Correctness of Tarjan's Algorithm

Stephan Merz

October 17, 2018

Contents

1 Reachability in graphs **2**

2 Strongly connected components **3**

3 Auxiliary functions **3**

4 Main functions used for Tarjan's algorithms **5**
4.1 Function definitions 5
4.2 Well-definedness of the functions 5

5 Auxiliary notions for the proof of partial correctness **11**

6 Predicates and lemmas about environments **14**

7 Partial correctness of the main functions **17**

8 Theorems establishing total correctness **34**

theory *Tarjan*
imports *Main*
begin

Tarjan's algorithm computes the strongly connected components of a finite graph using depth-first search. We formalize a functional version of the algorithm in Isabelle/HOL, following a development of Lvy et al. in Why3 that is available at http://pauillac.inria.fr/~levy/why3/graph/abs/scct/1-68bis/scc.html.

Make the simplifier expand let-constructions automatically

declare *Let-def* [*simp*]

Definition of an auxiliary data structure holding local variables during the execution of Tarjan's algorithm.

record *'v env* =
  black :: *'v set*

gray :: *'v set*
stack :: *'v list*
sccs :: *'v set set*
sn :: *nat*
num :: *'v ⇒ int*

definition *colored* **where**
colored e ≡ black e ∪ gray e

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

1 Reachability in graphs

abbreviation *edge* **where**
edge x y ≡ y ∈ successors x

definition *xedge-to* **where**
— *ys* is a suffix of *xs*, *y* appears in *ys*, and there is an edge from some node in the prefix of *xs* to *y*
xedge-to xs ys y ≡
y ∈ set ys
∧ (∃ zs. xs = zs @ ys ∧ (∃ z ∈ set zs. edge z y))

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*

Given some set S and two vertices x and y such that y is reachable from x, and x is an element of S but y is not, then there exists some vertices x' and

y' linked by an edge such that x' is an element of S, y' is not, x' is reachable from x, and y is reachable from y'.

lemma *reachable-crossing-set*:
  assumes *1*: *reachable* x y **and** *2*: $x \in S$ **and** *3*: $y \notin S$
  obtains x' y' **where**
    $x' \in S$ $y' \notin S$ *edge* x' y' *reachable* x x' *reachable* y' y
proof −
  from *assms*
  have $\exists x'\ y'.\ x' \in S \land y' \notin S \land$ *edge* x' y' $\land$ *reachable* x x' $\land$ *reachable* y' y
    by *induct* (*blast intro*: *reachable-edge reachable-trans*)+
  with *that* **show** *?thesis* **by** *blast*
qed

2 Strongly connected components

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 $S \equiv S \neq \{\} \land$ *is-subscc* $S \land (\forall S'.\ S \subseteq S' \land$ *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 vertices 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*)

3 Auxiliary functions

abbreviation *infty* (∞) **where**
  — integer exceeding any one used as a vertex number during the algorithm
  $\infty \equiv$ *int* (*card vertices*)

definition *set-infty* **where**

— set $f\ x$ to ∞ for all x in xs
set-infty xs f = *fold* ($\lambda x\ g.\ g\ (x := \infty)$) *xs f*

lemma *set-infty*:
(*set-infty xs f*) x = (*if* $x \in$ *set xs then* ∞ *else f x*)
unfolding *set-infty-def* **by** (*induct xs arbitrary*: *f*) *auto*

Split a list at the first occurrence of a given element. Returns the two sublists of elements before (and including) the element and those strictly after the element. If the element does not occur in the list, returns a pair formed by the entire list and the empty list.

fun *split-list* **where**
split-list x [] = ([], [])
| *split-list x* (*y* # *xs*) =
(*if x* = *y then* ([*x*], *xs*) *else*
(*let* (*l*, *r*) = *split-list x xs in*
(*y* # *l*, *r*)))

lemma *split-list-concat*:
— Concatenating the two sublists produced by *split-list* yields back the original list.
assumes $x \in$ *set xs*
shows (*fst* (*split-list x xs*)) @ (*snd* (*split-list x xs*)) = *xs*
using *assms* **by** (*induct xs*) (*auto simp*: *split-def*)

lemma *fst-split-list*:
assumes $x \in$ *set xs*
shows $\exists$ *ys*. *fst* (*split-list x xs*) = *ys* @ [*x*] $\wedge$ $x \notin$ *set ys*
using *assms* **by** (*induct xs*) (*auto simp*: *split-def*)

Push a vertex on the stack and increment the sequence number. The pushed vertex is associated with the (old) sequence number. It is also added to the set of gray nodes.

definition *add-stack-incr* **where**
add-stack-incr x e =
e ⦇ *gray* := *insert x* (*gray e*),
stack := *x* # (*stack e*),
sn := *sn e* +*1*,
num := (*num e*) (*x* := *int* (*sn e*)) ⦈

Add vertex x to the set of black vertices in e and remove it from the set of gray vertices.

definition *add-black* **where**
add-black x e = *e* ⦇ *black* := *insert x* (*black e*),
gray := (*gray e*) − {*x*} ⦈

4 Main functions used for Tarjan's algorithms

4.1 Function definitions

We define two mutually recursive functions that contain the essence of Tarjan's algorithm. Their arguments are respectively a single vertex and a set of vertices, as well as an environment that contains the local variables of the algorithm, and an auxiliary parameter representing the set of "gray" vertices, which is used only for the proof. The main function is then obtained by specializing the function operating on a set of vertices.

```
function (domintros) dfs1 and dfs where
  dfs1 x e  =
   (let (n1, e1) = dfs (successors x) (add-stack-incr x e) in
     if n1 < int (sn e) then (n1, add-black x e1)
     else
      (let (l,r) = split-list x (stack e1) in
        (∞,
          (| black = insert x (black e1),
            gray = gray e,
            stack = r,
            sccs = insert (set l) (sccs e1),
            sn = sn e1,
            num = set-infty l (num e1) |) )))
| dfs roots e =
   (if roots = {} then (∞, e)
   else
     (let x = SOME x. x ∈ roots;
          res1 = (if num e x ≠ −1 then (num e x, e) else dfs1 x e);
          res2 = dfs (roots − {x}) (snd res1)
     in (min (fst res1) (fst res2), snd res2) ))
  by pat-completeness auto
```

```
definition init-env where
  init-env ≡ (| black = {},           gray = {},
                stack = [],          sccs = {},
                sn = 0,              num = λ-. −1 |)
```

```
definition tarjan where
  tarjan ≡ sccs (snd (dfs vertices init-env))
```

4.2 Well-definedness of the functions

We did not prove termination when we defined the two mutually recursive functions *dfs1* and *dfs* defined above, and indeed it is easy to see that they do not terminate for arbitrary arguments. Isabelle allows us to define "partial" recursive functions, for which it introduces an auxiliary domain predicate that characterizes their domain of definition. We now make this more concrete and prove that the two functions terminate when called for nodes of the graph, also assuming an elementary well-definedness condition for environments. These conditions are met in the cases of interest, and in particular in the call to *dfs* in the main function *tarjan*. Intuitively, the reason is that every (possibly indirect) recursive call to *dfs* either decreases the set of roots or increases the set of nodes colored black or gray.

The set of nodes colored black never decreases in the course of the computation.

```
lemma black-increasing:
  dfs1-dfs-dom (Inl (x,e)) ⟹ black e ⊆ black (snd (dfs1 x e))
  dfs1-dfs-dom (Inr (roots,e)) ⟹ black e ⊆ black (snd (dfs roots e))
  by (induct rule: dfs1-dfs.pinduct,
      (fastforce simp: dfs1.psimps dfs.psimps case-prod-beta
                  add-black-def add-stack-incr-def)+)
```

Similarly, the set of nodes colored black or gray never decreases in the course of the computation.

```
lemma colored-increasing:
  dfs1-dfs-dom (Inl (x,e)) ⟹
    colored e ⊆ colored (snd (dfs1 x e)) ∧
    colored (add-stack-incr x e)
    ⊆ colored (snd (dfs (successors x) (add-stack-incr x e)))
  dfs1-dfs-dom (Inr (roots,e)) ⟹
    colored e ⊆ colored (snd (dfs roots e))
proof (induct rule: dfs1-dfs.pinduct)
  case (1 x e)
  from ‹dfs1-dfs-dom (Inl (x,e))›
  have black e ⊆ black (snd (dfs1 x e))
    by (rule black-increasing)
  with 1 show ?case
    by (auto simp: dfs1.psimps case-prod-beta add-stack-incr-def
                add-black-def colored-def)
next
  case (2 roots e) then show ?case
    by (fastforce simp: dfs.psimps case-prod-beta)
qed
```

The functions *dfs1* and *dfs* never assign the number of a vertex to -1.

```
lemma dfs-num-defined:
  ⟦dfs1-dfs-dom (Inl (x,e)); num (snd (dfs1 x e)) v = −1⟧ ⟹
    num e v = −1
  ⟦dfs1-dfs-dom (Inr (roots,e)); num (snd (dfs roots e)) v = −1⟧ ⟹
    num e v = −1
  by (induct rule: dfs1-dfs.pinduct,
      (auto simp: dfs1.psimps dfs.psimps case-prod-beta add-stack-incr-def
                add-black-def set-infty
           split: if-split-asm))
```

We are only interested in environments that assign positive numbers to colored nodes, and we show that calls to *dfs1* and *dfs* preserve this property.

```
definition colored-num where
  colored-num e ≡ ∀ v ∈ colored e. v ∈ vertices ∧ num e v ≠ −1

lemma colored-num:
  ⟦dfs1-dfs-dom (Inl (x,e)); x ∈ vertices; colored-num e⟧ ⟹
    colored-num (snd (dfs1 x e))
  ⟦dfs1-dfs-dom (Inr (roots,e)); roots ⊆ vertices; colored-num e⟧ ⟹
    colored-num (snd (dfs roots e))
proof (induct rule: dfs1-dfs.pinduct)
  case (1 x e)
  let ?rec = dfs (successors x) (add-stack-incr x e)
  from sclosed ‹x ∈ vertices›
  have successors x ⊆ vertices ..
  moreover
  from ‹colored-num e› ‹x ∈ vertices›
  have colored-num (add-stack-incr x e)
    by (auto simp: colored-num-def add-stack-incr-def colored-def)
  ultimately
  have rec: colored-num (snd ?rec)
    using 1 by blast
  have x: x ∈ colored (add-stack-incr x e)
    by (simp add: add-stack-incr-def colored-def)
  from ‹dfs1-dfs-dom (Inl (x,e))› colored-increasing
  have colrec: colored (add-stack-incr x e) ⊆ colored (snd ?rec)
    by blast
  show ?case
  proof (cases fst ?rec < int (sn e))
    case True
    with rec x colrec ‹dfs1-dfs-dom (Inl (x,e))› show ?thesis
      by (auto simp: dfs1.psimps case-prod-beta
                colored-num-def add-black-def colored-def)
  next
    case False
    let ?e' = snd (dfs1 x e)
    have colored e ⊆ colored (add-stack-incr x e)
      by (auto simp: colored-def add-stack-incr-def)
    with False x colrec ‹dfs1-dfs-dom (Inl (x,e))›
    have colored ?e' ⊆ colored (snd ?rec)
        ∃ xs. num ?e' = set-infty xs (num (snd ?rec))
      by (auto simp: dfs1.psimps case-prod-beta colored-def)
    with rec show ?thesis
      by (auto simp: colored-num-def set-infty split: if-split-asm)
  qed
next
  case (2 roots e)
  show ?case
  proof (cases roots = {})
```

```
    case True
    with ‹dfs1-dfs-dom (Inr (roots,e))› ‹colored-num e›
    show ?thesis by (auto simp: dfs.psimps)
  next
    case False
    let ?x = SOME x. x ∈ roots
    from False obtain r where r ∈ roots by blast
    hence ?x ∈ roots by (rule someI)
    with ‹roots ⊆ vertices› have x: ?x ∈ vertices ..
    let ?res1 = if num e ?x ≠ −1 then (num e ?x, e) else dfs1 ?x e
    let ?res2 = dfs (roots − {?x}) (snd ?res1)
    from 2 False ‹roots ⊆ vertices› x
    have colored-num (snd ?res1) by auto
    with 2 False ‹roots ⊆ vertices›
    have colored-num (snd ?res2)
      by blast
    moreover
    from False ‹dfs1-dfs-dom (Inr (roots,e))›
    have dfs roots e = (min (fst ?res1) (fst ?res2), snd ?res2)
      by (auto simp: dfs.psimps)
    ultimately show ?thesis by simp
  qed
qed
```

The following relation underlies the termination argument used for proving well-definedness of the functions *dfs1* and *dfs*. It is defined on the disjoint sum of the types of arguments of the two functions and relates the arguments of (mutually) recursive calls.

```
definition dfs1-dfs-term where
  dfs1-dfs-term ≡
    { (Inl(x, e::'v env), Inr(roots,e)) |
      x e roots .
      roots ⊆ vertices ∧ x ∈ roots ∧ colored e ⊆ vertices }
  ∪ { (Inr(roots, add-stack-incr x e), Inl(x, e)) |
      x e roots .
      colored e ⊆ vertices ∧ x ∈ vertices − colored e }
  ∪ { (Inr(roots, e::'v env), Inr(roots', e')) |
      roots roots' e e' .
      roots' ⊆ vertices ∧ roots ⊂ roots' ∧
      colored e' ⊆ colored e ∧ colored e ⊆ vertices }
```

In order to prove that the above relation is well-founded, we use the following function that embeds it into triples whose first component is the complement of the colored nodes, whose second component is the set of root nodes, and whose third component is 1 or 2 depending on the function being called. The third component corresponds to the first case in the definition of *dfs1-dfs-term*.

```
fun dfs1-dfs-to-tuple where
```

```
  dfs1-dfs-to-tuple (Inl(x::'v, e::'v env)) = (vertices − colored e, {x}, 1::nat)
| dfs1-dfs-to-tuple (Inr(roots, e::'v env)) = (vertices − colored e, roots, 2)

lemma wf-term: wf dfs1-dfs-term
proof −
  let ?r = (finite-psubset :: ('v set × 'v set) set)
           <*lex*> (finite-psubset :: ('v set × 'v set) set)
           <*lex*> pred-nat
  have wf ?r
    using wf-finite-psubset wf-pred-nat by blast
  moreover
  have dfs1-dfs-term ⊆ inv-image ?r dfs1-dfs-to-tuple
    unfolding dfs1-dfs-term-def pred-nat-def using vfin
    by (auto dest: finite-subset simp: add-stack-incr-def colored-def)
  ultimately show ?thesis
    using wf-inv-image wf-subset by blast
qed
```

The following theorem establishes sufficient conditions under which the two functions *dfs1* and *dfs* terminate. The proof proceeds by well-founded induction using the relation *dfs1-dfs-term* and makes use of the theorem *dfs1-dfs.domintros* that was generated by Isabelle from the mutually recursive definitions in order to characterize the domain conditions for these functions.

```
theorem dfs1-dfs-termination:
  ⟦x ∈ vertices − colored e; colored-num e⟧ ⟹ dfs1-dfs-dom (Inl(x, e))
  ⟦roots ⊆ vertices; colored-num e⟧ ⟹ dfs1-dfs-dom (Inr(roots, e))
proof −
  { fix args
    have (case args
          of Inl(x,e) ⇒
             x ∈ vertices − colored e ∧ colored-num e
          | Inr(roots,e) ⇒
             roots ⊆ vertices ∧ colored-num e)
        ⟶ dfs1-dfs-dom args (is ?P args ⟶ ?Q args)
    proof (rule wf-induct[OF wf-term])
      fix arg :: ('v × 'v env) + ('v set × 'v env)
      assume ih: ∀ arg'. (arg',arg) ∈ dfs1-dfs-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 x e where a: arg = Inl(x,e)
            using dfs1.cases by metis
          have ?Q (Inl(x,e))
          proof (rule dfs1-dfs.domintros)
```

let *?recarg* = *Inr* (*successors x*, *add-stack-incr x e*)
from *a P* **have** (*?recarg*, *arg*) ∈ *dfs1-dfs-term*
by (*auto simp*: *add-stack-incr-def colored-num-def dfs1-dfs-term-def*)
moreover
from *a P sclosed* **have** *?P ?recarg*
by (*auto simp*: *add-stack-incr-def colored-num-def colored-def*)
ultimately show *?Q ?recarg*
using *ih* **by** *auto*
qed
with *a* **show** *?thesis* **by** *simp*
next
case (*Inr b*)
then obtain *roots e* **where** *b*: *arg* = *Inr*(*roots*,*e*)
using *dfs.cases* **by** *metis*
let *?sx* = *SOME x*. *x* ∈ *roots*
let *?rec1arg* = *Inl* (*?sx*, *e*)
let *?rec2arg* = *Inr* (*roots* − {*?sx*}, *e*)
let *?rec3arg* = *Inr* (*roots* − {*?sx*}, *snd* (*dfs1 ?sx e*))
have *?Q* (*Inr*(*roots*,*e*))
proof (*rule dfs1-dfs.domintros*)
fix *x*
assume *1*: *x* ∈ *roots*
and *2*: *num e ?sx* = −*1*
and *3*: ¬ *dfs1-dfs-dom ?rec1arg*
from *1* **have** *sx*: *?sx* ∈ *roots* **by** (*rule someI*)
with *P b* **have** (*?rec1arg*, *arg*) ∈ *dfs1-dfs-term*
by (*auto simp*: *dfs1-dfs-term-def colored-num-def*)
moreover
from *sx 2 P b* **have** *?P ?rec1arg*
by (*auto simp*: *colored-num-def*)
ultimately show *False*
using *ih 3* **by** *auto*
next
fix *x*
assume *x* ∈ *roots*
hence *sx*: *?sx* ∈ *roots* **by** (*rule someI*)
from *sx b P* **have** (*?rec2arg*, *arg*) ∈ *dfs1-dfs-term*
by (*auto simp*: *dfs1-dfs-term-def colored-num-def*)
moreover
from *P b* **have** *?P ?rec2arg* **by** *auto*
ultimately show *dfs1-dfs-dom ?rec2arg*
using *ih* **by** *auto*
next
fix *x*
assume *1*: *x* ∈ *roots* **and** *2*: *num e ?sx* = −*1*
from *1* **have** *sx*: *?sx* ∈ *roots* **by** (*rule someI*)
have *dfs1-dfs-dom ?rec1arg*
proof −
from *sx P b* **have** (*?rec1arg*, *arg*) ∈ *dfs1-dfs-term*

```
          by (auto simp: dfs1-dfs-term-def colored-num-def)
        moreover
        from sx 2 P b have ?P ?rec1arg
          by (auto simp: colored-num-def)
        ultimately show ?thesis
          using ih by auto
      qed
      with P b sx have colored-num (snd (dfs1 ?sx e))
        by (auto elim: colored-num)
      moreover
      from this sx b P ‹dfs1-dfs-dom ?rec1arg›
      have (?rec3arg, arg) ∈ dfs1-dfs-term
        by (auto simp: dfs1-dfs-term-def colored-num-def
                 dest: colored-increasing)
      moreover
      from this P b ‹colored-num (snd (dfs1 ?sx e))›
      have ?P ?rec3arg by auto
      ultimately show dfs1-dfs-dom ?rec3arg
        using ih by auto
    qed
    with b show ?thesis by simp
   qed
  qed
 qed
}
note dom = this
from dom
show ⟦x ∈ vertices − colored e; colored-num e⟧ ⟹ dfs1-dfs-dom (Inl(x,e))
  by auto
from dom
show ⟦roots ⊆ vertices; colored-num e⟧ ⟹ dfs1-dfs-dom (Inr(roots,e))
  by auto
qed
```

5 Auxiliary notions for the proof of partial correctness

The proof of partial correctness is more challenging and requires some further concepts that we now define.

We need to reason about the relative order of elements in a list (specifically, the stack used in the algorithm).

```
definition precedes (- ⪯ - in - [100,100,100] 39) where
  — x has an occurrence in xs that precedes an occurrence of y.
  x ⪯ y in xs ≡ ∃ l r. xs = l @ (x # r) ∧ y ∈ set (x # r)

lemma precedes-mem:
  assumes x ⪯ y in xs
```

shows $x \in set\ xs\ y \in set\ xs$
using *assms* **unfolding** *precedes-def* **by** *auto*

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

lemma *precedes-in-tail*:
assumes $x \neq z$
shows $x \preceq y\ in\ (z\ \#\ zs) \longleftrightarrow x \preceq y\ in\ zs$
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 ‹$y \in set\ (x \# r)$› **show** *?rhs*
unfolding *precedes-def* **by** *blast*
qed
next
assume *?rhs* **thus** *?lhs* **by** (*rule precedes-append-left*)
qed

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

lemma *precedes-append-right-iff*:
assumes $y \notin set\ ys$
shows $x \preceq y\ in\ (xs @ ys) \longleftrightarrow x \preceq y\ in\ xs$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *l r* **where** *lr*: $xs @ ys = l @ (x \# r)$ $y \in set\ (x \# r)$
unfolding *precedes-def* **by** *blast*
then obtain *us* **where**
$(xs = l @ us \wedge us @ ys = x \# r) \vee (xs @ us = l \wedge ys = us @ (x \# r))$
by (*auto simp*: *append-eq-append-conv2*)
thus *?rhs*
proof
assume *us*: $xs = l @ us \wedge us @ ys = x \# r$
with ‹$y \in set\ (x \# r)$› *assms* **show** *?rhs*
unfolding *precedes-def* **by** (*metis Cons-eq-append-conv Un-iff set-append*)
next
assume *us*: $xs @ us = l \wedge ys = us @ (x \# r)$
with ‹$y \in set\ (x \# r)$› *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*::'*a*, *3*::'*a*, *1*::'*a*, *2*::*'a*]: then 1 precedes 2 and 2 precedes 3, but 1 does not precede 3.

lemma *precedes-trans*:
  assumes $x \preceq y$ *in xs* **and** $y \preceq z$ *in xs* **and** *distinct xs*
  shows $x \preceq z$ *in xs*
  using *assms* **unfolding** *precedes-def*
  by (*smt Un-iff append.assoc append-Cons-eq-iff distinct-append*
      not-distinct-conv-prefix set-append split-list-last)

lemma *precedes-antisym*:
  assumes $x \preceq y$ *in xs* **and** $y \preceq x$ *in xs* **and** *distinct xs*
  shows $x = y$
proof −
  from ‹$x \preceq y$ *in xs*› ‹*distinct xs*› **obtain** *as bs* **where**
    1: $xs = as$ @ $(x \# bs)$ $y \in set$ $(x \# bs)$ $y \notin set\ as$
    unfolding *precedes-def* **by** *force*
  from ‹$y \preceq x$ *in xs*› ‹*distinct xs*› **obtain** *cs ds* **where**
    2: $xs = cs$ @ $(y \# ds)$ $x \in set$ $(y \# ds)$ $x \notin set\ cs$
    unfolding *precedes-def* **by** *force*
  from *1 2* **have** *as* @ $(x \# bs) = cs$ @ $(y \# ds)$
    by *simp*
  then obtain *zs* **where**
    $(as = cs$ @ $zs \wedge zs$ @ $(x \# bs) = y \# ds)$
     $\vee$ $(as$ @ $zs = cs \wedge x \# bs = zs$ @ $(y \# ds))$ (**is** *?P* ∨ *?Q*)
    by (*auto simp*: *append-eq-append-conv2*)
  then show *?thesis*
  proof
    assume *?P* **with** ‹$y \notin set\ as$› **show** *?thesis*
      by (*cases zs*) *auto*
  next
    assume *?Q* **with** ‹$x \notin set\ cs$› **show** *?thesis*
      by (*cases zs*) *auto*
  qed
qed

6 Predicates and lemmas about environments

definition *subenv* **where**
  $subenv\ e\ e' \equiv$
    $(\exists s.\ stack\ e' = s$ @ $(stack\ e) \wedge set\ s \subseteq black\ e')$
  $\wedge\ black\ e \subseteq black\ e' \wedge gray\ e = gray\ e'$
  $\wedge\ sccs\ e \subseteq sccs\ e'$
  $\wedge\ (\forall x \in set\ (stack\ e).\ num\ e\ x = num\ e'\ x)$

lemma *subenv-refl* [*simp*]: *subenv e e*
  by (*auto simp*: *subenv-def*)

lemma *subenv-trans*:
  assumes *subenv e e'* **and** *subenv e' e''*

shows *subenv e e''*
using *assms* **unfolding** *subenv-def* **by** *force*

definition *wf-color* **where**
— conditions about colors, part of the invariant of the algorithm
wf-color e $\equiv$
colored e $\subseteq$ *vertices*
$\wedge$ *black e* $\cap$ *gray e* $= \{\}$
$\wedge$ ($\bigcup$ *sccs e*) $\subseteq$ *black e*
$\wedge$ *set* (*stack e*) $=$ *gray e* $\cup$ (*black e* $-$ $\bigcup$ *sccs e*)

definition *wf-num* **where**
— conditions about vertex numbers
wf-num e $\equiv$
int (*sn e*) $\leq \infty$
$\wedge$ ($\forall x$. $-1 \leq$ *num e x* $\wedge$ (*num e x* $= \infty$ $\vee$ *num e x* $<$ *int* (*sn e*)))
$\wedge$ *sn e* $=$ *card* (*colored e*)
$\wedge$ ($\forall x$. *num e x* $= \infty \longleftrightarrow x \in \bigcup$ *sccs e*)
$\wedge$ ($\forall x$. *num e x* $= -1 \longleftrightarrow x \notin$ *colored e*)
$\wedge$ ($\forall x \in$ *set* (*stack e*). $\forall y \in$ *set* (*stack e*).
(*num e x* $\leq$ *num e y* $\longleftrightarrow y \preceq x$ *in* (*stack e*)))

lemma *subenv-num*:
— If e and e' are two well-formed environments, and e is a sub-environment of e' then the number assigned by e' to any vertex is at least that assigned by e.
assumes *sub*: *subenv e e'*
and *e*: *wf-color e wf-num e*
and *e'*: *wf-color e' wf-num e'*
shows *num e x* $\leq$ *num e' x*

proof (*cases x* $\in$ *colored e*)
case *True* **then show** *?thesis* **unfolding** *colored-def*
proof
assume $x \in$ *gray e*
with *e sub* **show** *?thesis*
by (*auto simp*: *wf-color-def subenv-def*)
next
assume $x \in$ *black e*
show *?thesis*
proof (*cases x* $\in \bigcup$ *sccs e*)
case *True*
with *sub e e'* **have** *num e x* $= \infty$ *num e' x* $= \infty$
by (*auto simp*: *subenv-def wf-num-def*)
thus *?thesis* **by** *simp*
next
case *False*
with ‹$x \in$ *black e*› *e sub* **show** *?thesis*
by (*auto simp*: *wf-color-def subenv-def*)
qed

qed
next
case *False* **with** e e' **show** *?thesis*
unfolding *wf-num-def* **by** *metis*
qed

definition *no-black-to-white* **where**
— successors of black vertices cannot be white
$\textit{no-black-to-white}\ e \equiv \forall\, x\ y.\ \textit{edge}\ x\ y \wedge x \in \textit{black}\ e \longrightarrow y \in \textit{colored}\ e$

definition *wf-env* **where**
$\textit{wf-env}\ e \equiv$
$\textit{wf-color}\ e \wedge \textit{wf-num}\ e$
$\wedge\ \textit{no-black-to-white}\ e \wedge \textit{distinct}\ (\textit{stack}\ e)$
$\wedge\ (\forall\, x\ y.\ y \preceq x\ \textit{in}\ (\textit{stack}\ e) \longrightarrow \textit{reachable}\ x\ y)$
$\wedge\ (\forall\, y \in \textit{set}\ (\textit{stack}\ e).\ \exists\, g \in \textit{gray}\ e.\ y \preceq g\ \textit{in}\ (\textit{stack}\ e) \wedge \textit{reachable}\ y\ g)$
$\wedge\ \textit{sccs}\ e = \{\ C\ .\ C \subseteq \textit{black}\ e \wedge \textit{is-scc}\ C\ \}$

lemma *num-in-stack*:
assumes $\textit{wf-env}\ e$ **and** $x \in \textit{set}\ (\textit{stack}\ e)$
shows $\textit{num}\ e\ x \neq -1$
$\textit{num}\ e\ x < \textit{int}\ (\textit{sn}\ e)$
proof –
from *assms*
show $\textit{num}\ e\ x \neq -1$
by (*auto simp*: *wf-env-def wf-color-def wf-num-def colored-def*)
from ‹*wf-env e*›
have $\textit{num}\ e\ x < \textit{int}\ (\textit{sn}\ e) \vee x \in \bigcup\ \textit{sccs}\ e$
unfolding *wf-env-def wf-num-def* **by** *metis*
with *assms* **show** $\textit{num}\ e\ x < \textit{int}\ (\textit{sn}\ e)$
unfolding *wf-env-def wf-color-def* **by** *blast*
qed

Numbers assigned to different stack elements are distinct.

lemma *num-inj*:
assumes $\textit{wf-env}\ e$ **and** $x \in \textit{set}\ (\textit{stack}\ e)$
and $y \in \textit{set}\ (\textit{stack}\ e)$ **and** $\textit{num}\ e\ x = \textit{num}\ e\ y$
shows $x = y$
using *assms* **unfolding** *wf-env-def wf-num-def*
by (*metis precedes-refl precedes-antisym*)

The set of black elements at the top of the stack together with the first gray element always form a sub-SCC. This lemma is useful for the "else" branch of *dfs1*.

lemma *first-gray-yields-subscc*:
assumes *e*: $\textit{wf-env}\ e$
and *x*: $\textit{stack}\ e = \textit{ys}\ @\ (x\ \#\ \textit{zs})$
and *g*: $x \in \textit{gray}\ e$
and *ys*: $\textit{set}\ \textit{ys} \subseteq \textit{black}\ e$

shows *is-subscc* (*insert x* (*set ys*))
proof −
from *e x* **have** ∀ *y* ∈ *set ys*. ∃ *g* ∈ *gray e*. *reachable y g*
unfolding *wf-env-def* **by** *force*
moreover
have ∀ *g* ∈ *gray e*. *reachable g x*
proof
fix *g*
assume *g* ∈ *gray e*
with *e x ys* **have** *g* ∈ *set* (*x* # *zs*)
unfolding *wf-env-def wf-color-def* **by** *auto*
with *e x* **show** *reachable g x*
unfolding *wf-env-def precedes-def* **by** *blast*
qed
moreover
from *e x g* **have** ∀ *y* ∈ *set ys*. *reachable x y*
unfolding *wf-env-def* **by** (*simp add*: *split-list-precedes*)
ultimately show *?thesis*
unfolding *is-subscc-def*
by (*metis reachable-trans reachable-refl insertE*)
qed

7 Partial correctness of the main functions

We now define the pre- and post-conditions for proving that the functions *dfs1* and *dfs* are partially correct. The parameters of the preconditions, as well as the first parameters of the postconditions, coincide with the parameters of the functions *dfs1* and *dfs*. The final parameter of the postconditions represents the result computed by the function.

definition *dfs1-pre* **where**
dfs1-pre x e ≡
x ∈ *vertices*
∧ *x* ∉ *colored e*
∧ (∀ *g* ∈ *gray e*. *reachable g x*)
∧ *wf-env e*

definition *dfs1-post* **where**
dfs1-post x e res ≡
let n = *fst res*; *e′* = *snd res*
in *wf-env e′*
∧ *subenv e e′*
∧ *x* ∈ *black e′*
∧ *n* ≤ *num e′ x*
∧ (*n* = ∞ ∨ (∃ *y* ∈ *set* (*stack e′*). *num e′ y* = *n* ∧ *reachable x y*))
∧ (∀ *y*. *xedge-to* (*stack e′*) (*stack e*) *y* ⟶ *n* ≤ *num e′ y*)

definition *dfs-pre* **where**
dfs-pre roots e ≡

$\quad roots \subseteq vertices$
$\wedge (\forall x \in roots.\ \forall g \in gray\ e.\ reachable\ g\ x)$
$\wedge\ wf\text{-}env\ e$

definition *dfs-post* **where**
$dfs\text{-}post\ roots\ e\ res \equiv$
$\quad let\ n = fst\ res;\ e' = snd\ res$
$\quad in\ \ wf\text{-}env\ e'$
$\qquad \wedge\ subenv\ e\ e'$
$\qquad \wedge\ roots \subseteq colored\ e'$
$\qquad \wedge (\forall x \in roots.\ n \leq num\ e'\ x)$
$\qquad \wedge (n = \infty \vee (\exists x \in roots.\ \exists y \in set\ (stack\ e').\ num\ e'\ y = n \wedge reachable\ x\ y))$
$\qquad \wedge (\forall y.\ xedge\text{-}to\ (stack\ e')\ (stack\ e)\ y \longrightarrow n \leq num\ e'\ y)$

The following lemmas express some useful consequences of the pre- and post-conditions. In particular, the preconditions ensure that the function calls terminate.

lemma *dfs1-pre-domain*:
assumes *dfs1-pre x e*
shows $colored\ e \subseteq vertices$
$\qquad x \in vertices - colored\ e$
$\qquad x \notin set\ (stack\ e)$
$\qquad int\ (sn\ e) < \infty$
using *assms vfin*
unfolding *dfs1-pre-def wf-env-def wf-color-def wf-num-def colored-def*
by (*auto intro*: *psubset-card-mono*)

lemma *dfs1-pre-dfs1-dom*:
$dfs1\text{-}pre\ x\ e \Longrightarrow dfs1\text{-}dfs\text{-}dom\ (Inl(x,e))$
unfolding *dfs1-pre-def wf-env-def wf-color-def wf-num-def*
by (*auto simp*: *colored-num-def intro*!: *dfs1-dfs-termination*)

lemma *dfs-pre-dfs-dom*:
$dfs\text{-}pre\ roots\ e \Longrightarrow dfs1\text{-}dfs\text{-}dom\ (Inr(roots,e))$
unfolding *dfs-pre-def wf-env-def wf-color-def wf-num-def*
by (*auto simp*: *colored-num-def intro*!: *dfs1-dfs-termination*)

lemma *dfs-post-stack*:
assumes *dfs-post roots e res*
obtains *s* **where**
$\quad stack\ (snd\ res) = s\ @\ stack\ e$
$\quad set\ s \subseteq black\ (snd\ res)$
$\quad \forall x \in set\ (stack\ e).\ num\ (snd\ res)\ x = num\ e\ x$
using *assms* **unfolding** *dfs-post-def subenv-def* **by** *auto*

lemma *dfs-post-split*:

fixes *x e res*
defines $n' \equiv$ *fst res*
defines $e' \equiv$ *snd res*
defines $l \equiv$ *fst* (*split-list x* (*stack* e'))
defines $r \equiv$ *snd* (*split-list x* (*stack* e'))
assumes *post*: *dfs-post* (*successors x*) (*add-stack-incr x e*) *res*
(**is** *dfs-post ?roots ?e res*)
obtains *ys* **where**
$l = ys$ @ $[x]$
$x \notin$ *set ys*
set ys $\subseteq$ *black* e'
stack $e' = l$ @ r
is-subscc (*set l*)
$r =$ *stack e*
proof −
from *post* **have** *dist*: *distinct* (*stack* e')
unfolding *dfs-post-def wf-env-def* e'*-def* **by** *auto*
from *post* **obtain** *s* **where**
s: *stack* $e' = s$ @ (x # *stack e*) *set s* $\subseteq$ *black* e'
unfolding *add-stack-incr-def* e'*-def*
by (*auto intro*: *dfs-post-stack*)
then obtain *ys* **where** *ys*: $l = ys$ @ $[x]$ $x \notin$ *set ys stack* $e' = l$ @ r
unfolding *add-stack-incr-def l-def r-def*
by (*metis in-set-conv-decomp split-list-concat fst-split-list*)
with *s* **have** *l*: $l = (s$ @ $[x]) \wedge r =$ *stack e*
by (*metis dist append.assoc append.simps*(*1*) *append.simps*(*2*)
append-Cons-eq-iff distinct.simps(*2*) *distinct-append*)
from *post* **have** *wf-env* e' $x \in$ *gray* e'
by (*auto simp*: *dfs-post-def subenv-def add-stack-incr-def* e'*-def*)
with *s l* **have** *is-subscc* (*set l*)
by (*auto simp*: *add-stack-incr-def intro*: *first-gray-yields-subscc*)
with *s ys l that* **show** *?thesis* **by** *auto*
qed

A crucial lemma establishing a condition after the "then" branch following the recursive call in function *dfs1*.

lemma *dfs-post-reach-gray*:
fixes *x e res*
defines $n' \equiv$ *fst res*
defines $e' \equiv$ *snd res*
assumes *e*: *wf-env e*
and *post*: *dfs-post* (*successors x*) (*add-stack-incr x e*) *res*
(**is** *dfs-post ?roots ?e res*)
and n': $n' <$ *int* (*sn e*)
obtains *g* **where**
$g \neq x$ $g \in$ *gray* e' $x \preceq g$ *in* (*stack* e')
reachable x g reachable g x
proof −
from *post* **have** e': *wf-env* e' *subenv ?e* e'

```
    by (auto simp: dfs-post-def e′-def)
  hence x-e′: x ∈ set (stack e′) x ∈ vertices num e′ x = int(sn e)
   by (auto simp: add-stack-incr-def subenv-def wf-env-def wf-color-def colored-def)
  from e n′ have n′ ≠ ∞
    unfolding wf-env-def wf-num-def by simp
  with post e′ obtain sx y g where
    g: sx ∈ ?roots y ∈ set (stack e′) num e′ y = n′ reachable sx y
       g ∈ gray e′ g ∈ set (stack e′) y ⪯ g in (stack e′) reachable y g
    unfolding dfs-post-def e′-def n′-def wf-env-def
    by (fastforce intro: precedes-mem )
  with e′ have num e′ g ≤ num e′ y
    unfolding wf-env-def wf-num-def by metis
  with n′ x-e′ ‹num e′ y = n′›
  have num e′ g ≤ num e′ x g ≠ x by auto
  with ‹g ∈ set (stack e′)› ‹x ∈ set (stack e′)› e′
  have g ≠ x ∧ x ⪯ g in (stack e′) ∧ reachable g x
    unfolding wf-env-def wf-num-def by auto
  moreover
  from g have reachable x g
    by (metis reachable-succ reachable-trans)
  moreover
  note ‹g ∈ gray e′› that
  ultimately show ?thesis by auto
qed
```

The following lemmas represent steps in the proof of partial correctness.

```
lemma dfs1-pre-dfs-pre:
  — The precondition of dfs1 establishes that of the recursive call to dfs.
  assumes dfs1-pre x e
  shows dfs-pre (successors x) (add-stack-incr x e)
       (is dfs-pre ?roots′ ?e′)
proof –
  from assms sclosed have ?roots′ ⊆ vertices
    unfolding dfs1-pre-def by blast
  moreover
  from assms have ∀ y ∈ ?roots′. ∀ g ∈ gray ?e′. reachable g y
    unfolding dfs1-pre-def add-stack-incr-def
    by (auto dest: succ-reachable reachable-trans)
  moreover
  {
    from assms have wf-col′: wf-color ?e′
      by (auto simp: dfs1-pre-def wf-env-def wf-color-def
                  add-stack-incr-def colored-def)
    note 1 = dfs1-pre-domain[OF assms]
    from assms 1 have dist′: distinct (stack ?e′)
      unfolding dfs1-pre-def wf-env-def add-stack-incr-def by auto
    from assms have 3: sn e = card (colored e)
      unfolding dfs1-pre-def wf-env-def wf-num-def by simp
    from 1 have 4: int (sn ?e′) ≤ ∞
```

unfolding *add-stack-incr-def* **by** *simp*
with *assms* **have** *5*: $\forall x.\ -1 \leq num\ ?e'\ x \wedge (num\ ?e'\ x = \infty \vee num\ ?e'\ x < int\ (sn\ ?e'))$
unfolding *dfs1-pre-def wf-env-def wf-num-def add-stack-incr-def* **by** *auto*
from *1 vfin* **have** *finite* (*colored e*) **using** *finite-subset* **by** *blast*
with *1 3* **have** *6*: $sn\ ?e' = card\ (colored\ ?e')$
unfolding *add-stack-incr-def colored-def* **by** *auto*
from *assms 1 3* **have** *7*: $\forall y.\ num\ ?e'\ y = \infty \longleftrightarrow y \in \bigcup\ sccs\ ?e'$
by (*auto simp*: *dfs1-pre-def wf-env-def wf-num-def*
add-stack-incr-def colored-def)
from *assms 3* **have** *8*: $\forall y.\ num\ ?e'\ y = -1 \longleftrightarrow y \notin colored\ ?e'$
by (*auto simp*: *dfs1-pre-def wf-env-def wf-num-def add-stack-incr-def colored-def*)
from *assms 1* **have** $\forall y \in set\ (stack\ e).\ num\ ?e'\ y < num\ ?e'\ x$
unfolding *dfs1-pre-def add-stack-incr-def*
by (*auto dest*: *num-in-stack*)
moreover
have $\forall y \in set\ (stack\ e).\ x \preceq y\ in\ (stack\ ?e')$
unfolding *add-stack-incr-def* **by** (*auto intro*: *head-precedes*)
moreover
from *1* **have** $\forall y \in set\ (stack\ e).\ \neg(y \preceq x\ in\ (stack\ ?e'))$
unfolding *add-stack-incr-def* **by** (*auto dest*: *tail-not-precedes*)
moreover
{
fix *y z*
assume $y \in set\ (stack\ e)\ z \in set\ (stack\ e)$
with *1* **have** $x \neq y$ **by** *auto*
hence $y \preceq z\ in\ (stack\ ?e') \longleftrightarrow y \preceq z\ in\ (stack\ e)$
by (*simp add*: *add-stack-incr-def precedes-in-tail*)
}
ultimately
have *9*: $\forall y \in set\ (stack\ ?e').\ \forall z \in set\ (stack\ ?e').$
$num\ ?e'\ y \leq num\ ?e'\ z \longleftrightarrow z \preceq y\ in\ (stack\ ?e')$
using *assms*
unfolding *dfs1-pre-def wf-env-def wf-num-def add-stack-incr-def*
by *auto*
from *4 5 6 7 8 9* **have** *wf-num'*: *wf-num ?e'*
unfolding *wf-num-def* **by** *blast*
from *assms* **have** *nbtw'*: *no-black-to-white ?e'*
by (*auto simp*: *dfs1-pre-def wf-env-def no-black-to-white-def*
add-stack-incr-def colored-def)

have *stg'*: $\forall y \in set\ (stack\ ?e').\ \exists g \in gray\ ?e'.$
$y \preceq g\ in\ (stack\ ?e') \wedge reachable\ y\ g$
proof
fix *y*
assume *y*: $y \in set\ (stack\ ?e')$
show $\exists g \in gray\ ?e'.\ y \preceq g\ in\ (stack\ ?e') \wedge reachable\ y\ g$
proof (*cases* $y = x$)
case *True*

```
      then show ?thesis
        unfolding add-stack-incr-def by auto
    next
      case False
      with y have y ∈ set (stack e)
        by (simp add: add-stack-incr-def)
      with assms obtain g where
        g ∈ gray e ∧ y ⪯ g in (stack e) ∧ reachable y g
        unfolding dfs1-pre-def wf-env-def by blast
      thus ?thesis
        unfolding add-stack-incr-def
        by (auto dest: precedes-append-left[where ys=[x]])
    qed
  qed

  have str': ∀ y z. y ⪯ z in (stack ?e') ⟶ reachable z y
  proof (clarify)
    fix y z
    assume yz: y ⪯ z in stack ?e'
    show reachable z y
    proof (cases y = x)
      case True
      from yz[THEN precedes-mem(2)] stg'
      obtain g where g ∈ gray ?e' reachable z g by blast
      with True assms show ?thesis
        unfolding dfs1-pre-def add-stack-incr-def
        by (auto elim: reachable-trans)
    next
      case False
      with yz have yze: y ⪯ z in stack e
        by (simp add: add-stack-incr-def precedes-in-tail)
      with assms show ?thesis
        unfolding dfs1-pre-def wf-env-def by blast
    qed
  qed
  from assms have sccs (add-stack-incr x e) =
        {C . C ⊆ black (add-stack-incr x e) ∧ is-scc C}
    by (auto simp: dfs1-pre-def wf-env-def add-stack-incr-def)
  with wf-col' wf-num' nbtw' dist' str' stg'
  have wf-env ?e'
    unfolding wf-env-def by blast
 }
 ultimately show ?thesis
  unfolding dfs-pre-def by blast
qed
```

lemma *dfs-pre-dfs1-pre*:
— The precondition of *dfs* establishes that of the recursive call to *dfs1*, for any $x \in$ *roots* such that *num e x* $= -1$.

```
  assumes dfs-pre roots e and x ∈ roots and num e x = −1
  shows dfs1-pre x e
  using assms unfolding dfs-pre-def dfs1-pre-def wf-env-def wf-num-def by auto
```

Prove the post-condition of *dfs1* for the "then" branch in the definition of *dfs1*, assuming that the recursive call to *dfs* establishes its post-condition.

```
lemma dfs-post-dfs1-post-case1:
  fixes x e
  defines res1 ≡ dfs (successors x) (add-stack-incr x e)
  defines n1 ≡ fst res1
  defines e1 ≡ snd res1
  defines res ≡ dfs1 x e
  assumes pre: dfs1-pre x e
      and post: dfs-post (successors x) (add-stack-incr x e) res1
      and lt: fst res1 < int (sn e)
  shows dfs1-post x e res
proof −
  let ?e' = add-black x e1
  from pre have dom: dfs1-dfs-dom (Inl (x, e))
    by (rule dfs1-pre-dfs1-dom)
  from lt dom have dfs1: res = (n1, ?e')
    by (simp add: res1-def n1-def e1-def res-def case-prod-beta dfs1.psimps)
  from post have wf-env1: wf-env e1
    unfolding dfs-post-def e1-def by auto
  from post obtain s where s: stack e1 = s @ stack (add-stack-incr x e)
    unfolding e1-def by (blast intro: dfs-post-stack)
  from post have x-e1: x ∈ set (stack e1)
    by (auto intro: dfs-post-stack simp: e1-def add-stack-incr-def)
  from post have se1: subenv (add-stack-incr x e) e1
    unfolding dfs-post-def by (simp add: e1-def split-def)
  from pre lt post obtain g where
    g: g ≠ x g ∈ gray e1 x ≼ g in (stack e1)
       reachable x g reachable g x
    unfolding e1-def using dfs-post-reach-gray dfs1-pre-def by blast

  have wf-env': wf-env ?e'
  proof −
    from wf-env1 dfs1-pre-domain[OF pre] x-e1 have wf-color ?e'
      by (auto simp: dfs-pre-def wf-env-def wf-color-def add-black-def colored-def)
    moreover
    from se1
    have x ∈ gray e1 colored ?e' = colored e1
      by (auto simp: subenv-def add-stack-incr-def add-black-def colored-def)
    with wf-env1 have wf-num ?e'
      unfolding dfs-pre-def wf-env-def wf-num-def add-black-def by auto
    moreover
    from post wf-env1 have no-black-to-white ?e'
      unfolding dfs-post-def wf-env-def no-black-to-white-def
           add-black-def e1-def subenv-def colored-def
```

```
  by auto
moreover
{
  fix y
  assume y ∈ set (stack ?e′)
  hence y: y ∈ set (stack e1) by (simp add: add-black-def)
  with wf-env1 obtain z where
    z: z ∈ gray e1
       y ≼ z in stack e1
       reachable y z
    unfolding wf-env-def by blast
  have ∃ g ∈ gray ?e′.
         y ≼ g in (stack ?e′) ∧ reachable y g
  proof (cases z ∈ gray ?e′)
    case True with z show ?thesis by (auto simp: add-black-def)
  next
    case False
    with z have z = x by (simp add: add-black-def)
    with g z wf-env1 show ?thesis
      unfolding wf-env-def add-black-def
      by (auto elim: reachable-trans precedes-trans)
  qed
}
moreover
have sccs ?e′ = {C . C ⊆ black ?e′ ∧ is-scc C}
proof −
  {
    fix C
    assume C ∈ sccs ?e′
    with post have is-scc C ∧ C ⊆ black ?e′
      unfolding dfs-post-def wf-env-def add-black-def e1-def by auto
  }
  moreover
  {
    fix C
    assume C: is-scc C C ⊆ black ?e′
    have x ∉ C
    proof
      assume xC: x ∈ C
      with ‹is-scc C› g have g ∈ C
        unfolding is-scc-def by (auto dest: subscc-add)
      with wf-env1 g ‹C ⊆ black ?e′› show False
        unfolding wf-env-def wf-color-def add-black-def by auto
    qed
    with post C have C ∈ sccs ?e′
      unfolding dfs-post-def wf-env-def add-black-def e1-def by auto
  }
  ultimately show ?thesis by blast
qed
```

```
  ultimately show ?thesis — the remaining conjuncts carry over trivially
    using wf-env1 unfolding wf-env-def add-black-def by auto
qed

from pre have x ∉ set (stack e) x ∉ gray e
  unfolding dfs1-pre-def wf-env-def wf-color-def colored-def by auto
with se1 have subenv': subenv e ?e'
  unfolding subenv-def add-stack-incr-def add-black-def
  by (auto split: if-split-asm)

have xblack': x ∈ black ?e'
  unfolding add-black-def by simp

from lt have n1 < num (add-stack-incr x e) x
  unfolding add-stack-incr-def n1-def by simp
also have ... = num e1 x
  using se1 unfolding subenv-def add-stack-incr-def by auto
finally have xnum': n1 ≤ num ?e' x
  unfolding add-black-def by simp

from lt pre have n1 ≠ ∞
  unfolding dfs1-pre-def wf-env-def wf-num-def n1-def by simp
with post obtain sx y where
  sx ∈ successors x y ∈ set (stack ?e') num ?e' y = n1 reachable sx y
  unfolding dfs-post-def add-black-def n1-def e1-def by auto
with dfs1-pre-domain[OF pre]
have n1': ∃ y ∈ set (stack ?e'). num ?e' y = n1 ∧ reachable x y
  by (auto intro: reachable-trans)

{
  fix y
  assume xedge-to (stack ?e') (stack e) y
  then obtain zs z where
    y: stack ?e' = zs @ (stack e) z ∈ set zs y ∈ set (stack e) edge z y
    unfolding xedge-to-def by auto
  have n1 ≤ num ?e' y
  proof (cases z=x)
    case True
    with ‹edge z y› post show ?thesis
      unfolding dfs-post-def add-black-def n1-def e1-def by auto
  next
    case False
    with s y have xedge-to (stack e1) (stack (add-stack-incr x e)) y
      unfolding xedge-to-def add-black-def add-stack-incr-def by auto
    with post show ?thesis
      unfolding dfs-post-def add-black-def n1-def e1-def by auto
  qed
}
```

```
  with dfs1 wf-env' subenv' xblack' xnum' n1'
  show ?thesis unfolding dfs1-post-def by simp
qed
```

Prove the post-condition of *dfs1* for the "else" branch in the definition of *dfs1*, assuming that the recursive call to *dfs* establishes its post-condition.

```
lemma dfs-post-dfs1-post-case2:
  fixes x e
  defines res1 ≡ dfs (successors x) (add-stack-incr x e)
  defines n1 ≡ fst res1
  defines e1 ≡ snd res1
  defines res ≡ dfs1 x e
  assumes pre: dfs1-pre x e
    and post: dfs-post (successors x) (add-stack-incr x e) res1
    and nlt: ¬(n1 < int (sn e))
  shows dfs1-post x e res
proof −
  let ?split = split-list x (stack e1)
  let ?e' = (| black = insert x (black e1),
               gray = gray e,
               stack = snd ?split,
               sccs = insert (set (fst ?split)) (sccs e1),
               sn = sn e1,
               num = set-infty (fst ?split) (num e1) |)
  from pre have dom: dfs1-dfs-dom (Inl (x, e))
    by (rule dfs1-pre-dfs1-dom)
  from dom nlt have res: res = (∞, ?e')
    by (simp add: res1-def n1-def e1-def res-def case-prod-beta dfs1.psimps)
  from post have wf-e1: wf-env e1 subenv (add-stack-incr x e) e1
                    successors x ⊆ colored e1
    by (auto simp: dfs-post-def e1-def)
  hence gray': gray e1 = insert x (gray e)
    by (auto simp: subenv-def add-stack-incr-def)
  from post obtain l where
    l: fst ?split = l @ [x]
      x ∉ set l
      set l ⊆ black e1
      stack e1 = fst ?split @ snd ?split
      is-subscc (set (fst ?split))
      snd ?split = stack e
    unfolding e1-def by (blast intro: dfs-post-split)
  hence x: x ∈ set (stack e1) by auto
  from l have stack: set (stack e) ⊆ set (stack e1) by auto
  from wf-e1 l
  have dist: x ∉ set l    x ∉ set (stack e)
           set l ∩ set (stack e) = {}
           set (fst ?split) ∩ set (stack e) = {}
    unfolding wf-env-def by auto
```

with ‹*stack e1* = *fst ?split* @ *snd ?split*› ‹*snd ?split* = *stack e*›
have *prec*: $\forall y \in$ *set* (*stack e*). $\forall z$. $y \preceq z$ *in* (*stack e1*) $\longleftrightarrow$ $y \preceq z$ *in* (*stack e*)
by (*metis precedes-append-left-iff Int-iff empty-iff*)
from *post* **have** *numx*: *num e1 x* = *int* (*sn e*)
unfolding *dfs-post-def subenv-def add-stack-incr-def e1-def* **by** *auto*

All nodes contained in the same SCC as x are elements of *fst ?split*. Therefore, *set* (*fst ?split*) constitutes an SCC.

{
fix *y*
assume *xy*: *reachable x y* **and** *yx*: *reachable y x*
and *y*: $y \notin$ *set* (*fst ?split*)
from *l*(*1*) **have** $x \in$ *set* (*fst ?split*) **by** *simp*
with *xy y* **obtain** x' y' **where**
y': *reachable x* x' *edge* x' y' *reachable* y' *y*
$x' \in$ *set* (*fst ?split*) $y' \notin$ *set* (*fst ?split*)
using *reachable-crossing-set* **by** *metis*
with *wf-e1 l* **have** $y' \in$ *colored e1*
unfolding *wf-env-def no-black-to-white-def* **by** *auto*
from ‹*reachable x* x'› ‹*edge* x' y'› **have** *reachable x* y'
using *reachable-succ reachable-trans* **by** *blast*
moreover
from ‹*reachable* y' *y*› ‹*reachable y x*› **have** *reachable* y' *x*
by (*rule reachable-trans*)
ultimately have $y' \notin \bigcup$ *sccs e1*
using *wf-e1 gray'*
by (*auto simp*: *wf-env-def wf-color-def dest*: *sccE*)
with *wf-e1* ‹$y' \in$ *colored e1*› **have** *y'e1*: $y' \in$ *set* (*stack e1*)
unfolding *wf-env-def wf-color-def e1-def colored-def* **by** *auto*
with y' *l* **have** *y'e*: $y' \in$ *set* (*stack e*) **by** *auto*
with y' *post l* **have** *numy'*: *n1* $\leq$ *num e1* y'
unfolding *dfs-post-def e1-def n1-def xedge-to-def add-stack-incr-def*
by *force*
with *numx nlt* **have** *num e1 x* $\leq$ *num e1* y' **by** *auto*
with *y'e1 x wf-e1* **have** $y' \preceq x$ *in stack e1*
unfolding *wf-env-def wf-num-def e1-def n1-def* **by** *auto*
with *y'e* **have** $y' \preceq x$ *in stack e* **by** (*auto simp*: *prec*)
with *dist* **have** *False* **by** (*simp add*: *precedes-mem*)
}
hence $\forall y$. *reachable x y* $\wedge$ *reachable y x* $\longrightarrow$ $y \in$ *set* (*fst ?split*)
by *blast*
with *l* **have** *scc*: *is-scc* (*set* (*fst ?split*))
by (*simp add*: *is-scc-def is-subscc-def subset-antisym subsetI*)

have *wf-e'*: *wf-env ?e'*
proof −
have *wfc*: *wf-color ?e'*
proof −
from *post dfs1-pre-domain*[*OF pre*] *l*

```
    have gray ?e′ ⊆ vertices ∧ black ?e′ ⊆ vertices
         ∧ gray ?e′ ∩ black ?e′ = {}
         ∧ (⋃ sccs ?e′) ⊆ black ?e′
      by (auto simp: dfs-post-def wf-env-def wf-color-def e1-def subenv-def
                  add-stack-incr-def colored-def)
    moreover
    have set (stack ?e′) = gray ?e′ ∪ (black ?e′ − ⋃ sccs ?e′) (is ?lhs = ?rhs)
    proof
      from wf-e1 dist l show ?lhs ⊆ ?rhs
        by (auto simp: wf-env-def wf-color-def e1-def subenv-def
                    add-stack-incr-def colored-def)
    next
      from l have stack ?e′ = stack e gray ?e′ = gray e by simp+
      moreover
      from pre have gray e ⊆ set (stack e)
        unfolding dfs1-pre-def wf-env-def wf-color-def by auto
      moreover
      {
        fix v
        assume v ∈ black ?e′ − ⋃ sccs ?e′
        with l wf-e1
        have v ∈ black e1 v ∉ ⋃ sccs e1 v ∉ insert x (set l)
             v ∈ set (stack e1)
          unfolding wf-env-def wf-color-def by auto
        with l have v ∈ set (stack e) by auto
      }
      ultimately show ?rhs ⊆ ?lhs by auto
    qed
    ultimately show ?thesis
      unfolding wf-color-def colored-def by blast
  qed
  moreover
  from wf-e1 l dist prec gray′ have wf-num ?e′
    unfolding wf-env-def wf-num-def colored-def
    by (auto simp: set-infty)
  moreover
  from wf-e1 gray′ have no-black-to-white ?e′
    by (auto simp: wf-env-def no-black-to-white-def colored-def)
  moreover
  from wf-e1 l have distinct (stack ?e′)
    unfolding wf-env-def by auto
  moreover
  from wf-e1 prec
  have ∀ y z. y ⪯ z in (stack e) ⟶ reachable z y
    unfolding wf-env-def by (metis precedes-mem(1))
  moreover
  from wf-e1 prec stack dfs1-pre-domain[OF pre] gray′
  have ∀ y ∈ set (stack e). ∃ g ∈ gray e. y ⪯ g in (stack e) ∧ reachable y g
    unfolding wf-env-def by (metis insert-iff subsetCE precedes-mem(2))
```

```
    moreover
    from wf-e1 l scc have sccs ?e' = {C . C ⊆ black ?e' ∧ is-scc C}
      by (auto simp: wf-env-def dest: scc-partition)
    ultimately show ?thesis
      using l unfolding wf-env-def by simp
  qed

  from post l dist have sub: subenv e ?e'
    unfolding dfs-post-def subenv-def e1-def add-stack-incr-def
    by (auto simp: set-infty)

  from l have num: ∞ ≤ num ?e' x
    by (auto simp: set-infty)

  from l have ∀ y. xedge-to (stack ?e') (stack e) y ⟶ ∞ ≤ num ?e' y
    unfolding xedge-to-def by auto

  with res wf-e' sub num show ?thesis
    unfolding dfs1-post-def res-def by simp
qed
```

The following main lemma establishes the partial correctness of the two mutually recursive functions. The domain conditions appear explicitly as hypotheses, although we already know that they are subsumed by the preconditions. They are needed for the application of the "partial induction" rule generated by Isabelle for recursive functions whose termination was not proved. We will remove them in the next step.

```
lemma dfs-partial-correct:
  fixes x roots e
  shows
  ⟦dfs1-dfs-dom (Inl(x,e)); dfs1-pre x e⟧ ⟹ dfs1-post x e (dfs1 x e)
  ⟦dfs1-dfs-dom (Inr(roots,e)); dfs-pre roots e⟧ ⟹ dfs-post roots e (dfs roots e)
proof (induct rule: dfs1-dfs.pinduct)
  fix x e
  let ?res1 = dfs1 x e
  let ?res' = dfs (successors x) (add-stack-incr x e)
  assume ind: dfs-pre (successors x) (add-stack-incr x e)
          ⟹ dfs-post (successors x) (add-stack-incr x e) ?res'
    and pre: dfs1-pre x e
  have post: dfs-post (successors x) (add-stack-incr x e) ?res'
    by (rule ind) (rule dfs1-pre-dfs-pre[OF pre])
  show dfs1-post x e ?res1
  proof (cases fst ?res' < int (sn e))
    case True with pre post show ?thesis by (rule dfs-post-dfs1-post-case1)
  next
    case False
    with pre post show ?thesis by (rule dfs-post-dfs1-post-case2)
  qed
next
```

fix *roots e*
let *?res′* = *dfs roots e*
let *?dfs1* = *λx. dfs1 x e*
let *?dfs* = *λx e′. dfs (roots − {x}) e′*
assume *ind1*: ⋀*x*. ⟦ *roots* ≠ {}; *x* = (*SOME x. x* ∈ *roots*);
¬ *num e x* ≠ − *1*; *dfs1-pre x e* ⟧
⟹ *dfs1-post x e* (*?dfs1 x*)
and *ind′*: ⋀*x res1*.
⟦ *roots* ≠ {}; *x* = (*SOME x. x* ∈ *roots*);
res1 = (*if num e x* ≠ − *1 then* (*num e x*, *e*) *else ?dfs1 x*);
dfs-pre (*roots* − {*x*}) (*snd res1*) ⟧
⟹ *dfs-post* (*roots* − {*x*}) (*snd res1*) (*?dfs x* (*snd res1*))
and *pre*: *dfs-pre roots e*
from *pre* **have** *dom*: *dfs1-dfs-dom* (*Inr* (*roots*, *e*))
by (*rule dfs-pre-dfs-dom*)
show *dfs-post roots e ?res′*
proof (*cases roots* = {})
case *True*
with *pre dom* **show** *?thesis*
unfolding *dfs-pre-def dfs-post-def subenv-def xedge-to-def*
by (*auto simp*: *dfs.psimps*)
next
case *nempty*: *False*
define *x* **where** *x* = (*SOME x. x* ∈ *roots*)
with *nempty* **have** *x*: *x* ∈ *roots* **by** (*auto intro*: *someI*)
define *res1* **where**
res1 = (*if num e x* ≠ − *1 then* (*num e x*, *e*) *else ?dfs1 x*)
define *res2* **where**
res2 = *?dfs x* (*snd res1*)
have *post1*: *num e x* = −*1* ⟶ *dfs1-post x e* (*?dfs1 x*)
proof
assume *num*: *num e x* = −*1*
with *pre x* **have** *dfs1-pre x e*
by (*rule dfs-pre-dfs1-pre*)
with *nempty num x-def* **show** *dfs1-post x e* (*?dfs1 x*)
by (*simp add*: *ind1*)
qed
have *sub1*: *subenv e* (*snd res1*)
proof (*cases num e x* = −*1*)
case *True*
with *post1 res1-def* **show** *?thesis*
by (*auto simp*: *dfs1-post-def*)
next
case *False*
with *res1-def* **show** *?thesis* **by** *simp*
qed
have *wf1*: *wf-env* (*snd res1*)
proof (*cases num e x* = −*1*)
case *True*

```
    with res1-def post1 show ?thesis
      by (auto simp: dfs1-post-def)
  next
    case False
    with res1-def pre show ?thesis
      by (auto simp: dfs-pre-def)
  qed
  from post1 pre res1-def
  have res1: dfs-pre (roots − {x}) (snd res1)
    unfolding dfs-pre-def dfs1-post-def subenv-def by auto
  with nempty x-def res1-def ind′
  have post: dfs-post (roots − {x}) (snd res1) (?dfs x (snd res1))
    by blast
  with res2-def have sub2: subenv (snd res1) (snd res2)
    by (auto simp: dfs-post-def)
  from post res2-def have wf2: wf-env (snd res2)
    by (auto simp: dfs-post-def)
  from dom nempty x-def res1-def res2-def
  have res: dfs roots e = (min (fst res1) (fst res2), snd res2)
    by (auto simp add: dfs.psimps)
  show ?thesis
  proof −
    let ?n2 = min (fst res1) (fst res2)
    let ?e2 = snd res2

    from post res2-def
    have wf-env ?e2
      unfolding dfs-post-def by auto

    moreover
    from sub1 sub2 have sub: subenv e ?e2
      by (rule subenv-trans)

    moreover
    have x ∈ colored ?e2
    proof (cases num e x = −1)
      case True
      with post1 res1-def sub2 show ?thesis
        by (auto simp: dfs1-post-def subenv-def colored-def)
    next
      case False
      with pre sub show ?thesis
        by (auto simp: dfs-pre-def wf-env-def wf-num-def subenv-def colored-def)
    qed
    with post res2-def have roots ⊆ colored ?e2
      unfolding dfs-post-def by auto

    moreover
    have ∀ y ∈ roots. ?n2 ≤ num ?e2 y
```

```
proof
  fix y
  assume y: y ∈ roots
  show ?n2 ≤ num ?e2 y
  proof (cases y = x)
    case True
    show ?thesis
    proof (cases num e x = -1)
      case True
      with post1 res1-def have fst res1 ≤ num (snd res1) x
        unfolding dfs1-post-def by auto
      moreover
      from wf1 wf2 sub2 have num (snd res1) x ≤ num (snd res2) x
        unfolding wf-env-def by (auto elim: subenv-num)
      ultimately show ?thesis
        using ‹y=x› by simp
    next
      case False
      with res1-def wf1 wf2 sub2 have fst res1 ≤ num (snd res2) x
        unfolding wf-env-def by (auto elim: subenv-num)
      with ‹y=x› show ?thesis by simp
    qed
  next
    case False
    with y post res2-def have fst res2 ≤ num ?e2 y
      unfolding dfs-post-def by auto
    thus ?thesis by simp
  qed
qed

moreover
{
  assume n2: ?n2 ≠ ∞
  hence (fst res1 ≠ ∞ ∧ ?n2 = fst res1)
       ∨ (fst res2 ≠ ∞ ∧ ?n2 = fst res2) by auto
  hence ∃ r ∈ roots. ∃ y ∈ set (stack ?e2). num ?e2 y = ?n2 ∧ reachable r y
  proof
    assume n2: fst res1 ≠ ∞ ∧ ?n2 = fst res1
    have ∃ y ∈ set (stack (snd res1)).
              num (snd res1) y = (fst res1) ∧ reachable x y
    proof (cases num e x = -1)
      case True
      with post1 res1-def n2 show ?thesis
        unfolding dfs1-post-def by auto
    next
      case False
      with wf1 res1-def n2 have x ∈ set (stack (snd res1))
        unfolding wf-env-def wf-color-def wf-num-def colored-def by auto
      with False res1-def show ?thesis
```

```
          by auto
        qed
        with sub2 x n2 show ?thesis
          unfolding subenv-def by fastforce
      next
        assume fst res2 ≠ ∞ ∧ ?n2 = fst res2
        with post res2-def show ?thesis
          unfolding dfs-post-def by auto
      qed
    }
    hence ?n2 = ∞ ∨ (∃ r ∈ roots. ∃ y ∈ set (stack ?e2). num ?e2 y = ?n2 ∧
reachable r y)
      by blast

    moreover
    have ∀ y. xedge-to (stack ?e2) (stack e) y ⟶ ?n2 ≤ num ?e2 y
    proof (clarify)
      fix y
      assume y: xedge-to (stack ?e2) (stack e) y
      show ?n2 ≤ num ?e2 y
      proof (cases num e x = −1)
        case True
        from sub1 obtain s1 where
          s1: stack (snd res1) = s1 @ stack e
          by (auto simp: subenv-def)
        from sub2 obtain s2 where
          s2: stack ?e2 = s2 @ stack (snd res1)
          by (auto simp: subenv-def)
        from y obtain zs z where
          z: stack ?e2 = zs @ stack e z ∈ set zs
             y ∈ set (stack e) edge z y
          by (auto simp: xedge-to-def)
        with s1 s2 have z ∈ (set s1) ∪ (set s2) by auto
        thus ?thesis
        proof
          assume z ∈ set s1
          with s1 z have xedge-to (stack (snd res1)) (stack e) y
            by (auto simp: xedge-to-def)
          with post1 res1-def ‹num e x = −1›
          have fst res1 ≤ num (snd res1) y
            by (auto simp: dfs1-post-def)
          moreover
          with wf1 wf2 sub2 have num (snd res1) y ≤ num ?e2 y
            unfolding wf-env-def by (auto elim: subenv-num)
          ultimately show ?thesis by simp
        next
          assume z ∈ set s2
          with s1 s2 z have xedge-to (stack ?e2) (stack (snd res1)) y
            by (auto simp: xedge-to-def)
```

```
          with post res2-def show ?thesis
            by (auto simp: dfs-post-def)
        qed
      next
        case False
        with y post res1-def res2-def show ?thesis
          unfolding dfs-post-def by auto
      qed
    qed

    ultimately show ?thesis
      using res unfolding dfs-post-def by simp
  qed
 qed
qed
```

8 Theorems establishing total correctness

Combining the previous theorems, we show total correctness for both the auxiliary functions and the main function *tarjan*.

```
theorem dfs-correct:
  dfs1-pre x e ⟹ dfs1-post x e (dfs1 x e)
  dfs-pre roots e ⟹ dfs-post roots e (dfs roots e)
  using dfs-partial-correct dfs1-pre-dfs1-dom dfs-pre-dfs-dom by (blast+)

theorem tarjan-correct: tarjan = { C . is-scc C ∧ C ⊆ vertices }
proof −
  have dfs-pre vertices init-env
    by (auto simp: dfs-pre-def init-env-def wf-env-def wf-color-def colored-def
                   wf-num-def no-black-to-white-def is-scc-def precedes-def)
  hence res: dfs-post vertices init-env (dfs vertices init-env)
    by (rule dfs-correct)
  thus ?thesis
    by (auto simp: tarjan-def init-env-def dfs-post-def wf-env-def wf-color-def
                   colored-def subenv-def)
qed

end — context graph
end — theory Tarjan
```