \in \bigcup (set\ p)$

shows

$idx\ of\ p\ v < length\ p$ **and**

$v \in p \ \& \ idx\ of\ p\ v$

$\langle proof \rangle$

end

Auxiliary Lemmas Regarding the Operations **lemma** (in *fr-graph*)

vE-initial $[simp]$: $vE\ [\{v0\}]\ \{\} (E \cap \{v0\} \times UNIV) = \{\}$

$\langle proof \rangle$

context *invar-loc*

begin

lemma *vE-push*: $\llbracket (u,v) \in pE; u \in last\ p; v \notin \bigcup (set\ p); v \notin D \rrbracket$

$\implies vE\ (p\ @\ [\{v\}])\ D\ ((pE - \{(u,v)\}) \cup E \cap \{v\} \times UNIV) = insert\ (u,v)\ lvE$

$\langle proof \rangle$

lemma *vE-remove* $[simp]$:

$\llbracket p \neq []; (u,v) \in pE \rrbracket \implies vE\ p\ D\ (pE - \{(u,v)\}) = insert\ (u,v)\ lvE$

$\langle proof \rangle$

lemma *vE-pop* $[simp]$: $p \neq [] \implies vE\ (butlast\ p)\ (last\ p \cup D)\ pE = lvE$

$\langle proof \rangle$

lemma *pE-fin*: $p = [] \implies pE = \{\}$

$\langle proof \rangle$

lemma (in *invar-loc*) *lastp-un-D-closed*:

assumes NE : $p \neq []$

assumes NO' : $pE \cap (last\ p \times UNIV) = \{\}$

shows $E''(last\ p \cup D) \subseteq (last\ p \cup D)$

— On pop, the popped CNode and D are closed under transitions

$\langle proof \rangle$

end

Preservation of Invariant by Operations context *fr-graph*

begin

lemma (in *outer-invar-loc*) *invar-initial-aux*:

assumes $v0 \in it - D$

shows *invar* $v0 D$ (*initial* $v0 D$)

$\langle proof \rangle$

lemma *invar-initial*:

$\llbracket \text{outer-invar } it D0; v0 \in it; v0 \notin D0 \rrbracket \implies \text{invar } v0 D0$ (*initial* $v0 D0$)

$\langle proof \rangle$

lemma *outer-invar-initial*[*simp, intro!*]: *outer-invar* $V0 \{\}$

$\langle proof \rangle$

lemma *invar-pop*:

assumes *INV*: *invar* $v0 D0$ (p, D, pE)

assumes *NE*[*simp*]: $p \neq []$

assumes *NO'*: $pE \cap (\text{last } p \times UNIV) = \{\}$

shows *invar* $v0 D0$ (*pop* (p, D, pE))

$\langle proof \rangle$

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

lemma *invar-collapse*:

assumes *INV*: *invar* $v0 D0$ (p, D, pE)

assumes *NE*[*simp*]: $p \neq []$

assumes *E*: $(u, v) \in pE$ **and** $u \in \text{last } p$

assumes *BACK*: $v \in \bigcup (\text{set } p)$

defines $i \equiv \text{idx-of } p v$

defines $p' \equiv \text{collapse-aux } p i$

shows *invar* $v0 D0$ (*collapse* v ($p, D, pE - \{(u, v)\}$))

$\langle proof \rangle$

lemma *invar-push*:

assumes *INV*: *invar* $v0 D0$ (p, D, pE)

assumes *NE*[*simp*]: $p \neq []$

assumes *E*: $(u, v) \in pE$ **and** *UIL*: $u \in \text{last } p$

assumes *VNE*: $v \notin \bigcup (\text{set } p)$ $v \notin D$

shows *invar* $v0 D0$ (*push* v ($p, D, pE - \{(u, v)\}$))

$\langle proof \rangle$

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 $invar\ v0\ D0\ (p,D,pE - \{(u,v)\})$
 $\langle proof \rangle$

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 $D \supseteq E^* \{v0\}$

$\langle proof \rangle$

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 $path\ (vE\ []\ D\ pE)\ u\ q\ v \longleftrightarrow path\ E\ u\ q\ v$

$\langle proof \rangle$

lemma *invar-outer-newnode*:

assumes $A: v0 \notin D0\ \ v0 \in it$
assumes $OINV: outer-invar\ it\ D0$
assumes $INV: invar\ v0\ D0\ ([], D', pE)$
shows $outer-invar\ (it - \{v0\})\ D'$

$\langle proof \rangle$

lemma *invar-outer-Dnode*:

assumes $A: v0 \in D0\ \ v0 \in it$
assumes $OINV: outer-invar\ it\ D0$
shows $outer-invar\ (it - \{v0\})\ D0$

$\langle proof \rangle$

lemma *pE-fin'*: $invar\ x\ \sigma\ ([], D, pE) \implies pE = \{\}$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *unproc-decreasing*:

— As effect of selecting a pending edge, the set of unprocessed edges decreases

assumes [*simp*]: $p \neq []$ **and** $A: (u,v) \in pE\ \ u \in last\ p$
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 (λD. outer-invar {} D)*
<proof>

Short proof, as presented in the paper

```

context
  notes [refine] = refine-vcg
begin
  theorem skeleton ≤ SPEC ( $\lambda D.$  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  $D = E^* \text{“} V0$ 
    ⟨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 $P :: (\text{nat} \times 'v \text{ set}) \text{ list}$ 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 ::  $\text{nat} \Rightarrow (\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat}$ 
  — 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:
   $\llbracket P\ 0; 0 < u \rrbracket \Longrightarrow \text{find-max-nat } u\ P = \text{Max } \{i. i < u \wedge P\ i\}$ 
  ⟨proof⟩

```

```

lemma find-max-nat-param[param]:
  assumes  $(n, n') \in \text{nat-rel}$ 
  assumes  $\bigwedge j\ j'. \llbracket (j, j') \in \text{nat-rel}; j' < n' \rrbracket \Longrightarrow (P\ j, P'\ j') \in \text{bool-rel}$ 
  shows  $(\text{find-max-nat } n\ P, \text{find-max-nat } n'\ P') \in \text{nat-rel}$ 

```

$\langle proof \rangle$

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

lemma *find-max-nat-autoref*[*autoref-rules*]:

assumes $(n, n') \in \text{nat-rel}$

assumes $\bigwedge j j'. \llbracket (j, j') \in \text{nat-rel}; j' < n \rrbracket \implies (P\ j, P'\ \$j') \in \text{bool-rel}$

shows $(\text{find-max-nat}\ n\ P,$

$(OP\ \text{find-max-nat}\ ::\ \text{nat-rel} \rightarrow (\text{nat-rel} \rightarrow \text{bool-rel}) \rightarrow \text{nat-rel})\ \$n'\ \$P'$

$) \in \text{nat-rel}$

$\langle proof \rangle$

end

1.4.2 Gabow's Datastructure

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

type-synonym $'v\ oGS = 'v \rightarrow \text{node-state}$

definition $oGS\text{-}\alpha :: 'v\ oGS \Rightarrow 'v\ \text{set}$ **where** $oGS\text{-}\alpha\ I \equiv \{v. I\ v = \text{Some}\ \text{DONE}\}$

locale *oGS-invar* =

fixes $I :: 'v\ oGS$

assumes *I-no-stack*: $I\ v \neq \text{Some}\ (\text{STACK}\ j)$

type-synonym $'a\ GS$

$= 'a\ \text{list} \times \text{nat}\ \text{list} \times ('a \rightarrow \text{node-state}) \times (\text{nat} \times 'a\ \text{set})\ \text{list}$

locale *GS* =

fixes *SBIP* :: $'a\ GS$

begin

definition $S \equiv (\lambda(S, B, I, P). S)\ \text{SBIP}$

definition $B \equiv (\lambda(S, B, I, P). B)\ \text{SBIP}$

definition $I \equiv (\lambda(S, B, I, P). I)\ \text{SBIP}$

definition $P \equiv (\lambda(S, B, I, P). P)\ \text{SBIP}$

definition *seg-start* :: $\text{nat} \Rightarrow \text{nat}$ — Start index of segment, inclusive

where $\text{seg-start}\ i \equiv B!i$

definition *seg-end* :: $\text{nat} \Rightarrow \text{nat}$ — End index of segment, exclusive

where $\text{seg-end}\ i \equiv \text{if}\ i+1 = \text{length}\ B\ \text{then}\ \text{length}\ S\ \text{else}\ B!(i+1)$

definition *seg* :: $\text{nat} \Rightarrow 'a\ \text{set}$ — Collapsed set at index

where $\text{seg}\ i \equiv \{S!j \mid j. \text{seg-start}\ i \leq j \wedge j < \text{seg-end}\ i\}$

definition $p\text{-}\alpha \equiv \text{map}\ \text{seg}\ [0..<\text{length}\ B]$ — Collapsed path

definition $D\text{-}\alpha \equiv \{v. I\ v = \text{Some}\ \text{DONE}\}$ — Done nodes

definition $pE\text{-}\alpha \equiv \{ (u,v) . \exists j I. (j,I) \in \text{set } P \wedge u = S!j \wedge v \in I \}$
 — Pending edges

definition $\alpha \equiv (p\text{-}\alpha, D\text{-}\alpha, pE\text{-}\alpha)$ — Abstract state

end

lemma $GS\text{-}sel\text{-}simps[simp]$:

$GS.S (S,B,I,P) = S$

$GS.B (S,B,I,P) = B$

$GS.I (S,B,I,P) = I$

$GS.P (S,B,I,P) = P$

$\langle proof \rangle$

context GS **begin**

lemma $seg\text{-}start\text{-}indep[simp]$: $GS.seg\text{-}start (S',B',I',P') = seg\text{-}start$

$\langle proof \rangle$

lemma $seg\text{-}end\text{-}indep[simp]$: $GS.seg\text{-}end (S,B,I',P') = seg\text{-}end$

$\langle proof \rangle$

lemma $seg\text{-}indep[simp]$: $GS.seg (S,B,I',P') = seg$

$\langle proof \rangle$

lemma $p\text{-}\alpha\text{-}indep[simp]$: $GS.p\text{-}\alpha (S,B,I',P') = p\text{-}\alpha$

$\langle proof \rangle$

lemma $D\text{-}\alpha\text{-}indep[simp]$: $GS.D\text{-}\alpha (S',B',I',P') = D\text{-}\alpha$

$\langle proof \rangle$

lemma $pE\text{-}\alpha\text{-}indep[simp]$: $GS.pE\text{-}\alpha (S,B',I',P) = pE\text{-}\alpha$

$\langle proof \rangle$

definition $find\text{-}seg$ — Abs-path index for stack index

where $find\text{-}seg\ j \equiv Max \{ i. i < length\ B \wedge B!i \leq j \}$

definition $S\text{-}idx\text{-}of$ — Stack index for node

where $S\text{-}idx\text{-}of\ v \equiv case\ I\ v\ of\ Some\ (STACK\ i) \Rightarrow i$

end

locale $GS\text{-}invar = GS +$

assumes $B\text{-}in\text{-}bound$: $set\ B \subseteq \{ 0..<length\ S \}$

assumes $B\text{-}sorted$: $sorted\ B$

assumes $B\text{-}distinct$: $distinct\ B$

assumes $B0$: $S \neq [] \implies B \neq [] \wedge B!0 = 0$

assumes $S\text{-}distinct$: $distinct\ S$

assumes $I\text{-}consistent$: $(I\ v = Some\ (STACK\ j)) \longleftrightarrow (j < length\ S \wedge v = S!j)$

assumes $P\text{-}sorted$: $sorted\ (map\ fst\ P)$

assumes $P\text{-}distinct$: $distinct\ (map\ fst\ P)$

```

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

end

definition oGS-rel  $\equiv br\ oGS-\alpha\ oGS-invar$ 
lemma oGS-rel-sv[intro!,simp,relator-props]: single-valued oGS-rel
 $\langle proof \rangle$ 

definition GS-rel  $\equiv br\ GS-\alpha\ GS-invar$ 
lemma GS-rel-sv[intro!,simp,relator-props]: single-valued GS-rel
 $\langle proof \rangle$ 

context GS-invar
begin
  lemma empty-eq:  $S=[] \longleftrightarrow B=[]$ 
 $\langle proof \rangle$ 

  lemma B-in-bound':  $i < length\ B \implies B!i < length\ S$ 
 $\langle proof \rangle$ 

  lemma seg-start-bound:
    assumes A:  $i < length\ B$  shows seg-start  $i < length\ S$ 
 $\langle proof \rangle$ 

  lemma seg-end-bound:
    assumes A:  $i < length\ B$  shows seg-end  $i \leq length\ S$ 
 $\langle proof \rangle$ 

  lemma seg-start-less-end:  $i < length\ B \implies seg-start\ i < seg-end\ i$ 
 $\langle proof \rangle$ 

  lemma seg-end-less-start:  $\llbracket i < j; j < length\ B \rrbracket \implies seg-end\ i \leq seg-start\ j$ 
 $\langle proof \rangle$ 

  lemma find-seg-bounds:
    assumes A:  $j < length\ S$ 
    shows seg-start (find-seg  $j$ )  $\leq j$ 
    and  $j < seg-end$  (find-seg  $j$ )
    and find-seg  $j < length\ B$ 
 $\langle proof \rangle$ 

  lemma find-seg-correct:
    assumes A:  $j < length\ S$ 
    shows  $S!j \in seg$  (find-seg  $j$ ) and find-seg  $j < length\ B$ 
 $\langle proof \rangle$ 

  lemma set-p- $\alpha$ -is-set-S:

```

$\bigcup(\text{set } p\text{-}\alpha) = \text{set } S$
 <proof>

lemma *S-idx-uniq*:

$\llbracket i < \text{length } S; j < \text{length } S \rrbracket \implies S!i = S!j \longleftrightarrow i = j$
 <proof>

lemma *S-idx-of-correct*:

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

lemma *p-α-disjoint-sym*:

shows $\forall i j v. i < \text{length } p\text{-}\alpha \wedge j < \text{length } p\text{-}\alpha \wedge v \in p\text{-}\alpha!i \wedge v \in p\text{-}\alpha!j \longrightarrow i = j$
 <proof>

end

1.4.3 Refinement of the Operations

definition *GS-initial-impl* :: $'a \text{ oGS} \Rightarrow 'a \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ GS}$ **where**

GS-initial-impl $I v0 \text{ succs} \equiv$ (
 $[v0]$,
 $[0]$,
 $I(v0 \mapsto (\text{STACK } 0))$,
if succs = { } *then* [] *else* [(0, succs)])

context *GS*

begin

definition *push-impl* $v \text{ succs} \equiv$

let
 - = *stat-newnode* ();
 $j = \text{length } S$;
 $S = S@[v]$;
 $B = B@[j]$;
 $I = I(v \mapsto \text{STACK } j)$;
 $P = \text{if succs} = \{ \}$ *then* P *else* $P@[j, \text{succs}]$
in
 (S, B, I, P)

definition *mark-as-done*

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

definition *mark-as-done-abs* **where**

$\bigwedge l u I. \text{mark-as-done-abs } l u I$
 $\equiv (\lambda v. \text{if } v \in \{S!j \mid j. l \leq j \wedge j < u\} \text{ then Some DONE else } I v)$

lemma *mark-as-done-aux*:

fixes $l u I$
shows $\llbracket l < u; u \leq \text{length } S \rrbracket \implies \text{mark-as-done } l u I$
 $\leq \text{SPEC } (\lambda r. r = \text{mark-as-done-abs } l u I)$
 $\langle \text{proof} \rangle$

definition *pop-impl* \equiv

$\text{do } \{$
 $\text{let } lsi = \text{length } B - 1;$
 $\text{ASSERT } (lsi < \text{length } B);$
 $I \leftarrow \text{mark-as-done } (\text{seg-start } lsi) (\text{seg-end } lsi) I;$
 $\text{ASSERT } (B \neq []);$
 $\text{let } S = \text{take } (\text{last } B) S;$
 $\text{ASSERT } (B \neq []);$
 $\text{let } B = \text{butlast } B;$
 $\text{RETURN } (S, B, I, P)$
 $\}$

definition *sel-rem-last* \equiv

$\text{if } P = [] \text{ then}$
 $\text{RETURN } (\text{None}, (S, B, I, P))$
 $\text{else do } \{$
 $\text{let } (j, \text{succs}) = \text{last } P;$
 $\text{ASSERT } (\text{length } B - 1 < \text{length } B);$
 $\text{if } j \geq \text{seg-start } (\text{length } B - 1) \text{ then do } \{$
 $\text{ASSERT } (\text{succs} \neq \{\});$
 $v \leftarrow \text{SPEC } (\lambda x. x \in \text{succs});$
 $\text{let } \text{succs} = \text{succs} - \{v\};$
 $\text{ASSERT } (P \neq [] \wedge \text{length } P - 1 < \text{length } P);$
 $\text{let } P = (\text{if } \text{succs} = \{\} \text{ then butlast } P \text{ else } P[\text{length } P - 1 := (j, \text{succs})]);$
 $\text{RETURN } (\text{Some } v, (S, B, I, P))$
 $\} \text{ else RETURN } (\text{None}, (S, B, I, P))$
 $\}$

definition *find-seg-impl* $j \equiv \text{find-max-nat } (\text{length } B) (\lambda i. B!i \leq j)$

lemma (**in** *GS-invar*) *find-seg-impl*:

$j < \text{length } S \implies \text{find-seg-impl } j = \text{find-seg } j$
 $\langle \text{proof} \rangle$

definition *idx-of-impl* $v \equiv \text{do } \{$

$\text{ASSERT } (\exists i. I v = \text{Some } (\text{STACK } i));$

```

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

```

definition *collapse-impl* $v \equiv$

```

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*: $(I,D) \in oGS\text{-rel}$

assumes *A*: $v0 \notin D$

shows $GS.\alpha$ (*GS-initial-impl* I $v0$ *succs*) = $([\{v0\}], D, \{v0\} \times \text{succs})$ (is ?*G1*)

and *GS-invar* (*GS-initial-impl* I $v0$ *succs*) (is ?*G2*)

<proof>

context *GS-invar*

begin

lemma *push-correct*:

assumes *A*: $v \notin \bigcup (\text{set } p\text{-}\alpha)$ and *B*: $v \notin D\text{-}\alpha$

shows $GS.\alpha$ (*push-impl* v *succs*) = $(p\text{-}\alpha @ [\{v\}], D\text{-}\alpha, pE\text{-}\alpha \cup \{v\} \times \text{succs})$
(is ?*G1*)

and *GS-invar* (*push-impl* v *succs*) (is ?*G2*)

<proof>

lemma *no-last-out-P-aux*:

assumes *NE*: $p\text{-}\alpha \neq []$ and *NS*: $pE\text{-}\alpha \cap \text{last } p\text{-}\alpha \times UNIV = \{\}$

shows $\text{set } P \subseteq \{0..<\text{last } B\} \times UNIV$

<proof>

lemma *pop-correct*:

assumes *NE*: $p\text{-}\alpha \neq []$ and *NS*: $pE\text{-}\alpha \cap \text{last } p\text{-}\alpha \times UNIV = \{\}$

shows *pop-impl*

$\leq \Downarrow GS\text{-rel} (\text{SPEC } (\lambda r. r = (\text{butlast } p\text{-}\alpha, D\text{-}\alpha \cup \text{last } p\text{-}\alpha, pE\text{-}\alpha)))$

<proof>

lemma *sel-rem-last-correct*:

assumes *NE*: $p\text{-}\alpha \neq []$

shows

$\text{sel-rem-last} \leq \Downarrow (Id \times_r GS\text{-rel}) (\text{select-edge } (p\text{-}\alpha, D\text{-}\alpha, pE\text{-}\alpha))$

<proof>

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

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

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

end

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

lemma *opt-GSdef*: $f \equiv g \implies f \ s \equiv \text{case } s \ \text{of } (S, B, I, P) \Rightarrow g \ (S, B, I, P)$ $\langle \text{proof} \rangle$

lemma *ext-def*: $f \equiv g \implies f \ x \equiv g \ x$ $\langle \text{proof} \rangle$

context *fr-graph begin*

definition *push-impl* $v \ s \equiv \text{GS.push-impl } s \ v \ (E^{\{\!|v|\!\}})$

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* $v \ (S, B, I, P) \equiv (S@[v], B@[length \ S], I(v \mapsto \text{STACK } (length \ S)),$
if $E^{\{\!|v|\!\}} = \{\}$ *then* P *else* $P@[((length \ S, E^{\{\!|v|\!\}})]$
 $\langle \text{proof} \rangle$

lemma *GS- α -split*:

$\text{GS.}\alpha \ s = (p, D, pE) \longleftrightarrow (p = \text{GS.p-}\alpha \ s \wedge D = \text{GS.D-}\alpha \ s \wedge pE = \text{GS.pE-}\alpha \ s)$

$(p, D, pE) = \text{GS.}\alpha \ s \longleftrightarrow (p = \text{GS.p-}\alpha \ s \wedge D = \text{GS.D-}\alpha \ s \wedge pE = \text{GS.pE-}\alpha \ s)$

$\langle \text{proof} \rangle$

lemma *push-refine*:

assumes $A: (s, (p, D, pE)) \in \text{GS-rel} \quad (v, v') \in \text{Id}$

assumes $B: v \notin \bigcup (\text{set } p) \quad v \notin D$

shows $(\text{push-impl } v \ s, \text{push } v' \ (p, D, pE)) \in \text{GS-rel}$

$\langle \text{proof} \rangle$

definition *pop-impl* $s \equiv \text{GS.pop-impl } s$

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*: $p \neq [] \quad pE \cap \text{last } p \times UNIV = \{\}$
shows $\text{pop-impl } s \leq \Downarrow GS\text{-rel } (RETURN (\text{pop } (p, D, pE)))$
<proof>

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

definition *collapse-impl* $v \ s \equiv GS.\text{collapse-impl } s \ v$

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*: $v' \in \bigcup (\text{set } p)$
shows $\text{collapse-impl } v \ s \leq \Downarrow GS\text{-rel } (RETURN (\text{collapse } v' (p, D, pE)))$
<proof>

definition *select-edge-impl* $s \equiv GS.\text{sel-rem-last } s$

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 $\text{select-edge-impl } s \leq \Downarrow (Id \times_r GS\text{-rel}) (\text{select-edge } (p, D, pE))$
<proof>

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

lemma *initial-refine*:

$\llbracket v0 \notin D0; (I, D0) \in oGS\text{-rel}; (v0i, v0) \in Id \rrbracket$
 $\implies (\text{initial-impl } v0i \ I, \text{initial } v0 \ D0) \in GS\text{-rel}$
<proof>

definition *path-is-empty-impl* $s \equiv GS.S \ s = []$

lemma *path-is-empty-refine*:

$GS\text{-invar } s \implies \text{path-is-empty-impl } s \longleftrightarrow GS.p\text{-}\alpha \ s = []$
<proof>

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

$\equiv \text{case } I \text{ v of Some } (STACK \ -) \Rightarrow \text{True} \mid - \Rightarrow \text{False}$

lemma (in *GS-invar*) *is-on-stack-impl-correct*:
shows *is-on-stack-impl* $v \longleftrightarrow v \in \bigcup (\text{set } p\text{-}\alpha)$
<proof>

definition *is-on-stack-impl* $v \ s \equiv GS.is\text{-on}\text{-stack}\text{-impl } s \ v$

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*:
 $\llbracket GS\text{-invar } s \rrbracket \Longrightarrow \text{is-on-stack-impl } v \ s \longleftrightarrow v \in \bigcup (\text{set } (GS.p\text{-}\alpha \ s))$
<proof>

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

lemma (in *GS-invar*) *is-done-impl-correct*:
shows *is-done-impl* $v \longleftrightarrow v \in D\text{-}\alpha$
<proof>

definition *is-done-oimpl* $v \ I \equiv \text{case } I \text{ v of Some } DONE \Rightarrow \text{True} \mid - \Rightarrow \text{False}$

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

lemma *is-done-orefine*:
 $\llbracket oGS\text{-invar } s \rrbracket \Longrightarrow \text{is-done-oimpl } v \ s \longleftrightarrow v \in oGS\text{-}\alpha \ s$
<proof>

lemma *is-done-refine*:
 $\llbracket GS\text{-invar } s \rrbracket \Longrightarrow \text{is-done-impl } v \ s \longleftrightarrow v \in GS.D\text{-}\alpha \ s$
<proof>

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

end

1.4.4 Refined Skeleton Algorithm

context *fr-graph* **begin**

lemma *I-to-outer*:
assumes $((S, B, I, P), (\llbracket, D, \{\}\rrbracket)) \in GS\text{-rel}$
shows $(I, D) \in oGS\text{-rel}$
<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-oiimpl 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* ≤ ↓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 (λl.
  distinct l ∧ ∪(set l) = E* “V0 ∧ (∀ U ∈ set l. is-scc E U)
  ∧ (∀ i j. i < j ∧ j < length l → !j × !i ∩ E* = {}) )
end

```

1.7 Extended Invariant

```

locale csc-cc-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]] ⇒ !j × !i ∩ E* = {}
  — The output contains no forward edges
begin
  lemma l-no-empty: {} ∉ set l ⟨proof⟩
end

```

```

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

```

```

locale csc-invar-loc = invar-loc G v0 D0 p D pE + csc-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: csc-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 csc-outer-invar  $\equiv \lambda it (l,D). csc-outer-invar-loc\ G\ it\ l\ D$ 
  definition csc-invar  $\equiv \lambda v0\ D0\ (l,p,D,pE). csc-invar-loc\ G\ v0\ D0\ l\ p\ D\ pE$ 
end

```

1.8 Definition of the SCC-Algorithm

```

context fr-graph
begin
  definition compute-SCC :: 'v set list nres where
    compute-SCC  $\equiv do \{$ 
      let so = ( $\[], \{\}$ );
      (l,D)  $\leftarrow FOREACHi\ csc-outer-invar\ V0\ (\lambda v0\ (l,D0). do \{$ 
        if  $v0 \notin D0$  then do  $\{$ 
          let s = (l,initial v0 D0);

          (l,p,D,pE)  $\leftarrow$ 
            WHILEIT (csc-invar v0 D0)
              ( $\lambda(l,p,D,pE). p \neq []$ ) ( $\lambda(l,p,D,pE).$ 
                do  $\{$ 
                  — Select edge from end of path
                  (vo,(p,D,pE))  $\leftarrow select-edge\ (p,D,pE)$ ;

                  ASSERT ( $p \neq []$ );
                  case vo of
                    Some v  $\Rightarrow do \{$ 
                      if  $v \in \bigcup (set\ p)$  then do  $\{$ 
                        — Collapse
                        RETURN (l,collapse v (p,D,pE))
                       $\}$  else if  $v \notin D$  then do  $\{$ 
                        — Edge to new node. Append to path
                        RETURN (l,push v (p,D,pE))
                       $\}$ 
                     $\}$ 
                 $\}$ 
            )
         $\}$ 
      )
     $\}$ 

```

```

    } else RETURN (l,p,D,pE)
  }
  | None  $\Rightarrow$  do {
    — No more outgoing edges from current node on path
    ASSERT (pE  $\cap$  last p  $\times$  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=[]  $\wedge$  pE={});
RETURN (l,D)
} else
  RETURN (l,D0)
}) so;
RETURN l
}
end

```

1.9 Preservation of Invariant Extension

context *csc- $invar$ -ext*

begin

lemma *l-disjoint*:

assumes *A*: $i < j \quad j < \text{length } l$

shows $!!i \cap !!j = \{\}$

<proof>

corollary *l-distinct*: *distinct l*

<proof>

end

context *fr-graph*

begin

definition *csc- $invar$ -part* $\equiv \lambda(l,p,D,pE). \text{csc-}invar\text{-ext } G \ l \ D$

lemma *csc- $invar$ I*[*intro?*]:

assumes *invar v0 D0 PDPE*

assumes *invar v0 D0 PDPE* \implies *csc- $invar$ -part (l,PDPE)*

shows *csc- $invar$ v0 D0 (l,PDPE)*

<proof>

thm *csc- $invar$ I*[*of v-0 D-0 s l*]

lemma *csc-outer- $invar$ I*[*intro?*]:

assumes *outer- $invar$ it D*

assumes *outer- $invar$ it D* \implies *csc- $invar$ -ext G l D*

shows *csc-outer- $invar$ it (l,D)*

$\langle proof \rangle$

lemma *csc- $invar$ -initial*[*simp, intro!*]:
 assumes $A: v0 \in it \quad v0 \notin D0$
 assumes $INV: csc\text{-}outer\text{-}invar\ it\ (l, D0)$
 shows $csc\text{-}invar\text{-}part\ (l, initial\ v0\ D0)$
 $\langle proof \rangle$

lemma *csc- $invar$ -pop*:
 assumes $INV: csc\text{-}invar\ v0\ D0\ (l, p, D, pE)$
 assumes $invar\ v0\ D0\ (pop\ (p, D, pE))$
 assumes $NE[simp]: p \neq []$
 assumes $NO': pE \cap (last\ p \times UNIV) = \{\}$
 shows $csc\text{-}invar\text{-}part\ (last\ p \# l, pop\ (p, D, pE))$
 $\langle proof \rangle$

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

lemma *csc- $invar$ -unchanged*:
 assumes $INV: csc\text{-}invar\ v0\ D0\ (l, p, D, pE)$
 shows $csc\text{-}invar\text{-}part\ (l, p', D, pE')$
 $\langle proof \rangle$

corollary *csc- $invar$ -collapse*:
 assumes $INV: csc\text{-}invar\ v0\ D0\ (l, p, D, pE)$
 shows $csc\text{-}invar\text{-}part\ (l, collapse\ v\ (p', D, pE'))$
 $\langle proof \rangle$

corollary *csc- $invar$ -push*:
 assumes $INV: csc\text{-}invar\ v0\ D0\ (l, p, D, pE)$
 shows $csc\text{-}invar\text{-}part\ (l, push\ v\ (p', D, pE'))$
 $\langle proof \rangle$

lemma *csc- $outer$ - $invar$ -initial*: $csc\text{-}invar\text{-}ext\ G\ []\ \{\}$
 $\langle proof \rangle$

lemma *csc- $invar$ - $outer$ - $newnode$* :
 assumes $A: v0 \notin D0 \quad v0 \in it$
 assumes $OINV: csc\text{-}outer\text{-}invar\ it\ (l, D0)$
 assumes $INV: csc\text{-}invar\ v0\ D0\ (l', [], D', pE)$
 shows $csc\text{-}invar\text{-}ext\ G\ l'\ D'$
 $\langle proof \rangle$

lemma *csc- $invar$ - $outer$ - $Dnode$* :
 assumes $csc\text{-}outer\text{-}invar\ it\ (l, D)$
 shows $csc\text{-}invar\text{-}ext\ G\ l\ D$
 $\langle proof \rangle$

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

On termination, the invariant implies the specification

lemma *csc*-finI:
assumes *INV*: *csc*-outer-invar {} (l,D)
shows *fin*-l-is-scc: $\llbracket U \in \text{set } l \rrbracket \implies \text{is-scc } E \ U$
and *fin*-l-distinct: distinct l
and *fin*-l-is-reachable: $\bigcup (\text{set } l) = E^* \ \text{“ } V0$
and *fin*-l-no-fwd: $\llbracket i < j; j < \text{length } l \rrbracket \implies \llbracket j \times \llbracket i \cap E^* = \{\} \rrbracket$
 <proof>

end

1.10 Main Correctness Proof

context *fr*-graph

begin

lemma *invar*-from-*csc*-invarI: *csc*-invar v0 D0 (L,PDPE) \implies invar v0 D0 PDPE
 <proof>

lemma *outer*-invar-from-*csc*-invarI:
csc-outer-invar it (L,D) \implies 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 \leq *compute*-SCC-spec
 <proof>

Simple proof, for presentation

context
notes [*refine*]=*refine*-vcg
notes [[*goals*-limit = 1]]
begin
theorem *compute*-SCC \leq *compute*-SCC-spec
 <proof>
end

end

1.11 Refinement to Gabow's Data Structure

context *GS* **begin**

definition *seg*-set-impl l u \equiv 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  $\llbracket l < u; u \leq \text{length } S; \text{distinct } S \rrbracket \implies \text{seg-set-impl } l \ u$ 
   $\leq \text{SPEC } (\lambda r. r = \{S!j \mid j. l \leq j \wedge j < u\})$ 
  <proof>

lemma (in GS-invar) seg-set-impl-correct:
  assumes  $i < \text{length } B$ 
  shows  $\text{seg-set-impl } (\text{seg-start } i) (\text{seg-end } i) \leq \text{SPEC } (\lambda r. r = p\text{-}\alpha!i)$ 
  <proof>

definition last-seg-impl
   $\equiv \text{do } \{$ 
    ASSERT ( $\text{length } B - 1 < \text{length } B$ );
     $\text{seg-set-impl } (\text{seg-start } (\text{length } B - 1)) (\text{seg-end } (\text{length } B - 1))$ 
  }

lemma (in GS-invar) last-seg-impl-correct:
  assumes  $p\text{-}\alpha \neq []$ 
  shows  $\text{last-seg-impl} \leq \text{SPEC } (\lambda r. r = \text{last } p\text{-}\alpha)$ 
  <proof>

end

context fr-graph
begin

definition last-seg-impl s  $\equiv \text{GS.last-seg-impl } s$ 
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  $A: (s,(p,D,pE)) \in \text{GS-rel}$ 

```

assumes $NE: p \neq []$
shows $last-seg-impl\ s \leq \Downarrow Id\ (RETURN\ (last\ p))$
 ⟨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). csc-outer-invar it (l,oGS-α s))
  V0 (λv0 (l,I0). do {
    if ¬is-done-impl v0 I0 then do {
      let ls = (l,initial-impl v0 I0);

      (l,(S,B,I,P)) ← WHILEIT (λ(l,s). csc-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
        (v0,s) ← select-edge-impl s;

        case v0 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 $compute-SCC-impl-refine: compute-SCC-impl \leq \Downarrow Id\ compute-SCC$
 ⟨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 \equiv do\ \{\$
 $ASSERT\ (finite\ U0);$
 $ASSERT\ (finite\ (E^*\ \{\!-\!U0\}));$
 $SPEC\ (\lambda p.\ case\ p\ of$
 $Some\ (p,v) \Rightarrow \exists u0 \in U0.\ path\ E\ u0\ p\ v \wedge P\ v \wedge (\forall v \in set\ p.\ \neg P\ v)$
 | $None \Rightarrow \forall u0 \in U0.\ \forall v \in E^*\ \{\!-\!u0\}.\ \neg P\ v)$
 $\}$

lemma *find-path-ex-rule:*

assumes *finite* $U0$

assumes *finite* $(E^*\ \{\!-\!U0\})$

assumes $\exists v \in E^*\ \{\!-\!U0\}.\ P\ v$

shows *find-path* $E\ U0\ P \leq SPEC\ (\lambda r.$

$\exists p\ v.\ r = Some\ (p,v) \wedge P\ v \wedge (\forall v \in set\ p.\ \neg P\ v) \wedge (\exists u0 \in U0.\ path\ E\ u0\ p\ v))$

<proof>

1.13.1 Nontrivial Paths

definition *find-path1* $E\ u0\ P \equiv do\ \{\$
 $ASSERT\ (finite\ (E^*\ \{\!-\!u0\}));$
 $SPEC\ (\lambda p.\ case\ p\ of$
 $Some\ (p,v) \Rightarrow path\ E\ u0\ p\ v \wedge P\ v \wedge p \neq []$
 | $None \Rightarrow \forall v \in E^+\ \{\!-\!u0\}.\ \neg P\ v)$
 $\}$

lemma (**in** $-$) *find-path1-ex-rule:*

assumes *finite* $(E^*\ \{\!-\!u0\})$

assumes $\exists v \in E^+\ \{\!-\!u0\}.\ P\ v$

shows *find-path1* $E\ u0\ P \leq SPEC\ (\lambda r.$

$\exists p\ v.\ r = Some\ (p,v) \wedge p \neq [] \wedge P\ v \wedge path\ E\ u0\ p\ v)$

<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 (λr. 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 (λr. 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 $D \equiv \neg(\exists v \in D. \exists pl. pl \neq [] \wedge path\ E\ v\ pl\ v \wedge$
 $(\forall i < num-acc. \exists q \in set\ pl. i \in acc\ q))$

definition *fgl-outer-invar-ext* $\equiv \lambda it\ (brk, D).$

case *brk* *of* *None* $\Rightarrow no-acc-over\ D \mid Some\ (Vr, Vl) \Rightarrow ce-correct\ Vr\ Vl$

definition *fgl-outer-invar* $\equiv \lambda it\ (brk, D).$ *case* *brk* *of*

None $\Rightarrow outer-invar\ it\ D \wedge no-acc-over\ D$

$\mid Some\ (Vr, Vl) \Rightarrow ce-correct\ Vr\ Vl$

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** $brk :: ('Q\ set \times 'Q\ set)\ option$ **and** $p\ D\ pE$ +

assumes *no-acc*: $brk = None \Longrightarrow \neg(\exists v\ pl. pl \neq [] \wedge path\ lvE\ v\ pl\ v \wedge$

$(\forall i < num-acc. \exists q \in set\ pl. i \in acc\ q))$ — No accepting cycle over visited edges

assumes *acc*: $brk = Some\ (Vr, Vl) \Longrightarrow ce-correct\ Vr\ Vl$

begin

lemma *locale-this*: *fgl-invar-loc* $G\ v0\ D0\ brk\ p\ D\ pE$

$\langle proof \rangle$

lemma *invar-loc-this*: *invar-loc* $G\ v0\ D0\ p\ D\ pE$ $\langle proof \rangle$

lemma *eas-gba-graph-this*: *igb-graph* G $\langle proof \rangle$

end

definition (**in** *igb-graph*) *fgl-invar* $v0\ D0 \equiv$

$\lambda(brk, p, D, pE). fgl-invar-loc\ G\ v0\ D0\ brk\ p\ D\ pE$

1.17 Definition of the Lasso-Finding Algorithm

context *igb-fr-graph*

begin

definition *find-ce* $:: ('Q\ set \times 'Q\ set)\ option\ nres$ **where**

find-ce $\equiv do\ \{$

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 $fgl\text{-invar}\text{-part} \equiv \lambda(brk, p, D, pE).$

$fgl\text{-invar}\text{-loc}\text{-axioms} \ G \ brk \ p \ D \ pE$

lemma $fgl\text{-outer}\text{-invar}I[intro?]:$

[[
 $brk=None \implies outer\text{-invar} \ it \ D;$
 $[[brk=None \implies outer\text{-invar} \ it \ D]] \implies fgl\text{-outer}\text{-invar}\text{-ext} \ it \ (brk,D)]$
 $\implies fgl\text{-outer}\text{-invar} \ it \ (brk,D)$
 $\langle proof \rangle$

lemma $fgl\text{-invar}I[intro?]:$

[[
 $invar \ v0 \ D0 \ PDPE;$
 $invar \ v0 \ D0 \ PDPE \implies fgl\text{-invar}\text{-part} \ (B,PDPE)]$
 $\implies fgl\text{-invar} \ v0 \ D0 \ (B,PDPE)$
 $\langle proof \rangle$

lemma $fgl\text{-invar}\text{-initial}:$

assumes $OINV: fgl\text{-outer}\text{-invar} \ it \ (None,D0)$
assumes $A: v0 \in it \quad v0 \notin D0$
shows $fgl\text{-invar}\text{-part} \ (None, \text{initial} \ v0 \ D0)$
 $\langle proof \rangle$

lemma $fgl\text{-invar}\text{-pop}:$

assumes $INV: fgl\text{-invar} \ v0 \ D0 \ (None,p,D,pE)$
assumes $INV': invar \ v0 \ D0 \ (pop \ (p,D,pE))$
assumes $NE[simp]: p \neq []$
assumes $NO': pE \cap last \ p \times UNIV = \{\}$
shows $fgl\text{-invar}\text{-part} \ (None, \text{pop} \ (p,D,pE))$
 $\langle proof \rangle$

lemma $fgl\text{-invar}\text{-collapse}\text{-ce}\text{-aux}:$

assumes $INV: invar \ v0 \ D0 \ (p, D, pE)$
assumes $NE[simp]: p \neq []$
assumes $NONTRIV: vE \ p \ D \ pE \cap (last \ p \times last \ p) \neq \{\}$
assumes $ACC: \forall i < num\text{-acc}. \exists q \in last \ p. i \in acc \ q$
shows $fgl\text{-invar}\text{-part} \ (Some \ (\bigcup \ (set \ (butlast \ p))), last \ p), p, D, pE)$
 $\langle proof \rangle$

lemma $fgl\text{-invar}\text{-collapse}\text{-ce}:$

fixes $u \ v$
assumes $INV: fgl\text{-invar} \ v0 \ D0 \ (None,p,D,pE)$
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 $E: (u,v) \in pE \ \text{and} \ u \in last \ p$
assumes $BACK: v \in \bigcup \ (set \ p)$

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

lemma $fgl\text{-}invar\text{-}collapse\text{-}nce$:

fixes $u\ v$
assumes $INV: fgl\text{-}invar\ v0\ D0\ (None, p, D, pE)$
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 $NACC: j < num\text{-}acc\ \ \forall q \in last\ p'. j \notin acc\ q$
defines $i \equiv idx\text{-}of\ p\ v$
shows $fgl\text{-}invar\text{-}part\ (None, collapse\ v\ (p, D, pE'))$
 $\langle proof \rangle$

lemma $collapse\text{-}ne: ([], D', pE') \neq collapse\ v\ (p, D, pE)$
 $\langle proof \rangle$

lemma $fgl\text{-}invar\text{-}push$:

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

lemma $fgl\text{-}invar\text{-}skip$:

assumes $INV: fgl\text{-}invar\ v0\ D0\ (None, p, D, pE)$
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 $fgl\text{-}invar\text{-}part\ (None, p, D, (pE - \{(u, v)\}))$
 $\langle proof \rangle$

lemma $fgl\text{-}outer\text{-}invar\text{-}initial$:

$outer\text{-}invar\ V0\ \{\} \implies fgl\text{-}outer\text{-}invar\text{-}ext\ V0\ (None, \{\})$
 $\langle proof \rangle$

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)*
 \langle *proof* \rangle

lemma *fgl-outer-invar-newnode-nobrk*:
assumes *A*: $v0 \notin D0 \quad v0 \in it$
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')*
 \langle *proof* \rangle

lemma *fgl-outer-invar-newnode*:
assumes *A*: $v0 \notin D0 \quad v0 \in it$
assumes *OINV*: *fgl-outer-invar it (None, D0)*
assumes *INV*: *fgl-invar v0 D0 (brk, p, D', pE)*
assumes *CASES*: $(\exists Vr Vl. brk = Some (Vr, Vl)) \vee p = []$
shows *fgl-outer-invar-ext (it - {v0}) (brk, D')*
 \langle *proof* \rangle

lemma *fgl-outer-invar-Dnode*:
assumes *fgl-outer-invar it (None, D) v ∈ D*
shows *fgl-outer-invar-ext (it - {v}) (None, D)*
 \langle *proof* \rangle

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

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

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) \implies outer-invar it D

\langle proof \rangle

lemma *invar-from-fgl-invarI*: *fgl-invar v0 D0 (B,PDPE) \implies invar v0 D0 PDPE*

\langle proof \rangle

theorem *find-ce-correct*: *find-ce \leq find-ce-spec*

\langle proof \rangle

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 ($\lambda res. res \longleftrightarrow (\forall r. \neg is-acc-run r)$)

\langle proof \rangle

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* ($\lambda (pr, pl).$

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 ($\lambda res. case res of$*

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 explicitly 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 explicitly 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* (λ*V*. ∪(*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* $s \equiv$

let $(a,s) = s$ *in* (*butlast* a, pop s)

definition *ginitial* $:: 'Q \Rightarrow 'Q \text{ abs-gostate} \Rightarrow 'Q \text{ abs-gstate}$

where *ginitial* $v0\ s0 \equiv ([acc\ v0], initial\ v0\ (snd\ s0))$

definition *goinitial* $:: 'Q \text{ abs-gostate}$ **where** *goinitial* $\equiv (None, \{\})$

definition *go-is-no-brk* $:: 'Q \text{ abs-gostate} \Rightarrow bool$

where *go-is-no-brk* $s \equiv fst\ s = None$

definition *goD* $:: 'Q \text{ abs-gostate} \Rightarrow 'Q \text{ set}$ **where** *goD* $s \equiv snd\ s$

definition *goBrk* $:: 'Q \text{ abs-gostate} \Rightarrow 'Q \text{ ce}$ **where** *goBrk* $s \equiv fst\ s$

definition *gto-outer* $:: 'Q \text{ ce} \Rightarrow 'Q \text{ abs-gstate} \Rightarrow 'Q \text{ abs-gostate}$

where *gto-outer* $brk\ s \equiv let\ (A,p,D,pE)=s\ in\ (brk,D)$

definition *gselect-edge* $s \equiv do\ \{$

let $(a,s)=s;$

$(r,s) \leftarrow select-edge\ s;$

RETURN (r,a,s)

$\}$

definition *gfind-ce* $:: ('Q \text{ set} \times 'Q \text{ set}) \text{ option nres}$ **where**

gfind-ce $\equiv do\ \{$

let $os = goinitial;$

$os \leftarrow FOREACHci\ fgl-outer-invar\ V0\ (go-is-no-brk)\ (\lambda v0\ s0.\ do\ \{$

if $v0 \notin goD\ s0$ *then* *do* $\{$

let $s = (None, goinitial\ v0\ s0);$

$(brk,a,p,D,pE) \leftarrow WHILEIT\ (\lambda(brk,a,s).\ fgl-invar\ v0\ (goD\ s0)\ (brk,s))$

$(\lambda(brk,a,p,D,pE).\ brk=None \wedge p \neq [])\ (\lambda(-,a,p,D,pE).$

do $\{$

— *Select edge from end of path*

$(vo,(a,p,D,pE)) \leftarrow gselect-edge\ (a,p,D,pE);$

ASSERT $(p \neq []);$

case vo *of*

Some $v \Rightarrow do\ \{$

if $v \in \bigcup (set\ p)$ *then* *do* $\{$

— *Collapse*

let $(a,p,D,pE) = gcollapse\ v\ (a,p,D,pE);$

ASSERT $(p \neq []);$

ASSERT $(a \neq []);$

if *last* $a = \{0..<num-acc\}$ *then*

RETURN $(Some\ (\bigcup (set\ (butlast\ p)), last\ p), a, p, D, pE)$

else

RETURN $(None, a, p, D, pE)$

$\}$ *else if* $v \notin D$ *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*:

$\llbracket (v', v) \in Id; (s', s) \in gstate-rel \rrbracket$
 $\implies (gcollapse\ v'\ s', collapse\ v\ s) \in gstate-rel$
 $\langle proof \rangle$

lemma *gpush-refine*:

$\llbracket (v', v) \in Id; (s', s) \in gstate-rel \rrbracket \implies (gpush\ v'\ s', push\ v\ s) \in gstate-rel$
 $\langle proof \rangle$

lemma *gpop-refine*:

$\llbracket (s', s) \in gstate-rel \rrbracket \implies (gpopt\ s', pop\ s) \in gstate-rel$
 $\langle proof \rangle$

lemma *ginitial-refine*:

$(ginitial\ x\ (None, b), initial\ x\ b) \in gstate-rel$
 $\langle proof \rangle$

lemma *oinitial-b-refine*: $((None, \{\}), (None, \{\})) \in Id \times_r Id \langle proof \rangle$

lemma *gselect-edge-refine*: $\llbracket (s', s) \in gstate-rel \rrbracket \implies gselect-edge\ s'$

$\leq \Downarrow (\langle Id \rangle option-rel \times_r gstate-rel) (select-edge\ s)$
 $\langle proof \rangle$

lemma *last-acc-impl*:

assumes $p \neq []$
assumes $((a', p', D', pE'), (p, D, pE)) \in gstate-rel$
shows $(last\ a' = \{0..<num-acc\}) = (\forall i < num-acc. \exists q \in last\ p. i \in acc\ q)$
 $\langle proof \rangle$

lemma *fglr-aux1*:

assumes $V: (v', v) \in Id$ **and** $S: (s', s) \in gstate-rel$
and $P: \bigwedge a'\ p'\ D'\ pE'\ p\ D\ pE. ((a', p', D', pE'), (p, D, pE)) \in gstate-rel$

$\implies f' a' p' D' pE' \leq \Downarrow R (f p D pE)$
shows $(\text{let } (a', p', D', pE') = \text{gcollapse } v' s' \text{ in } f' a' p' D' pE')$
 $\leq \Downarrow R (\text{let } (p, D, pE) = \text{collapse } v s \text{ in } f p D pE)$
 $\langle \text{proof} \rangle$

lemma *gstate-invar-empty*:
 $\text{gstate-invar } (a, [], D, pE) \implies a = []$
 $\text{gstate-invar } ([], p, D, pE) \implies p = []$
 $\langle \text{proof} \rangle$

lemma *find-ce-refine*: $\text{gfind-ce} \leq \Downarrow \text{Id find-ce}$
 $\langle \text{proof} \rangle$

end

1.21.2 Refinement to Gabow's Data Structure

Preliminaries definition *Un-set-drop-impl* :: $\text{nat} \Rightarrow 'a \text{ set list} \Rightarrow 'a \text{ set nres}$

— Executable version of $\bigcup \text{set } (\text{drop } i A)$, using indexing to access A

where *Un-set-drop-impl* $i A \equiv$

$\text{do } \{$
 $(-, \text{res}) \leftarrow \text{WHILET } (\lambda(i, \text{res}). i < \text{length } A) (\lambda(i, \text{res}). \text{do } \{$
 $\text{ASSERT } (i < \text{length } A);$
 $\text{let } \text{res} = A!i \cup \text{res};$
 $\text{let } i = i + 1;$
 $\text{RETURN } (i, \text{res})$
 $\}) (i, \{\});$
 $\text{RETURN } \text{res}$
 $\}$

lemma *Un-set-drop-impl-correct*:

$\text{Un-set-drop-impl } i A \leq \text{SPEC } (\lambda r. r = \bigcup (\text{set } (\text{drop } i A)))$
 $\langle \text{proof} \rangle$

schematic-goal *Un-set-drop-code-aux*:

assumes $[\text{autoref-rules}]: (es\text{-impl}, \{\}) \in \langle R \rangle Rs$

assumes $[\text{autoref-rules}]: (un\text{-impl}, (\cup)) \in \langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs$

shows $(?c, \text{Un-set-drop-impl}) \in \text{nat-rel} \rightarrow \langle \langle R \rangle Rs \rangle \text{as-rel} \rightarrow \langle \langle R \rangle Rs \rangle \text{nres-rel}$

$\langle \text{proof} \rangle$

concrete-definition *Un-set-drop-code* **uses** *Un-set-drop-code-aux*

schematic-goal *Un-set-drop-tr-aux*:

$\text{RETURN } ?c \leq \text{Un-set-drop-code } es\text{-impl } un\text{-impl } i A$

$\langle \text{proof} \rangle$

concrete-definition *Un-set-drop-tr* **for** *es-impl un-impl* $i A$

uses *Un-set-drop-tr-aux*

lemma *Un-set-drop-autoref* $[\text{autoref-rules}]$:

assumes *GEN-OP* *es-impl* $\{\} (\langle R \rangle Rs)$

assumes *GEN-OP* *un-impl* $(\cup) (\langle R \rangle Rs \rightarrow \langle R \rangle Rs \rightarrow \langle R \rangle Rs)$

shows $(\lambda i A. RETURN (Un-set-drop-tr es-impl un-impl i A), Un-set-drop-impl)$
 $\in nat-rel \rightarrow \langle \langle R \rangle Rs \rangle as-rel \rightarrow \langle \langle R \rangle Rs \rangle nres-rel$
 $\langle proof \rangle$

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 $gGS-invar :: 'Q\ gGS \Rightarrow bool$

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

definition $gGS-\alpha :: 'Q\ gGS \Rightarrow 'Q\ abs-gstate$

where $gGS-\alpha\ s \equiv let\ (a,s)=s\ in\ (a,GS.\alpha\ s)$

definition $gGS-rel \equiv br\ gGS-\alpha\ gGS-invar$

lemma $gGS-rel-sv[relator-props,intro!,simp]$: *single-valued gGS-rel*
 $\langle proof \rangle$

definition $goGS-invar :: 'Q\ goGS \Rightarrow bool$ **where**

$goGS-invar\ s \equiv let\ (brk,ogs)=s\ in\ brk=None \longrightarrow oGS-invar\ ogs$

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

definition $goGS-rel \equiv br\ goGS-\alpha\ goGS-invar$

lemma $goGS-rel-sv[relator-props,intro!,simp]$: *single-valued goGS-rel*
 $\langle proof \rangle$

end

context *igb-fr-graph*

begin

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

$\langle proof \rangle$

definition *goinitial-impl* :: 'Q goGS

where *goinitial-impl* \equiv (None, Map.empty)

lemma *goinitial-impl-refine*: (*goinitial-impl*, *goinitial*) \in goGS-rel

$\langle proof \rangle$

definition *gto-outer-impl* :: 'Q ce \Rightarrow 'Q gGS \Rightarrow 'Q goGS

where *gto-outer-impl* *brk* *s* \equiv let (A,S,B,I,P)=*s* in (*brk*,I)

lemma *gto-outer-refine*:

assumes *A*: *brk* = None \longrightarrow (*p*=[] \wedge *pE*={})

assumes *B*: (*s*, (A,*p*, *D*, *pE*)) \in gGS-rel

assumes *C*: (*brk'*,*brk*) \in Id

shows (*gto-outer-impl* *brk'* *s*, *gto-outer* *brk* (A,*p*,*D*,*pE*)) \in goGS-rel

$\langle proof \rangle$

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

lemma *gpush-impl-refine*:

assumes *B*: (*s'*, (a,*p*,*D*,*pE*)) \in gGS-rel

assumes *A*: (*v'*,*v*) \in Id

assumes *PRE*: *v'* \notin \bigcup (set *p*) $\quad v'$ \notin *D*

shows (*gpush-impl* *v'* *s'*, *gpush* *v* (a,*p*,*D*,*pE*)) \in gGS-rel

$\langle proof \rangle$

definition *gpop-impl* :: 'Q gGS \Rightarrow 'Q gGS nres

where *gpop-impl* *s* \equiv do {

let (a,*s*)=*s*;

s \leftarrow *pop-impl* *s*;

ASSERT (a \neq []);

let a = butlast a;

RETURN (a,*s*)

}

lemma *gpop-impl-refine*:

assumes *A*: (*s'*, (a,*p*,*D*,*pE*)) \in gGS-rel

assumes *PRE*: *p* \neq [] $\quad pE \cap$ last *p* \times UNIV = {}

shows *gpop-impl* *s'* \leq \Downarrow gGS-rel (RETURN (*gpop* (a,*p*,*D*,*pE*)))

$\langle proof \rangle$

definition *gselect-edge-impl* :: 'Q gGS \Rightarrow ('Q option \times 'Q gGS) nres

where *gselect-edge-impl* *s* \equiv

do {

let (a,*s*)=*s*;

(*vo*,*s*) \leftarrow *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 $gselect\text{-edge-impl } s' \leq \Downarrow (Id \times_r gGS\text{-rel}) (gselect\text{-edge } (a, p, D, pE))$

<proof>

term $GS.idx\text{-of-impl}$

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

definition $gcollapse\text{-impl-aux} :: 'Q \Rightarrow 'Q gGS \Rightarrow 'Q gGS nres$ **where**

$gcollapse\text{-impl-aux } v s \equiv$

do {

let $(A,s)=s;$

~~*ASSERT* $(v \in \text{set } (GS.idx\text{-of-impl } s))$;~~

$i \leftarrow GS.idx\text{-of-impl } s v;$

$s \leftarrow collapse\text{-impl } v s;$

ASSERT $(i < length A);$

$us \leftarrow Un\text{-set-drop-impl } i A;$

let $A = take\ i\ A\ @\ [us];$

RETURN (A,s)

}

term *collapse*

lemma *gcollapse-alt*:

$gcollapse\ v\ APDPE = ($

let

$(a,p,D,pE)=APDPE;$

$i=idx\text{-of } p\ v;$

$s=collapse\ v\ (p,D,pE);$

$us=\bigcup (set\ (drop\ i\ a));$

$a = take\ i\ a\ @\ [us]$

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 $gcollapse\text{-impl-aux } v' s'$

$\leq \Downarrow gGS\text{-rel } (RETURN\ (gcollapse\ v\ (a, p, D, pE)))$

<proof>

definition $gcollapse-impl :: 'Q \Rightarrow 'Q \text{ gGS} \Rightarrow 'Q \text{ gGS nres}$
where $gcollapse-impl \ v \ s \equiv$
do {
 let $(A,S,B,I,P)=s$;
 $i \leftarrow GS.idx-of-impl \ (S,B,I,P) \ v$;
 ASSERT $(i+1 \leq \text{length } B)$;
 let $B = \text{take } (i+1) \ B$;
 ASSERT $(i < \text{length } A)$;
 $us \leftarrow Un-set-drop-impl \ i \ A$;
 let $A = \text{take } i \ A \ @ \ [us]$;
 RETURN (A,S,B,I,P)
}

lemma $gcollapse-impl-aux-opt-refine$:
 $gcollapse-impl \ v \ s \leq gcollapse-impl-aux \ v \ s$
<proof>

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

definition $ginitial-impl :: 'Q \Rightarrow 'Q \text{ goGS} \Rightarrow 'Q \text{ gGS}$
where $ginitial-impl \ v0 \ s0 \equiv ([acc \ v0], initial-impl \ v0 \ (\text{snd } s0))$

lemma $ginitial-impl-refine$:
assumes $A: v0 \notin goD \ s0 \quad go-is-no-brk \ s0$
assumes $REL: (s0i, s0) \in goGS-rel \quad (v0i, v0) \in Id$
shows $(ginitial-impl \ v0i \ s0i, ginitial \ v0 \ s0) \in \text{gGS-rel}$
<proof>

definition $gpath-is-empty-impl :: 'Q \text{ gGS} \Rightarrow \text{bool}$
where $gpath-is-empty-impl \ s = \text{path-is-empty-impl} \ (\text{snd } s)$

lemma $gpath-is-empty-refine$:
 $(s, (a, p, D, pE)) \in \text{gGS-rel} \implies gpath-is-empty-impl \ s \longleftrightarrow p = []$
<proof>

definition $gis-on-stack-impl :: 'Q \Rightarrow 'Q \text{ gGS} \Rightarrow \text{bool}$
where $gis-on-stack-impl \ v \ s = \text{is-on-stack-impl} \ v \ (\text{snd } s)$

lemma $gis-on-stack-refine$:
 $[(s, (a, p, D, pE)) \in \text{gGS-rel}] \implies gis-on-stack-impl \ v \ s \longleftrightarrow v \in \bigcup (\text{set } p)$
<proof>

definition $gis-done-impl :: 'Q \Rightarrow 'Q \text{ gGS} \Rightarrow \text{bool}$
where $gis-done-impl \ v \ s \equiv \text{is-done-impl} \ v \ (\text{snd } s)$

thm *is-done-refine*

lemma *gis-done-refine*: $(s, (a, p, D, pE)) \in gGS\text{-rel}$

\implies *gis-done-impl* $v\ s \longleftrightarrow (v \in D)$

<proof>

definition (**in** $-$) *on-stack-less* $I\ u\ v \equiv$

case $I\ v$ *of*

Some $(STACK\ j) \Rightarrow j < u$

$| - \Rightarrow False$

definition (**in** $-$) *on-stack-ge* $I\ l\ v \equiv$

case $I\ v$ *of*

Some $(STACK\ j) \Rightarrow l \leq j$

$| - \Rightarrow False$

lemma (**in** *GS-invar*) *set-butlast-p-refine*:

assumes *PRE*: $p \neq []$

shows *Collect* $(on\text{-}stack\text{-}less\ I\ (last\ B)) = \bigcup (set\ (butlast\ p\ \alpha))$ (**is** $?L = ?R$)

<proof>

lemma (**in** *GS-invar*) *set-last-p-refine*:

assumes *PRE*: $p \neq []$

shows *Collect* $(on\text{-}stack\text{-}ge\ I\ (last\ B)) = last\ p\ \alpha$ (**is** $?L = ?R$)

<proof>

definition *ce-impl* $:: 'Q\ gGS \Rightarrow (('Q\ set \times 'Q\ set)\ option \times 'Q\ gGS)\ nres$

where *ce-impl* $s \equiv$

do {

let $(a, S, B, I, P) = s$;

ASSERT $(B \neq [])$;

let $bls = Collect\ (on\text{-}stack\text{-}less\ I\ (last\ B))$;

let $ls = Collect\ (on\text{-}stack\text{-}ge\ I\ (last\ B))$;

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* $s \leq \Downarrow (Id \times_r\ gGS\text{-rel})$

$(RETURN\ (Some\ (\bigcup (set\ (butlast\ p)), last\ p), a, p, D, pE))$

<proof>

definition *last-is-acc-impl* $s \equiv$

do {

let $(a, -) = s$;

ASSERT $(a \neq [])$;

RETURN $(\forall i < num\text{-}acc.\ i \in last\ a)$

}

lemma *last-is-acc-impl-refine*:

assumes *A*: $(s, (a, p, D, pE)) \in \text{goGS-rel}$

assumes *PRE*: $a \neq []$

shows $\text{last-is-acc-impl } s \leq \text{RETURN } (\text{last } a = \{0..<\text{num-acc}\})$

<proof>

definition *go-is-no-brk-impl* :: $'Q \text{ goGS} \Rightarrow \text{bool}$

where $\text{go-is-no-brk-impl } s \equiv \text{fst } s = \text{None}$

lemma *go-is-no-brk-refine*:

$(s, s') \in \text{goGS-rel} \Longrightarrow \text{go-is-no-brk-impl } s \longleftrightarrow \text{go-is-no-brk } s'$

<proof>

definition *goD-impl* :: $'Q \text{ goGS} \Rightarrow 'Q \text{ oGS}$ **where** $\text{goD-impl } s \equiv \text{snd } s$

lemma *goD-refine*:

$\text{go-is-no-brk } s' \Longrightarrow (s, s') \in \text{goGS-rel} \Longrightarrow (\text{goD-impl } s, \text{goD } s') \in \text{oGS-rel}$

<proof>

definition *go-is-done-impl* :: $'Q \Rightarrow 'Q \text{ goGS} \Rightarrow \text{bool}$

where $\text{go-is-done-impl } v \ s \equiv \text{is-done-oimpl } v \ (\text{snd } s)$

thm *is-done-orefine*

lemma *go-is-done-impl-refine*: $\llbracket \text{go-is-no-brk } s'; (s, s') \in \text{goGS-rel}; (v, v') \in \text{Id} \rrbracket$

$\Longrightarrow \text{go-is-done-impl } v \ s \longleftrightarrow (v' \in \text{goD } s')$

<proof>

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

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

<proof>

definition *find-ce-impl* :: $('Q \text{ set} \times 'Q \text{ set}) \text{ option nres}$ **where**

$\text{find-ce-impl} \equiv \text{do } \{$

$\text{stat-start-nres};$

$\text{let } os = \text{goinitial-impl};$

$os \leftarrow \text{FOREACHci } (\lambda it \ os. \text{fgl-outer-invar } it \ (\text{goGS-}\alpha \ os)) \ V0$

$(\text{go-is-no-brk-impl}) \ (\lambda v0 \ s0.$

$\text{do } \{$

$\text{if } \neg \text{go-is-done-impl } v0 \ s0 \ \text{then } \text{do } \{$

$\text{let } s = (\text{None}, \text{ginitial-impl } v0 \ s0);$

$(\text{brk}, s) \leftarrow \text{WHILEIT}$

$(\lambda (\text{brk}, s). \text{fgl-invar } v0 \ (\text{oGS-}\alpha \ (\text{goD-impl } s0)) \ (\text{brk}, \text{snd } (\text{gGS-}\alpha \ s)))$

$(\lambda (\text{brk}, s). \text{brk} = \text{None} \wedge \neg \text{gpath-is-empty-impl } s) \ (\lambda (l, s).$

$\text{do } \{$

$\text{— Select edge from end of path}$

$(v0, s) \leftarrow \text{gselect-edge-impl } s;$

```

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*: $find\text{-}ce\text{-}impl \leq \Downarrow Id\ gfind\text{-}ce$
<proof>

end

1.22 Constructing a Lasso from Counterexample

1.22.1 Lassos in GBAs

context *igb-fr-graph* begin

definition *reconstruct-reach* :: $'Q\ set \Rightarrow 'Q\ set \Rightarrow ('Q\ list \times 'Q)\ nres$
— Reconstruct the reaching path of a lasso
where *reconstruct-reach* $Vr\ Vl \equiv do \{$
 $res \leftarrow find\text{-}path\ (E \cap Vr \times UNIV)\ V0\ (\lambda v.\ v \in Vl);$
 $ASSERT\ (res \neq None);$
 $RETURN\ (the\ res)$
 $\}$

lemma reconstruct-reach-correct:
assumes *CEC*: *ce-correct Vr Vl*
shows *reconstruct-reach Vr Vl*
 \leq *SPEC* $(\lambda(pr,va). \exists v0 \in V0. \text{path } E \ v0 \ pr \ va \ \wedge \ va \in Vl)$
 \langle *proof* \rangle

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

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

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

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

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

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

$\text{RETURN } (pr, p@p\text{-close})$
 $\}$

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

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

```

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*: $find-lasso \leq find-lasso-spec$
<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  $\rightarrow$  (SML) Gabow'-Skeleton'-Statistics.newnode
| constant stat-start  $\rightarrow$  (SML) Gabow'-Skeleton'-Statistics.start
| constant stat-stop  $\rightarrow$  (SML) Gabow'-Skeleton'-Statistics.stop

```

1.25 Automatic Refinement Setup

```

consts i-node-state :: interface

```

definition $node\text{-}state\text{-}rel \equiv \{(-1::int,DONE)\} \cup \{(int\ k,STACK\ k) \mid k.\ True\}$

lemma $node\text{-}state\text{-}rel\text{-}simps[simp]$:

```

(i,DONE) ∈ node-state-rel  $\longleftrightarrow$  i = -1
(i,STACK n) ∈ node-state-rel  $\longleftrightarrow$  i = int n
⟨proof⟩

```

lemma $node\text{-}state\text{-}rel\text{-}sv[simp,intro!,relator\text{-}props]$:

```

single-valued node-state-rel
⟨proof⟩

```

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

primrec $is\text{-}DONE$ **where**

```

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

```

lemma $node\text{-}state\text{-}rel\text{-}refine[autoref\text{-}rules]$:

```

(-1,DONE) ∈ node-state-rel
(int,STACK) ∈ nat-rel  $\rightarrow$  node-state-rel
( $\lambda i. i < 0, is-DONE$ ) ∈ node-state-rel  $\rightarrow$  bool-rel
(( $\lambda f\ g\ i. if\ i \geq 0$  then f (nat i) else g), case-node-state)
  ∈ (nat-rel  $\rightarrow$  R)  $\rightarrow$  R  $\rightarrow$  node-state-rel  $\rightarrow$  R
⟨proof⟩

```

lemma $[autoref\text{-}op\text{-}pat]$:

```

(x=DONE)  $\equiv$  is-DONE x

```

$(DONE=x) \equiv is-DONE\ x$
 $\langle proof \rangle$

consts *i-node* :: *interface*

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

+

assumes *G-refine*: (*G-impl*, *G*) ∈ (*mrel*, *node-rel*) *g-impl-rel-ext*
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*)

begin

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

lemmas [*autoref-rules*] = *G-refine node-eq-refine*

lemmas [*autoref-ga-rules*] = *node-hash node-hash-def-size*

lemma *locale-this*: *fr-graph-impl-loc mrel node-rel node-eq-impl node-hash-impl*
node-def-hash-size G-impl G
 $\langle proof \rangle$

abbreviation *oGSi-rel* ≡ (*node-rel*, *node-state-rel*)(*ahm-rel node-hash-impl*)

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

lemmas [*autoref-op-pat*] = *GS.S-def GS.B-def GS.I-def GS.P-def*

end

1.26 Generating the Code

thm *autoref-ga-rules*

```

context fr-graph-impl-loc
begin
  schematic-goal push-code-aux:  $(?c, \text{push-impl}) \in \text{node-rel} \rightarrow \text{GSi-rel} \rightarrow \text{GSi-rel}$ 
     $\langle \text{proof} \rangle$ 
  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:  $(?c, \text{pop-impl}) \in \text{GSi-rel} \rightarrow \langle \text{GSi-rel} \rangle \text{nres-rel}$ 
     $\langle \text{proof} \rangle$ 
  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  $(?c, \text{GS.S-idx-of}) \in \text{GSi-rel} \rightarrow \text{node-rel} \rightarrow \text{nat-rel}$ 
     $\langle \text{proof} \rangle$ 
  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  $(?c, \text{GS.idx-of-impl}) \in \text{GSi-rel} \rightarrow \text{node-rel} \rightarrow \langle \text{nat-rel} \rangle \text{nres-rel}$ 
     $\langle \text{proof} \rangle$ 
  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:
     $(?c, \text{collapse-impl}) \in \text{node-rel} \rightarrow \text{GSi-rel} \rightarrow \langle \text{GSi-rel} \rangle \text{nres-rel}$ 
     $\langle \text{proof} \rangle$ 
  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:
     $(?c, \text{select-edge-impl})$ 
     $\in \text{GSi-rel} \rightarrow \langle \langle \text{node-rel} \rangle \text{option-rel} \times_r \text{GSi-rel} \rangle \text{nres-rel}$ 
     $\langle \text{proof} \rangle$ 
  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  $\langle \text{proof} \rangle$ 

  term fr-graph.pop-impl
  lemma [autoref-op-pat]:
    push-impl  $\equiv$  OP push-impl
    collapse-impl  $\equiv$  OP collapse-impl

```

select-edge-impl \equiv *OP select-edge-impl*
pop-impl \equiv *OP pop-impl*
 ⟨*proof*⟩

end

schematic-goal *skeleton-code-aux*:
 (?*c*, *skeleton-impl*) \in ⟨*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* \leq *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* \leq *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* \leq *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* \leq *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* \leq *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*:

$(?c, last-seg-impl) \in GSi-rel \rightarrow \langle \langle node-rel \rangle list-set-rel \rangle nres-rel$
 $\langle proof \rangle$

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* $\langle proof \rangle$

lemma [*autoref-op-pat*]:

$last-seg-impl \equiv OP last-seg-impl$

$\langle proof \rangle$

end

schematic-goal *compute-SCC-code-aux*:

$(?c, compute-SCC-impl) \in \langle \langle \langle node-rel \rangle list-set-rel \rangle list-rel \rangle nres-rel$
 $\langle proof \rangle$

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* $?c \leq last-seg-code s$

$\langle proof \rangle$

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

```

begin

1.32 Workset Algorithm

A simple implementation is by a workset algorithm.

definition *ws-update* $E\ u\ p\ V\ ws \equiv RETURN\ (\$
 $V \cup E^{\leftarrow}\{u\},$
 $ws\ ++\ (\lambda v. \text{if } (u,v) \in E \wedge v \notin V \text{ then } Some\ (u\#p) \text{ else } None))$

definition *s-init* $U0 \equiv RETURN\ (None, U0, \lambda u. \text{if } u \in U0 \text{ then } Some\ [] \text{ else } None)$

definition *wset-find-path* $E\ U0\ P \equiv do\ \{$
 $ASSERT\ (\text{finite } U0);$
 $s0 \leftarrow s\text{-init } U0;$
 $(res, -, -) \leftarrow WHILET$
 $(\lambda(res, V, ws). res = None \wedge ws \neq Map.empty)$
 $(\lambda(res, V, ws). do\ \{$
 $ASSERT\ (ws \neq Map.empty);$
 $(u, p) \leftarrow SPEC\ (\lambda(u, p). ws\ u = Some\ p);$
 $let\ ws = ws\ |'(-\{u\});$

 $\text{if } P\ u\ \text{then}$
 $RETURN\ (Some\ (rev\ p, u), V, ws)$
 $\text{else } do\ \{$
 $ASSERT\ (\text{finite } (E^{\leftarrow}\{u\}));$
 $ASSERT\ (dom\ ws \subseteq V);$
 $(V, ws) \leftarrow ws\text{-update } E\ u\ p\ V\ ws;$
 $RETURN\ (None, V, ws)$
 $\}$
 $\})\ s0;$
 $RETURN\ res$
 $\}$

lemma *wset-find-path-correct*:

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

shows $wset\text{-find-path } E\ U0\ P \leq find\text{-path } E\ U0\ P$

<proof>

We refine the algorithm to use a foreach-loop

definition *ws-update-foreach* $E\ u\ p\ V\ ws \equiv$
 $FOREACH\ (LIST\text{-SET-REV-TAG } (E^{\leftarrow}\{u\}))\ (\lambda v\ (V, ws).$
 $\text{if } v \in V\ \text{then}$
 $RETURN\ (V, ws)$
 $\text{else } do\ \{$
 $ASSERT\ (v \notin dom\ ws);$
 $RETURN\ (insert\ v\ V, ws\ (v \mapsto u\#p))$
 $\}$
 $)\ (V, ws)$

lemma *ws-update-foreach-refine*[refine]:

assumes *FIN*: finite ($E'\{u\}$)

assumes *WSS*: $\text{dom } ws \subseteq V$

assumes *ID*: $(E',E) \in Id \quad (u',u) \in Id \quad (p',p) \in Id \quad (V',V) \in Id \quad (ws',ws) \in Id$

shows *ws-update-foreach* $E' u' p' V' ws' \leq \Downarrow Id$ (*ws-update* $E u p V ws$)

$\langle \text{proof} \rangle$

definition *s-init-foreach* $U0 \equiv \text{do } \{$

$(U0,ws) \leftarrow \text{FOREACH } U0 (\lambda x (U0,ws).$

$\text{RETURN } (\text{insert } x \ U0,ws(x \rightarrow [])) (\{\}, \text{Map.empty});$

$\text{RETURN } (\text{None}, U0,ws)$

$\}$

lemma *s-init-foreach-refine*[refine]:

assumes *FIN*: finite $U0$

assumes *ID*: $(U0',U0) \in Id$

shows *s-init-foreach* $U0' \leq \Downarrow Id$ (*s-init* $U0$)

$\langle \text{proof} \rangle$

definition *wset-find-path'* $E U0 P \equiv \text{do } \{$

ASSERT (*finite* $U0$);

$s0 \leftarrow \text{s-init-foreach } U0$;

$(res, -, -) \leftarrow \text{WHILET}$

$(\lambda(res, V, ws). res = \text{None} \wedge ws \neq \text{Map.empty})$

$(\lambda(res, V, ws). \text{do } \{$

ASSERT ($ws \neq \text{Map.empty}$);

$((u, p), ws) \leftarrow \text{op-map-pick-remove } ws$;

if $P u$ *then*

$\text{RETURN } (\text{Some } (\text{rev } p, u), V, ws)$

else do $\{$

$(V, ws) \leftarrow \text{ws-update-foreach } E u p V ws$;

$\text{RETURN } (\text{None}, V, ws)$

$\}$

$\})$

$s0$;

$\text{RETURN } res$

$\}$

lemma *wset-find-path'-refine*:

wset-find-path' $E U0 P \leq \Downarrow Id$ (*wset-find-path* $E U0 P$)

$\langle \text{proof} \rangle$

1.33 Refinement to efficient data structures

schematic-goal *wset-find-path'-refine-aux*:

fixes $U0 :: 'a \text{ set}$ **and** $P :: 'a \Rightarrow \text{bool}$ **and** $E :: ('a \times 'a) \text{ set}$

and $P \text{impl} :: 'a i \Rightarrow \text{bool}$

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

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

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

concrete-definition $wset\text{-}find\text{-}path\text{-}impl$ **for** $node\text{-}eq\text{-}impl \ succ_i \ U0' \ Pimpl$
uses $wset\text{-}find\text{-}path'\text{-}refine\text{-}aux$

1.34 Autoref Setup

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

lemma $[autoref\text{-}itype]$:
 $find\text{-}path ::_i \ \langle I \rangle_i i\text{-}slg \ \rightarrow_i \ \langle I \rangle_i i\text{-}set \ \rightarrow_i \ (I \rightarrow_i i\text{-}bool)$
 $\rightarrow_i \ \langle \langle \langle I \rangle_i i\text{-}list, I \rangle_i i\text{-}prod \rangle_i i\text{-}option \rangle_i i\text{-}nres \ \langle proof \rangle$

lemma $wset\text{-}find\text{-}path\text{-}autoref[autoref\text{-}rules]$:

fixes $node\text{-}rel :: ('ai \times 'a) \text{ set}$

assumes $eq: GEN\text{-}OP \ node\text{-}eq\text{-}impl \ (=) \ (node\text{-}rel \rightarrow node\text{-}rel \rightarrow bool\text{-}rel)$

assumes $hash: SIDE\text{-}GEN\text{-}ALGO \ (is\text{-}bounded\text{-}hashcode \ node\text{-}rel \ node\text{-}eq\text{-}impl \ node\text{-}hash\text{-}impl)$

assumes $hash\text{-}dsz: SIDE\text{-}GEN\text{-}ALGO \ (is\text{-}valid\text{-}def\text{-}hm\text{-}size \ TYPE('ai) \ node\text{-}def\text{-}hash\text{-}size)$

shows (

$wset\text{-}find\text{-}path\text{-}impl \ node\text{-}hash\text{-}impl \ node\text{-}def\text{-}hash\text{-}size \ node\text{-}eq\text{-}impl,$
 $find\text{-}path)$

$\in \langle node\text{-}rel \rangle slg\text{-}rel \rightarrow \langle node\text{-}rel \rangle list\text{-}set\text{-}rel \rightarrow (node\text{-}rel \rightarrow bool\text{-}rel)$

$\rightarrow \langle \langle \langle node\text{-}rel \rangle list\text{-}rel \times_\tau node\text{-}rel \rangle option\text{-}rel \rangle nres\text{-}rel$

$\langle proof \rangle$

end

schematic-goal $wset\text{-}find\text{-}path\text{-}transfer\text{-}aux$:

$RETURN \ ?c \leq wset\text{-}find\text{-}path\text{-}impl \ hash_i \ dsz_i \ eq_i \ E \ U0 \ P$

$\langle proof \rangle$
concrete-definition *wset-find-path-code*
for $E \text{ ?}U0.0 P$ **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* $E u0 P \equiv do \{$
 $res \leftarrow find-path E (E'\{u0\}) P;$
 $case\ res\ of\ None \Rightarrow RETURN\ None$
 $| Some\ (p,v) \Rightarrow RETURN\ (Some\ (u0\#p,v))$
 $\}$

lemma *find-path1-gen-correct*: $find-path1-gen\ E\ u0\ P \leq find-path1\ E\ u0\ P$
 $\langle proof \rangle$

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

fixes $node-rel :: ('ai \times 'a)\ set$
assumes [*autoref-rules*]: $(node-eq-impl, (=)) \in node-rel \rightarrow node-rel \rightarrow bool-rel$
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 $(?c, find-path1-gen :: (- \times -)\ set \Rightarrow -) \in \langle node-rel \rangle slg-rel \rightarrow node-rel \rightarrow (node-rel \rightarrow bool-rel) \rightarrow \langle \langle \langle node-rel \rangle list-rel \times_r node-rel \rangle option-rel \rangle nres-rel$
 $\langle proof \rangle$

lemma [*autoref-itype*]:

$find-path1 ::_i \langle I \rangle_i i-slg \rightarrow_i I \rightarrow_i (I \rightarrow_i i-bool)$
 $\rightarrow_i \langle \langle \langle I \rangle_i i-list, I \rangle_i i-prod \rangle_i i-option \rangle_i i-nres \langle proof \rangle$

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

lemma *find-path1-autoref*[*autoref-rules*]:

fixes $node-rel :: ('ai \times 'a)\ set$
assumes *eq*: *GEN-OP* $node-eq-impl (=) (node-rel \rightarrow node-rel \rightarrow bool-rel)$
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 $(find-path1-impl\ node-eq-impl\ node-hash-impl\ node-def-hash-size, find-path1)$

$\in \langle node-rel \rangle slg-rel \rightarrow node-rel \rightarrow (node-rel \rightarrow bool-rel) \rightarrow$
 $\langle \langle \langle node-rel \rangle list-rel \times_r node-rel \rangle Relators.option-rel \rangle nres-rel$
 $\langle proof \rangle$

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-ecxt 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

```

$\langle proof \rangle$
end
abbreviation $gGSi-rel \equiv \langle \langle nat-rel \rangle bs-set-rel \rangle as-rel \times_r GSi-rel$
abbreviation $(in -) ce-rel node-rel \equiv \langle \langle node-rel \rangle fun-set-rel \times_r \langle node-rel \rangle fun-set-rel \rangle option-rel$
abbreviation $goGSi-rel \equiv ce-rel node-rel \times_r oGSi-rel$
end

1.38 Automatic Refinement

context $impl-lasso-loc$
begin

schematic-goal $goinitial-code-aux: (?c, goinitial-impl) \in goGSi-rel$
 $\langle proof \rangle$
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 $ginitial-code-aux:$
 $(?c, ginitial-impl) \in node-rel \rightarrow goGSi-rel \rightarrow gGSi-rel$
 $\langle proof \rangle$

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

schematic-goal $gpath-is-empty-code-aux:$
 $(?c, gpath-is-empty-impl) \in gGSi-rel \rightarrow bool-rel$
 $\langle proof \rangle$

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 $goBrk-code-aux: (?c, goBrk-impl) \in goGSi-rel \rightarrow ce-rel node-rel$
 $\langle proof \rangle$

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 $gto-outer-code-aux:$
 $(?c, gto-outer-impl) \in ce-rel node-rel \rightarrow gGSi-rel \rightarrow goGSi-rel$
 $\langle proof \rangle$

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*:
 $(?c, go-is-done-impl) \in node-rel \rightarrow goGSi-rel \rightarrow bool-rel$
 $\langle proof \rangle$
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*:
 $(?c, go-is-no-brk-impl) \in goGSi-rel \rightarrow bool-rel$
 $\langle proof \rangle$
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*: $(?c, gselect-edge-impl)$
 $\in gGSi-rel \rightarrow \langle \langle node-rel \rangle option-rel \times_r gGSi-rel \rangle nres-rel$
 $\langle proof \rangle$
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*:
 $(?c, gis-on-stack-impl) \in node-rel \rightarrow gGSi-rel \rightarrow bool-rel$
 $\langle proof \rangle$
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*: $(?c, gcollapse-impl) \in node-rel \rightarrow gGSi-rel$
 $\rightarrow \langle gGSi-rel \rangle nres-rel$
 $\langle proof \rangle$
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*:
 $(?c, last-is-acc-impl) \in gGSi-rel \rightarrow \langle bool-rel \rangle nres-rel$
 $\langle proof \rangle$
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*: $(?c, ce-impl)$
 $\in gGSi-rel \rightarrow \langle ce-rel node-rel \times_r gGSi-rel \rangle nres-rel$

$\langle proof \rangle$
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*:
 $(?c, gis-done-impl) \in node-rel \rightarrow gGSi-rel \rightarrow bool-rel$
 $\langle proof \rangle$
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*:
 $(?c, gpush-impl) \in node-rel \rightarrow gGSi-rel \rightarrow gGSi-rel$
 $\langle proof \rangle$
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*: $(?c, gpop-impl) \in gGSi-rel \rightarrow \langle gGSi-rel \rangle nres-rel$
 $\langle proof \rangle$
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*: $(?c, find-ce-impl) \in \langle ce-rel \ node-rel \rangle nres-rel$
 $\langle proof \rangle$
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 \leq find-ce-code node-eq-impl node-hash-impl*
node-def-hash-size G-impl
 $\langle proof \rangle$
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* $\langle proof \rangle$
lemma [*autoref-op-pat*]:
 $find-ce-spec \equiv OP \ find-ce-spec$
 $\langle proof \rangle$
end

theorem *find-ce-autoref*[*autoref-rules*]:
— Main Correctness theorem (inside locale)
shows $(find-ce-code \ node-eq-impl \ node-hash-impl \ node-def-hash-size \ G-impl,$
 $find-ce-spec) \in \langle ce-rel \ node-rel \rangle nres-rel$
 $\langle proof \rangle$

end

context *impl-lasso-loc*
begin

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

lemma [*autoref-op-pat*]:
 reconstruct-reach $\equiv OP$ *reconstruct-reach*
 reconstruct-lasso $\equiv OP$ *reconstruct-lasso*
 $\langle proof \rangle$

end

schematic-goal *reconstruct-reach-code-aux*:
 shows $(?c, reconstruct-reach) \in \langle node-rel \rangle fun-set-rel \rightarrow$
 $\langle node-rel \rangle fun-set-rel \rightarrow$
 $\langle \langle node-rel \rangle list-rel \times_r node-rel \rangle nres-rel$
 $\langle proof \rangle$

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 $(?c, reconstruct-lasso) \in \langle node-rel \rangle fun-set-rel \rightarrow$
 $\langle node-rel \rangle fun-set-rel \rightarrow$
 $\langle \langle node-rel \rangle list-rel \times_r \langle node-rel \rangle list-rel \rangle nres-rel$
 $\langle proof \rangle$

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*:
 $RETURN ?c \leq reconstruct-lasso-code$ *eqi hi dszi G-impl Vr Vl*
 $\langle proof \rangle$

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 $(?c::?'c, find-lasso) \in ?R$
 $\langle proof \rangle$

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*:
 $RETURN ?c \leq find-lasso-code$ *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 *ft-rel* :: $- \Rightarrow (- \times ('a \text{ list} \times 'a \text{ list}) \text{ option}) \text{ set}$ **where**
ft-rel node-rel $\equiv \langle \langle \text{node-rel} \rangle \text{list-rel} \times_r \langle \text{node-rel} \rangle \text{list-rel} \rangle \text{Relators.option-rel}$

theorem *find-lasso-tr-correct*:

— Correctness theorem for the constant we extracted to SML

fixes *Re* and *node-rel* :: $('vi \times 'v) \text{ set}$

assumes *A*: $(G\text{-impl}, G) \in \text{igbg-impl-rel-ext } Re \text{ node-rel}$

and *node-eq-refine*: $(\text{node-eq-impl}, (=)) \in \text{node-rel} \rightarrow \text{node-rel} \rightarrow \text{bool-rel}$

and *node-hash*: *is-bounded-hashcode node-rel node-eq-impl node-hash-impl*

and *node-hash-def-size*: $(\text{is-valid-def-hm-size } \text{TYPE}('vi) \text{ node-def-hash-size})$

assumes *B*: *igb-fr-graph G*

shows *RETURN* (*find-lasso-tr node-eq-impl node-hash-impl node-def-hash-size*
G-impl)

$\leq \Downarrow (\text{ft-rel } \text{node-rel}) (\text{igb-graph.find-lasso-spec } G)$

⟨*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* $\equiv \text{igb-graph.find-lasso-spec}$

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

lemma [*autoref-op-pat*]: *igb-graph.find-lasso-spec* $\equiv \text{op-find-lasso-spec}$
 ⟨*proof*⟩

term *op-find-lasso-spec*

lemma [*autoref-itype*]:

op-find-lasso-spec

$::_i \text{igbg } Ie \ I \rightarrow_i \langle \langle \langle I \rangle_i \text{i-list}, \langle I \rangle_i \text{i-list} \rangle_i \text{i-prod} \rangle_i \text{i-option} \rangle_i \text{i-nres}$

⟨*proof*⟩

lemma *find-lasso-spec-autoref*[*autoref-rules-raw*]:

fixes *Re* and *node-rel* :: $('vi \times 'v) \text{ set}$

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 directly 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.