Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm

Peter Lammich

March 17, 2025

Abstract

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

Contents

1 Introduction

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

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

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

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

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

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

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

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

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

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

1.1 Skeleton for Gabow's SCC Algorithm

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

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

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

1.2 Statistics Setup

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

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

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

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

lemma [*autoref-rules*]:
  (*stat-newnode*,*stat-newnode*) ∈ *unit-rel* → *unit-rel*
  (*stat-start*,*stat-start*) ∈ *unit-rel* → *unit-rel*
  (*stat-stop*,*stat-stop*) ∈ *unit-rel* → *unit-rel*
  by *auto*

[2] GBGs are generalized Büchi automata without labels.

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

lemma *discard-stat-refine*[*refine*]:
  $m1 \le m2 \Longrightarrow$ *stat-newnode-nres* $\gg$ $m1 \le m2$
  $m1 \le m2 \Longrightarrow$ *stat-start-nres* $\gg$ $m1 \le m2$
  $m1 \le m2 \Longrightarrow$ *stat-stop-nres* $\gg$ $m1 \le m2$
  by *simp-all*

1.3 Abstract Algorithm

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

1.3.1 Preliminaries

definition *path-seg* :: *'a set list* ⇒ *nat* ⇒ *nat* ⇒ *'a set*
  — Set of nodes in a segment of the path
  where *path-seg p i j* ≡ $\bigcup\{p!k|k.\ i \le k \wedge k < j\}$

lemma *path-seg-simps*[*simp*]:
  $j \le i \Longrightarrow$ *path-seg p i j* = {}
  path-seg p i (*Suc i*) = *p*!*i*
  unfolding *path-seg-def*
  apply *auto* []
  apply (*auto simp*: *le-less-Suc-eq*) []
  done

lemma *path-seg-drop*:
  $\bigcup$(*set* (*drop i p*)) = *path-seg p i* (*length p*)
  unfolding *path-seg-def*
  by (*fastforce simp*: *in-set-drop-conv-nth Bex-def*)

lemma *path-seg-butlast*:
  $p \ne [] \Longrightarrow$ *path-seg p 0* (*length p* − *Suc 0*) = $\bigcup$(*set* (*butlast p*))
  apply (*cases p rule*: *rev-cases*, *simp*)
  apply (*fastforce simp*: *path-seg-def nth-append in-set-conv-nth*)
  done

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

lemma *idx-of-props*:
  assumes
    p-disjoint-sym: ∀ *i j v*. *i*<*length p* ∧ *j*<*length p* ∧ *v*∈*p*!*i* ∧ *v*∈*p*!*j* ⟶ *i*=*j*
  assumes *ON-STACK*: *v*∈$\bigcup$(*set p*)

shows
idx-of p v $<$ *length p* **and**
$v \in p$! *idx-of p v*
proof $-$
from *ON-STACK* **obtain** *i* **where** $i<$*length p* $v \in p$! *i*
by (*auto simp add*: *in-set-conv-nth*)
moreover hence $\forall j<$*length p*. $v \in p$! $j \longrightarrow i=j$
using *p-disjoint-sym* **by** *auto*
ultimately show *idx-of p v* $<$ *length p*
and $v \in p$! *idx-of p v* **unfolding** *idx-of-def*
by (*metis* (*lifting*) *theI'*)+
qed

lemma *idx-of-uniq*:
assumes
p-disjoint-sym: $\forall i\ j\ v.\ i<$*length p* $\wedge$ $j<$*length p* $\wedge$ $v \in p!i$ $\wedge$ $v \in p!j \longrightarrow i=j$
assumes *A*: $i<$*length p* $v \in p!i$
shows *idx-of p v* $= i$
proof $-$
from *A p-disjoint-sym* **have** $\forall j<$*length p*. $v \in p$! $j \longrightarrow i=j$ **by** *auto*
with *A* **show** *?thesis*
unfolding *idx-of-def*
by (*metis* (*lifting*) *the-equality*)
qed

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* $\equiv$ $D \cup \bigcup$(*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* $\equiv$ ($E \cap$ (*touched p D* $\times$ *UNIV*)) $-$ *pE*

lemma *vE-ss-E*: *vE p D pE* $\subseteq E$ — Visited edges are edges
unfolding *vE-def* **by** *auto*

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* **by** *unfold-locales*

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

lemma *outer-invar-this*[*simp*, *intro*!]: *outer-invar it D*
unfolding *outer-invar-def* **apply** *simp* **by** *unfold-locales*
end

locale *invar-loc* — Invariant of the inner loop
= *fr-graph G*
for G :: $('v, 'more)$ *graph-rec-scheme* +
fixes $v0$ :: $'v$
fixes $D0$ :: $'v$ *set*
fixes p :: $'v$ *set list*
fixes D :: $'v$ *set*
fixes pE :: $('v \times 'v)$ *set*

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

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

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

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

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

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

lemma *locale-this*: *invar-loc G v0 D0 p D pE* **by** *unfold-locales*

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

lemma *invar-this*[*simp*, *intro*!]: *invar v0 D0* (*p*,*D*,*pE*)
unfolding *invar-def* **apply** *simp* **by** *unfold-locales*

lemma *finite-reachableE-v0*[*simp*, *intro*!]: *finite* $(E^* \text{``}\{v0\})$
apply (*rule finite-subset*[*OF - finite-reachableE-V0*])
using *v0-initial* **by** *auto*

lemma *D-vis*: $E \cap D \times UNIV \subseteq lvE$ — All edges from done nodes are visited
unfolding *vE-def touched-def* **using** *pE-E-from-p p-not-D* **by** *blast*

lemma *vE-touched*: $lvE \subseteq ltouched \times ltouched$
— Visited edges only between touched nodes
using *E-from-p-touched D-closed* **unfolding** *vE-def touched-def* **by** *blast*

lemma *lvE-ss-E*: $lvE \subseteq E$ — Visited edges are edges
unfolding *vE-def* **by** *auto*

lemma *path-touched*: $\bigcup(set\ p) \subseteq ltouched$ **by** (*auto simp*: *touched-def*)
lemma *D-touched*: $D \subseteq ltouched$ **by** (*auto simp*: *touched-def*)

lemma *pE-by-vE*: $pE = (E \cap \bigcup(set\ p) \times UNIV) - lvE$
— Pending edges are edges from path not yet visited
unfolding *vE-def touched-def*
using *pE-E-from-p*
by *auto*

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

lemma *p-connected'*:
assumes *A*: *Suc i<length p*
shows $p!i \times p!Suc\ i \cap lvE \neq \{\}$
proof −
from *A p-not-D* **have** $p!i \in set\ p \quad p!Suc\ i \in set\ p$ **by** *auto*

with *p-connected*[*OF A*] **show** *?thesis* **unfolding** *vE-def touched-def*
by *blast*
qed

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*)
$(\lambda(p,D,pE).\ (unproc\text{-}edges\ v0\ p\ D\ pE,\ p))$

lemma *abs-wf-rel-wf*[*simp*, *intro*!]: *wf* (*abs-wf-rel v0*)
unfolding *abs-wf-rel-def*
by *auto*
end

1.3.3 Abstract Skeleton Algorithm

context *fr-graph*
begin

definition (**in** *fr-graph*) *initial* :: $'v \Rightarrow 'v\ set \Rightarrow 'v\ abs\text{-}state$
where *initial v0 D* $\equiv$ $([\{v0\}],\ D,\ (E \cap \{v0\} \times UNIV))$

definition (**in** −) *collapse-aux* :: $'a\ set\ list \Rightarrow nat \Rightarrow 'a\ set\ list$
where *collapse-aux p i* $\equiv$ *take i p* @ $[\bigcup(set\ (drop\ i\ p))]$

definition (**in** −) *collapse* :: $'a \Rightarrow 'a\ abs\text{-}state \Rightarrow 'a\ abs\text{-}state$
where *collapse v PDPE* $\equiv$
let
$(p,D,pE)=PDPE$;
$i=idx\text{-}of\ p\ v$;
$p = collapse\text{-}aux\ p\ i$
in (p,D,pE)

definition (**in** −)
select-edge :: $'a\ abs\text{-}state \Rightarrow ('a\ option \times 'a\ abs\text{-}state)\ nres$
where
select-edge PDPE $\equiv$ *do* {
let $(p,D,pE) = PDPE$;
$e \leftarrow SELECT\ (\lambda e.\ e \in pE \cap last\ p \times UNIV)$;

    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* (λ*e*. *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*)}))
  }
  unfolding *select-edge-def* **by** *simp*

lemma *collapse v* (*p*,*D*,*pE*)
  ≡ *let i*=*idx-of p v in* (*take i p* @ [⋃(*set* (*drop i p*))],*D*,*pE*)
  unfolding *collapse-def collapse-aux-def* **by** *simp*

lemma *push v* (*p*, *D*, *pE*) ≡ (*p* @ [{*v*}], *D*, *pE* ∪ *E* ∩ {*v*} × *UNIV*)
  unfolding *push-def* **by** *simp*

lemma *pop* (*p*, *D*, *pE*) ≡ (*butlast p*, *last p* ∪ *D*, *pE*)
  unfolding *pop-def* **by** *auto*

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

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

definition *skeleton* :: *'v set nres*
  — Abstract Skeleton Algorithm
  where

```
skeleton ≡ do {
  let D = {};
  r ← FOREACHi outer-invar V0 (λv0 D0. do {
    if v0∉D0 then do {
      let s = initial v0 D0;

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

        ASSERT (p≠[]);
        case vo of
          Some v ⇒ do { — Found outgoing edge to node v
            if v ∈ ⋃(set p) then do {
              — Back edge: Collapse path
              RETURN (collapse v (p,D,pE))
            } else if v∉D then do {
              — Edge to new node. Append to path
              RETURN (push v (p,D,pE))
            } else do {
              — Edge to done node. Skip
              RETURN (p,D,pE)
            }
          }
        | None ⇒ do {
            ASSERT (pE ∩ last p × UNIV = {});
            — No more outgoing edges from current node on path
            RETURN (pop (p,D,pE))
          }
      }) s;
      ASSERT (p=[] ∧ pE={});
      RETURN D
    } else
      RETURN D0
  }) D;
  RETURN r
}
```

end

1.3.4 Invariant Preservation

context *fr-graph* **begin**

lemma *set-collapse-aux*[*simp*]: $\bigcup$(*set* (*collapse-aux p i*)) = $\bigcup$(*set p*)
  apply (*subst* (*2*) *append-take-drop-id*[*of - p*,*symmetric*])
  apply (*simp del*: *append-take-drop-id*)

unfolding *collapse-aux-def* **by** *auto*

lemma *touched-collapse*[*simp*]: *touched* (*collapse-aux p i*) *D* = *touched p D*
unfolding *touched-def* **by** *simp*

lemma *vE-collapse-aux*[*simp*]: *vE* (*collapse-aux p i*) *D pE* = *vE p D pE*
unfolding *vE-def* **by** *simp*

lemma *touched-push*[*simp*]: *touched* (*p* @ [*V*]) *D* = *touched p D* ∪ *V*
unfolding *touched-def* **by** *auto*

end

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

context *invar-loc*
begin
lemma *cnode-connectedI*:
⟦*i*<*length p*; *u*∈*p*!*i*; *v*∈*p*!*i*⟧ ⟹ (*u*,*v*)∈(*lvE* ∩ *p*!*i*×*p*!*i*)*
using *p-sc*[*of p*!*i*] **by** (*auto simp*: *in-set-conv-nth*)

lemma *cnode-connectedI'*: ⟦*i*<*length p*; *u*∈*p*!*i*; *v*∈*p*!*i*⟧ ⟹ (*u*,*v*)∈(*lvE*)*
by (*metis inf.cobounded1 rtrancl-mono-mp cnode-connectedI*)

lemma *p-no-empty*: {} ∉ *set p*
proof
assume {}∈*set p*
then obtain *i* **where** *IDX*: *i*<*length p* *p*!*i*={}
by (*auto simp add*: *in-set-conv-nth*)
show *False* **proof** (*cases i*)
case *0* **with** *root-v0 IDX* **show** *False* **by** (*cases p*) *auto*
next
case [*simp*]: (*Suc j*)
from *p-connected'*[*of j*] *IDX* **show** *False* **by** *simp*
qed
qed

corollary *p-no-empty-idx*: *i*<*length p* ⟹ *p*!*i*≠{}
using *p-no-empty* **by** (*metis nth-mem*)

lemma *p-disjoint-sym*: ⟦*i*<*length p*; *j*<*length p*; *v*∈*p*!*i*; *v*∈*p*!*j*⟧ ⟹ *i*=*j*
by (*metis disjoint-iff-not-equal linorder-neqE-nat p-disjoint*)

lemma *pi-ss-path-seg-eq*[*simp*]:
assumes *A*: *i*<*length p* *u*≤*length p*
shows *p*!*i*⊆*path-seg p l u* ⟷ *l*≤*i* ∧ *i*<*u*
proof
assume *B*: *p*!*i*⊆*path-seg p l u*

from *A* **obtain** *x* **where** *x*∈*p*!*i* **by** (*blast dest*: *p-no-empty-idx*)
with *B* **obtain** *i'* **where** *C*: *x*∈*p*!*i'* *l*≤*i'* *i'*<*u*
  by (*auto simp*: *path-seg-def*)
from *p-disjoint-sym*[*OF* ‹*i*<*length p*› - ‹*x*∈*p*!*i*› ‹*x*∈*p*!*i'*›] ‹*i'*<*u*› ‹*u*≤*length p*›
have *i*=*i'* **by** *simp*
with *C* **show** *l*≤*i* ∧ *i*<*u* **by** *auto*
qed (*auto simp*: *path-seg-def*)

lemma *path-seg-ss-eq*[*simp*]:
  assumes *A*: *l1*<*u1* *u1*≤*length p* *l2*<*u2* *u2*≤*length p*
  shows *path-seg p l1 u1* ⊆ *path-seg p l2 u2* ⟷ *l2*≤*l1* ∧ *u1*≤*u2*
proof
  assume *S*: *path-seg p l1 u1* ⊆ *path-seg p l2 u2*
  have *p*!*l1* ⊆ *path-seg p l1 u1* **using** *A* **by** *simp*
  also note *S* **finally have** *1*: *l2*≤*l1* **using** *A* **by** *simp*
  have *p*!(*u1* − *1*) ⊆ *path-seg p l1 u1* **using** *A* **by** *simp*
  also note *S* **finally have** *2*: *u1*≤*u2* **using** *A* **by** *auto*
  from *1 2* **show** *l2*≤*l1* ∧ *u1*≤*u2* **..**
next
  assume *l2*≤*l1* ∧ *u1*≤*u2* **thus** *path-seg p l1 u1* ⊆ *path-seg p l2 u2*
    using *A*
    apply (*clarsimp simp*: *path-seg-def*) []
    apply (*metis dual-order.strict-trans1 dual-order.trans*)
    done
qed

lemma *pathI*:
  assumes *x*∈*p*!*i* *y*∈*p*!*j*
  assumes *i*≤*j* *j*<*length p*
  defines *seg* ≡ *path-seg p i* (*Suc j*)
  shows (*x*,*y*)∈(*lvE* ∩ *seg*×*seg*)*
  — We can obtain a path between cnodes on path
  using *assms*(*3*,*1*,*2*,*4*) **unfolding** *seg-def*
proof (*induction arbitrary*: *y rule*: *dec-induct*)
  case *base* **thus** *?case* **by** (*auto intro*!: *cnode-connectedI*)
next
  case (*step j*)

  let *?seg* = *path-seg p i* (*Suc j*)
  let *?seg'* = *path-seg p i* (*Suc* (*Suc j*))

  have *SSS*: *?seg* ⊆ *?seg'*
    apply (*subst path-seg-ss-eq*)
    using *step.hyps step.prems* **by** *auto*

  from *p-connected'*[*OF* ‹*Suc j* < *length p*›] **obtain** *u v* **where**
    UV: (*u*,*v*)∈*lvE* *u*∈*p*!*j* *v*∈*p*!*Suc j* **by** *auto*

  have *ISS*: *p*!*j* ⊆ *?seg'* *p*!*Suc j* ⊆ *?seg'*

using *step.hyps step.prems* **by** *simp-all*

from *p-no-empty-idx*[*of j*] ‹*Suc j* < *length p*› **obtain** x' **where** x'∈*p*!*j*
by *auto*
with *step.IH*[*of x'*] ‹*x*∈*p*!*i*› ‹*Suc j* < *length p*›
have *t*: $(x,x')\in(lvE\cap ?seg\times ?seg)^*$ **by** *auto*
have $(x,x')\in(lvE\cap ?seg'\times ?seg')^*$ **using** *SSS*
by (*auto intro*: *rtrancl-mono-mp*[*OF - t*])
also
from *cnode-connectedI*[*OF -* ‹x'∈*p*!*j*› ‹*u*∈*p*!*j*›] ‹*Suc j* < *length p*› **have**
t: $(x', u) \in (lvE \cap p\ !\ j \times p\ !\ j)^*$ **by** *auto*
have $(x', u) \in (lvE\cap ?seg'\times ?seg')^*$ **using** *ISS*
by (*auto intro*: *rtrancl-mono-mp*[*OF - t*])
also have $(u,v)\in lvE\cap ?seg'\times ?seg'$ **using** *UV ISS* **by** *auto*
also from *cnode-connectedI*[*OF* ‹*Suc j* < *length p*› ‹*v*∈*p*!*Suc j*› ‹*y*∈*p*!*Suc j*›]
have *t*: $(v, y) \in (lvE \cap p\ !\ Suc\ j \times p\ !\ Suc\ j)^*$ **by** *auto*
have $(v, y) \in (lvE\cap ?seg'\times ?seg')^*$ **using** *ISS*
by (*auto intro*: *rtrancl-mono-mp*[*OF - t*])
finally show $(x,y)\in(lvE\cap ?seg'\times ?seg')^*$ **.**
qed

lemma *p-reachable*: $\bigcup(set\ p) \subseteq E^*$ ‘‘{*v0*} — Nodes on path are reachable
proof
fix *v*
assume *A*: $v\in\bigcup(set\ p)$
then obtain *i* **where** *i*<*length p* **and** *v*∈*p*!*i*
by (*metis UnionE in-set-conv-nth*)
moreover from *A root-v0* **have** *v0*∈*p*!*0* **by** (*cases p*) *auto*
ultimately have
t: $(v0,v)\in(lvE \cap path\text{-}seg\ p\ 0\ (Suc\ i) \times path\text{-}seg\ p\ 0\ (Suc\ i))^*$
by (*auto intro*: *pathI*)
from *lvE-ss-E* **have** $(v0,v)\in E^*$ **by** (*auto intro*: *rtrancl-mono-mp*[*OF - t*])
thus $v\in E^*$ ‘‘{*v0*} **by** *auto*
qed

lemma *touched-reachable*: *ltouched* ⊆ E^* ‘‘*V0* — Touched nodes are reachable
unfolding *touched-def* **using** *p-reachable D-reachable* **by** *blast*

lemma *vE-reachable*: *lvE* ⊆ E^* ‘‘*V0* × E^* ‘‘*V0*
apply (*rule order-trans*[*OF vE-touched*])
using *touched-reachable* **by** *blast*

lemma *pE-reachable*: *pE* ⊆ E^* ‘‘{*v0*} × E^* ‘‘{*v0*}
proof *safe*
fix *u v*
assume *E*: $(u,v)\in pE$
with *pE-E-from-p p-reachable* **have** $(v0,u)\in E^*$ $(u,v)\in E$ **by** *blast+*
thus $(v0,u)\in E^*$ $(v0,v)\in E^*$ **by** *auto*
qed

lemma *D-closed-vE-rtrancl*: lvE^* ‘‘$D \subseteq D$
  by (*metis D-closed Image-closed-trancl eq-iff reachable-mono lvE-ss-E*)

lemma *D-closed-path*: $[\![path\ E\ u\ q\ w;\ u{\in}D]\!] \Longrightarrow set\ q \subseteq D$
proof −
  assume *a1*: *path E u q w*
  assume $u \in D$
  hence *f1*: $\{u\} \subseteq D$
    using *bot.extremum* **by** *force*
  have $set\ q \subseteq E^*$ ‘‘ $\{u\}$
    using *a1* **by** (*metis insert-subset path-nodes-reachable*)
  thus $set\ q \subseteq D$
    using *f1* **by** (*metis D-closed rtrancl-reachable-induct subset-trans*)
qed

lemma *D-closed-path-vE*: $[\![path\ lvE\ u\ q\ w;\ u{\in}D]\!] \Longrightarrow set\ q \subseteq D$
  by (*metis D-closed-path path-mono lvE-ss-E*)

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

  using *P ND*
proof (*induction*)
  case (*path-prepend u v l w*)
  from ‹$(u,v){\in}lvE$› *vE-touched* **have** $v{\in}ltouched$ **by** *auto*
  hence $v{\in}\bigcup(set\ p)$
    unfolding *touched-def*
  proof
    assume $v{\in}D$
    moreover from ‹*path lvE v l w*› **have** $(v,w){\in}lvE^*$ **by** (*rule path-is-rtrancl*)
    ultimately have $w{\in}D$ **using** *D-closed-vE-rtrancl* **by** *auto*
    with ‹$w{\in}last\ p$› *p-not-D* **have** *False*
      by (*metis IntI Misc.last-in-set Sup-inf-eq-bot-iff assms*(*2*)
        bex-empty path-prepend.hyps(*2*))
    thus *?thesis* **..**
  qed
  then obtain *i* **where** $i{<}length\ p$ $v{\in}p!i$
    by (*metis UnionE in-set-conv-nth*)
  have $i{=}length\ p - 1$
  proof (*rule ccontr*)
    assume $i{\neq}length\ p - 1$
    with ‹$i{<}length\ p$› **have** $i < length\ p - 1$ **by** *simp*
    with *vE-no-back*[*of i length p* − *1*] ‹$i{<}length\ p$›
    have $lvE \cap last\ p \times p!i = \{\}$

```
      by (simp add: last-conv-nth)
    with ‹(u,v)∈lvE› ‹u∈last p› ‹v∈p!i› show False by auto
  qed
  with ‹v∈p!i› have v∈last p by (simp add: last-conv-nth)
  with path-prepend.IH ‹w∈last p› ‹u∈last p› show ?case by auto
qed simp

lemma loop-in-lastnode:
  assumes P: path lvE u q u
  assumes [simp]: p≠[]
  assumes ND: set q ∩ last p ≠ {}
  shows u∈last p and set q ⊆ last p
  — A loop that touches the last node is completely inside the last node
proof –
  from ND obtain v where v∈set q   v∈last p by auto
  then obtain q1 q2 where [simp]: q=q1@v#q2
    by (auto simp: in-set-conv-decomp)
  from P have path lvE v (v#q2@q1) v
    by (auto simp: path-conc-conv path-cons-conv)
  from path-in-lastnode[OF this ‹p≠[]› ‹v∈last p› ‹v∈last p›]
  show set q ⊆ last p by simp
  from P show u∈last p
    apply (cases q, simp)

    apply simp
    using ‹set q ⊆ last p›
    apply (auto simp: path-cons-conv)
    done
qed


lemma no-D-p-edges: E ∩ D × ⋃(set p) = {}
  using D-closed p-not-D by auto

lemma idx-of-props:
  assumes ON-STACK: v∈⋃(set p)
  shows
    idx-of p v < length p and
    v ∈ p ! idx-of p v
  using idx-of-props[OF - assms] p-disjoint-sym by blast+

end
```

Auxiliary Lemmas Regarding the Operations **lemma** (**in** *fr-graph*)
vE-initial[*simp*]: *vE* [{*v0*}] {} (*E* ∩ {*v0*} × *UNIV*) = {}
 unfolding *vE-def touched-def* **by** *auto*

context *invar-loc*
begin

lemma *vE-push*: $\llbracket$ $(u,v) \in pE$; $u \in last\ p$; $v \notin \bigcup(set\ p)$; $v \notin D$ $\rrbracket$
$\Longrightarrow vE\ (p\ @\ [\{v\}])\ D\ ((pE - \{(u,v)\}) \cup E \cap \{v\} \times UNIV) = insert\ (u,v)\ lvE$
unfolding *vE-def touched-def* **using** *pE-E-from-p*
by *auto*

lemma *vE-remove*[*simp*]:
$\llbracket p \neq []$; $(u,v) \in pE \rrbracket \Longrightarrow vE\ p\ D\ (pE - \{(u,v)\}) = insert\ (u,v)\ lvE$
unfolding *vE-def touched-def* **using** *pE-E-from-p* **by** *blast*

lemma *vE-pop*[*simp*]: $p \neq [] \Longrightarrow vE\ (butlast\ p)\ (last\ p \cup D)\ pE = lvE$
unfolding *vE-def touched-def*
by (*cases p rule*: *rev-cases*) *auto*

lemma *pE-fin*: $p = [] \Longrightarrow pE = \{\}$
using *pE-by-vE* **by** *auto*

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
proof (*intro subsetI*, *elim ImageE*)
from *NO′* **have** *NO*: $(E - lvE) \cap (last\ p \times UNIV) = \{\}$
by (*simp add*: *pick-pending*[*OF NE*])

let $?i = length\ p - 1$
from *NE* **have** [*simp*]: $last\ p = p!?i$ **by** (*metis last-conv-nth*)

fix *u v*
assume *E*: $(u,v) \in E$
assume *UI*: $u \in last\ p \cup D$ **hence** $u \in p!?i \cup D$ **by** *simp*

{
assume $u \in last\ p$ $\quad v \notin last\ p$
moreover from *E NO* ‹$u \in last\ p$› **have** $(u,v) \in lvE$ **by** *auto*
ultimately have $v \in D \lor v \in \bigcup(set\ p)$
using *vE-touched* **unfolding** *touched-def* **by** *auto*
moreover {
assume $v \in \bigcup(set\ p)$
then obtain *j* **where** *V*: $j < length\ p$ $\quad v \in p!j$
by (*metis UnionE in-set-conv-nth*)
with ‹$v \notin last\ p$› **have** $j < ?i$ **by** (*cases* $j = ?i$) *auto*
from *vE-no-back*[*OF* ‹$j < ?i$› -] ‹$(u,v) \in lvE$› *V* ‹$u \in last\ p$› **have** *False* **by** *auto*
} ultimately have $v \in D$ **by** *blast*
} with *E UI D-closed* **show** $v \in last\ p \cup D$ **by** *auto*
qed

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*)
unfolding *invar-def initial-def*
apply *simp*
apply *unfold-locales*
apply *simp-all*
using *assms it-initial* **apply** *auto* []
using *D-reachable it-initial assms* **apply** *auto* []
using *D-closed* **apply** *auto* []
using *assms* **apply** *auto* []
done

lemma *invar-initial*:
$[\![$*outer-invar it D0*; $v0 \in it$; $v0 \notin D0]\!] \Longrightarrow$ *invar v0 D0* (*initial v0 D0*)
unfolding *outer-invar-def*
apply (*drule outer-invar-loc.invar-initial-aux*)
by *auto*

lemma *outer-invar-initial*[*simp*, *intro*!]: *outer-invar V0* {}
unfolding *outer-invar-def*
apply *unfold-locales*
by *auto*

lemma *invar-pop*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: $p \neq []$
assumes *NO'*: $pE \cap (last\ p \times UNIV) = \{\}$
shows *invar v0 D0* (*pop* (*p*,*D*,*pE*))
unfolding *invar-def pop-def*
apply *simp*
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

have [*simp*]: *set p* = *insert* (*last p*) (*set* (*butlast p*))
using *NE* **by** (*cases p rule*: *rev-cases*) *auto*

from *p-disjoint* **have** *lp-dj-blp*: $last\ p \cap \bigcup(set\ (butlast\ p)) = \{\}$
apply (*cases p rule*: *rev-cases*)
apply *simp*
apply (*fastforce simp*: *in-set-conv-nth nth-append*)
done

{

fix *i*
assume *A*: *Suc i* < *length* (*butlast p*)
hence *A'*: *Suc i* < *length p* **by** *auto*

from *nth-butlast*[*of i p*] *A* **have** [*simp*]: *butlast p* ! *i* = *p* ! *i* **by** *auto*
from *nth-butlast*[*of Suc i p*] *A*
have [*simp*]: *butlast p* ! *Suc i* = *p* ! *Suc i* **by** *auto*

from *p-connected*[*OF A'*]
have *butlast p* ! *i* × *butlast p* ! *Suc i* ∩ (*E* − *pE*) ≠ {}
by *simp*
} **note** *AUX-p-connected* = *this*

show *invar-loc G v0 D0* (*butlast p*) (*last p* ∪ *D*) *pE*
apply *unfold-locales*

unfolding *vE-pop*[*OF NE*]

apply *simp*

using *D-incr* **apply** *auto* []

using *pE-E-from-p NO'* **apply** *auto* []

using *E-from-p-touched* **apply** (*auto simp*: *touched-def*) []

using *D-reachable p-reachable NE* **apply** *auto* []

apply (*rule AUX-p-connected*, *assumption*+) []

using *p-disjoint* **apply** (*simp add*: *nth-butlast*)

using *p-sc* **apply** *simp*

using *root-v0* **apply** (*cases p rule*: *rev-cases*) **apply** *auto* [*2*]

using *root-v0 p-empty-v0* **apply** (*cases p rule*: *rev-cases*) **apply** *auto* [*2*]

apply (*rule lastp-un-D-closed*, *insert NO'*, *auto*) []

using *vE-no-back* **apply** (*auto simp*: *nth-butlast*) []

using *p-not-D lp-dj-blp* **apply** *auto* []
done
qed

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

lemma *invar-collapse*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* **and** *u*∈*last p*
assumes *BACK*: *v*∈⋃(*set p*)
defines *i* ≡ *idx-of p v*
defines *p'* ≡ *collapse-aux p i*
shows *invar v0 D0* (*collapse v* (*p*,*D*,*pE* − {(*u*,*v*)}))
unfolding *invar-def collapse-def*
apply *simp*
unfolding *i-def*[*symmetric*] *p'-def*[*symmetric*]
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

let *?thesis*=*invar-loc G v0 D0 p' D* (*pE* − {(*u*,*v*)})

have *SETP'*[*simp*]: ⋃(*set p'*) = ⋃(*set p*) **unfolding** *p'-def* **by** *simp*

have *IL*: *i* < *length p* **and** *VMEM*: *v*∈*p*!*i*
using *idx-of-props*[*OF BACK*] **unfolding** *i-def* **by** *auto*

have [*simp*]: *length p'* = *Suc i*
unfolding *p'-def collapse-aux-def* **using** *IL* **by** *auto*

have *P'-IDX-SS*: ∀ *j*<*Suc i*. *p*!*j* ⊆ *p'*!*j*
unfolding *p'-def collapse-aux-def* **using** *IL*
by (*auto simp add*: *nth-append path-seg-drop*)

from ‹*u*∈*last p*› **have** *u*∈*p*!(*length p* − *1*) **by** (*auto simp*: *last-conv-nth*)

have *defs-fold*:
vE p' D (*pE* − {(*u*,*v*)}) = *insert* (*u*,*v*) *lvE*
touched p' D = *ltouched*
by (*simp-all add*: *p'-def E*)

{
fix *j*
assume *A*: *Suc j* < *length p'*
hence *Suc j* < *length p* **using** *IL* **by** *simp*
from *p-connected*[*OF this*] **have** *p*!*j* × *p*!*Suc j* ∩ (*E*−*pE*) ≠ {} **.**
moreover from *P'-IDX-SS A* **have** *p*!*j*⊆*p'*!*j* **and** *p*!*Suc j* ⊆ *p'*!*Suc j*
by *auto*
ultimately have *p'* ! *j* × *p'* ! *Suc j* ∩ (*E* − (*pE* − {(*u*, *v*)})) ≠ {}
by *blast*
} **note** *AUX-p-connected* = *this*

have *P-IDX-EQ*[*simp*]: ∀ *j*. *j* < *i* ⟶ *p'*!*j* = *p*!*j*
unfolding *p'-def collapse-aux-def* **using** *IL*

by (*auto simp*: *nth-append*)

have P'-*LAST*[*simp*]: $p'!i$ = *path-seg p i* (*length p*) (**is** - = *?last-cnode*)
unfolding p'-*def* *collapse-aux-def* **using** *IL*
by (*auto simp*: *nth-append path-seg-drop*)

{
fix *j k*
assume *A*: $j < k$ $k <$ *length* p'
have $p' ! j \cap p' ! k = \{\}$
proof (*safe*, *simp*)
fix *v*
assume $v \in p'!j$ **and** $v \in p'!k$
with *A* **have** $v \in p!j$ **by** *simp*
show *False* **proof** (*cases*)
assume $k=i$
with ‹$v \in p'!k$› **obtain** k' **where** $v \in p!k'$ $i \le k'$ $k' <$ *length p*
by (*auto simp*: *path-seg-def*)
hence $p ! j \cap p ! k' = \{\}$
using *A* **by** (*auto intro*!: *p-disjoint*)
with ‹$v \in p!j$› ‹$v \in p!k'$› **show** *False* **by** *auto*
next
assume $k \neq i$ **with** *A* **have** $k<i$ **by** *simp*
hence $k<$*length p* **using** *IL* **by** *simp*
note *p-disjoint*[*OF* ‹$j<k$› *this*]
also have $p!j = p'!j$ **using** ‹$j<k$› ‹$k<i$› **by** *simp*
also have $p!k = p'!k$ **using** ‹$k<i$› **by** *simp*
finally show *False* **using** ‹$v \in p'!j$› ‹$v \in p'!k$› **by** *auto*
qed
qed
} **note** *AUX-p-disjoint* = *this*

{
fix *U*
assume *A*: $U \in$ *set* p'
then obtain *j* **where** $j<$*Suc i* **and** [*simp*]: $U=p'!j$
by (*auto simp*: *in-set-conv-nth*)
hence $U \times U \subseteq$ (*insert* (*u*, *v*) *lvE* $\cap$ $U \times U$)*
proof *cases*
assume [*simp*]: $j=i$
show *?thesis* **proof** (*clarsimp*)
fix *x y*
assume $x \in$*path-seg p i* (*length p*) $y \in$*path-seg p i* (*length p*)
then obtain *ix iy* **where**
IX: $x \in p!ix$ $i \le ix$ $ix<$*length p* **and**
IY: $y \in p!iy$ $i \le iy$ $iy<$*length p*
by (*auto simp*: *path-seg-def*)

```
      from IX have SS1: path-seg p ix (length p) ⊆ ?last-cnode
        by (subst path-seg-ss-eq) auto

      from IY have SS2: path-seg p i (Suc iy) ⊆ ?last-cnode
        by (subst path-seg-ss-eq) auto

      let ?rE = λR. (lvE ∩ R×R)
      let ?E = (insert (u,v) lvE ∩ ?last-cnode × ?last-cnode)

      from pathI[OF ‹x∈p!ix› ‹u∈p!(length p − 1)›] have
        (x,u)∈(?rE (path-seg p ix (Suc (length p − 1))))* using IX by auto
      hence (x,u)∈?E*
        apply (rule rtrancl-mono-mp[rotated])
        using SS1
        by auto

      also have (u,v)∈?E using ‹i<length p›
        apply (clarsimp)
        apply (intro conjI)
        apply (rule rev-subsetD[OF ‹u∈p!(length p − 1)›])
        apply (simp)
        apply (rule rev-subsetD[OF VMEM])
        apply (simp)
        done
      also
      from pathI[OF ‹v∈p!i› ‹y∈p!iy›] have
        (v,y)∈(?rE (path-seg p i (Suc iy)))* using IY by auto
      hence (v,y)∈?E*
        apply (rule rtrancl-mono-mp[rotated])
        using SS2
        by auto
      finally show (x,y)∈?E* .
    qed
  next
    assume j≠i
    with ‹j<Suc i› have [simp]: j<i by simp
    with ‹i<length p› have p!j∈set p
      by (metis Suc-lessD in-set-conv-nth less-trans-Suc)

    thus ?thesis using p-sc[of U] ‹p!j∈set p›
      apply (clarsimp)
      apply (subgoal-tac (a,b)∈(lvE ∩ p ! j × p ! j)*)
      apply (erule rtrancl-mono-mp[rotated])
      apply auto
      done
  qed
} note AUX-p-sc = this

{ fix j k
```

```
  assume A: j<k   k<length p'
  hence j<i by simp
  have insert (u, v) lvE ∩ p' ! k × p' ! j = {}
  proof −
    have {(u,v)} ∩ p' ! k × p' ! j = {}
      apply auto
      by (metis IL P-IDX-EQ Suc-lessD VMEM ‹j < i›
        less-irrefl-nat less-trans-Suc p-disjoint-sym)
    moreover have lvE ∩ p' ! k × p' ! j = {}
    proof (cases k<i)
      case True thus ?thesis
        using vE-no-back[of j k] A ‹i<length p› by auto
    next
      case False with A have [simp]: k=i by simp
      show ?thesis proof (rule disjointI, clarsimp simp: ‹j<i›)
        fix x y
        assume B: (x,y)∈lvE   x∈path-seg p i (length p)   y∈p!j
        then obtain ix where x∈p!ix   i≤ix   ix<length p
          by (auto simp: path-seg-def)
        moreover with A have j<ix by simp
        ultimately show False using vE-no-back[of j ix] B by auto
      qed
    qed
    ultimately show ?thesis by blast
  qed
} note AUX-vE-no-back = this

show ?thesis
  apply unfold-locales
  unfolding defs-fold

  apply simp

  using D-incr apply auto []

  using pE-E-from-p apply auto []

  using E-from-p-touched BACK apply (simp add: touched-def) apply blast

  apply (rule D-reachable)

  apply (rule AUX-p-connected, assumption+) []

  apply (rule AUX-p-disjoint, assumption+) []

  apply (rule AUX-p-sc, assumption+) []

  using root-v0
  apply (cases i)
```

apply (*simp add*: *p′-def collapse-aux-def*)
apply (*metis NE hd-in-set*)
apply (*cases p*, *simp-all add*: *p′-def collapse-aux-def*) []

apply (*simp add*: *p′-def collapse-aux-def*)

apply (*rule D-closed*)

apply (*drule* (*1*) *AUX-vE-no-back*, *auto*) []

using *p-not-D* **apply** *simp*
done
qed

lemma *invar-push*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: $p \neq []$
assumes *E*: $(u,v) \in pE$ **and** *UIL*: $u \in last\ p$
assumes *VNE*: $v \notin \bigcup (set\ p)$ $\quad v \notin D$
shows *invar v0 D0* (*push v* (*p*,*D*,$pE - \{(u,v)\}$))
unfolding *invar-def push-def*
apply *simp*
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

let *?thesis*
= *invar-loc G v0 D0* (p @ $[\{v\}]$) *D* ($pE - \{(u, v)\} \cup E \cap \{v\} \times UNIV$)

note *defs-fold* = *vE-push*[*OF E UIL VNE*] *touched-push*

{
fix *i*
assume *SILL*: *Suc i* < *length* (p @ $[\{v\}]$)
have (p @ $[\{v\}]$) ! i × (p @ $[\{v\}]$) ! *Suc i*
∩ ($E - (pE - \{(u, v)\} \cup E \cap \{v\} \times UNIV)) \neq \{\}$
proof (*cases* $i = length\ p - 1$)
case *True* **thus** *?thesis* **using** *SILL E pE-E-from-p UIL VNE*
by (*simp add*: *nth-append last-conv-nth*) *fast*
next
case *False*
with *SILL* **have** *SILL′*: *Suc i* < *length p* **by** *simp*

with *SILL′ VNE* **have** *X1*: $v \notin p!i$ $\quad v \notin p!Suc\ i$ **by** *auto*

from *p-connected*[*OF SILL′*] **obtain** *a b* **where**
$a \in p!i$ $\quad b \in p!Suc\ i$ $\quad (a,b) \in E$ $\quad (a,b) \notin pE$
by *auto*
with *X1* **have** $a \neq v$ $\quad b \neq v$ **by** *auto*
with ‹$(a,b) \in E$› ‹$(a,b) \notin pE$› **have** $(a,b) \in (E - (pE - \{(u, v)\} \cup E \cap \{v\} \times$

```
UNIV))
          by auto
        with ‹a∈p!i› ‹b∈p!Suc i›
        show ?thesis using  SILL'
          by (simp add: nth-append; blast)
      qed
    } note AUX-p-connected = this

    {
      fix U
      assume A: U ∈ set (p @ [{v}])
      have U × U ⊆ (insert (u, v) lvE ∩ U × U)*
      proof cases
        assume U∈set p
        with p-sc have U×U ⊆ (lvE ∩ U×U)* .
        thus ?thesis
          by (metis (lifting, no-types) Int-insert-left-if0 Int-insert-left-if1
            in-mono insert-subset rtrancl-mono-mp subsetI)
      next
        assume U∉set p with A have U={v} by simp
        thus ?thesis by auto
      qed
    } note AUX-p-sc = this

    {
      fix i j
      assume A: i < j    j < length (p @ [{v}])
      have insert (u, v) lvE ∩ (p @ [{v}]) ! j × (p @ [{v}]) ! i = {}
      proof (cases j=length p)
        case False with A have j<length p by simp
        from vE-no-back ‹i<j› this VNE show ?thesis
          by (auto simp add: nth-append)
      next
        from p-not-D A have PDDJ: p!i ∩ D = {}
          by (auto simp: Sup-inf-eq-bot-iff)
        case True thus ?thesis
          using A apply (simp add: nth-append)
          apply (rule conjI)
          using UIL A p-disjoint-sym
          apply (metis Misc.last-in-set NE UnionI VNE(1))

          using vE-touched VNE PDDJ apply (auto simp: touched-def) []
          done
      qed
    } note AUX-vE-no-back = this

    show ?thesis
      apply unfold-locales
      unfolding defs-fold
```

apply *simp*

using *D-incr* **apply** *auto* []

using *pE-E-from-p* **apply** *auto* []

using *E-from-p-touched VNE* **apply** (*auto simp*: *touched-def*) []

apply (*rule D-reachable*)

apply (*rule AUX-p-connected*, *assumption+*) []

using *p-disjoint* ‹$v \notin \bigcup (set\ p)$› **apply** (*auto simp*: *nth-append*) []

apply (*rule AUX-p-sc*, *assumption+*) []

using *root-v0* **apply** *simp*

apply *simp*

apply (*rule D-closed*)

apply (*rule AUX-vE-no-back*, *assumption+*) []

using *p-not-D VNE* **apply** *auto* []
done
qed

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* $(p,D,pE - \{(u, v)\})$
unfolding *invar-def*
apply *simp*
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*
let *?thesis* = *invar-loc G v0 D0 p D* $(pE - \{(u, v)\})$
note *defs-fold* = *vE-remove*[*OF NE E*]

show *?thesis*
apply *unfold-locales*
unfolding *defs-fold*

apply *simp*

using *D-incr* **apply** *auto* []

using *pE-E-from-p* **apply** *auto* []

using *E-from-p-touched VD* **apply** (*auto simp*: *touched-def*) []

apply (*rule D-reachable*)

using *p-connected* **apply** *auto* []

apply (*rule p-disjoint*, *assumption+*) []

apply (*drule p-sc*)
apply (*erule order-trans*)
apply (*rule rtrancl-mono*)
apply *blast* []

apply (*rule root-v0*, *assumption+*) []

apply (*rule p-empty-v0*, *assumption+*) []

apply (*rule D-closed*)

using *vE-no-back VD p-not-D*
apply *clarsimp*
apply (*metis Suc-lessD UnionI VNP less-trans-Suc nth-mem*)

apply (*rule p-not-D*)
done
qed

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

from *p-empty-v0 rtrancl-reachable-induct*[*OF order-refl D-closed*] *D-reachable*
show *?thesis* **by** *auto*
qed

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

```
  from INV interpret invar-loc G v0 D0 [] D pE unfolding invar-def by auto

  show ?thesis
  proof
    assume path lvE u q v
    thus path E u q v using path-mono[OF lvE-ss-E] by blast
  next
    assume path E u q v
    thus path lvE u q v using UR
    proof induction
      case (path-prepend u v p w)
      with fin-D-is-reachable[OF INV] have u∈D by auto
      with D-closed ‹(u,v)∈E› have v∈D by auto
      from path-prepend.prems path-prepend.hyps have v∈E* ``{v0} by auto
      with path-prepend.IH fin-D-is-reachable[OF INV] have path lvE v p w
        by simp
      moreover from ‹u∈D› ‹v∈D› ‹(u,v)∈E› D-vis have (u,v)∈lvE by auto
      ultimately show ?case by (auto simp: path-cons-conv)
    qed simp
  qed
qed

lemma invar-outer-newnode:
  assumes A: v0∉D0   v0∈it
  assumes OINV: outer-invar it D0
  assumes INV: invar v0 D0 ([],D',pE)
  shows outer-invar (it−{v0}) D'
proof −
  from OINV interpret outer-invar-loc G it D0 unfolding outer-invar-def .
  from INV interpret inv: invar-loc G v0 D0 [] D' pE
    unfolding invar-def by simp

  from fin-D-is-reachable[OF INV] have [simp]: v0∈D' by auto

  show ?thesis
    unfolding outer-invar-def
    apply unfold-locales
    using it-initial apply auto []
    using it-done inv.D-incr apply auto []
    using inv.D-reachable apply assumption
    using inv.D-closed apply assumption
    done
qed

lemma invar-outer-Dnode:
  assumes A: v0∈D0   v0∈it
  assumes OINV: outer-invar it D0
  shows outer-invar (it−{v0}) D0
proof −
```

from *OINV* **interpret** *outer-invar-loc G it D0* **unfolding** *outer-invar-def* **.**

show *?thesis*
unfolding *outer-invar-def*
apply *unfold-locales*
using *it-initial* **apply** *auto* []
using *it-done A* **apply** *auto* []
using *D-reachable* **apply** *assumption*
using *D-closed* **apply** *assumption*
done
qed

lemma *pE-fin'*: *invar x* σ ([], *D*, *pE*) $\Longrightarrow$ *pE*={}
unfolding *invar-def* **by** (*simp add*: *invar-loc.pE-fin*)

end

Termination

context *invar-loc*
begin
lemma *unproc-finite*[*simp*, *intro*!]: *finite* (*unproc-edges v0 p D pE*)
— The set of unprocessed edges is finite
proof −
have *unproc-edges v0 p D pE* $\subseteq$ E^* ``{*v0*} $\times$ $E^*$ ``{*v0*}
unfolding *unproc-edges-def*
using *pE-reachable*
by *auto*
thus *?thesis*
by (*rule finite-subset*) *simp*
qed

lemma *unproc-decreasing*:
— As effect of selecting a pending edge, the set of unprocessed edges decreases
assumes [*simp*]: *p*≠[] **and** *A*: (*u*,*v*)∈*pE*   *u*∈*last p*
shows *unproc-edges v0 p D* (*pE*−{(*u*,*v*)}) $\subset$ *unproc-edges v0 p D pE*
using *A* **unfolding** *unproc-edges-def*
by *fastforce*
end

context *fr-graph*
begin

lemma *abs-wf-pop*:
assumes *INV*: *invar v0 D0* (*p*,*D*,*pE*)
assumes *NE*[*simp*]: *p*≠[]
assumes *NO*: *pE* $\cap$ *last aba* $\times$ *UNIV* = {}
shows (*pop* (*p*,*D*,*pE*), (*p*, *D*, *pE*)) $\in$ *abs-wf-rel v0*
unfolding *pop-def*
apply *simp*
proof −

from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

let *?thesis* = ((*butlast p*, *last p* ∪ *D*, *pE*), *p*, *D*, *pE*) ∈ *abs-wf-rel v0*
have *unproc-edges v0* (*butlast p*) (*last p* ∪ *D*) *pE* = *unproc-edges v0 p D pE*
unfolding *unproc-edges-def*
apply (*cases p rule*: *rev-cases*, *simp*)
apply *auto*
done
thus *?thesis*
by (*auto simp*: *abs-wf-rel-def*)
qed

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*
unfolding *collapse-def*
apply *simp*
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

define *i* **where** *i* = *idx-of p v*
let *?thesis*
= ((*collapse-aux p i*, *D*, *pE*−{(*u*,*v*)}), (*p*, *D*, *pE*)) ∈ *abs-wf-rel v0*

have *unproc-edges v0* (*collapse-aux p i*) *D* (*pE*−{(*u*,*v*)})
= *unproc-edges v0 p D* (*pE*−{(*u*,*v*)})
unfolding *unproc-edges-def* **by** (*auto*)
also note *unproc-decreasing*[*OF NE E*]
finally show *?thesis*
by (*auto simp*: *abs-wf-rel-def*)
qed

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*
unfolding *push-def*
apply *simp*
proof −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

let *?thesis*
= ((*p*@[{*v*}], *D*, *pE*−{(*u*,*v*)} ∪ *E*∩{*v*}×*UNIV*), (*p*, *D*, *pE*)) ∈ *abs-wf-rel v0*

have *unproc-edges v0* (*p*@[{*v*}]) *D* (*pE*−{(*u*,*v*)} ∪ *E*∩{*v*}×*UNIV*)
= *unproc-edges v0 p D* (*pE*−{(*u*,*v*)})

unfolding *unproc-edges-def*
using *E A pE-reachable*
by *auto*
also note *unproc-decreasing*[*OF NE E*]
finally show *?thesis*
by (*auto simp*: *abs-wf-rel-def*)
qed

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 −
from *INV* **interpret** *invar-loc G v0 D0 p D pE* **unfolding** *invar-def* **by** *simp*

from *unproc-decreasing*[*OF NE E*] **show** *?thesis*
by (*auto simp*: *abs-wf-rel-def*)
qed
end

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

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

theorem *skeleton-spec*: *skeleton* ≤ *SPEC* (λ*D*. *outer-invar* {} *D*)
proof −
note [*simp del*] = *Union-iff*
note [[*goals-limit* = *4*]]

show *?thesis*
unfolding *skeleton-def select-edge-def select-def*
apply (*refine-vcg WHILEIT-rule*[*OF abs-wf-rel-wf*])
apply (*vc-solve solve*: *invar-preserve simp*: *pE-fin′ finite-V0*)
apply *auto*
done
qed

Short proof, as presented in the paper

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

    unfolding *skeleton-def select-edge-def select-def*
    by (*refine-vcg WHILEIT-rule*[*OF abs-wf-rel-wf*])
      (*auto intro*: *invar-preserve simp*: *pE-fin′ finite-V0*)
  end

end

1.3.5 Consequences of Invariant when Finished

context *fr-graph*
begin
  lemma *fin-outer-D-is-reachable*:
    — When outer loop terminates, exactly the reachable nodes are finished
    assumes *INV*: *outer-invar* {} *D*
    shows $D = E^* `` V0$
  proof −
    from *INV* **interpret** *outer-invar-loc G* {} *D* **unfolding** *outer-invar-def* **by** *auto*

    from *it-done rtrancl-reachable-induct*[*OF order-refl D-closed*] *D-reachable*
    show *?thesis* **by** *auto*
  qed

end

1.4 Refinement to Gabow's Data Structure

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

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

1.4.1 Preliminaries

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

lemma *find-max-nat-correct*:
  $[\![P\ 0;\ 0<u]\!] \Longrightarrow$ *find-max-nat u P* = *Max* {*i. i<u* ∧ *P i*}
  apply (*induction u*)
  apply *auto*

apply (*rule Max-eqI*[*THEN sym*])
apply *auto* [*3*]

apply (*case-tac u*)
apply *simp*
apply *clarsimp*
by (*metis less-SucI less-antisym*)

lemma *find-max-nat-param*[*param*]:
assumes $(n,n')\in nat\text{-}rel$
assumes $\bigwedge j\ j'.\ [\![(j,j')\in nat\text{-}rel;\ j'<n']\!] \Longrightarrow (P\ j,P'\ j')\in bool\text{-}rel$
shows $(find\text{-}max\text{-}nat\ n\ P, find\text{-}max\text{-}nat\ n'\ P') \in nat\text{-}rel$
using *assms*
by (*induction n arbitrary*: n') *auto*

context begin interpretation *autoref-syn* **.**
lemma *find-max-nat-autoref*[*autoref-rules*]:
assumes $(n,n')\in nat\text{-}rel$
assumes $\bigwedge j\ j'.\ [\![(j,j')\in nat\text{-}rel;\ j'<n']\!] \Longrightarrow (P\ j,P'\$j')\in bool\text{-}rel$
shows $(find\text{-}max\text{-}nat\ n\ P,$
$(OP\ find\text{-}max\text{-}nat ::: nat\text{-}rel \to (nat\text{-}rel\to bool\text{-}rel) \to nat\text{-}rel)\ \$n'\$P'$
$) \in nat\text{-}rel$
using *find-max-nat-param*[*OF assms*]
by *simp*

end

1.4.2 Gabow's Datastructure

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

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

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

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

type-synonym $'a\ GS$
$= 'a\ list \times nat\ list \times ('a \rightharpoonup node\text{-}state) \times (nat \times 'a\ set)\ list$
locale *GS* =
fixes $SBIP :: 'a\ GS$
begin
definition $S \equiv (\lambda(S,B,I,P).\ S)\ SBIP$
definition $B \equiv (\lambda(S,B,I,P).\ B)\ SBIP$
definition $I \equiv (\lambda(S,B,I,P).\ I)\ SBIP$

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

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

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

definition *seg* :: *nat* ⇒ *'a set* — Collapsed set at index
where *seg* $i \equiv \{S!j \mid j.$ *seg-start* $i \leq j \wedge j <$ *seg-end* $i\ \}$

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

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

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

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

end

lemma *GS-sel-simps*[*simp*]:
GS.S (*S*,*B*,*I*,*P*) = *S*
GS.B (*S*,*B*,*I*,*P*) = *B*
GS.I (*S*,*B*,*I*,*P*) = *I*
GS.P (*S*,*B*,*I*,*P*) = *P*
unfolding *GS.S-def GS.B-def GS.I-def GS.P-def*
by *auto*

context *GS* **begin**
lemma *seg-start-indep*[*simp*]: *GS.seg-start* (*S'*,*B*,*I'*,*P'*) = *seg-start*
unfolding *GS.seg-start-def*[*abs-def*] **by** (*auto*)
lemma *seg-end-indep*[*simp*]: *GS.seg-end* (*S*,*B*,*I'*,*P'*) = *seg-end*
unfolding *GS.seg-end-def*[*abs-def*] **by** *auto*
lemma *seg-indep*[*simp*]: *GS.seg* (*S*,*B*,*I'*,*P'*) = *seg*
unfolding *GS.seg-def*[*abs-def*] **by** *auto*
lemma *p-α-indep*[*simp*]: *GS.p-α* (*S*,*B*,*I'*,*P'*) = *p-α*
unfolding *GS.p-α-def* **by** *auto*

lemma *D-α-indep*[*simp*]: *GS.D-α* (*S'*,*B'*,*I*,*P'*) = *D-α*
unfolding *GS.D-α-def* **by** *auto*

lemma *pE-α-indep*[*simp*]: *GS.pE-α* (*S*,*B'*,*I'*,*P*) = *pE-α*
unfolding *GS.pE-α-def* **by** *auto*

definition *find-seg* — Abs-path index for stack index
where *find-seg* $j \equiv$ *Max* $\{i.\ i<$*length* $B \wedge B!i\leq j\}$

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

end

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

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

  assumes P-sorted: sorted (map fst P)
  assumes P-distinct: distinct (map fst P)
  assumes P-bound: set P ⊆ {0..<length S}×Collect ((≠) {})
begin
  lemma locale-this: GS-invar SBIP by unfold-locales

end

definition oGS-rel ≡ br oGS-α oGS-invar
lemma oGS-rel-sv[intro!,simp,relator-props]: single-valued oGS-rel
  unfolding oGS-rel-def by auto

definition GS-rel ≡ br GS.α GS-invar
lemma GS-rel-sv[intro!,simp,relator-props]: single-valued GS-rel
  unfolding GS-rel-def by auto

context GS-invar
begin
  lemma empty-eq: S=[] ⟷ B=[]
    using B-in-bound B0 by auto

  lemma B-in-bound': i<length B ⟹ B!i < length S
    using B-in-bound nth-mem by fastforce

  lemma seg-start-bound:
    assumes A: i<length B shows seg-start i < length S
    using B-in-bound nth-mem[OF A] unfolding seg-start-def by auto

  lemma seg-end-bound:
    assumes A: i<length B shows seg-end i ≤ length S
  proof (cases i+1=length B)
    case True thus ?thesis by (simp add: seg-end-def)
  next
    case False with A have i+1<length B by simp
    from nth-mem[OF this] B-in-bound have  B ! (i + 1) < length S by auto
```

```
  thus ?thesis using False by (simp add: seg-end-def)
qed

lemma seg-start-less-end: i<length B ⟹ seg-start i < seg-end i
  unfolding seg-start-def seg-end-def
  using B-in-bound' distinct-sorted-mono[OF B-sorted B-distinct]
  by auto

lemma seg-end-less-start: ⟦i<j; j<length B⟧ ⟹ seg-end i ≤ seg-start j
  unfolding seg-start-def seg-end-def
  by (auto simp: distinct-sorted-mono-iff[OF B-distinct B-sorted])

lemma find-seg-bounds:
  assumes A: j<length S
  shows seg-start (find-seg j) ≤ j
  and j < seg-end (find-seg j)
  and find-seg j < length B
proof −
  let ?M = {i. i<length B ∧ B!i≤j}
  from A have [simp]: B≠[] using empty-eq by (cases S) auto
  have NE: ?M≠{} using A B0 by (cases B) auto

  have F: finite ?M by auto

  from Max-in[OF F NE]
  have LEN: find-seg j < length B and LB: B!find-seg j ≤ j
    unfolding find-seg-def
    by auto

  thus find-seg j < length B by −

  from LB show LB': seg-start (find-seg j) ≤ j
    unfolding seg-start-def by simp

  moreover show UB': j < seg-end (find-seg j)
    unfolding seg-end-def
  proof (split if-split, intro impI conjI)
    show j<length S using A .

    assume find-seg j + 1 ≠ length B
    with LEN have P1: find-seg j + 1 < length B by simp

    show j < B ! (find-seg j + 1)
    proof (rule ccontr, simp only: linorder-not-less)
      assume P2: B ! (find-seg j + 1) ≤ j
      with P1 Max-ge[OF F, of find-seg j + 1, folded find-seg-def]
      show False by simp
    qed
  qed
```

qed

lemma *find-seg-correct*:
  assumes *A*: $j<length\ S$
  shows $S!j \in seg\ (find\text{-}seg\ j)$ **and** $find\text{-}seg\ j < length\ B$
  using *find-seg-bounds*[*OF A*]
    unfolding *seg-def* **by** *auto*

lemma *set-p-α-is-set-S*:
  $\bigcup(set\ p\text{-}\alpha) = set\ S$
  apply *rule*
  unfolding *p-α-def seg-def*[*abs-def*]
  using *seg-end-bound* **apply** *fastforce* []

  apply (*auto simp*: *in-set-conv-nth*)

  using *find-seg-bounds*
  apply (*fastforce simp*: *in-set-conv-nth*)
  done

lemma *S-idx-uniq*:
  $[\![i<length\ S;\ j<length\ S]\!] \Longrightarrow S!i=S!j \longleftrightarrow i=j$
  using *S-distinct*
  by (*simp add*: *nth-eq-iff-index-eq*)

lemma *S-idx-of-correct*:
  assumes *A*: $v\in\bigcup(set\ p\text{-}\alpha)$
  shows $S\text{-}idx\text{-}of\ v < length\ S$ **and** $S!S\text{-}idx\text{-}of\ v = v$
proof −
  from *A* **have** $v\in set\ S$ **by** (*simp add*: *set-p-α-is-set-S*)
  then obtain *j* **where** *G1*: $j<length\ S \quad v=S!j$ **by** (*auto simp*: *in-set-conv-nth*)
  with *I-consistent* **have** $I\ v = Some\ (STACK\ j)$ **by** *simp*
  hence $S\text{-}idx\text{-}of\ v = j$ **by** (*simp add*: *S-idx-of-def*)
  with *G1* **show** $S\text{-}idx\text{-}of\ v < length\ S$ **and** $S!S\text{-}idx\text{-}of\ v = v$ **by** *simp-all*
qed

lemma *p-α-disjoint-sym*:
  shows $\forall\, i\ j\ v.\ i<length\ p\text{-}\alpha \wedge j<length\ p\text{-}\alpha \wedge v\in p\text{-}\alpha!i \wedge v\in p\text{-}\alpha!j \longrightarrow i=j$
proof (*intro allI impI*, *elim conjE*)
  fix *i j v*
  assume *A*: $i < length\ p\text{-}\alpha \quad j < length\ p\text{-}\alpha \quad v \in p\text{-}\alpha\ !\ i \quad v \in p\text{-}\alpha\ !\ j$
  from *A* **have** *LI*: $i<length\ B$ **and** *LJ*: $j<length\ B$ **by** (*simp-all add*: *p-α-def*)

  from *A* **have** *B1*: $seg\text{-}start\ j < seg\text{-}end\ i$ **and** *B2*: $seg\text{-}start\ i < seg\text{-}end\ j$
    unfolding *p-α-def seg-def*[*abs-def*]
    apply *clarsimp-all*
    apply (*subst* (*asm*) *S-idx-uniq*)
    apply (*metis dual-order.strict-trans1 seg-end-bound*)
    apply (*metis dual-order.strict-trans1 seg-end-bound*)

```
      apply simp
      apply (subst (asm) S-idx-uniq)
      apply (metis dual-order.strict-trans1 seg-end-bound)
      apply (metis dual-order.strict-trans1 seg-end-bound)
      apply simp
      done

    from B1 have B1: (B!j < B!Suc i ∧ Suc i < length B) ∨ i=length B − 1
      using LI unfolding seg-start-def seg-end-def by (auto split: if-split-asm)

    from B2 have B2: (B!i < B!Suc j ∧ Suc j < length B) ∨ j=length B − 1
      using LJ unfolding seg-start-def seg-end-def by (auto split: if-split-asm)

    from B1 have B1: j<Suc i ∨ i=length B − 1
      using LI LJ distinct-sorted-strict-mono-iff[OF B-distinct B-sorted]
      by auto

    from B2 have B2: i<Suc j ∨ j=length B − 1
      using LI LJ distinct-sorted-strict-mono-iff[OF B-distinct B-sorted]
      by auto

    from B1 B2 show i=j
      using LI LJ
      by auto
  qed

end
```

1.4.3 Refinement of the Operations

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

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

```
definition mark-as-done
  where ⋀l u I. mark-as-done l u I ≡ do {
  (-,I)←WHILET
    (λ(l,I). l<u)
    (λ(l,I). do { ASSERT (l<length S); RETURN (Suc l,I(S!l ↦ DONE))})
    (l,I);
  RETURN I
}

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

lemma mark-as-done-aux:
  fixes l u I
  shows ⟦l<u; u≤length S⟧ ⟹ mark-as-done l u I
  ≤ SPEC (λr. r = mark-as-done-abs l u I)
  unfolding mark-as-done-def mark-as-done-abs-def
  apply (refine-rcg
    WHILET-rule[where
      I=λ(l',I').
        I' = (λv. if v∈{S!j | j. l≤j ∧ j<l'} then Some DONE else I v)
        ∧ l≤l' ∧ l'≤u
      and R=measure (λ(l',-). u−l')
    ]
    refine-vcg)

  apply (auto intro!: ext simp: less-Suc-eq)
  done

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

definition sel-rem-last ≡
  if P=[] then
    RETURN (None,(S,B,I,P))
  else do {
    let (j,succs) = last P;
```

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


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

lemma (in GS-invar) find-seg-impl:
  j<length S ⟹ find-seg-impl j = find-seg j
  unfolding find-seg-impl-def
  thm find-max-nat-correct
  apply (subst find-max-nat-correct)
  apply (simp add: B0)
  apply (simp add: B0)
  apply (simp add: find-seg-def)
  done


definition idx-of-impl v ≡ do {
    ASSERT (∃ i. I v = Some (STACK i));
    let j = S-idx-of v;
    ASSERT (j<length S);
    let i = find-seg-impl j;
    RETURN i
  }

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

end

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

from *REL* **have** [*simp*]: $D = oGS\text{-}\alpha\ I$ **and** *I*: *oGS-invar I*
by (*simp-all add*: *oGS-rel-def br-def*)

from *I* **have** [*simp*]: $\bigwedge j\ v.\ I\ v \neq Some\ (STACK\ j)$
by (*simp add*: *oGS-invar-def*)

show *?G1*
unfolding $GS.\alpha$*-def GS-initial-impl-def*
apply (*simp split del*: *if-split*) **apply** (*intro conjI*)

unfolding $GS.p\text{-}\alpha$*-def GS.seg-def*[*abs-def*] *GS.seg-start-def GS.seg-end-def*
apply (*auto*) []

using *A* **unfolding** $GS.D\text{-}\alpha$*-def* **apply** (*auto simp*: $oGS\text{-}\alpha$*-def*) []

unfolding $GS.pE\text{-}\alpha$*-def* **apply** *auto* []
done

show *?G2*
unfolding *GS-initial-impl-def*
apply *unfold-locales*
apply *auto*
done
qed

context *GS-invar*
begin
lemma *push-correct*:
assumes *A*: $v \notin \bigcup (set\ p\text{-}\alpha)$ **and** *B*: $v \notin D\text{-}\alpha$
shows $GS.\alpha\ (push\text{-}impl\ v\ succs) = (p\text{-}\alpha @ [\{v\}], D\text{-}\alpha, pE\text{-}\alpha \cup \{v\} \times succs)$
(**is** *?G1*)
and *GS-invar* (*push-impl v succs*) (**is** *?G2*)
proof −

note [*simp*] = *Let-def*

have *A1*: $GS.D\text{-}\alpha\ (push\text{-}impl\ v\ succs) = D\text{-}\alpha$
using *B*
by (*auto simp*: *push-impl-def* $GS.D\text{-}\alpha$*-def*)

have *iexI*: $\bigwedge a\ b\ j\ P.\ [\![a!j = b!j;\ P\ j]\!] \Longrightarrow \exists j'.\ a!j = b!j' \wedge P\ j'$
by *blast*

have *A2*: $GS.p\text{-}\alpha\ (push\text{-}impl\ v\ succs) = p\text{-}\alpha\ @\ [\{v\}]$
unfolding *push-impl-def* $GS.p\text{-}\alpha$*-def GS.seg-def*[*abs-def*]
GS.seg-start-def GS.seg-end-def
apply (*clarsimp split del*: *if-split*)

apply *clarsimp*

apply *safe*
apply (((*rule iexI*)?,
(*auto*
simp: *nth-append nat-in-between-eq*
dest: *order.strict-trans*[*OF - B-in-bound'*]
)) []
) +
done

have *iexI2*: $\bigwedge j\ I\ Q.\ [\![(j,I)\in set\ P;\ (j,I)\in set\ P \Longrightarrow Q\ j]\!] \Longrightarrow \exists j.\ Q\ j$
by *blast*

have *A3*: $GS.pE\text{-}\alpha\ (push\text{-}impl\ v\ succs) = pE\text{-}\alpha \cup \{v\} \times succs$
unfolding *push-impl-def GS.pE-α-def*
using *P-bound*
apply (*force simp*: *nth-append elim*!: *iexI2*)
done

show *?G1*
unfolding *GS.α-def*
by (*simp add*: *A1 A2 A3*)

show *?G2*
apply *unfold-locales*
unfolding *push-impl-def*
apply *simp-all*

using *B-in-bound B-sorted B-distinct* **apply** (*auto simp*: *sorted-append*) [*3*]
using *B-in-bound B0* **apply** (*cases S*) **apply** (*auto simp*: *nth-append*) [*2*]

using *S-distinct A* **apply** (*simp add*: *set-p-α-is-set-S*)

using *A I-consistent*
apply (*auto simp*: *nth-append set-p-α-is-set-S split*: *if-split-asm*) []

using *P-sorted P-distinct P-bound* **apply** (*auto simp*: *sorted-append*) [*3*]
done
qed

lemma *no-last-out-P-aux*:
assumes *NE*: $p\text{-}\alpha \neq []$ **and** *NS*: $pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV = \{\}$
shows $set\ P \subseteq \{0..<last\ B\} \times UNIV$
proof −
{
fix *j I*
assume *jII*: $(j,I)\in set\ P$
and *JL*: $last\ B \le j$
with *P-bound* **have** *JU*: $j<length\ S$ **and** *INE*: $I\neq\{\}$ **by** *auto*
with *JL JU* **have** $S!j \in last\ p\text{-}\alpha$

using *NE*
unfolding *p-α-def*
apply (*auto*
simp: *last-map seg-def seg-start-def seg-end-def last-conv-nth*)
done
moreover from *jII* **have** $\{S!j\} \times I \subseteq pE\text{-}\alpha$ **unfolding** *pE-α-def*
by *auto*
moreover note *INE NS*
ultimately have *False* **by** *blast*
} **thus** *?thesis* **by** *fastforce*
qed

lemma *pop-correct*:
assumes *NE*: $p\text{-}\alpha \neq []$ **and** *NS*: $pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV = \{\}$
shows *pop-impl*
$\leq \Downarrow GS\text{-}rel\ (SPEC\ (\lambda r.\ r=(butlast\ p\text{-}\alpha,\ D\text{-}\alpha \cup last\ p\text{-}\alpha,\ pE\text{-}\alpha)))$
proof −
have *iexI*: $\bigwedge a\ b\ j\ P.\ [\![a!j = b!j;\ P\ j]\!] \Longrightarrow \exists j'.\ a!j = b!j' \wedge P\ j'$
by *blast*

have [*simp*]: $\bigwedge n.\ n - Suc\ 0 \neq n \longleftrightarrow n \neq 0$ **by** *auto*

from *NE* **have** *BNE*: $B \neq []$
unfolding *p-α-def* **by** *auto*

{
fix *i j*
assume *B*: $j<B!i$ **and** *A*: $i<length\ B$
note *B*
also from *sorted-nth-mono*[*OF B-sorted*, *of i length B − 1*] *A*
have $B!i \leq last\ B$
by (*simp add*: *last-conv-nth*)
finally have $j < last\ B$ **.**
hence $take\ (last\ B)\ S\ !\ j = S\ !\ j$
and $take\ (B!(length\ B - Suc\ 0))\ S\ !j = S!j$
by (*simp-all add*: *last-conv-nth BNE*)
} **note** *AUX1=this*

{
fix *v j*
have (*mark-as-done-abs*
(*seg-start* (*length B* − *Suc 0*))
(*seg-end* (*length B* − *Suc 0*)) *I v* = *Some* (*STACK j*))
$\longleftrightarrow (j < length\ S \wedge j < last\ B \wedge v = take\ (last\ B)\ S\ !\ j)$
apply (*simp add*: *mark-as-done-abs-def*)
apply *safe* []
using *I-consistent*
apply (*clarsimp-all*
simp: *seg-start-def seg-end-def last-conv-nth BNE*

```
      simp: S-idx-uniq)

    apply (force)
    apply (subst nth-take)
    apply force
    apply force
    done
} note AUX2 = this

define ci where ci = (
  take (last B) S,
  butlast B,
  mark-as-done-abs
    (seg-start (length B − Suc 0)) (seg-end (length B − Suc 0)) I,
  P)

have ABS: GS.α ci = (butlast p-α, D-α ∪ last p-α, pE-α)
  apply (simp add: GS.α-def ci-def)
  apply (intro conjI)
  apply (auto
    simp del: map-butlast
    simp add: map-butlast[symmetric] butlast-upt
    simp add: GS.p-α-def GS.seg-def[abs-def] GS.seg-start-def GS.seg-end-def
    simp: nth-butlast last-conv-nth nth-take AUX1
    cong: if-cong
    intro!: iexI
    dest: order.strict-trans[OF - B-in-bound']
  ) []

  apply (auto
    simp: GS.D-α-def p-α-def last-map BNE seg-def mark-as-done-abs-def) []

  using AUX1 no-last-out-P-aux[OF NE NS]
  apply (auto simp: GS.pE-α-def mark-as-done-abs-def elim!: bex2I) []
  done

have INV: GS-invar ci
  apply unfold-locales
  apply (simp-all add: ci-def)

  using B-in-bound B-sorted B-distinct
  apply (cases B rule: rev-cases, simp)
  apply (auto simp: sorted-append order.strict-iff-order) []

  using B-sorted BNE apply (auto simp: sorted-butlast) []

  using B-distinct BNE apply (auto simp: distinct-butlast) []

  using B0 apply (cases B rule: rev-cases, simp add: BNE)
```

apply (*auto simp*: *nth-append split*: *if-split-asm*) []

using *S-distinct* **apply** (*auto*) []

apply (*rule AUX2*)

using *P-sorted P-distinct*
apply (*auto*) [*2*]

using *P-bound no-last-out-P-aux*[*OF NE NS*]
apply (*auto simp*: *in-set-conv-decomp*)
done

show *?thesis*
unfolding *pop-impl-def*
apply (*refine-rcg*
SPEC-refine refine-vcg order-trans[*OF mark-as-done-aux*])
apply (*simp-all add*: *BNE seg-start-less-end seg-end-bound*)
apply (*fold ci-def*)
unfolding *GS-rel-def*
apply (*rule brI*)
apply (*simp-all add*: *ABS INV*)
done
qed

lemma *sel-rem-last-correct*:
assumes *NE*: $p\text{-}\alpha \neq []$
shows
$sel\text{-}rem\text{-}last \le \Downarrow(Id \times_r GS\text{-}rel)\ (select\text{-}edge\ (p\text{-}\alpha, D\text{-}\alpha, pE\text{-}\alpha))$
proof −
{
fix *l i a b b′*
have $[\![i < length\ l;\ l!i = (a,b)]\!] \Longrightarrow map\ fst\ (l[i := (a,b')]) = map\ fst\ l$
by (*induct l arbitrary*: *i*) (*auto split*: *nat.split*)
} **note** *map-fst-upd-snd-eq* = *this*

from *NE* **have** *BNE*[*simp*]: $B \neq []$ **unfolding** *p-α-def* **by** *simp*

have *INVAR*: $sel\text{-}rem\text{-}last \le SPEC\ (GS\text{-}invar\ o\ snd)$
unfolding *sel-rem-last-def*
apply (*refine-rcg refine-vcg*)
using *locale-this* **apply** (*cases SBIP*) **apply** *simp*

apply *simp*

using *P-bound* **apply** (*cases P rule*: *rev-cases*, *auto*) []

apply *simp*

apply *simp* **apply** (*intro impI conjI*)

apply (*unfold-locales*, *simp-all*) []
using *B-in-bound B-sorted B-distinct B0 S-distinct I-consistent*
apply *auto* [*6*]

using *P-sorted P-distinct*
apply (*auto simp*: *map-butlast sorted-butlast distinct-butlast*) [*2*]

using *P-bound* **apply** (*auto dest*: *in-set-butlastD*) []

apply (*unfold-locales*, *simp-all*) []
using *B-in-bound B-sorted B-distinct B0 S-distinct I-consistent*
apply *auto* [*6*]

using *P-sorted P-distinct*
apply (*auto simp*: *last-conv-nth map-fst-upd-snd-eq*) [*2*]

using *P-bound*
apply (*cases P rule*: *rev-cases*, *simp*)
apply (*auto*) []

using *locale-this* **apply** (*cases SBIP*) **apply** *simp*
done

{
assume *NS*: $pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV = \{\}$
hence *sel-rem-last*
$\leq SPEC\ (\lambda r.\ case\ r\ of\ (None,SBIP') \Rightarrow SBIP'=SBIP \mid \text{-} \Rightarrow False)$
unfolding *sel-rem-last-def*
apply (*refine-rcg refine-vcg*)
apply (*cases SBIP*)
apply *simp*

apply *simp*
using *P-bound* **apply** (*cases P rule*: *rev-cases*, *auto*) []
apply *simp*

using *no-last-out-P-aux*[*OF NE NS*]
apply (*auto simp*: *seg-start-def last-conv-nth*) []

apply (*cases SBIP*)
apply *simp*
done
} **note** *SPEC-E* = *this*

```
{
 assume NON-EMPTY: pE-α∩last p-α×UNIV ≠ {}

 then obtain j succs P' where
   EFMT: P = P'@[(j,succs)]
   unfolding pE-α-def
   by (cases P rule: rev-cases) auto

 with P-bound have J-UPPER: j<length S and SNE: succs≠{}
   by auto

 have J-LOWER: seg-start (length B − Suc 0) ≤ j
 proof (rule ccontr)
   assume ¬(seg-start (length B − Suc 0) ≤ j)
   hence j < seg-start (length B − 1) by simp
   with P-sorted EFMT
   have P-bound': set P ⊆ {0..<seg-start (length B − 1)} × UNIV
     by (auto simp: sorted-append)
   hence pE-α ∩ last p-α×UNIV = {}
     by (auto
       simp: p-α-def last-conv-nth seg-def pE-α-def S-idx-uniq seg-end-def)
   thus False using NON-EMPTY by simp
 qed

 from J-UPPER J-LOWER have SJL: S!j∈last p-α
   unfolding p-α-def seg-def[abs-def] seg-end-def
   by (auto simp: last-map)

 from EFMT have SSS: {S!j}×succs⊆pE-α
   unfolding pE-α-def
   by auto


 {
   fix v
   assume v∈succs
   with SJL SSS have G: (S!j,v)∈pE-α ∩ last p-α×UNIV by auto

   {
     fix j' succs'
     assume S ! j' = S ! j    (j', succs') ∈ set P'
     with J-UPPER P-bound S-idx-uniq EFMT have j'=j by auto
     with P-distinct ‹(j', succs') ∈ set P'› EFMT have False by auto
   } note AUX3=this

   have G1: GS.pE-α (S,B,I,P' @ [(j, succs − {v})]) = pE-α − {(S!j, v)}
     unfolding GS.pE-α-def using AUX3
     by (auto simp: EFMT)
```

{
  assume *succs*$\subseteq\{v\}$
  hence $GS.pE\text{-}\alpha\ (S,B,I,P'\ @\ [(j,\ succs - \{v\})]) = GS.pE\text{-}\alpha\ (S,B,I,P')$
    unfolding *GS.pE-α-def* **by** *auto*

  with *G1* **have** $GS.pE\text{-}\alpha\ (S,B,I,P') = pE\text{-}\alpha - \{(S!j,\ v)\}$ **by** *simp*
} **note** *G2* = *this*

note *G G1 G2*
} **note** *AUX3* = *this*

have $sel\text{-}rem\text{-}last \le SPEC\ (\lambda r.\ case\ r\ of$
  $(Some\ v,SBIP') \Rightarrow \exists\, u.$
    $(u,v)\in(pE\text{-}\alpha\cap last\ p\text{-}\alpha\times UNIV)$
   $\wedge\ GS.\alpha\ SBIP' = (p\text{-}\alpha,D\text{-}\alpha,pE\text{-}\alpha-\{(u,v)\})$
| - $\Rightarrow$ *False*)
  unfolding *sel-rem-last-def*
  apply (*refine-rcg refine-vcg*)

  using *SNE* **apply** (*vc-solve simp*: *J-LOWER EFMT*)

  apply (*frule AUX3*(*1*))

  apply *safe*

  apply (*drule* (*1*) *AUX3*(*3*)) **apply** (*auto simp*: *EFMT GS.α-def*) []
  apply (*drule AUX3*(*2*)) **apply** (*auto simp*: *GS.α-def*) []
  done
} **note** *SPEC-NE*=*this*

have *SPEC*: $sel\text{-}rem\text{-}last \le SPEC\ (\lambda r.\ case\ r\ of$
  $(None,\ SBIP') \Rightarrow SBIP' = SBIP \wedge pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV = \{\} \wedge$
GS-invar SBIP
  | $(Some\ v,\ SBIP') \Rightarrow \exists\, u.\ (u,\ v) \in pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV$
    $\wedge\ GS.\alpha\ SBIP' = (p\text{-}\alpha,\ D\text{-}\alpha,\ pE\text{-}\alpha - \{(u,\ v)\})$
    $\wedge$ *GS-invar SBIP′*
)
  using *INVAR*
  apply ($cases\ pE\text{-}\alpha \cap last\ p\text{-}\alpha \times UNIV = \{\}$)
  apply (*frule SPEC-E*)
  apply (*auto split*: *option.splits simp*: *pw-le-iff*; *blast*; *fail*)
  apply (*frule SPEC-NE*)
  apply (*auto split*: *option.splits simp*: *pw-le-iff*; *blast*; *fail*)
  done

have *X1*: $(\exists\, y.\ (y=None \longrightarrow \Phi\ y) \wedge (\forall\, a\ b.\ y=Some\ (a,b) \longrightarrow \Psi\ y\ a\ b)) \longleftrightarrow$
  $(\Phi\ None \vee (\exists\, a\ b.\ \Psi\ (Some\ (a,b))\ a\ b))$ **for** $\Phi\ \Psi$
  by *auto*

```
  show ?thesis
    apply (rule order-trans[OF SPEC])
    unfolding select-edge-def select-def
    apply (simp
      add: pw-le-iff refine-pw-simps prod-rel-sv
      del: SELECT-pw
      split: option.splits prod.splits)
    apply (fastforce simp: br-def GS-rel-def GS.α-def)
    done
qed

lemma find-seg-idx-of-correct:
  assumes A: v∈⋃(set p-α)
  shows (find-seg (S-idx-of v)) = idx-of p-α v
proof -
  note S-idx-of-correct[OF A] idx-of-props[OF p-α-disjoint-sym A]
  from find-seg-correct[OF ‹S-idx-of v < length S›] have
    find-seg (S-idx-of v) < length p-α
    and S!S-idx-of v ∈ p-α!find-seg (S-idx-of v)
    unfolding p-α-def by auto
  from idx-of-uniq[OF p-α-disjoint-sym this] ‹S ! S-idx-of v = v›
  show ?thesis by auto
qed


lemma idx-of-correct:
  assumes A: v∈⋃(set p-α)
  shows idx-of-impl v ≤ SPEC (λx. x=idx-of p-α v ∧ x<length B)
  using assms
  unfolding idx-of-impl-def
  apply (refine-rcg refine-vcg)
  apply (metis I-consistent in-set-conv-nth set-p-α-is-set-S)
  apply (erule S-idx-of-correct)
  apply (simp add: find-seg-impl find-seg-idx-of-correct)
  by (metis find-seg-correct(2) find-seg-impl)

lemma collapse-correct:
  assumes A: v∈⋃(set p-α)
  shows collapse-impl v ≤⇓GS-rel (SPEC (λr. r=collapse v α))
proof -
  {
    fix i
    assume i<length p-α
    hence ILEN: i<length B by (simp add: p-α-def)

    let ?SBIP' = (S, take (Suc i) B, I, P)
```

{
  have [*simp*]: *GS.seg-start ?SBIP′ i* = *seg-start i*
    by (*simp add*: *GS.seg-start-def*)

  have [*simp*]: *GS.seg-end ?SBIP′ i* = *seg-end* (*length B* − *1*)
    using *ILEN* **by** (*simp add*: *GS.seg-end-def min-absorb2*)

  {
    fix *j*
    assume *B*: *seg-start i* ≤ *j*   *j* < *seg-end* (*length B* − *Suc 0*)
    hence *j*<*length S* **using** *ILEN seg-end-bound*
    proof −
      note *B*(*2*)
      also from ‹*i*<*length B*› **have** (*length B* − *Suc 0*) < *length B* **by** *auto*
      from *seg-end-bound*[*OF this*]
      have *seg-end* (*length B* − *Suc 0*) ≤ *length S* **.**
      finally show *?thesis* **.**
    qed

    have *i* ≤ *find-seg j* ∧ *find-seg j* < *length B*
      ∧ *seg-start* (*find-seg j*) ≤ *j* ∧ *j* < *seg-end* (*find-seg j*)
    proof (*intro conjI*)
      show *i*≤*find-seg j*
        by (*metis le-trans not-less B*(*1*) *find-seg-bounds*(*2*)
          seg-end-less-start ILEN ‹*j* < *length S*›)
    qed (*simp-all add*: *find-seg-bounds*[*OF* ‹*j*<*length S*›])
  } **note** *AUX1* = *this*

  {
    fix *Q* **and** *j*::*nat*
    assume *Q j*
    hence ∃ *i*. *S*!*j* = *S*!*i* ∧ *Q i*
      by *blast*
  } **note** *AUX-ex-conj-SeqSI* = *this*

  have *GS.seg ?SBIP′ i* = ⋃ (*seg* ‘ {*i*..<*length B*})
    unfolding *GS.seg-def*[*abs-def*]
    apply *simp*
    apply (*rule*)
    apply (*auto dest*!: *AUX1*) []

    apply (*auto*
      simp: *seg-start-def seg-end-def*
      split: *if-split-asm*
      intro!: *AUX-ex-conj-SeqSI*
)

```
    apply (metis diff-diff-cancel le-diff-conv le-eq-less-or-eq
      lessI trans-le-add1
      distinct-sorted-mono[OF B-sorted B-distinct, of i])

    apply (metis diff-diff-cancel le-diff-conv le-eq-less-or-eq
      trans-le-add1 distinct-sorted-mono[OF B-sorted B-distinct, of i])

    apply (metis (opaque-lifting, no-types) Suc-lessD Suc-lessI less-trans-Suc
      B-in-bound')
    done
  } note AUX2 = this

  from ILEN have GS.p-α (S, take (Suc i) B, I, P) = collapse-aux p-α i
    unfolding GS.p-α-def collapse-aux-def
    apply (simp add: min-absorb2 drop-map)
    apply (rule conjI)
    apply (auto
      simp: GS.seg-def[abs-def] GS.seg-start-def GS.seg-end-def take-map) []

    apply (simp add: AUX2)
    done
} note AUX1 = this

from A obtain i where [simp]: I v = Some (STACK i)
  using I-consistent set-p-α-is-set-S
  by (auto simp: in-set-conv-nth)

{
  have (collapse-aux p-α (idx-of p-α v), D-α, pE-α) =
    GS.α (S, take (Suc (idx-of p-α v)) B, I, P)
  unfolding GS.α-def
  using idx-of-props[OF p-α-disjoint-sym A]
  by (simp add: AUX1)
} note ABS=this

{
  have GS-invar (S, take (Suc (idx-of p-α v)) B, I, P)
    apply unfold-locales
    apply simp-all

    using B-in-bound B-sorted B-distinct
    apply (auto simp: sorted-wrt-take dest: in-set-takeD) [3]

    using B0 S-distinct apply auto [2]

    using I-consistent apply simp

    using P-sorted P-distinct P-bound apply auto [3]
    done
```

} **note** *INV=this*

show *?thesis*
unfolding *collapse-impl-def*
apply (*refine-rcg SPEC-refine refine-vcg order-trans*[*OF idx-of-correct*])
apply *fact*
apply *simp*
apply (*simp add*: *collapse-def* α*-def find-seg-impl GS-rel-def*)
apply (*rule brI*)
apply (*rule ABS*)
apply (*rule INV*)
done
qed

end

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

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

lemma *ext-def*: $f{\equiv}g \Longrightarrow f\ x \equiv g\ x$ **by** *auto*

context *fr-graph* **begin**
definition *push-impl v s* $\equiv$ *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* $(S,B,I,P) \equiv (S@[v],\ B@[length\ S],\ I(v{\mapsto}STACK\ (length\ S)),$
$if\ E``\{v\}=\{\}\ then\ P\ else\ P@[(length\ S,E``\{v\})])$
unfolding *push-impl-def GS.push-impl-def GS.P-def GS.S-def*
by (*auto simp*: *Let-def*)

lemma *GS-*α*-split*:
$GS.\alpha\ s = (p,D,pE) \longleftrightarrow (p=GS.p\text{-}\alpha\ s \wedge D=GS.D\text{-}\alpha\ s \wedge pE=GS.pE\text{-}\alpha\ s)$
$(p,D,pE) = GS.\alpha\ s \longleftrightarrow (p=GS.p\text{-}\alpha\ s \wedge D=GS.D\text{-}\alpha\ s \wedge pE=GS.pE\text{-}\alpha\ s)$
by (*auto simp add*: *GS.*α*-def*)

lemma *push-refine*:
assumes *A*: $(s,(p,D,pE)){\in}GS\text{-}rel$ $\quad (v,v'){\in}Id$
assumes *B*: $v{\notin}\bigcup(set\ p)$ $\quad v{\notin}D$
shows (*push-impl v s*, *push v'* $(p,D,pE)){\in}GS\text{-}rel$
proof –
from *A* **have** [*simp*]: $p=GS.p\text{-}\alpha\ s \wedge D=GS.D\text{-}\alpha\ s \wedge pE=GS.pE\text{-}\alpha\ s$ $\quad v'=v$
and *INV*: *GS-invar s*
by (*auto simp add*: *GS-rel-def br-def GS-*α*-split*)

from *INV B* **show** *?thesis*

by (*auto*
simp: *GS-rel-def br-def GS-invar.push-correct push-impl-def push-def*)
qed

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

lemma *pop-refine*:
assumes *A*: $(s,(p,D,pE)) \in GS\text{-}rel$
assumes *B*: $p \neq []$ $pE \cap last\ p \times UNIV = \{\}$
shows $pop\text{-}impl\ s \leq \Downarrow GS\text{-}rel\ (RETURN\ (pop\ (p,D,pE)))$
proof −
from *A* **have** [*simp*]: $p=GS.p\text{-}\alpha\ s \wedge D=GS.D\text{-}\alpha\ s \wedge pE=GS.pE\text{-}\alpha\ s$
and *INV*: *GS-invar s*
by (*auto simp add*: *GS-rel-def br-def GS-α-split*)

show *?thesis*
unfolding *pop-impl-def*[*abs-def*] *pop-def*
apply (*rule order-trans*[*OF GS-invar.pop-correct*])
using *INV B*
apply (*simp-all add*: *Un-commute RETURN-def*)
done
qed

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

definition *collapse-impl v s* ≡ *GS.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*: $(s,(p,D,pE)) \in GS\text{-}rel$ $(v,v') \in Id$
assumes *B*: $v' \in \bigcup (set\ p)$
shows $collapse\text{-}impl\ v\ s \leq \Downarrow GS\text{-}rel\ (RETURN\ (collapse\ v'\ (p,D,pE)))$
proof −
from *A* **have** [*simp*]: $p=GS.p\text{-}\alpha\ s \wedge D=GS.D\text{-}\alpha\ s \wedge pE=GS.pE\text{-}\alpha\ s$ $v'=v$
and *INV*: *GS-invar s*
by (*auto simp add*: *GS-rel-def br-def GS-α-split*)

show *?thesis*
unfolding *collapse-impl-def*[*abs-def*]
apply (*rule order-trans*[*OF GS-invar.collapse-correct*])
using *INV B* **by** (*simp-all add*: *GS.α-def RETURN-def*)
qed

definition *select-edge-impl s* ≡ *GS.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*))∈*GS-rel*
  assumes *NE*: $p \neq []$
  shows *select-edge-impl s* ≤ ⇓(*Id* $\times_r$ *GS-rel*) (*select-edge* (*p*,*D*,*pE*))
proof −
  from *A* **have** [*simp*]: *p*=*GS.p-α s* ∧ *D*=*GS.D-α s* ∧ *pE*=*GS.pE-α s*
    and *INV*: *GS-invar s*
    by (*auto simp add*: *GS-rel-def br-def GS-α-split*)

  from *INV NE* **show** *?thesis*
    unfolding *select-edge-impl-def*
    using *GS-invar.sel-rem-last-correct*[*OF INV*] *NE*
    by (*simp*)
qed

definition *initial-impl v0 I* ≡ *GS-initial-impl I v0* (*E*‘‘{*v0*})

lemma *initial-refine*:
  ⟦*v0*∉*D0*; (*I*,*D0*)∈*oGS-rel*; (*v0i*,*v0*)∈*Id*⟧
  ⟹ (*initial-impl v0i I*,*initial v0 D0*)∈*GS-rel*
  unfolding *initial-impl-def GS-rel-def br-def*
  apply (*simp-all add*: *GS-initial-correct*)
  apply (*auto simp*: *initial-def*)
  done

definition *path-is-empty-impl s* ≡ *GS.S s* = []
lemma *path-is-empty-refine*:
  GS-invar s ⟹ *path-is-empty-impl s* ⟷ *GS.p-α s*=[]
  unfolding *path-is-empty-impl-def GS.p-α-def GS-invar.empty-eq*
  by *auto*

definition (**in** *GS*) *is-on-stack-impl v*
  ≡ *case I v of Some* (*STACK* -) ⇒ *True* | - ⇒ *False*

lemma (**in** *GS-invar*) *is-on-stack-impl-correct*:
  shows *is-on-stack-impl v* ⟷ *v*∈⋃(*set p-α*)
  unfolding *is-on-stack-impl-def*
  using *I-consistent*[*of v*]
  apply (*force*
    simp: *set-p-α-is-set-S in-set-conv-nth*
    split: *option.split node-state.split*)
  done

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

lemma *is-on-stack-refine*:
⟦ *GS-invar s* ⟧ ⟹ *is-on-stack-impl v s* ⟷ *v*∈⋃(*set* (*GS.p-α s*))
unfolding *is-on-stack-impl-def GS-rel-def br-def*
by (*simp add*: *GS-invar.is-on-stack-impl-correct*)

definition (**in** *GS*) *is-done-impl v*
≡ *case I v of Some DONE* ⇒ *True* | - ⇒ *False*

lemma (**in** *GS-invar*) *is-done-impl-correct*:
shows *is-done-impl v* ⟷ *v*∈*D-α*
unfolding *is-done-impl-def D-α-def*
apply (*auto split*: *option.split node-state.split*)
done

definition *is-done-oimpl v I* ≡ *case I v of Some DONE* ⇒ *True* | - ⇒ *False*

definition *is-done-impl v s* ≡ *GS.is-done-impl s v*

lemma *is-done-orefine*:
⟦ *oGS-invar s* ⟧ ⟹ *is-done-oimpl v s* ⟷ *v*∈*oGS-α s*
unfolding *is-done-oimpl-def oGS-rel-def br-def*
by (*auto*
simp: *oGS-invar-def oGS-α-def*
split: *option.splits node-state.split*)

lemma *is-done-refine*:
⟦ *GS-invar s* ⟧ ⟹ *is-done-impl v s* ⟷ *v*∈*GS.D-α s*
unfolding *is-done-impl-def GS-rel-def br-def*
by (*simp add*: *GS-invar.is-done-impl-correct*)

lemma *oinitial-refine*: (*Map.empty*, {}) ∈ *oGS-rel*
by (*auto simp*: *oGS-rel-def br-def oGS-α-def oGS-invar-def*)

end

1.4.4 Refined Skeleton Algorithm

context *fr-graph* **begin**

lemma *I-to-outer*:
assumes ((*S*, *B*, *I*, *P*), ([], *D*, {})) ∈ *GS-rel*
shows (*I*,*D*)∈*oGS-rel*

```
    using assms
   unfolding GS-rel-def oGS-rel-def br-def oGS-α-def GS.α-def GS.D-α-def GS-invar-def
oGS-invar-def
    apply (auto simp: GS.p-α-def)
    done


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

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

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

Correctness Theorem **lemma** *skeleton-impl ≤ ⇓oGS-rel skeleton*

```
    using [[goals-limit = 1]]
    unfolding skeleton-impl-def skeleton-def
```

apply (*refine-rcg*
bind-refine′
select-edge-refine push-refine
pop-refine
collapse-refine
initial-refine
oinitial-refine
inj-on-id
)
using [[*goals-limit* = *5*]]
apply *refine-dref-type*

apply (*vc-solve* (*nopre*) *solve*: *asm-rl I-to-outer*
simp: *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
)

done

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*
unfolding *skeleton-impl-def skeleton-def*
by (*refine-rcg skeleton-refines*, *refine-dref-type*)
(*vc-solve* (*nopre*) *solve*: *asm-rl I-to-outer simp*: *skeleton-refine-simps*)

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* $\equiv$ *SPEC* $(\lambda l.$
$distinct\ l \wedge \bigcup(set\ l) = E^* \text{``} V0 \wedge (\forall\ U{\in}set\ l.\ is\text{-}scc\ E\ U)$
$\wedge\ (\forall\ i\ j.\ i{<}j \wedge j{<}length\ l \longrightarrow l!j \times l!i \cap E^* = \{\})\ )$
end

1.7 Extended Invariant

locale *cscc-invar-ext* = *fr-graph G*
for $G :: ('v,'more)$ *graph-rec-scheme* +
fixes $l :: 'v$ *set list* **and** $D :: 'v$ *set*
assumes *l-is-D*: $\bigcup(set\ l) = D$ — The output contains all done CNodes
assumes *l-scc*: $set\ l \subseteq Collect\ (is\text{-}scc\ E)$ — The output contains only SCCs
assumes *l-no-fwd*: $\bigwedge i\ j.\ [\![i{<}j;\ j{<}length\ l]\!] \Longrightarrow l!j \times l!i \cap E^* = \{\}$
— The output contains no forward edges
begin
lemma *l-no-empty*: $\{\}{\notin}set\ l$ **using** *l-scc* **by** (*auto simp*: *in-set-conv-decomp*)
end

locale *cscc-outer-invar-loc* = *outer-invar-loc G it D* + *cscc-invar-ext G l D*
for $G :: ('v,'more)$ *graph-rec-scheme* **and** *it l D*
begin
lemma *locale-this*: *cscc-outer-invar-loc G it l D* **by** *unfold-locales*
lemma *abs-outer-this*: *outer-invar-loc G it D* **by** *unfold-locales*
end

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

context *fr-graph*
begin
definition *cscc-outer-invar* $\equiv \lambda it\ (l,D).$ *cscc-outer-invar-loc G it l D*
definition *cscc-invar* $\equiv \lambda v0\ D0\ (l,p,D,pE).$ *cscc-invar-loc G v0 D0 l p D pE*
end

1.8 Definition of the SCC-Algorithm

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

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

            ASSERT (p≠[]);
            case vo of
              Some v ⇒ do {
                if v ∈ ⋃(set p) then do {
                  — Collapse
                  RETURN (l,collapse v (p,D,pE))
                } else if v∉D then do {
                  — Edge to new node. Append to path
                  RETURN (l,push v (p,D,pE))
                } else RETURN (l,p,D,pE)
              }
            | None ⇒ do {
                — No more outgoing edges from current node on path
                ASSERT (pE ∩ last p × UNIV = {});
                let V = last p;
                let (p,D,pE) = pop (p,D,pE);
                let l = V#l;
                RETURN (l,p,D,pE)
              }
          }) s;
          ASSERT (p=[] ∧ pE={});
          RETURN (l,D)
        } else
          RETURN (l,D0)
      }) so;
      RETURN l
    }
end
```

1.9 Preservation of Invariant Extension

```
context cscc-invar-ext
```

```
begin
  lemma l-disjoint:
    assumes A: i<j   j<length l
    shows l!i ∩ l!j = {}
  proof (rule disjointI)
    fix u
    assume u∈l!i   u∈l!j
    with l-no-fwd A show False by auto
  qed

  corollary l-distinct: distinct l
    using l-disjoint l-no-empty
    by (metis distinct-conv-nth inf-idem linorder-cases nth-mem)
end

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

  lemma cscc-invarI[intro?]:
    assumes invar v0 D0 PDPE
    assumes invar v0 D0 PDPE ⟹ cscc-invar-part (l,PDPE)
    shows cscc-invar v0 D0 (l,PDPE)
    using assms
    unfolding initial-def cscc-invar-def invar-def
    apply (simp split: prod.split-asm)
    apply intro-locales
    apply (simp add: invar-loc-def)
    apply (simp add: cscc-invar-part-def cscc-invar-ext-def)
    done

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

  lemma cscc-outer-invarI[intro?]:
    assumes outer-invar it D
    assumes outer-invar it D ⟹ cscc-invar-ext G l D
    shows cscc-outer-invar it (l,D)
    using assms
    unfolding initial-def cscc-outer-invar-def outer-invar-def
    apply (simp split: prod.split-asm)
    apply intro-locales
    apply (simp add: outer-invar-loc-def)
    apply (simp add: cscc-invar-ext-def)
    done

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

proof −
  from *INV* **interpret** *cscc-outer-invar-loc G it l D0*
    unfolding *cscc-outer-invar-def* **by** *simp*

  show *?thesis*
    unfolding *cscc-invar-part-def initial-def*
    apply *simp*
    by *unfold-locales*
qed

lemma *cscc-invar-pop*:
  assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
  assumes *invar v0 D0* (*pop* (*p*,*D*,*pE*))
  assumes *NE*[*simp*]: *p*≠[]
  assumes *NO′*: *pE* ∩ (*last p* × *UNIV*) = {}
  shows *cscc-invar-part* (*last p* # *l*, *pop* (*p*,*D*,*pE*))
proof −
  from *INV* **interpret** *cscc-invar-loc G v0 D0 l p D pE*
    unfolding *cscc-invar-def* **by** *simp*

  have *AUX-l-scc*: *is-scc E* (*last p*)
    unfolding *is-scc-pointwise*
  proof *safe*
    {
      assume *last p* = {} **thus** *False*
        using *p-no-empty* **by** (*cases p rule*: *rev-cases*) *auto*
    }

    fix *u v*
    assume *u*∈*last p*   *v*∈*last p*
    with *p-sc*[*of last p*] **have** (*u*,*v*) ∈ (*lvE* ∩ *last p* × *last p*)* **by** *auto*
    with *lvE-ss-E* **show** (*u*,*v*)∈(*E* ∩ *last p* × *last p*)*
      by (*metis Int-mono equalityE rtrancl-mono-mp*)

    fix *u′*
    assume *u′*∉*last p*   (*u*,*u′*)∈*E**   (*u′*,*v*)∈*E**

    from ‹*u′*∉*last p*› ‹*u*∈*last p*› ‹(*u*,*u′*)∈*E**›
      and *rtrancl-reachable-induct*[*OF order-refl lastp-un-D-closed*[*OF NE NO′*]]
    have *u′*∈*D* **by** *auto*
    with ‹(*u′*,*v*)∈*E**› **and** *rtrancl-reachable-induct*[*OF order-refl D-closed*]
    have *v*∈*D* **by** *auto*
    with ‹*v*∈*last p*› *p-not-D* **show** *False* **by** (*cases p rule*: *rev-cases*) *auto*
  qed

  {
    fix *i j*
    assume *A*: *i*<*j*   *j*<*Suc* (*length l*)
    have *l* ! (*j* − *Suc 0*) × (*last p* # *l*) ! *i* ∩ *E** = {}

proof (*rule disjointI*, *safe*)
fix *u v*
assume $(u,\ v) \in E^*$ $\quad u \in l\ !\ (j - Suc\ 0)$ $\quad v \in (last\ p\ \#\ l)\ !\ i$
from ‹$u \in l\ !\ (j - Suc\ 0)$› *A* **have** $u \in \bigcup (set\ l)$
by (*metis Ex-list-of-length Suc-pred UnionI length-greater-0-conv less-nat-zero-code not-less-eq nth-mem*)
with *l-is-D* **have** $u \in D$ **by** *simp*
with *rtrancl-reachable-induct*[*OF order-refl D-closed*] ‹$(u,v) \in E^*$›
have $v \in D$ **by** *auto*

show *False* **proof** *cases*
assume $i=0$ **hence** $v \in last\ p$ **using** ‹$v \in (last\ p\ \#\ l)\ !\ i$› **by** *simp*
with *p-not-D* ‹$v \in D$› **show** *False* **by** (*cases p rule*: *rev-cases*) *auto*
next
assume $i \neq 0$ **with** ‹$v \in (last\ p\ \#\ l)\ !\ i$› **have** $v \in l!(i - 1)$ **by** *auto*
with *l-no-fwd*[*of* $i - 1$ $\quad j - 1$]
and ‹$u \in l\ !\ (j - Suc\ 0)$› ‹$(u,\ v) \in E^*$› ‹$i \neq 0$› *A*
show *False* **by** *fastforce*
qed
qed
} **note** *AUX-l-no-fwd* = *this*

show *?thesis*
unfolding *cscc-invar-part-def pop-def* **apply** *simp*
apply *unfold-locales*
apply *clarsimp-all*
using *l-is-D* **apply** *auto* []

using *l-scc AUX-l-scc* **apply** *auto* []

apply (*rule AUX-l-no-fwd*, *assumption+*) []
done
qed

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

lemma *cscc-invar-unchanged*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*p'*,*D*,*pE'*)
using *INV* **unfolding** *cscc-invar-def cscc-invar-part-def cscc-invar-loc-def*
by *simp*

corollary *cscc-invar-collapse*:
assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*collapse v* (*p'*,*D*,*pE'*))
unfolding *collapse-def*
by (*simp add*: *cscc-invar-unchanged*[*OF INV*])

corollary *cscc-invar-push*:

assumes *INV*: *cscc-invar v0 D0* (*l*,*p*,*D*,*pE*)
shows *cscc-invar-part* (*l*,*push v* (*p'*,*D*,*pE'*))
unfolding *push-def*
by (*simp add*: *cscc-invar-unchanged*[*OF INV*])

lemma *cscc-outer-invar-initial*: *cscc-invar-ext G* [] {}
by *unfold-locales auto*

lemma *cscc-invar-outer-newnode*:
assumes *A*: *v0*∉*D0* *v0*∈*it*
assumes *OINV*: *cscc-outer-invar it* (*l*,*D0*)
assumes *INV*: *cscc-invar v0 D0* (*l'*,[],*D'*,*pE*)
shows *cscc-invar-ext G l' D'*
proof −
from *OINV* **interpret** *cscc-outer-invar-loc G it l D0*
unfolding *cscc-outer-invar-def* **by** *simp*
from *INV* **interpret** *inv*: *cscc-invar-loc G v0 D0 l'* [] *D' pE*
unfolding *cscc-invar-def* **by** *simp*

show *?thesis*
by *unfold-locales*

qed

lemma *cscc-invar-outer-Dnode*:
assumes *cscc-outer-invar it* (*l*, *D*)
shows *cscc-invar-ext G l D*
using *assms*
by (*simp add*: *cscc-outer-invar-def cscc-outer-invar-loc-def*)

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

On termination, the invariant implies the specification

lemma *cscc-finI*:
assumes *INV*: *cscc-outer-invar* {} (*l*,*D*)
shows *fin-l-is-scc*: $[\![U{\in}set\ l]\!] \Longrightarrow is\text{-}scc\ E\ U$
and *fin-l-distinct*: *distinct l*
and *fin-l-is-reachable*: $\bigcup(set\ l) = E^*$ `` $V0$
and *fin-l-no-fwd*: $[\![i<j;\ j<length\ l]\!] \Longrightarrow l!j \times l!i \cap E^* = \{\}$
proof −
from *INV* **interpret** *cscc-outer-invar-loc G* {} *l D*
unfolding *cscc-outer-invar-def* **by** *simp*

show $[\![U{\in}set\ l]\!] \Longrightarrow is\text{-}scc\ E\ U$ **using** *l-scc* **by** *auto*

show *distinct l* **by** (*rule l-distinct*)

show $\bigcup$(*set l*) = E^* `` *V0*
using *fin-outer-D-is-reachable*[*OF outer-invar-this*] *l-is-D*
by *auto*

show ⟦*i*<*j*; *j*<*length l*⟧ ⟹ *l*!*j* ×*l*!*i* ∩ E^* = {}
by (*rule l-no-fwd*)

qed

end

1.10 Main Correctness Proof

context *fr-graph*
begin
lemma *invar-from-cscc-invarI*: *cscc-invar v0 D0* (*L*,*PDPE*) ⟹ *invar v0 D0 PDPE*
unfolding *cscc-invar-def invar-def*
apply (*simp split*: *prod.splits*)
unfolding *cscc-invar-loc-def* **by** *simp*

lemma *outer-invar-from-cscc-invarI*:
cscc-outer-invar it (*L*,*D*) ⟹*outer-invar it D*
unfolding *cscc-outer-invar-def outer-invar-def*
apply (*simp split*: *prod.splits*)
unfolding *cscc-outer-invar-loc-def* **by** *simp*

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

theorem *compute-SCC-correct*: *compute-SCC* ≤ *compute-SCC-spec*
proof −
note [[*goals-limit* = *2*]]
note [*simp del*] = *Union-iff*

show *?thesis*
unfolding *compute-SCC-def compute-SCC-spec-def select-edge-def select-def*
apply (*refine-rcg*
WHILEIT-rule[**where** *R*=*inv-image* (*abs-wf-rel v0*) *snd* **for** *v0*]
refine-vcg
)

apply (*vc-solve*
rec: *cscc-invarI cscc-outer-invarI*
solve: *cscc-invar-preserve cscc-finI*
intro: *invar-from-cscc-invarI outer-invar-from-cscc-invarI*
dest!: *sym*[*of pop A* **for** *A*]

```
    simp: pE-fin'[OF invar-from-cscc-invarI] finite-V0
  )
  apply auto
  done
qed
```

Simple proof, for presentation

```
context
  notes [refine]=refine-vcg
  notes [[goals-limit = 1]]
begin
  theorem compute-SCC ≤ compute-SCC-spec
    unfolding compute-SCC-def compute-SCC-spec-def select-edge-def select-def
    by (refine-rcg
      WHILEIT-rule[where R=inv-image (abs-wf-rel v0) snd for v0])
    (vc-solve
      rec: cscc-invarI cscc-outer-invarI solve: cscc-invar-preserve cscc-finI
      intro: invar-from-cscc-invarI outer-invar-from-cscc-invarI
      dest!: sym[of pop A for A]
      simp: pE-fin'[OF invar-from-cscc-invarI] finite-V0, auto)
end

end
```

1.11 Refinement to Gabow's Data Structure

```
context GS begin
  definition seg-set-impl l u ≡ do {
    (-,res) ← WHILET
      (λ(l,-). l<u)
      (λ(l,res). do {
        ASSERT (l<length S);
        let x = S!l;
        ASSERT (x∉res);
        RETURN (Suc l,insert x res)
      })
      (l,{});

    RETURN res
  }

  lemma seg-set-impl-aux:
    fixes l u
    shows ⟦l<u; u≤length S; distinct S⟧ ⟹ seg-set-impl l u
    ≤ SPEC (λr. r = {S!j | j. l≤j ∧ j<u})
    unfolding seg-set-impl-def
    apply (refine-rcg
      WHILET-rule[where
        I=λ(l',res). res = {S!j | j. l≤j ∧ j<l'} ∧ l≤l' ∧ l'≤u
```

and *R=measure* $(\lambda(l',\text{-}).\ u-l')$
]
refine-vcg)

apply (*auto simp*: *less-Suc-eq nth-eq-iff-index-eq*)
done

lemma (**in** *GS-invar*) *seg-set-impl-correct*:
assumes *i<length B*
shows *seg-set-impl* (*seg-start i*) (*seg-end i*) $\leq$ *SPEC* ($\lambda r.$ *r=p-α!i*)
apply (*refine-rcg order-trans*[*OF seg-set-impl-aux*] *refine-vcg*)

using *assms*
apply (*simp-all add*: *seg-start-less-end seg-end-bound S-distinct*) [*3*]

apply (*auto simp*: *p-α-def assms seg-def*) []
done

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

lemma (**in** *GS-invar*) *last-seg-impl-correct*:
assumes *p-α* ≠ []
shows *last-seg-impl* $\leq$ *SPEC* ($\lambda r.$ *r=last p-α*)
unfolding *last-seg-impl-def*
apply (*refine-rcg order-trans*[*OF seg-set-impl-correct*] *refine-vcg*)
using *assms* **apply** (*auto simp add*: *p-α-def last-conv-nth*)
done

end

context *fr-graph*
begin

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

lemma *last-seg-impl-refine*:
assumes *A*: (*s*,(*p*,*D*,*pE*))∈*GS-rel*
assumes *NE*: *p*≠[]
shows *last-seg-impl s* $\leq$ ⇓*Id* (*RETURN* (*last p*))
proof −

from *A* **have**
[*simp*]: *p=GS.p-α s* ∧ *D=GS.D-α s* ∧ *pE=GS.pE-α s*
and *INV*: *GS-invar s*
by (*auto simp add*: *GS-rel-def br-def GS-α-split*)

show *?thesis*
unfolding *last-seg-impl-def*[*abs-def*]
apply (*rule order-trans*[*OF GS-invar.last-seg-impl-correct*])
using *INV NE*
apply (*simp-all*)
done
qed

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

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

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

```
        RETURN (l,I)
      } else RETURN (l,I0)
    }) so;
    stat-stop-nres;
    RETURN l
  }

lemma compute-SCC-impl-refine: compute-SCC-impl ≤ ⇓Id compute-SCC
proof −
  note [refine2] = bind-Let-refine2[OF last-seg-impl-refine]

  have [refine2]: ⋀s′ p D pE l′ l v′ v. ⟦
    (s′,(p,D,pE))∈GS-rel;
    (l′,l)∈Id;
    (v′,v)∈Id;
    v∈⋃(set p)
  ⟧ ⟹ do { s′←collapse-impl v′ s′; RETURN (l′,s′) }
    ≤ ⇓(Id ×r GS-rel) (RETURN (l,collapse v (p,D,pE)))
    apply (refine-rcg order-trans[OF collapse-refine] refine-vcg)
    apply assumption+
    apply (auto simp add: pw-le-iff refine-pw-simps)
    done

  note [[goals-limit = 1]]
  show ?thesis
    unfolding compute-SCC-impl-def compute-SCC-def
    apply (refine-rcg
      bind-refine′
      select-edge-refine push-refine
      pop-refine

      initial-refine
      oinitial-refine

      prod-relI IdI
      inj-on-id
    )

    apply refine-dref-type
    apply (vc-solve (nopre) solve: asm-rl I-to-outer
      simp: 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
    )

    done
qed

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* $E\ U0\ P \equiv do$ {
$ASSERT\ (finite\ U0)$;
$ASSERT\ (finite\ (E^* \text{``}U0))$;
$SPEC\ (\lambda p.\ case\ p\ of$
$Some\ (p,v) \Rightarrow \exists\, u0{\in}U0.\ path\ E\ u0\ p\ v \wedge P\ v \wedge (\forall\, v{\in}set\ p.\ \neg\ P\ v)$
$|\ None \Rightarrow \forall\, u0{\in}U0.\ \forall\, v{\in}E^* \text{``}\{u0\}.\ \neg P\ v)$
}

lemma *find-path-ex-rule*:
assumes $finite\ U0$
assumes $finite\ (E^* \text{``}U0)$
assumes $\exists\, v{\in}E^* \text{``}U0.\ P\ v$
shows $find\text{-}path\ E\ U0\ P \leq SPEC\ (\lambda r.$
$\exists\, p\ v.\ r = Some\ (p,v) \wedge P\ v \wedge (\forall\, v{\in}set\ p.\ \neg P\ v) \wedge (\exists\, u0{\in}U0.\ path\ E\ u0\ p\ v))$
unfolding *find-path-def*
using *assms*
by (*fastforce split*: *option.splits*)

1.13.1 Nontrivial Paths

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

lemma (**in** −) *find-path1-ex-rule*:
assumes $finite\ (E^* \text{``}\{u0\})$
assumes $\exists\, v{\in}E^+ \text{``}\{u0\}.\ P\ v$
shows $find\text{-}path1\ E\ u0\ P \leq SPEC\ (\lambda r.$
$\exists\, p\ v.\ r = Some\ (p,v) \wedge p{\neq}[] \wedge P\ v \wedge path\ E\ u0\ p\ v)$
unfolding *find-path1-def*
using *assms*
by (*fastforce split*: *option.splits*)

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* $((g\text{-}E\ G)^* \text{``}\ g\text{-}V0\ G)$
  shows *igb-fr-graph G*
proof −
  interpret *igb-graph G* **by** *fact*
  show *?thesis* **using** *assms*(*2*) **by** *unfold-locales*
qed

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 $\equiv$ ($\exists$ *pr pl*.
        $Vr \subseteq E^* \text{``}V0 \wedge Vl \subseteq E^* \text{``}V0$ — Only reachable nodes are covered
      $\wedge$ *set pr*$\subseteq$*Vr* $\wedge$ *set pl*$\subseteq$*Vl* — The paths are inside the specified sets
      $\wedge\ Vl\times Vl \subseteq (E \cap Vl\times Vl)^*$ — *Vl* is mutually connected
      $\wedge\ Vl\times Vl \cap E \neq \{\}$ — *Vl* is non-trivial
      $\wedge$ *is-lasso-prpl* (*pr*,*pl*)) — Paths form a lasso

  definition *find-ce-spec* :: $('Q\ set \times 'Q\ set)$ *option nres* **where**
    find-ce-spec $\equiv$ *SPEC* (λr. *case r of*
      None $\Rightarrow$ ($\forall$ *prpl*. $\neg$*is-lasso-prpl prpl*)
    | *Some* (*Vr*, *Vl*) $\Rightarrow$ *ce-correct Vr Vl*
)

definition *find-lasso-spec* :: $('Q\ list \times 'Q\ list)$ *option nres* **where**
find-lasso-spec $\equiv$ *SPEC* $(\lambda r.$ *case r of*
None $\Rightarrow (\forall\, prpl.\ \neg$*is-lasso-prpl prpl*$)$
$|$ *Some prpl* $\Rightarrow$ *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* $|$ *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*
$|$ *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*
by *unfold-locales*
lemma *invar-loc-this*: *invar-loc G v0 D0 p D pE* **by** *unfold-locales*
lemma *eas-gba-graph-this*: *igb-graph G* **by** *unfold-locales*
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 × 'Q set) option nres where
  find-ce ≡ do {
   let D = {};
   (brk,-)←FOREACHci fgl-outer-invar V0
    (λ(brk,-). brk=None)
    (λv0 (brk,D0). do {
     if v0∉D0 then do {
      let s = (None,initial v0 D0);

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

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

          ASSERT (p≠[]);

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

end

1.18 Invariant Preservation

context *igb-fr-graph*
begin

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

lemma *fgl-outer-invarI*[*intro?*]:
⟦
brk=None ⟹ *outer-invar it D*;
⟦*brk=None* ⟹ *outer-invar it D*⟧ ⟹ *fgl-outer-invar-ext it* (*brk*,*D*)⟧
⟹ *fgl-outer-invar it* (*brk*,*D*)
unfolding *outer-invar-def fgl-outer-invar-ext-def fgl-outer-invar-def*
apply (*auto split*: *prod.splits option.splits*)
done

lemma *fgl-invarI*[*intro?*]:
⟦ *invar v0 D0 PDPE*;
invar v0 D0 PDPE ⟹ *fgl-invar-part* (*B*,*PDPE*)⟧
⟹ *fgl-invar v0 D0* (*B*,*PDPE*)
unfolding *invar-def fgl-invar-part-def fgl-invar-def*
apply (*simp split*: *prod.split-asm*)
apply *intro-locales*
apply (*simp add*: *invar-loc-def*)
apply *assumption*
done

lemma *fgl-invar-initial*:
assumes *OINV*: *fgl-outer-invar it* (*None*,*D0*)
assumes *A*: *v0*∈*it*   *v0*∉*D0*
shows *fgl-invar-part* (*None*, *initial v0 D0*)
proof −
from *OINV* **interpret** *outer-invar-loc G it D0*
by (*simp add*: *fgl-outer-invar-def outer-invar-def*)

from *OINV* **have** *no-acc*: *no-acc-over D0*
by (*simp add*: *fgl-outer-invar-def fgl-outer-invar-ext-def*)

{
fix *v pl*

assume *pl* ≠ [] **and** *P*: *path* (*vE* [{*v0*}] *D0* (*E* ∩ {*v0*} × *UNIV*)) *v pl v*
hence *1*: *v*∈*D0*
by (*cases pl*) (*auto simp*: *path-cons-conv vE-def touched-def*)
have *2*: *path E v pl v* **using** *path-mono*[*OF vE-ss-E P*] **.**

note *1 2*
} **note** *AUX1=this*

show *?thesis*
unfolding *fgl-invar-part-def*
apply (*simp split*: *prod.splits add*: *initial-def*)
apply *unfold-locales*
using ‹*v0*∉*D0*›
using *AUX1 no-acc* **unfolding** *no-acc-over-def* **apply** *blast*
by *simp*
qed

lemma *fgl-invar-pop*:
assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
assumes *INV'*: *invar v0 D0* (*pop* (*p*,*D*,*pE*))
assumes *NE*[*simp*]: *p*≠[]
assumes *NO'*: *pE* ∩ *last p* × *UNIV* = {}
shows *fgl-invar-part* (*None*, *pop* (*p*,*D*,*pE*))
proof −
from *INV* **interpret** *fgl-invar-loc G v0 D0 None p D pE*
by (*simp add*: *fgl-invar-def*)

show *?thesis*
apply (*unfold fgl-invar-part-def pop-def*)
apply (*simp split*: *prod.splits*)
apply *unfold-locales*
unfolding *vE-pop*[*OF NE*]

using *no-acc* **apply** *auto* []
apply *simp*
done
qed

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

The last collapsed node on the path contains states from all accepting sets. As it is strongly connected and reachable, we get a counter-example. Here, we explicitely construct the lasso.

let *?Er* = *E* ∩ (⋃(*set* (*butlast p*)) × *UNIV*)

We choose a node in the last Cnode, that is reachable only using former Cnodes.

obtain *w* **where** (*v0*,*w*)∈*?Er** *w*∈*last p*

```
proof cases
  assume length p = 1
  hence v0∈last p
    using root-v0
    by (cases p) auto
  thus thesis by (auto intro: that)
next
  assume length p≠1
  hence length p > 1 by (cases p) auto
  hence Suc (length p − 2) < length p by auto
  from p-connected'[OF this] obtain u v where
    UIP: u∈p!(length p − 2) and VIP: v∈p!(length p − 1) and (u,v)∈lvE
    using ‹length p > 1› by auto
  from root-v0 have V0IP: v0∈p!0 by (cases p) auto

  from VIP have v∈last p by (cases p rule: rev-cases) auto

  from pathI[OF V0IP UIP] ‹length p > 1› have
    (v0,u)∈(lvE ∩ ⋃(set (butlast p)) × ⋃(set (butlast p)))*
    (is - ∈ ...*)
    by (simp add: path-seg-butlast)
  also have ... ⊆ ?Er using lvE-ss-E by auto
  finally (rtrancl-mono-mp[rotated]) have (v0,u)∈?Er* .
 also note ‹(u,v)∈lvE› UIP hence (u,v)∈?Er using lvE-ss-E ‹length p > 1›
    apply (auto simp: Bex-def in-set-conv-nth)
    by (metis One-nat-def Suc-lessE ‹Suc (length p − 2) < length p›
      diff-Suc-1 length-butlast nth-butlast)
  finally show ?thesis by (rule that) fact
qed
then obtain pr where
  P-REACH: path E v0 pr w and
  R-SS: set pr ⊆ ⋃(set (butlast p))
  apply −
  apply (erule rtrancl-is-path)
  apply (frule path-nodes-edges)
  apply (auto
    dest!: order-trans[OF - image-Int-subset]
    dest: path-mono[of - E, rotated])
  done

have [simp]: last p = p!(length p − 1) by (cases p rule: rev-cases) auto
```

From that node, we construct a lasso by inductively appending a path for each accepting set

```
{
  fix na
  assume na-def: na = num-acc

  have ∃ pl. pl≠[]
```

$\wedge$ *path* $(lvE \cap last\ p \times last\ p)$ *w pl w*
$\wedge$ $(\forall\, i<num\text{-}acc.\ \exists\, q\in set\ pl.\ i\in acc\ q)$
using *ACC*
unfolding *na-def*[*symmetric*]
proof (*induction na*)
case *0*

from *NONTRIV* **obtain** *u v*
where $(u,v)\in lvE \cap last\ p \times last\ p$ $\quad u\in last\ p$ $\quad v\in last\ p$
by *auto*
from *cnode-connectedI* ‹$w\in last\ p$› ‹$u\in last\ p$›
have $(w,u)\in(lvE \cap last\ p \times last\ p)^*$
by *auto*
also note ‹$(u,v)\in lvE \cap last\ p \times last\ p$›
also (*rtrancl-into-trancl1*) **from** *cnode-connectedI* ‹$v\in last\ p$› ‹$w\in last\ p$›
have $(v,w)\in(lvE \cap last\ p \times last\ p)^*$
by *auto*
finally obtain *pl* **where** $pl\neq[]$ $\quad$ *path* $(lvE \cap last\ p \times last\ p)$ *w pl w*
by (*rule trancl-is-path*)
thus *?case* **by** *auto*
next
case (*Suc n*)
from *Suc.prems* **have** $\forall\, i<n.\ \exists\, q\in last\ p.\ i\in acc\ q$ **by** *auto*
with *Suc.IH* **obtain** *pl* **where** *IH*:
$pl\neq[]$
path $(lvE \cap last\ p \times last\ p)$ *w pl w*
$\forall\, i<n.\ \exists\, q\in set\ pl.\ i\in acc\ q$
by *blast*

from *Suc.prems* **obtain** *v* **where** $v\in last\ p$ **and** $n\in acc\ v$ **by** *auto*
from *cnode-connectedI* ‹$w\in last\ p$› ‹$v\in last\ p$›
have $(w,v)\in(lvE \cap last\ p \times last\ p)^*$ **by** *auto*
then obtain *pl1* **where** *P1*: *path* $(lvE \cap last\ p \times last\ p)$ *w pl1 v*
by (*rule rtrancl-is-path*)
also from *cnode-connectedI* ‹$w\in last\ p$› ‹$v\in last\ p$›
have $(v,w)\in(lvE \cap last\ p \times last\ p)^*$ **by** *auto*
then obtain *pl2* **where** *P2*: *path* $(lvE \cap last\ p \times last\ p)$ *v pl2 w*
by (*rule rtrancl-is-path*)
also (*path-conc*) **note** *IH*(*2*)
finally (*path-conc*) **have**
P: *path* $(lvE \cap last\ p \times last\ p)$ *w* (*pl1*@*pl2*@*pl*) *w*
by *simp*
moreover from *IH*(*1*) **have** *pl1*@*pl2*@*pl* $\neq$ [] **by** *simp*
moreover have $\forall\, i'<n.\ \exists\, q\in set$ (*pl1*@*pl2*@*pl*). $i'\in acc\ q$ **using** *IH*(*3*) **by** *auto*
moreover have $v\in set$ (*pl1*@*pl2*@*pl*) **using** *P1 P2 P IH*(*1*)
apply (*cases pl2*, *simp-all add*: *path-cons-conv path-conc-conv*)
apply (*cases pl*, *simp-all add*: *path-cons-conv*)
apply (*cases pl1*, *simp-all add*: *path-cons-conv*)

done
with ‹$n \in acc\ v$› **have** $\exists\, q \in set\ (pl1 @ pl2 @ pl).\ n \in acc\ q$ **by** *auto*
ultimately show *?case*
apply (*intro exI conjI*)
apply *assumption+*
apply (*auto elim*: *less-SucE*)
done
qed
}
then obtain *pl* **where** *pl*: $pl \neq []$ $\quad$ *path* $(lvE \cap last\ p \times last\ p)\ w\ pl\ w$
$\forall\, i < num\text{-}acc.\ \exists\, q \in set\ pl.\ i \in acc\ q$ **by** *blast*
hence *path E w pl w* **and** *L-SS*: $set\ pl \subseteq last\ p$
apply −
apply (*frule path-mono*[*of - E*, *rotated*])
using *lvE-ss-E*
apply *auto* [*2*]

apply (*drule path-nodes-edges*)
apply (*drule order-trans*[*OF - image-Int-subset*])
apply *auto* []
done

have *LASSO*: *is-lasso-prpl* (*pr*,*pl*)
unfolding *is-lasso-prpl-def is-lasso-prpl-pre-def*
using ‹*path E w pl w*› *P-REACH pl* **by** *auto*

from *p-sc* **have** $last\ p \times last\ p \subseteq (lvE \cap last\ p \times last\ p)^*$ **by** *auto*
with *lvE-ss-E* **have** *VL-CLOSED*: $last\ p \times last\ p \subseteq (E \cap last\ p \times last\ p)^*$
apply (*erule-tac order-trans*)
apply (*rule rtrancl-mono*)
by *blast*

have *NONTRIV'*: $last\ p \times last\ p \cap E \neq \{\}$
by (*metis Int-commute NONTRIV disjoint-mono lvE-ss-E subset-refl*)

from *order-trans*[*OF path-touched touched-reachable*]
have *LP-REACH*: $last\ p \subseteq E^*$ ‘‘$V0$
and *BLP-REACH*: $\bigcup (set\ (butlast\ p)) \subseteq E^*$ ‘‘$V0$
apply −
apply (*cases p rule*: *rev-cases*)
apply *simp*
apply *auto* []

apply (*cases p rule*: *rev-cases*)
apply *simp*
apply *auto* []
done

show *?thesis*

apply (*simp add*: *fgl-invar-part-def*)
apply *unfold-locales*
apply *simp*

using *LASSO R-SS L-SS VL-CLOSED NONTRIV′ LP-REACH BLP-REACH*
unfolding *ce-correct-def*
apply *simp*
apply *blast*
done

qed

lemma *fgl-invar-collapse-ce*:
fixes *u v*
assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
defines $pE' \equiv pE - \{(u,v)\}$
assumes *CFMT*: $(p',D',pE'') = collapse\ v\ (p,D,pE')$
assumes *INV′*: *invar v0 D0* (p',D',pE'')
assumes *NE*[*simp*]: $p\neq[]$
assumes *E*: $(u,v)\in pE$ **and** $u\in last\ p$
assumes *BACK*: $v\in\bigcup(set\ p)$
assumes *ACC*: $\forall\, i<num\text{-}acc.\ \exists\, q\in last\ p'.\ i\in acc\ q$
defines *i-def*: $i \equiv idx\text{-}of\ p\ v$
shows *fgl-invar-part* (
Some $(\bigcup(set\ (butlast\ p')),\ last\ p')$,
collapse $v\ (p,D,pE')$)
proof −

from *CFMT* **have** *p′-def*: $p' = collapse\text{-}aux\ p\ i$ **and** [*simp*]: $D'=D \quad pE''=pE'$
by (*simp-all add*: *collapse-def i-def*)

from *INV* **interpret** *fgl-invar-loc G v0 D0 None p D pE*
by (*simp add*: *fgl-invar-def*)

from *idx-of-props*[*OF BACK*] **have** $i<length\ p$ **and** $v\in p!i$
by (*simp-all add*: *i-def*)

have $u\in last\ p'$
using ‹$u\in last\ p$› ‹$i<length\ p$›
unfolding *p′-def collapse-aux-def*
apply (*simp add*: *last-drop last-snoc*)
by (*metis Misc.last-in-set drop-eq-Nil last-drop not-le*)
moreover have $v\in last\ p'$
using ‹$v\in p!i$› ‹$i<length\ p$›
unfolding *p′-def collapse-aux-def*
by (*metis UnionI append-Nil Cons-nth-drop-Suc in-set-conv-decomp last-snoc*)
ultimately have $vE\ p'\ D\ pE' \cap last\ p' \times last\ p' \neq \{\}$
unfolding *p′-def pE′-def* **by** (*auto simp*: *E*)

have *p′≠[]* **by** (*simp add*: *p′-def collapse-aux-def*)

have [*simp*]: *collapse v* (*p*,*D*,*pE′*) = (*p′*,*D*,*pE′*)
unfolding *collapse-def p′-def i-def*
by *simp*

show *?thesis*
apply *simp*
apply (*rule fgl-invar-collapse-ce-aux*)
using *INV′* **apply** *simp*
apply *fact+*
done
qed

lemma *fgl-invar-collapse-nce*:
fixes *u v*
assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
defines *pE′* ≡ *pE* − {(*u*,*v*)}
assumes *CFMT*: (*p′*,*D′*,*pE′′*) = *collapse v* (*p*,*D*,*pE′*)
assumes *INV′*: *invar v0 D0* (*p′*,*D′*,*pE′′*)
assumes *NE*[*simp*]: *p≠[]*
assumes *E*: (*u*,*v*)∈*pE* **and** *u∈last p*
assumes *BACK*: *v*∈⋃(*set p*)
assumes *NACC*: *j<num-acc* ∀ *q∈last p′*. *j∉acc q*
defines *i* ≡ *idx-of p v*
shows *fgl-invar-part* (*None*, *collapse v* (*p*,*D*,*pE′*))
proof −
from *CFMT* **have** *p′-def*: *p′* = *collapse-aux p i* **and** [*simp*]: *D′=D* *pE′′=pE′*
by (*simp-all add*: *collapse-def i-def*)

have [*simp*]: *collapse v* (*p*,*D*,*pE′*) = (*p′*,*D*,*pE′*)
by (*simp add*: *collapse-def p′-def i-def*)

from *INV* **interpret** *fgl-invar-loc G v0 D0 None p D pE*
by (*simp add*: *fgl-invar-def*)

from *INV′* **interpret** *inv′*: *invar-loc G v0 D0 p′ D pE′* **by** (*simp add*: *invar-def*)

define *vE′* **where** *vE′* = *vE p′ D pE′*

have *vE′-alt*: *vE′* = *insert* (*u*,*v*) *lvE*
by (*simp add*: *vE′-def p′-def pE′-def E*)

from *idx-of-props*[*OF BACK*] **have** *i<length p* **and** *v∈p!i*
by (*simp-all add*: *i-def*)

have *u∈last p′*
using ‹*u∈last p*› ‹*i<length p*›
unfolding *p′-def collapse-aux-def*

apply (*simp add*: *last-drop last-snoc*)
by (*metis Misc.last-in-set drop-eq-Nil last-drop leD*)
moreover have $v \in last\ p'$
using ‹$v \in p!i$› ‹$i < length\ p$›
unfolding *p′-def collapse-aux-def*
by (*metis UnionI append-Nil Cons-nth-drop-Suc in-set-conv-decomp last-snoc*)
ultimately have $vE' \cap last\ p' \times last\ p' \neq \{\}$
unfolding *vE′-alt* **by** (*auto*)

have $p' \neq []$ **by** (*simp add*: *p′-def collapse-aux-def*)

{

We show that no visited strongly connected component contains states from all acceptance sets.

fix *w pl*

For this, we chose a non-trivial loop inside the visited edges

assume *P*: *path vE′ w pl w* **and** *NT*: $pl \neq []$

And show that there is one acceptance set disjoint with the nodes of the loop

have $\exists\, i < num\text{-}acc.\ \forall\, q \in set\ pl.\ i \notin acc\ q$
proof *cases*
assume $set\ pl \cap last\ p' = \{\}$
— Case: The loop is outside the last Cnode
with *path-restrict*[*OF P*] ‹$u \in last\ p'$› ‹$v \in last\ p'$› **have** *path lvE w pl w*
apply −
apply (*drule path-mono*[*of - lvE*, *rotated*])
unfolding *vE′-alt*
by *auto*
with *no-acc NT* **show** *?thesis* **by** *auto*
next
assume $set\ pl \cap last\ p' \neq \{\}$
— Case: The loop touches the last Cnode

Then, the loop must be completely inside the last CNode

from *inv′.loop-in-lastnode*[*folded vE′-def*, *OF P* ‹$p' \neq []$› *this*]
have $w \in last\ p'$ $set\ pl \subseteq last\ p'$ **.**
with *NACC* **show** *?thesis* **by** *blast*
qed
} **note** *AUX-no-acc* = *this*

show *?thesis*
apply (*simp add*: *fgl-invar-part-def*)
apply *unfold-locales*
using *AUX-no-acc*[*unfolded vE′-def*] **apply** *auto* []

apply *simp*
done

qed

lemma *collapse-ne*: $([],D',pE') \neq$ *collapse v* (p,D,pE)
  by (*simp add*: *collapse-def collapse-aux-def Let-def*)

lemma *fgl-invar-push*:
  assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
  assumes *BRK*[*simp*]: *brk*=*None*
  assumes *NE*[*simp*]: $p \neq []$
  assumes *E*: $(u,v) \in pE$ **and** *UIL*: $u \in last\ p$
  assumes *VNE*: $v \notin \bigcup (set\ p)$ $v \notin D$
  assumes *INV′*: *invar v0 D0* (*push v* $(p,D,pE - \{(u,v)\})$)
  shows *fgl-invar-part* (*None*, *push v* $(p,D,pE - \{(u,v)\})$)
proof −
  from *INV* **interpret** *fgl-invar-loc G v0 D0 None p D pE*
    by (*simp add*: *fgl-invar-def*)

  define *pE′* **where** $pE' = (pE - \{(u,v)\} \cup E \cap \{v\} \times UNIV)$

  have [*simp*]: *push v* $(p,D,pE - \{(u,v)\}) = (p@[\{v\}],D,pE')$
    by (*simp add*: *push-def pE′-def*)

  from *INV′* **interpret** *inv′*: *invar-loc G v0 D0* $(p@[\{v\}])$ *D* *pE′*
    by (*simp add*: *invar-def*)

  note *defs-fold* = *vE-push*[*OF E UIL VNE*, *folded pE′-def*]

  {

We show that there still is no loop that contains all accepting nodes. For this, we choose some loop.

    fix *w pl*
    assume *P*: *path* (*insert* (u,v) *lvE*) *w pl w* **and** [*simp*]: $pl \neq []$
    have $\exists i < num\text{-}acc.\ \forall q \in set\ pl.\ i \notin acc\ q$
    proof *cases*
      assume $v \in set\ pl$ — Case: The newly pushed last cnode is on the loop

Then the loop is entirely on the last cnode

      with *inv′.loop-in-lastnode*[*unfolded defs-fold*, *OF P*]
      have [*simp*]: $w=v$ **and** *SPL*: $set\ pl = \{v\}$ **by** *auto*

However, we then either have that the last cnode is contained in the last but one cnode, or that there is a visited edge inside the last cnode.

      from *P SPL* **have** $u=v \vee (v,v) \in lvE$
        apply (*cases pl*) **apply** (*auto simp*: *path-cons-conv*)
        apply (*case-tac list*)
        apply (*auto simp*: *path-cons-conv*)
        done

Both leads to a contradiction

hence *False* **proof**
assume $u=v$ — This is impossible, as u was on the original path, but v was not
with *UIL VNE* **show** *False* **by** *auto*
next
assume $(v,v)\in lvE$ — This is impossible, as all visited edges are from touched nodes, but v was untouched
with *vE-touched VNE* **show** *False* **unfolding** *touched-def* **by** *auto*
qed
thus *?thesis* **..**
next
assume *A*: $v\notin set\ pl$
— Case: The newly pushed last cnode is not on the loop

Then, the path lays inside the old visited edges

have *path lvE w pl w*
proof −
have $w\in set\ pl$ **using** *P* **by** (*cases pl*) (*auto simp*: *path-cons-conv*)
with *A* **show** *?thesis* **using** *path-restrict*[*OF P*]
apply −
apply (*drule path-mono*[*of - lvE*, *rotated*])
apply (*cases pl*, *auto*) []

apply *assumption*
done
qed

And thus, the proposition follows from the invariant on the old state

with *no-acc* **show** *?thesis*
apply *simp*
using ‹$pl\neq[]$›
by *blast*
qed
} **note** *AUX-no-acc* = *this*

show *?thesis*
unfolding *fgl-invar-part-def*
apply *simp*
apply *unfold-locales*
unfolding *defs-fold*

using *AUX-no-acc* **apply** *auto* []

apply *simp*
done
qed

lemma *fgl-invar-skip*:

assumes *INV*: *fgl-invar v0 D0* (*None*,*p*,*D*,*pE*)
assumes *BRK*[*simp*]: *brk*=*None*
assumes *NE*[*simp*]: *p*≠[]
assumes *E*: (*u*,*v*)∈*pE* **and** *UIL*: *u*∈*last p*
assumes *VID*: *v*∈*D*
assumes *INV'*: *invar v0 D0* (*p*, *D*, (*pE* − {(*u*,*v*)}))
shows *fgl-invar-part* (*None*, *p*, *D*, (*pE* − {(*u*,*v*)}))
proof −
from *INV* **interpret** *fgl-invar-loc G v0 D0 None p D pE*
by (*simp add*: *fgl-invar-def*)
from *INV'* **interpret** *inv'*: *invar-loc G v0 D0 p D* (*pE* − {(*u*,*v*)})
by (*simp add*: *invar-def*)

{

We show that there still is no loop that contains all accepting nodes. For this, we choose some loop.

fix *w pl*
assume *P*: *path* (*insert* (*u*,*v*) *lvE*) *w pl w* **and** [*simp*]: *pl*≠[]
from *P* **have** ∃ *i*<*num-acc*. ∀ *q*∈*set pl*. *i*∉*acc q*
proof (*cases rule*: *path-edge-rev-cases*)
case *no-use* — Case: The loop does not use the new edge

The proposition follows from the invariant for the old state

with *no-acc* **show** *?thesis*
apply *simp*
using ‹*pl*≠[]›
by *blast*
next
case (*split p1 p2*) — Case: The loop uses the new edge

As done is closed under transitions, the nodes of the edge have already been visited

from *split*(*2*) *D-closed-vE-rtrancl*
have *WID*: *w*∈*D*
using *VID* **by** (*auto dest*!: *path-is-rtrancl*)
from *split*(*1*) *WID D-closed-vE-rtrancl* **have** *u*∈*D*
apply (*cases rule*: *path-edge-cases*)
apply (*auto dest*!: *path-is-rtrancl*)
done

Which is a contradition to the assumptions

with *UIL p-not-D* **have** *False* **by** (*cases p rule*: *rev-cases*) *auto*
thus *?thesis* **..**
qed
} **note** *AUX-no-acc* = *this*

show *?thesis*
apply (*simp add*: *fgl-invar-part-def*)

```
    apply unfold-locales
    unfolding vE-remove[OF NE E]

    using AUX-no-acc apply auto []

    apply simp
    done

qed

lemma fgl-outer-invar-initial:
  outer-invar V0 {} ⟹ fgl-outer-invar-ext V0 (None, {})
  unfolding fgl-outer-invar-ext-def
  apply (simp add: no-acc-over-def)
  done

lemma fgl-outer-invar-brk:
  assumes INV: fgl-invar v0 D0 (Some (Vr,Vl),p,D,pE)
  shows fgl-outer-invar-ext anyIt (Some (Vr,Vl), anyD)
proof −
  from INV interpret fgl-invar-loc G v0 D0 Some (Vr,Vl) p D pE
    by (simp add: fgl-invar-def)

  from acc show ?thesis by (simp add: fgl-outer-invar-ext-def)
qed

lemma fgl-outer-invar-newnode-nobrk:
  assumes A: v0∉D0    v0∈it
  assumes OINV: fgl-outer-invar it (None,D0)
  assumes INV: fgl-invar v0 D0 (None,[],D',pE)
  shows fgl-outer-invar-ext (it−{v0}) (None,D')
proof −
  from OINV interpret outer-invar-loc G it D0
    unfolding fgl-outer-invar-def outer-invar-def by simp

  from INV interpret inv: fgl-invar-loc G v0 D0 None [] D' pE
    unfolding fgl-invar-def by simp

  from inv.pE-fin have [simp]: pE = {} by simp

  { fix v pl
    assume A: v∈D'    path E v pl v
    have path (E ∩ D' × UNIV) v pl v
      apply (rule path-mono[OF - path-restrict-closed[OF inv.D-closed A]])
      by auto
  } note AUX1=this

  show ?thesis
    unfolding fgl-outer-invar-ext-def
```

```
    apply simp
    using inv.no-acc AUX1
    apply (auto simp add: vE-def touched-def no-acc-over-def) []
    done
qed

lemma fgl-outer-invar-newnode:
  assumes A: v0∉D0    v0∈it
  assumes OINV: fgl-outer-invar it (None,D0)
  assumes INV: fgl-invar v0 D0 (brk,p,D′,pE)
  assumes CASES: (∃ Vr Vl. brk = Some (Vr, Vl)) ∨ p = []
  shows fgl-outer-invar-ext (it−{v0}) (brk,D′)
  using CASES
  apply (elim disjE1)
  using fgl-outer-invar-brk[of v0 D0 - - p D′ pE] INV
  apply −
  apply (auto, assumption) []
  using fgl-outer-invar-newnode-nobrk[OF A] OINV INV apply auto []
  done

lemma fgl-outer-invar-Dnode:
  assumes fgl-outer-invar it (None, D)    v∈D
  shows fgl-outer-invar-ext (it − {v}) (None, D)
  using assms
  by (auto simp: fgl-outer-invar-def fgl-outer-invar-ext-def)


lemma fgl-fin-no-lasso:
  assumes A: fgl-outer-invar {} (None, D)
  assumes B: is-lasso-prpl prpl
  shows False
proof −
  obtain pr pl where [simp]: prpl = (pr,pl) by (cases prpl)
  from A have NA: no-acc-over D
    by (simp add: fgl-outer-invar-def fgl-outer-invar-ext-def)

  from A have outer-invar {} D by (simp add: fgl-outer-invar-def)
  with fin-outer-D-is-reachable have [simp]: D=E* ‘‘V0 by simp

  from NA B show False
    apply (simp add: no-acc-over-def is-lasso-prpl-def is-lasso-prpl-pre-def)
    apply clarsimp
    apply (blast dest: path-is-rtrancl)
    done
qed

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

using *A* **by** (*simp add*: *fgl-outer-invar-def fgl-outer-invar-ext-def*)

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

end

1.19 Main Correctness Proof

context *igb-fr-graph*
begin
lemma *outer-invar-from-fgl-invarI*:
fgl-outer-invar it (*None*,*D*) $\Longrightarrow$ *outer-invar it D*
unfolding *fgl-outer-invar-def outer-invar-def*
by (*simp split*: *prod.splits*)

lemma *invar-from-fgl-invarI*: *fgl-invar v0 D0* (*B*,*PDPE*) $\Longrightarrow$ *invar v0 D0 PDPE*
unfolding *fgl-invar-def invar-def*
apply (*simp split*: *prod.splits*)
unfolding *fgl-invar-loc-def* **by** *simp*

theorem *find-ce-correct*: *find-ce* $\leq$ *find-ce-spec*
proof −
note [*simp del*] = *Union-iff*

show *?thesis*
unfolding *find-ce-def find-ce-spec-def select-edge-def select-def*
apply (*refine-rcg*
WHILEIT-rule[**where** *R*=*inv-image* (*abs-wf-rel v0*) *snd* **for** *v0*]
refine-vcg
)

using [[*goals-limit* = *5*]]

apply (*vc-solve*
rec: *fgl-invarI fgl-outer-invarI*
intro: *invar-from-fgl-invarI outer-invar-from-fgl-invarI*
dest!: *sym*[*of collapse a b* **for** *a b*]
simp: *collapse-ne*
simp: *pE-fin'*[*OF invar-from-fgl-invarI*] *finite-V0*
solve: *invar-preserve*
solve: *asm-rl*[*of* - $\cap$ - = {}]
solve: *fgl-invar-preserve*)
done

```
  qed
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 ≡ do {
    ce ← find-ce;
    RETURN (ce = None)
    }

  theorem abs-is-empty-correct:
    abs-is-empty ≤ SPEC (λres. res ⟷ (∀ r. ¬is-acc-run r))
    unfolding abs-is-empty-def
    apply (refine-rcg refine-vcg
      order-trans[OF find-ce-correct, unfolded find-ce-spec-def])
    unfolding ce-correct-def
    using lasso-accepted accepted-lasso
    apply (clarsimp split: option.splits)
    apply (metis is-lasso-prpl-of-lasso surj-pair)
    by (metis is-lasso-prpl-conv)

  definition abs-is-empty-ce ≡ do {
    ce ← find-ce;
    case ce of
      None ⇒ RETURN None
    | Some (Vr, Vl) ⇒ do {
        ASSERT (∃ pr pl. set pr ⊆ Vr ∧ set pl ⊆ Vl ∧ Vl × Vl ⊆ (E ∩ Vl×Vl)*
          ∧ is-lasso-prpl (pr,pl));
        (pr,pl) ← SPEC (λ(pr,pl).
           set pr ⊆ Vr
          ∧ set pl ⊆ Vl
          ∧ Vl × Vl ⊆ (E ∩ Vl×Vl)*
          ∧ is-lasso-prpl (pr,pl));
        RETURN (Some (pr,pl))
      }
    }

  theorem abs-is-empty-ce-correct: abs-is-empty-ce ≤ SPEC (λres. case res of
      None ⇒ (∀ r. ¬is-acc-run r)
    | Some (pr,pl) ⇒ is-acc-run (pr⌢pl^ω)
    )
    unfolding abs-is-empty-ce-def
    apply (refine-rcg refine-vcg
      order-trans[OF find-ce-correct, unfolded find-ce-spec-def])

    apply (clarsimp-all simp: ce-correct-def)
```

using *accepted-lasso finite-reachableE-V0* **apply** (*metis is-lasso-prpl-of-lasso surj-pair*)
apply *blast*
apply (*simp add*: *lasso-prpl-acc-run*)
done

end

1.21 Refinement

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

1.21.1 Addition of Explicit Accepting Sets

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

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

context *igb-fr-graph*
begin

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

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

lemma *gstate-rel-sv*[*relator-props*,*simp*,*intro*!]: *single-valued gstate-rel*
by (*simp add*: *gstate-rel-def*)

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

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

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

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

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

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

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

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

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

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

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

```
              if last a = {0..<num-acc} then
                RETURN (Some (⋃(set (butlast p)),last p),a,p,D,pE)
              else
                RETURN (None,a,p,D,pE)
            } else if v∉D then do {
              — Edge to new node. Append to path
              RETURN (None,gpush v (a,p,D,pE))
            } else RETURN (None,a,p,D,pE)
          }
        | None ⇒ do {
            — No more outgoing edges from current node on path
            ASSERT (pE ∩ last p × UNIV = {});
            RETURN (None,gpop (a,p,D,pE))
          }
      }) s;
      ASSERT (brk=None ⟶ (p=[] ∧ pE={}));
      RETURN (gto-outer brk (a,p,D,pE))
    } else RETURN s0
  }) os;
  RETURN (goBrk os)
}
```

lemma *gcollapse-refine*:
  ⟦$(v',v)\in Id$; $(s',s)\in gstate\text{-}rel$⟧
    $\Longrightarrow$ $(gcollapse\ v'\ s', collapse\ v\ s)\in gstate\text{-}rel$
  unfolding *gcollapse-def collapse-def collapse-aux-def gcollapse-aux-def*
  apply (*simp add*: *gstate-rel-def br-def Let-def*)
  unfolding *gstate-invar-def*[*abs-def*]
  apply (*auto split*: *prod.splits simp*: *take-map drop-map*)
  done

lemma *gpush-refine*:
  ⟦$(v',v)\in Id$; $(s',s)\in gstate\text{-}rel$⟧ $\Longrightarrow$ $(gpush\ v'\ s', push\ v\ s)\in gstate\text{-}rel$
  unfolding *gpush-def push-def*
  apply (*simp add*: *gstate-rel-def br-def*)
  unfolding *gstate-invar-def*[*abs-def*]
  apply (*auto split*: *prod.splits*)
  done

lemma *gpop-refine*:
  ⟦$(s',s)\in gstate\text{-}rel$⟧ $\Longrightarrow$ $(gpop\ s', pop\ s)\in gstate\text{-}rel$
  unfolding *gpop-def pop-def*
  apply (*simp add*: *gstate-rel-def br-def*)
  unfolding *gstate-invar-def*[*abs-def*]
  apply (*auto split*: *prod.splits simp*: *map-butlast*)
  done

lemma *ginitial-refine*:
  $(ginitial\ x\ (None,\ b),\ initial\ x\ b)\in gstate\text{-}rel$

unfolding *ginitial-def gstate-rel-def br-def gstate-invar-def initial-def*
by *auto*

lemma *oinitial-b-refine*: $((None,\{\}),(None,\{\}))\in Id\times_r Id$ **by** *simp*

lemma *gselect-edge-refine*: $[\![(s',s)\in gstate\text{-}rel]\!] \Longrightarrow gselect\text{-}edge\ s'$
$\leq\Downarrow(\langle Id\rangle option\text{-}rel \times_r gstate\text{-}rel)\ (select\text{-}edge\ s)$
unfolding *gselect-edge-def select-edge-def*
apply (*simp add*: *pw-le-iff refine-pw-simps prod-rel-sv*
split: *prod.splits option.splits*)

apply (*auto simp*: *gstate-rel-def br-def gstate-invar-def*)
done

lemma *last-acc-impl*:
assumes $p\neq[]$
assumes $((a',p',D',pE'),(p,D,pE))\in gstate\text{-}rel$
shows $(last\ a' = \{0..<num\text{-}acc\}) = (\forall\, i<num\text{-}acc.\ \exists\, q\in last\ p.\ i\in acc\ q)$
using *assms acc-bound* **unfolding** *gstate-rel-def br-def gstate-invar-def*
by (*auto simp*: *last-map*)

lemma *fglr-aux1*:
assumes V: $(v',v)\in Id$ **and** S: $(s',s)\in gstate\text{-}rel$
and P: $\bigwedge a'\ p'\ D'\ pE'\ p\ D\ pE.\ ((a',p',D',pE'),(p,D,pE))\in gstate\text{-}rel$
$\Longrightarrow f'\ a'\ p'\ D'\ pE' \leq\Downarrow R\ (f\ p\ D\ pE)$
shows $(let\ (a',p',D',pE') = gcollapse\ v'\ s'\ in\ f'\ a'\ p'\ D'\ pE')$
$\leq \Downarrow R\ (let\ (p,D,pE) = collapse\ v\ s\ in\ f\ p\ D\ pE)$
apply (*auto split*: *prod.splits*)
apply (*rule P*)
using *gcollapse-refine*[*OF V S*]
apply *simp*
done

lemma *gstate-invar-empty*:
$gstate\text{-}invar\ (a,[],D,pE) \Longrightarrow a=[]$
$gstate\text{-}invar\ ([],p,D,pE) \Longrightarrow p=[]$
by (*auto simp add*: *gstate-invar-def*)

lemma *find-ce-refine*: $gfind\text{-}ce \leq\Downarrow Id\ find\text{-}ce$
unfolding *gfind-ce-def find-ce-def*
unfolding *goinitial-def go-is-no-brk-def*[*abs-def*] *goD-def goBrk-def*
gto-outer-def
using [[*goals-limit* = *1*]]
apply (*refine-rcg*
gselect-edge-refine prod-relI[*OF IdI gpop-refine*]
prod-relI[*OF IdI gpush-refine*]
fglr-aux1 last-acc-impl oinitial-b-refine
inj-on-id
)

apply *refine-dref-type*
apply (*simp-all add*: *ginitial-refine*)
apply (*vc-solve* (*nopre*)
solve: *asm-rl*
simp: *gstate-rel-def br-def gstate-invar-empty*)
done
end

1.21.2 Refinement to Gabow's Data Structure

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

lemma *Un-set-drop-impl-correct*:
Un-set-drop-impl i A ≤ *SPEC* (λ*r*. *r*=$\bigcup$(*set* (*drop i A*)))
unfolding *Un-set-drop-impl-def*
apply (*refine-rcg*
WHILET-rule[**where** *I*=λ(*i'*,*res*). *res*=$\bigcup$(*set* ((*drop i* (*take i' A*)))) ∧ *i*≤*i'*
and *R*=*measure* (λ(*i'*,-). *length A* − *i'*)]
refine-vcg)
apply (*auto simp*: *take-Suc-conv-app-nth*)
done

schematic-goal *Un-set-drop-code-aux*:
assumes [*autoref-rules*]: (*es-impl*,{})∈⟨*R*⟩*Rs*
assumes [*autoref-rules*]: (*un-impl*,(∪))∈⟨*R*⟩*Rs*→⟨*R*⟩*Rs*→⟨*R*⟩*Rs*
shows (*?c*,*Un-set-drop-impl*)∈*nat-rel* → ⟨⟨*R*⟩*Rs*⟩*as-rel* → ⟨⟨*R*⟩*Rs*⟩*nres-rel*
unfolding *Un-set-drop-impl-def*[*abs-def*]
apply (*autoref* (*trace*, *keep-goal*))
done
concrete-definition *Un-set-drop-code* **uses** *Un-set-drop-code-aux*

schematic-goal *Un-set-drop-tr-aux*:
RETURN ?c ≤ *Un-set-drop-code es-impl un-impl i A*
unfolding *Un-set-drop-code-def*
by *refine-transfer*
concrete-definition *Un-set-drop-tr* **for** *es-impl un-impl i A*
uses *Un-set-drop-tr-aux*

lemma *Un-set-drop-autoref*[*autoref-rules*]:
  assumes *GEN-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 (λi *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*
  apply (*intro fun-relI nres-relI*)
  apply (*rule order-trans*[*OF Un-set-drop-tr.refine*])
  using *Un-set-drop-code.refine*[*of es-impl Rs R un-impl*,
    param-fo, *THEN nres-relD*]
  using *assms*
  by *simp*

Actual Refinement **type-synonym** *'Q gGS* = *nat set list* × *'Q GS*

type-synonym *'Q goGS* = *'Q ce* × *'Q oGS*

context *igb-graph*
begin

definition *gGS-invar* :: *'Q gGS* ⇒ *bool*
  where *gGS-invar s* ≡
  let (*a*,*S*,*B*,*I*,*P*) = *s in*
    GS-invar (*S*,*B*,*I*,*P*)
    ∧ *length a* = *length B*
    ∧ $\bigcup$(*set a*) ⊆ {*0*..<*num-acc*}

definition *gGS-α* :: *'Q gGS* ⇒ *'Q abs-gstate*
  where *gGS-α s* ≡ *let* (*a*,*s*)=*s in* (*a*,*GS.α s*)

definition *gGS-rel* ≡ *br gGS-α gGS-invar*

lemma *gGS-rel-sv*[*relator-props*,*intro*!,*simp*]: *single-valued gGS-rel*
  unfolding *gGS-rel-def* **by** *auto*

definition *goGS-invar* :: *'Q goGS* ⇒ *bool* **where**
  goGS-invar s ≡ *let* (*brk*,*ogs*)=*s in brk*=*None* ⟶ *oGS-invar ogs*

definition *goGS-α s* ≡ *let* (*brk*,*ogs*)=*s in* (*brk*,*oGS-α ogs*)

definition *goGS-rel* ≡ *br goGS-α goGS-invar*

lemma *goGS-rel-sv*[*relator-props*,*intro*!,*simp*]: *single-valued goGS-rel*
  unfolding *goGS-rel-def* **by** *auto*

end

context *igb-fr-graph*
begin
lemma *gGS-relE*:
assumes $(s',(a,p,D,pE))\in gGS\text{-}rel$
obtains S' B' I' P' **where** $s'=(a,S',B',I',P')$
and $((S',B',I',P'),(p,D,pE))\in GS\text{-}rel$
and *length a* = *length* B'
and $\bigcup(set\ a) \subseteq \{0..<num\text{-}acc\}$
using *assms*
apply (*cases* s')
apply (*simp add*: *gGS-rel-def br-def gGS-α-def GS.α-def*)
apply (*rule that*)
apply (*simp only*:)
apply (*auto simp*: *GS-rel-def br-def gGS-invar-def GS.α-def*)
done

definition *goinitial-impl* :: *'Q goGS*
where *goinitial-impl* ≡ (*None*,*Map.empty*)
lemma *goinitial-impl-refine*: (*goinitial-impl*,*goinitial*)∈*goGS-rel*
by (*auto*
simp: *goinitial-impl-def goinitial-def goGS-rel-def br-def*
simp: *goGS-α-def goGS-invar-def oGS-α-def oGS-invar-def*)

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

lemma *gto-outer-refine*:
assumes *A*: *brk* = *None* ⟶ (*p*=[] ∧ *pE*={})
assumes *B*: (*s*, (*A*,*p*, *D*, *pE*)) ∈ *gGS-rel*
assumes *C*: (*brk'*,*brk*)∈*Id*
shows (*gto-outer-impl brk' s*,*gto-outer brk* (*A*,*p*,*D*,*pE*))∈*goGS-rel*
proof (*cases s*)
fix *A S B I P*
assume [*simp*]: *s*=(*A*,*S*,*B*,*I*,*P*)
show *?thesis*
using *C*
apply (*cases brk*)
using *assms I-to-outer*[*of S B I P D*]
apply (*auto*
simp: *goGS-rel-def br-def goGS-α-def gto-outer-def*
gto-outer-impl-def goGS-invar-def
simp: *gGS-rel-def oGS-rel-def GS-rel-def gGS-α-def gGS-invar-def*
GS.α-def) []

using *B* **apply** (*auto*
simp: *gto-outer-def gto-outer-impl-def*
simp: *br-def goGS-rel-def goGS-invar-def goGS-α-def oGS-α-def*
simp: *gGS-rel-def gGS-α-def GS.α-def GS.D-α-def*

```
    )
    done
qed

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


lemma gpush-impl-refine:
  assumes B: (s',(a,p,D,pE))∈gGS-rel
  assumes A: (v',v)∈Id
  assumes PRE: v' ∉ ⋃(set p)     v' ∉ D
  shows (gpush-impl v' s', gpush v (a,p,D,pE))∈gGS-rel
proof -
  from B obtain S' B' I' P' where [simp]: s'=(a,S',B',I',P')
    and OSR: ((S',B',I',P'),(p,D,pE))∈GS-rel and L: length a = length B'
    and R: ⋃(set a) ⊆ {0..<num-acc}
    by (rule gGS-relE)
  {
    fix S B I P S' B' I' P'
    assume push-impl v (S, B, I, P) = (S', B', I', P')
    hence length B' = Suc (length B)
      by (auto simp add: push-impl-def GS.push-impl-def Let-def)
  } note AUX1=this

  from push-refine[OF OSR A PRE] A L acc-bound R show ?thesis
    unfolding gpush-impl-def gpush-def
      gGS-rel-def gGS-invar-def gGS-α-def GS-rel-def br-def
    apply (auto dest: AUX1)
    done
qed

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

lemma gpop-impl-refine:
  assumes A: (s',(a,p,D,pE))∈gGS-rel
  assumes PRE: p ≠ []     pE ∩ last p × UNIV = {}
  shows gpop-impl s' ≤ ⇓gGS-rel (RETURN (gpop (a,p,D,pE)))
proof -
  from A obtain S' B' I' P' where [simp]: s'=(a,S',B',I',P')
    and OSR: ((S',B',I',P'),(p,D,pE))∈GS-rel and L: length a = length B'
    and R: ⋃(set a) ⊆ {0..<num-acc}
```

by (*rule gGS-relE*)

from *PRE OSR* **have** [*simp*]: $a \neq []$ **using** *L*
by (*auto simp add*: *GS-rel-def br-def GS.α-def GS.p-α-def*)

{
fix $S\ B\ I\ P\ S'\ B'\ I'\ P'$
assume *nofail* (*pop-impl* ((S, B, I, P)::$'a$ *GS*))
inres (*pop-impl* ((S, B, I, P)::$'a$ *GS*)) (S', B', I', P')
hence *length* B' = *length* B − *Suc 0*
apply (*simp add*: *pop-impl-def GS.pop-impl-def Let-def refine-pw-simps*)
apply *auto*
done
} **note** *AUX1=this*

from *A L* **show** *?thesis*
unfolding *gpop-impl-def gpop-def gGS-rel-def gGS-α-def br-def*
apply (*simp add*: *Let-def*)
using *pop-refine*[*OF OSR PRE*]
apply (*simp add*: *pw-le-iff refine-pw-simps split*: *prod.splits*)
unfolding *gGS-rel-def gGS-invar-def gGS-α-def GS-rel-def GS.α-def br-def*
apply (*auto dest!*: *AUX1 in-set-butlastD iff*: *Sup-le-iff*)
done
qed

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

thm *select-edge-refine*
lemma *gselect-edge-impl-refine*:
assumes *A*: (s', *a*, *p*, *D*, *pE*) ∈ *gGS-rel*
assumes *PRE*: $p \neq []$
shows *gselect-edge-impl* $s' \leq \Downarrow(Id \times_r gGS\text{-}rel)$ (*gselect-edge* (*a*, *p*, *D*, *pE*))
proof −
from *A* **obtain** $S'\ B'\ I'\ P'$ **where** [*simp*]: $s'=(a,S',B',I',P')$
and *OSR*: $((S',B',I',P'),(p,D,pE)) \in GS\text{-}rel$ **and** *L*: *length a* = *length* B'
and *R*: $\bigcup(set\ a) \subseteq \{0..<num\text{-}acc\}$
by (*rule gGS-relE*)

{
fix $S\ B\ I\ P\ S'\ B'\ I'\ P'$ *vo*
assume *nofail* (*select-edge-impl* ((S, B, I, P)::$'a$ *GS*))
inres (*select-edge-impl* ((S, B, I, P)::$'a$ *GS*)) (*vo*, (S', B', I', P'))

hence *length B′ = length B*
apply (*simp add*: *select-edge-impl-def GS.sel-rem-last-def refine-pw-simps*
split: *if-split-asm prod.splits*)
apply *auto*
done
} **note** *AUX1=this*

show *?thesis*
using *select-edge-refine*[*OF OSR PRE*]
unfolding *gselect-edge-impl-def gselect-edge-def*
apply (*simp add*: *refine-pw-simps pw-le-iff prod-rel-sv*)

unfolding *gGS-rel-def br-def gGS-α-def gGS-invar-def GS-rel-def GS.α-def*
apply (*simp split*: *prod.splits*)
apply *clarsimp*
using *R*
apply (*auto simp*: *L dest*: *AUX1*)
done
qed

term *GS.idx-of-impl*

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

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

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

unfolding *gcollapse-def gcollapse-aux-def collapse-def collapse-aux-def*
by *auto*

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-impl-aux* $v'\ s'$
$\leq \Downarrow gGS\text{-}rel\ (RETURN\ (gcollapse\ v\ (a, p, D, pE)))$
proof −
note [*simp*] = *Let-def*

from *A* **obtain** $S'\ B'\ I'\ P'$ **where** [*simp*]: $s'=(a,S',B',I',P')$
and *OSR*: $((S',B',I',P'),(p,D,pE)) \in GS\text{-}rel$ **and** *L*: $length\ a = length\ B'$
and *R*: $\bigcup(set\ a) \subseteq \{0..<num\text{-}acc\}$
by (*rule gGS-relE*)

from *B* **have** [*simp*]: $v'=v$ **by** *simp*

from *OSR* **have** [*simp*]: $GS.p\text{-}\alpha\ (S',B',I',P') = p$
by (*simp add*: *GS-rel-def br-def GS.α-def*)

from *OSR PRE* **have** *PRE'*: $v \in \bigcup(set\ (GS.p\text{-}\alpha\ (S',B',I',P')))$
by (*simp add*: *GS-rel-def br-def GS.α-def*)

from *OSR* **have** *GS-invar*: *GS-invar* (S',B',I',P')
by (*simp add*: *GS-rel-def br-def*)

term *GS.B*
{
fix *s*
assume $collapse\ v\ (p, D, pE) = (GS.p\text{-}\alpha\ s,\ GS.D\text{-}\alpha\ s,\ GS.pE\text{-}\alpha\ s)$
hence $length\ (GS.B\ s) = Suc\ (idx\text{-}of\ p\ v)$
unfolding *collapse-def collapse-aux-def Let-def*
apply (*cases s*)
apply (*auto simp*: *GS.p-α-def*)
apply (*drule arg-cong*[**where** *f=length*])
using *GS-invar.p-α-disjoint-sym*[*OF GS-invar*]
and *PRE* ‹$GS.p\text{-}\alpha\ (S', B', I', P') = p$› *idx-of-props*(*1*)[*of p v*]
by *simp*
} **note** *AUX1* = *this*

show *?thesis*
unfolding *gcollapse-alt gcollapse-impl-aux-def*
apply *simp*
apply (*rule RETURN-as-SPEC-refine*)
apply (*refine-rcg*
order-trans[*OF GS-invar.idx-of-correct*[*OF GS-invar PRE'*]]

order-trans[*OF collapse-refine*[*OF OSR B PRE*, *simplified*]]
refine-vcg
)
using *PRE′* **apply** *simp*

apply (*simp add*: *L*)

using *Un-set-drop-impl-correct acc-bound R*
apply (*simp add*: *refine-pw-simps pw-le-iff*)
unfolding *gGS-rel-def GS-rel-def GS.α-def br-def gGS-α-def gGS-invar-def*
apply (*clarsimp simp*: *L dest*!: *AUX1*)
apply (*auto dest*!: *AUX1 simp*: *L*)
apply (*force dest*!: *in-set-dropD*) []
apply (*force dest*!: *in-set-takeD*) []
done
qed

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

lemma *gcollapse-impl-aux-opt-refine*:
gcollapse-impl v s ≤ *gcollapse-impl-aux v s*
unfolding *gcollapse-impl-def gcollapse-impl-aux-def collapse-impl-def*
GS.collapse-impl-def
apply (*simp add*: *refine-pw-simps pw-le-iff split*: *prod.splits*)
apply *blast*
done

lemma *gcollapse-impl-refine*:
assumes *A*: (*s′*, *a*, *p*, *D*, *pE*) ∈ *gGS-rel*
assumes *B*: (*v′*,*v*)∈*Id*
assumes *PRE*: *v*∈⋃(*set p*)
shows *gcollapse-impl v′ s′*
≤ ⇓ *gGS-rel* (*RETURN* (*gcollapse v* (*a*, *p*, *D*, *pE*)))
using *order-trans*[*OF*
gcollapse-impl-aux-opt-refine
gcollapse-impl-aux-refine[*OF assms*]]
.

definition *ginitial-impl* :: $'Q \Rightarrow 'Q\ goGS \Rightarrow 'Q\ gGS$
where *ginitial-impl v0 s0* ≡ ([*acc v0*],*initial-impl v0* (*snd s0*))
lemma *ginitial-impl-refine*:
assumes *A*: *v0*∉*goD s0* *go-is-no-brk s0*
assumes *REL*: (*s0i*,*s0*)∈*goGS-rel* (*v0i*,*v0*)∈*Id*
shows (*ginitial-impl v0i s0i*,*ginitial v0 s0*)∈*gGS-rel*
unfolding *ginitial-impl-def ginitial-def*
using *REL initial-refine*[*OF A*(*1*) - *REL*(*2*), *of snd s0i*] *A*(*2*)
apply (*auto*
simp: *gGS-rel-def br-def gGS-α-def gGS-invar-def goGS-rel-def goGS-α-def*
simp: *go-is-no-brk-def goD-def oGS-rel-def GS-rel-def goGS-invar-def*
split: *prod.splits*

)
using *acc-bound*
apply (*fastforce simp*: *initial-impl-def GS-initial-impl-def*)+
done

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

lemma *gpath-is-empty-refine*:
(*s*,(*a*,*p*,*D*,*pE*))∈*gGS-rel* ⟹ *gpath-is-empty-impl s* ⟷ *p*=[]
unfolding *gpath-is-empty-impl-def*
using *path-is-empty-refine*
by (*fastforce simp*: *gGS-rel-def br-def gGS-invar-def gGS-α-def GS.α-def*)

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

lemma *gis-on-stack-refine*:
⟦(*s*,(*a*,*p*,*D*,*pE*))∈*gGS-rel*⟧ ⟹ *gis-on-stack-impl v s* ⟷ *v*∈⋃(*set p*)
unfolding *gis-on-stack-impl-def*
using *is-on-stack-refine*
by (*fastforce simp*: *gGS-rel-def br-def gGS-invar-def gGS-α-def GS.α-def*)

definition *gis-done-impl* :: $'Q \Rightarrow 'Q\ gGS \Rightarrow bool$
where *gis-done-impl v s* ≡ *is-done-impl v* (*snd s*)
thm *is-done-refine*
lemma *gis-done-refine*: (*s*,(*a*,*p*,*D*,*pE*))∈*gGS-rel*
⟹ *gis-done-impl v s* ⟷ (*v* ∈ *D*)
using *is-done-refine*[*of* (*snd s*) *v*]
by (*auto*
simp: *gGS-rel-def br-def gGS-α-def gGS-invar-def GS.α-def*
gis-done-impl-def)

definition (**in** −) *on-stack-less I u v* ≡
case I v of

Some (*STACK j*) ⇒ *j*<*u*
| - ⇒ *False*

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

lemma (**in** *GS-invar*) *set-butlast-p-refine*:
assumes *PRE*: *p-α*≠[]
shows *Collect* (*on-stack-less I* (*last B*)) = ⋃(*set* (*butlast p-α*)) (**is** *?L*=*?R*)
proof (*intro equalityI subsetI*)
from *PRE* **have** [*simp*]: *B*≠[] **by** (*auto simp*: *p-α-def*)

have [*simp*]: *S*≠[]
by (*simp add*: *empty-eq*)

{
fix *v*
assume *v*∈*?L*
then obtain *j* **where** [*simp*]: *I v* = *Some* (*STACK j*) **and** *j*<*last B*
by (*auto simp*: *on-stack-less-def split*: *option.splits node-state.splits*)

from *I-consistent*[*of v j*] **have** [*simp*]: *j*<*length S* *v*=*S*!*j* **by** *auto*

from *B0* **have** *B*!*0*=*0* **by** *simp*
from ‹*j*<*last B*› **have** *j*<*B*!(*length B* − *1*) **by** (*simp add*: *last-conv-nth*)
from *find-seg-bounds*[*OF* ‹*j*<*length S*›] *find-seg-correct*[*OF* ‹*j*<*length S*›]
have *v*∈*seg* (*find-seg j*) *find-seg j* < *length B* **by** *auto*
moreover with ‹*j*<*B*!(*length B* − *1*)› **have** *find-seg j* < *length B* − *1*

proof −
have *f1*: ⋀x_1 *x*. ¬ (x_1::*nat*) < x_1 − *x*
using *less-imp-diff-less* **by** *blast*
have *j* ≤ *last B*
by (*metis* ‹*j* < *last B*› *less-le*)
hence *f2*: ⋀x_1. ¬ *last B* < x_1 ∨ ¬ x_1 ≤ *j*
using *f1* **by** (*metis diff-diff-cancel le-trans*)
have ⋀x_1. *seg-end* x_1 ≤ *j* ∨ ¬ x_1 < *find-seg j*
by (*metis* ‹*seg-start* (*find-seg j*) ≤ *j*› *calculation*(*2*)
le-trans seg-end-less-start)
thus *find-seg j* < *length B* − *1*
using *f1 f2*
by (*metis GS.seg-start-def* ‹*B* ≠ []› ‹*j* < *B* ! (*length B* − *1*)›
‹*seg-start* (*find-seg j*) ≤ *j*› *calculation*(*2*) *diff-diff-cancel*
last-conv-nth nat-neq-iff seg-start-less-end)
qed
ultimately show *v*∈*?R*

```
    by (auto simp: p-α-def map-butlast[symmetric] butlast-upt)
  }

  {
    fix v
    assume v∈?R
    then obtain i where i<length B − 1 and v∈seg i
      by (auto simp: p-α-def map-butlast[symmetric] butlast-upt)
    then obtain j where j < seg-end i and v=S!j
      by (auto simp: seg-def)
    hence j<B!(i+1) and i+1 ≤ length B − 1 using ‹i<length B − 1›
      by (auto simp: seg-end-def last-conv-nth split: if-split-asm)
    with sorted-nth-mono[OF B-sorted ‹i+1 ≤ length B − 1›] have j<last B
      by (auto simp: last-conv-nth)
    moreover from ‹j < seg-end i› have j<length S
      by (metis GS.seg-end-def add-diff-inverse-nat ‹i + 1 ≤ length B − 1›
        add-lessD1 less-imp-diff-less less-le-not-le nat-neq-iff
        seg-end-bound)

    with I-consistent ‹v=S!j› have I v = Some (STACK j) by auto
    ultimately show v∈?L
      by (auto simp: on-stack-less-def)
  }
qed

lemma (in GS-invar) set-last-p-refine:
  assumes PRE: p-α≠[]
  shows Collect (on-stack-ge I (last B)) = last p-α (is ?L=?R)
proof (intro equalityI subsetI)
  from PRE have [simp]: B≠[] by (auto simp: p-α-def)

  have [simp]: S≠[] by (simp add: empty-eq)

  {
    fix v
    assume v∈?L
    then obtain j where [simp]: I v = Some (STACK j) and j≥last B
      by (auto simp: on-stack-ge-def split: option.splits node-state.splits)

    from I-consistent[of v j] have [simp]: j<length S    v=S!j by auto
    hence v∈seg (length B − 1) using ‹j≥last B›
      by (auto simp: seg-def last-conv-nth seg-start-def seg-end-def)
    thus v∈last p-α by (auto simp: p-α-def last-map)
  }

  {
    fix v
    assume v∈?R
    hence v∈seg (length B − 1)
```

by (*auto simp*: *p-α-def last-map*)
then obtain *j* **where** *v=S!j*   *j≥last B*   *j<length S*
by (*auto simp*: *seg-def last-conv-nth seg-start-def seg-end-def*)
with *I-consistent* **have** *I v = Some* (*STACK j*) **by** *simp*
with ‹*j≥last B*› **show** *v∈?L* **by** (*auto simp*: *on-stack-ge-def*)
}
qed

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

lemma *ce-impl-refine*:
assumes *A*: (*s*,(*a*,*p*,*D*,*pE*))∈*gGS-rel*
assumes *PRE*: *p≠*[]
shows *ce-impl s* ≤ ⇓(*Id*$\times_r$*gGS-rel*)
(*RETURN* (*Some* (⋃(*set* (*butlast p*)),*last p*),*a*,*p*,*D*,*pE*))
proof −
from *A* **obtain** *S' B' I' P'* **where** [*simp*]: *s*=(*a*,*S'*,*B'*,*I'*,*P'*)
and *OSR*: ((*S'*,*B'*,*I'*,*P'*),(*p*,*D*,*pE*))∈*GS-rel* **and** *L*: *length a = length B'*
by (*rule gGS-relE*)

from *OSR* **have** [*simp*]: *GS.p-α* (*S'*,*B'*,*I'*,*P'*) = *p*
by (*simp add*: *GS-rel-def br-def GS.α-def*)

from *PRE* **have** *NE'*: *GS.p-α* (*S'*, *B'*, *I'*, *P'*) ≠ [] **by** *simp*
hence *BNE*[*simp*]: *B'≠*[] **by** (*simp add*: *GS.p-α-def*)

from *OSR* **have** *GS-invar*: *GS-invar* (*S'*,*B'*,*I'*,*P'*)
by (*simp add*: *GS-rel-def br-def*)

show *?thesis*
using *GS-invar.set-butlast-p-refine*[*OF GS-invar NE'*]
using *GS-invar.set-last-p-refine*[*OF GS-invar NE'*]
unfolding *ce-impl-def*
using *A*
by *auto*
qed

definition *last-is-acc-impl s ≡*
do {
let (*a*,-)=*s*;
ASSERT (*a≠*[]);

```
    RETURN (∀ i<num-acc. i∈last a)
  }

lemma last-is-acc-impl-refine:
  assumes A: (s,(a,p,D,pE))∈gGS-rel
  assumes PRE: a≠[]
  shows last-is-acc-impl s ≤ RETURN (last a = {0..<num-acc})
proof −
  from A PRE have last a ⊆ {0..<num-acc}
    unfolding gGS-rel-def gGS-invar-def br-def gGS-α-def by auto
  hence C: (∀ i<num-acc. i∈last a) ⟷ (last a = {0..<num-acc})
    by auto

  from A obtain gs where [simp]: s=(a,gs)
    by (auto simp: gGS-rel-def gGS-α-def br-def split: prod.splits)

  show ?thesis
    unfolding last-is-acc-impl-def
    by (auto simp: gGS-rel-def br-def gGS-α-def C PRE split: prod.splits)
qed

definition go-is-no-brk-impl :: 'Q goGS ⇒ bool
  where go-is-no-brk-impl s ≡ fst s = None
lemma go-is-no-brk-refine:
  (s,s')∈goGS-rel ⟹ go-is-no-brk-impl s ⟷ go-is-no-brk s'
  unfolding go-is-no-brk-def go-is-no-brk-impl-def
  by (auto simp: goGS-rel-def br-def goGS-α-def split: prod.splits)

definition goD-impl :: 'Q goGS ⇒ 'Q oGS where goD-impl s ≡ snd s
lemma goD-refine:
  go-is-no-brk s' ⟹ (s,s')∈goGS-rel ⟹ (goD-impl s, goD s')∈oGS-rel
  unfolding goD-impl-def goD-def
  by (auto
    simp: goGS-rel-def br-def goGS-α-def goGS-invar-def oGS-rel-def
       go-is-no-brk-def)

definition go-is-done-impl :: 'Q ⇒ 'Q goGS ⇒ bool
  where go-is-done-impl v s ≡ is-done-oimpl v (snd s)
thm is-done-orefine
lemma go-is-done-impl-refine: ⟦go-is-no-brk s'; (s,s')∈goGS-rel; (v,v')∈Id⟧
  ⟹ go-is-done-impl v s ⟷ (v'∈goD s')
  using is-done-orefine
  unfolding go-is-done-impl-def goD-def go-is-no-brk-def
  apply (fastforce simp: goGS-rel-def br-def goGS-invar-def goGS-α-def)
  done


definition goBrk-impl :: 'Q goGS ⇒ 'Q ce where goBrk-impl ≡ fst
```

```
lemma goBrk-refine: (s,s')∈goGS-rel ⟹ (goBrk-impl s, goBrk s')∈Id
  unfolding goBrk-impl-def goBrk-def
  by (auto simp: goGS-rel-def br-def goGS-α-def split: prod.splits)

definition find-ce-impl :: ('Q set × 'Q set) option nres where
  find-ce-impl ≡ do {
    stat-start-nres;
    let os=goinitial-impl;
    os←FOREACHci (λit os. fgl-outer-invar it (goGS-α os)) V0
      (go-is-no-brk-impl) (λv0 s0.
    do {
      if ¬go-is-done-impl v0 s0 then do {

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

        (brk,s)←WHILEIT
          (λ(brk,s). fgl-invar v0 (oGS-α (goD-impl s0)) (brk,snd (gGS-α s)))
          (λ(brk,s). brk=None ∧ ¬gpath-is-empty-impl s) (λ(l,s).
        do {
          — Select edge from end of path
          (vo,s) ← gselect-edge-impl s;

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

```
  RETURN (goBrk-impl os)
}

lemma find-ce-impl-refine: find-ce-impl ≤ ⇓Id gfind-ce
proof –
  note [refine2] = prod-relI[OF IdI[of None] ginitial-impl-refine]

  have [refine]: ⋀s a p D pE. ⟦
    (s,(a,p,D,pE))∈gGS-rel;
    p ≠ []; pE ∩ last p × UNIV = {}
    ⟧ ⟹
    gpop-impl s ≫= (λs. RETURN (None, s))
      ≤ SPEC (λc. (c, None, gpop (a,p,D,pE)) ∈ Id ×r gGS-rel)
    apply (drule (2) gpop-impl-refine)
    apply (fastforce simp add: pw-le-iff refine-pw-simps)
    done

  note [[goals-limit = 1]]

  note FOREACHci-refine-rcg'[refine del]

  show ?thesis
    unfolding find-ce-impl-def gfind-ce-def
    apply (refine-rcg
      bind-refine'
      prod-relI IdI
      inj-on-id

      gselect-edge-impl-refine gpush-impl-refine
      oinitial-refine ginitial-impl-refine
      bind-Let-refine2[OF gcollapse-impl-refine]
      if-bind-cond-refine[OF last-is-acc-impl-refine]
      ce-impl-refine
      goinitial-impl-refine
      gto-outer-refine
      goBrk-refine
      FOREACHci-refine-rcg'[where R=goGS-rel, OF inj-on-id]
    )

    apply refine-dref-type

    apply (simp-all add: go-is-no-brk-refine go-is-done-impl-refine)
    apply (auto simp: goGS-rel-def br-def) []
    apply (auto simp: goGS-rel-def br-def goGS-α-def gGS-α-def gGS-rel-def
                  goD-def goD-impl-def) []

    apply (auto dest: gpath-is-empty-refine ) []
    apply (auto dest: gis-on-stack-refine) []
    apply (auto dest: gis-done-refine) []
```

```
    done
  qed

end
```

1.22 Constructing a Lasso from Counterexample

1.22.1 Lassos in GBAs

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

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

  lemma reconstruct-reach-correct:
    assumes CEC: ce-correct Vr Vl
    shows reconstruct-reach Vr Vl
      ≤ SPEC (λ(pr,va). ∃v0∈V0. path E v0 pr va ∧ va∈Vl)
  proof −
    have FIN-aux: finite ((E ∩ Vr × UNIV)* `` V0)
      by (metis finite-reachableE-V0 finite-subset inf-sup-ord(1) inf-sup-ord(2)
        inf-top.left-neutral reachable-mono)

    {
      fix u p v
      assume P: path E u p v and SS: set p ⊆ Vr
      have path (E ∩ Vr×UNIV) u p v
        apply (rule path-mono[OF - path-restrict[OF P]])
        using SS by auto
    } note P-CONV=this

    from CEC obtain v0 pr va where v0∈V0   set pr ⊆ Vr   va∈Vl
      path (E ∩ Vr×UNIV) v0 pr va
      unfolding ce-correct-def is-lasso-prpl-def is-lasso-prpl-pre-def
      by (force simp: neq-Nil-conv path-simps dest: P-CONV)
    hence 1: va ∈ (E ∩ Vr × UNIV)* `` V0
      by (auto dest: path-is-rtrancl)

    show ?thesis
      using assms unfolding reconstruct-reach-def
      apply (refine-rcg refine-vcg order-trans[OF find-path-ex-rule])
      apply (clarsimp-all simp: FIN-aux finite-V0)

      using ‹va∈Vl› 1 apply auto []
```

apply (*auto dest*: *path-mono*[*of* $E \cap Vr \times UNIV\ E$, *simplified*]) []
done
qed

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

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

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

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

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

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

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

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

let $?E = E \cap UNIV \times Vl$

have *E-SS*: $E \cap Vl \times Vl \subseteq ?E$ **by** *auto*

from *CEC* **have**
REACH: $Vl \subseteq E^* `` V0$
and *CONN*: $Vl \times Vl \subseteq (E \cap Vl \times Vl)^*$
and *NONTRIV*: $Vl \times Vl \cap E \neq \{\}$
and *NES*[*simp*]: $Vl \neq \{\}$
and *ALL*: $\bigcup(acc`Vl) = \{0..<num\text{-}acc\}$
unfolding *ce-correct-def is-lasso-prpl-def*
apply *clarsimp-all*
apply *auto* []
apply *force*
done

define *term-rel*
where *term-rel* = (*inv-image* (*finite-psupset* $\{0..<num\text{-}acc\}$) (λ(-::'*Q*,-::'*Q* *list*,*cS*). *cS*))
hence *WF*: *wf term-rel* **by** *simp*

{ **fix** *va*
assume $va \in Vl$
hence *rec-loop-invar Vl va* (*va*, [], *acc va*)
unfolding *rec-loop-invar-def* **using** *REACH* **by** *auto*
} **note** *INVAR-INITIAL* = *this*

{
fix *v p cS va*
assume *rec-loop-invar Vl va* (*v*, *p*, *cS*)
hence *finite* $((?E)^* `` \{v\})$
apply −
apply (*rule finite-subset*[**where** $B=E^* `` V0$])
unfolding *rec-loop-invar-def*
using *REACH*
apply (*clarsimp-all dest*!: *path-is-rtrancl*)
apply (*drule rtrancl-mono-mp*[**where** $U=?E$ **and** $V=E$, *rotated*], (*auto*) [])
by (*metis rev-ImageI rtrancl-trans*)
} **note** *FIN1* = *this*

{
fix *va v* :: '*Q* **and** *p cS*
assume *INV*: *rec-loop-invar Vl va* (*v*,*p*,*cS*)
and *NC*: $cS \neq \{0..<num\text{-}acc\}$

from *NC INV* **obtain** *i* **where** $i<num\text{-}acc$ $i \notin cS$
unfolding *rec-loop-invar-def* **by** *auto blast*

with *ALL* **obtain** *v'* **where** *v'*: $v' \in Vl$ $\neg\, acc\ v' \subseteq cS$
by (*metis NES atLeastLessThan-iff cSUP-least in-mono zero-le*)

with *CONN INV* **have** $(v,v') \in (E \cap Vl \times Vl)^*$
unfolding *rec-loop-invar-def* **by** *auto*

hence $(v,v')\in ?E^*$ **using** *rtrancl-mono-mp*[*OF E-SS*] **by** *blast*
with v' **have** $\exists\, v'.\ (v,v')\in(?E)^* \wedge \neg\ acc\ v' \subseteq cS$ **by** *auto*
} **note** *ASSERT1* = *this*

{
fix *va v p cS v' p'*
assume *rec-loop-invar Vl va* $(v,\ p,\ cS)$
and *path* $(?E)\ v\ p'\ v'$
and $\neg\ (acc\ v' \subseteq cS)$
and $\forall\, v\in set\ p'.\ acc\ v \subseteq cS$
hence *rec-loop-invar Vl va* $(v',\ p@p',\ cS \cup acc\ v')$
unfolding *rec-loop-invar-def*
apply *simp*
apply (*intro conjI*)
apply (*auto simp*: *path-simps dest*: *path-mono*[*of ?E E*, *simplified*]) []

apply (*cases* p')
apply (*auto simp*: *path-simps*) [*2*]

apply (*cases* p' *rule*: *rev-cases*)
apply (*auto simp*: *path-simps*) [*2*]

apply (*erule path-set-induct*)
apply *auto* [*2*]
done
} **note** *INV-PRES* = *this*

{
fix *va v p cS v' p'*
assume *rec-loop-invar Vl va* $(v,\ p,\ cS)$
and *path ?E v* $p'\ v'$
and $\neg\ (acc\ v' \subseteq cS)$
and $\forall\, v\in set\ p'.\ acc\ v \subseteq cS$
hence $((v',\ p@p',\ cS \cup acc\ v'),\ (v,p,cS)) \in$ *term-rel*
unfolding *term-rel-def rec-loop-invar-def*
by (*auto simp*: *finite-psupset-def*)
} **note** *VAR* = *this*

have *CONN1*: $Vl \times Vl \subseteq (?E)^+$
proof *clarify*
fix *a b*
assume $a\in Vl$ $\quad b\in Vl$
from *NONTRIV* **obtain** *u v* **where** *E*: $(u,v)\in(E \cap Vl\times Vl)$ **by** *auto*
from *CONN* ‹$a\in Vl$› *E* **have** $(a,u)\in(E\cap Vl\times Vl)^*$ **by** *auto*
also note *E*
also (*rtrancl-into-trancl1*) **from** *CONN* ‹$b\in Vl$› *E* **have** $(v,b)\in(E\cap Vl\times Vl)^*$
by *auto*
finally show $(a,b)\in(?E)^+$ **using** *trancl-mono*[*OF - E-SS*] **by** *auto*
qed

{
  fix *va v p cS*
  assume *rec-loop-invar Vl va* (*v*, *p*, *cS*)
  hence (*v*,*va*) ∈ (*?E*)$^+$
    unfolding *rec-loop-invar-def*
    using *CONN1*
    by *auto*
} **note** *CLOSE1* = *this*

{
  fix *va v p cS*
  assume *rec-loop-invar Vl va* (*v*, *p*, *cS*)
  hence (*v*,*va*) ∈ (*?E*)*
    unfolding *rec-loop-invar-def*
    using *CONN rtrancl-mono*[*OF E-SS*]
    by *auto*
} **note** *CLOSE2* = *this*

{
  fix *pr vd pl va v0*
  assume *rec-loop-invar Vl va* (*vd*, [], {*0*..<*num-acc*})   *va* ∈ *Vl*   *v0* ∈ *V0*
    path E v0 pr va   *pl* ≠ []   *path ?E vd pl va*
  hence *is-lasso-prpl* (*pr*, *pl*)
    unfolding *is-lasso-prpl-def is-lasso-prpl-pre-def rec-loop-invar-def*
    by (*auto simp*: *neq-Nil-conv path-simps*
      dest!: *path-mono*[*of ?E E*, *simplified*]) []
} **note** *INV-POST1* = *this*

{
  fix *va v p p′ pr v0*
  assume *INV*: *rec-loop-invar Vl va* (*v*,*p*,{*0*..<*num-acc*})
    and *1*: *p*≠[]   *path ?E v p′ va*
    and *PR*: *v0*∈*V0*   *path E v0 pr va*

  from *INV* **have** ∀ *i*<*num-acc*. ∃ *q*∈*insert v* (*set p*). *i* ∈ *acc q*
    unfolding *rec-loop-invar-def*
    by *auto*
  moreover from *INV 1* **have** *insert v* (*set p*) ⊆ *set p* ∪ *set p′*
    unfolding *rec-loop-invar-def*
    apply (*cases p*)
    apply *blast*
    apply (*cases p′*)
    apply (*auto simp*: *path-simps*)
    done
  ultimately have *ACC*: ∀ *i*<*num-acc*. ∃ *q*∈*set p* ∪ *set p′*. *i* ∈ *acc q* **by** *blast*

  from *INV 1* **have** *PL*: *path E va* (*p* @ *p′*) *va*
    by (*auto simp*: *rec-loop-invar-def path-simps*

```
      dest!: path-mono[of ?E E, simplified]) []

    have is-lasso-prpl (pr,p@p')
      unfolding is-lasso-prpl-def is-lasso-prpl-pre-def
      apply (clarsimp simp: ACC)
      using PR PL ‹p≠[]› by auto
  } note INV-POST2 = this

  show ?thesis
    unfolding reconstruct-lasso-def
    apply (refine-rcg
      WF
      order-trans[OF reconstruct-reach-correct]
      order-trans[OF find-path-ex-rule]
      order-trans[OF find-path1-ex-rule]
      refine-vcg
    )

    apply (vc-solve
      del: subsetI
        solve: ASSERT1 INV-PRES asm-rl VAR CLOSE1 CLOSE2 INV-POST1
INV-POST2
      simp: INVAR-INITIAL FIN1 CEC)
    done
qed

definition find-lasso where find-lasso ≡ do {
  ce ← find-ce-spec;
  case ce of
    None ⇒ RETURN None
  | Some (Vr,Vl) ⇒ do {
      l ← reconstruct-lasso Vr Vl;
      RETURN (Some l)
    }
}

lemma (in igb-fr-graph) find-lasso-correct: find-lasso ≤ find-lasso-spec
  unfolding find-lasso-spec-def find-lasso-def find-ce-spec-def
  apply (refine-rcg refine-vcg order-trans[OF reconstruct-lasso-correct])
  apply auto
  done

end

end
```

1.23 Code Generation for the Skeleton Algorithm

```
theory Gabow-Skeleton-Code
```

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

1.24 Statistics

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

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

      val time = Unsynchronized.ref Time.zeroTime

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

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

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

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

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

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

1.25 Automatic Refinement Setup

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

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

lemma node-state-rel-sv[simp,intro!,relator-props]:
  single-valued node-state-rel
  unfolding node-state-rel-def
  by (auto intro: single-valuedI)

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

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

lemma node-state-rel-refine[autoref-rules]:
  (−1,DONE)∈node-state-rel
  (int,STACK)∈nat-rel→node-state-rel
  (λi. i<0,is-DONE)∈node-state-rel→bool-rel
  ((λf g i. if i≥0 then f (nat i) else g),case-node-state)
    ∈(nat-rel → R) → R → node-state-rel → R
  unfolding node-state-rel-def
    apply auto [3]
    apply (fastforce dest: fun-relD)
    done

lemma [autoref-op-pat]:
  (x=DONE) ≡ is-DONE x
  (DONE=x) ≡ is-DONE x
  apply (auto intro!: eq-reflection)
  apply ((cases x, simp-all) [])+
  done


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*
by *unfold-locales*

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

abbreviation *GSi-rel* ≡
⟨*node-rel*⟩*as-rel*
$\times_r$ ⟨*nat-rel*⟩*as-rel*
$\times_r$ *oGSi-rel*
$\times_r$ ⟨*nat-rel* $\times_r$ ⟨*node-rel*⟩*list-set-rel*⟩*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*,*push-impl*)∈*node-rel* → *GSi-rel* → *GSi-rel*
unfolding *push-impl-def-opt*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*))
done
concrete-definition (**in** −) *push-code* **uses** *fr-graph-impl-loc.push-code-aux*
lemmas [*autoref-rules*] = *push-code.refine*[*OF locale-this*]

schematic-goal *pop-code-aux*: (*?c*,*pop-impl*)∈*GSi-rel* → ⟨*GSi-rel*⟩*nres-rel*
unfolding *pop-impl-def-opt*[*abs-def*]
unfolding *GS.mark-as-done-def*

```
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done
concrete-definition (in −) pop-code uses fr-graph-impl-loc.pop-code-aux
lemmas [autoref-rules] = pop-code.refine[OF locale-this]

schematic-goal S-idx-of-code-aux:
  notes [autoref-rules] = IdI[of undefined::nat]
  shows (?c,GS.S-idx-of)∈GSi-rel → node-rel → nat-rel
  unfolding GS.S-idx-of-def[abs-def]
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done
concrete-definition (in −) S-idx-of-code
  uses fr-graph-impl-loc.S-idx-of-code-aux
lemmas [autoref-rules] = S-idx-of-code.refine[OF locale-this]

schematic-goal idx-of-code-aux:
  notes [autoref-rules] = IdI[of undefined::nat]
  shows (?c,GS.idx-of-impl)∈ GSi-rel → node-rel → ⟨nat-rel⟩nres-rel
  unfolding
    GS.idx-of-impl-def[abs-def, unfolded GS.find-seg-impl-def GS.S-idx-of-def,
      THEN opt-GSdef, unfolded GS-sel-simps, abs-def]
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done
concrete-definition (in −) idx-of-code uses fr-graph-impl-loc.idx-of-code-aux
lemmas [autoref-rules] = idx-of-code.refine[OF locale-this]

schematic-goal collapse-code-aux:
  (?c,collapse-impl)∈node-rel → GSi-rel → ⟨GSi-rel⟩nres-rel
  unfolding collapse-impl-def-opt[abs-def]
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done
concrete-definition (in −) collapse-code
  uses fr-graph-impl-loc.collapse-code-aux
lemmas [autoref-rules] = collapse-code.refine[OF locale-this]

term select-edge-impl
schematic-goal select-edge-code-aux:
  (?c,select-edge-impl)
    ∈ GSi-rel → ⟨⟨node-rel⟩option-rel ×_r GSi-rel⟩nres-rel
  unfolding select-edge-impl-def-opt[abs-def]

  using [[autoref-trace-failed-id]]
  using [[goals-limit=1]]
  apply (autoref (keep-goal,trace))
  done
```

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* **.**

  term *fr-graph.pop-impl*
  lemma [*autoref-op-pat*]:
    push-impl ≡ *OP push-impl*
    collapse-impl ≡ *OP collapse-impl*
    select-edge-impl ≡ *OP select-edge-impl*
    pop-impl ≡ *OP pop-impl*
    by *simp-all*

end

schematic-goal *skeleton-code-aux*:
  (*?c*,*skeleton-impl*) ∈ ⟨*oGSi-rel*⟩*nres-rel*
  unfolding *skeleton-impl-def*[*abs-def*] *initial-impl-def GS-initial-impl-def*
  unfolding *path-is-empty-impl-def is-on-stack-impl-def is-done-impl-def*
    is-done-oimpl-def
  unfolding *GS.is-on-stack-impl-def GS.is-done-impl-def*
  using [[*autoref-trace-failed-id*]]
  apply (*autoref* (*keep-goal*,*trace*))
  done

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

thm *skeleton-code.refine*

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

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

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

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

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

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

end

term *skeleton-tr*

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

end

1.27 Code Generation for SCC-Computation

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

1.28 Automatic Refinement to Efficient Data Structures

context *fr-graph-impl-loc*
begin
schematic-goal *last-seg-code-aux*:
(*?c*,*last-seg-impl*)∈*GSi-rel* → ⟨⟨*node-rel*⟩*list-set-rel*⟩*nres-rel*
  unfolding *last-seg-impl-def-opt*[*abs-def*]
  using [[*autoref-trace-failed-id*]]
  apply (*autoref* (*keep-goal*,*trace*))
  done
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* **.**

lemma [*autoref-op-pat*]:
last-seg-impl ≡ *OP last-seg-impl*
by *simp-all*
end

schematic-goal *compute-SCC-code-aux*:
(*?c*,*compute-SCC-impl*) ∈ ⟨⟨⟨*node-rel*⟩*list-set-rel*⟩*list-rel*⟩*nres-rel*
unfolding *compute-SCC-impl-def*[*abs-def*] *initial-impl-def GS-initial-impl-def*
unfolding *path-is-empty-impl-def is-on-stack-impl-def is-done-impl-def*
is-done-oimpl-def
unfolding *GS.is-on-stack-impl-def GS.is-done-impl-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*,*trace*))
done

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

schematic-goal *last-seg-tr-aux*: *RETURN ?c* ≤ *last-seg-code s*
unfolding *last-seg-code-def* **by** *refine-transfer*
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*
unfolding *compute-SCC-code-def* **by** *refine-transfer*
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*)
$\leq \Downarrow(\langle\langle$*node-rel*$\rangle$*list-set-rel*$\rangle$*list-rel*) (*fr-graph.compute-SCC-spec G*)
proof −
from *C* **interpret** *fr-graph G* **.**
have *I*: *fr-graph-impl-loc Re node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
apply *unfold-locales* **using** *A* **.**
then interpret *fr-graph-impl-loc Re node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G* **.**

note *compute-SCC-tr.refine*[*OF I*]
also note *compute-SCC-code.refine*[*OF I*, *THEN nres-relD*]
also note *compute-SCC-impl-refine*
also note *compute-SCC-correct*
finally show *?thesis* **using** *A* **by** *simp*
qed

1.30 Extraction of Benchmark Code

schematic-goal *list-set-of-list-aux*:
(*?c*,*set*)$\in\langle$*nat-rel*$\rangle$*list-rel* $\to$ $\langle$*nat-rel*$\rangle$*list-set-rel*
by *autoref*
concrete-definition *list-set-of-list* **uses** *list-set-of-list-aux*

term *compute-SCC-tr*

definition *compute-SCC-tr-nat* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ *nat list list*
where *compute-SCC-tr-nat* $\equiv$ *compute-SCC-tr*

end

1.31 Implementation of Safety Property Model Checker

theory *Find-Path-Impl*
imports
Find-Path
CAVA-Automata.Digraph-Impl
begin

1.32 Workset Algorithm

A simple implementation is by a workset algorithm.

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

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

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

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

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

  define inv where inv = (λ(res,V,ws). case res of
      None ⇒
        dom ws⊆V
      ∧ finite (dom ws)  — Derived
      ∧ V⊆E* ‘‘U0
      ∧ E‘‘(V−dom ws) ⊆ V
      ∧ (∀v∈V−dom ws. ¬ P v)
      ∧ U0 ⊆ V
      ∧ (∀v p. ws v = Some p
        ⟶ ((∀v∈set p. ¬P v) ∧ (∃u0∈U0. path E u0 (rev p) v)))
    | Some (p,v) ⇒ (∃u0∈U0. path E u0 p v ∧ P v ∧ (∀v∈set p. ¬P v)))

  define var where var = inv-image
      (brk-rel (finite-psupset (E* ‘‘U0) <*lex*> measure (card o dom)))
      (λ(res::('v list × 'v) option,V::'v set,ws::'v⇀'v list).
          (res≠None,V,ws))



  have [simp]: ⋀u p V. dom (λv. if (u, v) ∈ E ∧ v ∉ V then Some (u # p)
```

else None) = $E``\{u\} - V$
  by (*auto split*: *if-split-asm*)

{
  fix *V ws u p*
  assume *INV*: *inv* (*None*,*V*,*ws*)
  assume *WSU*: *ws u* = *Some p*

  from *INV WSU* **have**
    [*simp*]: $V \subseteq E^* ``U0$
    and [*simp*]: $u \in V$
    and *UREACH*: $\exists u0 \in U0.\ (u0,u) \in E^*$
    and [*simp*]: *finite* (*dom ws*)
    unfolding *inv-def*
    apply *simp-all*
    apply *auto* []
    apply *clarsimp*
    apply *blast*
    done
  have ($V \cup E$ `` $\{u\}$, $V$) $\in$ *finite-psupset* ($E^*$ `` *U0*) $\vee$
      $V \cup E$ `` $\{u\} = V \wedge$
      card (E `` $\{u\} - V \cup$ (*dom ws* $- \{u\}$)) < *card* (*dom ws*)
  proof (*subst disj-commute*, *intro disjCI conjI*)
    assume ($V \cup E$ `` $\{u\}$, $V$) $\notin$ *finite-psupset* ($E^*$ `` *U0*)
    thus $V \cup E$ `` $\{u\} = V$ **using** *UREACH*
      by (*auto simp*: *finite-psupset-def intro*: *rev-ImageI*)

    hence [*simp*]: $E``\{u\} - V = \{\}$ **by** *force*
    show *card* (E `` $\{u\} - V \cup$ (*dom ws* $- \{u\}$)) < *card* (*dom ws*)
      using *WSU*
      by (*auto intro*: *card-Diff1-less*)
  qed
} **note** *wf-aux=this*

{
  fix *V ws u p*
  assume *FIN*: *finite* ($E^* ``U0$)
  assume *inv* (*None*,*V*,*ws*)   *ws u* = *Some p*
  then obtain *u0* **where** $u0 \in U0$ $(u0,u) \in E^*$ **unfolding** *inv-def*
    by *clarsimp blast*
  hence $E``\{u\} \subseteq E^* ``U0$ **by** (*auto intro*: *rev-ImageI*)
  hence *finite* ($E``\{u\}$) **using** *FIN*(*1*) **by** (*rule finite-subset*)
} **note** *succs-finite=this*

{
  fix *V ws u p*
  assume *FIN*: *finite* ($E^* ``U0$)
  assume *INV*: *inv* (*None*,*V*,*ws*)

assume *WSU*: $ws\ u = Some\ p$
assume *NVD*: $\neg\ P\ u$

have $inv\ (None,\ V \cup E\ ``\ \{u\},$
$ws\ |`\ (-\ \{u\})\ ++$
$(\lambda v.\ if\ (u,\ v) \in E \wedge v \notin V\ then\ Some\ (u\ \#\ p)$
$else\ None))$
unfolding *inv-def*

apply (*simp*, *intro conjI*)
using *INV WSU* **apply** (*auto simp*: *inv-def*) []
using *INV WSU* **apply** (*auto simp*: *inv-def*) []
using *INV WSU* **apply** (*auto simp*: *succs-finite FIN*) []
using *INV* **apply** (*auto simp*: *inv-def*) []
using *INV* **apply** (*auto simp*: *inv-def*) []

using *INV WSU* **apply** (*auto*
simp: *inv-def*
intro: *rtrancl-image-advance*
) []

using *INV WSU* **apply** (*auto simp*: *inv-def*) []

using *INV NVD* **apply** (*auto simp*: *inv-def*) []
using *INV NVD* **apply** (*auto simp*: *inv-def*) []

using *INV WSU NVD* **apply** (*fastforce*
simp: *inv-def restrict-map-def*
intro!: *path-conc path1*
split: *if-split-asm*
) []
done
} **note** *ip-aux=this*

show *?thesis*
unfolding *wset-find-path-def find-path-def ws-update-def s-init-def*

apply (*refine-rcg refine-vcg le-ASSERTI*
WHILET-rule[**where**
$R = var$ **and** $I = inv$]
)

using [[*goals-limit* = *1*]]

apply (*auto simp*: *var-def*) []

apply (*auto*
simp: *inv-def dom-def*
split: *if-split-asm*) []

```
    apply simp
    apply (auto simp: inv-def) []
    apply (auto simp: var-def brk-rel-def) []

    apply (simp add: succs-finite)

    apply (auto simp: inv-def) []

    apply clarsimp
    apply (simp add: ip-aux)

    apply clarsimp
    apply (simp add: var-def brk-rel-def wf-aux) []

    apply (fastforce
      simp: inv-def
      split: option.splits
      intro: rev-ImageI
      dest: Image-closed-trancl) []
    done
  qed
```

We refine the algorithm to use a foreach-loop

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

  lemma ws-update-foreach-refine[refine]:
    assumes FIN: finite (E``{u})
    assumes WSS: dom ws ⊆ V
    assumes ID: (E',E)∈Id   (u',u)∈Id   (p',p)∈Id   (V',V)∈Id   (ws',ws)∈Id
    shows ws-update-foreach E' u' p' V' ws' ≤ ⇓Id (ws-update E u p V ws)
    unfolding ID[simplified]
    unfolding ws-update-foreach-def ws-update-def LIST-SET-REV-TAG-def
    apply (refine-rcg refine-vcg FIN
      FOREACH-rule[where I=λit (V',ws').
        V'=V ∪ (E``{u}−it)
      ∧ dom ws' ⊆ V'
      ∧ ws' = ws ++ (λv. if (u,v)∈E ∧ v∉it ∧ v∉V then Some (u#p) else None)]
    )
    using WSS
    apply (auto
      simp: Map.map-add-def
```

```
    split: option.splits if-split-asm
    intro!: ext[where 'a='a and 'b='b list option])

  done

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

lemma s-init-foreach-refine[refine]:
  assumes FIN: finite U0
  assumes ID: (U0',U0)∈Id
  shows s-init-foreach U0' ≤⇓Id (s-init U0)
  unfolding s-init-foreach-def s-init-def ID[simplified]

  apply (refine-rcg refine-vcg
    FOREACH-rule[where
      I = λit (U,ws).
        U = U0−it
      ∧ ws = (λx. if x∈U0−it then Some [] else None)]
  )

  apply (auto
    simp: FIN
    intro!: ext
  )
  done

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

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

lemma *wset-find-path′-refine*:
wset-find-path′ E U0 P ≤ ⇓*Id* (*wset-find-path E U0 P*)
unfolding *wset-find-path′-def wset-find-path-def*
unfolding *op-map-pick-remove-alt*
apply (*refine-rcg IdI*)
apply *assumption*
apply *simp-all*
done

1.33 Refinement to efficient data structures

schematic-goal *wset-find-path′-refine-aux*:
fixes *U0*::′*a set* **and** *P*::′*a* ⇒ *bool* **and** *E*::(′*a*×′*a*) *set*
and *Pimpl* :: ′*ai* ⇒ *bool*
and *node-rel* :: (′*ai* × ′*a*) *set*
and *node-eq-impl* :: ′*ai* ⇒ ′*ai* ⇒ *bool*
and *node-hash-impl*
and *node-def-hash-size*

assumes [*autoref-rules*]:
(*succi*,*E*)∈⟨*node-rel*⟩*slg-rel*
(*Pimpl*,*P*)∈*node-rel* → *bool-rel*
(*node-eq-impl*, (=)) ∈ *node-rel* → *node-rel* → *bool-rel*
(*U0*′,*U0*)∈⟨*node-rel*⟩*list-set-rel*
assumes [*autoref-ga-rules*]:
is-bounded-hashcode node-rel node-eq-impl node-hash-impl
is-valid-def-hm-size TYPE(′*ai*) *node-def-hash-size*
notes [*autoref-tyrel*] =
TYRELI[**where**
R=⟨*node-rel*,⟨*node-rel*⟩*list-rel*⟩*list-map-rel*]
TYRELI[**where** *R*=⟨*node-rel*⟩*map2set-rel* (*ahm-rel node-hash-impl*)]

shows (*?c*::*?*′*c*,*wset-find-path′ E U0 P*) ∈ *?R*
unfolding *wset-find-path′-def ws-update-foreach-def s-init-foreach-def*
using [[*autoref-trace-failed-id*]]
using [[*autoref-trace-intf-unif*]]
using [[*autoref-trace-pat*]]
apply (*autoref* (*keep-goal*))
done

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

1.34 Autoref Setup

context begin interpretation *autoref-syn* **.**
lemma [*autoref-itype*]:

find-path $::_i$ $\langle I\rangle_i$*i-slg* $\rightarrow_i$ $\langle I\rangle_i$*i-set* $\rightarrow_i$ $(I\rightarrow_i$*i-bool*$)$
$\rightarrow_i$ $\langle\langle\langle\langle I\rangle_i$*i-list*, $I\rangle_i$*i-prod*$\rangle_i$*i-option*$\rangle_i$*i-nres* **by** *simp*

lemma *wset-find-path-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 (
wset-find-path-impl node-hash-impl node-def-hash-size node-eq-impl,
find-path)
$\in$ $\langle$*node-rel*$\rangle$*slg-rel* $\rightarrow$ $\langle$*node-rel*$\rangle$*list-set-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*
proof −
note *EQI* = *GEN-OP-D*[*OF eq*]
note *HASHI* = *SIDE-GEN-ALGO-D*[*OF hash*]
note *DSZI* = *SIDE-GEN-ALGO-D*[*OF hash-dsz*]

note *wset-find-path-impl.refine*[*THEN nres-relD*, *OF - - EQI - HASHI DSZI*]
also note *wset-find-path'-refine*
also note *wset-find-path-correct*
finally show *?thesis*
by (*fastforce intro*!: *nres-relI*)
qed
end

schematic-goal *wset-find-path-transfer-aux*:
RETURN ?c $\leq$ *wset-find-path-impl hashi dszi eqi E U0 P*
unfolding *wset-find-path-impl-def*
by (*refine-transfer* (*post*))
concrete-definition *wset-find-path-code*
for *E ?U0.0 P* **uses** *wset-find-path-transfer-aux*
lemmas [*refine-transfer*] = *wset-find-path-code.refine*

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

1.35 Nontrivial paths

definition *find-path1-gen E u0 P* $\equiv$ *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*
unfolding *find-path1-gen-def find-path-def find-path1-def*
apply (*refine-rcg refine-vcg le-ASSERTI*)

apply (*auto*
intro: *path-prepend*
dest: *tranclD*
elim: *finite-subset*[*rotated*]
)
done

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

shows (*?c*,*find-path1-gen*::(-×-) *set* ⇒ -) ∈ ⟨*node-rel*⟩*slg-rel* → *node-rel* → (*node-rel* → *bool-rel*) → ⟨⟨⟨*node-rel*⟩*list-rel* $\times_r$ *node-rel*⟩*option-rel*⟩*nres-rel*
unfolding *find-path1-gen-def*[*abs-def*]
apply (*autoref* (*trace*,*keep-goal*))
done

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

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

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

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

∈ ⟨*node-rel*⟩*slg-rel* →*node-rel* → (*node-rel* → *bool-rel*) →
⟨⟨⟨*node-rel*⟩*list-rel* $\times_r$ *node-rel*⟩*Relators.option-rel*⟩*nres-rel*
proof −

note *EQI* = *GEN-OP-D*[*OF eq*]
note *HASHI* = *SIDE-GEN-ALGO-D*[*OF hash*]
note *DSZI* = *SIDE-GEN-ALGO-D*[*OF hash-dsz*]

note *R* = *find-path1-impl.refine*[*param-fo*, *THEN nres-relD*, *OF EQI HASHI DSZI*]

note *R*
also note *find-path1-gen-correct*
finally show *?thesis* **by** (*blast intro*: *nres-relI*)

qed

schematic-goal *find-path1-transfer-aux*:
RETURN ?c $\leq$ *find-path1-impl eqi hashi dszi E u P*
unfolding *find-path1-impl-def*
by *refine-transfer*
concrete-definition *find-path1-code* **for** *E u P* **uses** *find-path1-transfer-aux*
lemmas [*refine-transfer*] = *find-path1-code.refine*

end

1.36 Code Generation for GBG Lasso Finding Algorithm

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

1.37 Autoref Setup

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

lemma *locale-this*: *impl-lasso-loc mrel node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
by *unfold-locales*

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

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*
by *simp-all*
end

abbreviation *gGSi-rel* ≡ ⟨⟨*nat-rel*⟩*bs-set-rel*⟩*as-rel* $\times_r$ *GSi-rel*
abbreviation (**in** −) *ce-rel node-rel* ≡ ⟨⟨*node-rel*⟩*fun-set-rel* $\times_r$ ⟨*node-rel*⟩*fun-set-rel*⟩*option-rel*
abbreviation *goGSi-rel* ≡ *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*)∈*goGSi-rel*
unfolding *goinitial-impl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *goinitial-code*
uses *impl-lasso-loc.goinitial-code-aux*
lemmas [*autoref-rules*] = *goinitial-code.refine*[*OF locale-this*]

term *ginitial-impl*
schematic-goal *ginitial-code-aux*:
(*?c*,*ginitial-impl*)∈*node-rel* → *goGSi-rel* → *gGSi-rel*
unfolding *ginitial-impl-def*[*abs-def*] *initial-impl-def GS-initial-impl-def*

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *ginitial-code* **uses** *impl-lasso-loc.ginitial-code-aux*
lemmas [*autoref-rules*] = *ginitial-code.refine*[*OF locale-this*]

schematic-goal *gpath-is-empty-code-aux*:
(*?c*,*gpath-is-empty-impl*)∈*gGSi-rel*→*bool-rel*
unfolding *gpath-is-empty-impl-def*[*abs-def*] *path-is-empty-impl-def*

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gpath-is-empty-code*
uses *impl-lasso-loc.gpath-is-empty-code-aux*
lemmas [*autoref-rules*] = *gpath-is-empty-code.refine*[*OF locale-this*]

term *goBrk*
schematic-goal *goBrk-code-aux*: (*?c*,*goBrk-impl*)∈*goGSi-rel*→*ce-rel node-rel*
unfolding *goBrk-impl-def*[*abs-def*] *goBrk-impl-def*

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *goBrk-code* **uses** *impl-lasso-loc.goBrk-code-aux*
lemmas [*autoref-rules*] = *goBrk-code.refine*[*OF locale-this*]
thm *autoref-itype*(*1*)

term *gto-outer-impl*
schematic-goal *gto-outer-code-aux*:
(*?c*,*gto-outer-impl*)∈*ce-rel node-rel* → *gGSi-rel*→*goGSi-rel*
unfolding *gto-outer-impl-def*[*abs-def*] *gto-outer-impl-def*
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gto-outer-code*
uses *impl-lasso-loc.gto-outer-code-aux*
lemmas [*autoref-rules*] = *gto-outer-code.refine*[*OF locale-this*]

term *go-is-done-impl*
schematic-goal *go-is-done-code-aux*:
(*?c*,*go-is-done-impl*)∈*node-rel* → *goGSi-rel* → *bool-rel*
unfolding *go-is-done-impl-def*[*abs-def*] *is-done-oimpl-def*
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *go-is-done-code*
uses *impl-lasso-loc.go-is-done-code-aux*
lemmas [*autoref-rules*] = *go-is-done-code.refine*[*OF locale-this*]

schematic-goal *go-is-no-brk-code-aux*:
(*?c*,*go-is-no-brk-impl*)∈*goGSi-rel*→*bool-rel*
unfolding *go-is-no-brk-impl-def*[*abs-def*] *go-is-no-brk-impl-def*
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *go-is-no-brk-code*
uses *impl-lasso-loc.go-is-no-brk-code-aux*
lemmas [*autoref-rules*] = *go-is-no-brk-code.refine*[*OF locale-this*]

schematic-goal *gselect-edge-code-aux*: (*?c*,*gselect-edge-impl*)
∈ *gGSi-rel* → ⟨⟨*node-rel*⟩*option-rel* $\times_r$ *gGSi-rel*⟩*nres-rel*
unfolding *gselect-edge-impl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gselect-edge-code*
uses *impl-lasso-loc.gselect-edge-code-aux*
lemmas [*autoref-rules*] = *gselect-edge-code.refine*[*OF locale-this*]

term *gis-on-stack-impl*
schematic-goal *gis-on-stack-code-aux*:
(*?c*,*gis-on-stack-impl*)∈*node-rel* → *gGSi-rel* → *bool-rel*

unfolding *gis-on-stack-impl-def*[*abs-def*] *is-on-stack-impl-def*[*abs-def*]
GS.is-on-stack-impl-def[*abs-def*]

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gis-on-stack-code*
uses *impl-lasso-loc.gis-on-stack-code-aux*
lemmas [*autoref-rules*] = *gis-on-stack-code.refine*[*OF locale-this*]

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

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

schematic-goal *ce-code-aux*: (*?c*,*ce-impl*)
∈*gGSi-rel*→⟨*ce-rel node-rel* $\times_r$ *gGSi-rel*⟩*nres-rel*
unfolding *ce-impl-def*[*abs-def*] *on-stack-less-def*[*abs-def*]
on-stack-ge-def[*abs-def*]
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *ce-code* **uses** *impl-lasso-loc.ce-code-aux*
lemmas [*autoref-rules*] = *ce-code.refine*[*OF locale-this*]

schematic-goal *gis-done-code-aux*:
(*?c*,*gis-done-impl*)∈*node-rel*→*gGSi-rel*→*bool-rel*
unfolding *gis-done-impl-def*[*abs-def*] *is-done-impl-def GS.is-done-impl-def*

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gis-done-code* **uses** *impl-lasso-loc.gis-done-code-aux*
lemmas [*autoref-rules*] = *gis-done-code.refine*[*OF locale-this*]

schematic-goal *gpush-code-aux*:
(*?c*,*gpush-impl*)∈*node-rel* → *gGSi-rel*→*gGSi-rel*
unfolding *gpush-impl-def*[*abs-def*]

using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gpush-code* **uses** *impl-lasso-loc.gpush-code-aux*
lemmas [*autoref-rules*] = *gpush-code.refine*[*OF locale-this*]

schematic-goal *gpop-code-aux*: (*?c*,*gpop-impl*)∈*gGSi-rel*→⟨*gGSi-rel*⟩*nres-rel*
unfolding *gpop-impl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
by (*autoref* (*trace*,*keep-goal*))
concrete-definition (**in** −) *gpop-code* **uses** *impl-lasso-loc.gpop-code-aux*
lemmas [*autoref-rules*] = *gpop-code.refine*[*OF locale-this*]

schematic-goal *find-ce-code-aux*: (*?c*,*find-ce-impl*)∈⟨*ce-rel node-rel*⟩*nres-rel*
unfolding *find-ce-impl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*trace*,*keep-goal*))
done
concrete-definition (**in** −) *find-ce-code*
uses *impl-lasso-loc.find-ce-code-aux*
lemmas [*autoref-rules*] = *find-ce-code.refine*[*OF locale-this*]

schematic-goal *find-ce-tr-aux*: *RETURN ?c* ≤ *find-ce-code node-eq-impl node-hash-impl node-def-hash-size G-impl*
unfolding
find-ce-code-def
ginitial-code-def
gpath-is-empty-code-def
gselect-edge-code-def
gis-on-stack-code-def
gcollapse-code-def
last-is-acc-code-def
ce-code-def
gis-done-code-def
gpush-code-def
gpop-code-def
apply *refine-transfer*
done
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* **.**
lemma [*autoref-op-pat*]:
find-ce-spec ≡ *OP find-ce-spec*
by *auto*
end

theorem *find-ce-autoref*[*autoref-rules*]:
— Main Correctness theorem (inside locale)
shows (*find-ce-code node-eq-impl node-hash-impl node-def-hash-size G-impl*, *find-ce-spec*) ∈ ⟨*ce-rel node-rel*⟩*nres-rel*
proof −
note *find-ce-code.refine*[*OF locale-this*, *THEN nres-relD*]
also note *find-ce-impl-refine*
also note *find-ce-refine*
also note *find-ce-correct*
finally show *?thesis* **by** (*auto intro*: *nres-relI*)
qed

end

context *impl-lasso-loc*
begin

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

lemma [*autoref-op-pat*]:
reconstruct-reach ≡ *OP reconstruct-reach*
reconstruct-lasso ≡ *OP reconstruct-lasso*
by *auto*
end

schematic-goal *reconstruct-reach-code-aux*:
shows (*?c*, *reconstruct-reach*)∈⟨*node-rel*⟩*fun-set-rel* →
⟨*node-rel*⟩*fun-set-rel* →
⟨⟨*node-rel*⟩*list-rel* $\times_r$ *node-rel*⟩*nres-rel*
unfolding *reconstruct-lasso-def*[*abs-def*] *reconstruct-reach-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

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

schematic-goal *reconstruct-lasso-code-aux*:
shows (*?c*, *reconstruct-lasso*)∈⟨*node-rel*⟩*fun-set-rel* →
⟨*node-rel*⟩*fun-set-rel* →
⟨⟨*node-rel*⟩*list-rel* $\times_r$ ⟨*node-rel*⟩*list-rel*⟩*nres-rel*
unfolding *reconstruct-lasso-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]

apply (*autoref* (*keep-goal*, *trace*))
done

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

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

schematic-goal *find-lasso-code-aux*:
shows (*?c*::*?′c*, *find-lasso*)∈*?R*
unfolding *find-lasso-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done
concrete-definition (**in** −) *find-lasso-code*
uses *impl-lasso-loc.find-lasso-code-aux*
lemmas [*autoref-rules*] = *find-lasso-code.refine*[*OF locale-this*]

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

end

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

1.39 Main Correctness Theorem

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

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

assumes *B*: *igb-fr-graph G*
shows *RETURN* (*find-lasso-tr node-eq-impl node-hash-impl node-def-hash-size G-impl*)
$\leq \Downarrow$(*fl-rel node-rel*) (*igb-graph.find-lasso-spec G*)
proof −
from *B* **interpret** *igb-fr-graph G* **.**

have *I*: *impl-lasso-loc Re node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G*
apply *unfold-locales*
by *fact+*

then interpret *impl-lasso-loc Re node-rel node-eq-impl node-hash-impl node-def-hash-size G-impl G* **.**

note *find-lasso-tr.refine*[*OF I*]
also note *find-lasso-code.refine*[*OF I*, *THEN nres-relD*]
also note *find-lasso-correct*
finally show *?thesis* **.**
qed

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

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

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

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

lemma [*autoref-op-pat*]: *igb-graph.find-lasso-spec* ≡ *op-find-lasso-spec*
by *simp*

term *op-find-lasso-spec*

lemma [*autoref-itype*]:
op-find-lasso-spec
$::_i$ *i-igbg Ie I* $\rightarrow_i$ $\langle\langle\langle\langle I\rangle_i$*i-list*, $\langle I\rangle_i$*i-list*$\rangle_i$*i-prod*$\rangle_i$*i-option*$\rangle_i$*i-nres*
by *simp*

lemma *find-lasso-spec-autoref*[*autoref-rules-raw*]:
fixes *Re* **and** *node-rel* :: ($'vi \times 'v$) *set*
assumes *GR*: *SIDE-PRECOND* (*igb-fr-graph G*)
assumes *eq*: *GEN-OP node-eq-impl* (=) (*node-rel*→*node-rel*→*bool-rel*)
assumes *hash*: *SIDE-GEN-ALGO* (*is-bounded-hashcode node-rel node-eq-impl node-hash-impl*)
assumes *hash-dsz*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE*($'vi$) *node-def-hash-size*)
assumes *Gi*: (*G-impl*,*G*)∈*igbg-impl-rel-ext Re node-rel*
shows (*RETURN* (*find-lasso-tr node-eq-impl node-hash-impl node-def-hash-size G-impl*),

(*OP op-find-lasso-spec*
::: *igbg-impl-rel-ext Re node-rel* → ⟨*fl-rel node-rel*⟩*nres-rel*)$*G*) ∈ ⟨*fl-rel node-rel*⟩*nres-rel*
using *find-lasso-tr-correct*[*OF Gi GEN-OP-D*[*OF eq*] *SIDE-GEN-ALGO-D*[*OF hash*] *SIDE-GEN-ALGO-D*[*OF hash-dsz*]] **using** *GR*
apply (*fastforce intro*!: *nres-relI*)
done

end

end

2 Conclusion

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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