Simple proofs
for
simple programs

Jean-Jacques Lévy
Iscas - Inria

Journées LAC
Chambéry, 20-11-14

Plan

- Why3
- demo with merge sort
- and dfs for graphs
- conclusions

Goal

Write elegant proofs
for elegant programs

\+ training in program proofs checked by computers

\+ useful to teach algorithms

.. with Chen Ran (Iscas)

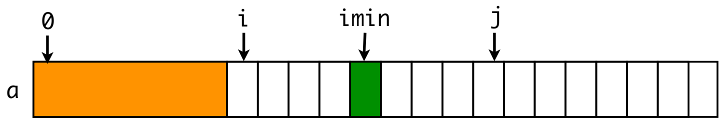
Why3

Why3

- 3rd release of system Why `http://why3.lri.fr`
  LRI (orsay) + Inria + Cnrs [Filliâtre, Paskevich, Marché…]
- small Pascal-like imperative programming language
  [with ML syntax !!]
- invariants + assertions in Hoare logic
  [+ recursive functions, inductive datatypes, inductive predicates]
- interfaces with modern automatic provers
  [**alt-ergo**, cvc3, cvc4, eprover, gappa, simplify, spass, yices, **z3**, …]
- interfaces with interactive proof assistants
  [**coq**, pvs, isabelle]

MLW programming language

```
let swap (a: array int) (i: int) (j: int) =
 let v = a[i] in
  a[i] <- a[j];
  a[j] <- v

let selection_sort (a: array int) =
  for i = 0 to length a - 1 do
    let imin = ref i in
    for j = i + 1 to length a - 1 do
      if a[j] < a[!imin] then imin := j
    done;
    swap a !imin i
  done
```



Why3 theories

- theories about arrays

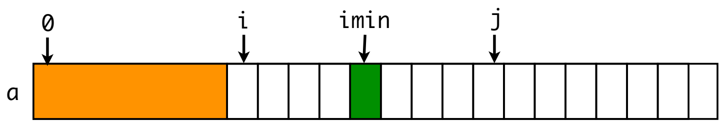
```
let swap (a: array int) (i: int) (j: int) =
  requires { 0 <= i < length a /\ 0 <= j < length a }
  ensures { exchange (old a) a i j }
 let v = a[i] in
  a[i] <- a[j];
  a[j] <- v
```

(see the why3 libraries)

http://why3.lri.fr

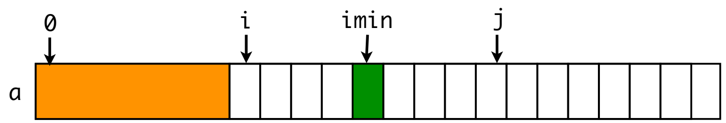
Hoare logic

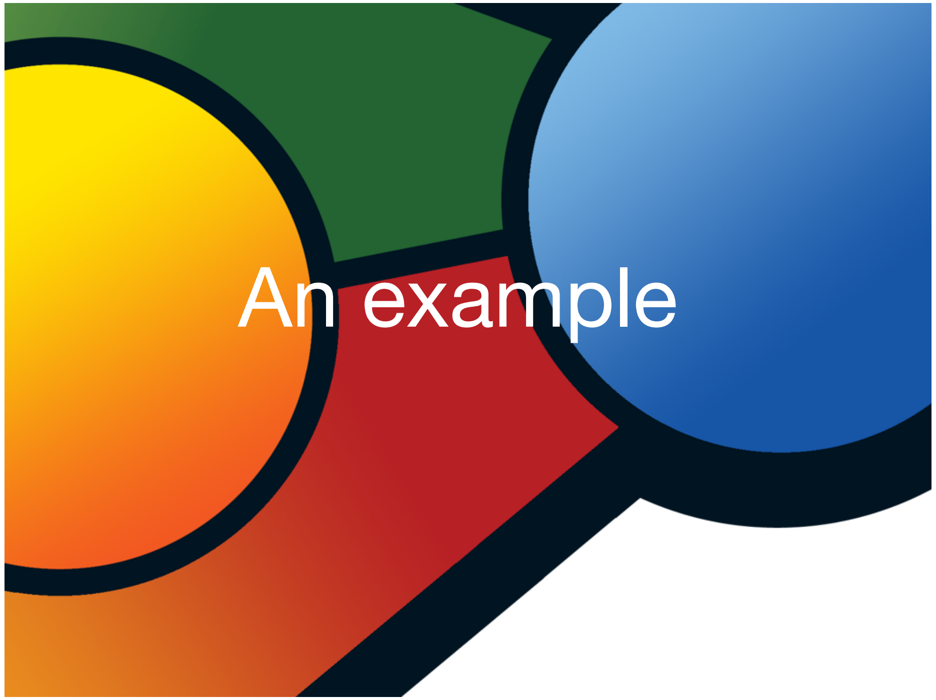
```
let swap (a: array int) (i: int) (j: int) =
 let v = a[i] in
  a[i] <- a[j];
  a[j] <- v

let selection_sort (a: array int) =
   for i = 0 to length a - 1 do
    let imin = ref i in
    for j = i + 1 to length a - 1 do
      invariant { i <= !imin < j }
      invariant { forall k: int. i <= k < j -> a[!imin] <= a[k] }
      if a[j] < a[!imin] then imin := j
    done;
    swap a !min i
  done
```



Full program

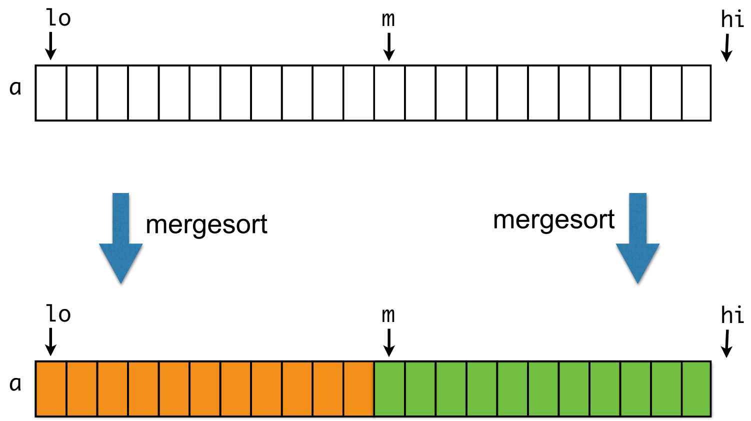
```
  let selection_sort (a: array int) =
    ensures { sorted a /\ permut (old a) a }
'L:
    for i = 0 to length a - 1 do
      invariant { sorted_sub a 0 i /\ permut (at a 'L) a}
      invariant { forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }
      let imin = ref i in
      for j = i + 1 to length a - 1 do
        invariant { i <= !imin < j }
        invariant { forall k: int. i <= k < j -> a[!imin] <= a[k] }
        if a[j] < a[!imin] then imin := j
      done;
      swap a !imin i ;
    done
```



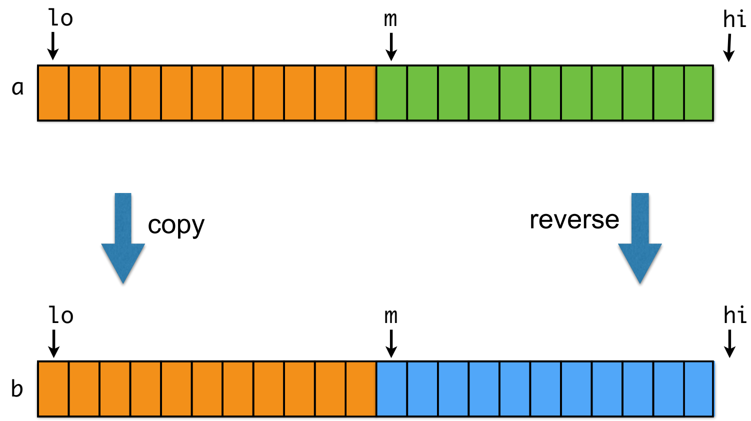


An example

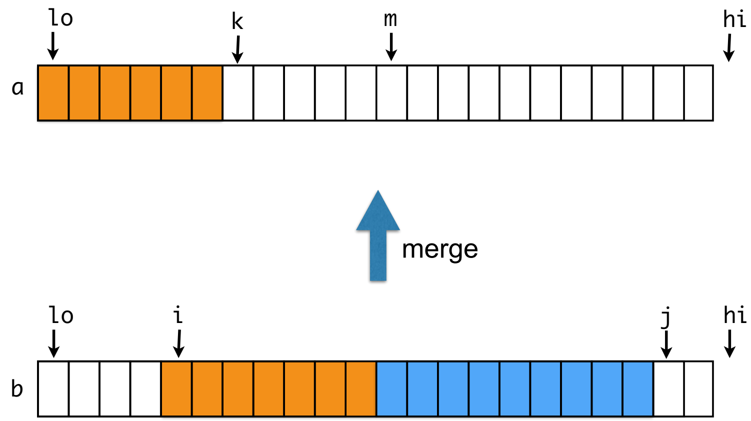
Mergesort (1/3)



Mergesort (2/3)



Mergesort (3/3)



Full program (1/2)

```
let rec mergesort1 (a b: array int) (lo hi: int) =
  requires {Array.length a = Array.length b /\
            0 <= lo <= (Array.length a) /\ 0 <= hi <= (Array.length a) }
  ensures { sorted_sub a lo hi /\ modified_inside (old a) a lo hi }
 if lo + 1 < hi then
 let m = div (lo+hi) 2 in
  assert{ lo < m < hi};
  mergesort1 a b lo m;
'L2: mergesort1 a b m hi;
  assert { array_eq_sub (at a 'L2) a lo m};
  for i = lo to m-1 do
    invariant { array_eq_sub b a lo i}
    b[i] <- a[i]
    done;
  assert{ array_eq_sub a b lo m};
  assert{ sorted_sub b lo m};
  for j = m to hi-1 do
    invariant { array_eq_sub_rev_offset b a m j (hi - j)}
    invariant { array_eq_sub a b lo m}
    b[j] <- a[m + hi - 1 - j]
    done;
  assert{ array_eq_sub a b lo m};
  assert{ sorted_sub b lo m};
  assert{ array_eq_sub_rev_offset b a m hi 0};
  assert{ dsorted_sub b m hi};
```

Full program (2/2)

```
'L4: let i = ref lo in
    let j = ref hi in
    for k = lo to hi-1 do
      invariant{ lo <= !i < hi /\ lo <= !j <= hi}
      invariant{ k = !i + hi - !j}
      invariant{ sorted_sub a lo k }
      invariant{ forall k1 k2: int. lo <= k1 < k -> !i <= k2 < !j -> a[k1] <= b[k2] }
      invariant{ bitonic b !i !j }
      invariant{ modified_inside a (at a 'L4) lo hi }
      assert { !i < !j };
      if b[!i] < b[!j - 1] then
        begin a[k] <- b[!i]; i := !i + 1 end
      else
        begin j := !j - 1; a[k] <- b[!j] end
    done

  let mergesort (a: array int) =
    ensures { sorted a }
  let n = Array.length a in
  let b = Array.make n 0 in
    mergesort1 a b 0 n
```

Full program (logic 1/2)

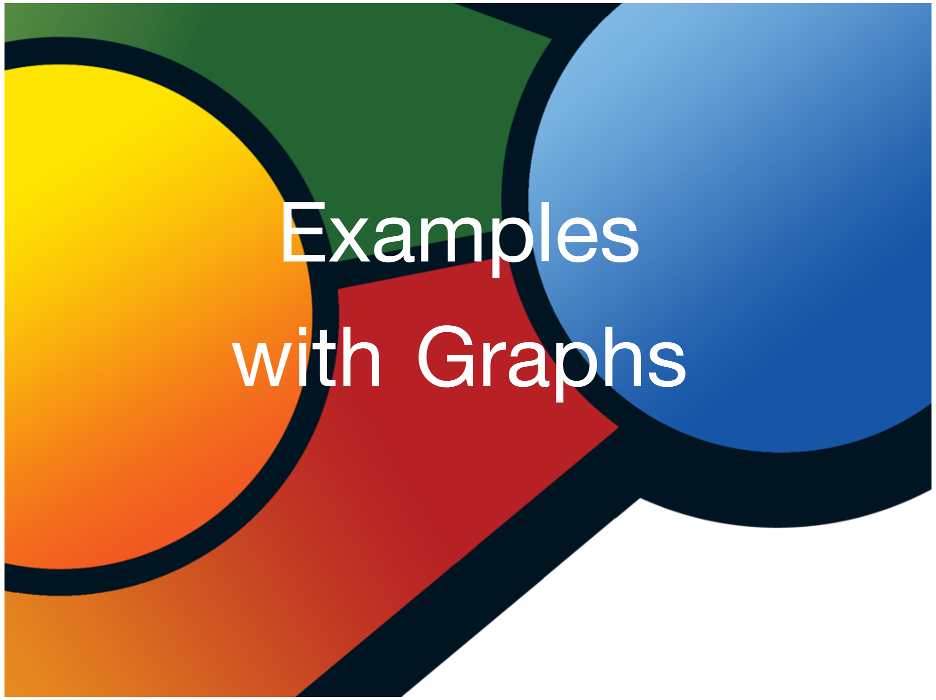
```
module MergeSort

  use import int.Int
  use import int.EuclideanDivision
  use import int.Div2
  use import ref.Ref
  use import array.Array
  use import array.ArraySorted
  use import array.ArrayPermut
  use import array.ArrayEq
  use map.Map as M
  clone map.MapSorted as N with type elt = int, predicate le = (<=)

  predicate map_eq_sub_rev_offset (a1 a2: M.map int int) (l u: int) (offset: int) =
    forall i: int. l <= i < u -> M.get a1 i = M.get a2 (offset + l + u - 1 - i)

  predicate array_eq_sub_rev_offset (a1 a2: array int) (l u: int) (offset: int) =
     map_eq_sub_rev_offset a1.elts a2.elts l u offset

  predicate map_dsorted_sub (a: M.map int int) (l u: int) =
    forall i1 i2 : int. l <= i1 <= i2 < u -> M.get a i2 <= M.get a i1

  predicate dsorted_sub (a: array int) (l u: int) =
    map_dsorted_sub a.elts l u
```

Full program (logic 2/2)

```
predicate map_bitonic_sub (a: M.map int int) (l u: int) =  l < u ->
  exists i: int. l <= i <= u /\ N.sorted_sub a l i /\ map_dsorted_sub a i u

predicate bitonic (a: array int) (l u: int) =
  map_bitonic_sub a.elts l u

lemma map_bitonic_incr : forall a: M.map int int, l u: int.
   map_bitonic_sub a l u -> map_bitonic_sub a (l+1) u

lemma map_bitonic_decr : forall a: M.map int int, l u: int.
   map_bitonic_sub a l u -> map_bitonic_sub a l (u-1)

predicate modified_inside (a1 a2: array int) (l u: int) =
  (Array.length a1 = Array.length a2) /\
  array_eq_sub a1 a2 0 l /\ array_eq_sub a1 a2 u (Array.length a1)
```

Coq files

```
Lemma sorted_sub_weakening: forall (a:(map.Map.map Z Z)) (l:Z) (u:Z) (l':Z)(u':Z),
  (l <= l')%Z -> (u' <= u)%Z -> sorted_sub2 a l u -> sorted_sub2 a l' u'.
Proof.
move=> a l u l' u' Hl_le_l' Hu'_le_u Hlu_sorted.
unfold sorted_sub2 =>  i1 i2 [Hl'_le_i1 Hi1_le_i2_lt_u'].
apply Hlu_sorted.
by omega.
Qed.

Lemma dsorted_sub_weakening: forall (a:(map.Map.map Z Z)) (l:Z) (u:Z) (l':Z) (u':Z),
  (l <= l')%Z -> (u' <= u)%Z -> map_dsorted_sub a l u -> map_dsorted_sub a l' u'.
Proof.
move=> a l u l' u' Hl_le_l' Hu'_le_u Hlu_dsorted.
unfold map_dsorted_sub =>  i1 i2 [Hl'_le_i1 Hi1_le_i2_lt_u].
apply Hlu_dsorted.
by omega.
Qed.

Lemma sorted_sub_diag:  forall (a:(map.Map.map Z Z)) (l:Z),
  sorted_sub2 a l l.
Proof.
move=> a l.
unfold sorted_sub2 => i1 i2 [Hl_le_i1 Hi1_le_i2_lt_l].
have Hl_lt_l: (l < l)%Z.
- by omega.
by apply Zlt_irrefl in Hl_lt_l.
Qed.
```

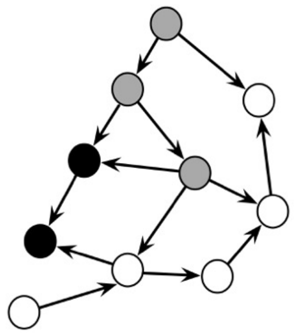


Coq files

```
(**| Why3 goal *)
Theorem map_bitonic_incr : forall (a:(map.Map.map Z Z)) (l:Z) (u:Z),
  (map_bitonic_sub a l u) -> (map_bitonic_sub a (l + 1%Z)%Z u).

Proof.
move=> a l u Hlu_bitonic.
unfold map_bitonic_sub => Hl1_lt_u.
unfold map_bitonic_sub in Hlu_bitonic.
have Hl_lt_u: (l < u)%Z.
- by omega.
apply Hlu_bitonic in Hl_lt_u.
move: Hl_lt_u=> [j  [Hl_le_j_le_u [Hlj_sorted Hju_dsorted]]].
move: Hl_le_j_le_u => [Hl_le_j Hj_le_u].
apply (Zle_lt_or_eq l j) in Hl_le_j.
case: Hl_le_j => [Hl_lt_j | Hl_eq_j].
- exists j.
  split.
  + by omega.
  + split.
    - apply (sorted_sub_weakening a l j).
      + by apply (Z.le_succ_diag_r).
      + reflexivity.
      + exact Hlj_sorted.
    - exact Hju_dsorted.
- exists (l+1)%Z.
  split.
  + by omega.
  + split.
    - by apply sorted_sub_diag.
    - apply (dsorted_sub_weakening a l u).
      + by omega.
      + by omega.
      + rewrite Hl_eq_j.
        exact Hju_dsorted.
Qed.
```

Depth-first search in graphs (1/4)

- reachability [the ‘white path theorem’]
- non white-to-black edges in undirected graphs

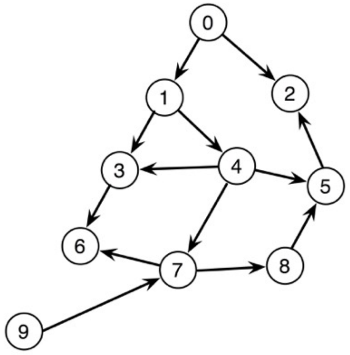


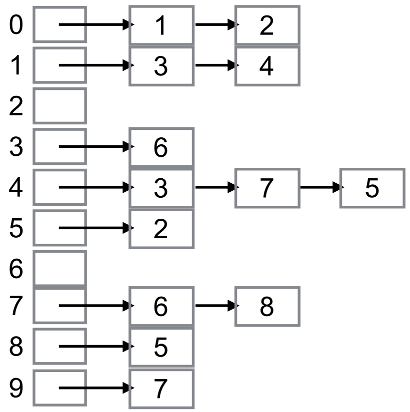
- acyclicity test
- articulation points
- strongly connected components

  Kosaraju, Tarjan

Depth-first search in graphs (2/4)

- representation as array of lists of successors

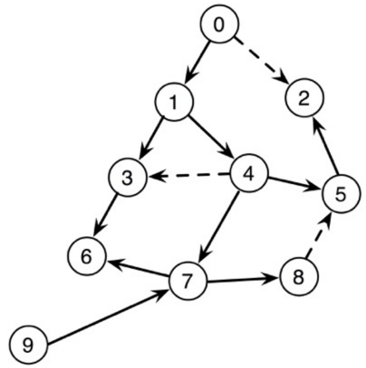




Depth-first search in graphs (4/4)

- spanning trees = call graph of DFS in `mlw`

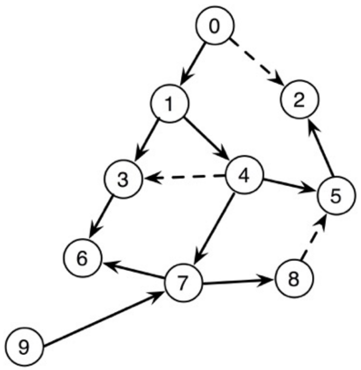
```
let rec dfs (g: graph) (x: int) (c: array color) =
  c[x] <- GRAY;
  let sons = ref (g[x]) in
  while !sons <> Nil do
    match !sons with
    | Nil -> ()
    | Cons y sons' ->
      if c[y] = WHITE then dfs g y c;
      sons := sons'
     end;
    done;
    c[x] <- BLACK

let dfs_main (g: graph) =
   let n = length (g) in
   let c = Array.make n WHITE in
   for x = 0 to n - 1 do
     if c[x] = WHITE then
       dfs g x c
   done
```



Depth-first search in graphs (3/4)

- spanning trees = call graph of DFS

```
let rec dfs (g: graph) (x: int) (c: array color) =
  c[x] <- GRAY;
  let sons = ref (g[x]) in
  FORALL y in sons do
      if c[y] = WHITE then dfs g y c;
   done;
   c[x] <- BLACK

let dfs_main (g: graph) =
   let n = length (g) in
   let c = Array.make n WHITE in
   for x = 0 to n - 1 do
     if c[x] = WHITE then
       dfs g x c
   done
```



Undirected graph: no W2B arc (1/3)

```
function order (g: graph) : int = length g
predicate vertex (g: graph) (x: int) = 0 <= x < order g
predicate out (g: graph) (x: int)  =
  forall y: int. vertex g x -> mem y g[x] -> vertex g y
predicate g_edge (g: graph) (x: int) =
  forall y : int. (vertex g x /\ mem y g[x]) <-> edge x y
predicate double (g: graph) (x: int) =
  forall y: int. vertex g x -> mem y g[x] -> mem x g[y]
predicate wf (g: graph) =
  forall x: int. vertex g x -> out g x /\ g_edge g x /\ double g x


type color = WHITE | GRAY | BLACK

predicate noW2Bedge (g: graph) (c: array color) =
   forall x y: int. vertex g x -> vertex g y ->
      c[x] = WHITE -> c[y] = BLACK -> not mem y g[x]

predicate white_monotony (g: graph) (c1 c2: array color) =
   forall x: int. vertex g x -> c2[x] = WHITE -> c1[x] = WHITE
```

Undirected graph: no W2B arc (2/3)

```
let rec dfs (g: graph) (x: int) (c: array color) =
  requires{ wf g /\ vertex g x /\ length c = order g }
  requires{ noW2Bedge g c }
  ensures { (old c)[x] = WHITE -> c[x] <> WHITE }
  ensures { white_monotony g (old c) c }
  ensures { noW2Bedge g c }
'L:
  c[x] <- GRAY;
  let sons = ref (g[x]) in
  while !sons <> Nil do
    invariant { white_monotony g (at c 'L) c }
    invariant { forall y: int. mem y !sons -> edge x y }
    invariant { forall y: int. edge x y -> c[y] = WHITE -> mem y !sons }
    invariant { noW2Bedge g c }
    match !sons with
    | Nil -> ()
    | Cons y sons' ->
      if c[y] = WHITE then dfs g y c;
      sons := sons'
     end;
    done;
    c[x] <- BLACK
```

Undirected graph: no W2B arc (3/3)

```
let dfs_main (g: graph) =
   requires { wf g }
   let n = length (g) in
   let c = make n WHITE in
   for x = 0 to n - 1 do
     invariant { noW2Bedge g c }
     if c[x] = WHITE then
       dfs g x c
   done
```

- why these invariants ?
- are they natural ?
- can be found automatically ?

White paths (1/2)

- if white path between x and y, then `dfs(x)` flips y to black

```
let rec dfs (g: graph) (x: int) (c: array color) =
  requires {wf g /\ vertex g x /\ Array.length c = order g}
  requires {c[x] = WHITE}
  ensures {white_monotony g (old c) c}
  ensures {whitepath_flip_whitepath x g (old c) c} (*new*)
  ensures {node_flip_whitepath x g (old c) c}
  ensures {whitepath_node_flip x g (old c) c}

'L0:
      c[x] <- GRAY;
      assert {forall y z: int, l: list int. mem y g[x] ->
              whitepath y l z g c -> whitepath x (Cons x l) z g (at c 'L0)};
'L:
```

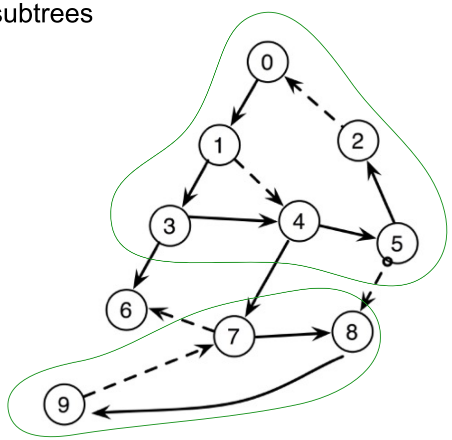
White paths (2/2)

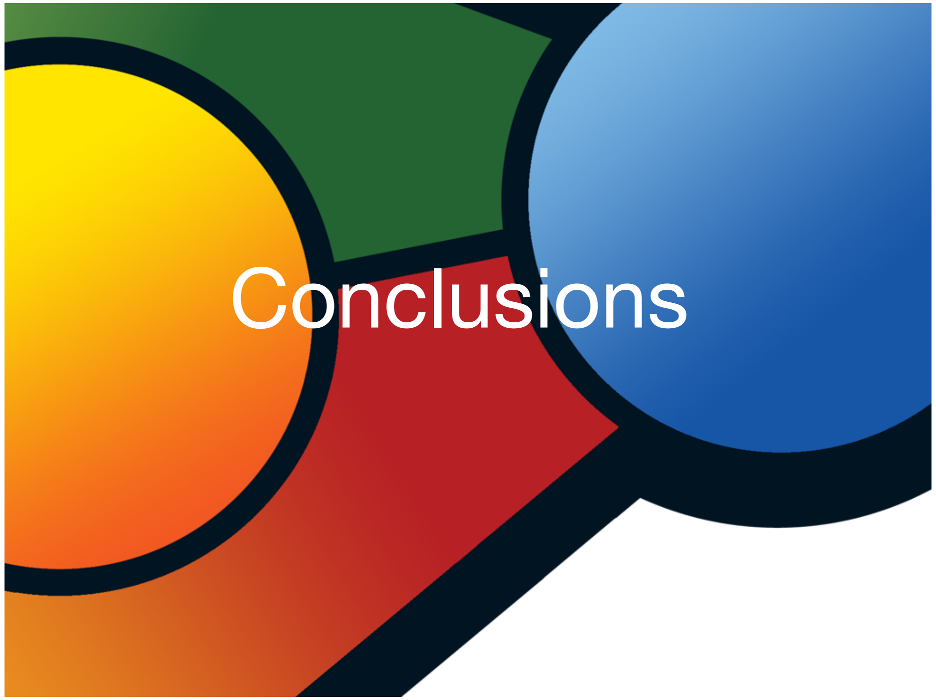
ghost variable

```
'L:
      let sons = ref (g[x]) in
      let ghost lv = ref Nil in
        while  ( !sons <> Nil) do
        invariant {(reverse !lv) ++ !sons = g[x]}
        invariant {white_monotony g (at c 'L) c}
        invariant {whitepath_monotony g (at c 'L) c}
        invariant {whitepath_flip_whitepath_in_list !lv g (at c 'L) c}  (*new*)
        invariant {node_flip_whitepath_in_list !lv g (at c 'L) c}
        invariant {whitepath_in_list_node_flip !lv g (at c 'L) c}

'L1:
        match !sons with
          | Nil -> ()
          | Cons y sons' ->
              if c[y] = WHITE then begin
                dfs g y c;
                sons := sons';
                lv := Cons y !lv;
              end
        end
        done;
      c[x] <- BLACK;
      assert {node_flip_whitepath x g (at c 'L0) c}  (*new*)
```

Strongly connected components (1/2)

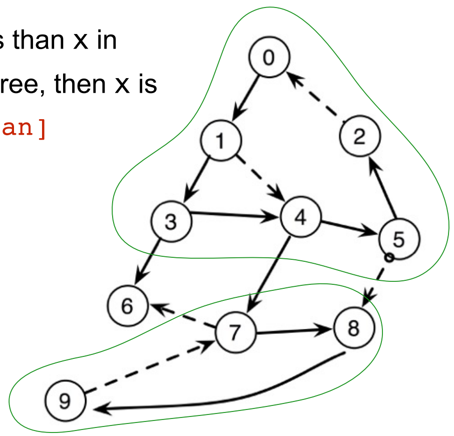
- in spanning trees, no left-to-right edge [Wengener, Pottier]
- SCC are prefixes of subtrees





Strongly connected components (2/2)

- if y connected to x by nodes less than x in post-order traversal of spanning tree, then x is connected to y [Kosaraju]
- if x cannot reach a node y less than x in pre-order traversal of spanning tree, then x is the root of its component [Tarjan]



Conclusion 1

- **Automatic** part of proof for **tedious** case analyzes
- **Interactive** proofs for the **conceptual** part of the algorithm

→ the ideal world

- From interactive part, one can call the automatic part
  - possible extensions of Why3 theories
  - but typing problems (inside Coq)

Conclusion 2

- Hoare logic prevents to write awkward denotational semantics
- Nobody cares about termination ?! 
- Explore **simple** programs about algorithms before jumping to **large** programs.
- Why3 **memory model** is naive. It's a «back-end for other systems».
- Also experimenting on **graph** algorithms and prove all algorithms in `Sedgewick`'s book.

Conclusion 3

- Why3 is **excellent** for mixing formal proofs and SMT's calls
- Still **rough** for beginners
- Concurrency ?
- Functional programs ?
- Hoare logic vs Type refinements (F* `[MSR]`)
- **Frama-C** project at french CEA extends Why3 to C programs.