Readable proofs of DFS in graphs using Why3
jean-jacques.levy@inria.fr
PCRI, 04-12-2015

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

Plan

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

... joint work (in progress) with Ran Chen

Starting with white nodes
The program

```ocaml
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
```

• a functional version with finite sets

The program

```ocaml
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

```ocaml
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 visited) in  
      dfs roots' (union visited b) 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`
The program

\[\text{predicate no\_black\_to\_white}(b : \text{set vertex}) = \forall x', y : \text{edge } x y \rightarrow \text{mem } x b \rightarrow \text{mem } x' (\text{union } b)\]

\[\text{predicates}R(x, y) = (\text{mem } x b) \wedge (\text{mem } y (\text{successors } x))\]

\[\text{let rec dfs}(r, b : \text{set vertex}) =\]

\[\text{variant}((\text{cardinal } r), (\text{cardinal } b)) =\]

\[\text{requires}((\text{subset } r), (\text{subset } b))\]

\[\text{ensures}((\text{subset } r \text{ result}), (\text{subset } b \text{ result}))\]

\[\text{if is\_empty } r \text{ then } \text{b} \text{ else}\]

\[\text{let } x = \text{choose } r \text{ in}\]

\[\text{let } r' = \text{remove } x r \text{ in}\]

\[\text{if mem } x (\text{union } b) \text{ then}\]

\[\text{dfs } r' g b\]

\[\text{else}\]

\[\text{let } b' = \text{dfs } (\text{successors } x) (\text{add } x g) b \text{ in}\]

\[\text{dfs } r' g (\text{union } b (\text{add } x b'))\]

\[\text{lemma} \text{black\_to\_white\_path\_gives\_ths\_gray} :\]

\[\forall y : \text{no\_black\_to\_white}(b) \rightarrow\]

\[\forall z, l : \text{path } z l \rightarrow \text{block } x b \rightarrow \neg \text{mem } x (\text{union } b) (\rightarrow z, l : \text{mem } y l \wedge \text{mem } y g)\]

\[\text{let dfs\_main}(r =\]

\[\text{requires}((\text{subset } r \text{ vertices}), (\text{subset } s \text{ result}))\]

\[\text{ensures}((\forall x, s x s \rightarrow \text{subset } s \text{ result}))\]

\[\text{dfs } r \text{ empty empty} \]
The program

**predicate** no.black.to.white (bg: set vertex) =
\forall x x', edge x x' \rightarrow mem x b \rightarrow mem x' (union b g)

**lemma** black.to.white.path.goes.thru.gray :
\forall g b. no.black.to.white b g → 
\forall l "induction" z. path x l z \rightarrow mem x b \rightarrow \neg mem z (union b g) →
\exists y. L.mem y l \land mem y g

```
let dfs_main r =
  requires (subset r vertices)
  ensures (\{x, access r s ↔ subset s result\})
dfs r empty empty
```

The program

**predicate** no.black.to.white (bg: set vertex) =
\forall x x'. edge x x' \rightarrow mem x b \rightarrow mem x' (union b g)

**lemma** black.to.white.path.goes.thru.gray :
\forall g b. no.black.to.white b g → 
\forall l "induction" z. path x l z \rightarrow mem x b \rightarrow \neg mem z (union b g) →
\exists y. L.mem y l \land mem y g

does not work with Why3!

although easy induction (proved with Coq)

![Diagram](random_walk_diagram.png)

Starting with any color
(random walk)
The program

```ocaml
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

```ocaml
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]

predicate white_vertex (z : vertex) (v : set vertex) = 
  (mem z v)

predicate whitepath (z : vertex) (l : list vertex) (v : set vertex) = 
  path z l z 
  /
  (forall y: vertex. 
    mem y v 
    /
    
    (forall y: vertex. 
      mem y v 
      /
      
      path z y z 
      /
      
      path z y v)
  ) 

**Random walk**

- with 3 lemmas (proved in Why3)

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

lemma whiterachable1 :
  forall x y z v. whitepath y l z (add x v) -> whitepath y l z v

lemma whiterachable2 :
  forall x y z v. not (mem x v) -> whitepath y l z v -> edge x y -> whitepath x (Cons x l) z v

apply (abc ... ) in h12; move: h12 => [h1la \ h1lb].
  - apply h9 in h1la.
  - exists x; exists nil; split.
  - by apply choose_def.
    - exact nila.
  - apply h10 in h1lb; move: h1lb => [h10a \ h10b].
  - move: h10a => [y [1 ['Hyr 'Ham]]].
    - exists y; exists 1; split.
      - by apply remove_def1 in hyr; move: hyr => [ _ hmyr].
      - by apply (whiterachable1 x).
        - move: h10b => [x' [_ ['Hyr 'Ham]]].
          - exists x; exists (x :: 1)\Nil; split.
            - by apply choose_def.
            - apply (whiterachable2 x).
              - exact h4.
              - by apply (whiterachable1 x).
                - exact hyr.

• 1 Coq proof (final postcond)
```

**Random walk**

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

**DFS**

- same proof for bfs or iterative dfs

**see web at jeanjacqueslevy.net/why3**
Starting with any color

DFS

let rec dfs (roots: set vertex) (visited: set vertex): set vertex
  variant [(Cardinal vertices - cardinal visited), (Cardinal roots)]
  requires [subset roots vertices]
  ensures [subset visited vertices]
  ensures [subset visited result]
  ensures [subset result vertices]
  ensures [forall z, mem z (diff result visited) -> exists x, mem x roots \AND 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
      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]
  ensures [subset visited vertices]
  ensures [subset visited result]
  ensures [subset result vertices]
  ensures [forall z, mem z (diff result visited) -> exists x, mem x roots \AND 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 \neq 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, edge x y \AND whitepath y l z (add x visited))];
        let r = dfs roots' r' in
        assert [forall z, mem z (diff r r') -> exists y, mem y roots' \AND whitepath y l z r'];
        assert [forall z, y l. whitepath y l z r' -> whitepath y l z (add x visited)];
      end
end

DFS

Lemma abc :
  forall z: a, r v. mem z (diff r v) -> z = x \OR mem z (diff r (add v x))

Lemma whiterelabel1 :
  forall x y l z v. whitepath y l z (add v x) -> whitepath y l z v

Lemma whiterelabel2 :
  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

• with 3 lemmas (proved in Why3)

• 1 Coq proof (final postcond)

• with same 3 lemmas (proved in Why3)

• 1 Coq proof (final postcond)

• same proof as in random walk
DFS (whitepath — nodeflip)

```
let rec dfs (roots: set vertex) (visited: set vertex) : set vertex
  variant [cardinal vertices — cardinal visited], (cardinal roots)
  requires [subset visited vertices]
  ensures [subset visited result]
  ensures [subset result vertices]
  ensures [forall z. mem z (diff result visited) -> exists x. mem x roots \& whitepath x l z visited]
  ensures [forall x l z. mem x roots -> whitepath x l z visited -> mem x 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
```

**both postconds**

```
(*---------- whitepath_nodeflip *)
(* case 1: whitepath_nodeflip *)
assert [forall y l z. mem y roots \& whitepath y l z r' -> mem x r];
(* case 2: not (whitepath_from roots' z r'*)
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
(*---------- whitepath_nodeflip *)
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
```

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

```
(*---------- whitepath_nodeflip *)
(* case 1: whitepath_from roots' z r' *)
assert [forall y l z. mem y roots \& whitepath y l z r' -> mem x r];
(* case 2: not (whitepath_from roots' z r' *)
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
assert [forall y l z. whitepath y l z visited -> not whitepath y l z r' ->
exists y'. (mem y' \& y' \& y \& (diff r' visited));
(* using lemma whitepath_from_whitepath, whitepath_fst_not_twice *)
assert [forall y l z. x \& whitepath y l z visited -> exists x'. edge x' x \& whitepath x' l z (add x visited)];
```

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

```
DFS

- more complex than iterative version (random walk)!
- see web at jeanjacqueslevy.net/why3

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!