Correctness of a Set-based Algorithm for Computing Strongly Connected Components of a Graph

Stephan Merz and Vincent Trélat

March 19, 2025

Abstract

We prove the correctness of a sequential algorithm for computing maximal strongly connected components (SCCs) of a graph due to Vincent Bloemen.

Contents

1 Overview

Computing the maximal strongly connected components (SCCs) of a finite directed graph is a celebrated problem in the theory of graph algorithms. Although Tarjan's algorithm [5] is perhaps the best-known solution, there are many others. In his PhD thesis, Bloemen [1] presents an algorithm that is itself based on earlier algorithms by Munro [4] and Dijkstra [2]. Just like these algorithms, Bloemen's solution is based on enumerating SCCs in a depth-first traversal of the graph. Gabow's algorithm that has already been formalized in Isabelle [3] also falls into this category of solutions. Nevertheless, Bloemen goes on to present a parallel variant of the algorithm suitable for execution on multi-core processors, based on clever data structures that minimize locking.

In the following, we encode the sequential version of the algorithm in the proof assistant Isabelle/HOL, and prove its correctness. Bloemen's thesis briefly and informally explains why the algorithm is correct. Our proof expands on these arguments, making them completely formal. The encoding is based on a direct representation of the algorithm as a pair of mutually recursive functions; we are not aiming at extracting executable code.

theory *SCC-Bloemen-Sequential*
imports *Main*
begin

The record below represents the variables of the algorithm. Most variables correspond to those used in Bloemen's presentation. Thus, the variable $\mathcal{S}$ associates to every node the set of nodes that have already been determined to be part of the same SCC. A core invariant of the algorithm will be that this mapping represents equivalence classes of nodes: for all nodes v and w, we maintain the relationship

$v \in \mathcal{S}\ w \longleftrightarrow \mathcal{S}\ v = \mathcal{S}\ w$.

In an actual implementation of this algorithm, this variable could conveniently be represented by a union-find structure. Variable *stack* holds the list of roots of these (not yet maximal) SCCs, in depth-first order, *visited* and *explored* represent the nodes that have already been seen, respectively that have been completely explored, by the algorithm, and *sccs* is the set of maximal SCCs that the algorithm has found so far.

Additionally, the record holds some auxiliary variables that are used in the proof of correctness. In particular, *root* denotes the node on which the algorithm was called, *cstack* represents the call stack of the recursion of function *dfs*, and *vsuccs* stores the successors of each node that have already been visited by the function *dfss* that loops over all successors of a given node.

record $'v$ *env* =

root :: *'v*
$\mathcal{S}$ :: *'v* $\Rightarrow$ *'v set*
explored :: *'v set*
visited :: *'v set*
vsuccs :: *'v* $\Rightarrow$ *'v set*
sccs :: *'v set set*
stack :: *'v list*
cstack :: *'v list*

The algorithm is initially called with an environment that initializes the root node and trivializes all other components.

definition *init-env* **where**
init-env v = (|
root = *v*,
$\mathcal{S}$ = ($\lambda u.$ {*u*}),
explored = {},
visited = {},
vsuccs = ($\lambda u.$ {}),
sccs = {},
stack = [],
cstack = []
|)

— Make the simplifier expand let-constructions automatically.
declare *Let-def*[*simp*]

2 Auxiliary lemmas about lists

We use the precedence order on the elements that appear in a list. In particular, stacks are represented as lists, and a node x precedes another node y on the stack if x was pushed on the stack later than y.

definition *precedes* (‹- $\preceq$ - *in* -› [*100*,*100*,*100*] *39*) **where**
$x \preceq y$ *in xs* $\equiv \exists l\ r.\ xs = l$ @ $(x \# r) \wedge y \in$ *set* $(x \# r)$

lemma *precedes-mem*:
assumes $x \preceq y$ *in xs*
shows $x \in$ *set xs* $y \in$ *set xs*
⟨*proof*⟩

lemma *head-precedes*:
assumes $y \in$ *set* $(x \# xs)$
shows $x \preceq y$ *in* $(x \# xs)$
⟨*proof*⟩

lemma *precedes-in-tail*:
assumes $x \neq z$
shows $x \preceq y$ *in* $(z \# zs) \longleftrightarrow x \preceq y$ *in zs*

⟨*proof*⟩

lemma *tail-not-precedes*:
assumes $y \preceq x$ *in* $(x \# xs)$ $x \notin$ *set* xs
shows $x = y$
⟨*proof*⟩

lemma *split-list-precedes*:
assumes $y \in$ *set* $(ys \mathbin{@} [x])$
shows $y \preceq x$ *in* $(ys \mathbin{@} x \# xs)$
⟨*proof*⟩

lemma *precedes-refl* [*simp*]: $(x \preceq x$ *in* $xs) = (x \in$ *set* $xs)$
⟨*proof*⟩

lemma *precedes-append-left*:
assumes $x \preceq y$ *in* xs
shows $x \preceq y$ *in* $(ys \mathbin{@} xs)$
⟨*proof*⟩

lemma *precedes-append-left-iff*:
assumes $x \notin$ *set* ys
shows $x \preceq y$ *in* $(ys \mathbin{@} xs) \longleftrightarrow x \preceq y$ *in* xs (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *precedes-append-right*:
assumes $x \preceq y$ *in* xs
shows $x \preceq y$ *in* $(xs \mathbin{@} ys)$
⟨*proof*⟩

lemma *precedes-append-right-iff*:
assumes $y \notin$ *set* ys
shows $x \preceq y$ *in* $(xs \mathbin{@} ys) \longleftrightarrow x \preceq y$ *in* xs (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Precedence determines an order on the elements of a list, provided elements have unique occurrences. However, consider a list such as [*2*,*3*,*1*,*2*]: then 1 precedes 2 and 2 precedes 3, but 1 does not precede 3.

lemma *precedes-trans*:
assumes $x \preceq y$ *in* xs **and** $y \preceq z$ *in* xs **and** *distinct* xs
shows $x \preceq z$ *in* xs
⟨*proof*⟩

lemma *precedes-antisym*:
assumes $x \preceq y$ *in* xs **and** $y \preceq x$ *in* xs **and** *distinct* xs
shows $x = y$
⟨*proof*⟩

3 Finite directed graphs

We represent a graph as an Isabelle locale that identifies a finite set of vertices (of some base type $'v$) and associates to each vertex its set of successor vertices.

locale *graph* =
  fixes *vertices* :: $'v$ *set*
    and *successors* :: $'v \Rightarrow 'v$ *set*
  assumes *vfin*: *finite vertices*
    and *sclosed*: $\forall x \in$ *vertices*. *successors* $x \subseteq$ *vertices*

context *graph*
begin

abbreviation *edge* **where**
  edge $x\ y \equiv y \in$ *successors* x

We inductively define reachability of nodes in the graph.

inductive *reachable* **where**
  reachable-refl[*iff*]: *reachable* $x\ x$
| *reachable-succ*[*elim*]: $[\![$*edge* $x\ y$; *reachable* $y\ z]\!] \Longrightarrow$ *reachable* $x\ z$

lemma *reachable-edge*: *edge* $x\ y \Longrightarrow$ *reachable* $x\ y$
  ⟨*proof*⟩

lemma *succ-reachable*:
  assumes *reachable* $x\ y$ **and** *edge* $y\ z$
  shows *reachable* $x\ z$
  ⟨*proof*⟩

lemma *reachable-trans*:
  assumes *y*: *reachable* $x\ y$ **and** *z*: *reachable* $y\ z$
  shows *reachable* $x\ z$
  ⟨*proof*⟩

In order to derive a "reverse" induction rule for *reachable*, we define an alternative reachability predicate and prove that the two coincide.

inductive *reachable-end* **where**
  re-refl[*iff*]: *reachable-end* $x\ x$
| *re-succ*: $[\![$*reachable-end* $x\ y$; *edge* $y\ z]\!] \Longrightarrow$ *reachable-end* $x\ z$

lemma *succ-re*:
  assumes *y*: *edge* $x\ y$ **and** *z*: *reachable-end* $y\ z$
  shows *reachable-end* $x\ z$
  ⟨*proof*⟩

lemma *reachable-re*:
  assumes *reachable* $x\ y$

shows *reachable-end x y*
⟨*proof*⟩

lemma *re-reachable*:
assumes *reachable-end x y*
shows *reachable x y*
⟨*proof*⟩

lemma *reachable-end-induct*:
assumes *r*: *reachable x y*
and *base*: $\bigwedge x.\ P\ x\ x$
and *step*: $\bigwedge x\ y\ z.\ [\![P\ x\ y;\ edge\ y\ z]\!] \Longrightarrow P\ x\ z$
shows *P x y*
⟨*proof*⟩

We also need the following variant of reachability avoiding certain edges. More precisely, y is reachable from x avoiding a set E of edges if there exists a path such that no edge from E appears along the path.

inductive *reachable-avoiding* **where**
ra-refl[*iff*]: *reachable-avoiding x x E*
| *ra-succ*[*elim*]: [*reachable-avoiding x y E*; *edge y z*; $(y,z) \notin E$] ⟹ *reachable-avoiding x z E*

lemma *edge-ra*:
assumes *edge x y* **and** $(x,y) \notin E$
shows *reachable-avoiding x y E*
⟨*proof*⟩

lemma *ra-trans*:
assumes *1*: *reachable-avoiding x y E* **and** *2*: *reachable-avoiding y z E*
shows *reachable-avoiding x z E*
⟨*proof*⟩

lemma *ra-cases*:
assumes *reachable-avoiding x y E*
shows $x{=}y \vee (\exists z.\ z \in successors\ x \wedge (x,z) \notin E \wedge reachable\text{-}avoiding\ z\ y\ E)$
⟨*proof*⟩

lemma *ra-mono*:
assumes *reachable-avoiding x y E* **and** $E' \subseteq E$
shows *reachable-avoiding x y E′*
⟨*proof*⟩

lemma *ra-add-edge*:
assumes *reachable-avoiding x y E*
shows *reachable-avoiding x y* $(E \cup \{(v,w)\})$
$\vee$ (*reachable-avoiding x v* $(E \cup \{(v,w)\})$ $\wedge$ *reachable-avoiding w y* $(E \cup \{(v,w)\})$)
⟨*proof*⟩

Reachability avoiding some edges obviously implies reachability. Conversely, reachability implies reachability avoiding the empty set.

lemma *ra-reachable*:
reachable-avoiding x y E $\Longrightarrow$ *reachable x y*
⟨*proof*⟩

lemma *ra-empty*:
reachable-avoiding x y {} = *reachable x y*
⟨*proof*⟩

4 Strongly connected components

A strongly connected component is a set S of nodes such that any two nodes in S are reachable from each other. This concept is represented by the predicate *is-subscc* below. We are ultimately interested in non-empty, maximal strongly connected components, represented by the predicate *is-scc*.

definition *is-subscc* **where**
is-subscc S ≡ ∀ *x* ∈ *S*. ∀ *y* ∈ *S*. *reachable x y*

definition *is-scc* **where**
is-scc S ≡ *S* ≠ {} ∧ *is-subscc S* ∧ (∀ *S'*. *S* ⊆ *S'* ∧ *is-subscc S'* ⟶ *S'* = *S*)

lemma *subscc-add*:
assumes *is-subscc S* **and** *x* ∈ *S*
and *reachable x y* **and** *reachable y x*
shows *is-subscc* (*insert y S*)
⟨*proof*⟩

lemma *sccE*:
— Two nodes that are reachable from each other are in the same SCC.
assumes *is-scc S* **and** *x* ∈ *S*
and *reachable x y* **and** *reachable y x*
shows *y* ∈ *S*
⟨*proof*⟩

lemma *scc-partition*:
— Two SCCs that contain a common element are identical.
assumes *is-scc S* **and** *is-scc S'* **and** *x* ∈ *S* ∩ *S'*
shows *S* = *S'*
⟨*proof*⟩

5 Algorithm for computing strongly connected components

We now introduce our representation of Bloemen's algorithm in Isabelle/HOL. The auxiliary function *unite* corresponds to the inner while loop in Bloemen's

pseudo-code [1, p.32]. It is applied to two nodes v and w (and the environment e holding the current values of the program variables) when a loop is found, i.e. when w is a successor of v in the graph that has already been visited in the depth-first search. In that case, the root of the SCC of node w determined so far must appear below the root of v's SCC in the *stack* maintained by the algorithm. The effect of the function is to merge the SCCs of all nodes on the top of the stack above (and including) w. Node w's root will be the root of the merged SCC.

```
definition unite :: 'v ⇒ 'v ⇒ 'v env ⇒ 'v env where
  unite v w e ≡
    let pfx = takeWhile (λx. w ∉ 𝒮 e x) (stack e);
        sfx = dropWhile (λx. w ∉ 𝒮 e x) (stack e);
        cc = ⋃ { 𝒮 e x | x . x ∈ set pfx ∪ {hd sfx} }
    in  e(|𝒮 := λx. if x ∈ cc then cc else 𝒮 e x,
          stack := sfx|)
```

We now represent the algorithm as two mutually recursive functions *dfs* and *dfss* in Isabelle/HOL. The function *dfs* corresponds to Bloemen's function SetBased, whereas *dfss* corresponds to the forall loop over the successors of the node on which *dfs* was called. Instead of using global program variables in imperative style, our functions explicitly pass environments that hold the current values of these variables.

A technical complication in the development of the algorithm in Isabelle is the fact that the functions need not terminate when their pre-conditions (introduced below) are violated, for example when *dfs* is called for a node that was already visited previously. We therefore cannot prove termination at this point, but will later show that the explicitly given pre-conditions ensure termination.

```
function (domintros) dfs :: 'v ⇒ 'v env ⇒ 'v env
and dfss :: 'v ⇒ 'v env ⇒ 'v env where
  dfs v e =
  (let e1 = e(|visited := visited e ∪ {v},
               stack := (v # stack e),
               cstack := (v # cstack e)|);
       e' = dfss v e1
   in if v = hd(stack e')
      then e'(|sccs := sccs e' ∪ {𝒮 e' v},
               explored := explored e' ∪ (𝒮 e' v),
               stack := tl(stack e'),
               cstack := tl(cstack e')|)
      else e'(|cstack := tl(cstack e')|))
| dfss v e =
   (let vs = successors v − vsuccs e v
    in  if vs = {} then e
        else let w = SOME x. x ∈ vs;
                 e' = (if w ∈ explored e then e
```

```
                else if w ∉ visited e
                then dfs w e
                else unite v w e);
          e'' = (e'(|vsuccs :=
                    (λx. if x=v then vsuccs e' v ∪ {w}
                        else vsuccs e' x)|))
      in  dfss v e'')
⟨proof⟩
```

6 Definition of the predicates used in the correctness proof

Environments are partially ordered according to the following definition.

definition *sub-env* **where**
```
  sub-env e e' ≡
    root e' = root e
  ∧ visited e ⊆ visited e'
  ∧ explored e ⊆ explored e'
  ∧ (∀ v. vsuccs e v ⊆ vsuccs e' v)
  ∧ (∀ v. 𝒮 e v ⊆ 𝒮 e' v)
  ∧ (⋃ {𝒮 e v | v . v ∈ set (stack e)})
    ⊆ (⋃ {𝒮 e' v | v . v ∈ set (stack e')})
```

lemma *sub-env-trans*:
assumes *sub-env e e'* **and** *sub-env e' e''*
shows *sub-env e e''*
⟨*proof*⟩

The set *unvisited e u* contains all edges (a,b) such that node a is in the same SCC as node u and the edge has not yet been followed, in the sense represented by variable *vsuccs*.

definition *unvisited* **where**
```
  unvisited e u ≡
   {(a,b) | a b. a ∈ 𝒮 e u ∧ b ∈ successors a − vsuccs e a}
```

6.1 Main invariant

The following definition characterizes well-formed environments. This predicate will be shown to hold throughout the execution of the algorithm. In words, it asserts the following facts:

- Only nodes reachable from the root (for which the algorithm was originally called) are visited.
- The two stacks *stack* and *cstack* do not contain duplicate nodes, and *stack* contains a subset of the nodes on *cstack*, in the same order.

- Any node higher on the *stack* (i.e., that was pushed later) is reachable from nodes lower in the *stack*. This property also holds for nodes on the call stack, but this is not needed for the correctness proof.
- Every explored node, and every node on the call stack, has been visited.
- Nodes reachable from fully explored nodes have themselves been fully explored.
- The set *vsuccs e n*, for any node n, is a subset of n's successors, and all these nodes are in *visited*. The set is empty if $n \notin$ *visited*, and it contains all successors if n has been fully explored or if n has been visited, but is no longer on the call stack.
- The sets $\mathcal{S}$ *e n* represent an equivalence relation. The equivalence classes of nodes that have not yet been visited are singletons. Also, equivalence classes for two distinct nodes on the *stack* are disjoint because the stack only stores roots of SCCs, and the union of the equivalence classes for these root nodes corresponds to the set of live nodes, i.e. those nodes that have already been visited but not yet fully explored.
- More precisely, an equivalence class is represented on the stack by the oldest node in the sense of the call order: any node in the class that is still on the call stack precedes the representative on the call stack and was therefore pushed later.
- Equivalence classes represent the maximal available information about strong connectedness: nodes represented by some node n on the *stack* can reach some node m that is lower in the stack only by taking an edge from some node in n's equivalence class that has not yet been followed. (Remember that m can reach n by one of the previous conjuncts.)
- Equivalence classes represent partial SCCs in the sense of the predicate *is-subscc*. Variable *sccs* holds maximal SCCs in the sense of the predicate *is-scc*, and their union corresponds to the set of explored nodes.

definition *wf-env* **where**
wf-env e ≡
(∀ *n* ∈ *visited e*. *reachable* (*root e*) *n*)
∧ *distinct* (*stack e*)
∧ *distinct* (*cstack e*)
∧ (∀ *n m*. *n* ⪯ *m in stack e* ⟶ *n* ⪯ *m in cstack e*)
∧ (∀ *n m*. *n* ⪯ *m in stack e* ⟶ *reachable m n*)
∧ *explored e* ⊆ *visited e*
∧ *set* (*cstack e*) ⊆ *visited e*

$\wedge$ ($\forall\, n \in \mathit{explored}\ e.\ \forall\, m.\ \mathit{reachable}\ n\ m \longrightarrow m \in \mathit{explored}\ e$)
$\wedge$ ($\forall\, n.\ \mathit{vsuccs}\ e\ n \subseteq \mathit{successors}\ n \cap \mathit{visited}\ e$)
$\wedge$ ($\forall\, n.\ n \notin \mathit{visited}\ e \longrightarrow \mathit{vsuccs}\ e\ n = \{\}$)
$\wedge$ ($\forall\, n \in \mathit{explored}\ e.\ \mathit{vsuccs}\ e\ n = \mathit{successors}\ n$)
$\wedge$ ($\forall\, n \in \mathit{visited}\ e - \mathit{set}\ (\mathit{cstack}\ e).\ \mathit{vsuccs}\ e\ n = \mathit{successors}\ n$)
$\wedge$ ($\forall\, n\ m.\ m \in \mathcal{S}\ e\ n \longleftrightarrow (\mathcal{S}\ e\ n = \mathcal{S}\ e\ m)$)
$\wedge$ ($\forall\, n.\ n \notin \mathit{visited}\ e \longrightarrow \mathcal{S}\ e\ n = \{n\}$)
$\wedge$ ($\forall\, n \in \mathit{set}\ (\mathit{stack}\ e).\ \forall\, m \in \mathit{set}\ (\mathit{stack}\ e).\ n \neq m \longrightarrow \mathcal{S}\ e\ n \cap \mathcal{S}\ e\ m = \{\}$)
$\wedge \bigcup \{\mathcal{S}\ e\ n \mid n.\ n \in \mathit{set}\ (\mathit{stack}\ e)\} = \mathit{visited}\ e - \mathit{explored}\ e$
$\wedge$ ($\forall\, n \in \mathit{set}\ (\mathit{stack}\ e).\ \forall\, m \in \mathcal{S}\ e\ n.\ m \in \mathit{set}\ (\mathit{cstack}\ e) \longrightarrow m \preceq n\ \mathit{in}\ \mathit{cstack}\ e$)
$\wedge$ ($\forall\, n\ m.\ n \preceq m\ \mathit{in}\ \mathit{stack}\ e \wedge n \neq m \longrightarrow$
($\forall\, u \in \mathcal{S}\ e\ n.\ \neg\ \mathit{reachable\text{-}avoiding}\ u\ m\ (\mathit{unvisited}\ e\ n)$))
$\wedge$ ($\forall\, n.\ \mathit{is\text{-}subscc}\ (\mathcal{S}\ e\ n)$)
$\wedge$ ($\forall\, S \in \mathit{sccs}\ e.\ \mathit{is\text{-}scc}\ S$)
$\wedge \bigcup\ (\mathit{sccs}\ e) = \mathit{explored}\ e$

6.2 Consequences of the invariant

Since every node on the call stack is an element of *visited* and every node on the *stack* also appears on *cstack*, all these nodes are also in *visited*.

lemma *stack-visited*:
assumes *wf-env e* $n \in$ *set* (*stack e*)
shows $n \in$ *visited e*
⟨*proof*⟩

Classes represented on the stack consist of visited nodes that have not yet been fully explored.

lemma *stack-class*:
assumes *wf-env e* $n \in$ *set* (*stack e*) $m \in \mathcal{S}\ e\ n$
shows $m \in$ *visited e* − *explored e*
⟨*proof*⟩

Conversely, every such node belongs to some class represented on the stack.

lemma *visited-unexplored*:
assumes *wf-env e* $m \in$ *visited e* $m \notin$ *explored e*
obtains n **where** $n \in$ *set* (*stack e*) $m \in \mathcal{S}\ e\ n$
⟨*proof*⟩

Every node belongs to its own equivalence class.

lemma *S-reflexive*:
assumes *wf-env e*
shows $n \in \mathcal{S}\ e\ n$
⟨*proof*⟩

No node on the stack has been fully explored.

lemma *stack-unexplored*:
assumes *1*: *wf-env e*

and *2*: $n \in set\ (stack\ e)$
and *3*: $n \in explored\ e$
shows P
⟨*proof*⟩

If w is reachable from visited node v, but no unvisited successor of a node reachable from v can reach w, then w must be visited.

lemma *reachable-visited*:
assumes *e*: *wf-env e*
and *v*: $v \in visited\ e$
and *w*: *reachable v w*
and *s*: $\forall\, n \in visited\ e.\ \forall\, m \in successors\ n - vsuccs\ e\ n.$
$reachable\ v\ n \longrightarrow \neg\ reachable\ m\ w$
shows $w \in visited\ e$
⟨*proof*⟩

Edges towards explored nodes do not contribute to reachability of unexplored nodes avoiding some set of edges.

lemma *avoiding-explored*:
assumes *e*: *wf-env e*
and *xy*: *reachable-avoiding x y E*
and *y*: $y \notin explored\ e$
and *w*: $w \in explored\ e$
shows *reachable-avoiding x y* $(E \cup \{(v,w)\})$
⟨*proof*⟩

6.3 Pre- and post-conditions of function *dfs*

Function *dfs* should be called for a well-formed environment and a node v that has not yet been visited and that is reachable from the root node, as well as from all nodes in the stack. No outgoing edges from node v have yet been followed.

definition *pre-dfs* **where**
pre-dfs v e ≡
wf-env e
∧ $v \notin visited\ e$
∧ *reachable* (*root e*) *v*
∧ $vsuccs\ e\ v = \{\}$
∧ $(\forall\, n \in set\ (stack\ e).\ reachable\ n\ v)$

Function *dfs* maintains the invariant *wf-env* and returns an environment e' that extends the input environment e. Node v has been visited and all its outgoing edges have been followed. Because the algorithm works in depth-first fashion, no new outgoing edges of nodes that had already been visited in the input environment have been followed, and the stack of e' is a suffix of the one of e such that v is still reachable from all nodes on the stack. The stack may have been shortened because SCCs represented at the top of the

stack may have been merged. The call stack is reestablished as it was in e. There are two possible outcomes of the algorithm:

- Either v has been fully explored, in which case the stacks of e and e' are the same, and the equivalence classes of all nodes represented on the stack are unchanged. This corresponds to the case where v is the root node of its (maximal) SCC.
- Alternatively, the stack of e' must be non-empty and v must be represented by the node at the top of the stack. The SCCs of the nodes lower on the stack are unchanged. This corresponds to the case where v is not the root node of its SCC, but some SCCs at the top of the stack may have been merged.

definition *post-dfs* **where**
$$
\begin{aligned}
&\mathit{post\text{-}dfs}\ v\ e\ e' \equiv \\
&\quad \mathit{wf\text{-}env}\ e' \\
&\wedge\ v \in \mathit{visited}\ e' \\
&\wedge\ \mathit{sub\text{-}env}\ e\ e' \\
&\wedge\ \mathit{vsuccs}\ e'\ v = \mathit{successors}\ v \\
&\wedge\ (\forall\, w \in \mathit{visited}\ e.\ \mathit{vsuccs}\ e'\ w = \mathit{vsuccs}\ e\ w) \\
&\wedge\ (\forall\, n \in \mathit{set}\ (\mathit{stack}\ e').\ \mathit{reachable}\ n\ v) \\
&\wedge\ (\exists\, \mathit{ns}.\ \mathit{stack}\ e = \mathit{ns}\ @\ (\mathit{stack}\ e')) \\
&\wedge\ (\quad (v \in \mathit{explored}\ e' \wedge \mathit{stack}\ e' = \mathit{stack}\ e \\
&\qquad \wedge\ (\forall\, n \in \mathit{set}\ (\mathit{stack}\ e').\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n)) \\
&\quad \vee\ (\mathit{stack}\ e' \neq [] \wedge v \in \mathcal{S}\ e'\ (\mathit{hd}\ (\mathit{stack}\ e')) \\
&\qquad \wedge\ (\forall\, n \in \mathit{set}\ (\mathit{tl}\ (\mathit{stack}\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n))) \\
&\wedge\ \mathit{cstack}\ e' = \mathit{cstack}\ e
\end{aligned}
$$

The initial environment is easily seen to satisfy *dfs*'s pre-condition.

lemma *init-env-pre-dfs*: $\mathit{pre\text{-}dfs}\ v\ (\mathit{init\text{-}env}\ v)$
⟨*proof*⟩

Any node represented by the top stack element of the input environment is still represented by the top element of the output stack.

lemma *dfs-S-hd-stack*:
assumes *wf*: *wf-env e*
and *post*: *post-dfs v e e'*
and *n*: $\mathit{stack}\ e \neq [] \ n \in \mathcal{S}\ e\ (\mathit{hd}\ (\mathit{stack}\ e))$
shows $\mathit{stack}\ e' \neq [] \ n \in \mathcal{S}\ e'\ (\mathit{hd}\ (\mathit{stack}\ e'))$
⟨*proof*⟩

Function *dfs* leaves the SCCs represented by elements in the (new) tail of the *stack* unchanged.

lemma *dfs-S-tl-stack*:
assumes *post*: *post-dfs v e e'*
and *nempty*: $\mathit{stack}\ e \neq []$
shows $\mathit{stack}\ e' \neq [] \ \forall\, n \in \mathit{set}\ (\mathit{tl}\ (\mathit{stack}\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$
⟨*proof*⟩

6.4 Pre- and post-conditions of function *dfss*

The pre- and post-conditions of function *dfss* correspond to the invariant of the loop over all outgoing edges from node v. The environment must be well-formed, node v must be visited and represented by the top element of the (non-empty) stack. Node v must be reachable from all nodes on the stack, and it must be the top node on the call stack. All outgoing edges of node v that have already been followed must either lead to completely explored nodes (that are no longer represented on the stack) or to nodes that are part of the same SCC as v.

definition *pre-dfss* **where**
$$
\begin{aligned}
&\mathit{pre\text{-}dfss}\ v\ e \equiv \\
&\quad \mathit{wf\text{-}env}\ e \\
&\wedge\ v \in \mathit{visited}\ e \\
&\wedge\ (\mathit{stack}\ e \neq []) \\
&\wedge\ (v \in \mathcal{S}\ e\ (\mathit{hd}\ (\mathit{stack}\ e))) \\
&\wedge\ (\forall\, w \in \mathit{vsuccs}\ e\ v.\ w \in \mathit{explored}\ e \cup \mathcal{S}\ e\ (\mathit{hd}\ (\mathit{stack}\ e))) \\
&\wedge\ (\forall\, n \in \mathit{set}\ (\mathit{stack}\ e).\ \mathit{reachable}\ n\ v) \\
&\wedge\ (\exists\, \mathit{ns}.\ \mathit{cstack}\ e = v\ \#\ \mathit{ns})
\end{aligned}
$$

The post-condition establishes that all outgoing edges of node v have been followed. As for function *dfs*, no new outgoing edges of previously visited nodes have been followed. Also as before, the new stack is a suffix of the old one, and the call stack is restored. In case node v is still on the stack (and therefore is the root node of its SCC), no node that is lower on the stack can be reachable from v. This condition guarantees the maximality of the computed SCCs.

definition *post-dfss* **where**
$$
\begin{aligned}
&\mathit{post\text{-}dfss}\ v\ e\ e' \equiv \\
&\quad \mathit{wf\text{-}env}\ e' \\
&\wedge\ \mathit{vsuccs}\ e'\ v = \mathit{successors}\ v \\
&\wedge\ (\forall\, w \in \mathit{visited}\ e - \{v\}.\ \mathit{vsuccs}\ e'\ w = \mathit{vsuccs}\ e\ w) \\
&\wedge\ \mathit{sub\text{-}env}\ e\ e' \\
&\wedge\ (\forall\, w \in \mathit{successors}\ v.\ w \in \mathit{explored}\ e' \cup \mathcal{S}\ e'\ (\mathit{hd}\ (\mathit{stack}\ e'))) \\
&\wedge\ (\forall\, n \in \mathit{set}\ (\mathit{stack}\ e').\ \mathit{reachable}\ n\ v) \\
&\wedge\ (\mathit{stack}\ e' \neq []) \\
&\wedge\ (\exists\, \mathit{ns}.\ \mathit{stack}\ e = \mathit{ns}\ @\ (\mathit{stack}\ e')) \\
&\wedge\ v \in \mathcal{S}\ e'\ (\mathit{hd}\ (\mathit{stack}\ e')) \\
&\wedge\ (\forall\, n \in \mathit{set}\ (\mathit{tl}\ (\mathit{stack}\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n) \\
&\wedge\ (\mathit{hd}\ (\mathit{stack}\ e') = v \longrightarrow (\forall\, n \in \mathit{set}\ (\mathit{tl}\ (\mathit{stack}\ e')).\ \neg\ \mathit{reachable}\ v\ n)) \\
&\wedge\ \mathit{cstack}\ e' = \mathit{cstack}\ e
\end{aligned}
$$

7 Proof of partial correctness

7.1 Lemmas about function *unite*

We start by establishing a few lemmas about function *unite* in the context where it is called.

lemma *unite-stack*:
  fixes $e\ v\ w$
  defines $e' \equiv unite\ v\ w\ e$
  assumes *wf*: *wf-env e*
    and *w*: $w \in successors\ v\ w \notin vsuccs\ e\ v\ w \in visited\ e\ w \notin explored\ e$
  obtains *pfx* **where** $stack\ e = pfx\ @\ (stack\ e')$
    $stack\ e' \neq []$
    $let\ cc = \bigcup\ \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$
     $in\ \mathcal{S}\ e' = (\lambda x.\ if\ x \in cc\ then\ cc\ else\ \mathcal{S}\ e\ x)$
    $w \in \mathcal{S}\ e'\ (hd\ (stack\ e'))$
⟨*proof*⟩

Function *unite* leaves intact the equivalence classes represented by the tail of the new stack.

lemma *unite-S-tl*:
  fixes $e\ v\ w$
  defines $e' \equiv unite\ v\ w\ e$
  assumes *wf*: *wf-env e*
    and *w*: $w \in successors\ v\ w \notin vsuccs\ e\ v\ w \in visited\ e\ w \notin explored\ e$
    and *n*: $n \in set\ (tl\ (stack\ e'))$
  shows $\mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$
⟨*proof*⟩

The stack of the result of *unite* represents the same vertices as the input stack, potentially in fewer equivalence classes.

lemma *unite-S-equal*:
  fixes $e\ v\ w$
  defines $e' \equiv unite\ v\ w\ e$
  assumes *wf*: *wf-env e*
    and *w*: $w \in successors\ v\ w \notin vsuccs\ e\ v\ w \in visited\ e\ w \notin explored\ e$
  shows $(\bigcup\ \{\mathcal{S}\ e'\ n \mid n.\ n \in set\ (stack\ e')\}) = (\bigcup\ \{\mathcal{S}\ e\ n \mid n.\ n \in set\ (stack\ e)\})$
⟨*proof*⟩

The head of the stack represents a (not necessarily maximal) SCC.

lemma *unite-subscc*:
  fixes $e\ v\ w$
  defines $e' \equiv unite\ v\ w\ e$
  assumes *pre*: *pre-dfss v e*
    and *w*: $w \in successors\ v\ w \notin vsuccs\ e\ v\ w \in visited\ e\ w \notin explored\ e$
   shows *is-subscc* $(\mathcal{S}\ e'\ (hd\ (stack\ e')))$
⟨*proof*⟩

The environment returned by function *unite* extends the input environment.

lemma *unite-sub-env*:
fixes *e v w*
defines $e' \equiv$ *unite v w e*
assumes *pre*: *pre-dfss v e*
and *w*: $w \in$ *successors v* $w \notin$ *vsuccs e v* $w \in$ *visited e* $w \notin$ *explored e*
shows *sub-env e e'*
⟨*proof*⟩

The environment returned by function *unite* is well-formed.

lemma *unite-wf-env*:
fixes *e v w*
defines $e' \equiv$ *unite v w e*
assumes *pre*: *pre-dfss v e*
and *w*: $w \in$ *successors v* $w \notin$ *vsuccs e v* $w \in$ *visited e* $w \notin$ *explored e*
shows *wf-env e'*
⟨*proof*⟩

7.2 Lemmas establishing the pre-conditions

The precondition of function *dfs* ensures the precondition of *dfss* at the call of that function.

lemma *pre-dfs-pre-dfss*:
assumes *pre-dfs v e*
shows *pre-dfss v* (*e*(|*visited* := *visited e* ∪ {*v*},
stack := *v* # *stack e*,
cstack := *v* # *cstack e*|))
(**is** *pre-dfss v ?e'*)
⟨*proof*⟩

Similarly, we now show that the pre-conditions of the different function calls in the body of function *dfss* are satisfied. First, it is very easy to see that the pre-condition of *dfs* holds at the call of that function.

lemma *pre-dfss-pre-dfs*:
assumes *pre-dfss v e* **and** $w \notin$ *visited e* **and** $w \in$ *successors v*
shows *pre-dfs w e*
⟨*proof*⟩

The pre-condition of *dfss* holds when the successor considered in the current iteration has already been explored.

lemma *pre-dfss-explored-pre-dfss*:
fixes *e v w*
defines $e'' \equiv$ *e*(|*vsuccs* := (λ*x*. *if x=v then vsuccs e v* ∪ {*w*} *else vsuccs e x*)|)
assumes *1*: *pre-dfss v e* **and** *2*: $w \in$ *successors v* **and** *3*: $w \in$ *explored e*
shows *pre-dfss v e''*
⟨*proof*⟩

The call to *dfs* establishes the pre-condition for the recursive call to *dfss* in the body of *dfss*.

lemma *pre-dfss-post-dfs-pre-dfss*:
fixes *e v w*
defines $e' \equiv dfs\ w\ e$
defines $e'' \equiv e'(\!|vsuccs := (\lambda x.\ if\ x{=}v\ then\ vsuccs\ e'\ v \cup \{w\}\ else\ vsuccs\ e'\ x)|\!)$
assumes *pre*: *pre-dfss v e*
and *w*: $w \in successors\ v\ w \notin visited\ e$
and *post*: *post-dfs w e e'*
shows *pre-dfss v e''*
⟨*proof*⟩

Finally, the pre-condition for the recursive call to *dfss* at the end of the body of function *dfss* also holds if *unite* was applied.

lemma *pre-dfss-unite-pre-dfss*:
fixes *e v w*
defines $e' \equiv unite\ v\ w\ e$
defines $e'' \equiv e'(\!|vsuccs := (\lambda x.\ if\ x{=}v\ then\ vsuccs\ e'\ v \cup \{w\}\ else\ vsuccs\ e'\ x)|\!)$
assumes *pre*: *pre-dfss v e*
and *w*: $w \in successors\ v\ w \notin vsuccs\ e\ v\ w \in visited\ e\ w \notin explored\ e$
shows *pre-dfss v e''*
⟨*proof*⟩

7.3 Lemmas establishing the post-conditions

Assuming the pre-condition of function *dfs* and the post-condition of the call to *dfss* in the body of that function, the post-condition of *dfs* is established.

lemma *pre-dfs-implies-post-dfs*:
fixes *v e*
defines $e1 \equiv e(\!|visited := visited\ e \cup \{v\},$
$stack := (v\ \#\ stack\ e),$
$cstack{:=}(v\ \#\ cstack\ e)|\!)$
defines $e' \equiv dfss\ v\ e1$
defines $e'' \equiv e'(\!|\ cstack := tl(cstack\ e')|\!)$
assumes *1*: *pre-dfs v e*
and *2*: *dfs-dfss-dom* $(Inl(v,\ e))$
and *3*: *post-dfss v e1 e'*
shows *post-dfs v e* (*dfs v e*)
⟨*proof*⟩

The following lemma is central for proving partial correctness: assuming termination (represented by the predicate *dfs-dfss-dom*) and the pre-condition of the functions, both *dfs* and *dfss* establish their post-conditions. The first part of the theorem follows directly from the preceding lemma and the computational induction rule generated by Isabelle, the second part is proved directly, distinguishing the different cases in the definition of function *dfss*.

lemma *pre-post*:
shows
$[\![dfs\text{-}dfss\text{-}dom\ (Inl(v,e));\ pre\text{-}dfs\ v\ e]\!] \Longrightarrow post\text{-}dfs\ v\ e\ (dfs\ v\ e)$

⟦*dfs-dfss-dom* (*Inr*(*v*,*e*)); *pre-dfss v e*⟧ ⟹ *post-dfss v e* (*dfss v e*)
⟨*proof*⟩

We can now show partial correctness of the algorithm: applied to some node v and the empty environment, it computes the set of strongly connected components in the subgraph reachable from node v. In particular, if v is a root of the graph, the algorithm computes the set of SCCs of the graph.

theorem *partial-correctness*:
fixes *v*
defines *e* ≡ *dfs v* (*init-env v*)
assumes *dfs-dfss-dom* (*Inl* (*v*, *init-env v*))
shows *sccs e* = {*S* . *is-scc S* ∧ (∀ *n*∈*S*. *reachable v n*)}
(**is** - = *?rhs*)
⟨*proof*⟩

8 Proof of termination and total correctness

We define a binary relation on the arguments of functions *dfs* and *dfss*, and prove that this relation is well-founded and that all calls within the function bodies respect the relation, assuming that the pre-conditions of the initial function call are satisfied. By well-founded induction, we conclude that the pre-conditions of the functions are sufficient to ensure termination.

Following the internal representation of the two mutually recursive functions in Isabelle as a single function on the disjoint sum of the types of arguments, our relation is defined as a set of argument pairs injected into the sum type. The left injection *Inl* takes arguments of function *dfs*, the right injection *Inr* takes arguments of function *dfss*.[1] The conditions on the arguments in the definition of the relation overapproximate the arguments in the actual calls.

definition *dfs-dfss-term*::((′*v* × ′*v env* + ′*v* × ′*v env*) × (′*v* × ′*v env* + ′*v* × ′*v env*)) *set* **where**
dfs-dfss-term ≡
{ (*Inr*(*v*, *e1*), *Inl*(*v*, *e*)) | *v e e1*.
v ∈ *vertices* − *visited e* ∧ *visited e1* = *visited e* ∪ {*v*} }
∪ { (*Inl*(*w*, *e*), *Inr*(*v*, *e*)) | *v w e*. *v* ∈ *vertices*}
∪ { (*Inr*(*v*, *e*′′), *Inr*(*v*, *e*)) | *v e e*′′.
v ∈ *vertices* ∧ *sub-env e e*′′
∧ (∃ *w* ∈ *vertices*. *w* ∉ *vsuccs e v* ∧ *w* ∈ *vsuccs e*′′ *v*)}

Informally, termination is ensured because at each call, either a new vertex is visited (hence the complement of the set of visited nodes w.r.t. the finite set of vertices decreases) or a new successor is added to the set *vsuccs e v* of some vertex *v*.

[1] Note that the types of the arguments of *dfs* and *dfss* are actually identical. We nevertheless use the sum type in order to remember the function that was called.

In order to make this argument formal, we inject the argument tuples that appear in our relation into tuples consisting of the sets mentioned in the informal argument. However, there is one added complication because the call of *dfs* from *dfss* does not immediately add the vertex to the set of visited nodes (this happens only at the beginning of function *dfs*). We therefore add a third component of 0 or 1 to these tuples, reflecting the fact that there can only be one call of *dfs* from *dfss* for a given vertex *v*.

fun *dfs-dfss-to-tuple* **where**
  dfs-dfss-to-tuple (*Inl*(*v*::'*v*, *e*::'*v env*)) =
   (*vertices* − *visited e*, *vertices* × *vertices* − {(*u*,*u'*) | *u u'*. *u'* ∈ *vsuccs e u*}, *0*)
| *dfs-dfss-to-tuple* (*Inr*(*v*::'*v*, *e*::'*v env*)) =
  (*vertices* − *visited e*, *vertices* × *vertices* − {(*u*,*u'*) | *u u'*. *u'* ∈ *vsuccs e u*}, *1*::*nat*)

The triples defined in this way can be ordered lexicographically (with the first two components ordered as finite subsets and the third one following the predecessor relation on natural numbers). We prove that the injection of the above relation into sets of triples respects the lexicographic ordering and conclude that our relation is well-founded.

lemma *wf-term*: *wf dfs-dfss-term*
⟨*proof*⟩

The following theorem establishes sufficient conditions that ensure termination of the two functions *dfs* and *dfss*. The proof proceeds by well-founded induction using the relation *dfs-dfss-term*. Isabelle represents the termination domains of the functions by the predicate *dfs-dfss-dom* and generates a theorem *dfs-dfss.domintros* for proving membership of arguments in the termination domains. The actual formulation is a litte technical because the mutual induction must again be encoded in a single induction argument over the sum type representing the arguments of both functions.

theorem *dfs-dfss-termination*:
  ⟦*v* ∈ *vertices* ; *pre-dfs v e*⟧ ⟹ *dfs-dfss-dom*(*Inl*(*v*, *e*))
  ⟦*v* ∈ *vertices* ; *pre-dfss v e*⟧ ⟹ *dfs-dfss-dom*(*Inr*(*v*, *e*))
⟨*proof*⟩

Putting everything together, we prove the total correctness of the algorithm when applied to some (root) vertex.

theorem *correctness*:
  assumes *v* ∈ *vertices*
  shows *sccs* (*dfs v* (*init-env v*)) = {*S* . *is-scc S* ∧ (∀ *n*∈*S*. *reachable v n*)}
  ⟨*proof*⟩

end
end

References

[1] V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. PhD thesis, University of Twente, Enschede, The Netherlands, 2019.

[2] E. W. Dijkstra. Finding the maximum strong components in a directed graph. In *Selected Writings in Computing: A Personal Perspective*, Texts and Monographs in Computer Science, pages 22–30. Springer, 1982.

[3] P. Lammich. Verified efficient implementation of gabow's strongly connected components algorithm. *Archive of Formal Proofs*, May 2014. https://isa-afp.org/entries/Gabow_SCC.html, Formal proof development.

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

[5] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.