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* ⇒ *'v set*
explored :: *'v set*
visited :: *'v set*
vsuccs :: *'v* ⇒ *'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*,
$\mathcal{S}$ = (λ*u*. {*u*}),
explored = {},
visited = {},
vsuccs = (λ*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* (‹- ⪯ - *in* -› [*100*,*100*,*100*] *39*) **where**
x ⪯ *y in xs* ≡ ∃ *l r*. *xs* = *l* @ (*x* # *r*) ∧ *y* ∈ *set* (*x* # *r*)

lemma *precedes-mem*:
assumes *x* ⪯ *y in xs*
shows *x* ∈ *set xs y* ∈ *set xs*
using *assms* **unfolding** *precedes-def* **by** *auto*

lemma *head-precedes*:
assumes *y* ∈ *set* (*x* # *xs*)
shows *x* ⪯ *y in* (*x* # *xs*)
using *assms* **unfolding** *precedes-def* **by** *force*

lemma *precedes-in-tail*:
assumes *x* ≠ *z*
shows *x* ⪯ *y in* (*z* # *zs*) ⟷ *x* ⪯ *y in zs*

using *assms* **unfolding** *precedes-def* **by** (*auto simp*: *Cons-eq-append-conv*)

lemma *tail-not-precedes*:
assumes $y \preceq x$ *in* (x # *xs*) $x \notin$ *set xs*
shows $x = y$
using *assms* **unfolding** *precedes-def*
by (*metis Cons-eq-append-conv Un-iff list.inject set-append*)

lemma *split-list-precedes*:
assumes $y \in$ *set* (*ys* @ [x])
shows $y \preceq x$ *in* (*ys* @ x # *xs*)
using *assms* **unfolding** *precedes-def*
by (*metis append-Cons append-assoc in-set-conv-decomp*
rotate1.simps(*2*) *set-ConsD set-rotate1*)

lemma *precedes-refl* [*simp*]: ($x \preceq x$ *in xs*) = ($x \in$ *set xs*)
proof
assume $x \preceq x$ *in xs* **thus** $x \in$ *set xs*
by (*simp add*: *precedes-mem*)
next
assume $x \in$ *set xs*
from *this*[*THEN split-list*] **show** $x \preceq x$ *in xs*
unfolding *precedes-def* **by** *auto*
qed

lemma *precedes-append-left*:
assumes $x \preceq y$ *in xs*
shows $x \preceq y$ *in* (*ys* @ *xs*)
using *assms* **unfolding** *precedes-def* **by** (*metis append.assoc*)

lemma *precedes-append-left-iff*:
assumes $x \notin$ *set ys*
shows $x \preceq y$ *in* (*ys* @ *xs*) $\longleftrightarrow$ $x \preceq y$ *in xs* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *l r* **where** *lr*: *ys* @ *xs* = *l* @ (x # *r*) $y \in$ *set* (x # *r*)
unfolding *precedes-def* **by** *blast*
then obtain *us* **where**
(*ys* = *l* @ *us* $\wedge$ *us* @ *xs* = x # *r*) $\vee$ (*ys* @ *us* = *l* $\wedge$ *xs* = *us* @ (x # *r*))
by (*auto simp*: *append-eq-append-conv2*)
thus *?rhs*
proof
assume *us*: *ys* = *l* @ *us* $\wedge$ *us* @ *xs* = x # *r*
with *assms* **have** *us* = []
by (*metis Cons-eq-append-conv in-set-conv-decomp*)
with *us lr* **show** *?rhs*
unfolding *precedes-def* **by** *auto*
next
assume *us*: *ys* @ *us* = *l* $\wedge$ *xs* = *us* @ (x # *r*)

```
    with ‹y ∈ set (x # r)› show ?rhs
      unfolding precedes-def by blast
  qed
next
  assume ?rhs thus ?lhs by (rule precedes-append-left)
qed

lemma precedes-append-right:
  assumes x ≼ y in xs
  shows x ≼ y in (xs @ ys)
  using assms unfolding precedes-def by force

lemma precedes-append-right-iff:
  assumes y ∉ set ys
  shows x ≼ y in (xs @ ys) ⟷ x ≼ y in xs (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain l r where lr: xs @ ys = l @ (x # r) y ∈ set (x # r)
    unfolding precedes-def by blast
  then obtain us where
    (xs = l @ us ∧ us @ ys = x # r) ∨ (xs @ us = l ∧ ys = us @ (x # r))
    by (auto simp: append-eq-append-conv2)
  thus ?rhs
  proof
    assume us: xs = l @ us ∧ us @ ys = x # r
    with ‹y ∈ set (x # r)› assms show ?rhs
      unfolding precedes-def by (metis Cons-eq-append-conv Un-iff set-append)
  next
    assume us: xs @ us = l ∧ ys = us @ (x # r)
    with ‹y ∈ set (x # r)› assms
    show ?rhs by auto — contradiction
  qed
next
  assume ?rhs thus ?lhs by (rule precedes-append-right)
qed
```

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 ≼ y in xs and y ≼ z in xs and distinct xs
  shows x ≼ z in xs
  using assms unfolding precedes-def
 by (metis assms(2) disjoint-iff distinct-append precedes-append-left-iff precedes-mem(2))

lemma precedes-antisym:
  assumes x ≼ y in xs and y ≼ x in xs and distinct xs
  shows x = y
proof –
```

```
  from ‹x ≼ y in xs› ‹distinct xs› obtain as bs where
    1: xs = as @ (x # bs) y ∈ set (x # bs) y ∉ set as
    unfolding precedes-def by force
  from ‹y ≼ x in xs› ‹distinct xs› obtain cs ds where
    2: xs = cs @ (y # ds) x ∈ set (y # ds) x ∉ set cs
    unfolding precedes-def by force
  from 1 2 have as @ (x # bs) = cs @ (y # ds)
    by simp
  then obtain zs where
    (as = cs @ zs ∧ zs @ (x # bs) = y # ds)
     ∨ (as @ zs = cs ∧ x # bs = zs @ (y # ds))  (is ?P ∨ ?Q)
    by (auto simp: append-eq-append-conv2)
  then show ?thesis
  proof
    assume ?P with ‹y ∉ set as› show ?thesis
      by (cases zs) auto
  next
    assume ?Q with ‹x ∉ set cs› show ?thesis
      by (cases zs) auto
  qed
qed
```

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 ⇒ 'v set
  assumes vfin: finite vertices
    and sclosed: ∀ x ∈ vertices. successors x ⊆ vertices

context graph
begin

abbreviation edge where
  edge x y ≡ y ∈ 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⟧ ⟹ reachable x z

lemma reachable-edge: edge x y ⟹ reachable x y
  by auto

lemma succ-reachable:
```

assumes *reachable x y* **and** *edge y z*
shows *reachable x z*
using *assms* **by** *induct auto*

lemma *reachable-trans*:
assumes *y*: *reachable x y* **and** *z*: *reachable y z*
shows *reachable x z*
using *assms* **by** *induct auto*

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*
using *z y* **by** (*induction*) (*auto intro*: *re-succ*)

lemma *reachable-re*:
assumes *reachable x y*
shows *reachable-end x y*
using *assms* **by** (*induction*) (*auto intro*: *succ-re*)

lemma *re-reachable*:
assumes *reachable-end x y*
shows *reachable x y*
using *assms* **by** (*induction*) (*auto intro*: *succ-reachable*)

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*
using *r*[*THEN reachable-re*] **proof** (*induction*)
case (*re-refl x*)
from *base* **show** *?case* **.**
next
case (*re-succ x y z*)
with *step* **show** *?case* **by** *blast*
qed

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*) ∉ *E*⟧ ⟹ *reachable-avoiding x z E*

lemma *edge-ra*:
  assumes *edge x y* **and** (*x*,*y*) ∉ *E*
  shows *reachable-avoiding x y E*
  using *assms* **by** (*meson reachable-avoiding.simps*)

lemma *ra-trans*:
  assumes *1*: *reachable-avoiding x y E* **and** *2*: *reachable-avoiding y z E*
  shows *reachable-avoiding x z E*
  using *2 1* **by** *induction auto*

lemma *ra-cases*:
  assumes *reachable-avoiding x y E*
  shows *x*=*y* ∨ (∃ *z*. *z* ∈ *successors x* ∧ (*x*,*z*) ∉ *E* ∧ *reachable-avoiding z y E*)
using *assms* **proof** (*induction*)
  case (*ra-refl x S*)
  then show *?case* **by** *simp*
next
  case (*ra-succ x y S z*)
  then show *?case*
    by (*metis ra-refl reachable-avoiding.ra-succ*)
qed

lemma *ra-mono*:
  assumes *reachable-avoiding x y E* **and** *E*′ ⊆ *E*
  shows *reachable-avoiding x y E*′
using *assms* **by** *induction auto*

lemma *ra-add-edge*:
  assumes *reachable-avoiding x y E*
  shows *reachable-avoiding x y* (*E* ∪ {(*v*,*w*)})
      ∨ (*reachable-avoiding x v* (*E* ∪ {(*v*,*w*)}) ∧ *reachable-avoiding w y* (*E* ∪ {(*v*,*w*)}))
using *assms* **proof** (*induction*)
  case (*ra-refl x E*)
  then show *?case* **by** *simp*
next
  case (*ra-succ x y E z*)
  then show *?case*
    using *reachable-avoiding.ra-succ* **by** *auto*
qed

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

lemma *ra-reachable*:
  reachable-avoiding x y E ⟹ *reachable x y*
  by (*induction rule*: *reachable-avoiding.induct*) (*auto intro*: *succ-reachable*)

lemma *ra-empty*:
  reachable-avoiding $x\ y$ $\{\}$ = *reachable* $x\ y$
proof
  assume *reachable-avoiding* $x\ y$ $\{\}$
  thus *reachable* $x\ y$
    by (*rule ra-reachable*)
next
  assume *reachable* $x\ y$
  hence *reachable-end* $x\ y$
    by (*rule reachable-re*)
  thus *reachable-avoiding* $x\ y$ $\{\}$
    by *induction auto*
qed

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 $S \equiv \forall x \in S.\ \forall y \in S.$ *reachable* $x\ y$

definition *is-scc* **where**
  is-scc $S \equiv S \neq \{\} \wedge$ *is-subscc* $S \wedge (\forall S'.\ S \subseteq S' \wedge$ *is-subscc* $S' \longrightarrow S' = S)$

lemma *subscc-add*:
  assumes *is-subscc* S **and** $x \in S$
    and *reachable* $x\ y$ **and** *reachable* $y\ x$
  shows *is-subscc* (*insert* y S)
using *assms* **unfolding** *is-subscc-def* **by** (*metis insert-iff reachable-trans*)

lemma *sccE*:
  — Two nodes that are reachable from each other are in the same SCC.
  assumes *is-scc* S **and** $x \in S$
    and *reachable* $x\ y$ **and** *reachable* $y\ x$
  shows $y \in S$
  using *assms* **unfolding** *is-scc-def*
  by (*metis insertI1 subscc-add subset-insertI*)

lemma *scc-partition*:
  — Two SCCs that contain a common element are identical.
  assumes *is-scc* S **and** *is-scc* S' **and** $x \in S \cap S'$
  shows $S = S'$
  using *assms* **unfolding** *is-scc-def is-subscc-def*
  by (*metis IntE assms*(*2*) *sccE subsetI*)

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'')
  by pat-completeness (force+)
```

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''
  using assms unfolding sub-env-def
  by (metis (no-types, lifting) subset-trans)
```

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

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*
using *assms* **unfolding** *wf-env-def*
by (*meson precedes-refl subset-iff*)

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*
using *assms* **unfolding** *wf-env-def* **by** *blast*

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*) $m \in \mathcal{S}\ e\ n$
using *assms* **unfolding** *wf-env-def*
by (*smt* (*verit*, *ccfv-threshold*) *Diff-iff Union-iff mem-Collect-eq*)

Every node belongs to its own equivalence class.

```
lemma S-reflexive:
  assumes wf-env e
  shows n ∈ 𝒮 e n
  using assms by (auto simp: wf-env-def)
```

No node on the stack has been fully explored.

```
lemma stack-unexplored:
  assumes 1: wf-env e
      and 2: n ∈ set (stack e)
      and 3: n ∈ explored e
  shows P
  using stack-class[OF 1 2] S-reflexive[OF 1] 3
  by blast
```

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 ∈ visited e
      and w: reachable v w
      and s: ∀ n ∈ visited e. ∀ m ∈ successors n − vsuccs e n.
                 reachable v n ⟶ ¬ reachable m w
  shows w ∈ visited e
using w v s proof (induction)
  case (reachable-refl x)
  then show ?case by simp
next
  case (reachable-succ x y z)
  then have y ∈ vsuccs e x by blast
  with e have y ∈ visited e
    unfolding wf-env-def by (meson le-infE subset-eq)
  with reachable-succ reachable.reachable-succ show ?case
    by blast
qed
```

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 ∉ explored e
      and w: w ∈ explored e
    shows reachable-avoiding x y (E ∪ {(v,w)})
using xy y proof (induction)
  case (ra-refl x E)
  then show ?case by simp
next
```

case (*ra-succ x y E z*)
from *e* ‹*z* ∈ *successors y*› ‹*z* ∉ *explored e*›
have *y* ∉ *explored e*
unfolding *wf-env-def* **by** (*meson reachable-edge*)
with *ra-succ.IH* **have** *reachable-avoiding x y* (*E* ∪ {(*v*,*w*)}) **.**
moreover
from *w* ‹(*y*,*z*) ∉ *E*› ‹*z* ∉ *explored e*› **have** (*y*,*z*) ∉ *E* ∪ {(*v*,*w*)}
by *auto*
ultimately show *?case*
using ‹*z* ∈ *successors y*› **by** *auto*
qed

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* ∉ *visited e*
∧ *reachable* (*root e*) *v*
∧ *vsuccs e v* = {}
∧ (∀ *n* ∈ *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**
post-dfs v e e' $\equiv$
wf-env e'
$\wedge$ *v* $\in$ *visited e'*
$\wedge$ *sub-env e e'*
$\wedge$ *vsuccs e' v* = *successors v*
$\wedge$ ($\forall$ *w* $\in$ *visited e. vsuccs e' w* = *vsuccs e w*)
$\wedge$ ($\forall$ *n* $\in$ *set* (*stack e'*). *reachable n v*)
$\wedge$ ($\exists$ *ns. stack e* = *ns* @ (*stack e'*))
$\wedge$ ((*v* $\in$ *explored e'* $\wedge$ *stack e'* = *stack e*
$\wedge$ ($\forall$ *n* $\in$ *set* (*stack e'*). $\mathcal{S}$ *e' n* = $\mathcal{S}$ *e n*))
$\vee$ (*stack e'* $\neq$ [] $\wedge$ *v* $\in$ $\mathcal{S}$ *e'* (*hd* (*stack e'*))
$\wedge$ ($\forall$ *n* $\in$ *set* (*tl* (*stack e'*)). $\mathcal{S}$ *e' n* = $\mathcal{S}$ *e n*)))
$\wedge$ *cstack e'* = *cstack e*

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

lemma *init-env-pre-dfs*: *pre-dfs v* (*init-env v*)
by (*auto simp*: *pre-dfs-def wf-env-def init-env-def is-subscc-def*
dest: *precedes-mem*)

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*: *stack e* $\neq$ [] *n* $\in$ $\mathcal{S}$ *e* (*hd* (*stack e*))
shows *stack e'* $\neq$ [] *n* $\in$ $\mathcal{S}$ *e'* (*hd* (*stack e'*))
proof –
have *1*: *stack e'* $\neq$ [] $\wedge$ *n* $\in$ $\mathcal{S}$ *e'* (*hd* (*stack e'*))
proof (*cases stack e'* = *stack e* $\wedge$ ($\forall$ *n* $\in$ *set* (*stack e'*). $\mathcal{S}$ *e' n* = $\mathcal{S}$ *e n*))
case *True*
with *n* **show** *?thesis*
by *auto*
next
case *2*: *False*
with *post* **have** *stack e'* $\neq$ []
by (*simp add*: *post-dfs-def*)
from *n* **have** *hd* (*stack e*) $\in$ *set* (*stack e*)
by *simp*
with *2 n post* **obtain** *u* **where**
u: *u* $\in$ *set* (*stack e'*) *n* $\in$ $\mathcal{S}$ *e' u*
unfolding *post-dfs-def sub-env-def* **by** *blast*
show *?thesis*
proof (*cases u* = *hd* (*stack e'*))
case *True*
with *u* ‹*stack e'* $\neq$ []› **show** *?thesis*
by *simp*
next
case *False*

with *u* **have** $u \in set\ (tl\ (stack\ e'))$
by (*metis empty-set equals0D list.collapse set-ConsD*)
with *u 2 post* **have** $u \in set\ (tl\ (stack\ e)) \wedge n \in \mathcal{S}\ e\ u$
unfolding *post-dfs-def*
by (*metis Un-iff append-self-conv2 set-append tl-append2*)
with *n wf* ‹$hd\ (stack\ e) \in set\ (stack\ e)$› **show** *?thesis*
unfolding *wf-env-def*
by (*metis* (*no-types*, *opaque-lifting*) *disjoint-iff-not-equal distinct.simps*(*2*) *list.collapse list.set-sel*(*2*))
qed
qed
from *1* **show** $stack\ e' \neq []$ **by** *simp*
from *1* **show** $n \in \mathcal{S}\ e'\ (hd\ (stack\ e'))$ **by** *simp*
qed

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*: $stack\ e \neq []$
shows $stack\ e' \neq []$ $\forall n \in set\ (tl\ (stack\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$
proof −
have *1*: $stack\ e' \neq [] \wedge (\forall n \in set\ (tl\ (stack\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n)$
proof (*cases* $stack\ e' = stack\ e \wedge (\forall n \in set\ (stack\ e').\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n)$)
case *True*
with *nempty* **show** *?thesis*
by (*simp add*: *list.set-sel*(*2*))
next
case *False*
with *post* **show** *?thesis*
by (*auto simp*: *post-dfs-def*)
qed
from *1* **show** $stack\ e' \neq []$
by *simp*
from *1* **show** $\forall n \in set\ (tl\ (stack\ e')).\ \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$
by *simp*
qed

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 $e' \equiv \mathit{unite}\ v\ w\ e$
assumes *wf*: $\mathit{wf\text{-}env}\ e$
and *w*: $w \in \mathit{successors}\ v\ w \notin \mathit{vsuccs}\ e\ v\ w \in \mathit{visited}\ e\ w \notin \mathit{explored}\ e$
obtains *pfx* **where** $\mathit{stack}\ e = \mathit{pfx}\ @\ (\mathit{stack}\ e')$

$$
\begin{aligned}
&\mathit{stack}\ e' \neq [] \\
&\mathit{let}\ \mathit{cc} = \bigcup\ \{\mathcal{S}\ e\ n\ |n.\ n \in \mathit{set}\ \mathit{pfx} \cup \{\mathit{hd}\ (\mathit{stack}\ e')\}\} \\
&\ \mathit{in}\ \mathcal{S}\ e' = (\lambda x.\ \mathit{if}\ x \in \mathit{cc}\ \mathit{then}\ \mathit{cc}\ \mathit{else}\ \mathcal{S}\ e\ x)
\end{aligned}
$$

```
                  w ∈ 𝒮 e′ (hd (stack e′))
proof −
  define pfx where pfx = takeWhile (λx. w ∉ 𝒮 e x) (stack e)
  define sfx where sfx = dropWhile (λx. w ∉ 𝒮 e x) (stack e)
  define cc where cc = ⋃ {𝒮 e x |x. x ∈ set pfx ∪ {hd sfx}}

  have stack e = pfx @ sfx
    by (simp add: pfx-def sfx-def)
  moreover
  have stack e′ = sfx
    by (simp add: e′-def unite-def sfx-def)
  moreover
  from wf w have w ∈ ⋃ {𝒮 e n | n. n ∈ set (stack e)}
    by (simp add: wf-env-def)
  then obtain n where n ∈ set (stack e) w ∈ 𝒮 e n
    by auto
  hence sfx: sfx ≠ [] ∧ w ∈ 𝒮 e (hd sfx)
    unfolding sfx-def
    by (metis dropWhile-eq-Nil-conv hd-dropWhile)
  moreover
  have 𝒮 e′ = (λx. if x ∈ cc then cc else 𝒮 e x)
    by (rule,
        auto simp add: e′-def unite-def pfx-def sfx-def cc-def)
  moreover
  from sfx have w ∈ cc
    by (auto simp: cc-def)
  from S-reflexive[OF wf, of hd sfx]
  have hd sfx ∈ cc
    by (auto simp: cc-def)
  with ‹w ∈ cc› ‹𝒮 e′ = (λx. if x ∈ cc then cc else 𝒮 e x)›
  have w ∈ 𝒮 e′ (hd sfx)
    by simp
  ultimately show ?thesis
    using that e′-def unite-def pfx-def sfx-def cc-def
    by meson
qed
```

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′ ≡ unite v w e
  assumes wf: wf-env e
      and w: w ∈ successors v w ∉ vsuccs e v w ∈ visited e w ∉ explored e
      and n: n ∈ set (tl (stack e′))
  shows 𝒮 e′ n = 𝒮 e n
proof −
  from assms obtain pfx where
    pfx: stack e = pfx @ (stack e′) stack e′ ≠ []
```

let cc = $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e'*)}}
in $\mathcal{S}$ *e'* = (λ*x*. *if x* ∈ *cc then cc else* $\mathcal{S}$ *e x*)
by (*blast dest*: *unite-stack*)
define *cc* **where** *cc* ≡ $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e'*)}}

have *n* ∉ *cc*
proof
assume *n* ∈ *cc*
then obtain *m* **where**
m ∈ *set pfx* ∪ {*hd* (*stack e'*)} *n* ∈ $\mathcal{S}$ *e m*
by (*auto simp*: *cc-def*)
with *S-reflexive*[*OF wf*, *of n*] *n wf* ‹*stack e* = *pfx* @ *stack e'*› ‹*stack e'* ≠ []›
show *False*
unfolding *wf-env-def*
by (*smt* (*z3*) *Diff-triv Un-iff Un-insert-right append.right-neutral disjoint-insert*(*1*)

distinct.simps(*2*) *distinct-append empty-set insertE insert-Diff*
list.exhaust-sel
list.simps(*15*) *set-append*)
qed
with *pfx* **show** $\mathcal{S}$ *e' n* = $\mathcal{S}$ *e n*
by (*auto simp add*: *cc-def*)
qed

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'* ≡ *unite v w e*
assumes *wf*: *wf-env e*
and *w*: *w* ∈ *successors v w* ∉ *vsuccs e v w* ∈ *visited e w* ∉ *explored e*
shows ($\bigcup$ {$\mathcal{S}$ *e' n* | *n*. *n* ∈ *set* (*stack e'*)}) = ($\bigcup$ {$\mathcal{S}$ *e n* | *n*. *n* ∈ *set* (*stack e*)})
proof −
from *assms* **obtain** *pfx* **where**
pfx: *stack e* = *pfx* @ (*stack e'*) *stack e'* ≠ []
let cc = $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e'*)}}
in $\mathcal{S}$ *e'* = (λ*x*. *if x* ∈ *cc then cc else* $\mathcal{S}$ *e x*)
by (*blast dest*: *unite-stack*)
define *cc* **where** *cc* ≡ $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e'*)}}

from *pfx* **have** *Se'*: ∀ *x*. $\mathcal{S}$ *e' x* = (*if x* ∈ *cc then cc else* $\mathcal{S}$ *e x*)
by (*auto simp*: *cc-def*)
from *S-reflexive*[*OF wf*, *of hd* (*stack e'*)]
have *S-hd*: $\mathcal{S}$ *e'* (*hd* (*stack e'*)) = *cc*
by (*auto simp*: *Se' cc-def*)
from ‹*stack e'* ≠ []›
have *ste'*: *set* (*stack e'*) = {*hd* (*stack e'*)} ∪ *set* (*tl* (*stack e'*))
by (*metis insert-is-Un list.exhaust-sel list.simps*(*15*))

from ‹*stack e* = *pfx* @ *stack e*′› ‹*stack e*′ ≠ []›
have *stack e* = *pfx* @ (*hd* (*stack e*′) # *tl* (*stack e*′))
by *auto*
hence $\bigcup$ {$\mathcal{S}$ *e n* | *n*. *n* ∈ *set* (*stack e*)}
= *cc* ∪ ($\bigcup$ {$\mathcal{S}$ *e n* | *n*. *n* ∈ *set* (*tl* (*stack e*′))})
by (*auto simp add*: *cc-def*)
also from *S-hd unite-S-tl*[*OF wf w*]
have ... = $\mathcal{S}$ *e*′ (*hd* (*stack e*′)) ∪ ($\bigcup$ {$\mathcal{S}$ *e*′ *n* | *n*. *n* ∈ *set* (*tl* (*stack e*′))})
by (*auto simp*: *e*′*-def*)
also from *ste*′
have ... = $\bigcup$ {$\mathcal{S}$ *e*′ *n* | *n*. *n* ∈ *set* (*stack e*′)}
by *auto*
finally show *?thesis*
by *simp*
qed

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

lemma *unite-subscc*:
fixes *e v w*
defines *e*′ ≡ *unite v w e*
assumes *pre*: *pre-dfss v e*
and *w*: *w* ∈ *successors v w* ∉ *vsuccs e v w* ∈ *visited e w* ∉ *explored e*
shows *is-subscc* ($\mathcal{S}$ *e*′ (*hd* (*stack e*′)))
proof −
from *pre* **have** *wf*: *wf-env e*
by (*simp add*: *pre-dfss-def*)
from *assms* **obtain** *pfx* **where**
pfx: *stack e* = *pfx* @ (*stack e*′) *stack e*′ ≠ []
let cc = $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e*′)}}
in $\mathcal{S}$ *e*′ = (λ*x*. *if x* ∈ *cc then cc else* $\mathcal{S}$ *e x*)
by (*blast dest*: *unite-stack*[*OF wf*])

define *cc* **where** *cc* ≡ $\bigcup$ {$\mathcal{S}$ *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e*′)}}

from *wf w* **have** *w* ∈ $\bigcup$ {$\mathcal{S}$ *e n* | *n*. *n* ∈ *set* (*stack e*)}
by (*simp add*: *wf-env-def*)
hence *w* ∈ $\mathcal{S}$ *e* (*hd* (*stack e*′))
apply (*simp add*: *e*′*-def unite-def*)
by (*metis dropWhile-eq-Nil-conv hd-dropWhile*)

have *is-subscc cc*
proof (*clarsimp simp*: *is-subscc-def*)
fix *x y*
assume *x* ∈ *cc y* ∈ *cc*
then obtain *nx ny* **where**
nx: *nx* ∈ *set pfx* ∪ {*hd* (*stack e*′)} *x* ∈ $\mathcal{S}$ *e nx* **and**
ny: *ny* ∈ *set pfx* ∪ {*hd* (*stack e*′)} *y* ∈ $\mathcal{S}$ *e ny*
by (*auto simp*: *cc-def*)
with *wf* **have** *reachable x nx reachable ny y*

by (*auto simp*: *wf-env-def is-subscc-def*)
from *w pre* **have** *reachable v w*
by (*auto simp*: *pre-dfss-def*)
from *pre* **have** *reachable* (*hd* (*stack e*)) *v*
by (*auto simp*: *pre-dfss-def wf-env-def is-subscc-def*)
from *pre* **have** *stack e* = *hd* (*stack e*) # *tl* (*stack e*)
by (*auto simp*: *pre-dfss-def*)
with *nx* ‹*stack e* = *pfx* @ (*stack e′*)› ‹*stack e′* ≠ []›
have *hd* (*stack e*) ⪯ *nx in stack e*
by (*metis Un-iff Un-insert-right head-precedes list.exhaust-sel list.simps*(*15*)
set-append sup-bot.right-neutral)
with *wf* **have** *reachable nx* (*hd* (*stack e*))
by (*auto simp*: *wf-env-def*)
from ‹*stack e* = *pfx* @ (*stack e′*)› ‹*stack e′* ≠ []› *ny*
have *ny* ⪯ *hd* (*stack e′*) *in stack e*
by (*metis List.set-insert empty-set insert-Nil list.exhaust-sel set-append split-list-precedes*)
with *wf* **have** *reachable* (*hd* (*stack e′*)) *ny*
by (*auto simp*: *wf-env-def is-subscc-def*)
from *wf* ‹*stack e′* ≠ []› ‹*w* ∈ 𝒮 *e* (*hd* (*stack e′*))›
have *reachable w* (*hd* (*stack e′*))
by (*auto simp*: *wf-env-def is-subscc-def*)

from ‹*reachable x nx*› ‹*reachable nx* (*hd* (*stack e*))›
‹*reachable* (*hd* (*stack e*)) *v*› ‹*reachable v w*›
‹*reachable w* (*hd* (*stack e′*))›
‹*reachable* (*hd* (*stack e′*)) *ny*› ‹*reachable ny y*›
show *reachable x y*
using *reachable-trans* **by** *meson*
qed
with *S-reflexive*[*OF wf*, *of hd* (*stack e′*)] *pfx*
show *?thesis*
by (*auto simp*: *cc-def*)
qed

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

lemma *unite-sub-env*:
fixes *e v w*
defines *e′* ≡ *unite v w e*
assumes *pre*: *pre-dfss v e*
and *w*: *w* ∈ *successors v w* ∉ *vsuccs e v w* ∈ *visited e w* ∉ *explored e*
shows *sub-env e e′*
proof −
from *pre* **have** *wf*: *wf-env e*
by (*simp add*: *pre-dfss-def*)
from *assms* **obtain** *pfx* **where**
pfx: *stack e* = *pfx* @ (*stack e′*) *stack e′* ≠ []
let cc = ⋃ {𝒮 *e n* |*n*. *n* ∈ *set pfx* ∪ {*hd* (*stack e′*)}}
in 𝒮 *e′* = (λ*x*. *if x* ∈ *cc then cc else* 𝒮 *e x*)
by (*blast dest*: *unite-stack*[*OF wf*])

define *cc* **where** $cc \equiv \bigcup \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$
have $\forall n.\ \mathcal{S}\ e\ n \subseteq \mathcal{S}\ e'\ n$
proof (*clarify*)
fix *n u*
assume *u*: $u \in \mathcal{S}\ e\ n$
show $u \in \mathcal{S}\ e'\ n$
proof (*cases* $n \in cc$)
case *True*
then obtain *m* **where**
m: $m \in set\ pfx \cup \{hd\ (stack\ e')\}\ n \in \mathcal{S}\ e\ m$
by (*auto simp*: *cc-def*)
with *wf S-reflexive*[*OF wf*, *of n*] *u* **have** $u \in \mathcal{S}\ e\ m$
by (*auto simp*: *wf-env-def*)
with *m pfx* **show** *?thesis*
by (*auto simp*: *cc-def*)
next
case *False*
with *pfx u* **show** *?thesis*
by (*auto simp*: *cc-def*)
qed
qed
moreover
have $root\ e' = root\ e \wedge visited\ e' = visited\ e$
$\wedge\ explored\ e' = explored\ e \wedge vsuccs\ e' = vsuccs\ e$
by (*simp add*: *e′-def unite-def*)
ultimately show *?thesis*
using *unite-S-equal*[*OF wf w*]
by (*simp add*: *e′-def sub-env-def*)
qed

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 −
from *pre* **have** *wf*: *wf-env e*
by (*simp add*: *pre-dfss-def*)
from *assms* **obtain** *pfx* **where**
pfx: $stack\ e = pfx\ @\ (stack\ e')\ stack\ e' \neq []$
$let\ cc = \bigcup \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$
$in\ \mathcal{S}\ e' = (\lambda x.\ if\ x \in cc\ then\ cc\ else\ \mathcal{S}\ e\ x)$
by (*blast dest*: *unite-stack*[*OF wf*])
define *cc* **where** $cc \equiv \bigcup \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$

from *pfx* **have** *Se′*: $\forall x.\ \mathcal{S}\ e'\ x = (if\ x \in cc\ then\ cc\ else\ \mathcal{S}\ e\ x)$
by (*auto simp add*: *cc-def*)

have *cc-Un*: $cc = \bigcup \{\mathcal{S}\ e\ x \mid x.\ x \in cc\}$
proof
  from *S-reflexive*[*OF wf*]
  show $cc \subseteq \bigcup \{\mathcal{S}\ e\ x \mid x.\ x \in cc\}$
    by (*auto simp*: *cc-def*)
next
  {
    fix *n x*
    assume $x \in cc$ $n \in \mathcal{S}\ e\ x$
    with *wf* **have** $n \in cc$
      unfolding *wf-env-def cc-def*
      by (*smt* (*verit*) *Union-iff mem-Collect-eq*)
  }
  thus $(\bigcup \{\mathcal{S}\ e\ x \mid x.\ x \in cc\}) \subseteq cc$
    by *blast*
qed

from *S-reflexive*[*OF wf*, *of hd* (*stack e'*)]
have *hd-cc*: $\mathcal{S}\ e'\ (hd\ (stack\ e')) = cc$
  by (*auto simp*: *cc-def Se'*)

{
  fix *n m*
  assume *n*: $n \in set\ (tl\ (stack\ e'))$
    and *m*: $m \in \mathcal{S}\ e\ n \cap cc$
  from *m* **obtain** *l* **where**
    $l \in set\ pfx \cup \{hd\ (stack\ e')\}$ $m \in \mathcal{S}\ e\ l$
    by (*auto simp*: *cc-def*)
  with *n m wf* ‹*stack e = pfx @ stack e'*› ‹$stack\ e' \neq []$›
  have *False*
    unfolding *wf-env-def*
  by (*metis* (*no-types*, *lifting*) *Int-iff UnCI UnE disjoint-insert*(*1*) *distinct.simps*(*2*)

        distinct-append emptyE hd-Cons-tl insert-iff list.set-sel(*1*) *list.set-sel*(*2*)

          mk-disjoint-insert set-append)
}
hence *tl-cc*: $\forall n \in set\ (tl\ (stack\ e')).\ \mathcal{S}\ e\ n \cap cc = \{\}$
  by *blast*

from *wf*
have $\forall n \in visited\ e'.\ reachable\ (root\ e')\ n$
    $distinct\ (cstack\ e')$
    $explored\ e' \subseteq visited\ e'$
    $set\ (cstack\ e') \subseteq visited\ e'$
    $\forall n \in explored\ e'.\ \forall m.\ reachable\ n\ m \longrightarrow m \in explored\ e'$
    $\forall n.\ vsuccs\ e'\ n \subseteq successors\ n \cap visited\ e'$
    $\forall n.\ n \notin visited\ e' \longrightarrow vsuccs\ e'\ n = \{\}$

$\forall\, n \in explored\ e'.\ vsuccs\ e'\ n = successors\ n$
$\forall\, n \in visited\ e' - set\ (cstack\ e').\ vsuccs\ e'\ n = successors\ n$
$\forall\, S \in sccs\ e'.\ is\text{-}scc\ S$
$\bigcup\ (sccs\ e') = explored\ e'$
by (*auto simp*: *wf-env-def e′-def unite-def*)

moreover
from *wf* ‹*stack e* = *pfx* @ *stack e′*›
have *distinct* (*stack e′*)
by (*auto simp*: *wf-env-def*)

moreover
have $\forall\, n\ m.\ n \preceq m\ in\ stack\ e' \longrightarrow n \preceq m\ in\ cstack\ e'$
proof (*clarify*)
fix *n m*
assume $n \preceq m\ in\ stack\ e'$
with ‹*stack e* = *pfx* @ *stack e′*› *wf*
have $n \preceq m\ in\ cstack\ e$
unfolding *wf-env-def*
by (*metis precedes-append-left*)
thus $n \preceq m\ in\ cstack\ e'$
by (*simp add*: *e′-def unite-def*)
qed

moreover
from *wf* ‹*stack e* = *pfx* @ *stack e′*›
have $\forall\, n\ m.\ n \preceq m\ in\ stack\ e' \longrightarrow reachable\ m\ n$
unfolding *wf-env-def* **by** (*metis precedes-append-left*)

moreover
have $\forall\, n\ m.\ m \in \mathcal{S}\ e'\ n \longleftrightarrow (\mathcal{S}\ e'\ n = \mathcal{S}\ e'\ m)$
proof (*clarify*)
fix *n m*
show $m \in \mathcal{S}\ e'\ n \longleftrightarrow (\mathcal{S}\ e'\ n = \mathcal{S}\ e'\ m)$
proof
assume *l*: $m \in \mathcal{S}\ e'\ n$
show $\mathcal{S}\ e'\ n = \mathcal{S}\ e'\ m$
proof (*cases* $n \in cc$)
case *True*
with *l* **show** *?thesis*
by (*simp add*: *Se′*)
next
case *False*
with *l wf* **have** $\mathcal{S}\ e\ n = \mathcal{S}\ e\ m$
by (*simp add*: *wf-env-def Se′*)
with *False cc-Un wf* **have** $m \notin cc$
unfolding *wf-env-def e′-def*
by (*smt* (*verit*, *best*) *Union-iff mem-Collect-eq*)
with ‹$\mathcal{S}\ e\ n = \mathcal{S}\ e\ m$› *False* **show** *?thesis*

```
        by (simp add: Se')
      qed
    next
      assume r: 𝒮 e' n = 𝒮 e' m
      show m ∈ 𝒮 e' n
      proof (cases n ∈ cc)
        case True
        with r pfx have 𝒮 e' m = cc
          by (auto simp: cc-def)
        have m ∈ cc
        proof (rule ccontr)
          assume m ∉ cc
          with pfx have 𝒮 e' m = 𝒮 e m
            by (auto simp: cc-def)
          with S-reflexive[OF wf, of m] ‹𝒮 e' m = cc› ‹m ∉ cc›
          show False
            by simp
        qed
        with pfx True show m ∈ 𝒮 e' n
          by (auto simp: cc-def)
      next
        case False
        hence 𝒮 e' n = 𝒮 e n
          by (simp add: Se')
        have m ∉ cc
        proof
          assume m: m ∈ cc
          with ‹𝒮 e' n = 𝒮 e n› r have 𝒮 e n = cc
            by (simp add: Se')
          with S-reflexive[OF wf, of n] have n ∈ cc
            by simp
          with ‹n ∉ cc› show False ..
        qed
        with r ‹𝒮 e' n = 𝒮 e n› have 𝒮 e m = 𝒮 e n
          by (simp add: Se')
        with S-reflexive[OF wf, of m] have m ∈ 𝒮 e n
          by simp
        with ‹𝒮 e' n = 𝒮 e n› show ?thesis
          by simp
      qed
    qed
  qed

  moreover
  have ∀ n. n ∉ visited e' ⟶ 𝒮 e' n = {n}
  proof (clarify)
    fix n
    assume n ∉ visited e'
    hence n ∉ visited e
```

by (*simp add*: *e′-def unite-def*)
moreover have $n \notin cc$
proof
assume $n \in cc$
then obtain *m* **where** $m \in set\ pfx \cup \{hd\ (stack\ e')\}$ $n \in \mathcal{S}\ e\ m$
by (*auto simp*: *cc-def*)
with ‹$stack\ e = pfx\ @\ stack\ e'$› ‹$stack\ e' \neq []$›
have $m \in set\ (stack\ e)$
by *auto*
with *stack-class*[*OF wf this* ‹$n \in \mathcal{S}\ e\ m$›] ‹$n \notin visited\ e$›
show *False*
by *simp*
qed
ultimately show $\mathcal{S}\ e'\ n = \{n\}$
using *wf* **by** (*auto simp*: *wf-env-def Se′*)
qed

moreover
have $\forall n \in set\ (stack\ e').\ \forall m \in set\ (stack\ e').\ n \neq m \longrightarrow \mathcal{S}\ e'\ n \cap \mathcal{S}\ e'\ m = \{\}$
proof (*clarify*)
fix *n m*
assume $n \in set\ (stack\ e')$ $m \in set\ (stack\ e')$ $n \neq m$
show $\mathcal{S}\ e'\ n \cap \mathcal{S}\ e'\ m = \{\}$
proof (*cases* $n = hd\ (stack\ e')$)
case *True*
with ‹$m \in set\ (stack\ e')$› ‹$n \neq m$› ‹$stack\ e' \neq []$›
have $m \in set\ (tl\ (stack\ e'))$
by (*metis hd-Cons-tl set-ConsD*)
with *True hd-cc tl-cc unite-S-tl*[*OF wf w*]
show *?thesis*
by (*auto simp*: *e′-def*)
next
case *False*
with ‹$n \in set\ (stack\ e')$› ‹$stack\ e' \neq []$›
have $n \in set\ (tl\ (stack\ e'))$
by (*metis hd-Cons-tl set-ConsD*)
show *?thesis*
proof (*cases* $m = hd\ (stack\ e')$)
case *True*
with ‹$n \in set\ (tl\ (stack\ e'))$› *hd-cc tl-cc unite-S-tl*[*OF wf w*]
show *?thesis*
by (*auto simp*: *e′-def*)
next
case *False*
with ‹$m \in set\ (stack\ e')$› ‹$stack\ e' \neq []$›
have $m \in set\ (tl\ (stack\ e'))$
by (*metis hd-Cons-tl set-ConsD*)
with ‹$n \in set\ (tl\ (stack\ e'))$›
have $\mathcal{S}\ e'\ m = \mathcal{S}\ e\ m \wedge \mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$

        by (*auto simp*: *e′-def unite-S-tl*[*OF wf w*])
      moreover
      from ‹$m \in set\ (stack\ e')$› ‹$n \in set\ (stack\ e')$› ‹$stack\ e = pfx\ @\ stack\ e'$›
      have $m \in set\ (stack\ e) \wedge n \in set\ (stack\ e)$
        by *auto*
      ultimately show *?thesis*
        using *wf* ‹$n \neq m$› **by** (*auto simp*: *wf-env-def*)
    qed
  qed
qed

moreover
{
  from *unite-S-equal*[*OF wf w*]
  have $\bigcup \{\mathcal{S}\ e'\ n \mid n.\ n \in set\ (stack\ e')\} = \bigcup \{\mathcal{S}\ e\ n \mid n.\ n \in set\ (stack\ e)\}$
    by (*simp add*: *e′-def*)
  with *wf*
  have $\bigcup \{\mathcal{S}\ e'\ n \mid n.\ n \in set\ (stack\ e')\} = visited\ e - explored\ e$
    by (*simp add*: *wf-env-def*)
}
hence $\bigcup \{\mathcal{S}\ e'\ n \mid n.\ n \in set\ (stack\ e')\} = visited\ e' - explored\ e'$
  by (*simp add*: *e′-def unite-def*)

moreover
have $\forall n \in set\ (stack\ e').\ \forall m \in \mathcal{S}\ e'\ n.$
      $m \in set\ (cstack\ e') \longrightarrow m \preceq n\ in\ cstack\ e'$
proof (*clarify*)
  fix *n m*
  assume $n \in set\ (stack\ e')$ $m \in \mathcal{S}\ e'\ n$ $m \in set\ (cstack\ e')$
  from ‹$m \in set\ (cstack\ e')$› **have** $m \in set\ (cstack\ e)$
    by (*simp add*: *e′-def unite-def*)
  have $m \preceq n\ in\ cstack\ e$
  proof (*cases* $n = hd\ (stack\ e')$)
    case *True*
    with ‹$m \in \mathcal{S}\ e'\ n$› **have** $m \in cc$
      by (*simp add*: *hd-cc*)
    then obtain *l* **where**
      $l \in set\ pfx \cup \{hd\ (stack\ e')\}$ $m \in \mathcal{S}\ e\ l$
      by (*auto simp*: *cc-def*)
    with ‹$stack\ e = pfx\ @\ stack\ e'$› ‹$stack\ e' \neq []$›
    have $l \in set\ (stack\ e)$
      by *auto*
    with ‹$m \in \mathcal{S}\ e\ l$› ‹$m \in set\ (cstack\ e)$› *wf*
    have $m \preceq l\ in\ cstack\ e$
      by (*auto simp*: *wf-env-def*)
    moreover
    from ‹$l \in set\ pfx \cup \{hd\ (stack\ e')\}$› *True*
         ‹$stack\ e = pfx\ @\ stack\ e'$› ‹$stack\ e' \neq []$›
    have $l \preceq n\ in\ stack\ e$

```
      by (metis List.set-insert empty-set hd-Cons-tl insert-Nil set-append split-list-precedes)
      with wf have l ≼ n in cstack e
        by (auto simp: wf-env-def)
      ultimately show ?thesis
        using wf unfolding wf-env-def
        by (meson precedes-trans)
    next
      case False
      with ‹n ∈ set (stack e')› ‹stack e' ≠ []›
      have n ∈ set (tl (stack e'))
        by (metis list.collapse set-ConsD)
      with unite-S-tl[OF wf w] ‹m ∈ 𝒮 e' n›
      have m ∈ 𝒮 e n
        by (simp add: e'-def)
      with ‹n ∈ set (stack e')› ‹stack e = pfx @ stack e'›
           ‹m ∈ set (cstack e)› wf
      show ?thesis
        by (auto simp: wf-env-def)
    qed
    thus m ≼ n in cstack e'
      by (simp add: e'-def unite-def)
  qed

  moreover
  have ∀ n m. n ≼ m in stack e' ∧ n ≠ m ⟶
        (∀ u ∈ 𝒮 e' n. ¬ reachable-avoiding u m (unvisited e' n))
  proof (clarify)
    fix x y u
    assume xy: x ≼ y in stack e' x ≠ y
       and u: u ∈ 𝒮 e' x reachable-avoiding u y (unvisited e' x)
    show False
    proof (cases x = hd (stack e'))
      case True
      hence 𝒮 e' x = cc
        by (simp add: hd-cc)
      with ‹u ∈ 𝒮 e' x› obtain x' where
        x': x' ∈ set pfx ∪ {hd (stack e')} u ∈ 𝒮 e x'
        by (auto simp: cc-def)
      from ‹stack e = pfx @ stack e'› ‹stack e' ≠ []›
      have stack e = pfx @ (hd (stack e') # tl (stack e'))
        by auto
      with x' True have x' ≼ x in stack e
        by (simp add: split-list-precedes)
      moreover
      from xy ‹stack e = pfx @ stack e'› have x ≼ y in stack e
        by (simp add: precedes-append-left)
      ultimately have x' ≼ y in stack e
        using wf by (auto simp: wf-env-def elim: precedes-trans)
      from ‹x' ≼ x in stack e› ‹x ≼ y in stack e› wf ‹x ≠ y›
```

have $x' \neq y$
  by (*auto simp*: *wf-env-def dest*: *precedes-antisym*)
let *?unv* = $\bigcup$ {*unvisited e y* | *y*. *y* ∈ *set pfx* ∪ {*hd* (*stack e'*)}}
from ‹$\mathcal{S}$ *e' x* = *cc*› **have** *?unv* = *unvisited e' x*
  by (*auto simp*: *unvisited-def cc-def e'-def unite-def*)
with ‹*reachable-avoiding u y* (*unvisited e' x*)›
have *reachable-avoiding u y ?unv*
  by *simp*
with *x'* **have** *reachable-avoiding u y* (*unvisited e x'*)
  by (*blast intro*: *ra-mono*)
with ‹$x' \preceq y$ *in stack e*› ‹$x' \neq y$› ‹*u* ∈ $\mathcal{S}$ *e x'*› *wf*
show *?thesis*
  by (*auto simp*: *wf-env-def*)
next
case *False*
with ‹$x \preceq y$ *in stack e'*› ‹*stack e'* ≠ []›
have *x* ∈ *set* (*tl* (*stack e'*))
  by (*metis list.exhaust-sel precedes-mem*(*1*) *set-ConsD*)
with ‹*u* ∈ $\mathcal{S}$ *e' x*› **have** *u* ∈ $\mathcal{S}$ *e x*
  by (*auto simp add*: *unite-S-tl*[*OF wf w*] *e'-def*)
moreover
from ‹$x \preceq y$ *in stack e'*› ‹*stack e* = *pfx* @ *stack e'*›
have $x \preceq y$ *in stack e*
  by (*simp add*: *precedes-append-left*)
moreover
from *unite-S-tl*[*OF wf w*] ‹*x* ∈ *set* (*tl* (*stack e'*))›
have *unvisited e' x* = *unvisited e x*
  by (*auto simp*: *unvisited-def e'-def unite-def*)
ultimately show *?thesis*
  using ‹$x \neq y$› ‹*reachable-avoiding u y* (*unvisited e' x*)› *wf*
  by (*auto simp*: *wf-env-def*)
qed
qed

moreover
have ∀ *n*. *is-subscc* ($\mathcal{S}$ *e' n*)
proof
fix *n*
show *is-subscc* ($\mathcal{S}$ *e' n*)
proof (*cases n* ∈ *cc*)
case *True*
hence $\mathcal{S}$ *e' n* = *cc*
  by (*simp add*: *Se'*)
with *unite-subscc*[*OF pre w*] *hd-cc*
show *?thesis*
  by (*auto simp*: *e'-def*)
next
case *False*

```
      with wf show ?thesis
        by (simp add: Se' wf-env-def)
    qed
  qed

  ultimately show ?thesis
    unfolding wf-env-def by blast
qed
```

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 −
  from assms have wf: wf-env e
    by (simp add: pre-dfs-def)

  from assms have v: v ∉ visited e
    by (simp add: pre-dfs-def)

  from assms stack-visited[OF wf]
  have ∀ n ∈ visited ?e'. reachable (root ?e') n
       distinct (stack ?e')
       distinct (cstack ?e')
       explored ?e' ⊆ visited ?e'
       set (cstack ?e') ⊆ visited ?e'
       ∀ n ∈ explored ?e'. ∀ m. reachable n m ⟶ m ∈ explored ?e'
       ∀ n. vsuccs ?e' n ⊆ successors n
       ∀ n ∈ explored ?e'. vsuccs ?e' n = successors n
       ∀ n ∈ visited ?e' − set(cstack ?e'). vsuccs ?e' n = successors n
       ∀ n. n ∉ visited ?e' ⟶ vsuccs ?e' n = {}
       (∀ n m. m ∈ 𝒮 ?e' n ⟷ (𝒮 ?e' n = 𝒮 ?e' m))
       (∀ n. n ∉ visited ?e' ⟶ 𝒮 ?e' n = {n})
       ∀ n. is-subscc (𝒮 ?e' n)
       ∀ S ∈ sccs ?e'. is-scc S
       ⋃ (sccs ?e') = explored ?e'
    by (auto simp: pre-dfs-def wf-env-def)

  moreover
  have ∀ n m. n ≼ m in stack ?e' ⟶ reachable m n
  proof (clarify)
    fix x y
    assume x ≼ y in stack ?e'
```

```
    show reachable y x
    proof (cases x=v)
      assume x=v
      with ‹x ≼ y in stack ?e'› assms show ?thesis
        apply (simp add: pre-dfs-def)
        by (metis insert-iff list.simps(15) precedes-mem(2) reachable-refl)
    next
      assume x ≠ v
      with ‹x ≼ y in stack ?e'› wf show ?thesis
        by (simp add: pre-dfs-def wf-env-def precedes-in-tail)
    qed
  qed

  moreover
  from wf v have ∀ n. vsuccs ?e' n ⊆ visited ?e'
    by (auto simp: wf-env-def)

  moreover
  from wf v
  have (∀ n ∈ set (stack ?e'). ∀ m ∈ set (stack ?e'). n ≠ m ⟶ S ?e' n ∩ S ?e'
m = {})
    apply (simp add: wf-env-def)
    by (metis singletonD)

  moreover
  have ⋃ {S ?e' v | v . v ∈ set (stack ?e')} = visited ?e' − explored ?e'
  proof −
    have ⋃ {S ?e' v | v . v ∈ set (stack ?e')} =
          (⋃ {S e v | v . v ∈ set (stack e)}) ∪ S e v
      by auto
    also from wf v have ... = visited ?e' − explored ?e'
      by (auto simp: wf-env-def)
    finally show ?thesis .
  qed

  moreover
  have ∀ n m. n ≼ m in stack ?e' ∧ n ≠ m ⟶
          (∀ u ∈ S ?e' n. ¬ reachable-avoiding u m (unvisited ?e' n))
  proof (clarify)
    fix x y u
    assume asm: x ≼ y in stack ?e' x ≠ y u ∈ S ?e' x
             reachable-avoiding u y (unvisited ?e' x)
    show False
    proof (cases x = v)
      case True
      with wf v ‹u ∈ S ?e' x› have u = v vsuccs ?e' v = {}
        by (auto simp: wf-env-def)
      with ‹reachable-avoiding u y (unvisited ?e' x)›[THEN ra-cases]
           True ‹x ≠ y› wf
```

```
      show ?thesis
        by (auto simp: wf-env-def unvisited-def)
    next
      case False
      with asm wf show ?thesis
        by (auto simp: precedes-in-tail wf-env-def unvisited-def)
    qed
  qed

  moreover
  have ∀ n m. n ⪯ m in stack ?e′ ⟶ n ⪯ m in cstack ?e′
  proof (clarsimp)
    fix n m
    assume n ⪯ m in (v # stack e)
    with assms show n ⪯ m in (v # cstack e)
      unfolding pre-dfs-def wf-env-def
     by (metis head-precedes insertI1 list.simps(15) precedes-in-tail precedes-mem(2)
precedes-refl)
  qed

  moreover
  have ∀ n ∈ set (stack ?e′). ∀ m ∈ 𝒮 ?e′ n. m ∈ set (cstack ?e′) ⟶ m ⪯ n in
cstack ?e′
  proof (clarify)
    fix n m
    assume n ∈ set (stack ?e′) m ∈ 𝒮 ?e′ n m ∈ set (cstack ?e′)
    show m ⪯ n in cstack ?e′
    proof (cases n = v)
      case True
      with wf v ‹m ∈ 𝒮 ?e′ n› show ?thesis
        by (auto simp: wf-env-def)
    next
      case False
      with ‹n ∈ set (stack ?e′)› ‹m ∈ 𝒮 ?e′ n›
      have n ∈ set (stack e) m ∈ 𝒮 e n
        by auto
      with wf v False ‹m ∈ 𝒮 e n› ‹m ∈ set (cstack ?e′)›
      show ?thesis
        apply (simp add: wf-env-def)
        by (metis (mono-tags, lifting) precedes-in-tail singletonD)
    qed
  qed

  ultimately have wf-env ?e′
    unfolding wf-env-def by (meson le-inf-iff)

  moreover
  from assms
  have ∀ w ∈ vsuccs ?e′ v. w ∈ explored ?e′ ∪ 𝒮 ?e′ (hd (stack ?e′))
```

by (*simp add*: *pre-dfs-def*)

moreover
from ‹$\forall n\ m.\ n \preceq m$ *in stack ?e′* $\longrightarrow$ *reachable m n*›
have $\forall n \in$ *set* (*stack ?e′*). *reachable n v*
by (*simp add*: *head-precedes*)

moreover
from *wf v* **have** $\mathcal{S}$ *?e′* (*hd* (*stack ?e′*)) = {*v*}
by (*simp add*: *pre-dfs-def wf-env-def*)

ultimately show *?thesis*
by (*auto simp*: *pre-dfss-def*)
qed

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*
using *assms* **unfolding** *pre-dfss-def pre-dfs-def wf-env-def*
by (*meson succ-reachable*)

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* := ($\lambda x.$ *if* x=v *then vsuccs e v* $\cup$ {*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 −
from *1* **have** *v*: $v \in$ *visited e*
by (*simp add*: *pre-dfss-def*)
have *wf-env e′′*
proof −
from *1* **have** *wf*: *wf-env e*
by (*simp add*: *pre-dfss-def*)
hence $\forall v \in$ *visited e′′*. *reachable* (*root e′′*) *v*
distinct (*stack e′′*)
distinct (*cstack e′′*)
$\forall n\ m.\ n \preceq m$ *in stack e′′* $\longrightarrow n \preceq m$ *in cstack e′′*
$\forall n\ m.\ n \preceq m$ *in stack e′′* $\longrightarrow$ *reachable m n*
explored e′′ $\subseteq$ *visited e′′*
set (*cstack e′′*) $\subseteq$ *visited e′′*
$\forall n \in$ *explored e′′*. $\forall m.$ *reachable n m* $\longrightarrow m \in$ *explored e′′*
$\forall n\ m.\ m \in \mathcal{S}\ e''\ n \longleftrightarrow (\mathcal{S}\ e''\ n = \mathcal{S}\ e''\ m)$
$\forall n.\ n \notin$ *visited e′′* $\longrightarrow \mathcal{S}\ e''\ n = \{n\}$
$\forall n \in$ *set* (*stack e′′*). $\forall\ m \in$ *set* (*stack e′′*).

$n \neq m \longrightarrow \mathcal{S}\ e''\ n \cap \mathcal{S}\ e''\ m = \{\}$
$\bigcup\ \{\mathcal{S}\ e''\ n \mid n.\ n \in set\ (stack\ e'')\} = visited\ e'' - explored\ e''$
$\forall\, n \in set\ (stack\ e'').\ \forall\, m \in \mathcal{S}\ e''\ n.$
$m \in set\ (cstack\ e'') \longrightarrow m \preceq n\ in\ cstack\ e''$
$\forall\, n.\ is\text{-}subscc\ (\mathcal{S}\ e''\ n)$
$\forall\, S \in sccs\ e''.\ is\text{-}scc\ S$
$\bigcup\ (sccs\ e'') = explored\ e''$
by (*auto simp*: *wf-env-def e''-def*)
moreover
from *wf 2 3* **have** $\forall\, v.\ vsuccs\ e''\ v \subseteq successors\ v \cap visited\ e''$
by (*auto simp*: *wf-env-def e''-def*)
moreover
from *wf v* **have** $\forall\, n.\ n \notin visited\ e'' \longrightarrow vsuccs\ e''\ n = \{\}$
by (*auto simp*: *wf-env-def e''-def*)
moreover
from *wf 2*
have $\forall\, v.\ v \in explored\ e'' \longrightarrow vsuccs\ e''\ v = successors\ v$
by (*auto simp*: *wf-env-def e''-def*)
moreover
have $\forall\, x\ y.\ x \preceq y\ in\ stack\ e'' \wedge x \neq y \longrightarrow$
$(\forall\, u \in \mathcal{S}\ e''\ x.\ \neg\ reachable\text{-}avoiding\ u\ y\ (unvisited\ e''\ x))$
proof (*clarify*)
fix *x y u*
assume $x \preceq y\ in\ stack\ e''\ x \neq y$
$u \in \mathcal{S}\ e''\ x$
$reachable\text{-}avoiding\ u\ y\ (unvisited\ e''\ x)$
hence *prec*: $x \preceq y\ in\ stack\ e\ u \in \mathcal{S}\ e\ x$
by (*auto simp*: *e''-def*)
with *stack-unexplored*[*OF wf*] **have** $y \notin explored\ e$
by (*blast dest*: *precedes-mem*)
have $(unvisited\ e\ x = unvisited\ e''\ x)$
$\vee\ (unvisited\ e\ x = unvisited\ e''\ x \cup \{(v,w)\})$
by (*auto simp*: *e''-def unvisited-def split*: *if-splits*)
thus *False*
proof
assume $unvisited\ e\ x = unvisited\ e''\ x$
with *prec* ‹$x \neq y$› ‹$reachable\text{-}avoiding\ u\ y\ (unvisited\ e''\ x)$› *wf*
show *?thesis*
unfolding *wf-env-def* **by** *metis*
next
assume $unvisited\ e\ x = unvisited\ e''\ x \cup \{(v,w)\}$
with *wf* ‹$reachable\text{-}avoiding\ u\ y\ (unvisited\ e''\ x)$›
‹$y \notin explored\ e$› ‹$w \in explored\ e$› *prec* ‹$x \neq y$›
show *?thesis*
using *avoiding-explored*[*OF wf*] **unfolding** *wf-env-def*
by (*metis* (*no-types*, *lifting*))
qed
qed
moreover

from *wf 2*
have $\forall n \in$ *visited* $e'' -$ *set* (*cstack* e''). *vsuccs* e'' n = *successors* n
by (*auto simp*: e''*-def wf-env-def*)
ultimately show *?thesis*
unfolding *wf-env-def* **by** *meson*
qed
with *1 3* **show** *?thesis*
by (*auto simp*: *pre-dfss-def* e''*-def*)
qed

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* := (λx. *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 −
from *pre*
have *wf-env e* $v \in$ *visited e stack e* $\neq$ [] $v \in \mathcal{S}$ *e* (*hd* (*stack e*))
by (*auto simp*: *pre-dfss-def*)
with *post* **have** *stack* $e' \neq$ [] $v \in \mathcal{S}$ e' (*hd* (*stack* e'))
by (*auto dest*: *dfs-S-hd-stack*)

from *post* **have** $w \in$ *visited* e'
by (*simp add*: *post-dfs-def*)

have *wf-env* e''
proof −
from *post* **have** *wf'*: *wf-env* e'
by (*simp add*: *post-dfs-def*)
hence $\forall n \in$ *visited* e''. *reachable* (*root* e'') n
distinct (*stack* e'')
distinct (*cstack* e'')
$\forall n\ m.\ n \preceq m$ *in stack* $e'' \longrightarrow n \preceq m$ *in cstack* e''
$\forall n\ m.\ n \preceq m$ *in stack* $e'' \longrightarrow$ *reachable m n*
explored $e'' \subseteq$ *visited* e''
set (*cstack* e'') $\subseteq$ *visited* e''
$\forall n \in$ *explored* e''. $\forall m$. *reachable n m* $\longrightarrow m \in$ *explored* e''
$\forall n\ m.\ m \in \mathcal{S}\ e''\ n \longleftrightarrow (\mathcal{S}\ e''\ n = \mathcal{S}\ e''\ m)$
$\forall n.\ n \notin$ *visited* $e'' \longrightarrow \mathcal{S}\ e''\ n = \{n\}$
$\forall n \in$ *set* (*stack* e''). $\forall\ m \in$ *set* (*stack* e'').
$n \neq m \longrightarrow \mathcal{S}\ e''\ n \cap \mathcal{S}\ e''\ m = \{\}$
$\bigcup\ \{\mathcal{S}\ e''\ n \mid n.\ n \in$ *set* (*stack* e'')$\}$ = *visited* e'' − *explored* e''
$\forall n \in$ *set* (*stack* e''). $\forall\ m \in \mathcal{S}\ e''\ n.\ m \in$ *set* (*cstack* e'') $\longrightarrow m \preceq n$ *in cstack* e''

$\forall\, n.$ *is-subscc* $(\mathcal{S}\ e''\ n)$
$\forall\, S \in$ *sccs* $e''.$ *is-scc* S
$\bigcup$ (*sccs* e'') = *explored* e''
by (*auto simp*: *wf-env-def* e''*-def*)
moreover
from *wf'* *w* **have** $\forall\, n.$ *vsuccs* $e''\ n \subseteq$ *successors* n
by (*auto simp*: *wf-env-def* e''*-def*)
moreover
from *wf'* ‹$w \in$ *visited* e'› **have** $\forall\, n.$ *vsuccs* $e''\ n \subseteq$ *visited* e''
by (*auto simp*: *wf-env-def* e''*-def*)
moreover
from *post* ‹$v \in$ *visited* e›
have $\forall\, n.\ n \notin$ *visited* $e'' \longrightarrow$ *vsuccs* $e''\ n = \{\}$
apply (*simp add*: *post-dfs-def wf-env-def sub-env-def* e''*-def*)
by (*meson subsetD*)
moreover
from *wf'* *w*
have $\forall\, n \in$ *explored* $e''.$ *vsuccs* $e''\ n =$ *successors* n
by (*auto simp*: *wf-env-def* e''*-def*)
moreover
have $\forall\, n\ m.\ n \preceq m$ *in stack* $e'' \wedge n \neq m \longrightarrow$
($\forall\, u \in \mathcal{S}\ e''\ n.\ \neg$ *reachable-avoiding* $u\ m$ (*unvisited* $e''\ n$))
proof (*clarify*)
fix $x\ y\ u$
assume $x \preceq y$ *in stack* $e''\ x \neq y$
$u \in \mathcal{S}\ e''\ x$
reachable-avoiding $u\ y$ (*unvisited* $e''\ x$)
hence *1*: $x \preceq y$ *in stack* $e'\ u \in \mathcal{S}\ e'\ x$
by (*auto simp*: e''*-def*)
with *stack-unexplored*[*OF wf'*] **have** $y \notin$ *explored* e'
by (*auto dest*: *precedes-mem*)
have (*unvisited* $e'\ x$ = *unvisited* $e''\ x$)
$\vee$ (*unvisited* $e'\ x$ = *unvisited* $e''\ x \cup \{(v,w)\}$)
by (*auto simp*: e''*-def unvisited-def split*: *if-splits*)
thus *False*
proof
assume *unvisited* $e'\ x$ = *unvisited* $e''\ x$
with *1* ‹$x \neq y$› ‹*reachable-avoiding* $u\ y$ (*unvisited* $e''\ x$)› *wf'*
show *?thesis*
unfolding *wf-env-def* **by** *metis*
next
assume *unv*: *unvisited* $e'\ x$ = *unvisited* $e''\ x \cup \{(v,w)\}$
from *post*
have $w \in$ *explored* e'
$\vee$ ($w \in \mathcal{S}\ e'$ (*hd* (*stack* e')) $\wedge$ ($\forall\, n \in$ *set* (*tl* (*stack* e')). $\mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$))
by (*auto simp*: *post-dfs-def*)
thus *?thesis*
proof
assume $w \in$ *explored* e'

with *wf′ unv* ‹*reachable-avoiding u y* (*unvisited e″ x*)›
‹$y \notin$ *explored e′*› *1* ‹$x \neq y$›
show *?thesis*
using *avoiding-explored*[*OF wf′*] **unfolding** *wf-env-def*
by (*metis* (*no-types*, *lifting*))
next
assume *w*: $w \in \mathcal{S}\ e'$ (*hd* (*stack e′*))
$\wedge$ ($\forall n \in$ *set* (*tl* (*stack e′*)). $\mathcal{S}\ e'\ n = \mathcal{S}\ e\ n$)
from ‹*reachable-avoiding u y* (*unvisited e″ x*)›[*THEN ra-add-edge*]
unv
have *reachable-avoiding u y* (*unvisited e′ x*)
$\vee$ *reachable-avoiding w y* (*unvisited e′ x*)
by *auto*
thus *?thesis*
proof
assume *reachable-avoiding u y* (*unvisited e′ x*)
with ‹$x \preceq y$ *in stack e″*› ‹$x \neq y$› ‹$u \in \mathcal{S}\ e''\ x$› *wf′*
show *?thesis*
by (*auto simp*: *e″-def wf-env-def*)
next
assume *reachable-avoiding w y* (*unvisited e′ x*)
from *unv* **have** $v \in \mathcal{S}\ e'\ x$
by (*auto simp*: *unvisited-def*)
from ‹$x \preceq y$ *in stack e″*› **have** $x \in$ *set* (*stack e′*)
by (*simp add*: *e″-def precedes-mem*)
have $x =$ *hd* (*stack e′*)
proof (*rule ccontr*)
assume $x \neq$ *hd* (*stack e′*)
with ‹$x \in$ *set* (*stack e′*)› ‹*stack e′* $\neq$ []›
have $x \in$ *set* (*tl* (*stack e′*))
by (*metis hd-Cons-tl set-ConsD*)
with *w* ‹$v \in \mathcal{S}\ e'\ x$› **have** $v \in \mathcal{S}\ e\ x$
by *auto*
moreover
from *post* ‹*stack e′* $\neq$ []› ‹$x \in$ *set* (*stack e′*)› ‹$x \in$ *set* (*tl* (*stack e′*))›
have $x \in$ *set* (*tl* (*stack e*))
unfolding *post-dfs-def*
by (*metis Un-iff self-append-conv2 set-append tl-append2*)
moreover
from *pre* **have** *wf-env e stack e* $\neq$ [] $v \in \mathcal{S}\ e$ (*hd* (*stack e*))
by (*auto simp*: *pre-dfss-def*)
ultimately show *False*
unfolding *wf-env-def*
by (*metis* (*no-types*, *lifting*) *distinct.simps*(*2*) *hd-Cons-tl in-
sert-disjoint*(*2*)
list.set-sel(*1*) *list.set-sel*(*2*) *mk-disjoint-insert*)
qed
with ‹*reachable-avoiding w y* (*unvisited e′ x*)›
‹$x \preceq y$ *in stack e″*› ‹$x \neq y$› *w wf′*

```
          show ?thesis
            by (auto simp add: e''-def wf-env-def)
        qed
      qed
    qed
  qed

  moreover
  from wf' ‹∀ n. vsuccs e'' n ⊆ successors n›
  have ∀ n ∈ visited e'' − set (cstack e''). vsuccs e'' n = successors n
    by (auto simp: wf-env-def e''-def split: if-splits)
  ultimately show ?thesis
    unfolding wf-env-def by (meson le-inf-iff)
qed

show pre-dfss v e''
proof −
  from pre post
  have v ∈ visited e''
    by (auto simp: pre-dfss-def post-dfs-def sub-env-def e''-def)
  moreover
  {
    fix u
    assume u: u ∈ vsuccs e'' v
    have u ∈ explored e'' ∪ 𝒮 e'' (hd (stack e''))
    proof (cases u = w)
      case True
      with post show ?thesis
        by (auto simp: post-dfs-def e''-def)
    next
      case False
      with u pre post
      have u ∈ explored e ∪ 𝒮 e (hd (stack e))
        by (auto simp: pre-dfss-def post-dfs-def e''-def)
      then show ?thesis
      proof
        assume u ∈ explored e
        with post show ?thesis
          by (auto simp: post-dfs-def sub-env-def e''-def)
      next
        assume u ∈ 𝒮 e (hd (stack e))
        with ‹wf-env e› post ‹stack e ≠ []›
        show ?thesis
          by (auto simp: e''-def dest: dfs-S-hd-stack)
      qed
    qed
  }
  moreover
  from pre post
```

have $\forall n \in$ *set* (*stack* e''). *reachable n v*
  unfolding *pre-dfss-def post-dfs-def*
  using e''*-def* **by** *force*
moreover
from ‹*stack* $e' \neq []$› **have** *stack* $e'' \neq []$
  by (*simp add*: e''*-def*)
moreover
from ‹$v \in \mathcal{S}\ e'\ (hd\ (stack\ e'))$› **have** $v \in \mathcal{S}\ e''\ (hd\ (stack\ e''))$
  by (*simp add*: e''*-def*)
moreover
from *pre post* **have** $\exists\, ns.\ cstack\ e'' = v \ \#\ ns$
  by (*auto simp*: *pre-dfss-def post-dfs-def* e''*-def*)
ultimately show *?thesis*
  using ‹*wf-env* e''› **unfolding** *pre-dfss-def* **by** *blast*
qed
qed

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 −
  from *pre* **have** *wf*: *wf-env e*
    by (*simp add*: *pre-dfss-def*)
  from *pre* **have** $v \in$ *visited e*
    by (*simp add*: *pre-dfss-def*)
  from *pre w* **have** $v \notin$ *explored e*
    unfolding *pre-dfss-def wf-env-def*
    by (*meson reachable-edge*)

  from *unite-stack*[*OF wf w*] **obtain** *pfx* **where**
    pfx: *stack e* = *pfx* @ *stack* e' *stack* $e' \neq []$
      $let\ cc = \bigcup\ \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$
       $in\ \mathcal{S}\ e' = (\lambda x.\ if\ x \in cc\ then\ cc\ else\ \mathcal{S}\ e\ x)$
      $w \in \mathcal{S}\ e'\ (hd\ (stack\ e'))$
    by (*auto simp*: e'*-def*)
  define *cc* **where** $cc \equiv \bigcup\ \{\mathcal{S}\ e\ n\ |n.\ n \in set\ pfx \cup \{hd\ (stack\ e')\}\}$

  from *unite-wf-env*[*OF pre w*] **have** *wf'*: *wf-env* e'
    by (*simp add*: e'*-def*)

  from ‹*stack e* = *pfx* @ *stack* e'› ‹*stack* $e' \neq []$›
  have *hd* (*stack e*) $\in$ *set pfx* $\cup$ {*hd* (*stack* e')}
    by (*simp add*: *hd-append*)

with *pre* **have** $v \in cc$
by $(auto\ simp\colon pre\text{-}dfss\text{-}def\ cc\text{-}def)$
from $S\text{-}reflexive[OF\ wf,\ of\ hd\ (stack\ e')]$
have $hd\ (stack\ e') \in cc$
by $(auto\ simp\colon cc\text{-}def)$
with *pfx* $\langle v \in cc\rangle$ **have** $v \in \mathcal{S}\ e'\ (hd\ (stack\ e'))$
by $(auto\ simp\colon cc\text{-}def)$

from $unite\text{-}sub\text{-}env[OF\ pre\ w]$ **have** $sub\text{-}env\ e\ e'$
by $(simp\ add\colon e'\text{-}def)$

have $wf\text{-}env\ e''$
proof $-$
from wf'
have $\forall n \in visited\ e''.\ reachable\ (root\ e'')\ n$
$distinct\ (stack\ e'')$
$distinct\ (cstack\ e'')$
$\forall n\ m.\ n \preceq m\ in\ stack\ e'' \longrightarrow n \preceq m\ in\ cstack\ e''$
$\forall n\ m.\ n \preceq m\ in\ stack\ e'' \longrightarrow reachable\ m\ n$
$explored\ e'' \subseteq visited\ e''$
$set\ (cstack\ e'') \subseteq visited\ e''$
$\forall n \in explored\ e''.\ \forall m.\ reachable\ n\ m \longrightarrow m \in explored\ e''$
$\forall n\ m.\ m \in \mathcal{S}\ e''\ n \longleftrightarrow (\mathcal{S}\ e''\ n = \mathcal{S}\ e''\ m)$
$\forall n.\ n \notin visited\ e'' \longrightarrow \mathcal{S}\ e''\ n = \{n\}$
$\forall n \in set\ (stack\ e'').\ \forall m \in set\ (stack\ e'').$
$n \neq m \longrightarrow \mathcal{S}\ e''\ n \cap \mathcal{S}\ e''\ m = \{\}$
$\bigcup\ \{\mathcal{S}\ e''\ n \mid n.\ n \in set\ (stack\ e'')\} = visited\ e'' - explored\ e''$
$\forall n \in set\ (stack\ e'').\ \forall m \in \mathcal{S}\ e''\ n.$
$m \in set\ (cstack\ e'') \longrightarrow m \preceq n\ in\ cstack\ e''$
$\forall n.\ is\text{-}subscc\ (\mathcal{S}\ e''\ n)$
$\forall S \in sccs\ e''.\ is\text{-}scc\ S$
$\bigcup\ (sccs\ e'') = explored\ e''$
by $(auto\ simp\colon wf\text{-}env\text{-}def\ e''\text{-}def)$

moreover
from $wf'\ w\ \langle sub\text{-}env\ e\ e'\rangle$
have $\forall n.\ vsuccs\ e''\ n \subseteq successors\ n \cap visited\ e''$
by $(auto\ simp\colon wf\text{-}env\text{-}def\ sub\text{-}env\text{-}def\ e''\text{-}def)$

moreover
from $wf'\ \langle v \in visited\ e\rangle\ \langle sub\text{-}env\ e\ e'\rangle$
have $\forall n.\ n \notin visited\ e'' \longrightarrow vsuccs\ e''\ n = \{\}$
by $(auto\ simp\colon wf\text{-}env\text{-}def\ sub\text{-}env\text{-}def\ e''\text{-}def)$

moreover
from $wf'\ \langle v \notin explored\ e\rangle$
have $\forall n \in explored\ e''.\ vsuccs\ e''\ n = successors\ n$
by $(auto\ simp\colon wf\text{-}env\text{-}def\ e''\text{-}def\ e'\text{-}def\ unite\text{-}def)$

```
moreover
from wf′ ‹w ∈ successors v›
have ∀ n ∈ visited e′′ − set (cstack e′′). vsuccs e′′ n = successors n
  by (auto simp: wf-env-def e′′-def e′-def unite-def)

moreover
have ∀ x y. x ⪯ y in stack e′′ ∧ x ≠ y ⟶
            (∀ u ∈ 𝒮 e′′ x. ¬ reachable-avoiding u y (unvisited e′′ x))
proof (clarify)
  fix x y u
  assume xy: x ⪯ y in stack e′′ x ≠ y
     and u: u ∈ 𝒮 e′′ x reachable-avoiding u y (unvisited e′′ x)
  hence prec: x ⪯ y in stack e′ u ∈ 𝒮 e′ x
    by (simp add: e′′-def)+
  show False
  proof (cases x = hd (stack e′))
    case True
    with ‹v ∈ 𝒮 e′ (hd (stack e′))›
    have unvisited e′ x = unvisited e′′ x
        ∨ (unvisited e′ x = unvisited e′′ x ∪ {(v,w)})
      by (auto simp: e′′-def unvisited-def split: if-splits)
    thus False
    proof
      assume unvisited e′ x = unvisited e′′ x
      with prec ‹x ≠ y› ‹reachable-avoiding u y (unvisited e′′ x)› wf′
      show ?thesis
        unfolding wf-env-def by metis
    next
      assume unvisited e′ x = unvisited e′′ x ∪ {(v,w)}
      with ‹reachable-avoiding u y (unvisited e′′ x)›[THEN ra-add-edge]
      have reachable-avoiding u y (unvisited e′ x)
          ∨ reachable-avoiding w y (unvisited e′ x)
        by auto
      thus ?thesis
      proof
        assume reachable-avoiding u y (unvisited e′ x)
        with prec ‹x ≠ y› wf′ show ?thesis
          by (auto simp: wf-env-def)
      next
        assume reachable-avoiding w y (unvisited e′ x)
        with ‹x = hd (stack e′)› ‹w ∈ 𝒮 e′ (hd (stack e′))›
            ‹x ⪯ y in stack e′› ‹x ≠ y› wf′
        show ?thesis
          by (auto simp: wf-env-def)
      qed
    qed
  next
    case False
    with ‹x ⪯ y in stack e′› ‹stack e′ ≠ []›
```

```
      have x ∈ set (tl (stack e'))
        by (metis list.exhaust-sel precedes-mem(1) set-ConsD)
      with unite-S-tl[OF wf w] ‹u ∈ 𝒮 e' x›
      have u ∈ 𝒮 e x
        by (simp add: e'-def)
      moreover
      from ‹x ≼ y in stack e'› ‹stack e = pfx @ stack e'›
      have x ≼ y in stack e
        by (simp add: precedes-append-left)
      moreover
      from ‹v ∈ 𝒮 e' (hd (stack e'))› ‹x ∈ set (tl (stack e'))›
          ‹stack e' ≠ []› wf'
      have v ∉ 𝒮 e' x
        unfolding wf-env-def
          by (metis (no-types, lifting) Diff-cancel Diff-triv distinct.simps(2) in-
sert-not-empty
                list.exhaust-sel list.set-sel(1) list.set-sel(2) mk-disjoint-insert)
      hence unvisited e'' x = unvisited e' x
        by (auto simp: unvisited-def e''-def split: if-splits)
      moreover
      from ‹x ∈ set (tl (stack e'))› unite-S-tl[OF wf w]
      have unvisited e' x = unvisited e x
        by (simp add: unvisited-def e'-def unite-def)
      ultimately show ?thesis
        using ‹x ≠ y› ‹reachable-avoiding u y (unvisited e'' x)› wf
        by (auto simp: wf-env-def)
    qed
  qed

  ultimately show ?thesis
    unfolding wf-env-def by meson
qed

show pre-dfss v e''
proof –
  from pre have v ∈ visited e''
    by (simp add: pre-dfss-def e''-def e'-def unite-def)

  moreover
  {
    fix u
    assume u: u ∈ vsuccs e'' v
    have u ∈ explored e'' ∪ 𝒮 e'' (hd (stack e''))
    proof (cases u = w)
      case True
      with ‹w ∈ 𝒮 e' (hd (stack e'))› show ?thesis
        by (simp add: e''-def)
    next
      case False
```

```
        with u have u ∈ vsuccs e v
          by (simp add: e''-def e'-def unite-def)
        with pre have u ∈ explored e ∪ 𝒮 e (hd (stack e))
          by (auto simp: pre-dfss-def)
        then show ?thesis
        proof
          assume u ∈ explored e
          thus ?thesis
            by (simp add: e''-def e'-def unite-def)
        next
          assume u ∈ 𝒮 e (hd (stack e))
          with ‹hd (stack e) ∈ set pfx ∪ {hd (stack e')}›
          have u ∈ cc
            by (auto simp: cc-def)
          moreover
          from S-reflexive[OF wf, of hd (stack e')] pfx
          have 𝒮 e' (hd (stack e')) = cc
            by (auto simp: cc-def)
          ultimately show ?thesis
            by (simp add: e''-def)
        qed
      qed
    }
    hence ∀ w ∈ vsuccs e'' v. w ∈ explored e'' ∪ 𝒮 e'' (hd (stack e''))
      by blast

    moreover
    from pre ‹stack e = pfx @ stack e'›
    have ∀ n ∈ set (stack e''). reachable n v
      by (auto simp: pre-dfss-def e''-def)

    moreover
    from ‹stack e' ≠ []› have stack e'' ≠ []
      by (simp add: e''-def)

    moreover
    from ‹v ∈ 𝒮 e' (hd (stack e'))› have v ∈ 𝒮 e'' (hd (stack e''))
      by (simp add: e''-def)

    moreover
    from pre have ∃ ns. cstack e'' = v # ns
      by (auto simp: pre-dfss-def e''-def e'-def unite-def)

    ultimately show ?thesis
      using ‹wf-env e''› unfolding pre-dfss-def by blast
  qed
qed
```

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 ≡ e⦇visited := visited e ∪ {v},
                stack := (v # stack e),
                cstack:=(v # cstack e)⦈
  defines e′ ≡ dfss v e1
  defines e″ ≡ 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 −
  from 1 have wf: wf-env e
    by (simp add: pre-dfs-def)
  from 1 have v: v ∉ visited e
    by (simp add: pre-dfs-def)
  from 3 have wf′: wf-env e′
    by (simp add: post-dfss-def)
  from 3 have cst′: cstack e′ = v # cstack e
    by (simp add: post-dfss-def e1-def)
  show ?thesis
  proof (cases v = hd(stack e′))
    case True
    have notempty: stack e′ = v # stack e
    proof −
      from 3 obtain ns where
        ns: stack e1 = ns @ (stack e′) stack e′ ≠ []
        by (auto simp: post-dfss-def)
      have ns = []
      proof (rule ccontr)
        assume ns ≠ []
        with ns have hd ns = v
          apply (simp add: e1-def)
          by (metis hd-append2 list.sel(1))
        with True ns ‹ns ≠ []› have ¬ distinct (stack e1)
          by (metis disjoint-iff-not-equal distinct-append hd-in-set)
        with wf v stack-visited[OF wf] show False
          by (auto simp: wf-env-def e1-def)
      qed
      with ns show ?thesis
        by (simp add: e1-def)
    qed
    have e2: dfs v e = e′⦇sccs := sccs e′ ∪ {𝒮 e′ v},
                         explored := explored e′ ∪ (𝒮 e′ v),
                         stack := tl (stack e′),
```

cstack := *tl* (*cstack e′*)⦈ (**is** - = *?e2*)
using *True 2 dfs.psimps*[*of v e*] **unfolding** *e1-def e′-def*
by (*fastforce simp*: *e1-def e′-def*)

have *sub*: *sub-env e e1*
by (*auto simp*: *sub-env-def e1-def*)

from *notempty* **have** *stack2*: *stack ?e2* = *stack e*
by (*simp add*: *e1-def*)

moreover from *3* **have** $v \in$ *visited ?e2*
by (*auto simp*: *post-dfss-def sub-env-def e1-def*)

moreover
from *sub 3* **have** *sub-env e e′*
unfolding *post-dfss-def* **by** (*auto elim*: *sub-env-trans*)
with *stack2* **have** *subenv*: *sub-env e ?e2*
by (*fastforce simp*: *sub-env-def*)

moreover have *wf-env ?e2*
proof −
from *wf′*
have $\forall n \in$ *visited ?e2*. *reachable* (*root ?e2*) *n*
distinct (*stack ?e2*)
$\forall n.$ *vsuccs ?e2 n* $\subseteq$ *successors n* $\cap$ *visited ?e2*
$\forall n.$ $n \notin$ *visited ?e2* $\longrightarrow$ *vsuccs ?e2 n* = {}
$\forall n\ m.$ $m \in \mathcal{S}$ *?e2 n* $\longleftrightarrow$ ($\mathcal{S}$ *?e2 n* = $\mathcal{S}$ *?e2 m*)
$\forall n.$ $n \notin$ *visited ?e2* $\longrightarrow$ $\mathcal{S}$ *?e2 n* = {*n*}
$\forall n.$ *is-subscc* ($\mathcal{S}$ *?e2 n*)
$\bigcup$ (*sccs ?e2*) = *explored ?e2*
by (*auto simp*: *wf-env-def distinct-tl*)

moreover
from *1 cst′* **have** *distinct* (*cstack ?e2*)
by (*auto simp*: *pre-dfs-def wf-env-def*)

moreover
from *1 stack2* **have** $\forall n\ m.$ $n \preceq m$ *in stack ?e2* $\longrightarrow$ *reachable m n*
by (*auto simp*: *pre-dfs-def wf-env-def*)

moreover
from *1 stack2 cst′*
have $\forall n\ m.$ $n \preceq m$ *in stack ?e2* $\longrightarrow$ $n \preceq m$ *in cstack ?e2*
by (*auto simp*: *pre-dfs-def wf-env-def*)

moreover
from *notempty wf′* **have** *explored ?e2* $\subseteq$ *visited ?e2*
apply (*simp add*: *wf-env-def*)
using *stack-class*[*OF wf′*]

```
  by (smt (verit, del-insts) Diff-iff insert-subset list.simps(15) subset-eq)

moreover
from 3 cst' have set (cstack ?e2) ⊆ visited ?e2
  by (simp add: post-dfss-def wf-env-def e1-def)

moreover
{
  fix u
  assume u ∈ explored ?e2
  have vsuccs ?e2 u = successors u
  proof (cases u ∈ explored e')
    case True
    with wf' show ?thesis
      by (auto simp: wf-env-def)
  next
    case False
    with ‹u ∈ explored ?e2› have u ∈ 𝒮 e' v
      by simp
    show ?thesis
    proof (cases u = v)
      case True
      with 3 show ?thesis
        by (auto simp: post-dfss-def)
    next
      case False
      have u ∈ visited e' − set (cstack e')
      proof
        from notempty ‹u ∈ 𝒮 e' v› stack-class[OF wf'] False
        show u ∈ visited e'
          by auto
      next
        show u ∉ set (cstack e')
        proof
          assume u: u ∈ set (cstack e')
          with notempty ‹u ∈ 𝒮 e' v› ‹wf-env e'› have u ≼ v in cstack e'
            by (auto simp: wf-env-def)
          with cst' u False wf' show False
            unfolding wf-env-def
            by (metis head-precedes precedes-antisym)
        qed
      qed
      with 3 show ?thesis
        by (auto simp: post-dfss-def wf-env-def)
    qed
  qed
}
note explored-vsuccs = this
```

```
moreover have ∀ n ∈ explored ?e2. ∀ m. reachable n m ⟶ m ∈ explored ?e2
 proof (clarify)
  fix x y
  assume asm: x ∈ explored ?e2 reachable x y
  show y ∈ explored ?e2
  proof (cases x ∈ explored e')
    case True
    with ‹wf-env e'› ‹reachable x y› show ?thesis
      by (simp add: wf-env-def)
  next
    case False
    with asm have x ∈ 𝒮 e' v
       by simp
    with ‹explored ?e2 ⊆ visited ?e2› have x ∈ visited e'
      by auto
    from ‹x ∈ 𝒮 e' v› wf' have reachable v x
      by (auto simp: wf-env-def is-subscc-def)
    have y ∈ visited e'
    proof (rule ccontr)
      assume y ∉ visited e'
      with reachable-visited[OF wf' ‹x ∈ visited e'› ‹reachable x y›]
      obtain n m where
        n ∈ visited e' m ∈ successors n − vsuccs e' n
        reachable x n reachable m y
        by blast
      from wf' ‹m ∈ successors n − vsuccs e' n›
      have n ∉ explored e'
        by (auto simp: wf-env-def)
      obtain n' where
        n' ∈ set (stack e') n ∈ 𝒮 e' n'
        by (rule visited-unexplored[OF wf' ‹n ∈ visited e'› ‹n ∉ explored e'›])
      have n' = v
      proof (rule ccontr)
        assume n' ≠ v
        with ‹n' ∈ set (stack e')› ‹v = hd (stack e')›
        have n' ∈ set (tl (stack e'))
          by (metis emptyE hd-Cons-tl set-ConsD set-empty)
        moreover
        from ‹n ∈ 𝒮 e' n'› ‹wf-env e'› have reachable n n'
          by (auto simp: wf-env-def is-subscc-def)
        with ‹reachable v x› ‹reachable x n› reachable-trans
        have reachable v n'
          by blast
        ultimately show False
          using 3 ‹v = hd (stack e')›
          by (auto simp: post-dfss-def)
      qed
      with ‹n ∈ 𝒮 e' n'› ‹m ∈ successors n − vsuccs e' n› explored-vsuccs
      show False
```

```
        by auto
      qed
      show ?thesis
      proof (cases y ∈ explored e′)
        case True
        then show ?thesis
          by simp
      next
        case False
        obtain n where ndef: n ∈ set (stack e′) (y ∈ 𝒮 e′ n)
          by (rule visited-unexplored[OF wf′ ‹y ∈ visited e′› False])
        show ?thesis
        proof (cases n = v)
          case True
          with ndef show ?thesis by simp
        next
          case False
          with ndef notempty have n ∈ set (tl (stack e′))
            by simp
          moreover
          from wf′ ndef have reachable y n
            by (auto simp: wf-env-def is-subscc-def)
          with ‹reachable v x› ‹reachable x y›
          have reachable v n
            by (meson reachable-trans)
          ultimately show ?thesis
            using ‹v = hd (stack e′)› 3
            by (simp add: post-dfss-def)
        qed
      qed
    qed
  qed

  moreover
  from 3 cst′
  have ∀ n ∈ visited ?e2 − set (cstack ?e2). vsuccs ?e2 n = successors n
    apply (simp add: post-dfss-def wf-env-def)
    by (metis (no-types, lifting) Diff-empty Diff-iff empty-set insertE
              list.exhaust-sel list.sel(1) list.simps(15))

  moreover
  from wf′ notempty
  have ∀ n m. n ∈ set (stack ?e2) ∧ m ∈ set (stack ?e2) ∧ n ≠ m
              ⟶ (𝒮 ?e2 n ∩ 𝒮 ?e2 m = {})
    by (simp add: wf-env-def)

  moreover
  have ⋃ {𝒮 ?e2 n | n . n ∈ set (stack ?e2)} = visited ?e2 − explored ?e2
  proof −
```

from *wf′ notempty*
have $(\bigcup \{\mathcal{S}\ ?e2\ n \mid n\ .\ n \in set\ (stack\ ?e2)\}) \cap \mathcal{S}\ e'\ v = \{\}$
by (*auto simp*: *wf-env-def*)
with *notempty*
have $\bigcup \{\mathcal{S}\ ?e2\ n \mid n\ .\ n \in set\ (stack\ ?e2)\} =$
$(\bigcup \{\mathcal{S}\ e'\ n \mid n\ .\ n \in set\ (stack\ e')\}) - \mathcal{S}\ e'\ v$
by *auto*
also from *wf′*
have $\ldots = (visited\ e' - explored\ e') - \mathcal{S}\ e'\ v$
by (*simp add*: *wf-env-def*)
finally show *?thesis*
by *auto*
qed

moreover
have $\forall\, n \in set\ (stack\ ?e2).\ \forall\, m \in \mathcal{S}\ ?e2\ n.\ m \in set\ (cstack\ ?e2) \longrightarrow m \preceq n$ *in cstack ?e2*
proof (*clarsimp simp*: *cst′*)
fix *n m*
assume $n \in set\ (tl\ (stack\ e'))$
$m \in \mathcal{S}\ e'\ n\ m \in set\ (cstack\ e)$
with *3* **have** $m \in \mathcal{S}\ e\ n$
by (*auto simp*: *post-dfss-def e1-def*)
with *wf notempty* ‹$n \in set\ (tl\ (stack\ e'))$› ‹$m \in set\ (cstack\ e)$›
show $m \preceq n$ *in cstack e*
by (*auto simp*: *wf-env-def*)
qed

moreover
{
fix *x y u*
assume *xy*: $x \preceq y$ *in stack ?e2* $x \neq y$
and *u*: $u \in \mathcal{S}\ ?e2\ x$ *reachable-avoiding u y* (*unvisited ?e2 x*)
from *xy notempty stack2*
have $x \preceq y$ *in stack e′*
by (*metis head-precedes insert-iff list.simps*(*15*) *precedes-in-tail precedes-mem*(*2*))
with *wf′* ‹$x \neq y$› *u* **have** *False*
by (*auto simp*: *wf-env-def unvisited-def*)
}

moreover have $\forall\, S \in sccs\ ?e2.\ is\text{-}scc\ S$
proof (*clarify*)
fix *S*
assume *asm*: $S \in sccs\ ?e2$
show *is-scc S*
proof (*cases* $S = \mathcal{S}\ e'\ v$)
case *True*
with *S-reflexive*[*OF wf′*] **have** $S \neq \{\}$

```
        by blast
      from wf′ True have subscc: is-subscc S
        by (simp add: wf-env-def)
      {
        assume ¬ is-scc S
        with ‹S ≠ {}› ‹is-subscc S› obtain S′ where
          S′-def: S′ ≠ S S ⊆ S′ is-subscc S′
          unfolding is-scc-def by blast
        then obtain x where x ∈ S′ ∧ x ∉ S
          by blast
        with True S′-def wf′
        have xv: reachable v x ∧ reachable x v
          unfolding wf-env-def is-subscc-def by (metis in-mono)
        from ‹∀ v w. w ∈ 𝒮 ?e2 v ⟷ (𝒮 ?e2 v = 𝒮 ?e2 w)›
        have v ∈ explored ?e2
          by auto
        with ‹∀ x ∈ explored ?e2. ∀ y. reachable x y ⟶ y ∈ explored ?e2›
             xv ‹S = 𝒮 e′ v› ‹x ∈ S′ ∧ x ∉ S›
        have x ∈ explored e′
          by auto
        with wf′ xv have v ∈ explored e′
          by (auto simp: wf-env-def)
        with notempty have False
          by (auto intro: stack-unexplored[OF wf′])
      }
      then show ?thesis
        by blast
    next
      case False
      with asm wf′ show ?thesis
        by (auto simp: wf-env-def)
    qed
  qed

  ultimately show ?thesis
    unfolding wf-env-def by meson
qed

moreover
from ‹wf-env ?e2› have v ∈ explored ?e2
  by (auto simp: wf-env-def)

moreover
from 3 have vsuccs ?e2 v = successors v
  by (simp add: post-dfss-def)

moreover
from 1 3 have ∀ w ∈ visited e. vsuccs ?e2 w = vsuccs e w
  by (auto simp: pre-dfs-def post-dfss-def e1-def)
```

moreover
from *stack2 1*
have $\forall n \in set\ (stack\ ?e2).\ reachable\ n\ v$
  by (*simp add*: *pre-dfs-def*)

moreover
from *stack2* **have** $\exists ns.\ stack\ e = ns\ @\ (stack\ ?e2)$
  by *auto*

moreover
from *3* **have** $\forall n \in set\ (stack\ ?e2).\ \mathcal{S}\ ?e2\ n = \mathcal{S}\ e\ n$
  by (*auto simp*: *post-dfss-def e1-def*)

moreover
from *cst'* **have** $cstack\ ?e2 = cstack\ e$
  by *simp*

ultimately show *?thesis*
  unfolding *post-dfs-def* **using** *e2* **by** *simp*
next
  case *False*
  with *2* **have** *e'*: $dfs\ v\ e = e''$
    by (*simp add*: *dfs.psimps e''-def e'-def e1-def*)

  moreover have *wf-env e''*
  proof –
    from *wf'*
    have $\forall n \in visited\ e''.\ reachable\ (root\ e'')\ n$
      $distinct\ (stack\ e'')$
      $distinct\ (cstack\ e'')$
      $\forall n\ m.\ n \preceq m\ in\ stack\ e'' \longrightarrow reachable\ m\ n$
      $explored\ e'' \subseteq visited\ e''$
      $\forall n \in explored\ e''.\ \forall m.\ reachable\ n\ m \longrightarrow m \in explored\ e''$
      $\forall n.\ vsuccs\ e''\ n \subseteq successors\ n \cap visited\ e''$
      $\forall n.\ n \notin visited\ e'' \longrightarrow vsuccs\ e''\ n = \{\}$
      $\forall n \in explored\ e''.\ vsuccs\ e''\ n = successors\ n$
      $\forall n\ m.\ m \in \mathcal{S}\ e''\ n \longleftrightarrow (\mathcal{S}\ e''\ n = \mathcal{S}\ e''\ m)$
      $\forall n.\ n \notin visited\ e'' \longrightarrow \mathcal{S}\ e''\ n = \{n\}$
      $\forall n \in set\ (stack\ e'').\ \forall m \in set\ (stack\ e'').$
        $n \neq m \longrightarrow \mathcal{S}\ e''\ n \cap \mathcal{S}\ e''\ m = \{\}$
      $\bigcup\ \{\mathcal{S}\ e''\ n \mid n.\ n \in set\ (stack\ e'')\} = visited\ e'' - explored\ e''$
      $\forall n.\ is\text{-}subscc\ (\mathcal{S}\ e''\ n)$
      $\forall S \in sccs\ e''.\ is\text{-}scc\ S$
      $\bigcup\ (sccs\ e'') = explored\ e''$
    by (*auto simp*: *e''-def wf-env-def distinct-tl*)

  moreover have $\forall n\ m.\ n \preceq m\ in\ stack\ e'' \longrightarrow n \preceq m\ in\ cstack\ e''$
  proof (*clarsimp simp add*: *e''-def*)

```
  fix n m
  assume nm: n ≼ m in stack e′
  with 3 have n ≼ m in cstack e′
    unfolding post-dfss-def wf-env-def
    by meson
  moreover
  have n ≠ v
  proof
    assume n = v
    with nm have n ∈ set (stack e′)
      by (simp add: precedes-mem)
    with 3 ‹n = v› have v = hd (stack e′)
      unfolding post-dfss-def wf-env-def
      by (metis (no-types, opaque-lifting) IntI equals0D list.set-sel(1))
    with ‹v ≠ hd (stack e′)› show False
      by simp
  qed
  ultimately show n ≼ m in tl (cstack e′)
    by (simp add: cst′ precedes-in-tail)
qed

moreover
from 3 have set (cstack e″) ⊆ visited e″
  by (simp add: post-dfss-def wf-env-def e″-def e1-def subset-eq)

moreover
from 3
have ∀ n ∈ visited e″ − set (cstack e″). vsuccs e″ n = successors n
  apply (simp add: post-dfss-def wf-env-def e″-def e1-def)
  by (metis (no-types, opaque-lifting) DiffE DiffI set-ConsD)

moreover
have ∀ n ∈ set (stack e″). ∀ m ∈ 𝒮 e″ n.
        m ∈ set (cstack e″) ⟶ m ≼ n in cstack e″
proof (clarsimp simp: e″-def)
  fix m n
  assume asm: n ∈ set (stack e′) m ∈ 𝒮 e′ n
              m ∈ set (tl (cstack e′))
  with wf′ cst′ have m ≠ v m ≼ n in cstack e′
    by (auto simp: wf-env-def)
  with cst′ show m ≼ n in tl (cstack e′)
    by (simp add: precedes-in-tail)
qed

moreover
from wf′
have (∀ x y. x ≼ y in stack e″ ∧ x ≠ y ⟶
         (∀ u ∈ 𝒮 e″ x. ¬ reachable-avoiding u y (unvisited e″ x)))
  by (force simp: e″-def wf-env-def unvisited-def)
```

ultimately show *?thesis*
unfolding *wf-env-def* **by** *blast*
qed

moreover
from *3* **have** $v \in visited\ e''$
by (*auto simp*: *post-dfss-def sub-env-def e''-def e1-def*)

moreover
have *subenv*: $sub\text{-}env\ e\ e''$
proof −
have *sub-env e e1*
by (*auto simp*: *sub-env-def e1-def*)
with *3* **have** $sub\text{-}env\ e\ e'$
by (*auto simp*: *post-dfss-def elim*: *sub-env-trans*)
thus *?thesis*
by (*auto simp add*: *sub-env-def e''-def*)
qed

moreover
from *3* **have** $vsuccs\ e''\ v = successors\ v$
by (*simp add*: *post-dfss-def e''-def*)

moreover
from *1 3* **have** $\forall w \in visited\ e.\ vsuccs\ e''\ w = vsuccs\ e\ w$
by (*auto simp*: *pre-dfs-def post-dfss-def e1-def e''-def*)

moreover
from *3* **have** $\forall n \in set\ (stack\ e'').\ reachable\ n\ v$
by (*auto simp*: *e''-def post-dfss-def*)

moreover
from *3* ‹$v \neq hd\ (stack\ e')$›
have $\exists ns.\ stack\ e = ns\ @\ (stack\ e'')$
apply (*simp add*: *post-dfss-def e''-def e1-def*)
by (*metis append-Nil list.sel*(*1*) *list.sel*(*3*) *tl-append2*)

moreover
from *3*
have $stack\ e'' \neq []$ $v \in \mathcal{S}\ e''\ (hd\ (stack\ e''))$
$\forall n \in set\ (tl\ (stack\ e'')).\ \mathcal{S}\ e''\ n = \mathcal{S}\ e\ n$
by (*auto simp*: *post-dfss-def e1-def e''-def*)

moreover
from *cst'* **have** $cstack\ e'' = cstack\ e$
by (*simp add*: *e''-def*)

ultimately show *?thesis* **unfolding** *post-dfs-def*

```
      by blast
    qed
  qed
```

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-dfss-dom (Inl(v,e)); pre-dfs v e⟧ ⟹ post-dfs v e (dfs v e)
  ⟦dfs-dfss-dom (Inr(v,e)); pre-dfss v e⟧ ⟹ post-dfss v e (dfss v e)
proof (induct rule: dfs-dfss.pinduct)
  fix v e
  assume dom: dfs-dfss-dom (Inl(v,e))
    and predfs: pre-dfs v e
    and prepostdfss: ⋀e1. ⟦ e1 = e (|visited := visited e ∪ {v}, stack := v # stack
e,
                              cstack := v # cstack e|); pre-dfss v e1 ⟧
                         ⟹ post-dfss v e1 (dfss v e1)
  then show post-dfs v e (dfs v e)
    using pre-dfs-implies-post-dfs pre-dfs-pre-dfss by auto
next
  fix v e
  assume dom: dfs-dfss-dom (Inr(v,e))
    and predfss: pre-dfss v e
    and prepostdfs:
        ⋀vs w.
          ⟦ vs = successors v − vsuccs e v; vs ≠ {}; w = (SOME x. x ∈ vs);
            w ∉ explored e; w ∉ visited e; pre-dfs w e ⟧
          ⟹ post-dfs w e (dfs w e)
    and prepostdfss:
        ⋀vs w e' e''.
          ⟦ vs = successors v − vsuccs e v; vs ≠ {}; 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|) ;
            pre-dfss v e'' ⟧
          ⟹ post-dfss v e'' (dfss v e'')
  show post-dfss v e (dfss v e)
  proof −
    let ?vs = successors v − vsuccs e v
    from predfss have wf: wf-env e
      by (simp add: pre-dfss-def)
    from predfss have v ∈ visited e
```

```
  by (simp add: pre-dfss-def)
from predfss have v ∉ explored e
  by (meson DiffD2 list.set-sel(1) pre-dfss-def stack-class)

show ?thesis
proof (cases ?vs = {})
  case True
  with dom have dfss v e = e
    by (simp add: dfss.psimps)
  moreover
  from True wf have vsuccs e v = successors v
    unfolding wf-env-def
    by (meson Diff-eq-empty-iff le-infE subset-antisym)
  moreover
  have sub-env e e
    by (simp add: sub-env-def)
  moreover
  from predfss ‹vsuccs e v = successors v›
  have ∀ w ∈ successors v. w ∈ explored e ∪ 𝒮 e (hd (stack e))
       ∀ n ∈ set (stack e). reachable n v
       stack e ≠ []
       v ∈ 𝒮 e (hd (stack e))
    by (auto simp: pre-dfss-def)
  moreover have ∃ ns. stack e = ns @ (stack e)
    by simp
  moreover
  {
    fix n
    assume asm: hd (stack e) = v
                n ∈ set (tl (stack e))
                reachable v n
    with ‹stack e ≠ []› have v ≼ n in stack e
      by (metis head-precedes hd-Cons-tl list.set-sel(2))
    moreover
    from wf ‹stack e ≠ []› asm have v ≠ n
      unfolding wf-env-def
      by (metis distinct.simps(2) list.exhaust-sel)
    moreover
    from wf have v ∈ 𝒮 e v
      by (rule S-reflexive)
    moreover
    {
      fix a b
      assume a ∈ 𝒮 e v b ∈ successors a − vsuccs e a
      with ‹vsuccs e v = successors v› have a ≠ v
        by auto
      from ‹stack e ≠ []› ‹hd (stack e) = v›
      have v ∈ set (stack e)
        by auto
```

```
      with ‹a ≠ v› ‹a ∈ 𝒮 e v› wf have a ∈ visited e
        unfolding wf-env-def by (metis singletonD)
      have False
      proof (cases a ∈ set (cstack e))
        case True
        with ‹v ∈ set (stack e)› ‹a ∈ 𝒮 e v› ‹wf-env e›
        have a ≼ v in cstack e
          by (auto simp: wf-env-def)
        moreover
        from predfss obtain ns where cstack e = v # ns
          by (auto simp: pre-dfss-def)
        moreover
        from wf have distinct (cstack e)
          by (simp add: wf-env-def)
        ultimately have a = v
          using tail-not-precedes by force
        with ‹a ≠ v› show ?thesis ..
      next
        case False
        with ‹a ∈ visited e› wf have vsuccs e a = successors a
          by (auto simp: wf-env-def)
        with ‹b ∈ successors a − vsuccs e a› show ?thesis
          by simp
      qed
    }
    hence unvisited e v = {}
      by (auto simp: unvisited-def)

    ultimately have ¬ reachable-avoiding v n {}
      using wf unfolding wf-env-def by metis
    with ‹reachable v n› have False
      by (simp add: ra-empty)
  }
  ultimately show ?thesis
    using wf by (auto simp: post-dfss-def)
next
  case vs-case: False
  define w where w = (SOME x. x ∈ ?vs)
  define e′ where e′ = (if w ∈ explored e then e
                        else if w ∉ visited e then dfs w e
                        else unite v w e)
  define e″ where e″ = (e′⦇vsuccs := λx. if x=v then vsuccs e′ v ∪ {w} else
vsuccs e′ x⦈)

  from dom vs-case have dfss: dfss v e = dfss v e″
    apply (simp add: dfss.psimps e″-def)
    using e′-def w-def by auto

  from vs-case have wvs: w ∈ ?vs
```

```
  unfolding w-def by (metis some-in-eq)
show ?thesis
proof (cases w ∈ explored e)
  case True
  hence e': e' = e
    by (simp add: e'-def)
  with predfss wvs True
  have pre-dfss v e''
    by (auto simp: e''-def pre-dfss-explored-pre-dfss)
  with prepostdfss vs-case
  have post'': post-dfss v e'' (dfss v e'')
    by (auto simp: w-def e'-def e''-def)

  moreover
  from post''
  have ∀ u ∈ visited e − {v}. vsuccs (dfss v e'') u = vsuccs e u
    by (auto simp: post-dfss-def e' e''-def)

  moreover
  have sub-env e e''
    by (auto simp: sub-env-def e' e''-def)
  with post'' have sub-env e (dfss v e'')
    by (auto simp: post-dfss-def elim: sub-env-trans)

  moreover
  from e' have stack e'' = stack e 𝒮 e'' = 𝒮 e
    by (auto simp add: e''-def)

  moreover
  have cstack e'' = cstack e
    by (simp add: e''-def e')

  ultimately show ?thesis
    by (auto simp: dfss post-dfss-def)
next
  case notexplored: False
  then show ?thesis
  proof (cases w ∉ visited e)
    case True
    with e'-def notexplored have e' = dfs w e
      by auto
    with True notexplored pre-dfss-pre-dfs predfss
        prepostdfs vs-case w-def
    have postdfsw: post-dfs w e e'
      by (metis DiffD1 some-in-eq)
    with predfss wvs True ‹e' = dfs w e›
    have pre-dfss v e''
      by (auto simp: e''-def pre-dfss-post-dfs-pre-dfss)
    with prepostdfss vs-case
```

```
have post'': post-dfss v e'' (dfss v e'')
  by (auto simp: w-def e'-def e''-def)

moreover
have ∀ u ∈ visited e − {v}. vsuccs (dfss v e'') u = vsuccs e u
proof
  fix u
  assume u ∈ visited e − {v}
  with postdfsw
  have u: vsuccs e' u = vsuccs e u u ∈ visited e'' − {v}
    by (auto simp: post-dfs-def sub-env-def e''-def)
  with post'' have vsuccs (dfss v e'') u = vsuccs e'' u
    by (auto simp: post-dfss-def)
  with u show vsuccs (dfss v e'') u = vsuccs e u
    by (simp add: e''-def)
qed

moreover
have sub-env e (dfss v e'')
proof −
  from postdfsw have sub-env e e'
    by (simp add: post-dfs-def)
  moreover
  have sub-env e' e''
    by (auto simp: sub-env-def e''-def)
  moreover
  from post'' have sub-env e'' (dfss v e'')
    by (simp add: post-dfss-def)
  ultimately show ?thesis
    by (metis sub-env-trans)
qed

moreover
from postdfsw post''
have ∃ ns. stack e = ns @ (stack (dfss v e''))
  by (auto simp: post-dfs-def post-dfss-def e''-def)

moreover
{
  fix n
  assume n: n ∈ set (tl (stack (dfss v e'')))
  with post'' have 𝒮 (dfss v e'') n = 𝒮 e' n
    by (simp add: post-dfss-def e''-def)
  moreover
  from ‹pre-dfss v e''› n post''
  have stack e' ≠ [] ∧ n ∈ set (tl (stack e''))
    apply (simp add: pre-dfss-def post-dfss-def e''-def)
      by (metis (no-types, lifting) Un-iff list.set-sel(2) self-append-conv2
set-append tl-append2)
```

```
    with postdfsw have 𝒮 e′ n = 𝒮 e n
      apply (simp add: post-dfs-def e″-def)
      by (metis list.set-sel(2))
    ultimately have 𝒮 (dfss v e″) n = 𝒮 e n
      by simp
  }

  moreover
  from postdfsw have cstack e″ = cstack e
    by (auto simp: post-dfs-def e″-def)

  ultimately show ?thesis
    by (auto simp: dfss post-dfss-def)

next
  case False
  hence e′: e′ = unite v w e using notexplored
    using e′-def by simp
  from False have w ∈ visited e
    by simp
  from wf wvs notexplored False obtain pfx where
    pfx: stack e = pfx @ (stack e′) stack e′ ≠ []
    unfolding e′ by (blast dest: unite-stack)

  from predfss wvs notexplored False ‹e′ = unite v w e›
  have pre-dfss v e″
    by (auto simp: e″-def pre-dfss-unite-pre-dfss)

  with prepostdfss vs-case ‹e′ = unite v w e›  ‹w ∉ explored e› ‹w ∈ visited
e›
  have post″: post-dfss v e″ (dfss v e″)
    by (auto simp: w-def e″-def)

  moreover
  from post″
  have ∀ u ∈ visited e − {v}. vsuccs (dfss v e″) u = vsuccs e u
    by (auto simp: post-dfss-def e″-def e′ unite-def)

  moreover
  have sub-env e (dfss v e″)
  proof −
    from predfss wvs ‹w ∈ visited e› notexplored
    have sub-env e e′
      unfolding e′ by (blast dest: unite-sub-env)
    moreover
    have sub-env e′ e″
      by (auto simp: sub-env-def e″-def)
    moreover
    from post″ have sub-env e″ (dfss v e″)
```

by (*simp add*: *post-dfss-def*)
ultimately show *?thesis*
by (*metis sub-env-trans*)
qed

moreover
from *post″* ‹*stack e* = *pfx* @ *stack e′*›
have ∃ *ns*. *stack e* = *ns* @ (*stack* (*dfss v e″*))
by (*auto simp*: *post-dfss-def e″-def*)

moreover
{
fix *n*
assume *n*: *n* ∈ *set* (*tl* (*stack* (*dfss v e″*)))
with *post″* **have** 𝒮 (*dfss v e″*) *n* = 𝒮 *e″ n*
by (*simp add*: *post-dfss-def*)
moreover
from *n post″* ‹*stack e′* ≠ []›
have *n* ∈ *set* (*tl* (*stack e″*))
apply (*simp add*: *post-dfss-def e″-def*)
by (*metis* (*no-types*, *lifting*) *Un-iff list.set-sel*(*2*) *self-append-conv2* *set-append tl-append2*)
with *wf wvs* ‹*w* ∈ *visited e*› *notexplored*
have 𝒮 *e″ n* = 𝒮 *e n*
by (*auto simp*: *e″-def e′ dest*: *unite-S-tl*)
ultimately have 𝒮 (*dfss v e″*) *n* = 𝒮 *e n*
by *simp*
}

moreover
from *post″* **have** *cstack* (*dfss v e″*) = *cstack e*
by (*simp add*: *post-dfss-def e″-def e′ unite-def*)

ultimately show *?thesis*
by (*simp add*: *dfss post-dfss-def*)
qed
qed
qed
qed
qed

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 −
  from assms init-env-pre-dfs[of v]
  have post: post-dfs v (init-env v) e
    by (auto dest: pre-post)
  hence wf: wf-env e
    by (simp add: post-dfs-def)
  from post have cstack e = []
    by (auto simp: post-dfs-def init-env-def)
  have stack e = []
  proof (rule ccontr)
    assume stack e ≠ []
    hence hd (stack e) ≼ hd (stack e) in stack e
      by simp
    with wf ‹cstack e = []› show False
      unfolding wf-env-def
      by (metis empty-iff empty-set precedes-mem(2))
  qed
  with post have vexp: v ∈ explored e
    by (simp add: post-dfs-def)
  from wf ‹stack e = []› have explored e = visited e
    by (auto simp: wf-env-def)
  have sccs e ⊆ ?rhs
  proof
    fix S
    assume S: S ∈ sccs e
    with wf have is-scc S
      by (simp add: wf-env-def)
    moreover
    from S wf have S ⊆ explored e
      unfolding wf-env-def
      by blast
    with post ‹explored e = visited e› have ∀ n∈S. reachable v n
      by (auto simp: post-dfs-def wf-env-def sub-env-def init-env-def)
    ultimately show S ∈ ?rhs
      by auto
  qed
  moreover
  {
    fix S
    assume is-scc S ∀ n∈S. reachable v n
    from ‹∀ n∈S. reachable v n› vexp wf
    have S ⊆ ⋃ (sccs e)
      unfolding wf-env-def by (metis subset-eq)
    with ‹is-scc S› obtain S' where S': S' ∈ sccs e ∧ S ∩ S' ≠ {}
      unfolding is-scc-def
      by (metis Union-disjoint inf.absorb-iff2 inf-commute)
    with wf have is-scc S'
```

by (*simp add*: *wf-env-def*)
with S' ‹*is-scc S*› **have** $S \in$ *sccs e*
by (*auto dest*: *scc-partition*)
}
ultimately show *?thesis* **by** *blast*
qed

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*.

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**

[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.

```
  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 −
  let ?r = (finite-psubset :: ('v set × 'v set) set)
           <*lex*> (finite-psubset :: ((('v × 'v) set) × ('v × 'v) set) set)
           <*lex*> pred-nat
  have wf (finite-psubset :: ('v set × 'v set) set)
    by (rule wf-finite-psubset)
  moreover
  have wf (finite-psubset :: ((('v × 'v) set) × ('v × 'v) set) set)
    by (rule wf-finite-psubset)
  ultimately have wf ?r
    using wf-pred-nat by blast
  moreover
  have dfs-dfss-term ⊆ inv-image ?r dfs-dfss-to-tuple
  proof (clarify)
    fix a b
    assume (a,b) ∈ dfs-dfss-term
    hence (∃ v w e e''. a = Inr(v,e'') ∧ b = Inr(v,e) ∧ v ∈ vertices ∧ sub-env e e''
                  ∧ w ∈ vertices ∧ w ∉ vsuccs e v ∧ w ∈ vsuccs e'' v)
        ∨ (∃ v e e1. a = Inr(v,e1) ∧ b = Inl(v,e) ∧ v ∈ vertices − visited e
              ∧ visited e1 = visited e ∪ {v})
        ∨ (∃ v w e. a = Inl(w,e) ∧ b = Inr(v,e))
        (is ?c1 ∨ ?c2 ∨ ?c3)
      by (auto simp: dfs-dfss-term-def)
    then show (a,b) ∈ inv-image ?r dfs-dfss-to-tuple
    proof
      assume ?c1
      then obtain v w e e'' where
        ab: a = Inr(v, e'') b = Inr(v,e) and
        vw: v ∈ vertices w ∈ vertices w ∈ vsuccs e'' v w ∉ vsuccs e v and
        sub: sub-env e e''
        by blast
      from sub have vertices − visited e'' ⊆ vertices − visited e
        by (auto simp: sub-env-def)
      moreover
      from sub vw
      have (vertices × vertices − {(u,u') | u u'. u' ∈ vsuccs e'' u})
          ⊂ (vertices × vertices − {(u,u') | u u'. u' ∈ vsuccs e u})
        by (auto simp: sub-env-def)
```

```
      ultimately show ?thesis
        using vfin ab by auto
    next
      assume ?c2 ∨ ?c3
      with vfin show ?thesis
        by (auto simp: pred-nat-def)
    qed
  qed
  ultimately show ?thesis
    using wf-inv-image wf-subset by blast
qed
```

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 −
  { fix args
    have (case args
          of Inl(v,e) ⇒
            v ∈ vertices ∧ pre-dfs v e
          | Inr(v,e) ⇒
             v ∈ vertices ∧ pre-dfss v e)
        ⟶ dfs-dfss-dom args (is ?P args ⟶ ?Q args)
    proof (rule wf-induct[OF wf-term])
      fix arg :: ('v × 'v env) + ('v × 'v env)
      assume ih: ∀ arg'. (arg', arg) ∈ dfs-dfss-term ⟶ (?P arg' ⟶ ?Q arg')
      show ?P arg ⟶ ?Q arg
      proof
        assume P: ?P arg
        show ?Q arg
        proof (cases arg)
          case (Inl a)
          then obtain v e where a: arg = Inl(v, e)
            using dfs.cases by metis
          with P have pre: v ∈ vertices ∧ pre-dfs v e
            by simp
          let ?e1 = e⦇visited := visited e ∪ {v}, stack := v # stack e, cstack := v
# cstack e⦈
          let ?recarg = Inr(v, ?e1)
```

```
  from a pre
  have (?recarg, arg) ∈ dfs-dfss-term
    by (auto simp: pre-dfs-def dfs-dfss-term-def)
  moreover
  from pre have ?P ?recarg
    by (auto dest: pre-dfs-pre-dfss)
  ultimately have ?Q ?recarg
    using ih a by auto
  then have ?Q (Inl(v, e))
    by (auto intro: dfs-dfss.domintros)
  then show ?thesis
    by (simp add: a)
next
  case (Inr b)
  then obtain v e where b: arg = Inr(v, e)
    using dfs.cases by metis
  with P have pre: v ∈ vertices ∧ pre-dfss v e
    by simp
  let ?sw = SOME w. w ∈ successors v ∧ w ∉ vsuccs e v
  have ?Q (Inr(v, e))
  proof (rule dfs-dfss.domintros)
    fix w
    assume w ∈ successors v
           ?sw ∉ explored e
           ?sw ∉ visited e
           ¬ dfs-dfss-dom (Inl (?sw, e))
    show w ∈ vsuccs e v
    proof (rule ccontr)
      assume w ∉ vsuccs e v
      with ‹w ∈ successors v› have sw: ?sw ∈ successors v − vsuccs e v
        by (metis (mono-tags, lifting) Diff-iff some-eq-imp)
      with pre ‹?sw ∉ visited e› have pre-dfs ?sw e
        by (blast intro: pre-dfss-pre-dfs)
      moreover
      from pre sw sclosed have ?sw ∈ vertices
        by blast
      moreover
      from pre have (Inl(?sw,e), Inr(v,e)) ∈ dfs-dfss-term
        by (simp add: dfs-dfss-term-def)
      ultimately have dfs-dfss-dom (Inl(?sw,e))
        using ih b by auto
      with ‹¬ dfs-dfss-dom (Inl (?sw, e))›
      show False ..
    qed
  next
    let ?e′ = dfs ?sw e
    let ?e″= ?e′⦇vsuccs := λx. if x = v then vsuccs ?e′ v ∪ {?sw}
                               else vsuccs ?e′ x⦈
    fix w
```

assume *asm*: $w \in$ *successors* v $w \notin$ *vsuccs* e v
?sw $\notin$ *visited* e *?sw* $\notin$ *explored* e
from ‹$w \in$ *successors* v› ‹$w \notin$ *vsuccs* e v›
have *sw*: *?sw* $\in$ *successors* v $-$ *vsuccs* e v
by (*metis* (*no-types*, *lifting*) *Diff-iff some-eq-imp*)
with *pre* ‹*?sw* $\notin$ *visited* e› **have** *pre-dfs ?sw e*
by (*blast intro*: *pre-dfss-pre-dfs*)
moreover
from *pre sw sclosed* **have** *?sw* $\in$ *vertices*
by *blast*
moreover
from *pre* **have** (*Inl*(*?sw*, *e*), *Inr*(*v*,*e*)) $\in$ *dfs-dfss-term*
by (*simp add*: *dfs-dfss-term-def*)
ultimately have *dfs-dfss-dom* (*Inl*(*?sw*, *e*))
using *ih b* **by** *auto*
from *this* ‹*pre-dfs ?sw e*› **have** *post*: *post-dfs ?sw e ?e′*
by (*rule pre-post*)
hence *sub-env e ?e′*
by (*simp add*: *post-dfs-def*)
moreover
have *sub-env ?e′ ?e″*
by (*auto simp*: *sub-env-def*)
ultimately have *sub-env e ?e″*
by (*rule sub-env-trans*)
with *pre* ‹*?sw* $\in$ *vertices*› *sw*
have (*Inr*(*v*, *?e″*), *Inr*(*v*, *e*)) $\in$ *dfs-dfss-term*
by (*auto simp*: *dfs-dfss-term-def*)
moreover
from *pre post sw* ‹*?sw* $\notin$ *visited* e› **have** *pre-dfss v ?e″*
by (*blast intro*: *pre-dfss-post-dfs-pre-dfss*)
ultimately show *dfs-dfss-dom*(*Inr*(*v*, *?e″*))
using *pre ih b* **by** *auto*
next
let *?e″* = e⦇*vsuccs* := λx. *if* $x = v$ *then* *vsuccs* e v $\cup$ {*?sw*} *else vsuccs*
e x⦈
fix w
assume $w \in$ *successors* v $w \notin$ *vsuccs* e v
?sw $\notin$ *visited* e *?sw* $\in$ *explored* e
with *pre* **have** *False*
unfolding *pre-dfss-def wf-env-def*
by (*meson subsetD*)
thus *?Q* (*Inr*(*v*, *?e″*))
by *simp*
next
fix w
assume *asm*: $w \in$ *successors* v $w \notin$ *vsuccs* e v
?sw $\in$ *visited* e *?sw* $\in$ *explored* e
let *?e″* = e⦇*vsuccs* := λx. *if* $x = v$ *then* *vsuccs* e v $\cup$ {*?sw*} *else vsuccs*
e x⦈

```
let ?recarg = Inr(v, ?e'')

from ‹w ∈ successors v› ‹w ∉ vsuccs e v›
have sw: ?sw ∈ successors v − vsuccs e v
  by (metis (no-types, lifting) Diff-iff some-eq-imp)

have (?recarg, arg) ∈ dfs-dfss-term
proof −
  have sub-env e ?e''
    by (auto simp: sub-env-def)
  moreover
  from sw pre sclosed
  have ∃ u ∈ vertices. u ∉ vsuccs e v ∧ u ∈ vsuccs ?e'' v
    by auto
  ultimately show ?thesis
    using pre b unfolding dfs-dfss-term-def by blast
qed

moreover
from pre sw ‹?sw ∈ explored e› have ?P ?recarg
  by (auto dest: pre-dfss-explored-pre-dfss)

ultimately show ?Q ?recarg
  using ih b by blast
next
fix w
assume asm: w ∈ successors v w ∉ vsuccs e v
            ?sw ∈ visited e ?sw ∉ explored e
let ?eu = unite v ?sw e
 let ?e'' = ?eu⦇vsuccs := λx. if x = v then vsuccs ?eu v ∪ {?sw} else
vsuccs ?eu x⦈
let ?recarg = Inr(v, ?e'')

from ‹w ∈ successors v› ‹w ∉ vsuccs e v›
have sw: ?sw ∈ successors v − vsuccs e v
  by (metis (no-types, lifting) Diff-iff some-eq-imp)

have (?recarg, arg) ∈ dfs-dfss-term
proof −
  from pre asm sw have sub-env e ?eu
    by (blast dest: unite-sub-env)
  hence sub-env e ?e''
    by (auto simp: sub-env-def)
  moreover
  from sw pre sclosed
  have ∃ u ∈ vertices. u ∉ vsuccs e v ∧ u ∈ vsuccs ?e'' v
    by auto
  ultimately show ?thesis
    using pre b unfolding dfs-dfss-term-def by blast
```

qed

moreover
from *pre sw* ‹*?sw* ∈ *visited e*› ‹*?sw* ∉ *explored e*› **have** *?P ?recarg*
by (*auto dest*: *pre-dfss-unite-pre-dfss*)

ultimately show *?Q ?recarg*
using *ih b* **by** *auto*
qed
then show *?thesis*
by (*simp add*: *b*)
qed
qed
qed
}
note *dom=this*
from *dom*
show ⟦ *v* ∈ *vertices* ; *pre-dfs v e*⟧ ⟹ *dfs-dfss-dom*(*Inl*(*v*, *e*))
by *auto*
from *dom*
show ⟦ *v* ∈ *vertices* ; *pre-dfss v e*⟧ ⟹ *dfs-dfss-dom*(*Inr*(*v*, *e*))
by *auto*
qed

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*)}
using *assms init-env-pre-dfs*[*of v*]
by (*simp add*: *dfs-dfss-termination partial-correctness*)

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.