Formal verification of an algorithm for computing strongly connected components

Vincent Trélat

Supervized by Stephan Merz

May 27, 2022

***

My warmest and most sincere thanks to Stephan Merz, who has been a great help in the development the whole project and who reviewed this paper with great care.

Contents

1 Preamble

1.1 Academic context

This research work was carried out as part of my curriculum at the French École des Mines de Nancy. All documents such as codes or source papers are available on a GitHub repository.

1.2 Formal methods

Formal methods are a field of computer science related to mathematical logic and reasoning. The whole purpose of the discipline is to give precise, mathematical definitions to computer science concepts. Formal methods find applications in a variety of fields, both concrete, such as the railway industry or self-driving cars, and abstract, such as computational architecture. Although the purpose of giving such definitions is to enable formal verification, many techniques besides theorem proving, such as model-based testing, run-time monitoring, model checking etc. are used.

1.3 Isabelle (HOL)

> "Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus."
>
> isabelle.in.tum.de

Isabelle [4] is a really powerful proof assistant coming with a higher order logic (HOL) proving environment. Isabelle proofs are written in the Isar ("intelligible semi-automated reasoning") language that is designed to make proofs readable and comprehensible for a mathematically inclined reader, with minimal overhead introduced by the formalism. In fact, "assistant" refers to the fact that the machine checks the proof provided by the user, in contrast to automatic theorem proving where the machine finds the proof itself. The tools for automation are intended to help the user write the proof at a conveniently high level, without needing to work at the level of a logical calculus, for example.

2 Introduction

The objective of this project is to mechanize a proof of correctness of a set-based algorithm inspired by Tarjan's algorithm [5] for computing the strongly connected components of a graph. A similar work has been done on Tarjan's algorithm [3]. The algorithm was first published in Vincent Bloemen's thesis [1] who furthermore gives and explains a few invariants, and was later reused in [2] with the aim of working on a parallel version of the algorithm.

In this report, a few arguments are given for the understanding of the formal proof. Some important invariants are explained and the main lemmas are briefly detailed. Some of the proofs will be explicitly given in slightly less rigorous mathematical terms for the sake of better understandability.

3 Formalisation

3.1 Graphs and reachability

Definition 1 (Directed graph)**.** A directed graph $\mathcal{G}$ is the data of a set of nodes $\mathcal{V}$ and a set of oriented edges $\mathcal{E}$.

Definition 2 (Reachability)**.** For two vertices x and y of $\mathcal{V}$, the reachability relation is noted "$\Rightarrow^*$" such that $x \Rightarrow^* y$ iff x can reach y in $\mathcal{G}$.

Remark 1. The relation $\Rightarrow^*$ is in fact the transitive closure of the binary relation $\Rightarrow$ defining edges in a graph.

Definition 3 (Successors set for a node)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $v \in \mathcal{V}$. The set of successors of $v \in \mathcal{V}$ is $\textsc{Post}(v)$ such that:

$$\textsc{Post}(v) = \{w \in \mathcal{V} \mid (v, w) \in \mathcal{E}\}$$

3.2 Strongly connected components

3.2.1 Directed graphs

Definition 4 (SCC)**.** Let $\mathcal{G}$= ($\mathcal{V}$, $\mathcal{E}$) be a directed graph. $\mathcal{C} \subseteq \mathcal{V}$ is a strongly connected component (SCC) of $\mathcal{G}$if:

$$\forall x, y \in \mathcal{C}, (x \Rightarrow^* y) \wedge (y \Rightarrow^* x)$$

i.e. there is a path between every x and y in $\mathcal{C}$.

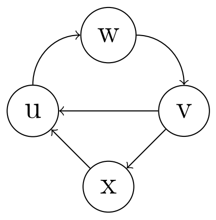
$\mathcal{C}$ is maximal, or $\mathcal{C}$ is a maximal SCC of $\mathcal{G}$ if there is no other SCC containing $\mathcal{C}$, *i.e.* if:

$$\forall \mathcal{X}, (\mathcal{C} \subseteq \mathcal{X}) \wedge (\forall x, y \in \mathcal{X}, (x \Rightarrow^* y) \wedge (y \Rightarrow^* x)) \Longrightarrow \mathcal{C} = \mathcal{X}$$

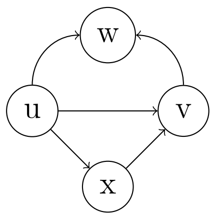
Definition 5. (Strong connectedness) Let $\mathcal{G}$= ($\mathcal{V}$, $\mathcal{E}$) be a directed graph. $\mathcal{G}$ is strongly connected if $\mathcal{V}$ is a SCC.

3.2.2 Examples

Let us give some visual examples of a strongly connected component in a directed graph.



(a) Strongly connected component



(b) Not strongly connected component

Figure 1: Basic example of what is a small SCC

In Figure 1, two small directed graphs are shown. The first one (Figure 1a) is strongly connected, but the second one (Figure 1b) is not because the node x is not reachable from w for instance.

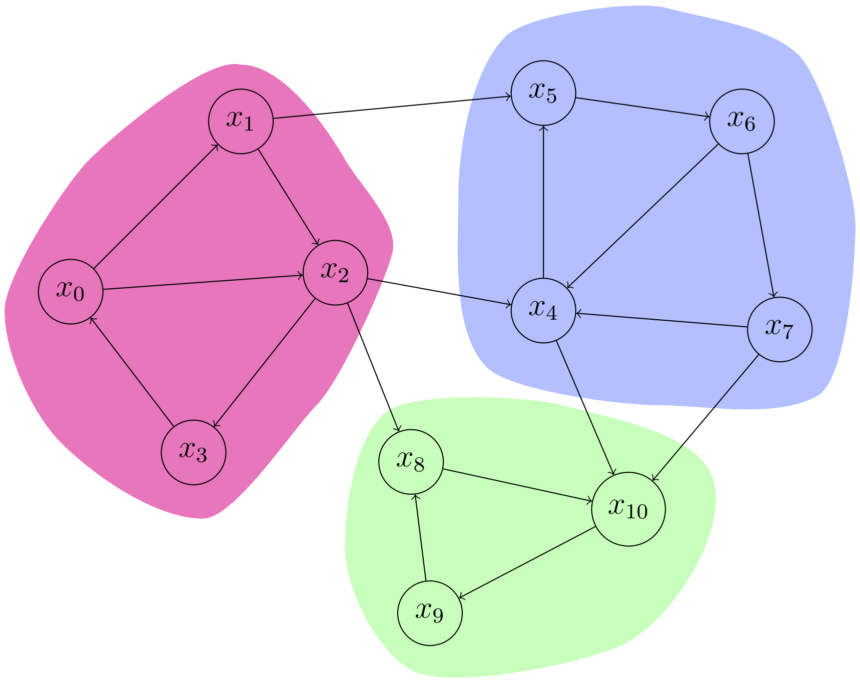


Figure 2: Example of a graph where each colored set of nodes is a – maximal – SCC

Let us give another example on a larger graph. FIGURE 2 shows a directed graph on which each colored set of nodes is a – maximal – SCC. Therefore, one can informally understand that SCCs roughly describe the cycles in a graph.

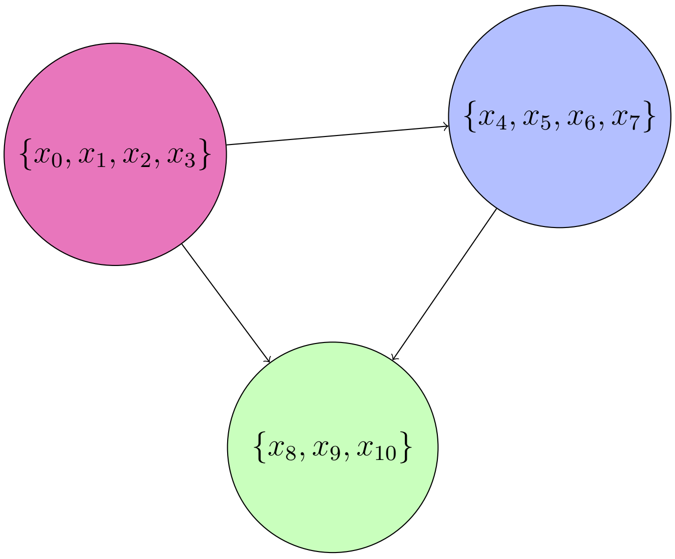


Figure 3: Reduced visualization of the graph represented in FIGURE 2

One may want to have a more general view of and consider only the distinct components of a graph. Thus, the graph show on FIGURE 2 can be reduced to a graph in which the previously colored nodes are replaced by a single node containing all the nodes of the same SCCs, *i.e.* all the equivalent nodes, as shown in FIGURE 3.

4 A sequential set-based algorithm

4.1 Formalisation

Definition 6 (SCC mapping)**.** In the following algorithm, the SCCs are progressively tracked in a collection of disjoint sets through a map $\mathcal{S} : \mathcal{V} \longrightarrow \mathcal{P}(\mathcal{V})$, where $\mathcal{P}(\mathcal{V})$ is the powerset of $\mathcal{V}$, s.t. the following invariant is maintained:

$$\forall v, w \in \mathcal{V}, w \in \mathcal{S}(v) \iff \mathcal{S}(v) = \mathcal{S}(w) \tag{1}$$

Remark 2. In the following, the same notation $\mathcal{S}$ will be used to denote both the function defined above and the induced equivalence relation[1] since $\mathcal{S}$ associates to each node its class of equivalence.

[1]For the relation $(x, y) \mapsto x \in \mathcal{S}(y) \ \wedge \ y \in \mathcal{S}(x)$

Remark 3. In particular, $\forall v \in \mathcal{V}, v \in \mathcal{S}(v)$.

Definition 7 (SCC union)**.** Let UNITE be the function taking as parameters a map $\mathcal{S}$ as defined previously and two vertices u and v of $\mathcal{V}$ such that UNITE$(\mathcal{S}, u, v)$ merges the two mapped sets $\mathcal{S}(u)$ and $\mathcal{S}(v)$ and maintains the invariant (1) by updating $\mathcal{S}$.

Let us give an example:
Let $\mathcal{V} = \{u, v, w\}$ such that there is the following mapping: $\mathcal{S}(u) = \{u\}$ and $\mathcal{S}(v) = \mathcal{S}(w) = \{v, w\}$.
Then, UNITE$(\mathcal{S}, u, v) = \mathcal{S}(u) = \mathcal{S}(v) = \mathcal{S}(w) = \{u, v, w\}$.

4.2 The algorithm

This section gives a pseudo-code of the set-based algorithm for which we will write a formal proof. See [1] for the original paper.

The algorithm only takes as input a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a starting node v_0 as the root of its exploration. It returns a partition `SCCs` of $\mathcal{V}$ being the set of SCCs in the subgraph of $\mathcal{G}$ reachable from v_0, where each element of `SCCs` is a maximal strongly connected components of $\mathcal{G}$. The equivalence relation $\mathcal{S}$ is initialized so that at the beginning, each node is its own SCC. That is, $\mathcal{S}(v) = \{v\}$ for all $v \in \mathcal{V}$, or seen as a set of disjoint sets: $\mathcal{S} = \bigcup\{\{v\} \mid v \in \mathcal{V}\}$.

Algorithm 1: Sequential set-based SCC algorithm

```
Data: $\mathcal{G}= (\mathcal{V}, \mathcal{E})$, $v_0$;
Result: SCCs;
Initialize an empty set EXPLORED;
Initialize an empty set VISITED;
Initialize an empty stack R;
Initialize $\mathcal{S}\colon v \in \mathcal{V} \mapsto \{v\}$;
setBased($v_0$);
function setBased: $v \in \mathcal{V} \rightarrow$ None
    VISITED := VISITED $\cup \{v\}$;
    R.push($v$);
    foreach $w \in$ POST(v) do
        if $w \in$ EXPLORED then
            continue;
        end
        else if $w \notin$ VISITED then
            setBased($w$);
        end
        else
            while $\mathcal{S}(v) \neq \mathcal{S}(w)$ do
                $r$ := R.pop();
                UNITE($\mathcal{S}, r$, R.top());
            end
        end
    end
    if $v$ = R.top() then
        report SCC $\mathcal{S}(v)$;
        EXPLORED := EXPLORED $\cup\, \mathcal{S}(v)$;
        R.pop();
    end
```

4.3 Informal proof

Note that this proof is said informal only because it is not checked by a mechanized proof assistant. Both logical and mathematical arguments developed below are absolutely relevant.

Lemma 1. (First invariant)

$$\forall x, y \in \mathtt{R}, x \neq y \implies \mathcal{S}(x) \cap \mathcal{S}(y) = \varnothing$$

Note the misuse of the set notation $x, y \in \mathtt{R}$ which just means that x and y are in the stack $\mathtt{R}$.

Proof. Let us consider the stacj $\mathtt{R}$ at some step of the algorithm. Let $x = \mathrm{hd}(R)$, *i.e.* the first element of the stack.

Then let $y \in \mathtt{POST}(x)$. If y is already explored, then we can skip it.

Otherwise, if y is not in the visited set, then we explore it by calling the function `setBased` on y and since y is not visited, $\mathcal{S}(y) = \{y\}$.

Finally, if y is in the visited set, it means that either y in on the stack or it has a representative $z \in \mathcal{S}(y)$ on the stack. However, one can understand that the *while* loop below is executed until the equivalence relation is fully updated. In particular, the invariant is maintained, and there is only one representative of $\mathcal{S}(y)$ on the stack, being the earliest visited node in $\mathcal{S}(y)$ the traversal. ■

Remark 4. It is worth noting that the representative of a partial — with respect to the final result — SCC is the earliest node of this SCC to be visited regarding the graph traversal. This ensures that a representative on the stack is in fact the root of its SCC.

Lemma 2.

$$\biguplus_{r \in \mathtt{R}} \mathcal{S}(r) = \textsc{Live} := \textsc{Visited} \setminus \textsc{Explored}$$

Proof. The disjointness of all on-stack partial SCCs is given by lemma 1. Nodes from $\textsc{Visited} \setminus \textsc{Explored}$ have a (unique) representative in $\mathtt{R}$ because they are being processed. So, $\textsc{Live} \subseteq \mathtt{R}$.

By L.6-7 of Algorithm 1, $\textsc{Visited} \subseteq \mathtt{R}$. L.9-10 ensure that no explored node is pushed in $\mathtt{R}$. L.24-25 keep the invariant by unstacking explored nodes from $\mathtt{R}$, so $\mathtt{R} \cap \textsc{Explored} = \varnothing$. Thus, $\mathtt{R} = \textsc{Visited} \setminus \textsc{Explored} = \textsc{Live}$. ■

Corollary 2.1 (Strong version)**.**

$$\forall v \in \textsc{Live}, \exists!\ r \in \mathtt{R} \cap \mathcal{S}(v), \mathcal{S}(v) = \mathcal{S}(r)$$

Proof. Let $v \in \textsc{Live} = \biguplus_{r \in \mathtt{R}} \mathcal{S}(r)$. v is in a unique partial SCC $\mathscr{S} := \mathcal{S}(v)$. Because of lemma 1, there cannot exist $x \neq y \in \mathtt{R}$ s.t. $\mathcal{S}(x) = \mathcal{S}(y) = \mathscr{S}$. Thus, there exists a unique $x \in \mathtt{R}$ s.t. $\mathcal{S}(x) = \mathscr{S}$ (and $x \in \mathtt{R} \cap \mathscr{S}$). ■

Corollary 2.2 (Weak version)**.**

$$\forall v \in \mathcal{V}, \forall w \in \textsc{Post}(v), w \in \textsc{Live} \implies \exists w' \in \mathtt{R}, \mathcal{S}(w') = \mathcal{S}(w)$$

Proof. Holds because of corollary 2.1. ■

Remark 5. In the algorithm 1, this property is maintained by L.16-18. These lines also illustrate how the algorithm "reads" the SCCs. Corollary 2.2 shows that when the mapped representatives of the top two nodes of `R` are united (until $\mathcal{S}(w') = \mathcal{S}(v) = \mathcal{S}(w)$ since w' has a path to v), then all united components are in the same SCC.

Remark 6. Because `R` only contains exactly one representative for each partial SCC (corollary 2.1 and remark 4), after each step of the main loop – *i.e.* the DFS – every partial SCC is actually maximal in the current set of visited nodes.

Theorem 1. The sequential algorithm 1 is correct, *i.e.* it returns the set of maximal SCCs reachable from v_0.

Proof. Holds by remark 6. ■

An example of the execution of the algorithm is given in Appendix, in the section 5.2.

4.4 Prerequisites for the formal proof

Since the informal proof seems to be convincing, the formal – checked automatically – proof can be written in Isabelle (HOL) based on the basis of the reasoning developed above.

4.4.1 Environment setup

The first definitions should be the different structures used in the algorithm. In particular, a record containing all the sets needed and described in the pseudo-code of algorithm 1. The environment has a generic type parameter, which is used to represent the type of the nodes in the graph (often integers):

```
record 'v env =
  𝒮 ::  "'v ⇒ 'v set"
  explored ::  "'v set"
  visited ::  "'v set"
  sccs ::  "'v set set"
  stack ::  "'v list"
```

The following lines define a graph structure and some useful natural relations:

```
locale graph =
  fixes vertices ::  "'v set" and successors ::  "'v ⇒ 'v set"
  assumes vfin:  "finite vertices"
  and sclosed:  "∀x ∈ vertices.  successors x ⊆ vertices"
```

The use of successors instead of an adjacency matrix, for instance, is a consequence of the fact that the algorithm is only concerned with the topological ordering of the nodes. For instance, nodes can represent integers, logical propositions or sets of states in a proving system for example.

4.4.2 Reachability

Now that graphs are defined, the reachability can be defined. Defining an edge is simply some rewriting of being a successor of one node.

```
abbreviation edge where
  "edge x y ≡ y ∈ successors x"
```

Regarding the reachability binary relation, a choice has to be made since there are several ways to define it. In particular, there are two possible keywords, `inductive` and `fun`, respectively for an inductive or recursive definition. If both definitions are valid, the inductive one is kept for the following reasons. Although a recursive definition allows one to do some rewriting in the middle of terms, a recursive definition expresses both the positive and negative information[2] whereas the inductive one only expresses the positive information directly. Therefore, with an inductive definition, the negative information has to be proved. One would be right to argue that it would be more convenient to be able to tell without proving it that two nodes are not reachable from each other, but this does not interest us for the following. Another important point is that there is no datatype for a recursive definition, especially in this case with the transitive closure of the $\Rightarrow^*$ relation. Thus, the choice of the inductive definition is not a choice of simplicity but of necessity. Lastly, Isabelle will generate a simple inductive rule for the proofs split into the reflexive case, which stands in the definition, and the transitive case, which has to be proved.

sm: I don't really understand the prose above. A recursive function definition has to be based on some underlying inductive definition, which is simply not available for graphs. You say this, but the beginning of the paragraph seems to say otherwise. I'd just say that Isabelle provides a construction for inductive predicate definitions, which is appropriate here, and then explain that the two clauses represent the reflexive case and the extension of reachability by prepending an edge.

```
inductive reachable where
  reachable_refl[iff]:  "reachable x x"
| reachable_succ[elim]:  "⟦edge x y; reachable y z⟧ ⟹ reachable x z"
```

[2]In this case, the positive information designates the fact of being reachable and the negative information designates the fact of not being reachable.

sm: I'm afraid that the blue color used for `reachable` here can be confusing, although Isabelle indeed displays the definition in this way. I'd make it black throughout.

In order to be able to use those relations in the proofs later, it is essential to prove a list of lemmas, namely all the different natural properties that Isabelle cannot deduce[3] from nothing[4]. For instance, the following lemmas are essential.

```
lemma succ_reachable:
    assumes "reachable x y" and "edge y z"
    shows"reachable x z"
    using assms by induct auto
```

Mathematical writing: $\forall x, \forall y, \forall z, (x \Rightarrow^* y \wedge y \Rightarrow z) \Longrightarrow x \Rightarrow^* z$

sm: Note that this is the "mirror" of clause `reachable_succ` (appending an edge).

```
lemma reachable_trans:
    assumes y:  "reachable x y" and z:  "reachable y z"
    shows "reachable x z"
    using assms by induct auto
```

Mathematical writing: $\forall x, \forall y, \forall z, (x \Rightarrow^* y \wedge y \Rightarrow^* z) \Longrightarrow x \Rightarrow^* z$

As the formal proofs will enventually deal with strongly connected components, it is also essential to formally define SCCs. For the purpose of the proof, the property of being a SCC is called `sub_scc` and being a *maximal* SCC is called `is_scc` :

```
definition is_subscc where
    "is_subscc S ≡ ∀ x ∈ S. ∀ y ∈ S. reachable x y"
```

Mathematical writing: A set S is a SCC if $\forall x \in S, \forall y \in S, x \Rightarrow^* y$

```
definition is_scc where
    "is_scc S ≡ S ≠ {} ∧ is_subscc S
    ∧ (∀ S'. S ⊆ S' ∧ is_subscc S' ⟶ S' = S)"
```

Mathematical writing: A non-empty SCC S is maximal if for all SCC S', $S \subseteq S' \Longrightarrow S' = S$

Once again, there are some lemmas to prove, such as telling Isabelle when an element can be added to a SCC, or that two vertices that are reachable from each other are in the same SCC, or that two SCCs having a common element are identical.

sm: "Telling Isabelle" sounds like you are introducing an axiom. Rather, these facts are deduced using Isabelle from the above definitions.

[3]That is an abuse of language. The idea is for example that for the moment, there is no formal link between `edge` and `reachable`. The goal is to formalize it so Isabelle is logically able to both use and simplify some results in the proofs.

[4]There is actually a theorem fetcher that is particularly useful to find a basic set of lemmas.

4.4.3 Equivalence relation and graph partition

In the algorithm 1, the SCCs are progressively tracked in `sccs` and the equivalence relation $\mathcal{S}$ is updated with the UNITE function. In Isabelle, a first recursive definition was written as follows:

sm: Note that this is different from the UNITE function introduced earlier, and perhaps rename one of the two. Also, I'd introduce this function together with the `dfs` and `dfss` functions where it is used. Finally, the heading of the subsection doesn't seem to fit the text.

```
function unite ::  "'v ⇒ 'v ⇒ 'v env ⇒ 'v env" where
"unite v w e =
  (if (𝒮 e v = 𝒮 e w) then e
  else let r = hd(stack e);
           r'= hd(tl(stack e));
           joined = 𝒮 e r ∪ 𝒮 e r;
           e'= e⦇
              stack := tl(stack e),
              𝒮 := (λ n.  if n ∈ joined then joined else 𝒮 e n)
           ⦈
  in unite v w e')"
  by pat_completeness auto
```

However, this definition makes the proofs too difficult due to the recursion. An imperative version was therefore written:

```
definition unite ::  "'v ⇒ 'v ⇒ 'v env ⇒ 'v env" where
  "unite v w e ≡
     let pfx = takeWhile (λx.  w ∉ 𝒮 e x) (stack e);
         sfx = dropWhile (λx.  w ∉ 𝒮 e x) (stack e);
         cc = ⋃ {𝒮 e x | x.  x ∈ set pfx ∪ {hd sfx}}
     in e⦇𝒮 := λx.  if x ∈ cc then cc else 𝒮 e x, stack := sfx⦈"
```

The idea of this definition is to create a partition of `stack e = pfx @ sfx` such that `pfx` contains the nodes which are to be merged into $\mathcal{S}$ `e w` and `sfx` contains the root of $\mathcal{S}$ `e w` followed by the rest of the stack. Then, `cc` – which stands for *connected component* – contains all the nodes which are equivalent to `w` in the sub-graph currently explored. The function `takeWhile` applied to a boolean function `P`[5] seen as a property and a list `xs` returns the elements of `xs` which satisfy `P` and stops at the first element not satisfying `P`. The function `dropWhile` is the opposite of `takeWhile`. Both the imperative and recursive versions of `unite` are equivalent.

sm: The two are equivalent within the context of the algorithm, not in general. And the equivalence has not been proved. In fact, the non-recursive (why "imperative"?) definition is intended to be simpler for the proof because it avoids introducing separate pre- and post-conditions for the function and proving such a "contract".

[5] `P :: 'a ⇒ bool`

4.4.4 Ordering relation

In the proof, a precedence relation[6]noted • $\preceq$ • in • will be needed on the stack. Let x and y be two nodes and R be a stack. Informally, x precedes y in R if y was pushed in R before x (see FIGURE 4).

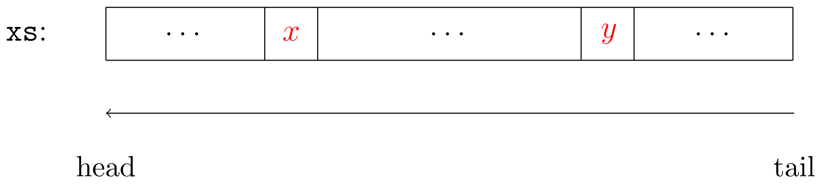


Figure 4: The ordering relation on stacks

Definition 8 (Ordering relation)**.** Let x and y be two nodes and xs be a stack.

$$x \preceq y \text{ in } xs \equiv \exists\ h,\ \exists\ r,\ (xs = h@[x]@r) \wedge (y \in [x]@r)$$

The idea is to later use the following property: if $x \preceq y$ in xs, then $y \Rightarrow^* x$. It is defined in Isabelle as follows:

```
definition precedes ("_ ≼ _ in _" [100,100,100] 39) where
    "x ≼ y in xs ≡ ∃h r.  xs = h @ (x # r) ∧ y ∈ set (x # r)"
```

All the different properties (*i.e.* lemmas) which follow this definition in the Isabelle implementation are detailed in the natural mathematical writing in the Appendix. The right part of the notation represents the orders of priority for each operand since $\preceq$ is an infix operator.

4.4.5 Implementation of the algorithm

Now that the environment is set up, the actual algorithm – seen as a function – can be implemented. Since Isabelle does not support loops, the implementation will be split into two mutually recursive functions. The main function is called `dfs` and takes its name after the Depth First Search algorithm because the algorithm 1 roughly consists in a deep traversal of a graph. The second function is called `dfss` and represents the *foreach* loop of the algorithm 1. The two functions are mutually recursive because they recursively call each other. In particular, `dfss` will call both `dfs` and itself, depending on the case.

```
function dfs ::  "'v ⇒ 'v env ⇒ 'v env" and
         dfss::  "'v ⇒ 'v set ⇒ 'v env ⇒ 'v env" where
```

[6]In fact, a partial order is being defined on stacks.

```
"dfs v e =
    (let e1 = e⦇visited := visited e ∪ {v}, stack := (v # stack e)⦈;
         e' = dfss v (successors v) e1
    in if v = hd(stack e')
         then e'⦇sccs:=sccs e' ∪ 𝒮 e' v, explored:=explored e' ∪ (𝒮 e' v),
stack:=tl(stack e')⦈
         else e')"
| "dfss v vs e =
    (if vs = {} then e
    else (let w = SOME x.  x ∈ vs
         in (let e' = (if w ∈ explored e then e
                  else if w ∉ visited e then dfs w e
                  else unite v w e)
             in dfss v (v - {w}) e')))"
  by pat_completeness (force+)
```

The two last keywords require explanations as well : `pat_completeness` stands for *pattern completeness* and ensures that there is no missing patterns. The keyword `force` is used[7] to help Isabelle know – by proving it – that both `dfs` and `dfss` are actually functions and that those functions are well defined with respect to the usual logical and mathematical meaning.

sm: `force` finishes the proof of pattern completeness, the proof of termination remains open, and it would actually show that these are well-defined functions. Also, you should say that `force` is more aggressive *than* ***auto***.

4.5 Formal proof

4.5.1 General scheme

As the algorithm is composed of two mutually recursive functions, the correctness of the algorithm is proved by induction on the environment structure (cf 4.4.1). Since both `dfs` and `dfss` are quite complex, the proof is split into several parts. The idea is to prove for each function that its execution given some pre-conditions on the input environment implies some post-conditions on the output environment. Then, it has to be made for the mutually recursive calls as well, so that given the same pre-conditions on one function, the pre-conditions on the other function are also satisfied. Finally, it has to be proved that if the pre-conditions are satisfied for one function, and if the pre-conditions imply the post-conditions on the other function, then the post-conditions are also satisfied for the first function.

sm: mutual induction on the functions? There is no induction principle for environments.

[7] `force` is more aggressive in instantiation than `auto` and seems to find the right instance.

4.5.2 Well-formedness of the environment

The whole proof relies on one big invariant regarding the environment structure. It defines the fact for an environment to be well-formed. This invariant is a conjunction of several properties and is defined as follows:

```
definition wf_env where
  "wf_env e ≡
      distinct (stack e)
    ∧ set (stack e) ⊆ visited e
    ∧ explored e ⊆ visited e
    ∧ explored e ∩ set (stack e) = {}
    ∧ (∀ v w.  w ∈ 𝒮 e v ⟷ (𝒮 e v = 𝒮 e w))
    ∧ (∀v ∈ set (stack e).∀ w ∈ set (stack e).v ≠ w ⟶ 𝒮 e v ∩ 𝒮 e w = {})
    ∧ (∀ v.  v ∉ visited e ⟶ 𝒮 e v = {v})
    ∧ ⋃ {𝒮 e v | v.  v ∈ set (stack e)} = visited e - explored e
    ∧ (∀ x y.  x ≼ y in stack e ⟶ reachable y x)
    ∧ (∀ x.  is_subscc (𝒮 e x))
    ∧ (∀ x ∈ explored e.  ∀ y.  reachable x y ⟶ y ∈ explored e)
    ∧ (∀ S ∈ sccs e.  is_scc S)"
```

Let us take a closer look to this invariant, taken in the same order as the definition above:

- First, the stack is a list of distinct elements.
- All elements of the stack are visited.
- The set of explored nodes is a subset of the set of visited nodes.
- Explored nodes cannot be in the stack.
- The three next properties are about the equivalence relation $\mathcal{S}$.
- The union of the sets of equivalent nodes in the stack is equal to the set of visited nodes minus the set of explored nodes.
- A node in the stack can reach all nodes before it in the stack (*i.e.* pushed later).
- $\mathcal{S}$ represents a set of strongly connected components (not maximal).
- For all explored nodes, the sub-graph induced by their successors is totally explored.
- `sccs` is a set of maximal SCCs

These properties are natural and most of them are easy to prove. Actually, there is a bit of redundancy here. For example, the second and fourth conjunct follow from the fifth and eighth. This is not a bad thing per se since it may help automatic proof, but could be discussed.

It is also useful to induce a notion of monotonicity on the environments during the execution of the algorithm. This is defined as follows through the definition of an ordering relation on environments:

```
definition sub_env where
"sub_env e e' ≡
   visited e ⊆ visited e'
 ∧ explored e ⊆ explored e'
 ∧ (∀ v.  𝒮 e v ⊆ 𝒮 e' v)
 ∧ (⋃{𝒮 e v | v.  v ∈ set (stack e)}) ⊆ (⋃{𝒮 e' v | v.  v ∈ set (stack e')})"
```

In particular, the last conjunct expresses the fact that the equivalence relation on the stack is monotonic.

4.5.3 `dfs` pre- and post-conditions

The pre-conditions of `dfs` are rather simple. The environment must be well-formed and the node must not be visited. There is also a condition on the reachability of the nodes in the stack and the node on which the function is called, but once again this condition is rather natural to consider since `dfs` is performing a DFS graph traversal:

```
definition pre_dfs where
 "pre_dfs v e ≡
   wf_env e
 ∧ v ∉ visited e
 ∧ (∀ n ∈ set (stack e).  reachable n v)"
```

The post-conditions are a little more complex since it has to consider the new environment with the new visited / explored nodes, the new state of the stack and the updates in $\mathcal{S}$:

```
definition post_dfs where
"post_dfs v prev_e e ≡
   wf_env e
 ∧ (∀ x.  reachable v x ⟶ x ∈ visited e)
 ∧ sub_env prev_e e
 ∧ (∀ n ∈ set (stack e).  reachable n v)
 ∧ (∃ ns.  stack prev_e = ns @ (stack e))
 ∧ (∀ m n.  m ≼ n in prec_e ⟶
      (∀ u ∈ 𝒮 prev_e m.  reachable u v ∧ reachable v n ⟶ 𝒮 e m = 𝒮 e n))
 ∧ ((v ∈ explored e ∧ stack e = stack prev_e) ∨ v ∈ 𝒮 e (hd (stack e)))"
```

sm: Add some explanations, and continue ...

sm: Don't forget to write a conclusion, explaining what has been done, what is missing / could be improved, and what your experience has been.

5 Appendix

5.1 Some lemmas

Those lemmas refer to the precedence relation introduced in SECTION 4.4.4.

Let x, y, z be three nodes, and let xs, ys, zs be three lists of nodes representing stacks. By abuse of language, if an element is on a stack, it is in the set of elements contained in the stack so the following statement can be written: x is on $xs \iff x \in xs$. However, xs in not seen as the set representing xs since an element may occur several times in a stack. The operator @ denotes the concatenation and operates on two lists: $[x_0, \ldots, x_n]@[y_0, \ldots, y_m] = [x_0, \ldots, x_n, y_0, \ldots, y_m]$.

(i) $x \preceq y \text{ in } xs \implies (x \in xs) \wedge (y \in xs)$

(ii) $y \in [x]@xs \implies x \preceq y \text{ in } ([x]@xs)$

(iii) $x \neq z \implies (x \preceq y \text{ in } ([z]@zs) \implies x \preceq y \text{ in } zs)$

(iv) $(y \preceq x \text{ in } ([x]@xs)) \wedge (x \notin xs) \implies (x = y)$

(v) $y \in (ys@[x]) \implies y \preceq x \text{ in } (ys@[x]@xs)$

(vi) $(x \preceq x \text{ in } xs) = (x \in xs)$

(vii) $x \preceq y \text{ in } xs \implies x \preceq y \text{ in } (ys@xs)$

(viii) $x \notin ys \implies (x \preceq y \text{ in } (ys@xs) \iff x \preceq y \text{ in } xs)$

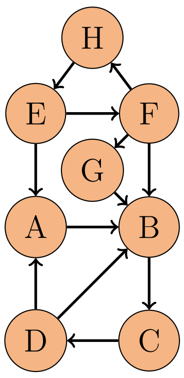
(ix) $x \preceq y \text{ in } xs \implies x \preceq y \text{ in } (xs@ys)$

(x) $y \notin ys \implies x \preceq y \text{ in } (xs@ys) \iff x \preceq y \text{ in } xs$

(xi)(transitivity)
$$(x \preceq y \text{ in } xs) \wedge (y \preceq z \text{ in } xs) \wedge \underbrace{(\forall\ 0 \leq i < j \leq \text{length}(xs), xs[i] \neq xs[j])}_{\text{all elements of } xs \text{ are distinct}} \implies x \preceq z \text{ in } xs$$

(xi)(antisymmetry)
$$(x \preceq y \text{ in } xs) \wedge (y \preceq x \text{ in } xs) \wedge \underbrace{(\forall\ 0 \leq i < j \leq \text{length}(xs), xs[i] \neq xs[j])}_{\text{all elements of } xs \text{ are distinct}} \implies x = y$$

5.2 An example of execution of algorithm 1.

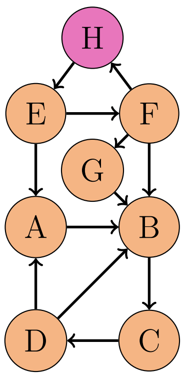


R = []

VISITED = {}

DEAD = {}

$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

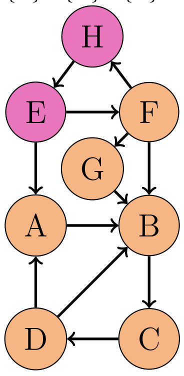


R = [H]

VISITED = {H}

DEAD = {}

$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

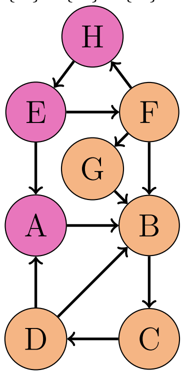


R = [H, E]

VISITED = {H, E}

DEAD = {}

$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

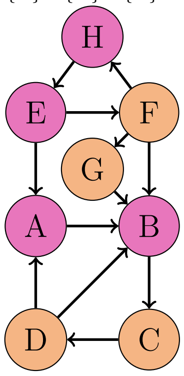


R = [H, E, A]

VISITED = {H, E, A}

DEAD = {}

$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

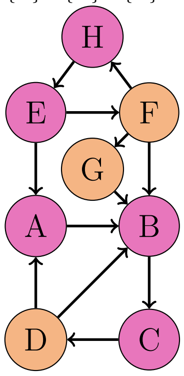


R = [H, E, A, B]

VISITED = {H, E, A, B}

DEAD = {}

$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

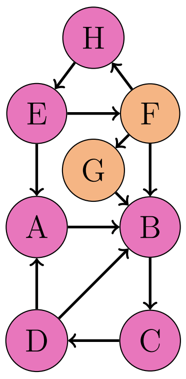


R = [H, E, A, B, C]

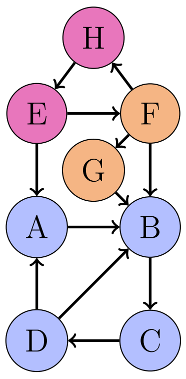
VISITED = {H, E, A, B, C}

DEAD = {}

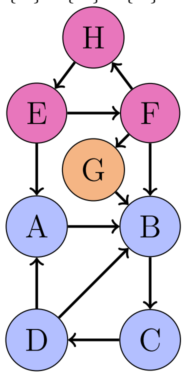
$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



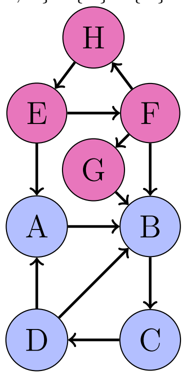
R = [H, E, A, B, C, D]
VISITED = {H, E, A, B, C, D}
DEAD = {}
$\mathcal{S} = \{A\} \cup \{B\} \cup \{C\} \cup \{D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



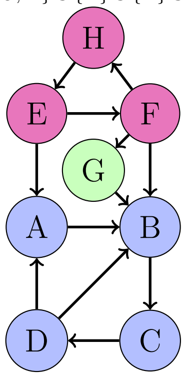
R = [H, E]
VISITED = {H, E, A, B, C, D}
DEAD = {A, B, C, D}
$\mathcal{S} = \{A, B, C, D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



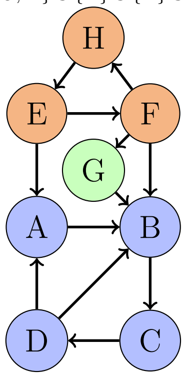
R = [H, E, F]
VISITED = {H, E, A, B, C, D, F}
DEAD = {A, B, C, D}
$\mathcal{S} = \{A, B, C, D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



R = [H, E, F, G]
VISITED = {H, E, A, B, C, D, F, G}
DEAD = {A, B, C, D}
$\mathcal{S} = \{A, B, C, D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



R = [H, E, F]
VISITED = {H, E, A, B, C, D, F, G}
DEAD = {A, B, C, D, G}
$\mathcal{S} = \{A, B, C, D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$



R = []
VISITED = {H, E, A, B, C, D, F, G}
DEAD = {A, B, C, D, G, E, F, H}
$\mathcal{S} = \{A, B, C, D\} \cup \{E, F, H\} \cup \{G\}$

References

[1] V. Bloemen. *Strong connectivity and shortest paths for checking models.* PhD, University of Twente, Enschede, The Netherlands, July 2019. ISBN: 9789036547864.

[2] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. Multi-core on-the-fly SCC decomposition. *ACM SIGPLAN Notices*, 51(8):8:1–8:12, February 2016.

[3] Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. page 18.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, Berlin, Heidelberg, 2002. Book Title: Isabelle/HOL.

[5] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.