Correctness of Tarjan's Algorithm

Stephan Merz

October 17, 2018

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/1-68bis/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*
⟨*proof*⟩

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

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

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*⟩

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

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

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 $y \in S$
⟨*proof*⟩

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 $S = S'$
⟨*proof*⟩

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)
⟨*proof*⟩

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 $(fst\ (split\text{-}list\ x\ xs))\ @\ (snd\ (split\text{-}list\ x\ xs)) = xs$
  ⟨*proof*⟩

lemma *fst-split-list*:
  assumes $x \in set\ xs$
  shows $\exists\, ys.\ fst\ (split\text{-}list\ x\ xs) = ys\ @\ [x] \wedge x \notin set\ ys$
  ⟨*proof*⟩

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) ))
  ⟨proof⟩

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*:
  $dfs1\text{-}dfs\text{-}dom\ (Inl\ (x,e)) \Longrightarrow black\ e \subseteq black\ (snd\ (dfs1\ x\ e))$
  $dfs1\text{-}dfs\text{-}dom\ (Inr\ (roots,e)) \Longrightarrow black\ e \subseteq black\ (snd\ (dfs\ roots\ e))$
  ⟨*proof*⟩

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

lemma *colored-increasing*:
  $dfs1\text{-}dfs\text{-}dom\ (Inl\ (x,e)) \Longrightarrow$
    $colored\ e \subseteq colored\ (snd\ (dfs1\ x\ e)) \wedge$
    $colored\ (add\text{-}stack\text{-}incr\ x\ e)$
    $\subseteq colored\ (snd\ (dfs\ (successors\ x)\ (add\text{-}stack\text{-}incr\ x\ e)))$
  $dfs1\text{-}dfs\text{-}dom\ (Inr\ (roots,e)) \Longrightarrow$
    $colored\ e \subseteq colored\ (snd\ (dfs\ roots\ e))$
⟨*proof*⟩

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

lemma *dfs-num-defined*:
  $[\![dfs1\text{-}dfs\text{-}dom\ (Inl\ (x,e));\ num\ (snd\ (dfs1\ x\ e))\ v = -1]\!] \Longrightarrow$
    $num\ e\ v = -1$
  $[\![dfs1\text{-}dfs\text{-}dom\ (Inr\ (roots,e));\ num\ (snd\ (dfs\ roots\ e))\ v = -1]\!] \Longrightarrow$
    $num\ e\ v = -1$
  ⟨*proof*⟩

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**
  $colored\text{-}num\ e \equiv \forall\, v \in colored\ e.\ v \in vertices \wedge num\ e\ v \neq -1$

lemma *colored-num*:
  $[\![dfs1\text{-}dfs\text{-}dom\ (Inl\ (x,e));\ x \in vertices;\ colored\text{-}num\ e]\!] \Longrightarrow$
    $colored\text{-}num\ (snd\ (dfs1\ x\ e))$
  $[\![dfs1\text{-}dfs\text{-}dom\ (Inr\ (roots,e));\ roots \subseteq vertices;\ colored\text{-}num\ e]\!] \Longrightarrow$
    $colored\text{-}num\ (snd\ (dfs\ roots\ e))$
⟨*proof*⟩

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**
  $dfs1\text{-}dfs\text{-}term \equiv$
    $\{\ (Inl(x,\ e{::}'v\ env),\ Inr(roots,e))\ |$
      $x\ e\ roots\ .$
      $roots \subseteq vertices \wedge x \in roots \wedge colored\ e \subseteq vertices\ \}$
  $\cup\ \{\ (Inr(roots,\ add\text{-}stack\text{-}incr\ x\ e),\ Inl(x,\ e))\ |$

$x\ e\ roots\ .$
$colored\ e \subseteq vertices \wedge x \in vertices - colored\ e\ \}$
$\cup\ \{\ (Inr(roots,\ e{::}'v\ env),\ Inr(roots',\ e'))\ |$
$roots\ roots'\ e\ e'\ .$
$roots' \subseteq vertices \wedge roots \subset roots' \wedge$
$colored\ e' \subseteq colored\ e \wedge colored\ e \subseteq vertices\ \}$

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

lemma *wf-term*: *wf dfs1-dfs-term*
⟨*proof*⟩

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*:
$[\![x \in vertices - colored\ e;\ colored\text{-}num\ e]\!] \Longrightarrow dfs1\text{-}dfs\text{-}dom\ (Inl(x,\ e))$
$[\![roots \subseteq vertices;\ colored\text{-}num\ e]\!] \Longrightarrow dfs1\text{-}dfs\text{-}dom\ (Inr(roots,\ e))$
⟨*proof*⟩

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* (- $\preceq$ - *in* - [*100*,*100*,*100*] *39*) **where**
— x has an occurrence in xs that precedes an occurrence of y.
$x \preceq y\ in\ xs \equiv \exists\, l\ r.\ xs = l\ @\ (x\ \#\ r) \wedge y \in set\ (x\ \#\ r)$

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

⟨*proof*⟩

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

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

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

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

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

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

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*⟩

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

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*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *subenv-trans*:
assumes $subenv\ e\ e'$ **and** $subenv\ e'\ e''$
shows $subenv\ e\ e''$
⟨*proof*⟩

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

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

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 $num\ e\ x \le num\ e'\ x$

⟨*proof*⟩

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

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

lemma *num-in-stack*:
assumes *wf-env e* **and** $x \in set\ (stack\ e)$
shows $num\ e\ x \neq -1$
$num\ e\ x < int\ (sn\ e)$
⟨*proof*⟩

Numbers assigned to different stack elements are distinct.

lemma *num-inj*:
assumes *wf-env e* **and** $x \in set\ (stack\ e)$
and $y \in set\ (stack\ e)$ **and** $num\ e\ x = num\ e\ y$
shows $x = y$
⟨*proof*⟩

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*: *wf-env e*
and *x*: $stack\ e = ys\ @\ (x\ \#\ zs)$
and *g*: $x \in gray\ e$
and *ys*: $set\ ys \subseteq black\ e$
shows *is-subscc* (*insert x* (*set ys*))
⟨*proof*⟩

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**
  $dfs1\text{-}pre\ x\ e \equiv$
    $x \in vertices$
  $\wedge\ x \notin colored\ e$
  $\wedge\ (\forall g \in gray\ e.\ reachable\ g\ x)$
  $\wedge\ wf\text{-}env\ e$

definition *dfs1-post* **where**
  $dfs1\text{-}post\ x\ e\ res \equiv$
    $let\ n = fst\ res;\ e' = snd\ res$
    $in\ \ wf\text{-}env\ e'$
      $\wedge\ subenv\ e\ e'$
      $\wedge\ x \in black\ e'$
      $\wedge\ n \leq num\ e'\ x$
      $\wedge\ (n = \infty \vee (\exists y \in set\ (stack\ e').\ num\ e'\ y = n \wedge reachable\ x\ y))$
      $\wedge\ (\forall y.\ xedge\text{-}to\ (stack\ e')\ (stack\ e)\ y \longrightarrow n \leq num\ e'\ y)$

definition *dfs-pre* **where**
  $dfs\text{-}pre\ roots\ e \equiv$
    $roots \subseteq vertices$
  $\wedge\ (\forall x \in roots.\ \forall g \in gray\ e.\ reachable\ g\ x)$
  $\wedge\ wf\text{-}env\ e$

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

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 $dfs1\text{-}pre\ x\ e$
  shows $colored\ e \subseteq vertices$
    $x \in vertices - colored\ e$
    $x \notin set\ (stack\ e)$
    $int\ (sn\ e) < \infty$
  ⟨*proof*⟩

lemma *dfs1-pre-dfs1-dom*:
  $dfs1\text{-}pre\ x\ e \Longrightarrow dfs1\text{-}dfs\text{-}dom\ (Inl(x,e))$

⟨*proof*⟩

lemma *dfs-pre-dfs-dom*:
  dfs-pre roots e $\Longrightarrow$ *dfs1-dfs-dom* (*Inr*(*roots*,*e*))
  ⟨*proof*⟩

lemma *dfs-post-stack*:
  assumes *dfs-post roots e res*
  obtains *s* **where**
    stack (*snd res*) = *s* @ *stack e*
    set s $\subseteq$ *black* (*snd res*)
    $\forall$ *x* $\in$ *set* (*stack e*). *num* (*snd res*) *x* = *num e x*
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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** $x \in roots$ **and** $num\ e\ x = -1$
shows *dfs1-pre x e*
⟨*proof*⟩

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*⟩

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*⟩

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 pre-conditions. 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*$]\!] \Longrightarrow$ *dfs1-post x e* (*dfs1 x e*)
$[\![$*dfs1-dfs-dom* (*Inr*(*roots*,*e*)); *dfs-pre roots e*$]\!] \Longrightarrow$ *dfs-post roots e* (*dfs roots e*)
⟨*proof*⟩

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 $\Longrightarrow$ *dfs1-post x e* (*dfs1 x e*)
dfs-pre roots e $\Longrightarrow$ *dfs-post roots e* (*dfs roots e*)
⟨*proof*⟩

theorem *tarjan-correct*: *tarjan* = { *C* . *is-scc C* $\wedge$ *C* $\subseteq$ *vertices* }
⟨*proof*⟩

end — context graph
end — theory Tarjan