Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle

Ran Chen
Institute of Software of the Chinese Academy of Sciences, Beijing
chenr@ios.ac.cn

Cyril Cohen
Université Côte d'Azur, Inria
cyril.cohen@inria.fr

Jean-Jacques Lévy
Irif & Inria Paris
jean-jacques.levy@inria.fr

Stephan Merz
Université de Lorraine, CNRS, Inria, LORIA
stephan.merz@inria.fr

Laurent Théry
Université Côte d'Azur, Inria
laurent.thery@inria.fr

Abstract

Comparing provers on a formalization of the same problem is always a valuable exercise. In this paper, we present the formal proof of correctness of a non-trivial algorithm from graph theory that was carried out in three proof assistants: Why3, Coq, and Isabelle.

2012 ACM Subject Classification Logic and verification, Automated reasoning, Higher order logic

Keywords and phrases Mathematical logic, Formal proof, Graph algorithm, Program verification

1 Introduction

Graph algorithms are notoriously obscure in the sense that it is hard to grasp why exactly they work. Therefore proof of correctness are more than welcome in this domain. In this paper, we consider Tarjan's algorithm [28] for discovering the strongly connected components in a directed graph and present a formal proof of its correctness in three different systems: Why3, Coq and Isabelle/HOL. The algorithm is treated at an abstract level with a functional programming style manipulating finite sets, stacks and mappings, but it respects the linear time behaviour of the original presentation.

To our knowledge this is the first time that the formal correctness proof of a non-trivial program is carried out in three very different proof assistants: Why3 is based on a first-order logic with inductive predicates and automatic provers, Coq on an expressive theory of higher-order logic and dependent types, and Isabelle/HOL combines higher-order logic with automatic provers. We claim that our proof is direct, readable, elegant, and follows Tarjan's presentation. Crucially for our comparison, the algorithm is defined at the same level of abstraction in all three systems, and the proof relies on the same arguments in the three formal systems. Note that a similar exercise but for a much more elementary proof (the irrationality of square root of 2) and using many more proof assistants (17) was presented in [32].

Formal and informal proofs of algorithms about graphs were already performed in [24, 30, 25, 13, 17, 29, 19, 27, 26, 15, 8]. Some of them are part of a larger library, others focus on the treatment of pointers or on concurrent algorithms. In particular, only Lammich and

Neumann [17] gave an alternative formal proof of Tarjan's algorithm within their framework for verifying graph algorithms in Isabelle/HOL.

We expose here the key parts of the proofs. The interested reader can access the details of the proofs and run them on the web [7, 9, 20]. In this paper, we recall the principles of the algorithm in section 2; we describe the proofs in the three systems in sections 3, 4, and 5 by emphasizing the differences induced by the logics which are used; we conclude in sections 6 and 7 by commenting the developments and advantages of each proof system.

2 The algorithm

In a directed graph, two vertices x and y are strongly connected if there exists a path from x to y and a path from y to x. A strongly connected component (scc) is a maximal set of vertices where all pairs of vertices are strongly connected. A fundamental property relates sccs and depth-first search (DFS) traversal in a directed graph: each scc is a prefix of a single subtree in the corresponding spanning forest (see figure 1c). Its root is named the base of the scc. Tarjan's algorithm [28] relies on the detection of these bases and collects the sccs in a pushdown stack. It performs a single DFS traversal of the graph assigning a serial number $num[x]$ to any vertex x in the order of the visit. It computes the following function for every vertex x:

$$LOWLINK(x) = \min\{num[y] \mid x \overset{*}{\Longrightarrow} z \hookrightarrow y \ \wedge\ x \text{ and } y \text{ are in the same scc}\}$$

The relation $x \Longrightarrow z$ means that z is a son of x in the spanning forest, the relation $\overset{*}{\Longrightarrow}$ is its transitive and reflexive closure, and $z \hookrightarrow y$ means that there is a cross-edge from z to y in the spanning forest (a cross-edge is an edge of the graph which is not an edge in the spanning forest). In figure 1c, $\Longrightarrow$ is drawn in thick lines and $\hookrightarrow$ in dotted lines; in figure 1b the table of the values of the $LOWLINK$ function is shown. The minimum of the empty set is assumed to be $+\infty$ (this is a slight simplification w.r.t. the original algorithm).

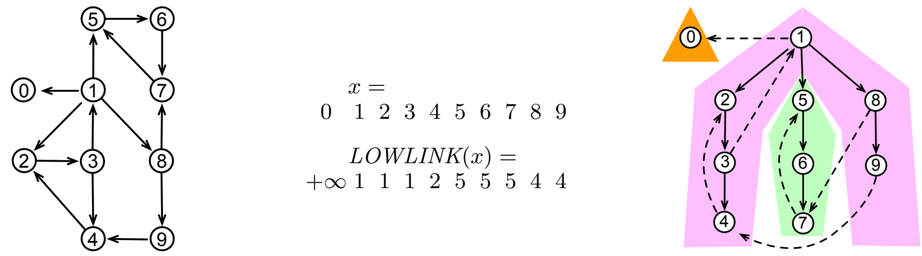
The base x of an scc is found when $LOWLINK(x) \geq num[x]$, and the component is formed by the nodes of the subtree rooted at x and pruned of the sccs already discovered in that subtree. Notice that $LOWLINK(x)$ need neither be the lowest serial number of a vertex accessible from x, nor of an ancestor of x in the spanning forest. Take for instance, vertices 8 or 9 in figure 1c. Moreover, the DFS traversal sets to $+\infty$ the serial numbers of vertices in already discovered sccs. The definition of $LOWLINK$ can therefore be written as:

$$LOWLINK(x) = \min\{num[y] \mid x \overset{*}{\Longrightarrow} z \hookrightarrow y\}$$

Our implementation of graphs uses an abstract type *vertex* for vertices, a constant *vertices* for the finite set of all vertices in the graph, and a *successors* function from vertices to their adjacency set. The algorithm maintains an environment e implemented as a record of type *env* with four fields: a stack *e.stack*, a set *e.sccs* of strongly connected components, a fresh serial number *e.sn*, and a function *e.num* from vertices to serial numbers.

```
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}
```

The DFS traversal is organized by two mutually recursive functions *dfs1* and *dfs*. The function *dfs1* visits a new vertex x and computes $LOWLINK(x)$. Furthermore it adds a new scc when x is the base of a new scc. The function *dfs* takes as argument a set r of roots and



(a) A graph example **(b)** *LOWLINK* **(c)** Spanning forest

Figure 1 The vertices are numbered and pushed onto the stack in the order of their visit by the recursive function *dfs1*. When the first component $\{0\}$ is discovered, vertex 0 is popped; similarly when the second component $\{5, 6, 7\}$ is found, its vertices are popped; finally all vertices are popped when the third component $\{1, 2, 3, 4, 8, 9\}$ is found. Notice that there is no cross-edge to a vertex with a number less than 5 when the second component is discovered. Similarly in the first component, there is no edge to a vertex with a number less than 0. In the third component, there is no edge to a vertex less than 1 since we have set the serial number of vertex 0 to $+\infty$ when 0 was popped.

an environment e. It calls *dfs1* on non-visited vertices in r and returns a pair consisting of an integer and the modified environment. The integer is the minimum of the values computed by *dfs1* on non-visited vertices in r and the serial numbers of already visited vertices in r. If the set of roots is empty, the returned integer is $+\infty$.

The main procedure *tarjan* initializes the environment with an empty stack, an empty set of sccs, the fresh serial number 0 and the constant function giving the number -1 to each vertex. The result is the set of components returned by the function *dfs* called on all vertices in the graph.

```
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)

let tarjan () =
  let e = {stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in e'.sccs
```

In the body of *dfs1*, the auxiliary function *add_stack_incr* updates the environment by pushing x on the stack, assigning it the current fresh serial number, and incrementing that number in view of future calls. The function *dfs1* performs a recursive call to *dfs* for the adjacent vertices of x as roots and the updated environment. If the returned integer value *n1* is less than the number *n0* assigned to x, the function simply returns *n1* and the current environment. Otherwise, the function declares that a new scc has been found, consisting of all vertices that are contained on top of x in the current stack. Therefore the stack is popped until x; the popped vertices are stored as a new set in *e.sccs*; and their serial numbers are all set to $+\infty$, ensuring that they do not interfere with future calculations of min values. The auxiliary functions *split* and *set_infty* are used to carry out these updates.

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

```
let rec set_infty s f = match s with Nil → f
  | Cons x s' → (set_infty s' f)[x ← +∞]  end
let rec split x s = match s with Nil → (Nil, Nil)
  | Cons y s' → if x = y then (Cons x Nil, s')
    else let (s1', s2) = split x s' in (Cons y s1', s2)  end
```

Figure 1 illustrates the behavior of the algorithm by an example. We presented the algorithm as a functional program, using data structures available in the Why3 standard library [3]. For lists we have the constructors *Nil* and *Cons*; the function *elements* returns the set of elements of a list. For finite sets, we have the empty set *empty*, and the functions *add* to add an element to a set, *remove* to remove an element from a set, *choose* to pick an arbitrary element in a (non-empty) set, and *is_empty* to test for emptiness. We also use maps with functions *const* denoting the constant function, `_[_]` to access the value of an element, and `_[_ ← _]` for creating a map obtained from an existing map by setting an element to a given value.

For a correspondence between our presentation and the imperative programs used in standard textbooks, the reader is referred to [8]. The present version can be directly translated into Coq or Isabelle functions, and it respects the linear running time behaviour of the algorithm, since vertices could be easily implemented by integers, $+\infty$ by the cardinal of *vertices*, finite sets by lists of integers and mappings by mutable arrays (see for instance [7]).

Thus for each environment e in the algorithm, the working stack *e.stack* corresponds to a cut of the spanning forest where strongly connected components to its left are pruned and stored in *e.sccs*. In this stack, any vertex can reach any vertex higher in the stack. And if a vertex is a base of an scc, no cross-edge can reach some vertex lower than this base in the stack, otherwise that last vertex would be in the same scc with a strictly lower serial number.

We therefore have to organize the proofs of the algorithm around these arguments. To maintain these invariants we will distinguish, as is common for DFS algorithms, three sets of vertices: white vertices are the non-visited ones, black vertices are those that are already fully visited, and gray vertices are those that are still being visited. Clearly, these sets are disjoint and white vertices can be considered as forming the complement in *vertices* of the union of the gray and black ones.

The previously mentioned invariant properties can now be expressed for vertices in the stack: no such vertex is white, any vertex can reach all vertices higher in the stack, any vertex can reach some gray vertex lower in the stack. Moreover, vertices in the stack respect the numbering order, i.e. a vertex x is lower than y in the stack if and only if the number assigned to x is strictly less than the number assigned to y.

3 The proof in Why3

The Why3 system comprises the programming language WhyML used in previous section and a many sorted first-order logic with inductive data types and inductive predicates to express the logical assertions. The system generates proof obligations w.r.t. the assertions, pre- and post-conditions and lemmas inserted in the WhyML program. The system is interfaced with off-the-shelf automatic provers and interactive proof assistants.

From the Why3 library, we use pre-defined theories for integer arithmetic, polymorphic lists, finite sets and mappings. There is also a small theory for paths in graphs. Here we define graphs, paths and sccs as follows.

```
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 y: vertex) = ∃l. path x l y
predicate in_same_scc (x y: vertex) = reachable x y ∧ reachable y x
predicate is_subscc (s: set vertex) =  ∀x y. mem x s → mem y s → in_same_scc x y
predicate is_scc (s: set vertex) = not is_empty s
  ∧ is_subscc s ∧ (∀s'. subset s s' → is_subscc s' → s == s')
```

where *mem* and *subset* denote membership and the subset relation for finite sets.

We add two ghost fields in environments for the black and gray sets of vertices. These fields are used in the proofs but not used in the calculation of the sccs, which is checked by the type-checker of the language.[1]

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

The functions now become:

```
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
    (+∞, {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})
with dfs r e = ... (* unmodified *)
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
```

with a new function *add_black* turning a vertex from gray to black and the modified *add_stack_incr* adding a new gray vertex with a fresh serial number to the current stack.

```
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}
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]}
```

The main invariant ($\mathcal{I}$) of our program states that the environment is well-formed:

```
predicate wf_env (e: env) =
  let {stack = s; black = b; gray = g} = e in
  wf_color e ∧ wf_num e ∧ simplelist s ∧ no_black_to_white b g ∧
  (∀x y. lmem x s → lmem y s → e.num[x] ≤ e.num[y] → reachable x y) ∧
  (∀y. lmem y s → ∃x. mem x g ∧ e.num[x] ≤ e.num[y] ∧ reachable y x) ∧
  (∀cc. mem cc e.sccs ↔ subset cc b ∧ is_scc cc)
```

where *lmem* stands for membership in a list. The well-formedness property is the conjunction of seven clauses. The two first clauses express elementary conditions about the colored sets of vertices and the numbering function (see [7, 8] for a detailed description). The third clause states that there are no repetitions in the stack, and the fourth that there is no edge from a

[1] In Why3-1.2.0, this check is performed differently

black vertex to a white vertex. The next two clauses formally express the property already stated above: any vertex in the stack reaches all higher vertices and any vertex in the stack can reach a lower gray vertex. The last clause states that the *sccs* field is the set of all sccs all of whose vertices are black.

Since at the end of the *tarjan* function, all vertices are black, the *sccs* field will contain exactly the set of all strongly connected components.

```
let tarjan () = returns{r → ∀cc. mem cc r ↔ subset cc vertices ∧ is_scc cc}
  let e = {black = empty; gray = empty;
    stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in assert{subset vertices e'.black};
    e'.sccs
```

Our functions *dfs1* and *dfs* modify the environment in a monotonic way. Namely they augment the set of visited vertices (the black ones); they keep invariant the set of the ones currently under visit (the gray set); they increase the stack with new black vertices; they also discover new sccs and they keep invariant the serial numbers of vertices in the stack,

```
predicate subenv (e e': env) =
  subset e.black e'.black ∧ e.gray == e'.gray
  ∧ (∃s. e'.stack = s ++ e.stack ∧ subset (elements s) e'.black)
  ∧ subset e.sccs e'.sccs ∧ (∀x. lmem x e.stack → e.num[x] = e'.num[x])
```

Once these invariants are expressed, it remains to locate them in the program text and to add assertions which help to prove them. The pre-conditions of *dfs1* are quite natural: the vertex x must be a white vertex of the graph, and it must be reachable from all gray vertices. Moreover invariant ($\mathcal{I}$) must hold. The post-conditions of *dfs1* are of three kinds. Firstly ($\mathcal{I}$) and the monotony property *subenv* hold in the resulting environment. Vertex x is black at the end of *dfs1*. Finally we express properties of the integer value n returned by this function which should be *LOWLINK*(x) as noted previously. In this proof, we give three implicit properties for characterizing n. First, the returned value is never higher than the number of x in the final environment. Secondly, the returned value is either $+\infty$ or the number of a vertex in the stack reachable from x. Finally, if there is an edge from a vertex y' in the new part of the stack to a vertex y in its old part, the resulting value n must be lower or equal to the serial number of y.

```
let rec dfs1 x e  =
(* pre-condition *)
requires{mem x vertices ∧ not mem x (union e.black e.gray)}
requires{∀y. mem y e.gray → reachable y x}
requires{wf_env e} (* I *)
(* post-condition *)
returns{(_, e') → wf_env e' ∧ subenv e e'}
returns{(_, e') → mem x e'.black}
returns{(n, e') → n ≤ e'.num[x]}
returns{(n, e') → n = +∞ ∨ num_of_reachable_in_stack n x e'}
returns{(n, e') → ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
```

The auxiliary predicates used above are formally defined in the following way.

```
predicate num_of_reachable_in_stack (n: int) (x: vertex)(e: env) =
     ∃y. lmem y e.stack ∧ n = e.num[y] ∧ reachable x y
predicate xedge_to (s1 s3: list vertex) (y: vertex) =
     (∃s2. s1 = s2 ++ s3 ∧ ∃y'. lmem y' s2 ∧ edge y' y) ∧ lmem y s3
```

Notice that the definition of *xedge_to* fits the definition of *LOWLINK* when the cross edge ends at a vertex residing in the stack before the call of *dfs1*. The pre- and post-conditions for the function *dfs* are quite similar up to a generalization to sets of vertices considered as the roots of the algorithm (see [7]).

We now add seven assertions in the body of the *dfs1* function to help the automatic provers. In contrast, the function *dfs* needs no extra assertions in its body. In *dfs1*, when the number *n0* of x is strictly greater than the number *n1* resulting from the call to its successors, the first assertion states that *n1* cannot be $+\infty$; it helps proving the next assertion. The second assertion states that a lower gray vertex is reachable from x and that thus the scc of x is not fully black at end of *dfs1*. In that assertion the inequality $y \neq x$ is redundant, but helps showing the *sccs* constraint at the end of *dfs1*. When $n1 \geq n0$, the next four assertions show that the strongly connected component *elements s2* of x is on top of x in the current stack and that then x is the base of that scc. The seventh assertion helps proving that the coloring constraint is preserved at the end of *dfs1*.

```
let n0 = e.sn in
let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
  assert{n1 ≠ +∞};
  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_subscc (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})
```

where *inter* is set intersection, and *is_last* is defined below.

```
predicate is_last (x: α) (s: list α) = ∃s'. s = s' ++ Cons x Nil
```

All proofs are discovered by the automatic provers except for two proofs carried out interactively in Coq. One is the proof of the black extension of the stack in case $n1 < n0$. The provers could not work with the existential quantifier, although the Coq proof is quite short. The second Coq proof is the fifth assertion in the body of *dfs1*, which asserts that any y in the scc of x belongs to *s2*. It is a maximality assertion which states that the set *elements s2* is a complete scc. The proof of that assertion is by contradiction. If y is not in *s2*, there must be an edge from *x'* in *s2* to some *y'* not in *s2* such that x reaches *x'* and *y'* reaches y. There are three cases, depending on the position of *y'*. Case 1 is when *y'* is in *sccs*: this is not possible since x would then be in *sccs* which contradicts x being gray. Case 2 is when *y'* is an element of *s3*: the serial number of *y'* is strictly less than the one of x which is *n0*. If $x' \neq x$, the cross-edge from *x'* to *y'* contradicts $n1 \geq n0$ (post-condition 5); if $x' = x$, then *y'* is a successor of x and again it contradicts $n1 \geq n0$ (post-condition 3). Case 3 is when *y'* is white, then $x' \neq x$ is impossible since *x'* is then black in *s2* and would be the origin of a black-to-white edge to *y'*; if $x' = x$, then *y'* is not white by post-condition 2 of *dfs*.

Some quantitative information about the Why3 proof is listed in table 1. Alt-Ergo 2.3 and CVC4 1.5 proved the bulk of the proof obligations.[2] The proof uses 49 lemmas that were all proved automatically, but with an interactive interface providing hints to apply inlining, splitting, or induction strategies. This includes 13 lemmas on sets, 16 on lists, 5 on lists

[2] In addition to the results reported in the table, Spass was used to discharge one proof obligation.

provers	Alt-Ergo	CVC4	E-prover	Z3	#VC	#PO
49 lemmas	1.91	26.11	3.33		70	49
split	0.09	0.16			6	6
add_stack_incr	0.01				1	1
add_black	0.02				1	1
set_infty	0.03				1	1
dfs1	77.89	150.2	19.99	13.67	79	20
dfs	4.71	3.52		0.26	58	25
tarjan	0.85				15	5
total	85.51	179.99	23.32	13.93	231	108

Table 1 Performance results with provers in Why3-0.88.3 (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.

without repetitions, 3 on paths, 5 on sccs and 7 very specialized lemmas directly involved in the proof obligations of the algorithm. Among the lemmas, a critical one is the lemma *xpath_xedge* on paths which reduces a predicate on paths to a predicate on edges. In fact, most of the Why3 proof works on edges which are handled more robustly by the automatic provers than paths. Another important lemma is *subscc_after_last_gray* which shows that the stack elements on top of the last gray vertex form a subset of an scc. This means that another program with the *split* call before the if-statement would make a simpler proof, but it would be a non-linear-time program. The two Coq proofs are only 9 and 81 lines long (the Coq files of 677 and 680 lines include preambles that are automatically generated during the translation from Why3 to Coq). The interested reader is refered to [7] where the full proof is available.

The proof explained so far only showed the partial correctness of the algorithm. But after adding two lemmas about union and difference for finite sets, termination is automatically proved by the following lexicographic ordering on the number of white vertices and roots.

```
let rec dfs1 x e  = variant{cardinal (diff vertices (union e.black e.gray)), 0}
with dfs r e = variant{cardinal (diff vertices (union e.black e.gray)), 1, cardinal r}
```

4 The proof in Coq

Coq is based on type theory and the calculus of constructions, a higher order lambda-calculus, for expressing formulae and proofs. Some basic notions of graph theory are provided by the Mathematical Components Library [18]. Our formalization is parameterized by a finite type V for the vertices and the function *successors* such that *successors* x is the adjacency set of any vertex x. The boolean *gconnect* x y indicates that a path connects the vertex x to the vertex y. It is straightforward to define the set *gsccs* of the sccs using *gconnect*. Components are represented as sets of sets (*{set {set V}}*). We use library operations for creating singletons (*[set x]*), taking unions ($S_1 \cup S_2$), differences ($S_1 \setminus S_2$), complements ($\sim: S$), and unions of all sets of a set of sets (*cover* S).

Coq proposes several mechanisms to put together properties (boolean conjunction, propositional conjunction, record, inductive family) that have their own specificities. In order to make the presentation more readable for a non-Coq expert, we write them all with the propositional conjunction $[\wedge\ \texttt{P}_1, \ldots \ \&\ \texttt{P}_n]$. We refer to [9] for the actual code.

The Coq proof differs from the one in Why3: it uses natural numbers only and does not mention colors (white, gray and black). In particular, the number ∞ is defined as the cardinality of V, vertices with ∞`.+1` as serial number correspond to the white vertices of the previous section and the environment is defined as a record with only two fields, a set of sccs and the mapping assigning serial numbers to vertices:

```
Record env := Env {esccs : {set {set V}}; num: {ffun V → nat}}.
```

Given an environment e, the set of visited vertices is *visited* e (the vertices with serial number less or equal to ∞), the current fresh serial number is *sn* e (the cardinal of visited vertices), and the stack is *stack* e (the list of elements x which satisfy *num* e x $<$ *sn* e, sorted by increasing serial number).

Another difference with the Why3 algorithm is the disentanglement of the mutually recursive function *tarjan* into two separate functions The first one *dfs1* treats a vertex x and the second one *dfs* a set of vertices *roots* in an environment e.

```
Definition dfs1 dfs x e :=
  let: (n1, e1) as res := dfs (successors x) (visit x e) in
  if n1 < sn e then res else (∞, store (stack e1 \ stack e) e1).

Definition dfs dfs1 dfs (roots : {set V}) e :=
 if [pick x in roots] isn't Some x then (∞, e) else
 let: (n1, e1) := if num e x ≤ ∞ then (num e x, e) else dfs1 x e in
 let: (n2, e2) := dfs (roots \ [set x]) e1 in (minn n1 n2, e2).
```

where *visit* x e produces the environment where x gets the next serial number, *store* stores a new strongly connected component.

Then, the two functions are glued together in a recursive function *rec* where the parameter k controls the maximal recursive height.

```
Fixpoint rec k r e := if k is k'.+1 then dfs (dfs1 (rec k')) (rec k') r e else (∞, e).
```

If k is not zero (i.e. it is a successor of some *k'*), *rec* calls *dfs* taking care that its parameters can only use recursive calls to *rec* with a smaller recursive height, here *k'*. This ensures termination. A dummy value is returned in the case where k is zero. Finally, the top level *tarjan* calls *rec* with the proper initial arguments.

```
Definition tarjan := let: (_, e) := rec (∞ * ∞.+2) V (Env ∅ [ffun ⇒ ∞.+1]) in esccs e.
```

Initially, the roots are all the vertices (V) and the environment has no component and all vertices are not visited (their number is `∞.+1`). As both *dfs* and *dfs1* cannot be applied more than the number of vertices, the value `∞ * ∞.+2` encodes the lexicographic product of the two maximal heights. It gives *rec* enough fuel to never encounter the dummy value so *tarjan* correctly terminates the computation. This allows us to separate the proof of the termination from the algorithm itself, and this last statement is of course proved formally later and named *rec_terminates*.

The invariants of the Coq proof are usually shorter than in the Why3 proof since they do not mention colors. We first define well-formed environments and their valid extension:

```
Definition wf_env e := [∧ esccs e ⊆ gsccs,
 ∀x, num e x < ∞ → num e x < sn e,
 ∀x, (num e x = ∞) = (x ∈ cover (esccs e))  &
 ∀x y, num e x ≤ num e y < sn e → gconnect x y].

Definition subenv e1 e2 := [∧ esccs e1 ⊆ esccs e2,
 ∀x, num e1 x < ∞ → num e2 x = num e1 x  & ∀x, num e2 x < sn e1 → num e1 x < sn e1].
```

Then we state that new visited vertices are the ones reachable by paths accessible from roots with non-visited vertices (i.e. by white paths in the colored setting). The function *nexts* such that *nexts* D X returns the set of vertices reachable from the set X by a path which only contains vertices in D except maybe the last one.

```
Definition outenv (roots : {set V}) (e e' : env) := [∧
  ∀x y, x ∈ stack e' \ stack e → y ∈ stack e' \ stack e → gconnect x y,
  ∀x, x ∈ stack e' \ stack e → ∃y, y ∈ stack e ∧ gconnect x y &
  visited e' = visited e ∪ nexts (~: visited e) roots ].
```

The post-condition is the conjunction of these three properties and the characterization of the output rank:

```
Definition dfs_spec (ne' : nat * env) (roots : {set V}) e := let: (n, e') := ne' in
  [∧ n = \min_(x in nexts (~: visited e) roots) inord (num e' x),
     wf_env e', subenv e e' & outenv roots e e'].
```

Here, the argument *ne'* is the result of a *dfs*. The output rank *n* is the minimum of the serial numbers of the vertices which can be reached from the roots through a path where all the vertices except maybe the last one were not already visited. Note that this characterization differs from the notion of *LOWLINK* which requires that the last vertex was visited.

Finally, we express correctness as the implication between pre- and post-conditions:

```
Definition dfs_correct dfs (roots : {set V}) e := wf_env e →
  (∀x y, x ∈ stack e → y ∈ roots → gconnect x y) → dfs_spec (dfs roots e) roots e.
Definition dfs1_correct dfs1 x e := wf_env e → x ∉ visited e →
  (∀x y, x ∈ stack e → y ∈ [set x] → gconnect x y) → dfs_spec (dfs1 x e) [set x] e.
```

These invariants are expressed differently from the formulation in Why3, but they reflect essentially the same ideas. Rephrasing the invariants made it possible to reduce by approximately 50% the size of the Coq proofs. The two central theorems are:

```
Lemma dfsP dfs1 dfsrec (roots : {set V}) e : (∀x, x ∈ roots → dfs1_correct dfs1 x e) →
   (∀x, x ∈ roots → ∀e1, subenv e e1 → dfs_correct dfsrec (roots \ [set x]) e1) →
  dfs_correct (dfs dfs1 dfsrec) roots e.

Lemma dfs1P dfs x e : dfs_correct dfs (successors x) (visit x e) →
  dfs1_correct (dfs1 dfs) x e.
```

They state that *dfs* and *dfs1* are correct if their respective recursive calls are correct. The proof of the first lemma is straightforward since *dfs* simply iterates on a list. It mostly requires book-keeping between what is known and what needs to be proved. This is done in about 54 lines. The second one is more intricate and requires 124 lines. Gluing these two theorems together and proving termination gives us an extra 12 lines to prove the theorem

```
Theorem rec_terminates k (roots : {set V}) e :
  k ≥ #|~: visited e| * ∞.+1 + #|roots| → dfs_correct (rec k) roots e.
```

The correctness of *tarjan* follows directly in 19 lines of straightforward proof.

```
Theorem tarjan_correct : tarjan = gsccs.
```

We now provide some quantitative information. The Coq contribution is composed of two files. The *extra_nocolors* file defines the *bigmin* operator and some notions of graph theory that we intend to add to Mathematical Components. This file is 294 lines long. The main file is *tarjan_nocolors* and is 605 lines long. It is compiled in 12 seconds with a memory footprint of 800 Mb (3/4 of which is resident) on a Intel® i7 2.60GHz quad-core laptop running Linux. The proofs are performed in the SSReflect proof language [14] with very little automation. The proof script is mostly procedural, alternating book-keeping tactics (*move*) with transformational ones (mostly *rewrite* and *apply*), but often intermediate steps

Number of lines	1	2	3	4	5	6	11	12	16	19	54	124
Number of proofs	19	7	5	2	1	2	2	1	1	1	1	1

Table 2 Distribution of the numbers of lines of the 43 proofs in the file *tarjan_nocolors*.

are explicitly declared with the *have* tactic. There are more than fifty of such intermediate steps in the 320 lines of proof of the file *tarjan_nocolors*. Table 2 gives the distribution of the numbers of lines of these proofs. Most of them are very short (26 are less than 2 lines) and the only complicated proof is the one corresponding to the lemma *dfs1P*.

5 The proof in Isabelle/HOL

Isabelle/HOL [21] is the encoding of simply typed higher-order logic in the logical framework Isabelle [23]. Unlike Why3, it is not primarily intended as an environment for program verification and does not contain specific syntax for stating pre- and post-conditions or intermediate assertions in function definitions. Logics and formalisms for program verification have been developed within Isabelle/HOL (e.g., [16]), but they target imperative rather than functional programming, so we simply formalize the algorithm as an Isabelle function. Isabelle/HOL provides an extensive library of data structures and proofs. In this development we mainly rely on the set and list libraries. We start by introducing a *locale*, fixing parameters and assumptions for the remainder of the proof. We explicitly assume that the set of vertices is finite.

```
locale graph =
  fixes vertices :: ν set  and  successors :: ν ⇒ ν set
  assumes finite vertices  and  ∀v ∈ vertices. successors v ⊆ vertices
```

We introduce reachability in graphs using an inductive predicate definition, rather than via an explicit reference to paths as in the Why3 definition. Isabelle then generates appropriate induction theorems for use in proofs.

```
inductive reachable where
  reachable x x
| ⟦y ∈ successors x; reachable y z⟧ ⟹ reachable x z
```

The definition of strongly connected components mirrors that used in Why3. The following lemma states that SCCs are disjoint; its one-line proof is found automatically using *Sledgehammer* [2], which heuristically selects suitable lemmas from the set of available facts (including Isabelle's library), invokes several automatic provers, and finally reconstructs a proof that is checked by the Isabelle kernel.

```
lemma scc-partition:
  assumes is-scc S and is-scc S' and x ∈ S ∩ S'
  shows   S = S'
```

Environments are represented by records, similar to the formalization in Why3, except that there is no distinction between regular and "ghost" fields. Also, the definition of the well-formedness predicate closely mirrors that used in Why3.[3]

```
record ν env =
  black :: ν set      gray :: ν set
  stack :: ν list     sccs :: ν set set      sn :: nat      num :: ν ⇒ int
definition wf_env where wf_env e ≡
  wf_color e ∧ wf_num e ∧ distinct (stack e) ∧ no_black_to_white e
```

[3] We use the infix operator $\preceq$ to denote precedence in lists.

```
∧ (∀x y. y ⪯ x in (stack e) ⟶ reachable x y)
∧ (∀y ∈ set (stack e).  ∃g ∈ gray e. y ⪯ g in (stack e)  ∧  reachable y g)
∧ sccs e = { C . C ⊆ black e ∧ is_scc C }
```

The definition of the two mutually recursive functions *dfs1* and *dfs* again closely follows their representation in Why3.

```
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) ))
```

The **function** keyword introduces the definition of a recursive function. Isabelle checks that the definition is well-formed and generates appropriate simplification and induction theorems. Because HOL is a logic of total functions, it introduces two proof obligations: the first one requires the user to prove that the cases in the function definitions cover all type-correct arguments; this holds trivially for the above definitions. The second obligation requires exhibiting a well-founded ordering on the function parameters that ensures the termination of recursive function invocations, and Isabelle provides a number of heuristics that work in many cases. However, the functions defined above will in fact not terminate for arbitrary calls, in particular for environments that assign sequence number -1 to non-white vertices. The *domintros* attribute instructs Isabelle to consider these functions as "partial". More precisely, it introduces an explicit predicate representing the domains for which the functions are defined. This "domain condition" appears as a hypothesis in the simplification rules that mirror the function definitions so that the user can assert the equality of the left- and right-hand sides of the definitions only if the domain predicate holds. Isabelle also proves (mutually inductive) rules for proving when the domain condition is guaranteed to hold. Our first objective is therefore to establish sufficient conditions that ensure the termination of the two functions. Assuming the domain condition, we prove that the functions never decrease the set of colored vertices and that vertices are never explicitly assigned the number -1 by our functions. Denoting the union of gray and black vertices as *colored*, we introduce the predicate

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

and show that this predicate is an invariant of the functions. We then prove that the triple defined as

```
  (vertices - colored e, {x}, 1)
  (vertices - colored e, roots, 2)
```

for the arguments of *dfs1* and *dfs*, respectively, decreases w.r.t. lexicographical ordering on finite subset inclusion and $<$ on natural numbers across recursive function calls, provided that *colored_num* holds when the function is called and x is a white vertex. These conditions are therefore sufficient to ensure that the domain condition holds:[4]

```
theorem dfs1_dfs_termination:
  ⟦x ∈ vertices - colored e; colored_num e⟧ ⟹ dfs1_dfs_dom (Inl(x,e))
  ⟦roots ⊆ vertices; colored_num e⟧ ⟹ dfs1_dfs_dom (Inr(roots,e))
```

The proof of partial correctness follows the same ideas as the proof presented for Why3. We define the pre- and post-conditions of the two functions as predicates in Isabelle. For example, the predicates for *dfs1* are defined as follows:

```
definition dfs1_pre where dfs1_pre e ≡
  wf_env e ∧ x ∈ vertices ∧ x ∉ colored e ∧ (∀g ∈ gray e. reachable g x)
definition dfs1_post where dfs1_post x e res ≡
  let n = fst res; e' = snd res
  in  wf_env e' ∧ subenv e e' ∧ roots ⊆ colored e'
    ∧ (∀x ∈ roots. n ≤ num e' x)
    ∧ (n = +∞ ∨ (∃x ∈ roots. ∃y in set (stack e'). num e' y = n ∧ reachable x y))
```

We now show the following theorems:

- The pre-condition of each function establishes the pre-condition of every recursive call appearing in the body of that function. For the second recursive call in the body of *dfs* we also assume the post-condition of the first recursive call.
- The pre-condition of each function, plus the post-conditions of each recursive call in the body of that function, establishes the post-condition of the function.

Combining these results, we establish partial correctness:

```
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(roots,e)); dfs_pre roots e⟧ ⟹ dfs_post roots e (dfs roots e)
```

We define the initial environment and the overall function.

```
definition init_env where init_env ≡
  ⦇ black = {},  gray = {},  stack = [],  sccs = {},  sn = 0,  num = λ_. -1 ⦈
definition tarjan where tarjan ≡
  sccs (snd (dfs vertices init_env))
```

It is trivial to show that the arguments to the call of *dfs* in the definition of *tarjan* satisfy the pre-condition of *dfs*. Putting together the theorems establishing termination and partial correctness, we obtain the desired total correctness results.

```
theorem dfs_correct:
  dfs1_pre x e ⟹ dfs1_post x e (dfs1 x e)
  dfs_pre roots e ⟹ dfs_post roots e (dfs roots e)
theorem tarjan_correct:
  tarjan = { C . is_scc C ∧ C ⊆ vertices }
```

The intermediate assertions appearing in the Why3 code guided the overall proof: they are established either as separate lemmas or as intermediate steps within the proofs of the above theorems. Similarly to the Coq proof, the overall induction proof was explicitly decomposed into individual lemmas as laid out above. In particular, whereas Why3 identifies the predicates that can be used from the function code and its annotation with pre- and post-conditions, these assertions appear explicitly in the intermediate lemmas used in the

[4] Observe that Isabelle introduces a single operator corresponding to the two mutually recursive functions whose domain is the disjoint sum of the domains of both functions.

$i = 1$	$i \leq 5$	$i \leq 10$	$i \leq 20$	$i \leq 30$	$i = 35$	$i = 43$	$i = 48$
28	8	4	1	2	1	1	1

Table 3 Distribution of interactions in the Isabelle proofs.

proof of theorem *dfs_partial_correct*. The induction rules that Isabelle generated from the function definitions were helpful for finding the appropriate decomposition of the overall correctness proof.

Despite the extensive use of *Sledgehammer* for invoking automatic back-end provers, including the SMT solvers CVC4 and Z3, from Isabelle, we found that in comparison to Why3, significantly more user interactions were necessary in order to guide the proof. Although many of those were straightforward, a few required thinking about how a given assertion could be derived from the facts available in the context. Table 3 indicates the distribution of the number of interactions used for the proofs of the 46 lemmas the theory contains. These numbers cannot be compared directly to those shown in Table 2 for the Coq proof because an Isabelle interaction is typically much coarser-grained than a line in a Coq proof. As in the case of Why3 and Coq, the proofs of partial correctness of *dfs1* (split into two lemmas following the case distinction) and of *dfs* required the most effort. It took about one person-month to carry out the case study, starting from an initial version of the Why3 proof. Processing the entire Isabelle theory on a laptop with a 2.7 GHz Intel® Core i5 (dual-core) processor and 8 GB of RAM takes 35 seconds of CPU time.

6 General comments about the proof

Our formal proofs refer to colors, finite sets, and the stack, although the informal correctness argument is about properties of strongly connected components in spanning trees. The algorithmician would explain the algorithm with spanning trees as in Tarjan's article. It would be nice to extract a program from such a proof, but programmers like to understand the proof in terms of variables and data that their program is using.

A first version of the formal proof used *ranks* in the working stack and a flat representation of environments by adding extra arguments to functions for the black, gray, scc sets and the stack. That was perfect for the automatic provers of Why3. But after remodelling the proof in Coq and Isabelle/HOL, it was simpler to gather these extra arguments in records and have a single extra argument for environments. Also *ranks* disappeared in favor of the *num* function and the precedence relation, which are easier to understand. The automatic provers have more difficulties with the inlining of environments, but with a few hints they could still succeed.

Our proof is mainly about the correctness of Tarjan's algorithm. It relies on surprisingly few and elementary concepts of finite graphs. With the exception of the use of the Mathematical Components library for Coq, we therefore did not use existing libraries formalizing advanced concepts of graph theory [11, 22].

Finally, coloring of vertices is usual for graph algorithms. The stack used in our algorithm is also not necessary since it is just used to efficiently output new strongly connected components. The Coq formalization actually shows that proof can be done with just serial numbers and the store of connected components. The stack and current serial number could be added back using a program refinement, in order to recover a linear time computation.

There is always a tension between the concision of the proof, its clarity and its relation to the real program. In our presentation, we have allowed for a few redundancies.

	Why3	Coq	Isabelle/HOL
expressivity	–	+	+
readability	+	–	+
stability	–	+	+
ease of use	–	–	–
automation	+	–	+
ignore termination	+	–	–
trusted base	–	+	+
automatic proof line-count	395	0	314 ui
manual proof line-count	90	898	1690

Table 4 Compared usage of the three formal systems in the case of our three proofs

7 Conclusion

The formal proof expressed in this article was initially designed and implemented in Why3 [8] as the result of a long process, nearly a 2-year half-time work with many attempts of proofs about various graph algorithms (depth first search, Kosaraju strong connectivity, bi-connectivity, articulation points, minimum spanning tree). Why3 has a clear separation between programs and the logic. It makes the correctness proof quite readable for a programmer. Also first-order logic is easy to understand. Moreover, one can prove partial correctness without caring about termination.

Another important feature of Why3 is its interface with various off-the-shelf theorem provers (mainly SMT provers). Thus the system benefits from the current technology in theorem provers and clerical sub-goals can be delegated to these provers, which makes the overall proof shorter and easier to understand. Although the proof must be split in more elementary pieces, this has the benefit of improving its readability. Several hints about inlining or induction reasoning are still needed and two Coq proofs were used. The system records sessions and facilitates incremental proofs. However, the automatic provers are sometimes no longer able to handle a proof obligation after seemingly minor modifications to the formulation of the algorithm or the predicates, making the proof somewhat unstable.

The Coq and Isabelle proofs were inspired by the Why3 proof. Their development therefore required much less time although their text is longer. The Coq proof uses SSReflect and the Mathematical Components library, which helps reduce the size of the proof compared to classical Coq. The proof also uses the bigops library and several other higher-order features which makes it more abstract and closer to Tarjan's original proof.

In Coq, one could prove termination using well-foundedness [1, 4], but because of nested recursion the `Function` command fails, and both `Equations` and `Program Fixpoint` require the addition of an extra proof argument to the function. Instead, we define the functionals *dfs1* and *dfs* and recombine them in *rec* and *tarjan* by recursion on a natural number used as fuel. We prove partial correctness on functionals and postpone termination on *rec*.

Our Coq proof does not use significant automation.[5] All details are explicitly expressed, but many of them were already present in the Mathematical Components library. Moreover, a proof certificate is produced and a functional program could in principle be extracted. The absence of automation makes the system very stable to use since the proof script is explicit, but it requires a higher degree of expertise from the user.

The Isabelle/HOL proof can be seen as a mid-point between the Why3 and Coq proofs. It uses higher order logic and the level of abstraction is close to the one of the Coq proof,

[5] Hammers exist for Coq [10, 12] but unfortunately they currently perform badly when used in conjunction with the Mathematical Components library.

although more readable in this case study. The proof makes use of Isabelle's extensive support for automation. In particular, *Sledgehammer* [2] was very useful for finding individual proof steps. It heuristically selects lemmas and facts available in the context and then calls automatic provers (SMT solvers and superposition-based provers for first-order logic). When one of these provers finds a proof, Sledgehammer attempts to find a proof that can be certified by the Isabelle kernel, using various proof methods such as combinations of rewriting and first-order reasoning (blast, fastforce etc.), calls to the *metis* prover or reconstruction of SMT proofs through the *smt* proof method. Unlike in Why3, the automatic provers used to find the initial proof are not part of the trusted code base because ultimately the proof is checked by the kernel. The price to pay is that the degree of automation in Isabelle is still significantly lower compared to Why3. Adapting the proof to modified definitions was fast: the Isabelle/jEdit GUI eagerly processes the proof script and quickly indicates those steps that require attention.

The Isabelle proof also faces the termination problem to achieve general consistency. We chose to delay handling termination, using the *domintros* attribute. The proofs of termination and of partial correctness are independent; in particular, we obtain a weaker predicate ensuring termination than the one used for partial correctness. Although the basic principle of the termination proof is very similar to the Coq proof and relies on considering functionals of which the recursive functions are fixpoints, the technical formulation is more flexible because we rely on proving well-foundedness of an appropriate relation rather than computing an explicit upper bound on the number of recursive calls.

One strong point of Isabelle/HOL is its nice LaTeX output and the flexibility of its parser, supporting mathematical symbols. Combined with the hierarchical Isar proof language [31], the proof is in principle understandable without actually running the system, although some familiarity with the system is still required.

In the end, the three systems Why3, Coq, and Isabelle/HOL are mature, and each one has its own advantages w.r.t. readability, expressivity, stability, ease of use, automation, partial-correctness, code extraction, trusted base and length of proof (see table 4). Coming up with invariants that are both strong enough and understandable was by far the hardest part in this work. This effort requires creativity and understanding, although proof assistants provide some help: missing predicates can be discovered by understanding which parts of the proof fail. We think that formalizing the proof in all three systems was very rewarding and helped us better understand the state of the art in computer-aided deductive program verification. It could be also interesting to implement this proof in other formal systems and establish comparisons based on this quite challenging example.[6]

Another interesting work would be to verify an implementation of this algorithm with imperative programs and concrete data structures. This will make the proof more complex, since mutable variables and mutable data structures have to be considered. There is support for verifying imperative programs in general-purpose proof assistants [5, 6, 16], and it would be interesting to also develop them simultaneously in various formal systems and to understand how these proofs can be derived from ours.

A final and totally different remark is about teaching of algorithms. Do we want students to formally prove algorithms, or to present algorithms with assertions, pre- and post-conditions, and make them prove these assertions informally as exercises? In both cases, we believe that our work could make a useful contribution.

---

[6] We have set up a Web page `http://www-sop.inria.fr/marelle/Tarjan/contributions.html` in order to collect formalizations.

References

1. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and LOgic Programming Systems (FLOPS'06)*, volume 3945 of *LNCS*, pages 114–129, Fuji Susono, Japan, April 2006.
2. Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Automated Reasoning*, 51(1):109–128, 2013.
3. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.86.1.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.86.1 edition, May 2015. Available at `why3.lri.fr/download/manual-0.86.1.pdf`.
4. Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016.
5. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proc. 16th ACM SIGPLAN Intl. Conf. Functional Programming*, pages 418–430, Tokyo, Japan, 2011. ACM.
6. Arthur Charguéraud. Higher-order representation predicates in separation logic. In *Proc. 5th ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP 2016, pages 3–14, New York, NY, USA, 2016. ACM.
7. Ran Chen and Jean-Jacques Lévy. Full scripts of Tarjan SCC Why3 proof. Technical report, Iscas and Inria, 2017. `jeanjacqueslevy.net/why3`.
8. Ran Chen and Jean-Jacques Lévy. A semi-automatic proof of strong connectivity. In Andrei Paskevich and Thomas Wies, editors, *Proc. 9th Working Conf. Verified Software: Theories, Tools, and Experiments (VSTTE 2017)*, volume 10712 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2017.
9. Cyril Cohen and Laurent Théry. Full script of Tarjan SCC Coq/ssreflect proof, 2017. Available at `https://www-sop.inria.fr/marelle/Tarjan/`.
10. Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.
11. Christian Doczkal, Guillaume Combette, and Damien Pous. A formal proof of the minor-exclusion property for treewidth-two graphs. In Jeremy Avigad and Assia Mahboubi, editors, *Proc. 9th Intl. Conf. Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pages 178–195. Springer, 2018.
12. Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *CAV (2)*, volume 10427 of *LNCS*, pages 126–133. Springer, 2017.
13. Jean-Christophe Filliâtre et al. The Why3 gallery of verified programs. Technical report, CNRS, Inria, U. Paris-Sud, 2015. `toccata.lri.fr/gallery/why3.en.html`.
14. Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.
15. Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *Proc. 40th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL '13, pages 523–536, New York, NY, USA, 2013. ACM.
16. Peter Lammich. Refinement to Imperative/HOL. In Christian Urban and Xingyuan Zhang, editors, *Proc. 6th Intl. Conf. Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 253–269, Nanjing, China, 2015. Springer.
17. Peter Lammich and René Neumann. A framework for verifying depth-first search algorithms. In *Proc. 4th ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP '15, pages 137–146, New York, NY, USA, 2015. ACM.
18. Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: `https://math-comp.github.io/mcb/`, 2016.
19. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In *CADE*, 2003.

20 Stephan Merz. Isabelle formalization of Tarjan's algorithm, 2018. Available at `https://members.loria.fr/SMerz/papers/cpp2019.html`.

21 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.

22 Lars Noschinski. A graph library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015.

23 Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

24 Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.

25 François Pottier. Depth-first search and strong connectivity in Coq. In *Journées Francophones des Langages Applicatifs (JFLA 2015)*, January 2015.

26 Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying concurrent graph algorithms. In Atsushi Igarashi, editor, *Proc. 14th Asian Symp. Programming Languages and Systems (APLAS 2016)*, volume 10017 of *LNCS*, pages 314–334, Hanoi, Vietnam, 2016. Springer.

27 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI '15, pages 77–87, New York, NY, USA, 2015. ACM.

28 Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

29 Laurent Théry. Formally-proven Kosaraju's algorithm. Inria report, Hal-01095533, 2015.

30 Ingo Wengener. A simplified correctness proof for a well-known algorithm computing strongly connected components. *Information Processing Letters*, 83(1):17–19, 2002.

31 Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *12th Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLS'99)*, volume 1690 of *LNCS*, pages 167–184, Nice, France, 1999. Springer.

32 Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, Berlin, Heidelberg, 2006.