# Readable proofs of DFS in graphs using Why3

jean-jacques.levy@inria.fr

PCRI, 04-12-2015

# Plan

- motivation
- dfs with white coloring
- random walk in graphs
- dfs with arbitrary coloring
- further algorithms

.. joint work (in progress) with Ran Chen

# Motivation

- learn formal proofs of programs
- never formal proofs are fully published in an article/journal
- how to publish formal proofs ?
- pretty proofs for simple algorithms
- algorithms on graphs = a good testbed
- Why3 allows mix of automatic and interactive proofs
- Coq proofs seem to me unreadable by normal human being

# Starting with white nodes

# The program

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
    forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

- a functional version with finite sets

```
let rec dfs (roots visited: set vertex): set vertex =
 if is_empty roots then visited
  else
  let x = choose roots in
  let roots' = remove x roots in
  if mem x visited then
    dfs roots' visited
  else
    let b = dfs (successors x) (add x visited) in
    dfs roots’ (union visited b)

let dfs_main (roots: set vertex) : set vertex =
  dfs roots empty
```

# The program

```
let rec dfs (roots visited: set vertex): set vertex =
 if is_empty roots then visited
  else
  let x = choose roots in
  let roots' = remove x roots in
  if mem x visited then
    dfs roots' visited
  else
    let b = dfs (successors x) (add x visited) in
    dfs roots’ (union visited b)

let dfs_main (roots: set vertex) : set vertex =
  dfs roots empty
```

- goal: result of `dfs_main` is set of vertices accessible from `roots`
- invariant: no edge from visited vertex to unvisited vertex
- postcondition: `roots` are in result of `dfs`

# The program

```
let rec dfs (roots visited: set vertex) (ghost grays: set vertex) =
 if is_empty roots then visited
  else
  let x = choose roots in
  let roots' = remove x roots in
  if mem x visited then
    dfs roots' visited grays
  else
    let b = dfs (successors x) (add x visited) (add x grays) in
    dfs roots’ (union visited b) grays

let dfs_main (roots: set vertex) : set vertex =
  dfs roots empty empty
```

- goal: result of `dfs_main` is set of vertices accessible from `roots`
- invariant: no edge from non-gray visited vertex to unvisited vertex
- postcondition: non-gray `roots` are in result of `dfs`

# The program

```
let rec dfs (roots grays blacks: set vertex) : set vertex =
 if is_empty roots then blacks
  else
  let x = choose roots in
  let roots' = remove x roots in
  if mem x (union grays blacks) then
    dfs roots' grays blacks
  else
    let b = dfs (successors x) (add x grays) blacks in
    dfs roots’ grays (add x (union blacks b))

let dfs_main (roots: set vertex) : set vertex =
  dfs roots empty empty
```

- goal: result of `dfs_main` is set of vertices accessible from `roots`
- invariant: no edge from black vertex to white vertex
- postcondition: non-gray `roots` are in result of `dfs`

# Paths

```
type vertex

constant vertices : set vertex

function successors vertex : set vertex

axiom successors_vertices : ∀x. mem x vertices → subset (successors x) vertices

predicate edge (x y : vertex) = mem x vertices ∧ mem y (successors x)
```

```
inductive path vertex (list vertex) vertex =
  | Path_empty : ∀x : vertex. path x Nil x
  | Path_cons :
      ∀x y z : vertex, l : list vertex.
      edge x y → path y l z → path x (Cons x l) z

predicate reachable (x z : vertex) = ∃l. path x l z

predicate access (r s : set vertex) =
    ∀z. mem z s → ∃x. mem x r ∧ reachable x z
```

# Paths

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)
```

```
let rec dfs r g b :
  variant{(cardinal vertices − cardinal g), cardinal r} =
  requires {subset r vertices}
  requires {subset g vertices}
  requires {no_black_to_white b g}
  ensures {subset b result}
  ensures {no_black_to_white result g}
  ensures {∀x. mem x r → ¬ mem x g → mem x result}
  ensures {access (union b r) result}

 if is_empty r then b
  else
  let x = choose r in
  let r' = remove x r in
  if mem x (union g b) then
    dfs r' g b
  else
    let b' = dfs (successors x) (add x g) b in
    dfs r' g (union b (add x b'))
```

# Paths

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)
```

```
let rec dfs r g b :
  variant{(cardinal vertices − cardinal g), cardinal r} =
  requires {subset r vertices}
  requires {subset g vertices}
  requires {no_black_to_white b g}
  ensures {subset b result}
  ensures {no_black_to_white result g}
  ensures {∀x. mem x r → ¬ mem x g → mem x result}
  ensures {access (union b r) result}

 if is_empty r then b
  else
  let x = choose r in
  let r' = remove x r in
  if mem x (union g b) then
    dfs r' g b
  else
    let b' = dfs (successors x) (add x g) b in
    assert{access (add x b) b'};
    dfs r' g (union b (add x b'))
```

# The program

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)
```

```
lemma black_to_white_path_goes_thru_gray :
    ∀g b. no_black_to_white b g →
      ∀x l z. path x l z → mem x b → ¬ mem z (union b g) →
        ∃y. L.mem y l ∧ mem y g
```

```
let dfs_main r =
  requires {subset r vertices}
  ensures {∀s. access r s ↔ subset s result}
  dfs r empty empty
```

# The program

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)
```

```
lemma black_to_white_path_goes_thru_gray :
    ∀g b. no_black_to_white b g →
      ∀x l "induction" z. path x l z → mem x b → ¬ mem z (union b g) →
        ∃y. L.mem y l ∧ mem y g
```

```
let dfs_main r =
  requires {subset r vertices}
  ensures {∀s. access r s ↔ subset s result}
  dfs r empty empty
```

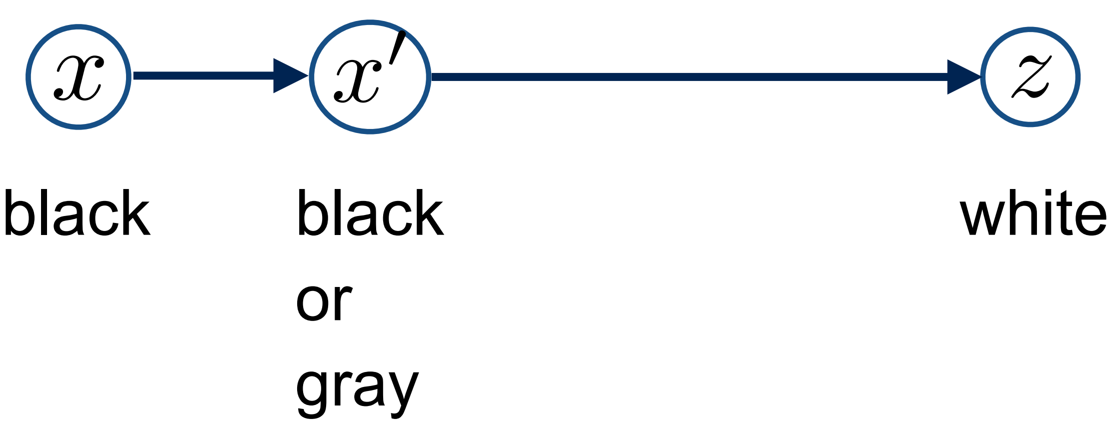
# The program

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)
```

```
lemma black_to_white_path_goes_thru_gray :
    ∀g b. no_black_to_white b g →
      ∀x l "induction" z. path x l z → mem x b → ¬ mem z (union b g) →
          ∃y. L.mem y l ∧ mem y g
```

does not work with Why3 !

although easy induction (proved with Coq)



# The program

```
Theorem black_to_white_path_goes_thru_gray : forall (grays:(set vertex))
  (blacks:(set vertex)), (no_black_to_white blacks grays) ->
  forall (x:vertex) (l:(list vertex)) (z:vertex), (path x l z) -> ((mem x
  blacks) -> ((~ (mem z (union blacks grays))) -> exists y:vertex,
  (list.Mem.mem y l) /\ (mem y grays))).

Proof.
  move=> grays blacks hnbtw x l z hxlz.
  elim: {x l z} hxlz => [x |x x' z l'].
  - move=> hxb hxnotbg.
    have hxbg: mem x (union blacks grays).
      by apply union_def1; left.
    by apply hxnotbg in hxbg.
  - move=> hxx' hx'l'z HIx'z hxb hnotzbg.
    apply (hnbtw x x') in hxb.
    apply union_def1 in hxb.
    move: hxb => [hx'b | hx'g].
    + apply HIx'z in hx'b.
      move: hx'b => [y hyl'].
      exists y; move: hyl' => [hmemyl' hzg].
      apply conj.
      - by simpl; right.
      - by [].
    + by [].
    - exists x'; apply conj.
      + apply path_mem in hx'l'z.
        simpl.
        move: hx'l'z => [hmemx'l' | eqx'z].
        - by right.
        - rewrite eqx'z in hx'g.
          have hzbg: mem z (union blacks grays).
          + by apply union_def1; right.
          by apply hnotzbg in hzbg.
      + exact hx'g.
    - exact hxx'.
Qed.
```

Starting with any color

(random walk)

# The program

```
let rec dfs (roots grays blacks others: set vertex) : set vertex =
 if is_empty roots then blacks
  else
  let x = choose roots in
  let roots' = remove x roots in
  if mem x (union grays blacks) then
    dfs roots' grays blacks others
  else
    let b = dfs (successors x) (add x grays) (add x blacks) others in
    dfs roots’ grays (union blacks b) others

let dfs_main (roots others: set vertex) : set vertex =
  dfs roots empty empty others
```

- follow previous proof
- but hacky

# Random walk

```
let rec random_search roots visited =
  if is_empty roots then
    visited
 else
   let x = choose roots in
   let roots' = remove x roots in
    if mem x visited then
      random_search roots' visited
    else
      random_search (union roots' (successors x)) (add x visited)
```

- one step of any traversal strategy
- works well with paths [dowek, munoz]

$$\textbf{predicate}\ \ white\_vertex\,(x:\ vertex)\,(v:\ set\ \ vertex) = \ \neg\,(mem\ x\ v)$$

$$\textbf{predicate}\ \ whitepath\,(x:\ vertex)\,(l:list\ vertex)\,(z:vertex)\,(v:set\ vertex) = \\ path\ x\ l\ z\ \wedge\ (\forall y.\,L.mem\ y\ l\ \rightarrow\ white\_vertex\ y\ v)\ \wedge\ white\_vertex\ z\ v$$

# Random walk

```
let rec random_search roots visited
  variant {(cardinal vertices - cardinal visited), (cardinal roots)} =
  requires {subset roots vertices }
  requires {subset visited vertices }
  ensures {subset visited result}
  ensures {forall z. mem z (diff result visited) -> exists x l. mem x roots /\ whitepath x l z visited }

  if is_empty roots then
     visited
  else
    let x = choose roots in
    let roots' = remove x roots in
     if mem x visited then
       random_search roots' visited
     else
       let r = random_search (union roots' (successors x)) (add x visited) in
       (*----------- nodeflip_whitepath -----------*)
       (* case 1: nodeflip z visited r /\ z = x *)
          assert {forall z. z = x -> whitepath x Nil z visited};
       (* case 2: nodeflip z visited r /\ z <> x *)
          assert {forall z. mem z (diff r (add x visited)) ->
            (exists y l. mem y roots' /\ whitepath y l z (add x visited))
                  \/
            (exists y l. edge x y /\ whitepath y l z (add x visited)) };
       r
```

# Random walk

- with 3 lemmas (proved in Why3)

```
lemma abc :
  forall z x:'a, r v. mem z (diff r v) -> z = x \/ mem z (diff r  (add x v))

lemma whitereachable1 :
  forall x y l z v. whitepath y l z (add x v) -> whitepath y l z v

lemma whitereachable2 :
  forall x y l z v. not (mem x v) -> whitepath y l z v -> edge x y -> whitepath x (Cons x l) z v
```

```
axiom H4 : subset o1 vertices

axiom H5 : subset o vertices

constant r : set vertex

axiom H6 : subset o r

axiom H7 :
  forall z:vertex.
   mem1 z (diff r o) ->
    (exists x1:vertex, l:list vertex. mem1 x1 o1 /\ whitepath x1 l z o)

axiom H8 :
  forall z:vertex. z = x -> whitepath x (Nil:list vertex) z visited

axiom H9 :
  forall z:vertex.
   mem1 z (diff r (add x visited)) ->
    (exists y:vertex, l:list vertex.
      mem1 y rootsqt /\ whitepath y l z (add x visited)) \/
     (exists y:vertex, l:list vertex.
       edge x y /\ whitepath y l z (add x visited))

constant z : vertex

axiom H10 : mem1 z (diff r visited)

goal WP_parameter_random_search :
  exists x1:vertex, l:list vertex.
   mem1 x1 roots /\ whitepath x1 l z visited
```

- 1 Coq proof (final postcond)

# Random walk

- with 3 lemmas (proved in Why3)

```
lemma abc :
  forall z x:'a, r v. mem z (diff r v) -> z = x \/ mem z (diff r  (add x v))

lemma whitereachable1 :
  forall x y l z v. whitepath y l z (add x v) -> whitepath y l z v

lemma whitereachable2 :
  forall x y l z v. not (mem x v) -> whitepath y l z v -> edge x y -> whitepath x (Cons x l) z v
```

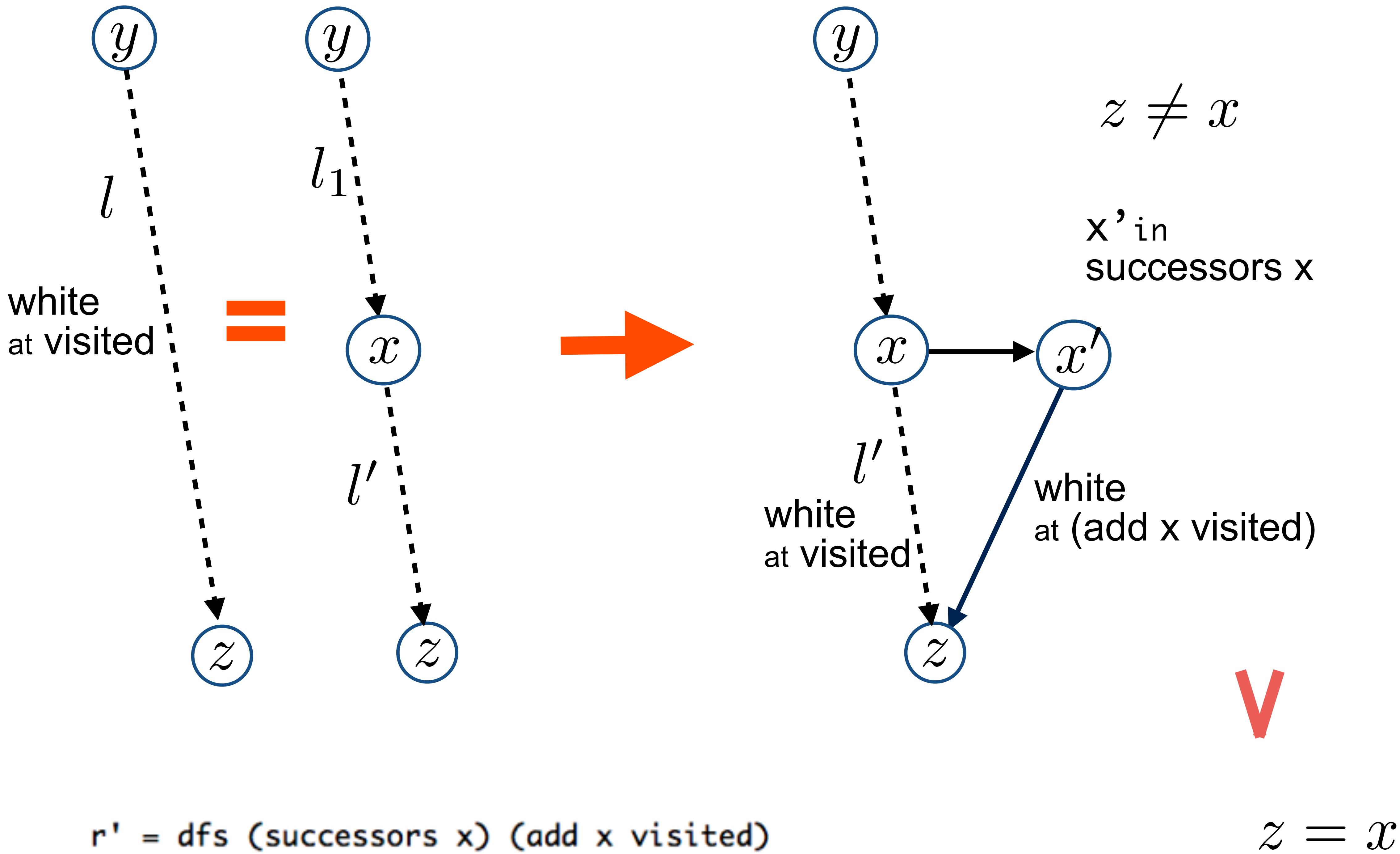
```
apply (abc _ x) in h11; move: h11 => [h11a | h11b].
- apply h9 in h11a.
  exists x; exists nil; split.
  + by apply choose_def.
  + exact h11a.
- apply h10 in h11b; move: h11b => {h10} [h10a | h10b].
  + move: h10a => [y [l [hyr hwp]]].
    exists y; exists l; split.
    - by apply remove_def1 in hyr; move: hyr => [ _ hmemyr].
    - by apply (whitereachable1 x).
  + move: h10b => [x' [l' [hyr hwp]]].
    exists x; exists (x :: l')%list; split.
    + by apply choose_def.
    + apply (whitereachable2 _ x').
     - exact h4.
     - by apply (whitereachable1 x).
     - exact hyr.
```

- 1 Coq proof (final postcond)

# Random walk

```
let rec random_search roots visited
  variant {(cardinal vertices - cardinal visited), (cardinal roots)} =
  requires {subset roots vertices }
  requires {subset visited vertices }
  ensures {subset visited result}
  ensures {forall x l z. mem x roots -> whitepath x l z visited -> mem z result }
   if is_empty roots then
     visited
   else
     let x = choose roots in
     let roots' = remove x roots in
      if mem x visited then
         random_search roots' visited
      else begin
          let r = random_search (union roots' (successors x)) (add x visited) in
          (* -------------- whitepath_nodeflip ------------ *)
          (* case 1: whitepath roots' l z /\ not (L.mem x l \/ z = x) *)
            assert {forall y l z. mem y roots' -> whitepath y l z visited -> not (L.mem x l \/ x = z)
                 -> whitepath y l z (add x visited)};
          (* case 2: whitepath roots' l z /\ (L.mem x l \/ z = x) *)
            assert {forall y l z. whitepath y l z visited -> (L.mem x l \/ z = x)
                 -> exists l'. whitepath x l' z visited};
          (* case 2-1: whitepath x l z visited /\ z = x *)
            assert {forall z. z = x -> mem z r};
          (* case 2-2: whitepath x l z visited /\ z <> x *)
          (* using lemma whitepath_whitepath_fst_not_twice *)
            assert {forall l z. z <> x -> whitepath x l z visited
               -> exists x' l'. edge x x' /\ whitepath x' l' z (add x visited) };
          r
      end
```

# DFS




```
r' = dfs (successors x) (add x visited)
```

# Random walk

- same proof for bfs or iterative dfs

- see web at `jeanjacqueslevy.net/why3`

# Starting with any color

(dfs)

# DFS

```
let rec dfs roots visited =
  if is_empty roots then
    visited
  else
    let x = choose roots in
    let roots' = remove x roots in
     if mem x visited then
        dfs roots' visited
     else
        let r' = dfs (successors x) (add x visited) in
        dfs roots' r'
```

# DFS (nodeflip — whitepath)

```
let rec dfs (roots: set vertex) (visited: set vertex): set vertex
  variant {(cardinal vertices - cardinal visited), (cardinal roots)} =
  requires {subset roots vertices }
  requires {subset visited vertices }
  ensures {subset visited result}
  ensures {subset result vertices}
  ensures {forall z. mem z (diff result visited) -> exists x l. mem x roots /\ whitepath x l z visited}

    if is_empty roots then visited
    else
      let x = choose roots in
      let roots' = remove x roots in
      if mem x visited then
        dfs roots' visited
      else begin
        assert {forall z. z = x -> whitepath x Nil z visited};
        let r' = dfs (successors x) (add x visited) in
        assert {forall z. mem z (diff r' (add x visited)) ->
              (exists y l. edge x y /\ whitepath y l z (add x visited)) };
        let r = dfs roots' r' in
        assert {forall z. mem z (diff r r') -> exists y l. mem y roots' /\ whitepath y l z r'};
        assert {forall z y l. whitepath y l z r' -> whitepath y l z (add x visited)};
        r
     end
```

- with same 3 lemmas (proved in Why3)

- 1 Coq proof (final postcond)

# DFS

- with 3 lemmas (proved in Why3)

```
lemma abc :
  forall z x:'a, r v. mem z (diff r v) -> z = x \/ mem z (diff r  (add x v))

lemma whitereachable1 :
  forall x y l z v. whitepath y l z (add x v) -> whitepath y l z v

lemma whitereachable2 :
  forall x y l z v. not (mem x v) -> whitepath y l z v -> edge x y -> whitepath x (Cons x l) z v
```

- same proof as in random walk

- 1 Coq proof (final postcond)

# DFS (whitepath — nodeflip)

```
let rec dfs (roots: set vertex) (visited: set vertex): set vertex
  variant {(cardinal vertices - cardinal visited), (cardinal roots)} =
  requires {subset roots vertices }
  requires {subset visited vertices }
  ensures {subset visited result}
  ensures {subset result vertices}
  ensures {forall z. mem z (diff result visited) -> exists x l. mem x roots /\ whitepath x l z visited}
  ensures {forall x l z. mem x roots -> whitepath x l z visited -> mem z result }

    if is_empty roots then visited
    else
      let x = choose roots in
      let roots' = remove x roots in
      if mem x visited then
        dfs roots' visited
      else
        let r' = dfs (successors x) (add x visited) in
        let r = dfs roots' r' in
        (*-------- nodeflip_whitepath ---------------------------*)
```

- both postconds

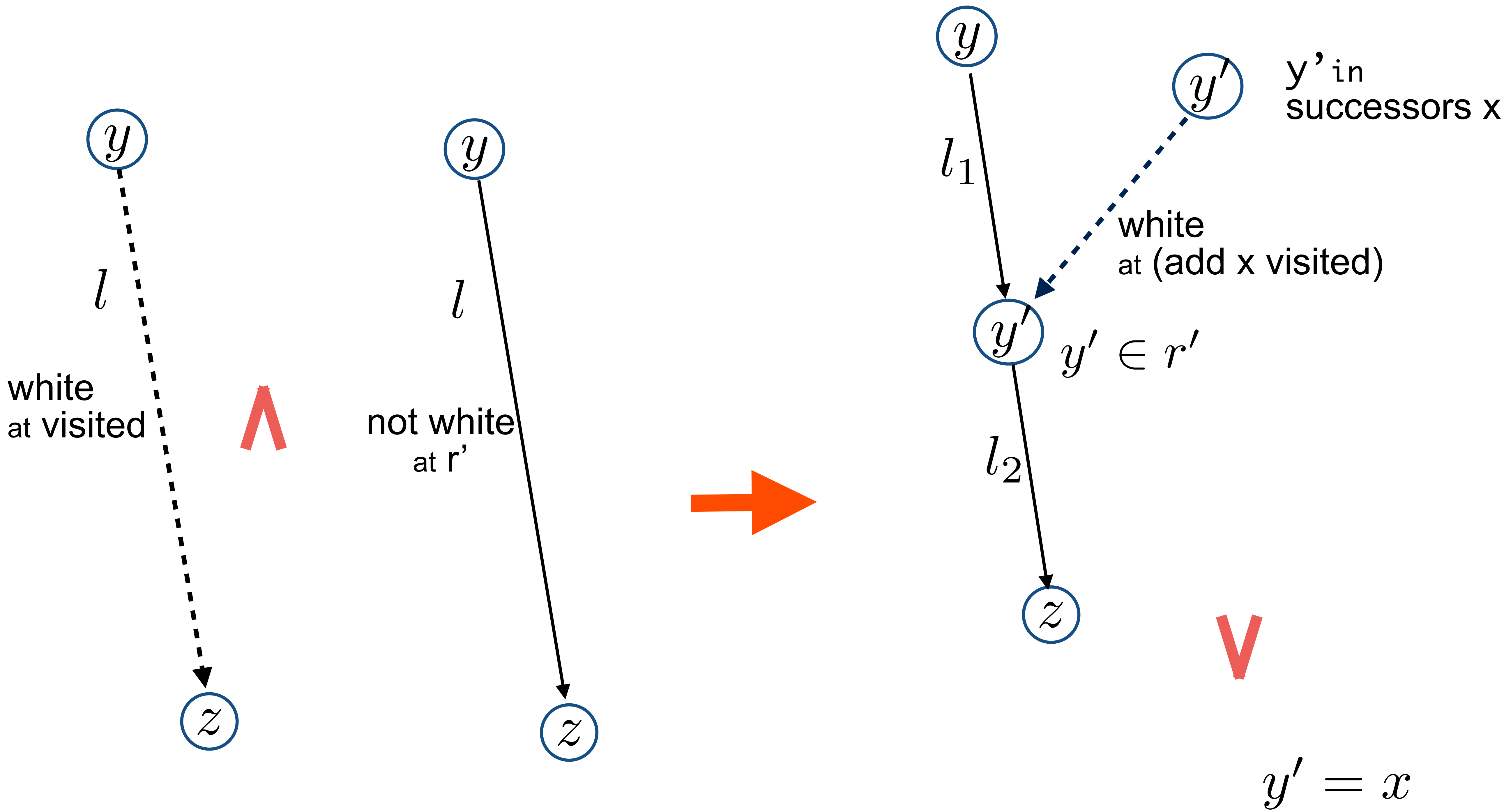
# DFS (whitepath — nodeflip)

- with same 3 lemmas (proved in Why3)

```
(*-------- whitepath_nodeflip ------------------------------------------*)
 (* case 1: whiteaccessfrom roots' z r' *)
   assert {forall y l z. mem y roots' -> whitepath y l z r' -> mem z r };
 (* case 2: not (whiteaccessfrom roots' z r') *)
   assert {forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
       exists y'. (L.mem y' l \/ y' = z) /\ mem y' (diff r' visited) };
   assert {forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
       exists y'. (L.mem y' l \/ y' = z) /\
         (y' = x  \/ whiteaccessfrom (successors x) y' (add x visited)) };
(**)   assert {forall y'. whiteaccessfrom (successors x) y' (add x visited) ->
       exists l'. whitepath x l' y' visited };
   assert {forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
       exists y' l'. (L.mem y' l \/ y' = z) /\ whitepath x l' y' visited };
   assert {forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
       exists l'. whitepath x l' z visited };
 (* case 3-1: whitepath x l z /\ z = x *)
   assert {mem x r'};
 (* case 3-2: whitepath x l z /\ z <> x *)
 (* using lemma whitepath_whitepath_fst_not_twice *)
   assert {forall l z. z <> x -> whitepath x l z visited
       -> exists x' l'. edge x x' /\ whitepath x' l' z (add x visited) };
r
```

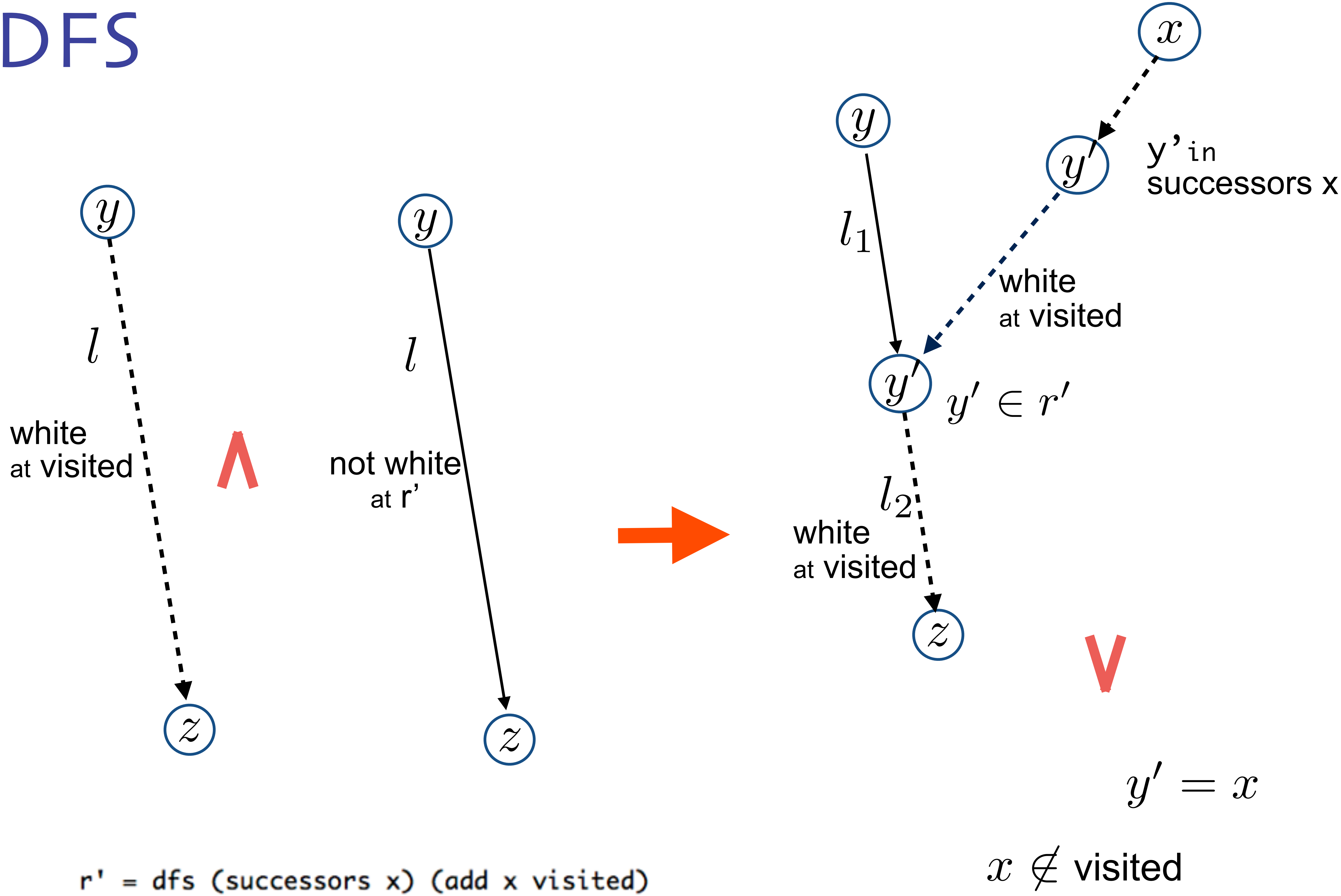
- 3 Coq proofs (final postcond + Y lemma + fst_not_twice)

# DFS




```
r' = dfs (successors x) (add x visited)
```

# DFS




```
r' = dfs (successors x) (add x visited)
```

# DFS

- more complex than iterative version (random walk) !

- see web at `jeanjacqueslevy.net/why3`

# Conclusions

# Conclusion

- readable proofs ?
- simple algorithms should have simple proofs

→ to be shown with a good formal precision

- further algorithms (in next  talk?)
  - graphs represented with arrays + lists
  - dag check, articulation points, sccK, sscT
- progress in using better meta-language in Why3 proofs ?
- Why3 is a beautiful system but not so easy to use !