Correctness of Tarjan's Algorithm

Stephan Merz

August 5, 2017

Contents

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/scc.html.

declare *Let-def* [*simp*] — expand let-constructions automatically

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

record $'v$ *env* =
  blacks:: $'v$ *set*
  stack:: $'v$ *list*

```
  sccs:: 'v set set
  sn:: nat
  num:: 'v ⇒ int


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

```
definition edge where
  — the edge relation over vertices
  edge x y ≡ x ∈ vertices ∧ 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[simp]: reachable x x
| reachable-succ: ⟦x ∈ vertices; y ∈ successors x⟧ ⟹ reachable x y
| reachable-trans: ⟦reachable x y; reachable y z⟧ ⟹ reachable x z
```

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 ∈ S and 3: y ∉ S
  obtains x' y' where
    x' ∈ S y' ∉ S edge x' y' reachable x x' reachable y' y
proof −
  from assms
  have ∃ x' y'. x' ∈ S ∧ y' ∉ S ∧ edge x' y' ∧ reachable x x' ∧ reachable y' y
    unfolding edge-def using reachable-refl reachable-trans
    by induct (blast+)
  with that show ?thesis by blast
qed
```

2 Strongly connected components

definition *is-subscc* **where**
$is\text{-}subscc\ S \equiv \forall x \in S.\ \forall y \in S.\ reachable\ x\ y$

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

lemma *subscc-add*:
assumes $is\text{-}subscc\ S$ **and** $x \in S$
and $reachable\ x\ y$ **and** $reachable\ y\ x$
shows $is\text{-}subscc\ (insert\ y\ S)$
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 $is\text{-}scc\ S$ **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 $is\text{-}scc\ S$ **and** $is\text{-}scc\ S'$ **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

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

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

lemma *set-infty*:
$(set\text{-}infty\ xs\ f)\ x = (if\ x \in set\ xs\ then\ \infty\ else\ f\ x)$
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 strictly before and 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**
$split\text{-}list\ x\ []\ = ([], [])$

| *split-list* x $(y \# xs)$ =
  (*if* $x = y$ *then* $([], xs)$ *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 $(fst\ (split\text{-}list\ x\ xs))\ @\ (x\ \#\ snd\ (split\text{-}list\ x\ xs)) = xs$
using *assms* **by** (*induct xs*) (*auto simp*: *split-def*)

lemma *split-list-fst*:
$x \notin set\ (fst\ (split\text{-}list\ x\ xs))$
by (*induct xs*) (*auto simp*: *split-def*)

lemma *unique-split-list*:
— An element that occurs only once identifies a unique decomposition of a list.
assumes $x \notin set\ xs$ **and** $x \notin set\ ys$
shows $xs\ @\ x\ \#\ ys = xs'\ @\ x\ \#\ ys' \longleftrightarrow (xs = xs' \wedge ys = ys')$
using *assms*
by(*auto simp*: *append-eq-Cons-conv Cons-eq-append-conv append-eq-append-conv2*)

Push a vertex on the stack and increment the sequence number. The pushed vertex is associated with the (old) sequence number.

definition *add-stack-incr* **where**
add-stack-incr x e =
  (*let* n = *sn* e *in*
    e ⦇ *stack* := x # (*stack* e),
      sn := $n+1$,
      num := (*num* e) (x := *int* n) ⦈)

definition *add-blacks* **where**
— Add vertex x to the set of black vertices in e.
add-blacks x e = e ⦇ *blacks* := *insert* x (*blacks* e) ⦈

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 grays =
   (let (n1, e1) =
       dfs' (successors x) (add-stack-incr x e) (insert x grays) in
     if n1 < int (sn e) then (n1, add-blacks x e1)
     else
      (let (l,r) = split-list x (stack e1) in
        (∞,
          ⦇ blacks = insert x (blacks e1),
            stack = r,
            sccs = insert (insert x (set l)) (sccs e1),
            sn = sn e1,
            num = set-infty (x # l) (num e1) ⦈ )))
| dfs' roots e grays =
   (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 grays)
     in
     (let res2 = dfs' (roots − {x}) (snd res1) grays in
        (min (fst res1) (fst res2), (snd res2)) )))
  by pat-completeness auto
```

```
definition init-env where
  init-env ≡ ⦇ blacks = {},
               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 for 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 gray nodes.

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

definition *grays-num-defined* **where**
  grays-num-defined e grays $\equiv \forall\, x \in$ *grays*. *num e x* $\neq -1$

lemma *grays-num-defined*:
  $[\![$*dfs1-dfs′-dom* (*Inl* (*x*,*e*,*grays*)); *grays-num-defined e grays*$]\!] \Longrightarrow$
    grays-num-defined (*snd* (*dfs1 x e grays*)) *grays*
  $[\![$*dfs1-dfs′-dom* (*Inr* (*roots*,*e*,*grays*)); *grays-num-defined e grays*$]\!] \Longrightarrow$
    grays-num-defined (*snd* (*dfs′ roots e grays*)) *grays*
proof (*induct rule*: *dfs1-dfs′.pinduct*)
  case (*1 x e grays*)
  then show *?case*
    by (*auto simp*: *dfs1.psimps case-prod-beta grays-num-defined-def*
               add-blacks-def add-stack-incr-def set-infty infty-def)
next
  case (*2 roots e grays*)
  then show *?case*
    by (*fastforce simp*: *dfs′.psimps case-prod-beta*)
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 $\equiv$
    { (*Inl*(*x*, *e*::′*v env*, *grays*), *Inr*(*roots*,*e*,*grays*)) |
      x e grays roots .
      roots $\subseteq$ *vertices* $\wedge$ *x* $\in$ *roots* $\wedge$ *grays* $\subseteq$ *vertices* }
  $\cup$ { (*Inr*(*roots*, *e*::′*v env*, *insert x grays*), *Inl*(*x*, *e′*, *grays*)) |
      x e e′ grays roots .
      grays $\subseteq$ *vertices* $\wedge$ *x* $\in$ *vertices* $-$ *grays* }
  $\cup$ { (*Inr*(*roots*, *e*::′*v env*, *grays*), *Inr*(*roots′*, *e′*, *grays*)) |
      roots roots′ e e′ grays .
      roots′ $\subseteq$ *vertices* $\wedge$ *roots* $\subset$ *roots′* $\wedge$ *grays* $\subseteq$ *vertices* }

We prove well-foundedness of the above relation using the following function that embeds it into triples whose first component is the complement of the gray 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*, *grays*)) = (*vertices* $-$ *grays*, {*x*}, *1*::*nat*)
| *dfs1-dfs′-to-tuple* (*Inr*(*roots*, *e*::′*v env*, *grays*)) = (*vertices* $-$ *grays*, *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)
  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:
  ⟦grays ⊆ vertices; x ∈ vertices − grays; grays-num-defined e grays⟧
   ⟹ dfs1-dfs'-dom (Inl(x,e,grays))
  ⟦grays ⊆ vertices; roots ⊆ vertices; grays-num-defined e grays⟧
   ⟹ dfs1-dfs'-dom (Inr(roots,e,grays))
proof −
  { fix args
    have (case args
          of Inl(x,e,grays) ⇒
             grays ⊆ vertices ∧ x ∈ vertices − grays ∧ grays-num-defined e grays
          | Inr(roots,e,grays) ⇒
             grays ⊆ vertices ∧ roots ⊆ vertices ∧ grays-num-defined e grays)
        ⟶ dfs1-dfs'-dom args (is ?P args ⟶ ?Q args)
    proof (rule wf-induct[OF wf-term])
      fix arg :: ('v × 'v env × 'v set) + ('v set × 'v env × 'v set)
      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 grays where a: arg = Inl(x,e,grays)
            using dfs1.cases by metis
          have ?Q (Inl(x,e,grays))
          proof (rule dfs1-dfs'.domintros)
            let ?recarg = Inr (successors x, add-stack-incr x e, insert x grays)
            from a P have (?recarg, arg) ∈ dfs1-dfs'-term
              by (auto simp: dfs1-dfs'-term-def)
            moreover
            from a P sclosed have ?P ?recarg
```

```
      by (auto simp: add-stack-incr-def grays-num-defined-def)
    ultimately show ?Q ?recarg
      using ih by auto
  qed
  with a show ?thesis by simp
next
  case (Inr b)
  then obtain roots e grays where b: arg = Inr(roots,e,grays)
    using dfs'.cases by metis
  let ?sx = SOME x. x ∈ roots
  let ?rec1arg = Inl (?sx, e, grays)
  let ?rec2arg = Inr (roots − {?sx}, e, grays)
  let ?rec3arg = Inr (roots − {?sx}, snd (dfs1 ?sx e grays), grays)
  have ?Q (Inr(roots,e,grays))
  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)
    moreover
    from sx 2 P b have ?P ?rec1arg
      by (auto simp: grays-num-defined-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)
    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)
    from sx b P have (?rec3arg, arg) ∈ dfs1-dfs'-term
      by (auto simp: dfs1-dfs'-term-def)
    moreover
    have dfs1-dfs'-dom ?rec1arg
    proof −
      from sx P b have (?rec1arg, arg) ∈ dfs1-dfs'-term
        by (auto simp: dfs1-dfs'-term-def)
      moreover
```

```
          from sx 2 P b have ?P ?rec1arg
            by (auto simp: grays-num-defined-def)
          ultimately show ?thesis
            using ih by auto
        qed
        with P b have grays-num-defined (snd (dfs1 ?sx e grays)) grays
          by (force elim: grays-num-defined)
        with P b 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 ⟦grays ⊆ vertices; x ∈ vertices − grays; grays-num-defined e grays⟧
    ⟹ dfs1-dfs'-dom (Inl(x,e,grays))
  by auto
from dom
show ⟦grays ⊆ vertices; roots ⊆ vertices; grays-num-defined e grays⟧
 ⟹ dfs1-dfs'-dom (Inr(roots,e,grays))
  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 ∈ 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 $x \neq z$
  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*)
proof (*cases* $y \in$ *set ys*)
  case *True*
  from *this*[*THEN split-list*] **show** *?thesis*
    unfolding *precedes-def* **by** *force*
next
  case *False*
  with *assms* **show** *?thesis*
    unfolding *precedes-def* **by** *auto*
qed

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 ∧ 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 ∧ 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
```

6 Predicates and lemmas about environments

```
definition subenv where
```

$subenv\ e\ e' \equiv$
$(\exists\, s.\ stack\ e' = s\ @\ (stack\ e) \wedge set\ s \subseteq blacks\ e')$
$\wedge\ blacks\ e \subseteq blacks\ e'$
$\wedge\ sccs\ e \subseteq sccs\ e'$
$\wedge\ (\forall\, x \in set\ (stack\ e).\ num\ e\ x = num\ e'\ x)$

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
$wf\text{-}color\ e\ grays \equiv$
$grays \subseteq vertices \wedge blacks\ e \subseteq vertices$
$\wedge\ blacks\ e \cap grays = \{\}$
$\wedge\ (\bigcup\ sccs\ e) \subseteq blacks\ e$
$\wedge\ set\ (stack\ e) = grays \cup (blacks\ e - \bigcup\ sccs\ e)$

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

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 grays wf-num e grays*
and e': $wf\text{-}color\ e'\ grays\ wf\text{-}num\ e'\ grays$
shows $num\ e\ x \leq num\ e'\ x$
using *assms* **unfolding** *wf-color-def wf-num-def subenv-def*
by (*smt Diff-partition UnE UnI1 UnI2 Union-Un-distrib*)

definition *no-black-to-white* **where**
— successors of black vertices must be black or gray
$no\text{-}black\text{-}to\text{-}white\ e\ grays \equiv$
$\forall\, x\ y.\ edge\ x\ y \wedge x \in blacks\ e \longrightarrow y \in blacks\ e \cup grays$

definition *wf-env* **where**
  wf-env e grays ≡
    wf-color e grays ∧ *wf-num e grays*
  ∧ *no-black-to-white e grays* ∧ *distinct* (*stack e*)
  ∧ (∀ *g* ∈ *grays*. ∀ *y* ∈ *set* (*stack e*).
        y ⪯ *g in* (*stack e*) ⟶ *reachable g y*)
  ∧ (∀ *y* ∈ *set* (*stack e*). ∃ *g* ∈ *grays*.
        y ⪯ *g in* (*stack e*) ∧ *reachable y g*)

lemma *num-in-stack*:
  assumes *wf-env e grays* **and** *x* ∈ *set* (*stack e*)
  shows *num e x* ≠ −*1*
        num e x < *int* (*sn e*)
  using *assms* **unfolding** *wf-env-def wf-color-def wf-num-def* **by** (*blast+*)

definition *num-of-reachable* **where**
  — some vertex in *e*'s stack has number *n* and is reachable from *x*
  num-of-reachable n x e ≡
    ∃ *y* ∈ *set* (*stack e*). *num e y* = *n* ∧ *reachable x y*

lemma *subscc-after-last-gray*:
  assumes *e*: *wf-env e* (*insert x grays*)
    and *x*: *stack e* = *ys* @ (*x* # *zs*)
    and *ys*: *set ys* ⊆ *blacks e*
  shows *is-subscc* (*insert x* (*set ys*))
proof −
  from *e x* **have** ∀ *y* ∈ *set ys*. ∃ *g* ∈ *insert x grays*. *reachable y g*
    unfolding *wf-env-def* **by** *force*
  moreover
  have ∀ *g* ∈ *grays*. *reachable g x*
  proof
    fix *g*
    assume *g* ∈ *grays*
    with *e x ys* **have** *g* ∈ *insert x* (*set zs*)
      unfolding *wf-env-def wf-color-def* **by** *auto*
    with *x* **have** *x* ⪯ *g in stack e*
      unfolding *precedes-def* **by** *fastforce*
    with *e x* ‹*g* ∈ *grays*› **show** *reachable g x*
      unfolding *wf-env-def* **by** *auto*
  qed
  moreover
  from *e x* **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 grays $\equiv$
$x \in$ *vertices*
$\wedge\ x \notin$ *grays* $\cup$ *blacks e*
$\wedge\ (\forall g \in$ *grays. reachable g x*$)$
$\wedge$ *wf-env e grays*
$\wedge\ (\forall C.\ C \in$ *sccs e* $\longleftrightarrow$ *is-scc C* $\wedge\ C \subseteq$ *blacks e*$)$

definition *dfs1-post* **where**
dfs1-post x e grays res $\equiv$
let $n =$ *fst res*; $e' =$ *snd res*
in *wf-env* e' *grays*
$\wedge$ *subenv e* e'
$\wedge\ (\forall C.\ C \in$ *sccs* $e' \longleftrightarrow$ *is-scc C* $\wedge\ C \subseteq$ *blacks* $e')$
$\wedge\ x \in$ *blacks* e'
$\wedge\ n \leq$ *num* e' x
$\wedge\ (n = \infty \vee$ *num-of-reachable n x* $e')$
$\wedge\ (\forall y.$ *xedge-to* $($*stack* $e')$ $($*stack e*$)$ $y \longrightarrow n \leq$ *num* e' $y)$

definition *dfs′-pre* **where**
dfs′-pre roots e grays $\equiv$
roots $\subseteq$ *vertices*
$\wedge\ (\forall x \in$ *roots*. $\forall g \in$ *grays. reachable g x*$)$
$\wedge$ *wf-env e grays*
$\wedge\ (\forall C.\ C \in$ *sccs e* $\longleftrightarrow$ *is-scc C* $\wedge\ C \subseteq$ *blacks e*$)$

definition *dfs′-post* **where**
dfs′-post roots e grays res $\equiv$
let $n =$ *fst res*; $e' =$ *snd res*
in *wf-env* e' *grays*
$\wedge$ *subenv e* e'
$\wedge\ (\forall C.\ C \in$ *sccs* $e' \longleftrightarrow$ *is-scc C* $\wedge\ C \subseteq$ *blacks* $e')$
$\wedge$ *roots* $\subseteq$ *blacks* $e' \cup$ *grays*
$\wedge\ (\forall x \in$ *roots*. $n \leq$ *num* e' $x)$
$\wedge\ (n = \infty \vee (\exists x \in$ *roots. num-of-reachable n x* $e'))$
$\wedge\ (\forall y.$ *xedge-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.

lemma *dfs1-pre-domain*:
assumes *dfs1-pre x e grays*

shows *grays* ∪ *blacks e* ⊆ *vertices*
x ∈ *vertices* − (*grays* ∪ *blacks e*)
x ∉ *set* (*stack e*)
int (*sn e*) < ∞
using *assms vfin*
unfolding *dfs1-pre-def wf-env-def wf-color-def wf-num-def infty-def*
by (*auto intro*: *psubset-card-mono*)

lemma *dfs1-pre-dfs1-dom*:
dfs1-pre x e grays ⟹ *dfs1-dfs′-dom* (*Inl*(*x*,*e*,*grays*))
unfolding *dfs1-pre-def wf-env-def wf-color-def wf-num-def*
by (*auto simp*: *grays-num-defined-def intro*!: *dfs1-dfs′-termination*)

lemma *dfs′-pre-dfs′-dom*:
dfs′-pre roots e grays ⟹ *dfs1-dfs′-dom* (*Inr*(*roots*,*e*,*grays*))
unfolding *dfs′-pre-def wf-env-def wf-color-def wf-num-def*
by (*auto simp*: *grays-num-defined-def intro*!: *dfs1-dfs′-termination*)

lemma *dfs′-post-stack*:
assumes *dfs′-post roots e grays res*
obtains *s* **where**
stack (*snd res*) = *s* @ *stack e*
set s ⊆ *blacks* (*snd res*)
∀ *x* ∈ *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 grays res*
defines *n′* ≡ *fst res*
defines *e′* ≡ *snd res*
defines *l* ≡ *fst* (*split-list x* (*stack e′*))
defines *r* ≡ *snd* (*split-list x* (*stack e′*))
assumes *post′*: *dfs′-post* (*successors x*) (*add-stack-incr x e*)
(*insert x grays*) *res*
(**is** *dfs′-post ?roots ?e ?grays res*)
shows *stack e′* = *l* @ (*x* # *r*)
set l ⊆ *blacks e′*
is-subscc (*insert x* (*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* @ *stack ?e set s* ⊆ *blacks e′*
unfolding *e′-def* **by** (*blast intro*: *dfs′-post-stack*)
hence *stack e′* = *s* @ (*x* # *stack e*)
unfolding *add-stack-incr-def* **by** *simp*

moreover
from *s* **show** *s'*: *stack e'* = *l* @ (*x* # *r*)
unfolding *add-stack-incr-def l-def r-def* **by** (*simp add*: *split-list-concat*)
ultimately have *l* = *s* ∧ *r* = *stack e*
by (*metis dist unique-split-list distinct.simps(2)*
distinct-append not-distinct-conv-prefix)
with *s* **show** *set l* ⊆ *blacks e' r* = *stack e*
unfolding *dfs'-post-def* **by** (*simp*+)
with *post' s'* **show** *is-subscc* (*insert x* (*set l*))
unfolding *dfs'-post-def e'-def*
by (*auto elim*: *subscc-after-last-gray*)
qed

A crucial lemma establishing a condition after the recursive call in function *dfs1*.

lemma *dfs'-post-reach-gray*:
fixes *x e grays res*
defines *n'* ≡ *fst res*
defines *e'* ≡ *snd res*
assumes *wf-e*: *wf-env e grays*
and *post'*: *dfs'-post* (*successors x*) (*add-stack-incr x e*)
(*insert x grays*) *res*
(**is** *dfs'-post ?roots ?e ?grays res*)
and *n'*: *n'* < *int* (*sn e*)
obtains *g* **where**
g ∈ *grays g* ∈ *set* (*stack e'*) *num e' g* < *num e' x*
reachable x g reachable g x
proof −
from *dfs'-post-stack*[*OF post'*] **obtain** *s* **where**
stack (*snd res*) = *s* @ *stack ?e*
∀ *z* ∈ *set* (*stack ?e*). *num* (*snd res*) *z* = *num ?e z*
by *metis*
with *post'* **have** *x-e'*: *x* ∈ *set* (*stack e'*) *x* ∈ *vertices num e' x* = *int*(*sn e*)
unfolding *add-stack-incr-def dfs'-post-def wf-env-def wf-color-def e'-def* **by** *auto*
from *wf-e n'* **have** *n'* ≠ ∞
unfolding *wf-env-def wf-num-def* **by** *simp*
with *post'* **obtain** *sx y* **where**
y: *sx* ∈ *?roots y* ∈ *set* (*stack e'*) *num e' y* = *n' reachable sx y*
unfolding *dfs'-post-def num-of-reachable-def e'-def n'-def* **by** *auto*
from *post'* ‹*y* ∈ *set* (*stack e'*)› **obtain** *g* **where**
g: *g* ∈ *?grays* ∧ *y* ⪯ *g in* (*stack e'*) ∧ *reachable y g*
unfolding *dfs'-post-def wf-env-def e'-def* **by** *smt*
hence *g* ∈ *set* (*stack e'*) **by** (*blast intro*: *precedes-mem*)
with *post' y g* **have** *ng*: *num e' g* ≤ *num e' y*
unfolding *dfs'-post-def wf-env-def wf-num-def e'-def* **by** *metis*
with *n' x-e' y* **have** *gx*: *g* ≠ *x* **by** *auto*
with *g ng y x-e' n'* ‹*g* ∈ *set* (*stack e'*)›
have *gx*: *g* ∈ *grays* ∧ *g* ∈ *set* (*stack e'*) ∧ *num e' g* < *num e' x* ∧ *reachable x g*

by (*auto intro*: *reachable-succ reachable-trans*)
with ‹$x \in$ *set* (*stack e'*)› *post' g* **have** *reachable g x*
unfolding *dfs'-post-def wf-env-def wf-num-def e'-def le-less* **by** *meson*
with *gx that* **show** *?thesis* **by** *blast*
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 grays*
shows *dfs'-pre* (*successors x*) (*add-stack-incr x e*) (*insert x grays*)
(**is** *dfs'-pre ?roots' ?e' ?grays'*)
proof –
from *assms sclosed* **have** *?roots'* $\subseteq$ *vertices*
unfolding *dfs1-pre-def* **by** *blast*
moreover
from *assms* **have** $\forall$ *y* $\in$ *?roots'*. $\forall$ *g* $\in$ *?grays'*. *reachable g y*
unfolding *dfs1-pre-def* **by** (*metis insertE reachable-succ reachable-trans*)
moreover
{
from *assms* **have** *wf-col'*: *wf-color ?e' ?grays'*
unfolding *dfs1-pre-def wf-env-def wf-color-def add-stack-incr-def*
by *auto*
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* (*grays* $\cup$ *blacks e*)
unfolding *dfs1-pre-def wf-env-def wf-num-def* **by** *simp*
from *1* **have** *4*: *int* (*sn ?e'*) $\leq \infty$
unfolding *add-stack-incr-def* **by** *simp*
with *assms* **have** *5*: $\forall$ *x*. *−1* $\leq$ *num ?e' x* $\wedge$ (*num ?e' x* = ∞ $\vee$ *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* (*grays* $\cup$ *blacks e*) **using** *finite-subset* **by** *blast*
with *1 3* **have** *6*: *sn ?e'* = *card* (*?grays'* $\cup$ *blacks ?e'*)
unfolding *add-stack-incr-def* **by** *auto*
from *assms 1 3* **have** *7*: $\forall$ *y*. *num ?e' y* = $\infty \longleftrightarrow$ *y* $\in \bigcup$ *sccs ?e'*
unfolding *dfs1-pre-def wf-env-def wf-num-def add-stack-incr-def infty-def*
by *auto*
from *assms 3* **have** *8*: $\forall$ *y*. *num ?e' y* = *−1* $\longleftrightarrow$ *y* $\notin$ *?grays'* $\cup$ *blacks ?e'*
unfolding *dfs1-pre-def wf-env-def wf-num-def add-stack-incr-def*
by *auto*
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' ?grays'*
  unfolding *wf-num-def* **by** *blast*
from *assms* **have** *nbtw'*: *no-black-to-white ?e' ?grays'*
  unfolding *dfs1-pre-def wf-env-def no-black-to-white-def add-stack-incr-def*
  by *auto*
{
  fix *g y*
  assume *g*: $g \in ?grays'$ **and** *y*: $y \in set\ (stack\ ?e')$
    and *yg*: $y \preceq g\ in\ stack\ ?e'$
  have *reachable g y*
  proof (*cases* $y = x$)
    case *True* **with** *assms g* **show** *?thesis*
      unfolding *dfs1-pre-def* **by** *auto*
  next
    case *False*
    with *yg* **have** *yge*: $y \preceq g\ in\ stack\ e$
      by (*simp add*: *add-stack-incr-def precedes-in-tail*)
    moreover
    with *1* **have** $g \neq x$
      by (*auto dest*: *precedes-mem*)
    ultimately show *?thesis*
      using *g assms* **unfolding** *dfs1-pre-def wf-env-def*
      by (*auto dest*: *precedes-mem*)
  qed
}
hence *gts'*: $\forall\, x \in ?grays'.\ \forall\, y \in set\ (stack\ ?e').$
      $y \preceq x\ in\ (stack\ ?e') \longrightarrow reachable\ x\ y$
  by *blast*
{
  fix *y*
  assume *y*: $y \in set\ (stack\ ?e')$
  have $\exists\, g \in ?grays'.\ y \preceq g\ in\ (stack\ ?e') \land reachable\ y\ g$
  proof (*cases* $y = x$)

case *True*
then show *?thesis*
unfolding *add-stack-incr-def* **by** *auto*
next
case *False*
with *y* **have** $y \in set$ (*stack e*)
by (*simp add*: *add-stack-incr-def*)
with *assms* **obtain** *g* **where** $g \in grays \land y \preceq g$ *in* (*stack e*) $\land$ *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
}
with *wf-col′ wf-num′ nbtw′ dist′ gts′*
have *wf-env ?e′ ?grays′*
unfolding *wf-env-def* **by** *blast*
}
moreover
from *assms* **have** $\forall C.\ C \in sccs$ (*add-stack-incr x e*)
$\longleftrightarrow$ *is-scc* $C \land C \subseteq$ *blacks* (*add-stack-incr x e*)
unfolding *dfs1-pre-def add-stack-incr-def* **by** *auto*
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 grays* **and** $x \in roots$ **and** *num e x* $= -1$
shows *dfs1-pre x e grays*
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 grays*
defines *res1* $\equiv$ *dfs′* (*successors x*) (*add-stack-incr x e*) (*insert x grays*)
defines *n1* $\equiv$ *fst res1*
defines *e1* $\equiv$ *snd res1*
defines *res* $\equiv$ *dfs1 x e grays*
assumes *pre*: *dfs1-pre x e grays*
and *post′*: *dfs′-post* (*successors x*) (*add-stack-incr x e*)
(*insert x grays*) *res1*
and *lt*: *fst res1* $<$ *int* (*sn e*)
shows *dfs1-post x e grays res*
proof –
let *?e′* = *add-blacks x e1*
from *pre* **have** *dom*: *dfs1-dfs′-dom* (*Inl* (*x*, *e*, *grays*))

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* (*insert x grays*)
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* ∈ *grays g* ∈ *set* (*stack e1*) *num e1 g* < *num e1 x*
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' grays*
proof −
from *wf-env1 dfs1-pre-domain*[*OF pre*] **have** *wf-color ?e' grays*
unfolding *dfs'-pre-def wf-env-def wf-color-def add-blacks-def* **by** *force*
moreover
from *wf-env1* **have** *wf-num ?e' grays*
unfolding *dfs'-pre-def wf-env-def wf-num-def add-blacks-def* **by** *auto*
moreover
from *post' wf-env1* **have** *no-black-to-white ?e' grays*
unfolding *dfs'-post-def wf-env-def no-black-to-white-def*
add-blacks-def edge-def e1-def
by *auto*
moreover
{
fix *y*
assume *y* ∈ *set* (*stack ?e'*)
hence *y*: *y* ∈ *set* (*stack e1*) **by** (*simp add*: *add-blacks-def*)
with *wf-env1* **obtain** *z* **where**
z: *z* ∈ *insert x grays*
y ⪯ *z in stack e1*
reachable y z
unfolding *wf-env-def* **by** *blast*
have ∃ *g* ∈ *grays*.
y ⪯ *g in* (*stack ?e'*) ∧ *reachable y g*
proof (*cases z* ∈ *grays*)
case *True* **with** *z* **show** *?thesis* **by** (*auto simp*: *add-blacks-def*)
next
case *False*
with *z* **have** *z* = *x* **by** *simp*
with *y z g x-e1 wf-env1*
have *y* ⪯ *g in stack e1*
unfolding *wf-env-def wf-num-def* **by** *smt*

with *g z* ‹*z*=*x*› **show** *?thesis*
  by (*auto elim*: *reachable-trans simp*: *add-blacks-def*)
qed
}
ultimately show *?thesis* — the remaining conjuncts carry over trivially
  using *wf-env1* **unfolding** *wf-env-def add-blacks-def* **by** *auto*
qed

from *pre* **have** $x \notin set\ (stack\ e)$
  unfolding *dfs1-pre-def wf-env-def wf-color-def* **by** *auto*
with *se1* **have** *subenv'*: *subenv e ?e'*
  unfolding *subenv-def add-stack-incr-def add-blacks-def* **by** *auto metis*

have *sccs'*: $\forall C.\ C \in sccs\ ?e' \longleftrightarrow is\text{-}scc\ C \wedge C \subseteq blacks\ ?e'$ (**is** $\forall C.\ ?P\ C$)
proof
  fix *C*
  {
    assume $C \in sccs\ ?e'$
    with *post'* **have** $is\text{-}scc\ C \wedge C \subseteq blacks\ ?e'$
      unfolding *dfs'-post-def add-blacks-def e1-def* **by** *auto*
  }
  moreover
  {
    assume *C*: $is\text{-}scc\ C\ C \subseteq blacks\ ?e'$
    have $x \notin C$
    proof
      assume *xC*: $x \in C$
      with ‹*is-scc C*› *g* **have** $g \in C$
        unfolding *is-scc-def* **by** (*auto dest*: *subscc-add*)
      with *wf-env'* ‹$C \subseteq blacks\ ?e'$› ‹$g \in grays$› **show** *False*
        unfolding *wf-env-def wf-color-def* **by** *auto*
    qed
    with *post' C* **have** $C \in sccs\ ?e'$
      unfolding *dfs'-post-def add-blacks-def e1-def* **by** *auto*
  }
  ultimately show *?P C* **by** *blast*
qed

have *xblack'*: $x \in blacks\ ?e'$
  unfolding *add-blacks-def* **by** *simp*

from *lt* **have** $n1 < num\ (add\text{-}stack\text{-}incr\ x\ e)\ x$
  unfolding *add-stack-incr-def n1-def* **by** *simp*
also have $\ldots = num\ e1\ x$
  using *se1* **unfolding** *subenv-def add-stack-incr-def* **by** *auto*
finally have *xnum'*: $n1 \leq num\ ?e'\ x$
  unfolding *add-blacks-def* **by** *simp*

from *lt pre* **have** $n1 \neq \infty$

```
    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 num-of-reachable-def add-blacks-def n1-def e1-def by
auto
  with dfs1-pre-domain[OF pre] have n1′: num-of-reachable n1 x ?e′
    unfolding num-of-reachable-def
    by (force intro: reachable-succ 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 edge-def dfs′-post-def add-blacks-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-blacks-def add-stack-incr-def by auto
      with post′ show ?thesis
        unfolding dfs′-post-def add-blacks-def n1-def e1-def by auto
    qed
  }

  with dfs1 wf-env′ subenv′ sccs′ 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 grays
  defines res1 ≡ dfs′ (successors x) (add-stack-incr x e) (insert x grays)
  defines n1 ≡ fst res1
  defines e1 ≡ snd res1
  defines res ≡ dfs1 x e grays
  assumes pre: dfs1-pre x e grays
      and post′: dfs′-post (successors x) (add-stack-incr x e)
                          (insert x grays) res1
      and nlt: ¬(n1 < int (sn e))
  shows dfs1-post x e grays res
proof −
  let ?split = split-list x (stack e1)
  let ?e′ = (| blacks = insert x (blacks e1),
```

stack = *snd ?split*,
sccs = *insert* ((*insert x* (*set* (*fst ?split*)))) (*sccs e1*),
sn = *sn e1*,
num = *set-infty* (*x* # *fst ?split*) (*num e1*) ⦈
from *pre* **have** *dom*: *dfs1-dfs′-dom* (*Inl* (*x*, *e*, *grays*))
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′* **obtain** *l* **where**
l: *stack e1* = *l* @ (*x* # *stack e*)
fst ?split = *l*
snd ?split = *stack e*
set l ⊆ *blacks e1*
is-subscc (*insert x* (*set l*))
unfolding *e1-def* **using** *dfs′-post-split* **by** *metis*
hence *x*: *x* ∈ *set* (*stack e1*) **by** *auto*
from *l* **have** *stack*: *set* (*stack e*) ⊆ *set* (*stack e1*) **by** *auto*
from *post′* **have** *wf-e1*: *wf-env e1* (*insert x grays*)
unfolding *dfs′-post-def e1-def* **by** *auto*
with *l* **have** *dist*: *x* ∉ *set l x* ∉ *set* (*stack e*) *set l* ∩ *set* (*stack e*) = {}
unfolding *wf-env-def* **by** *auto*
with *l*
have *prec*: ∀ *y* ∈ *set* (*stack e*). ∀ *z*. *y* ⪯ *z in* (*stack e1*) ⟷ *y* ⪯ *z in* (*stack e*)
by (*metis disjoint-insert*(*1*) *insert-Diff precedes-append-left-iff precedes-in-tail*)
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 *l*.
— Therefore, *set l* ∪ {*x*} constitutes an SCC.
{
fix *y*
assume *xy*: *reachable x y* **and** *yx*: *reachable y x*
and *y*: *y* ∉ *insert x* (*set l*)
from *xy y* **obtain** *x′ y′* **where**
y′: *reachable x x′ edge x′ y′ reachable y′ y*
x′ ∈ *insert x* (*set l*) *y′* ∉ *insert x* (*set l*)
using *reachable-crossing-set* **by** (*metis insertI1*)
with *post′ l* **have** *y′* ∈ *blacks e1* ∪ (*insert x grays*)
unfolding *edge-def dfs′-post-def wf-env-def no-black-to-white-def e1-def*
by (*smt insertE split-beta subsetCE*)
have *y′* ∉ ⋃ *sccs e1*
proof
assume *y′* ∈ ⋃ *sccs e1*
with *post′* **obtain** *C* **where**
C ∈ *sccs e1 y′* ∈ *C is-scc C*
unfolding *dfs′-post-def e1-def* **by** (*meson UnionE*)
moreover
from ‹*reachable x x′*› ‹*edge x′ y′*› **have** *reachable x y′*
using *edge-def reachable-succ reachable-trans* **by** *blast*

```
    moreover
    from ‹reachable y′ y› ‹reachable y x› have reachable y′ x
      by (rule reachable-trans)
    ultimately have x ∈ C by (blast intro: sccE)
    with ‹C ∈ sccs e1› post′ show False
      unfolding dfs′-post-def wf-env-def wf-color-def e1-def by auto
  qed
  with post′ ‹y′ ∈ blacks e1 ∪ (insert x grays)›
  have y′e1: y′ ∈ set (stack e1)
    unfolding dfs′-post-def wf-env-def wf-color-def e1-def by auto
  with y′ l have y′e: y′ ∈ set (stack e) by auto
  with y′ post′ l have numy′: n1 ≤ num e1 y′
    unfolding dfs′-post-def e1-def n1-def edge-def xedge-to-def add-stack-incr-def
    by force
  with numx nlt have num e1 x ≤ num e1 y′ by auto
  with y′e1 x post′ have y′ ⪯ x in stack e1
    unfolding dfs′-post-def wf-env-def wf-num-def e1-def n1-def by force
  with y′e have y′ ⪯ x in stack e by (auto simp: prec)
  with dist have False by (simp add: precedes-mem)
}
hence ∀ y. reachable x y ∧ reachable y x ⟶ y ∈ insert x (set l)
  by blast
with l have scc: is-scc (insert x (set l))
  by (simp add: is-scc-def is-subscc-def subset-antisym subsetI)

have wf-e′: wf-env ?e′ grays
proof −
  have wfc: wf-color ?e′ grays
  proof −
    from wf-e1 dfs1-pre-domain[OF pre] l
     have grays ⊆ vertices ∧ blacks ?e′ ⊆ vertices
         ∧ grays ∩ blacks ?e′ = {}
         ∧ (⋃ sccs ?e′) ⊆ blacks ?e′
      unfolding wf-env-def wf-color-def by auto
    moreover
    have set (stack ?e′) = grays ∪ (blacks ?e′ − ⋃ sccs ?e′) (is ?lhs = ?rhs)
    proof
      from wf-e1 dist l show ?lhs ⊆ ?rhs
        unfolding wf-env-def wf-color-def by auto
    next

      {
        fix v
        assume v ∈ ?rhs hence v ∈ ?lhs
        proof
          assume v ∈ grays with pre l show ?thesis
            unfolding dfs1-pre-def wf-env-def wf-color-def by auto
        next
          assume v: v ∈ blacks ?e′ − ⋃ sccs ?e′
```

```
          hence v ∈ blacks e1 − ⋃ sccs e1 by auto
          with wf-e1 have v ∈ set (stack e1)
            unfolding wf-env-def wf-color-def by auto
          with l v show ?thesis
            by (metis DiffE Sup-insert Un-iff insert-iff list.simps(15)
                      select-convs(2) set-append simps(3))
        qed
      }
      thus ?rhs ⊆ ?lhs by blast
    qed
    ultimately show ?thesis
      unfolding wf-color-def by blast
  qed
  moreover
  from wf-e1 l dist prec have wf-num ?e′ grays
    unfolding wf-env-def wf-num-def by (auto simp: set-infty infty-def)

  moreover
  from post′ have no-black-to-white ?e′ grays
    by (auto simp: dfs′-post-def wf-env-def no-black-to-white-def e1-def edge-def)
  moreover
  from wf-e1 l have distinct (stack ?e′)
    unfolding wf-env-def by auto
  moreover
  from wf-e1 prec stack
  have ∀ g ∈ grays. ∀ y ∈ set (stack e). y ⪯ g in (stack e) ⟶ reachable g y
    unfolding wf-env-def by auto
  moreover
  from wf-e1 prec stack dfs1-pre-domain[OF pre]
  have ∀ y ∈ set (stack e). ∃ g ∈ grays. y ⪯ g in (stack e) ∧ reachable y g
    unfolding wf-env-def by (metis insert-iff subsetCE precedes-mem(2))
  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)

{
  fix C
  assume C ∈ sccs ?e′
  with post′ scc l have is-scc C ∧ C ⊆ blacks ?e′
    unfolding dfs′-post-def e1-def by auto
}
moreover
{
  fix C
  assume C: is-scc C C ⊆ blacks ?e′
```

```
    have C ∈ sccs ?e′
    proof (cases x ∈ C)
      case True
      with l scc ‹is-scc C› show ?thesis
        by (metis scc-partition IntI insertCI select-convs(3))
    next
      case False
      with C post′ show ?thesis
        unfolding dfs′-post-def e1-def by auto
    qed
  }
  ultimately have sccs: ∀ C. C ∈ sccs ?e′ ⟷ is-scc C ∧ C ⊆ blacks ?e′
    by blast

  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 sccs 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 grays
  shows
  ⟦dfs1-dfs′-dom (Inl(x,e,grays)); dfs1-pre x e grays⟧
   ⟹ dfs1-post x e grays (dfs1 x e grays)
  ⟦dfs1-dfs′-dom (Inr(roots,e,grays)); dfs′-pre roots e grays⟧
   ⟹ dfs′-post roots e grays (dfs′ roots e grays)
proof (induct rule: dfs1-dfs′.pinduct)
  fix x e grays
  let ?res1 = dfs1 x e grays
  let ?res′ = dfs′ (successors x) (add-stack-incr x e) (insert x grays)
  assume ind: dfs′-pre (successors x) (add-stack-incr x e) (insert x grays)
          ⟹ dfs′-post (successors x) (add-stack-incr x e)
                      (insert x grays) ?res′
    and pre: dfs1-pre x e grays
  have post′: dfs′-post (successors x) (add-stack-incr x e) (insert x grays) ?res′
    by (rule ind) (rule dfs1-pre-dfs′-pre[OF pre])
  show dfs1-post x e grays ?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 grays*
let *?res'* = *dfs' roots e grays*
let *?dfs1* = λx. *dfs1 x e grays*
let *?dfs'* = $\lambda x\ e'$. *dfs'* (*roots* − {*x*}) *e' grays*
assume *ind1*: $\bigwedge x$. ⟦ *roots* ≠ {}; *x* = (*SOME x. x* ∈ *roots*);
¬ *num e x* ≠ − *1*; *dfs1-pre x e grays* ⟧
⟹ *dfs1-post x e grays* (*?dfs1 x*)
and *ind'*: $\bigwedge x$ *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*) *grays* ⟧
⟹ *dfs'-post* (*roots* − {*x*}) (*snd res1*) *grays* (*?dfs' x* (*snd res1*))
and *pre*: *dfs'-pre roots e grays*
from *pre* **have** *dom*: *dfs1-dfs'-dom* (*Inr* (*roots*, *e*, *grays*))
by (*rule dfs'-pre-dfs'-dom*)
show *dfs'-post roots e grays ?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 grays* (*?dfs1 x*)
proof
assume *num*: *num e x* = −*1*
with *pre x* **have** *dfs1-pre x e grays*
by (*rule dfs'-pre-dfs1-pre*)
with *nempty num x-def* **show** *dfs1-post x e grays* (*?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*) *grays*
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*) *grays*
unfolding *dfs'-pre-def dfs1-post-def* **by** *auto*
with *nempty x-def res1-def*
have *post*: *dfs'-post* (*roots* − {*x*}) (*snd res1*) *grays* (*?dfs' x* (*snd res1*))
by (*rule ind'*)
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*) *grays*
by (*auto simp*: *dfs'-post-def*)
from *dom nempty x-def res1-def res2-def*
have *res*: *dfs' roots e grays* = (*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 grays*
$\forall C.\ C \in$ *sccs ?e2* $\longleftrightarrow$ *is-scc C* $\wedge$ $C \subseteq$ *blacks ?e2*
unfolding *dfs'-post-def* **by** *auto*

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

moreover
have $x \in$ *blacks ?e2* $\cup$ *grays*
proof (*cases num e x* = −*1*)
case *True*
with *post1 res1-def* **have** $x \in$ *blacks* (*snd res1*)
unfolding *dfs1-post-def* **by** *auto*
with *sub2* **show** *?thesis*
unfolding *subenv-def* **by** *auto*
next

```
  case False
  with pre have x ∈ blacks e ∪ grays
    unfolding dfs'-pre-def wf-env-def wf-num-def by auto
  with sub show ?thesis by (auto simp: subenv-def)
qed
with post res2-def have roots ⊆ blacks ?e2 ∪ grays
  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 (blast intro: 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 intro: 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. num-of-reachable ?n2 r ?e2
  proof
    assume n2: fst res1 ≠ ∞ ∧ ?n2 = fst res1
    have num-of-reachable (fst res1) x (snd res1)
```

```
    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 by auto
      with False res1-def show ?thesis
        unfolding num-of-reachable-def by auto
    qed
    with sub2 x n2 show ?thesis
      unfolding subenv-def num-of-reachable-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 = ∞ ∨ (∃ x ∈ roots. num-of-reachable ?n2 x ?e2)
  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 (blast intro: 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 grays ⟹ dfs1-post x e grays (dfs1 x e grays)
  dfs'-pre roots e grays ⟹ dfs'-post roots e grays (dfs' roots e grays)
  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
               wf-num-def no-black-to-white-def infty-def is-scc-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)
qed

end — context graph
end — theory Tarjan
```