Verification in HOL of an algorithm for computing SCCs

Vincent Trélat

École Nationale Supérieure des Mines de Nancy
Département Informatique

January 14, 2022

Definition 1

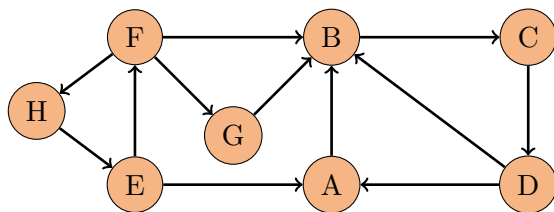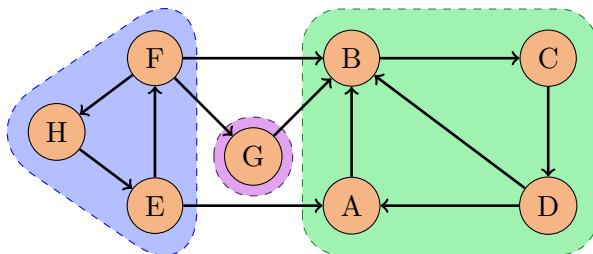Let $\mathcal{G} := (\mathcal{V}, \mathcal{E})$ be a directed graph and $\mathcal{C} \subseteq \mathcal{V}$. $\mathcal{C}$ is a SCC of $\mathcal{G}$ if:

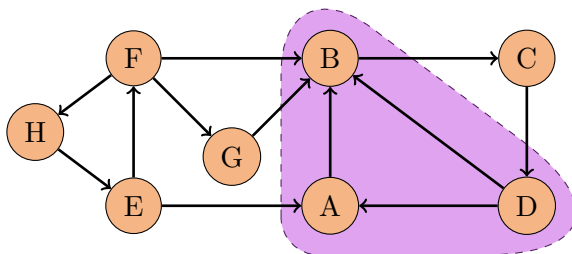$$\forall x, y \in \mathcal{C}, (x \Rightarrow y) \wedge (y \Rightarrow x)$$

Definition 1

Let $\mathcal{G} := (\mathcal{V}, \mathcal{E})$ be a directed graph and $\mathcal{C} \subseteq \mathcal{V}$. $\mathcal{C}$ is a SCC of $\mathcal{G}$ if:

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

- Networks: connection and data sharing
- Model checking: counter-examples finding

- Networks: connection and data sharing
- Model checking: counter-examples finding

Efficient algorithms (ex: Tarjan)

- Formal verification of correctness is worthwhile
- Parallelization is another challenge

Isabelle / HOL

- Generic proof assistant

- Formalisation of mathematical proofs

- Higher-Order Logic theorem proving environment

- Powerful proof tools and language (Isar)

- Mutual induction, recursion and datatypes, complex pattern matching

(Type definition)

```
datatype 'a list = Empty | Cons 'a "'a list"
```

- Generic / polymorphic and static type
- Implicit constructor definition
- Recursive structure giving an induction principle for that type

(Function definition)

```
fun concat ::  "'a list ⇒ 'a list ⇒ 'a list" where
  "concat Empty xs = xs"
| "concat (Cons x xs) ys = Cons x (concat xs ys)"
```

```
fun rev ::  "'a list ⇒ 'a list" where
  "rev Empty = Empty"
| "rev (Cons x xs) = concat (rev xs) (Cons x Empty)"
```

(Theorem statement)

```
theorem rev_rev [simp]:   "rev (rev x) = x"
```

(Theorem statement)

```
theorem rev_rev [simp]:  "rev (rev x) = x"
 apply (induction x)
 apply auto
```

(Theorem statement)

```
theorem rev_rev [simp]:  "rev (rev x) = x"
 apply (induction x)
 apply auto
```

(Subgoal)

```
⋀ x1 x.
  rev (rev x) = x ⟹
  rev (concat (rev x) (Cons x1 Empty) = Cons x1 x
```

(Adding a first lemma)

```
lemma rev_concat [simp]:
"rev (concat xs ys) = concat (rev ys) (rev xs)"
    apply (induction xs)
    apply auto
```

(Adding a first lemma)

```
lemma rev_concat [simp]:
"rev (concat xs ys) = concat (rev ys) (rev xs)"
   apply (induction xs)
   apply auto
```

(Subgoals)

```
1.   rev ys = concat (rev ys) Empty
2.   ⋀ x1 xs.
   rev (concat xs ys) = concat (rev ys) (rev xs) ⟹
   rev (concat (Cons x1 xs) ys) = concat (rev ys) (rev (Cons x1 xs))
```

(Adding a second lemma)

```
lemma concat_empty [simp]:  "concat xs Empty = xs"
  apply (induction xs)
  apply auto
```

(Adding a third lemma: associative property)

```
lemma concat_assoc [simp]:  "concat (concat xs ys) zs =
concat xs (concat ys zs)"
  apply (induction xs)
  apply auto
```

```
theorem rev_rev [simp]:  "rev (rev x) = x"
  apply (induction x)
  apply auto
```

No subgoals!

Midterm presentation of the research course
└─Correctness
  └─Description of the algorithm

1 Introduction
  Definition
  Motivation

2 Example of the proof process

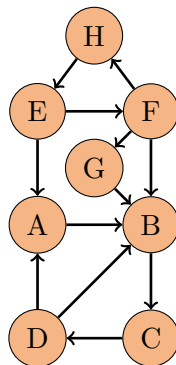**3** A sequential set-based SCC algorithm
  Description of the algorithm
  Implementation in Isabelle

Midterm presentation of the research course
└─ Correctness
    └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 function $setBased: v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   foreach $w \in POST(v)$ do
9   if $w \in DEAD$ then
10   continue;
11   else if $w \notin VISITED$ then
12   setBased(w);
13   else
14   while $\mathcal{S}(v) \neq \mathcal{S}(w)$ do
15   r := R.pop();
16   UNITE($\mathcal{S}, r,$ R.top());

17   if $v = R.top()$ then
18   report SCC $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$$R = \{\}$$
$$VISITED = \{\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1   Initialize an empty set DEAD;
2   Initialize an empty set VISITED;
3   Initialize an empty stack R;
4   setBased(v_0);
5   **function** *setBased:* $v \in \mathcal{V} \to None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;
11   **else if** $w \notin VISITED$ **then**
12   setBased(w);
13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15   r := R.pop();
16   UNITE($\mathcal{S}, r,$ R.top());
17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$$R = \{H\}$$
$$VISITED = \{H\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
　└─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;

1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 **function** *setBased:* $v \in \mathcal{V} \rightarrow None$
6 　　VISITED := VISITED $\cup \{v\}$;
7 　　R.push(v);
8 　　**foreach** $w \in POST(v)$ **do**
9 　　　　**if** $w \in DEAD$ **then**
10 　　　　　　continue;
11 　　　　**else if** $w \notin VISITED$ **then**
12 　　　　　　setBased(w);
13 　　　　**else**
14 　　　　　　**while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15 　　　　　　　　r := R.pop();
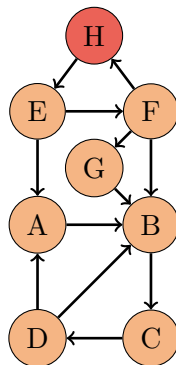16 　　　　　　　　UNITE($\mathcal{S}, r$, R.top());
17 　　**if** $v = R.top()$ **then**
18 　　　　**report SCC** $\mathcal{S}(v)$;
19 　　　　DEAD := DEAD $\cup \mathcal{S}(v)$;
20 　　　　R.pop();

$R = \{H, E\}$

$VISITED = \{H, E\}$

$DEAD = \{\}$

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

Midterm presentation of the research course
└─ Correctness
 └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1   Initialize an empty set DEAD;
2   Initialize an empty set VISITED;
3   Initialize an empty stack R;
4   setBased(v_0);
5   **function** *setBased:* $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;
11   **else if** $w \notin VISITED$ **then**
12   setBased(w);
13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
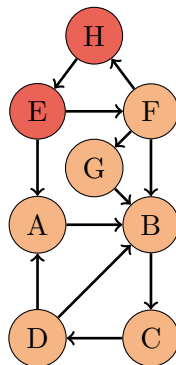15 r := R.pop();
16   UNITE($\mathcal{S}, r,$ R.top());
17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$$R = \{H, E, A\}$$
$$VISITED = \{H, E, A\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1. Initialize an empty set DEAD;
2. Initialize an empty set VISITED;
3. Initialize an empty stack R;
4. setBased(v_0);
5. **function** *setBased*: $v \in \mathcal{V} \rightarrow None$
6.   VISITED := VISITED $\cup \{v\}$;
7.   R.push(v);
8.   **foreach** $w \in POST(v)$ **do**
9.   **if** $w \in DEAD$ **then**
10.   continue;
11.   **else if** $w \notin VISITED$ **then**
12.   setBased(w);
13.   **else**
14.   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
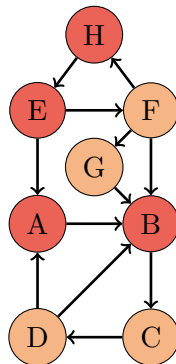15. r := R.pop();
16.   UNITE($\mathcal{S}, r, $R.top());
17.   **if** $v = R.top()$ **then**
18.   **report SCC** $\mathcal{S}(v)$;
19.   DEAD := DEAD $\cup \mathcal{S}(v)$;
20.   R.pop();

$$R = \{H, E, A, B\}$$
$$VISITED = \{H, E, A, B\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1   Initialize an empty set DEAD;
2   Initialize an empty set VISITED;
3   Initialize an empty stack R;
4   setBased(v_0);
5   **function** *setBased*: $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;

11   **else if** $w \notin VISITED$ **then**
12   setBased(w);

13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
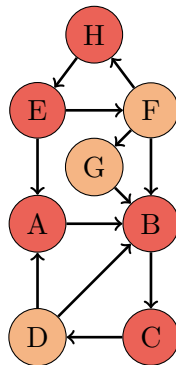15 r := R.pop();
16   UNITE($\mathcal{S}, r,$ R.top());

17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$$R = \{H, E, A, B, C\}$$
$$VISITED = \{H, E, A, B, C\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1   Initialize an empty set DEAD;
2   Initialize an empty set VISITED;
3   Initialize an empty stack R;
4   setBased(v_0);
5 **function** *setBased:* $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;
11   **else if** $w \notin VISITED$ **then**
12   setBased(w);
13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15   r := R.pop();
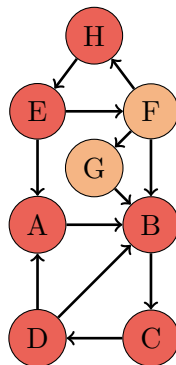16   UNITE($\mathcal{S}, r,$ R.top());
17   **if** $v =$ R.top() **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$$R = \{H, E, A, C, B, D\}$$
$$VISITED = \{H, E, A, B, C, D\}$$
$$DEAD = \{\}$$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 **function** *setBased:* $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;
11   **else if** $w \notin VISITED$ **then**
12   setBased(w);
13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15   r := R.pop();
16   UNITE($\mathcal{S}, r, $R.top());
17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$R = \{H, E\}$

$VISITED = \{H, E, A, B, C, D\}$

$DEAD = \{A, B, C, D\}$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;

1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 **function** *setBased*: $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;

11   **else if** $w \notin VISITED$ **then**
12   setBased(w);

13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
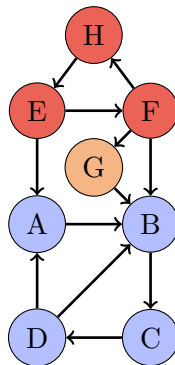15 $r := $ R.pop();
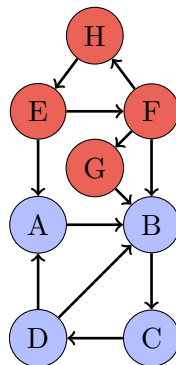16   UNITE($\mathcal{S}, r,$ R.top());

17   **if** $v = $ R.top() **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

```
Data: A graph 𝒢 = (𝒱, ℰ), a starting node v₀;
1  Initialize an empty set DEAD;
2  Initialize an empty set VISITED;
3  Initialize an empty stack R;
4  setBased(v₀);
5  function setBased: v ∈ 𝒱 → None
6      VISITED := VISITED ∪ {v};
7      R.push(v);
8      foreach w ∈ POST(v) do
9          if w ∈ DEAD then
10             continue;

11         else if w ∉ VISITED then
12             setBased(w);

13         else
14             while 𝒮(v) ≠ 𝒮(w) do
15                 r := R.pop();
16                 UNITE(𝒮, r, R.top());

17     if v = R.top() then
18         report SCC 𝒮(v);
19         DEAD := DEAD ∪ 𝒮(v);
20         R.pop();
```

$R = \{H, E, F, G\}$

$\text{VISITED} = \{H, E, A, B, C, D, F, G\}$

$\text{DEAD} = \{A, B, C, D\}$

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

Midterm presentation of the research course
└─ Correctness
  └─ Description of the algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 **function** setBased: $v \in \mathcal{V} \rightarrow None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;

11   **else if** $w \notin VISITED$ **then**
12   setBased(w);

13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15   r := R.pop();
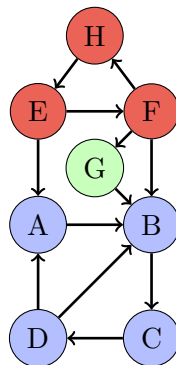16   UNITE($\mathcal{S}, r$, R.top());

17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$R = \{H, E, F\}$

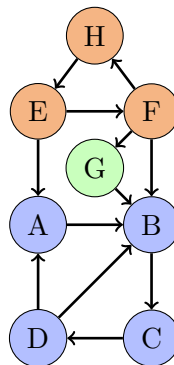$VISITED = \{H, E, A, B, C, D, F, G\}$

$DEAD = \{A, B, C, D, G\}$

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

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;
1 Initialize an empty set DEAD;
2 Initialize an empty set VISITED;
3 Initialize an empty stack R;
4 setBased(v_0);
5 **function** *setBased:* $v \in \mathcal{V} \to None$
6   VISITED := VISITED $\cup \{v\}$;
7   R.push(v);
8   **foreach** $w \in POST(v)$ **do**
9   **if** $w \in DEAD$ **then**
10   continue;

11   **else if** $w \notin VISITED$ **then**
12   setBased(w);

13   **else**
14   **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**
15   r := R.pop();
16   UNITE($\mathcal{S}, r,$ R.top());

17   **if** $v = R.top()$ **then**
18   **report SCC** $\mathcal{S}(v)$;
19   DEAD := DEAD $\cup \mathcal{S}(v)$;
20   R.pop();

$R = \{\}$

VISITED = {H, E, A, B, C, D, F, G}

DEAD = {A, B, C, D, G, E, F, H}

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

(Finite directed graphs)

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

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

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

(Finite directed graphs)

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

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

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

(SCC)

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

(Maximal SCC)

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

Proof process

(Well-formed environment)

```
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
```

```
definition pre_dfs where
  "pre_dfs v e ≡ wf_env e ∧ v ∉ visited e"
definition post_dfs where "post_dfs v e ≡ wf_env e"
```

```
definition pre_dfss where "pre_dfs v vs e ≡ wf_env e"
definition post_dfss where "post_dfs v vs e ≡ wf_env e"
```

```
lemma pre_dfs_pre_dfss:
  assumes "pre_dfs v e"
  shows "pre_dfss v (successors v) (e(|visited:=visited e ∪ {v},
stack:= v # stack e|))"
```

```
lemma pre_dfss_pre_dfs:
  fixes w
  assumes "pre_dfss v vs e" and "w ∉ visited e"
  shows "pre_dfs w e"
```

```
lemma pre_dfs_implies_post_dfs:...
```

```
lemma pre_dfss_implies_post_dfss:...
```

Possible prospects

- Finish the entire proof (with termination and functions domains)
- Parallel algorithm for computing SCC ? Proof ?