Semi-automatic proof of Strong connectivity

jean-jacques.levy@inria.fr

journées PPS, 12-10-2017

Plan

- motivation
- algorithm
- formal proof
- other systems
- conclusion

.. joint work (in progress) with **Ran Chen** [VSTTE 2017])

also cooperation with Cyril Cohen, Laurent Théry, Stephan Merz

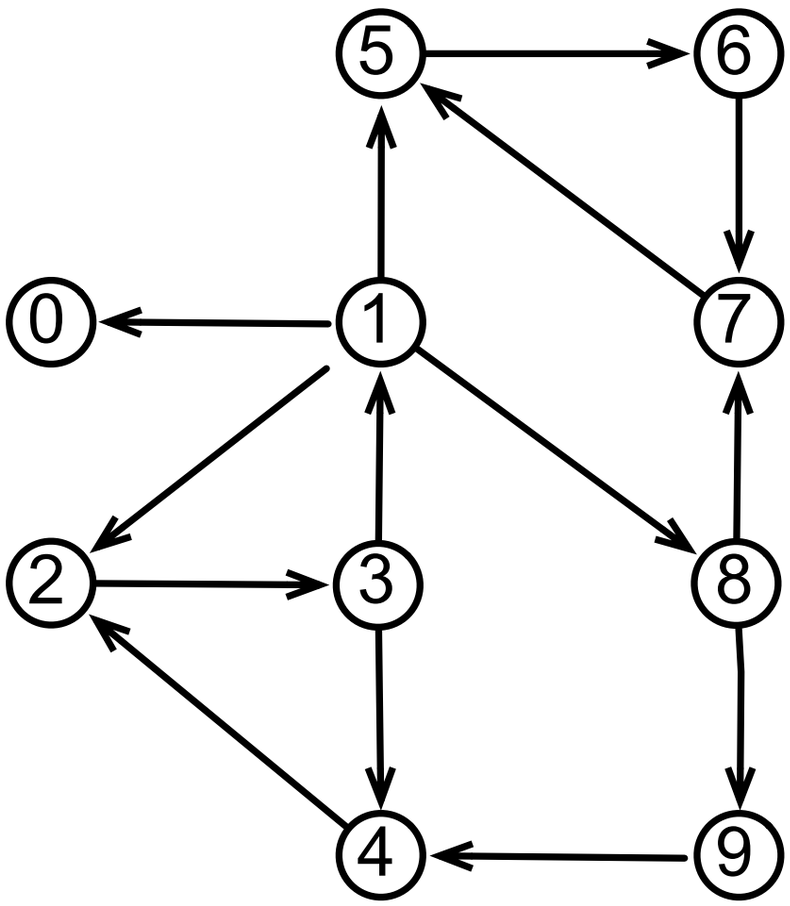
Motivation

- nice algorithms → **simple** formal proofs
- **fully** published in articles or journals
- how to publish formal proofs ?
- formal proofs should be **exact** and **readable** (by human)
- mix automatic and interactive proofs
- first-order logic is **easy** to understand, but **not** expressive
- algorithms on graphs = a good testbed

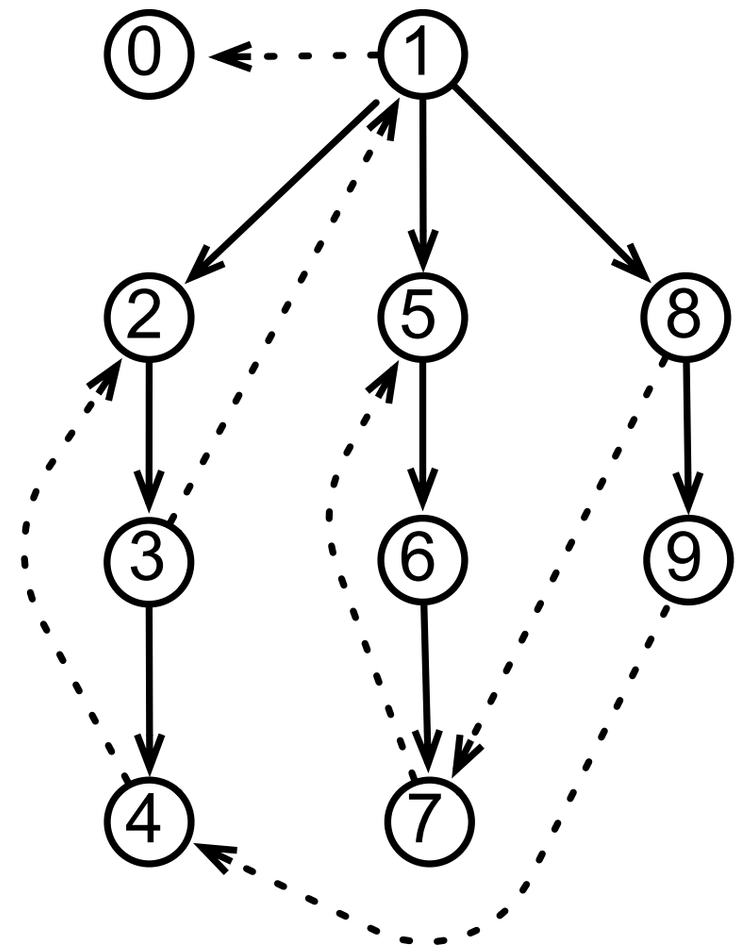
One-pass linear-time algorithm

[tarjan 1972]

Depth-first-search

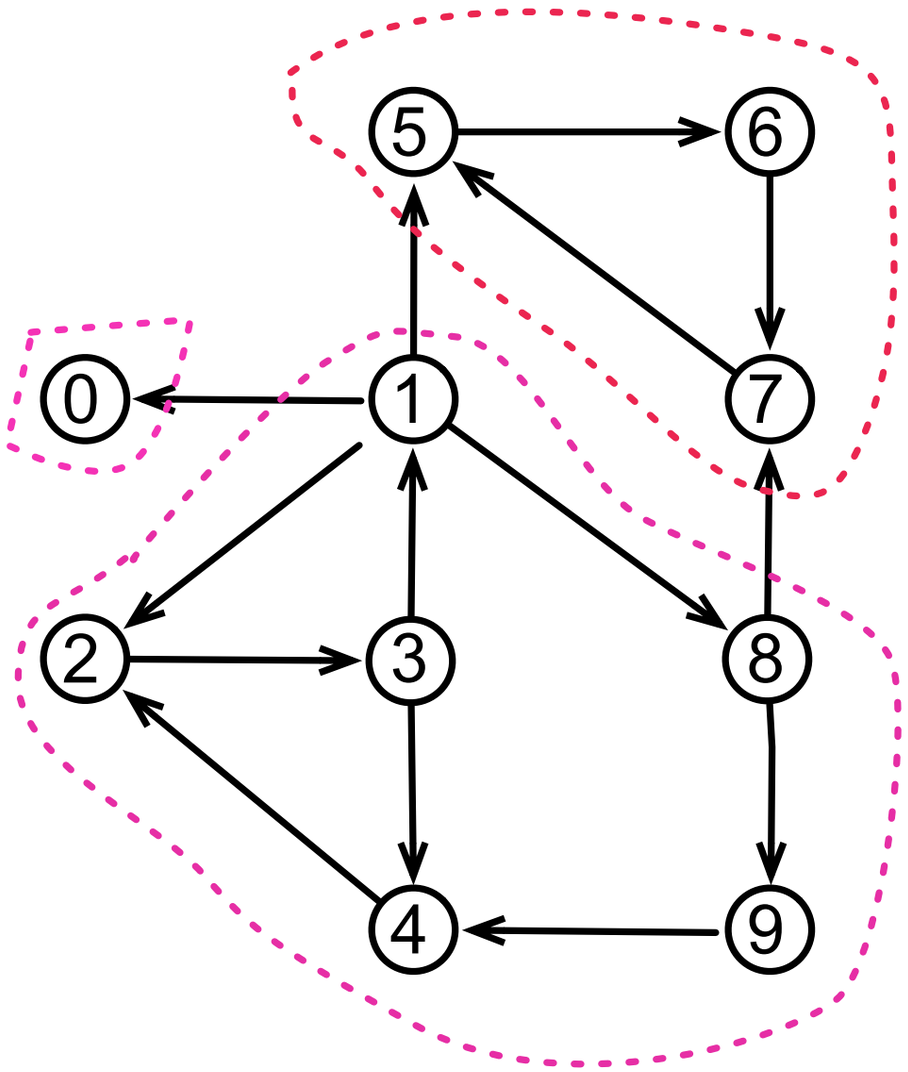


graph

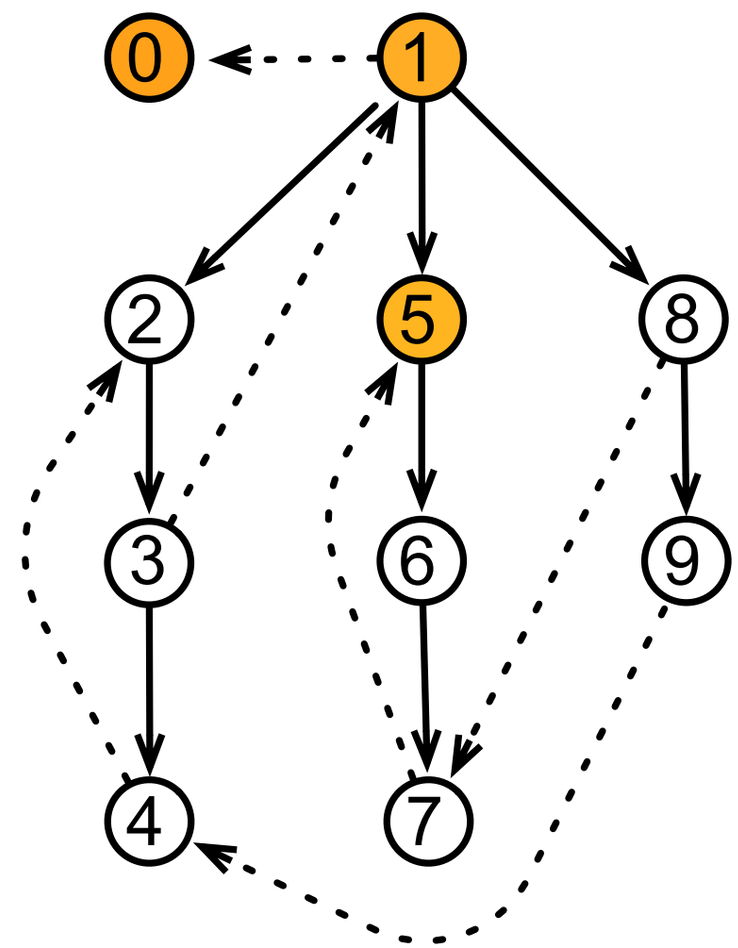


spanning tree (forest)

The algorithm (1/3)

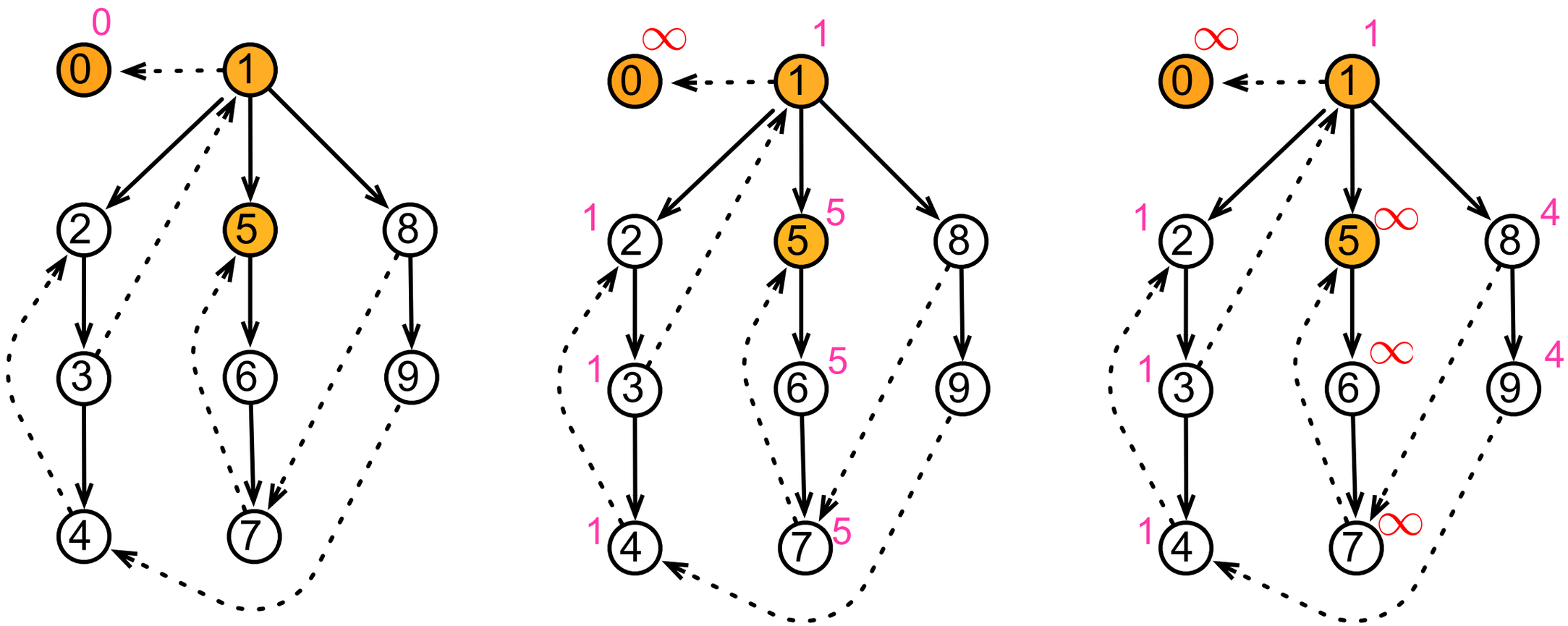


3 SCCs (strongly connected components)



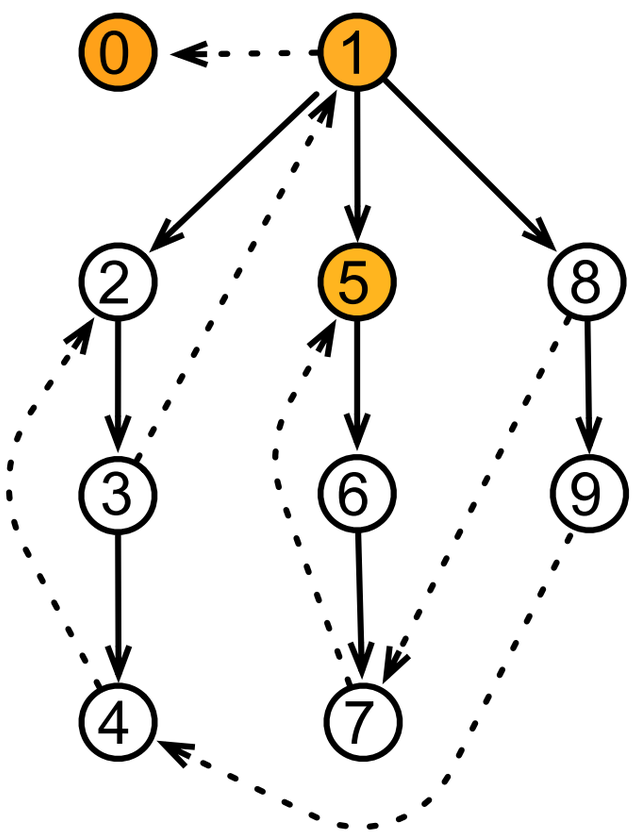
3 vertices are their bases

The algorithm (2/3)



$$LOWLINK(x) = \min\ (\ \{num[x]\}\ \cup\ \{num[y]\ \mid\ x \xrightarrow{*} \cdots\!\rightarrow y \wedge x \text{ and } y \text{ are in same connected component}\}\ )$$

The algorithm (3/3)



successive values of the working stack

0	1	1	1	1	1	1	1	1	1	0	increasing rank
		2	2	2	2	2	2	2	2	1	
			3	3	3	3	3	3	3	2	
				4	4	4	4	4	4	3	
					5	5	5	8	8	4	
						6	6		9	5	
							7			6	

The program

```
let rec printSCC (x: int) (s: stack int)
    (num: array int) (sn: ref int) =
 Stack.push x s;
 num[x] ← !sn; sn := !sn + 1;
 let low = ref num[x] in
 foreach y in (successors x) do
   let m = if num[y] = -1
     then printSCC y s num sn
     else num[y] in
   low := Math.min m !low
 done;
 if !low = num[x] then begin
   repeat
     let y = Stack.pop s in
     Printf.printf "%d " y;
     num[y] ← max_int;
     if y = x then break;
   done;
   Printf.printf "\n";
   low := max_int;
 end;
 return !low;
```

- print each component on a line

Imperative style

Proof in algorithms books (1/2)

- consider the spanning trees (forest)
- tree structure of strongly connected components
- 2-3 lemmas about ancestors in spanning trees

LEMMA 10. *Let v and w be vertices in G which lie in the same strongly connected component. Let F be a spanning forest of G generated by repeated depth-first search. Then v and w have a common ancestor in F. Further, if u is the highest numbered common ancestor of v and w, then u lies in the same strongly connected component as v and w.*

$$\mathit{LOWLINK}(x) = \min\ (\ \{\mathit{num}[x]\} \ \cup\ \{\mathit{num}[y] \mid x \overset{*}{\Longrightarrow}\!\!\hookrightarrow y \wedge x \text{ and } y \text{ are in same connected component}\}\ )$$

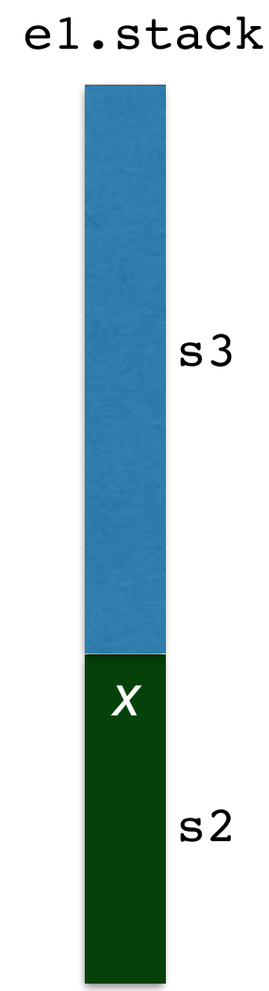
LEMMA 12. *Let G be a directed graph with* LOWLINK *defined as above relative to some spanning forest F of G generated by depth-first search. Then v is the root of some strongly connected component of G if and only if* LOWLINK $(v) = v$.

Proof in algorithms book (2/2)

- give the program
- proof  program
- that part of the proof is very informal

Our program (1/3)

```
let rec dfs1 x e =
  let n = e.sn in
  let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
  let (s2, s3) = split x e1.stack in
  if n1 < n then (n1, e1) else
    (max_int(), {stack = s3; sccs = add (elements s2) e1.sccs;
      sn = e1.sn; num = set_max_int s2 e1.num})

with  dfs roots e = if is_empty roots then (max_int(), e) else
  let x = choose roots in
  let roots' = remove x roots in
  let (n1, e1) = if e.num[x] ≠ -1 then (e.num[x], e) else dfs1 x e in
  let (n2, e2) = dfs roots' e1 in (min n1 n2, e2)

let tarjan () =
  let e0 = {stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e0 in e'.sccs
```



returns *LOWLINK*(x) and new environment

Functional programming

Formal proof

using Why3

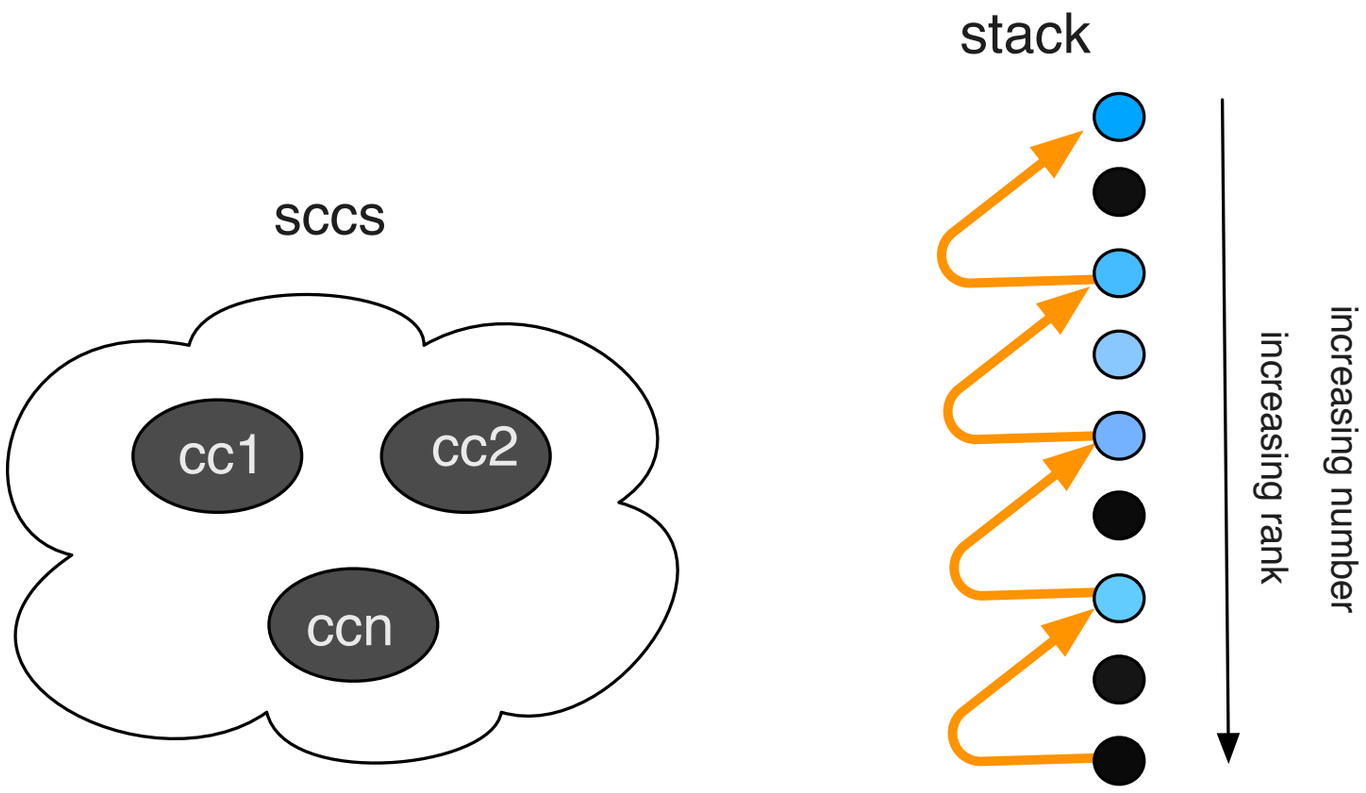
Plan of proof (1/2)

- define **reachability** in graphs and SCCs
- prove a few lemmas about positions in stacks (**ranks**)
- define **invariants** on environments
- give **pre-post conditions** for functions
- add a few intermediate **assertions** in function bodies

- avoid paths, prefer edges

Plan of proof (2/2)

- vertices have colors
  - white = unvisited - gray = being visited - black = visited
- invariant on environment



vertex in stack reaches all vertices with higher rank

Invariants

```
type env = {ghost blacks: set vertex; ghost grays: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}
```

Invariants

```
type env = {ghost blacks: set vertex; ghost grays: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}

predicate wf_color (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in
  subset (union g b) vertices /\
  inter b g == empty /\
  elements s == union g (diff b (set_of ccs)) /\
  subset (set_of ccs) b
```

Invariants

```
type env = {ghost blacks: set vertex; ghost grays: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}

predicate wf_color (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in
  subset (union g b) vertices /\
  inter b g == empty /\
  elements s == union g (diff b (set_of ccs)) /\
  subset (set_of ccs) b

predicate wf_num (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in
  (forall x. -1 <= f[x] < n <= max_int() \/ f[x] = max_int())  /\
  n = cardinal (union g b) /\
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)
```

Invariants

```
type env = {ghost blacks: set vertex; ghost grays: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}

predicate wf_color (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in
  subset (union g b) vertices /\
  inter b g == empty /\
  elements s == union g (diff b (set_of ccs)) /\
  subset (set_of ccs) b

predicate wf_num (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in
  (forall x. -1 <= f[x] < n <= max_int() \/ f[x] = max_int())  /\
  n = cardinal (union g b) /\
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)

predicate no_black_to_white (blacks grays: set vertex) =
  forall x x'. edge x x' -> mem x blacks -> mem x' (union blacks grays)
```

Invariants

```
type env = {ghost blacks: set vertex; ghost grays: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}

predicate wf_color (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in
  subset (union g b) vertices /\
  inter b g == empty /\
  elements s == union g (diff b (set_of ccs)) /\
  subset (set_of ccs) b

predicate wf_num (e: env) =
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in
  (forall x. -1 <= f[x] < n <= max_int() \/ f[x] = max_int())  /\
  n = cardinal (union g b) /\
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)

predicate no_black_to_white (blacks grays: set vertex) =
  forall x x'. edge x x' -> mem x blacks -> mem x' (union blacks grays)

predicate wf_env (e: env) = let {stack = s; blacks = b; grays = g} = e in
  wf_color e /\ wf_num e /\
  no_black_to_white b g /\ simplelist s /\
  (forall x y. mem x g -> lmem y s -> rank x s <= rank y s -> reachable x y) /\
  (forall y. lmem y s -> exists x. mem x g /\ rank x s <= rank y s /\ reachable y x)
```

Pre/Post-conditions

```
let rec dfs1 x e  =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
```

Pre/Post-conditions

```
let rec dfs1 x e  =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
```

Pre/Post-conditions

```
let rec dfs1 x e  =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
```

```
(* monotony *)
returns {(_, e') -> subenv e e'}
```

Pre/Post-conditions

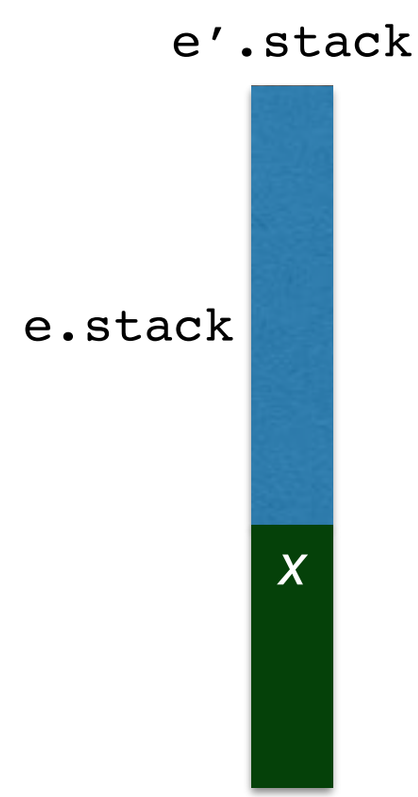
```
let rec dfs1 x e  =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
```

```
(* monotony *)
returns {(_, e') -> subenv e e'}
```

$e.sccs \subseteq e'.sccs$

$e.blacks \subseteq e'.blacks$

$e.grays = e'.grays$



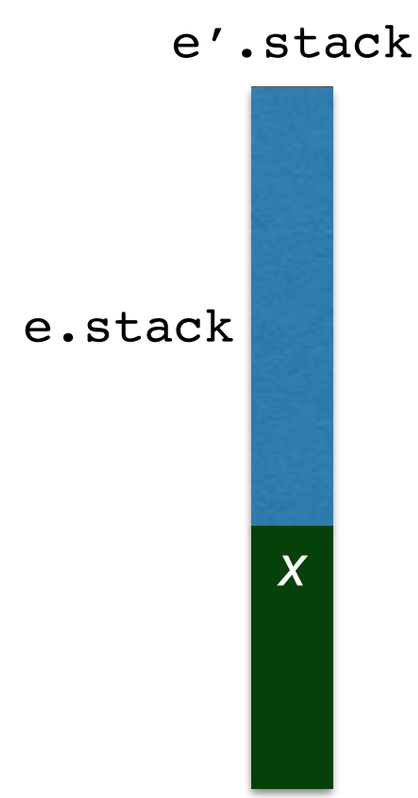
Pre/Post-conditions

```
let rec dfs1 x e  =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
(* post-cond *)
returns {(n, e') -> n <= e'.num[x]} (* PC1 *)
returns {(n, e') -> n = max_int() \/ num_of_reachable n x e'} (* PC2 *)
returns {(n, e') -> forall y. xedge_to e'.stack e.stack y -> n <= e'.num[y]} (* PC3 *)
returns {(_, e') -> mem x e'.blacks} (* PC4 *)
(* monotony *)
returns {(_, e') -> subenv e e'}
```

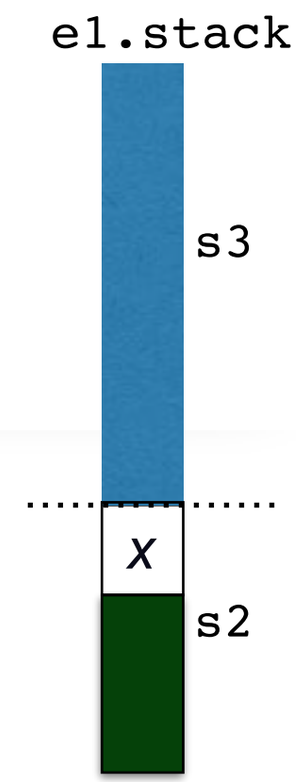
$e.sccs \subseteq e'.sccs$

$e.blacks \subseteq e'.blacks$

$e.grays = e'.grays$



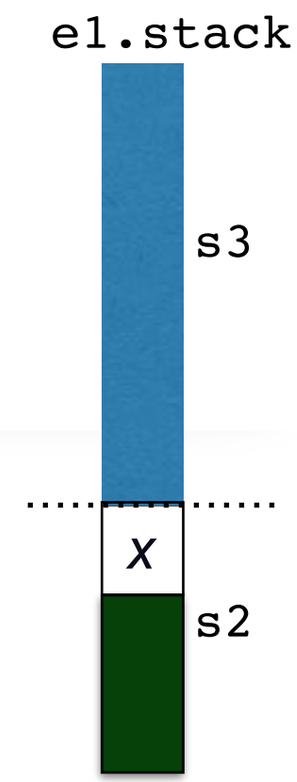
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in


if n1 < n then begin

  (n1, add_blacks x e1) end
else  begin


(max_int(), {blacks = add x e1.blacks; grays = e.grays;
   stack = s3; sccs = add (elements s2) e1.sccs;
   sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

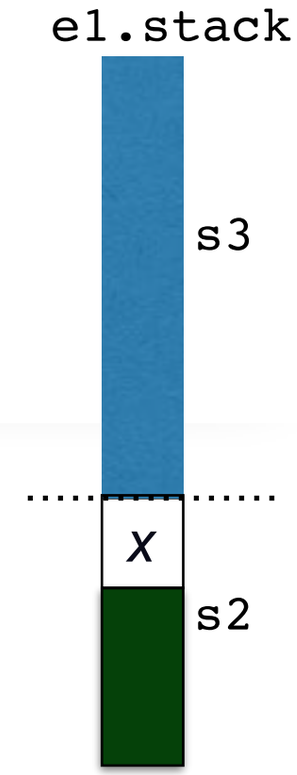
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin

  (n1, add_blacks x e1) end
else  begin

(max_int(), {blacks = add x e1.blacks; grays = e.grays;
   stack = s3; sccs = add (elements s2) e1.sccs;
   sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

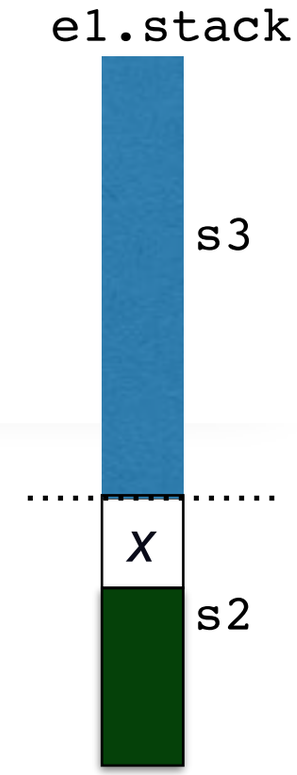
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else  begin

(max_int(), {blacks = add x e1.blacks; grays = e.grays;
   stack = s3; sccs = add (elements s2) e1.sccs;
   sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

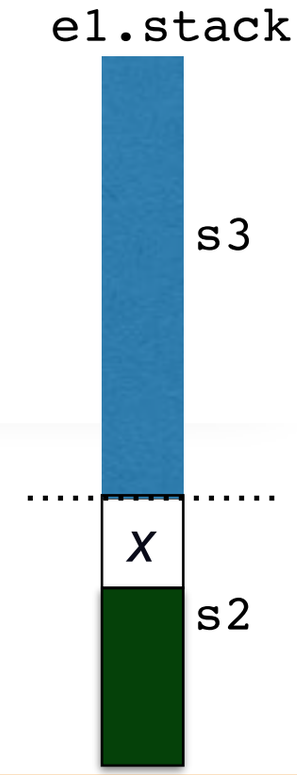
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else  begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
     stack = s3; sccs = add (elements s2) e1.sccs;
     sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

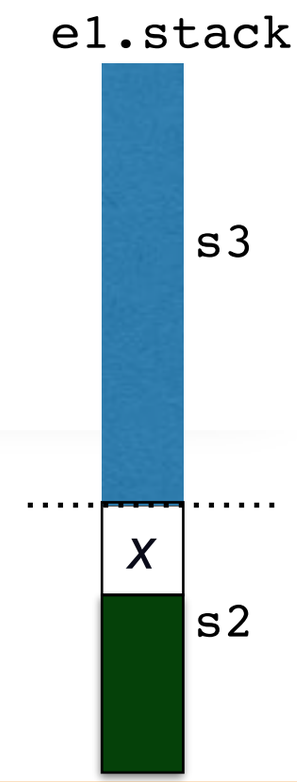
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else  begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
     stack = s3; sccs = add (elements s2) e1.sccs;
     sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

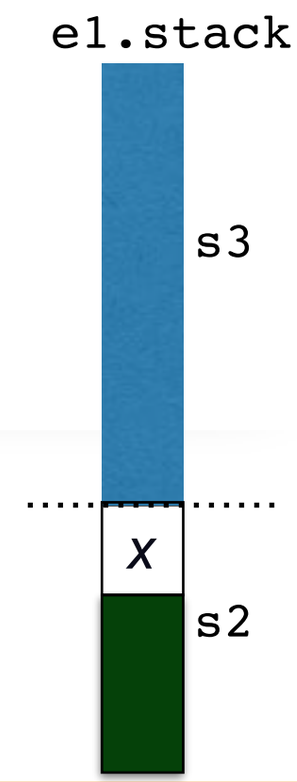
Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else  begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
     stack = s3; sccs = add (elements s2) e1.sccs;
     sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

Assertions




```
let n = e.sn in
let (n1, e1) =
    dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subscc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else  begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
     stack = s3; sccs = add (elements s2) e1.sccs;
     sn = e1.sn; num = set_max_int s2 e1.num}) end
```

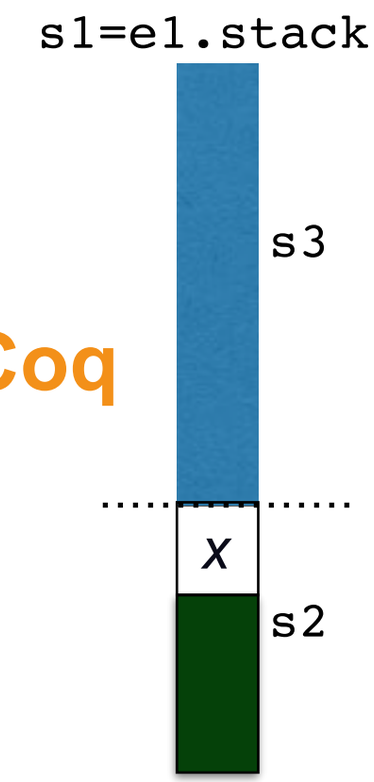
Coq

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

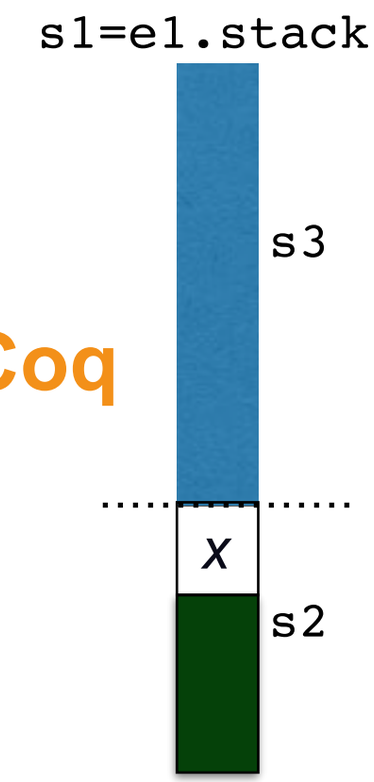


- proof by contradiction: $\exists y,\ \ \text{in\_same\_scc}\ y\ x\ \wedge\ y \notin s2$
- $\exists x' y',\ \ \text{reachable}\ x\ x'\ \wedge\ \text{edge}\ x'\ y'\ \wedge\ \text{reachable}\ y'\ y\ \wedge\ x' \in s2\ \wedge\ y' \notin s2$

Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

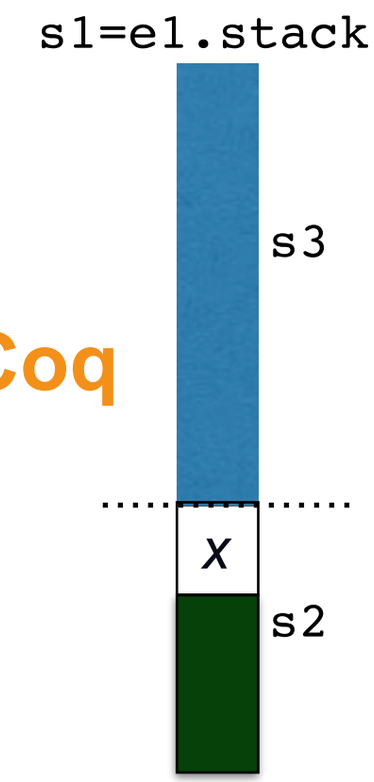


- proof by contradiction: $\exists y,\ \ \text{in\_same\_scc}\ y\ x\ \wedge\ y \notin s2$
- $\exists x' y',\ \ \text{reachable}\ x\ x'\ \wedge\ \text{edge}\ x'\ y'\ \wedge\ \text{reachable}\ y'\ y\ \wedge\ x' \in s2\ \wedge\ y' \notin s2$
- 3 cases:

Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq



- proof by contradiction: $\exists y,\ \ \text{in\_same\_scc}\ y\ x\ \wedge\ y \notin s2$
- $\exists x' y',\ \ \text{reachable}\ x\ x'\ \wedge\ \text{edge}\ x'\ y'\ \wedge\ \text{reachable}\ y'\ y\ \wedge\ x' \in s2\ \wedge\ y' \notin s2$
- 3 cases:

[1] y' is white

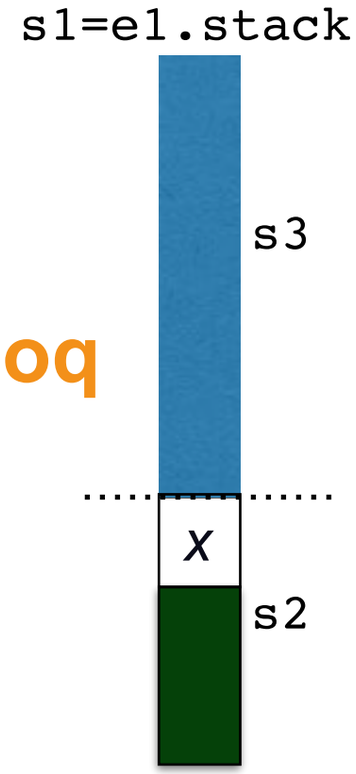
$x' = x$ then $y' \in \text{successors}\ x$ → y' is black

$x' \neq x$ then x' is black → $\neg$ no_black_to_white $b1\ g1$

Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq



- proof by contradiction: $\exists y,$ in_same_scc $y\ x\ \wedge\ y \notin s2$
- $\exists x' y',$ reachable $x\ x'\ \wedge$ edge $x'\ y'\ \wedge$ reachable $y'\ y\ \wedge\ x' \in s2\ \wedge\ y' \notin s2$
- 3 cases:

[1] y' is white

$x' = x$ then $y' \in$ successors x → y' is black

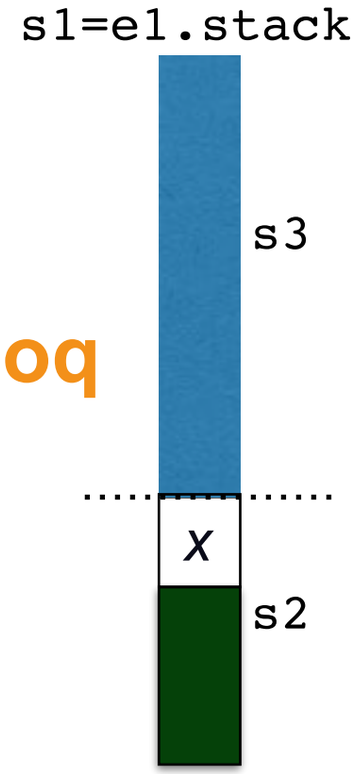
$x' \neq x$ then x' is black → $\neg$ no_black_to_white $b1\ g1$

[2] $y' \in$ e1.sccs then in_same_scc $y'\ x$ → x is black

Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq



- proof by contradiction: $\exists y,\ \ \text{in\_same\_scc}\ y\ x\ \wedge\ y \notin s2$
- $\exists x' y',\ \ \text{reachable}\ x\ x'\ \wedge\ \text{edge}\ x'\ y'\ \wedge\ \text{reachable}\ y'\ y\ \wedge\ x' \in s2\ \wedge\ y' \notin s2$
- 3 cases:

[1] y' is white

$x' = x$ then $y' \in \text{successors}\ x$ → y' is black

$x' \neq x$ then x' is black → $\neg$ no_black_to_white $b1\ g1$

[2] $y' \in \text{e1.sccs}$ then $\text{in\_same\_scc}\ y'\ x$ → x is black

[3] $y' \in s3$ → $\text{rank}\ y'\ s1\ <\ \text{rank}\ x\ s1$ → $\text{e1.num}[y'] < \text{e1.num}[x] = \text{e.num}[x] = n$

$x' = x$ then $y' \in \text{successors}\ x$ → $n1\ \leq\ \text{e1.num}[y']$

$x' \neq x$ then $\text{xedge\_to}\ s1\ (\text{Cons}\ x\ s3)\ y'$ →

Proof stats

provers	Alt-Ergo	CVC3	CVC4	Coq	E-prover	Spass	Yices	Z3	all	#VC	#PO
38 lemmas	2.35	0.23	5.79		0.66	0.75	0.21		9.99	77	38
split	0.09	0.2							0.29	6	6
add_stack_incr	0.01								0.01	1	1
add_blacks	0.01								0.01	1	1
set_max_int	0.02								0.02	1	1
dfs1	53.52	12.88	36.39	3.06	28.06			9.01	142.92	218	24
dfs	4.6	0.23	11.63					0.31	16.77	51	35
tarjan	0.44								0.44	16	6
total	61.04	13.54	53.81	3.06	28.72	0.75	0.21	9.32	170.45	371	112

[http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html]

Other systems

Coq / ssreflect

[cyril cohen, laurent théry, JJL]

- port in 1 week
- graphs and finite sets already in mathematical components
- problems with termination (hacky & higher-order)
- 920 lines

[http://github.com/CohenCyril/tarjan]

Isabelle / HOL

[stephan merz]

- port in 1 month
- use many strategies (metis, blast, sledgehammer)
- still problems with proving termination
- 31 pages

[http://jeanjacqueslevy.net/why3/graph/abs/scct/isa/Tarjan.pdf]

F*

[kenji maillard, catalin hritcu]

- start discuss with them
- Z3 single automatic prover
- ??

Conclusion

Future work

- library for formal proofs on graphs
- other graph algorithms
- **beyond** graphs …
- teaching formal methods on **test cases**
- **imperative** programs

[http://jeanjacqueslevy.net/why3]