Topological Sorting and Dominators (v2.2)

This project is to be carried out using the Why3 tool, in combination with automated provers (Alt-Ergo (v2.0), CVC4 (v1.4) and Z3). You may use Coq for discharging particular proof obligations, although the project can be completed without it. To get started, you need to install the **latest version** of the Why3 and automated provers, and to download the skeleton file `topo0.mlw`, which contains the template that you need to complete. Both the details of the installation procedure and the skeleton file may be found on the web page of the course.[1]

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `francois.bobot@cea.fr` and `jean-marie.Madiot@inria.fr`, no later than **Friday, February 23th, 2018** at 22:00 UTC+1. This e-mail should be entitled "Project", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The source file `topo.mlw`, completed with your specifications, implementations, and proofs.
- The content of the sub-directory `topo` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you add should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give a pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To proof this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain]."*

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g. why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard was it, what were the major difficulties, was there any unexpected result, and any other information that you think are important to consider for the evaluation of the work you did.

0 Goal

The goal of this project is to prove two real algorithms on graphs. Algorithm on graph are known to be difficult. The first algorithm is topological sorting which detect if a graph contains a cycle, otherwise it returns a bijection between the vertex and the integer that verify the property that if an edge exists from v

[1] `https://francois.bobot.eu/mpri2017/`

to u, the integer associated to v is smaller than the integer associated to u. Such bijection will be called the topological sorting of the graph.

The second algorithm is the computation of the immediate dominators of all the vertex of a graph given a topological sorting of the graph and a `root`vertex from which every vertex is accessible. A vertex u is the dominator of a vertex v ($\neq$ `root`) if all non-empty path from `root`to v contains u. The immediate dominator of v is a dominator of v which is dominated by all the other dominator of v.

The definition of graphs, paths and dominators, and the implementations to prove are given. The work will consists in:

- Fill the definition of `sort` and `topological_sort`
- Prove the function `topo`, `dfs`, `topo_sort` following a step by step guide
- Prove the function `idoms` with only hints

1 Definitions

The map module is referenced as `M` and the set module is referenced as `S` in the following and in the code.

The type for vertex and graph are abstract, = can be used on vertex:

```
type vertex
type graph
```

A graph is only defined through the two functions that gives the set of vertex and the set of edges:

```
function vertices graph: S.set vertex

function edges graph: S.set (vertex , vertex)

axiom edges_use_vertices:
  forall g:graph. forall x y:vertex.
    S.mem (x,y) (edges g) → (S.mem x (vertices g) ∧ S.mem y (vertices g))
```

For convenience the set of predecessors and successors are defined:

```
function preds graph vertex: S.set vertex
function succs graph vertex: S.set vertex
```

Now you need to fill the definition of `sort` and `topological_sort`. The definition of the topological sort is splitted in two. The predicate `sort` only specifies the mapping from vertex to integer `type tsort = M.map vertex int` and the predicate `topological_sort` specifies the second mapping `type order = M.map int vertex` as the bijection of the first.

The predicate `sort (g:graph) (tsort:tsort)` specifies that:

- *Result in interval*: for every vertex v of the graph `M.get tsort v` is between 0 (included) and the number of vertex of the graph (excluded).
- *Edges are well ordered*: For every edges in the graph, the source is associated in `tsort` to an integer strictly smaller than the destination
- *Injectivity*: The mapping `tsort` associates different values for different vertex.

The predicate `order (g:graph) (order:order) (tsort:tsort)`:

- *Bijection*: that `order` and `tsort` are inverse of one another.
- *Good sorting*: that `sort g tsort` is verified.

- *Ordered elements are vertices of the graph*: it is indeed a corollary but it is easier to prove it directly. Precisely it specifies that for every integer between `0` and `S.cardinal (vertices g)`, `M.get order i` is a vertices of `g`

Finally the paths are defined as list of vertex, the predicate `path_to g p v` indicates that `p` is a path in `g` that arrives in `v`. Useful functions is `elements` that return the set of elements contained in a list, `++` which is the concatenation of list.

2 Tests Definitions

In order to check that your definition of `sort` and `topological_sort` are the one we wait, you should try to prove simple properties of topological order in the module `TestDefinition`. For example:

- `no_empty_path`: For any graph, the empty list (`Nil`) is not a path to any vertex.
- `path_for_two_edges`: if there are edges $v_1 \to v_2 \to v_3$ then `Cons v1 (Cons v2 Nil)` is a path to `v3` in the graph `g`.
- `pred_smaller`: in a topological sort, all predecessors of a vertex are associated to a smaller integer than this vertex.
- `no_direct_cycle1`: if a graph has a topological sorting, $v_1 \to v_2 \to v_1$ is impossible
- `no_direct_cycle2`: if a graph has a topological sorting, $v_1 \to v_1$ is impossible
- `path_ordered`: if a graph has a topological sorting, if there is a path from `v1` to `v2`, then `v1` is associated to a strictly smaller integer than `v2`.
- `no_cycle`: if a graph has a topological sorting, then it has no cycle.

3 Topological sort

3.1 Implementation

The implementation is splited in three function. The function `dfs` fill if not already done the integer associated to a vertex and all its predecessors. The function `topo_tsort` does the same thing for all the vertex and so return the mapping from vertex to integer. finally the function `topo` build the mapping from integer to vertex from the result of the function `topo_tsort`.

The predicate `inv` will be completed gradually during this section.

3.2 Prove termination of the functions

First we are going to prove the termination of the functions.

- Using `S.subset` and `S.mem` add requirements and invariant in `dfs` and `topo_tsort` in order to state the relations between `v`, `seen`, `!p` and `(vertices g)`, `(preds g v)`.
- Prove termination of the functions (using function `S.cardinal`).

3.3 Prove function `topo`

Supposing the contract if `topo_tsort` correct we are going to prove the function `topo`

Prove precondition of the call `H.find tsort !p` by adding a simple invariant ('S.subset' can be used).

Could use inline and split for seeing what is not yet proved in the postcondition.

The easiest property of the conjunction to prove is the one that `order (tsort v) = v`. Add an invariant derived from the property by remarking that the property is true for the vertexes that are not anymore in `!p`.

The last two properties of the conjunction quantify on the indices. We don't yet have any simple way to reference the set of indices already seen (since we are proving the bijection). So we are going to add a ghost set of the indices not yet seen (dual of `p`) named `todo`. For initializing this ghost reference we define an auxiliary function that given two integers return the set of all integers between them. It is useful that this auxiliary function characterize the cardinal of the result.

- Add before the loop the initialization of the `todo` variable.
- Add in the loop the update of the `todo` variable.
- Add an invariant that state the relation between the cardinal of `!p` and the cardinal of `!todo`.
- Add an invariant that state that if a vertex is in `!p` its associated indices is in `!todo`
- Check that the new invariants are proved.

Now we could add invariants derived from the last two properties not proved, using `!todo`.

The function `topo` should not be completely proved. You can try to simplify the proof by removing transformations (splitting, inline) perhaps not needed.

3.4 Prove that the result defines all vertices of g

Prove the second post-condition which specify **`result`**`.defined` by adding two invariants using (`S.subset` and `S.diff`). These invariants give a lower and upper bound on `tsort.defined`.

In order to prove the preservation of these invariants we need to add properties to `dfs`.

- the upper bound is always verified (can be added to the predicate `inv`)
- that `tsort.defined` only grows
- finally that `v` is defined after the run of `dfs`

3.5 Prove that the result is in the interval

We are going to prove the first property of predicate sort. For that we need first to add the property that the order of addition to `tsort` is postfix. We express that by adding as precondition, post-condition and loop invariant of `dfs` that all the vertex in `seen` are not defined in `tsort`.

Now it is possible to add a derived version of the first property in `inv` in order to prove that the result is in the interval.

3.6 Prove the injectivity

We could add a derived version of the third property in `inv` in order to prove the injectivity of the result. You need to strengthen the property just added in `inv` in order to show that the value associated to the defined elements is smaller than the current number of defined elements.

3.7 Prove that the edges are well ordered

In order to prove the second property, the main one, we need to specify precisely which vertex are defined. We need to add the specification for proving that at the end of the loop in `dfs` all the vertices of `preds g v` are defined. We also need to add the specification that the origin of an edge is always defined if the destination is defined.

Now we can add a derived version of the second property in `inv` in order to prove it.

3.8 Prove that `Cycle_found` is raised only if it exists a cycle

The exception is raised when the current vertex is already in the callstack, ie in the set `seen`. Since the callstack represent a path in the graph, that indeed correspond to finding a cycle.

So in order to prove the existence of a cycle we need the requirement that there exists a path from `v` to any vertices of `seen`. As usual if the provers are not able to handle the existential well we can use a ghost parameter `seen_path` of type `M.map vertex path` in order to replace the existential, and explicitly tell what is the new witness for the precondition of the recursive call to `dfs`.

The computation of the witnesses should be done by implementing a ghost function that takes the graph, the set `seen`, the old `seen_path`, the old current vertex `v` and the new vertex `u` and returns the new witness. The function need to suppose that `(u,v)` is an edges of `g` and that `seen_path` is a witness for the path from `v` to vertexes of `seen`.

For the first call to `dfs` in `topo_tsort` you could use `Const.const Nil`.

Once this requirement proved we can replace **`raises`** `{ Cycle_found` $\rightarrow$ **`true`** `}` by **`raises`** `{ Cycle_found` $\rightarrow$ `cycle_in g }` in all functions.

4 Proof of `idoms`

This part of the project is not guided, the important point is not so much that it is fully proved at the end, but more what you tried to solve it.

The goal is to compute the immediate dominators of each vertex of the graph. The functions require that the argument `root` is the root of the graph (there is a path from `root` to all the other vertex). A vertex u is the dominator of a vertex v ($\neq$ `root`) if all non-empty path from `root`to v contains u. The immediate dominator of v is a dominator of v which is dominated by all the other dominator of v.

Here some hints:

- The `root` vertex is special, so it is handled in the inner loop by raising an exception. It makes the specification simpler.
- Instead of adding many assertions that are only used to prove one assertion you could use `by`. `A by B` is equivalent to `A` when no splitting is done, otherwise with some splitting Why3 tries to prove `B` and `B` $\Rightarrow$ `A`.
- It is better to use the specific equality on sets `S.(==) s1 s2` than `s1 = s2` for provers because it can be used as trigger pattern. The axioms of extensionality convert between the two.
- The derived predicate for `domine` (called for example `partial_domine`) should only look at paths that contains some vertex given as arguments.
- The hardest part is the inner function `find_common`. you could keep it for the end. The main argument is that since `n2` $<$ `n1` and all partial dominators of `v` different from the vertex of `n2` domine the vertex of `n2` then the vertex of `n1` can't be a partial dominators of `v`.
- A lemma that cut a path `path_to g (Cons start (p1 ++ (Cons x p2))) stop` into two `path_to` is useful.

- Other lemmas could be needed.
- The proof have been mainly done using Alt-ergo (v2.0), occasionally CVC4 (v1.4).

5 Conclusion

If things are not clear you should not hesitate to mail a question to `francois.bobot@cea.fr` and `jean-marie.Madiot@inria.fr`. All the questions and answers will be shared in a faq on the website. I hope you will enjoy the challenge!

6 Annex

In order to fix the version of the source code to start with:

```
theory Graph

  use export int.Int
  use set.Fset as S
  use map.Map as M

  (* the graph is defined by a set of vertices and a set of edges *)
  type vertex
  type graph

  function vertices graph: S.set vertex

  function edges graph: S.set (vertex , vertex)

  axiom edges_use_vertices:
  forall g:graph. forall x y:vertex.
    S.mem (x,y) (edges g) → (S.mem x (vertices g) ∧ S.mem y (vertices g))

  (** direct predecessors *)
  function preds graph vertex: S.set vertex

  axiom preds_def:  forall g:graph. forall v:vertex. forall u:vertex.
   S.mem (u,v) (edges g) ↔ S.mem u (preds g v)

  (** direct successors *)
  function succs graph vertex: S.set vertex
  axiom succs_def:  forall g:graph. forall v:vertex. forall u:vertex.
   S.mem (u,v) (edges g) ↔ S.mem v (succs g u)

  type tsort = M.map vertex int

  predicate sort (g: graph) (tsort:tsort) =
    (* Result in interval [0;S.cardinal (vertices g)[ *)
    true
    ∧
    (* Edges are well ordered *)
    true
```

```
    ∧
    (* Injectivity *)
    true


  type order = M.map int vertex

  predicate topological_sort (g: graph) (order: order) (tsort: tsort) =
    (* order and tsort are in bijection *)
    (* ... order (tsort v) = v *)
    true ∧
    (* ... tsort (order v) = v *)
    true ∧
    (* tsort is a good sorting *)
    sort g tsort ∧
    (* corollary hard to prove: ordered elements are vertices of the graph  *)
    true

  lemma topological_sort_corrolaries:
      forall g. forall order. forall tsort.
       topological_sort g order tsort →
        (* no backward edges *)
        (forall i, j: int.
        0 ≤ i ≤ j < S.cardinal (vertices g) →
        ¬ (S.mem (M.get order j, M.get order i) (edges g)))

end

theory Path
  use import Graph
  use import list.List
  use import list.Length
  use set.Fset as S

  type path = list vertex

  predicate path_to  (g : graph) (p : path) (to_: vertex) =
    match p with
    | Nil → false
    | Cons a l →
      match l with
      | Nil → S.mem (a,to_) (edges g)
      | Cons b _ → S.mem (a,b) (edges g) ∧ path_to g l to_
      end
    end

  predicate cycle_in (g: graph) =
    exists p. exists v. path_to g (Cons v p) v

end
```

```
module TestDefinition
  use import Graph
  use import Path
  use import list.List

  lemma pred_smaller:
     true

  lemma no_direct_cycle1:
     true

  lemma no_direct_cycle2:
     true


  let rec lemma path_ordered (g:graph) (tsort:tsort) (p:path) (v1 v2:vertex)
   requires { true }
   requires { true }
   ensures  { true }
   variant { p }
   =
   ()

 lemma no_cycle: true

end

module Hashtbl

  use import map.Map
  use import set.Fset as S

  type key

  type t α model { mutable contents: map key α;
                   mutable defined: set key;
                 }

  function ([]) (h: t α) (k: key) : α = Map.([]) h.contents k

  val create () : t α ensures { result.defined = S.empty }

  val clear (h: t α) : unit writes {h} ensures { h.defined = S.empty }

  val add (h: t α) (k: key) (v: α) : unit writes {h}
    ensures { h.contents = (old h.contents[k<-v]) }
    ensures { h.defined = S.add k (old h.defined) }

  val mem (h: t α) (k: key) : bool
    ensures { result = (S.mem k h.defined) }
```

```
  val find (h: t α) (k: key) : α
    requires { S.mem k h.defined }
    ensures  { result = h[k] }

  val remove (h: t α) (k: key) : unit writes {h}
    ensures { h.defined = S.remove k (old h.defined) }

  val size (h: t α) : int
    ensures { result = S.cardinal h.defined }

end

(** Topological sorting by depth-first search using preds *)
module Topo

  use import ref.Ref
  use import Graph
  use import Path
  use import list.List
  use import list.Length
  use set.Fset as S
  use map.Map as M
  clone import Hashtbl as H with type key = vertex
  use map.Const

  type marked = (S.set vertex)

  exception Cycle_found

  predicate inv (g:graph) (m:H.t int) =
    true

  let rec dfs (g:graph) (v:vertex)
              (seen:marked) (tsort:H.t int)
              : unit
    requires { inv g tsort }
    requires { true }
    requires { true }
    requires { true }
    requires { true }
    diverges
    ensures { true }
    ensures { true }
    ensures { true }
    ensures { inv g tsort }
    ensures { true }
    raises  { Cycle_found → true  }
   =
  'Init:
   if S.mem v seen then raise Cycle_found;
   if ¬ (H.mem tsort v) then begin
```

```
'Init_loop:
  begin
   let p = ref (preds g v) in
   let seen' = S.add v seen in
   while ¬ (S.is_empty !p) do
    invariant { inv g tsort }
    invariant { true }
    invariant { true }
    invariant { true }
    invariant { true }
    let u = S.choose !p in
    dfs g u seen' tsort;
    p := S.remove u !p
    done;
  end;
   H.add tsort v (H.size tsort)
  end


  let topo_tsort (g:graph): H.t int
    raises  { Cycle_found → true  }
    ensures { sort g result.contents }
    ensures { S.(==) result.defined (vertices g) }
    diverges
    =
  'Init:
    let tsort = H.create () in
    let p = ref (vertices g) in
    while ¬ (S.is_empty !p) do
      invariant { inv g tsort }
      invariant { true }
      invariant { true }
      invariant { true }
      let u = S.choose !p in
      dfs g u (S.empty) tsort;
      p := S.remove u !p
   done;
   tsort

  use import array.Array

  type topo = {
    order: array vertex;
    tsort: H.t int
  }


  let topo (g:graph): topo
    raises  { Cycle_found → true }
    requires { ¬ (S.is_empty (vertices g)) }
    ensures { topological_sort g result.order.elts result.tsort.contents }
```

```
      ensures { S.(==) result.tsort.defined (vertices g) }
      ensures { result.order.length = S.cardinal (vertices g) }
      diverges
     =
      let tsort = topo_tsort g in
      let order = Array.make (S.cardinal (vertices g)) (S.choose (vertices g)) in
      let p = ref (vertices g) in
      while ¬ (S.is_empty !p) do
        invariant { true }
        invariant { true }
        invariant { true }
        invariant { true }
        invariant { true }
        invariant { true }
        let u = S.choose !p in
        order[H.find tsort u] <- u;
        p := S.remove u !p;
     done;
     {order = order; tsort = tsort}

end

theory Dominator
  use import Graph
  use import list.List
  use import list.Length
  use import list.Elements
  use set.Fset as S
  use import Path

  predicate domine (g:graph) (root x y:vertex) =
   y ≠ root ∧
   forall p. path_to g (Cons root p) y → S.mem x (S.add root (elements p))

  (** immediate dominator *)
  predicate idomine (g:graph) (root x y:vertex) =
   domine g root x y ∧ (forall x'. x' ≠ x → domine g root x' y → domine g root x' x)

end

module Dominators

  use import ref.Ref
  use import Graph
  use import Path
  use import Dominator
  use import list.List
  use import list.Length
  use import list.Elements
  use import list.Append
  use set.Fset as S
```

```
 use map.Map as M
 use map.Const
 use import array.Array
 use import Topo

 predicate not_disjoint (s1 s2: S.set vertex) =
  ¬ (S.is_empty (S.inter s1 s2))

 lemma not_disjoint_mem:
   forall s1 s2: S.set vertex. not_disjoint s1 s2 ↔ (exists x. S.mem x s1 ∧ S.mem x s2)

lemma not_disjoint_singleton:
   forall s1: S.set vertex. forall x: vertex. not_disjoint s1 (S.singleton x) ↔ S.mem x s1

 exception Root

 let idoms g (root:vertex) (topo:Topo.topo) (ghost exi_path : M.map vertex path)
    : array int
  requires { S.mem root (vertices g) }
  requires { S.is_empty (preds g root) }
  requires { forall v. v ≠ root → S.mem v (vertices g) → path_to g (Cons root (M.get exi_path v))

  requires { topological_sort g topo.Topo.order.elts topo.Topo.tsort.Topo.H.contents }
  requires { S.(==) topo.Topo.tsort.Topo.H.defined (vertices g) }
  requires { topo.Topo.order.length = S.cardinal (vertices g) }

  ensures { forall i. 0 < i < S.cardinal (vertices g) →
             idomine g root (topo.Topo.order[result[i]]) (topo.Topo.order[i]) }
  ensures { result[0] = 0 }
  diverges
  =
   let a = Array.make (S.cardinal (vertices g)) (-1) in
   a[0] <- 0;
   for nv=1 to (S.cardinal (vertices g) - 1) do
     'Start_body:
     let v = topo.Topo.order[nv] in
     let rec find_common n1 n2
       raises  { Root → idomine g root root v }
       diverges
      =
       if n1 = n2 then n1
       else
       let n1, n2 = if n2 < n1 then (n1,n2) else (n2,n1) in
       if a[n1] = 0 then raise Root
       else find_common a[n1] n2
     in
   try
     let p = ref (preds g v) in
     assert { ¬ (S.is_empty !p) };
     let u = S.choose !p in
     (if u = root then raise Root);
```

```
      let nu = Topo.H.find topo.Topo.tsort u in
      let idom_v = ref nu in
      p := S.remove u !p;
      while ¬ (S.is_empty !p) do
       let u = S.choose !p in
       (if u = root then raise Root);
       let nu = Topo.H.find topo.Topo.tsort u in
       idom_v := find_common !idom_v nu;
       p := S.remove u !p;
      done;
      a[nv] <- !idom_v;
    with Root →
      a[nv] <- 0;
    end
    done;
    a

end
```