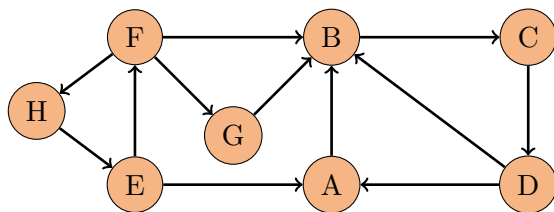


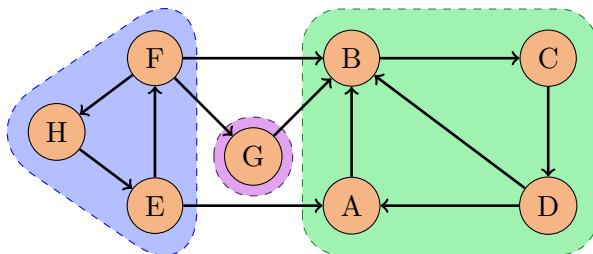
Formal verification in Isabelle / HOL of an algorithm for computing SCCs

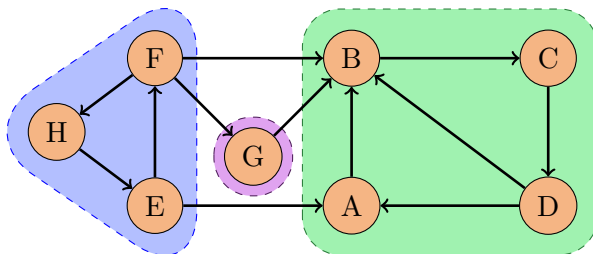
Vincent Trélat

École Nationale Supérieure des Mines de Nancy
Département Informatique

January 14, 2022



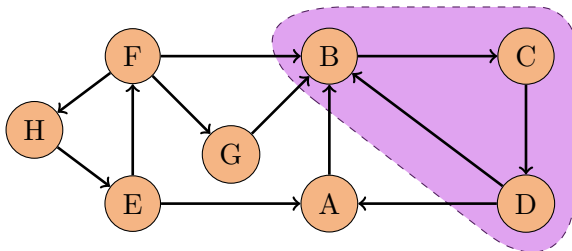




Definition 1

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:

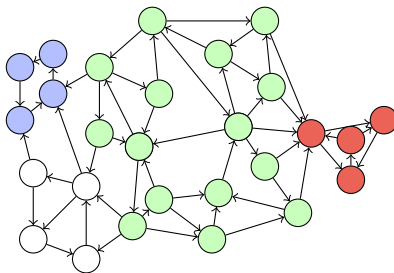
$$\forall x, y \in \mathcal{C}, (x \Rightarrow y) \wedge (y \Rightarrow x)$$



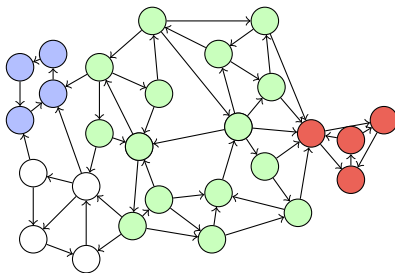
Definition 1

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  $\Rightarrow$  'a list  $\Rightarrow$  'a list where  
  concat Empty xs = xs  
| concat (Cons x xs) ys = Cons x (concat xs ys)
```

```
fun rev :: 'a list  $\Rightarrow$  'a list where  
  rev Empty = Empty  
| rev (Cons x xs) = concat (rev xs) (Cons x Empty)
```

(Theorem statement)

```
theorem rev_rev : rev (rev xs) = xs
```

(Theorem statement)

```
theorem rev_rev : rev (rev xs) = xs  
  apply (induction xs)  
  apply auto
```

(Theorem statement)

```
theorem rev_rev : rev (rev xs) = xs
  apply (induction xs)
  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: rev (concat xs ys) = concat(rev ys) (rev xs)
  apply (induction xs)
  apply auto
```

(Adding a first lemma)

```
lemma rev_concat: rev (concat xs ys) = concat (rev ys) (rev xs)
  apply (induction xs)
  apply auto
```

(Subgoals)

1. $\text{rev } \text{ys} = \text{concat } (\text{rev } \text{ys}) \text{ Empty}$
2. $\bigwedge x1 \text{ xs.}$
 $\text{rev } (\text{concat } \text{xs } \text{ys}) = \text{concat } (\text{rev } \text{ys}) (\text{rev } \text{xs}) \implies$
 $\text{rev } (\text{concat } (\text{Cons } x1 \text{ xs}) \text{ys}) = \text{concat } (\text{rev } \text{ys}) (\text{rev } (\text{Cons } x1 \text{ xs}))$

(Adding a second lemma)

```
lemma concat_empty:  concat xs Empty = xs
  apply (induction xs)
  apply auto
```

(Adding a third lemma: associative property)

```
lemma concat_assoc:  concat (concat xs ys) zs = concat xs
(concat ys zs)
  apply (induction xs)
  apply auto
```

```
theorem rev_rev:  rev (rev xs) = xs
  apply (induction xs)
  apply auto
```

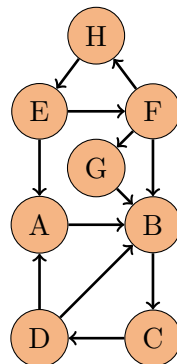
No subgoals!

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

```



$R = []$
 $\text{VISITED} = \{\}$
 $\text{DEAD} = \{\}$

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

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

Initialize an empty set DEAD;

Initialize an empty set VISITED;

Initialize an empty stack R;

setBased(v_0);

function setBased: $v \in \mathcal{V} \rightarrow \text{None}$

VISITED := VISITED $\cup \{v\}$;

R.push(v);

foreach $w \in \text{POST}(v)$ **do**

if $w \in \text{DEAD}$ **then**

 continue;

else if $w \notin \text{VISITED}$ **then**

 setBased(w);

else

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

$r := \text{R.pop}()$;

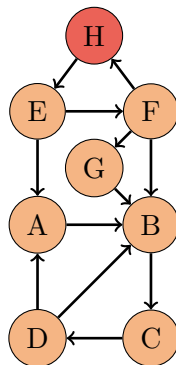
 UNITE($\mathcal{S}, r, \text{R.top}()$);

if $v = \text{R.top}()$ **then**

report SCC $\mathcal{S}(v)$;

 DEAD := DEAD $\cup \mathcal{S}(v)$;

 R.pop();



$R = [H]$

VISITED = {H}

DEAD = {}

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

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

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** $S(v) \neq S(w)$ **do**

15 $r := R.pop()$;

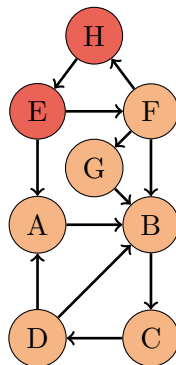
16 $UNITE(S, r, R.top())$;

17 **if** $v = R.top()$ **then**

18 **report SCC** $S(v)$;

19 $DEAD := DEAD \cup S(v)$;

20 $R.pop()$;



$R = [H, E]$

$VISITED = \{H, E\}$

$DEAD = \{\}$

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

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

Initialize an empty set DEAD;

Initialize an empty set VISITED;

Initialize an empty stack R;

setBased(v_0);

function setBased: $v \in \mathcal{V} \rightarrow \text{None}$

 VISITED := VISITED $\cup \{v\}$;

 R.push(v);

foreach $w \in \text{POST}(v)$ **do**

if $w \in \text{DEAD}$ **then**

 continue;

else if $w \notin \text{VISITED}$ **then**

 setBased(w);

else

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

$r := \text{R.pop}()$;

 UNITE($\mathcal{S}, r, \text{R.top}()$);

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

$r := \text{R.pop}()$;

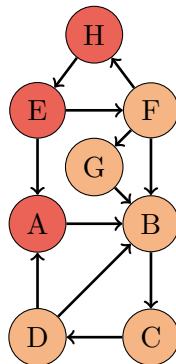
 UNITE($\mathcal{S}, r, \text{R.top}()$);

if $v = \text{R.top}()$ **then**

report SCC $\mathcal{S}(v)$;

 DEAD := DEAD $\cup \mathcal{S}(v)$;

 R.pop();



$R = [H, E, A]$

$\text{VISITED} = \{H, E, A\}$

$\text{DEAD} = \{\}$

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

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

Initialize an empty set DEAD;

Initialize an empty set VISITED;

Initialize an empty stack R;

setBased(v_0);

function setBased: $v \in \mathcal{V} \rightarrow \text{None}$

VISITED := VISITED $\cup \{v\}$;

R.push(v);

foreach $w \in \text{POST}(v)$ **do**

if $w \in \text{DEAD}$ **then**

 continue;

else if $w \notin \text{VISITED}$ **then**

 setBased(w);

else

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

$r := \text{R.pop}()$;

 UNITE($\mathcal{S}, r, \text{R.top}()$);

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

$r := \text{R.pop}()$;

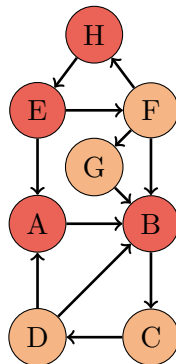
 UNITE($\mathcal{S}, r, \text{R.top}()$);

if $v = \text{R.top}()$ **then**

report SCC $\mathcal{S}(v)$;

 DEAD := DEAD $\cup \mathcal{S}(v)$;

 R.pop();



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

$\text{VISITED} = \{H, E, A, B\}$

$\text{DEAD} = \{\}$

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

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

Initialize an empty set DEAD;

2 Initialize an empty set VISITED;

3 Initialize an empty stack R;

4 $\text{setBased}(v_0)$;

5 **function** $\text{setBased}: v \in \mathcal{V} \rightarrow \text{None}$

6 VISITED := VISITED $\cup \{v\}$;

7 R.push(v);

8 **foreach** $w \in \text{POST}(v)$ **do**

9 **if** $w \in \text{DEAD}$ **then**

10 continue;

11 **else if** $w \notin \text{VISITED}$ **then**

12 $\text{setBased}(w)$;

13 **else**

14 **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

15 $r := \text{R.pop}()$;

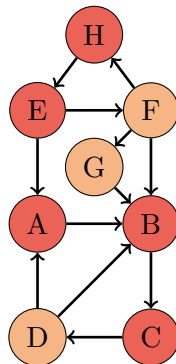
16 $\text{UNITE}(\mathcal{S}, r, \text{R.top}())$;

17 **if** $v = \text{R.top}()$ **then**

18 **report** SCC $\mathcal{S}(v)$;

19 $\text{DEAD} := \text{DEAD} \cup \mathcal{S}(v)$;

20 $\text{R.pop}()$;



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

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

$\text{DEAD} = \{\}$

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

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

Initialize an empty set DEAD;

2 Initialize an empty set VISITED;

3 Initialize an empty stack R;

4 $\text{setBased}(v_0)$;

5 **function** $\text{setBased}: v \in \mathcal{V} \rightarrow \text{None}$

6 VISITED := VISITED $\cup \{v\}$;

7 R.push(v);

8 **foreach** $w \in \text{POST}(v)$ **do**

9 **if** $w \in \text{DEAD}$ **then**

10 continue;

11 **else if** $w \notin \text{VISITED}$ **then**

12 $\text{setBased}(w)$;

13 **else**

14 **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

15 $r := \text{R.pop}()$;

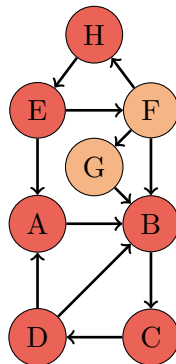
16 $\text{UNITE}(\mathcal{S}, r, \text{R.top}())$;

17 **if** $v = \text{R.top}()$ **then**

18 **report** SCC $\mathcal{S}(v)$;

19 $\text{DEAD} := \text{DEAD} \cup \mathcal{S}(v)$;

20 $\text{R.pop}()$;



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

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

$\text{DEAD} = \{\}$

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

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

Initialize an empty set DEAD;

Initialize an empty set VISITED;

Initialize an empty stack R;

setBased(v_0):

function setBased: $v \in \mathcal{V} \rightarrow \text{None}$

VISITED := VISITED $\cup \{v\}$;

R.push(v);

foreach $w \in \text{POST}(v)$ do

if $w \in \text{DEAD}$ then

continue;

else if $w \notin \text{VISITED}$ then

setBased(w);

else

while $\mathcal{S}(v) \neq \mathcal{S}(w)$ do

$r := \text{R.pop}()$;

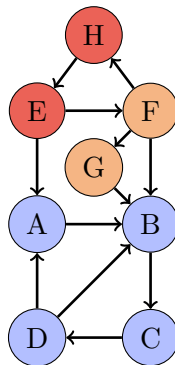
UNITE($\mathcal{S}, r, \text{R.top}()$);

if $v = \text{R.top}()$ then

report SCC $\mathcal{S}(v)$;

DEAD := DEAD $\cup \mathcal{S}(v)$;

R.pop();



$R = [H, E]$

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

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

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

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

Initialize an empty set DEAD;

2 Initialize an empty set VISITED;

3 Initialize an empty stack R;

4 $\text{setBased}(v_0)$;

5 **function** $\text{setBased}: v \in \mathcal{V} \rightarrow \text{None}$

6 VISITED := VISITED $\cup \{v\}$;

7 R.push(v);

8 **foreach** $w \in \text{POST}(v)$ **do**

9 **if** $w \in \text{DEAD}$ **then**

10 continue;

11 **else if** $w \notin \text{VISITED}$ **then**

12 $\text{setBased}(w)$;

13 **else**

14 **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

15 $r := \text{R.pop}()$;

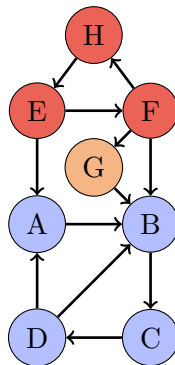
16 $\text{UNITE}(\mathcal{S}, r, \text{R.top}())$;

17 **if** $v = \text{R.top}()$ **then**

18 **report** SCC $\mathcal{S}(v)$;

19 DEAD := DEAD $\cup \mathcal{S}(v)$;

20 R.pop();



$R = [H, E, F]$

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

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

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

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

Initialize an empty set DEAD;

2 Initialize an empty set VISITED;

3 Initialize an empty stack R;

4 $\text{setBased}(v_0)$;

5 **function** $\text{setBased}: v \in \mathcal{V} \rightarrow \text{None}$

6 VISITED := VISITED $\cup \{v\}$;

7 R.push(v);

8 **foreach** $w \in \text{POST}(v)$ **do**

9 **if** $w \in \text{DEAD}$ **then**

10 continue;

11 **else if** $w \notin \text{VISITED}$ **then**

12 $\text{setBased}(w)$;

13 **else**

14 **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

15 $r := \text{R.pop}()$;

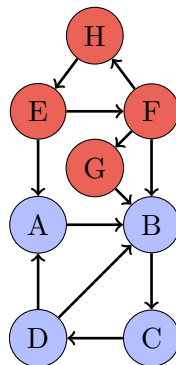
16 $\text{UNITE}(\mathcal{S}, r, \text{R.top}())$;

17 **if** $v = \text{R.top}()$ **then**

18 **report SCC** $\mathcal{S}(v)$;

19 DEAD := DEAD $\cup \mathcal{S}(v)$;

20 R.pop();



$R = [H, E, F, G]$

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

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

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

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

Initialize an empty set DEAD;

2 Initialize an empty set VISITED;

3 Initialize an empty stack R;

4 $\text{setBased}(v_0)$;

5 **function** $\text{setBased}: v \in \mathcal{V} \rightarrow \text{None}$

6 VISITED := VISITED $\cup \{v\}$;

7 R.push(v);

8 **foreach** $w \in \text{POST}(v)$ **do**

9 **if** $w \in \text{DEAD}$ **then**

10 continue;

11 **else if** $w \notin \text{VISITED}$ **then**

12 $\text{setBased}(w)$;

13 **else**

14 **while** $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do**

15 $r := \text{R.pop}()$;

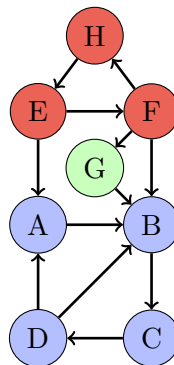
16 $\text{UNITE}(\mathcal{S}, r, \text{R.top}())$;

17 **if** $v = \text{R.top}()$ **then**

18 **report SCC** $\mathcal{S}(v)$;

19 $\text{DEAD} := \text{DEAD} \cup \mathcal{S}(v)$;

20 $\text{R.pop}()$;



$R = [H, E, F]$

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

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

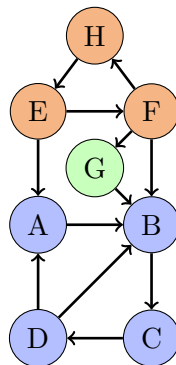
$\mathcal{S} = \{A, B, C, D\} \cup \{E\} \cup \{F\} \cup \{G\} \cup \{H\}$

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

```



$R = []$

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

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

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

(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  $\Rightarrow$  'v set"  
  assumes vfin: "finite vertices"  
  and sclosed: " $\forall x \in \text{vertices}. \text{successors } x \subseteq \text{vertices}$ "
```

```
abbreviation edge where  
"edge x y  $\equiv$  y  $\in$  successors x"
```

```
inductive reachable where  
  reachable_refl[iff]: "reachable x x"  
| reachable_succ[elim]:  
  "[edge x y; reachable y z]  $\implies$  reachable x z"
```

(SCC)

```
definition is_subscs where
```

```
"is_subscs S  $\equiv$   $\forall x \in S. \forall y \in S. \text{reachable } x y$ "
```

(Maximal SCC)

```
definition is_scc where
```

```
"is_scc S  $\equiv$  S  $\neq$  {}
```

```
 $\wedge$  is_subscs S
```

```
 $\wedge$  ( $\forall S'. S \subseteq S' \wedge \text{is\_subscs } S' \longrightarrow S' = S$ )"
```

Proof process

(Well-formed environment)

definition wf_env **where**

"wf_env e \equiv "

distinct (stack e)

\wedge set (stack e) \subseteq visited e

\wedge explored e \subseteq visited e

\wedge explored e \cap set (stack e) = $\{\}$

\wedge ($\forall v\ w. w \in \mathcal{S}\ e\ v \longleftrightarrow \mathcal{S}\ e\ v = \mathcal{S}\ e\ w$)

\wedge ($\forall v \in \text{set}(\text{stack } e). \forall w \in \text{set}(\text{stack } e). v \neq w \longrightarrow$
 $\mathcal{S}\ e\ v \cap \mathcal{S}\ e\ w = \{\}$)

\wedge ($\forall v. v \notin \text{visited } e \longrightarrow \mathcal{S}\ e\ v = \{v\}$)

$\wedge \bigcup \{\mathcal{S}\ e\ v \mid v. v \in \text{set}(\text{stack } e)\} = \text{visited } e - \text{explored } e$

```
definition pre_dfs where
  "pre_dfs v e  $\equiv$  wf_env e  $\wedge$  v  $\notin$  visited e"
definition post_dfs where "post_dfs v e  $\equiv$  wf_env e"
```

```
definition pre_dfss where "pre_dfs v vs e  $\equiv$  wf_env e"
definition post_dfss where "post_dfs v vs e  $\equiv$  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:  
  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
 - The algorithm is already written¹
 - A consistent work on the structure of the parallel workers is already in progress

¹V. Bloemen, 2019. *Strong Connectivity and Shortest Paths for Checking Models* 