Comparing a Formal Proof in Why3, Coq, Isabelle

Jean-Jacques Lévy
Inria Paris - Irif

VIP meeting 20-11-2018
Motivation

• nice algorithms should have simple formal proofs
• to be fully published in articles or journals
• how to publish formal proofs?
• algorithms on graphs = a good testbed (better than $\sqrt{2}$)
• formal proofs have to be checked by computer

.. with Ran Chen, Cyril Cohen, Stephan Merz, Laurent Théry

VSTTE 2017, CPP 2019 (?)

http://www-sop.inria.fr/marelle/Tarjan/contributions.html
A one-pass linear-time algorithm

Tarjan, 1972
Strongly connected components

- maximum subsets of vertices connected by a path
- depth-first search algorithm pushing vertices on a stack in their order of visit
- computing oldest vertex reachable by at most one "cross-edge"
- when not strictly less than currently visited vertex, a new SCC is on top of current vertex in working stack
- the SCC is popped and algorithm continues
Strongly connected components

diagram: graph, spanning forest, stack
Strongly connected components

LOWLINK(x) = min\{num[y] \mid x \xrightarrow{*} z \xleftarrow{} y \}
\wedge x \text{ and } y \text{ are in the same connected component}
Program

type vertex
constant vertices: set vertex
function successors vertex : set vertex
type env = {stack: list vertex;
    sccs: set (set vertex);
    sn: int; num: map vertex int}

let tarjan () =
    let e = {stack = Nil; sccs = empty;
        sn = 0; num = const (-1)} in
    let (_, e') = dfs vertices e in e'.sccs
let rec dfs1 x e =
  let n0 = e.sn in
  let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
  if n1 < n0 then (n1, e1) else
  let (s2, s3) = split x e1.stack in
  (+∞, {stack = s3;
        sccs = add (elements s2) e1.sccs;
        sn = e1.sn; num = set_infty s2 e1.num})

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

type vertex
constant vertices: set vertex
function successors vertex : set vertex
type env = {ghost black: set vertex;
            ghost gray: set vertex;
            stack: list vertex; sccs: set (set vertex);
            sn: int; num: map vertex int}

let tarjan () =
    let e = {black = empty; gray = empty;
             stack = Nil; sccs = empty; sn = 0;
             num = const (-1)} in
    let (_, e') = dfs vertices e in e'.sccs
let rec dfs1 x e =
    let n0 = e.sn in
    let (n1, e1) = dfs (successors x)
        (add_stack_incr x e) in
    if n1 < n0 then (n1, add_black x e1) else
    let (s2, s3) = split x e1.stack in
        (+∞, {stack = s3;
            black = add x e1.black; gray = e.gray;
            sccs = add (elements s2) e1.sccs;
            sn = e1.sn; num = set_infty s2 e1.num})

let add_stack_incr x e =
    let n = e.sn in
    {black = e.black; gray = add x e.gray;
     stack = Cons x e.stack; sccs = e.sccs;
     sn = n+1; num = e.num[x ←n]}

let add_black x e =
    {black = add x e.black; gray = remove x e.gray;
     stack = e.stack; sccs = e.sccs;
     sn = e.sn; num = e.num}
Invariant

(1) consistent colors
(2) consistent numbering
(3) vertices pairwise distinct in stack
(4) no edge from black to white
(5) in stack any vertex reaches any higher vertex
(6) in stack any vertex reaches a gray lower vertex
(7) the sccs field is the set of black SCCs
Why3 Proof
Pre/Post-conditions

let rec dfs1 x e =
(* pre-condition *)
requires {mem x vertices}
requires {∀y. mem y e.gray → reachable y x}
requires {not mem x (union e.black e.gray)}
requires {wf_env e} (* I *)

(* post-condition *)
returns {(_, e') → wf_env e' ∧ subenv e e'}
returns {(_, e') → mem x e'.black}
returns {(_, e') → n ≤ e'.num[x]}
returns {(_, e') → n = +∞ ∨ num_of_reachable n x e'}
returns {(_, e') → ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
Assertions

let n0 = e.nn in
let (n1, e1) =
    dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
    assert{∃y. y ≠ x ∧ precedes x y e1.stack ∧
        reachable x y};
    assert{∃y. y ≠ x ∧ mem y e1.gray ∧
        e1.num[y] < e1.num[x] ∧ in_same_scc x y};
    (n1, add_black x e1) end
else

let (s2, s3) = split x e1.stack in

assert\{is_last x s2 ∧ s3 = e.stack ∧
        subset (elements s2) (add x e1.black)\};
assert\{is_subsccc (elements s2)\};
assert\{∀y. in_same_scc y x → lmem y s2\};
assert\{is_scc (elements s2)\};
assert\{inter e.gray (elements s2) == empty\};
(+∞, \{black = add x e1.black; gray = e.gray;
        stack = s3; sccs = add (elements s2) e1.sccs;
        sn = e1.sn; num = set_infty s2 e1.num\})
Coq proof

- 3 cases on $y'$
  - $y'$ in sccs
  - $y'$ is white vertex
  - $y'$ in s3
## Assertions

<table>
<thead>
<tr>
<th>prover</th>
<th>Alt-Ergo</th>
<th>CVC4</th>
<th>E-prover</th>
<th>Z3</th>
<th>#VC</th>
<th>#PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>49 lemmas</td>
<td>1.91</td>
<td>26.11</td>
<td>3.33</td>
<td>70</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>split</td>
<td>0.09</td>
<td>0.16</td>
<td></td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>add_stack_incr</td>
<td>0.01</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>add_black</td>
<td>0.02</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>set_infty</td>
<td>0.03</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dfs1</td>
<td>77.89</td>
<td>150.2</td>
<td>19.99</td>
<td>13.67</td>
<td>79</td>
<td>20</td>
</tr>
<tr>
<td>dfs</td>
<td>4.71</td>
<td>3.52</td>
<td>0.26</td>
<td>58</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>tarjan</td>
<td>0.85</td>
<td></td>
<td></td>
<td>15</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>85.51</td>
<td>179.99</td>
<td>23.32</td>
<td>13.93</td>
<td>231</td>
<td>108</td>
</tr>
</tbody>
</table>

**Table 1.** Performance results of the provers (in seconds, on a 3.3 GHz Intel Core i5 processor). Total time is 341.15 seconds. The two last columns contain the numbers of verification conditions and proof obligations. Notice that there may be several VCs per proof obligation.

+ 2 Coq proofs (16 loc + 141 loc)
Coq Proof
Functions

Record env := Env {black : {set V};
  stack : seq V;
  sccs : {set {set V}};
  sn : nat;  num : {ffun V \rightarrow nat}}.

Definition dfs1 dfs x e :=
  let: (n1, e1) :=
    dfs [set y in successors x] (add_stack x e) in
  if n1 < sn e then (n1, add_black x e1)
  else (infty, add_sccs x e1).

Definition dfs dfs1 dfs’ r e :=
  if [pick x in r] isn't Some x then (infty, e)
  else let r' := r \x in
    let: (n1, e1) :=
      if num e x != 0 then (num e x, e) else dfs1 x e in
    let: (n2, e2) := dfs’ r’ e1 in (minn n1 n2, e2).
Functions

Fixpoint tarjan_rec n :=
  if n is n1.+1 then
    dfs (dfs1 (tarjan_rec n1)) (tarjan_rec n1)
  else fun r e => (infty, e).

Let N := #|V| * #|V|.+1 + #|V|.

Definition tarjan := sccs (tarjan_rec N setT e0).2.
Proof

Definition dfs_correct

\[(dfs : \{\text{set } V\} \to \text{env} \to \text{nat} \times \text{env}) \ r \ e :=\]
\[\text{pre_dfs } r \ e \to\]
\[\text{let } (n, e') := dfs \ r \ e \text{ in post_dfs } r \ e \ e' \ n.\]

Definition dfs1_correct

\[(dfs_1 : V \to \text{env} \to \text{nat} \times \text{env}) \ x \ e :=\]
\[(x \in \text{white } e) \to \text{pre_dfs } [\text{set } x] \ e \to\]
\[\text{let } (n, e') := dfs_1 \ x \ e \text{ in post_dfs } [\text{set } x] \ e \ e' \ n.\]
Proof

Lemma dfs_is_correct dfs' (r : {set V}) e :
(∀x, x ∈ r → dfs1_correct dfs' x e) →
(∀x, x ∈ r → ∀e1, white e1 \subset white e →
  dfs_correct dfs' (r #: x) e1) →
  dfs_correct (dfs dfs' dfs') r e.

Lemma dfs1_is_correct dfs' (x : V) e :
(dfs_correct dfs' [set y | edge x y] (add_stack x e)) →
  dfs1_correct (dfs1 dfs') x e.

Theorem tarjan_rec_terminates n r e :
n ≥ |white e| * |V|.+1 + |r| →
  dfs_correct (tarjan_rec n) r e.
Isabelle/HOL Proof
Proof

function (domintros) dfs1 and dfs where

dfs1 x e =

(let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
if n1 < int (sn e) then (n1, add_black x e1)
else (let (l, r) = split_list x (stack e1) in
(+∞, (| black = insert x (black e1), gray = gray e,
stack = r, sn = sn e1, sccs = insert (set l) (sccs e1),
num = set_infty l (num e1) |) )))

and

dfs roots e =

(if roots = {} then (+∞, e)
else (let x = SOME x . x ∈ roots;
res1 = (if num e x ≠ -1 then (num e x, e) else dfs1 x e);
res2 = dfs (roots - {x}) (snd res1)
in (min (fst res1) (fst res2), snd res2) )))
**Proof**

Definition colored_num where colored_num e ≡ 
∀v ∈ colored e. v ∈ vertices ∧ num e v ≠ -1

**Theorem** dfs1_dfs_termination :

[x ∈ vertices - colored e; colored_num e] → dfs1_dfs_dom (Inl(x, e)) 
[r ⊆ vertices; colored_num e] → dfs1_dfs_dom (Inr(r, e))

**Theorem** dfs_partial_correct:

[dfs1_dfs_dom (Inl(x, e)); dfs1_pre x e] → dfs1_post x e (dfs1 x e) 
[dfs1_dfs_dom (Inr(r, e)); dfs_pre r e] → dfs_post r e (dfs r e)

**Theorem** dfs_correct:

dfs1_pre x e → dfs1_post x e (dfs1 x e) 
dfs_pre r e → dfs_post roots e (dfs r e)
Conclusion
# Why3 - Coq - Isabelle

<table>
<thead>
<tr>
<th></th>
<th>why3</th>
<th>coq</th>
<th>isabelle/HOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>expressivity</td>
<td>-</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>readability</td>
<td>+++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>stability</td>
<td>-</td>
<td>+++</td>
<td>+</td>
</tr>
<tr>
<td>ease of use</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>automation</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>partial correctness</td>
<td>+++</td>
<td>- -</td>
<td>-</td>
</tr>
<tr>
<td>code extraction</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>trusted base</td>
<td>-</td>
<td>+++</td>
<td>+++</td>
</tr>
<tr>
<td># lines auto</td>
<td>392</td>
<td>0</td>
<td>? (314ui)</td>
</tr>
<tr>
<td># lines manual</td>
<td>157</td>
<td>1535</td>
<td>1690</td>
</tr>
</tbody>
</table>

... other systems ?

http://www-sop.inria.fr/marelle/Tarjan/contributions.html
Todo list

- proof of implementation
- other algorithms (biconnected, planarity, minimum spanning tree)
- teaching