# Formal methods in higher-order logic: application to strongly connected components algorithms

Vincent Trélat

École Nationale Supérieure des Mines de Nancy
Département Informatique

January 14, 2022

# Outline

1. Introduction
   - Motivation
   - Isabelle/HOL
   - Proof process example
2. SCC algorithms correctness
   - Definitions

# Motivation

- Networks: connection and data sharing
- Model checking: counter-examples finding
- Graph theory: structure analysis and reduction

# 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

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

# Isabelle/HOL

Example

Simple proofs on a basic data structure:

- Definitions
- Functions
- Theorems
- Proofs

## (Type definition)

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

- Recursive structure
- Generic and static type
- Implicit constructor 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 reverse ::  "'a list ⇒ 'a list" where
  "reverse Empty = Empty"
| "reverse (Cons x xs) = concat (reverse xs) (Cons x
Empty)"
```

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

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

**(Theorem writing)**

```
theorem reverse_reverse [simp]:  "reverse (reverse x) = x"
apply (induction x)
apply auto
```

**(Subgoal)**

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

## (Theorem writing)

```
theorem reverse_reverse [simp]:  "reverse (reverse x) = x"
 apply (induction x)
 apply auto
```

## (Subgoal)

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

### (Theorem writing)

```
theorem reverse_reverse [simp]:  "reverse (reverse x) = x"
 apply (induction x)
 apply auto
```

### (Subgoal)

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

## (Adding a first lemma)

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

## (Subgoals)

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

## (Adding a first lemma)

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

## (Subgoals)

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

## (Adding a second lemma)

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

No subgoals!

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

## (Subgoal)

```
1.  ⋀ x1 xs.
   reverse (concat xs ys) = concat (reverse ys) (reverse xs) ⟹
   reverse (concat (Cons x1 xs) ys) = concat (reverse ys) (reverse (Cons
x1 xs))
```

## (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
```

No subgoals!

```
theorem reverse_reverse [simp]:  "reverse (reverse x) = x"
  apply (induction x)
  apply auto
```

No subgoals!

SCC algorithms

## Definition 1

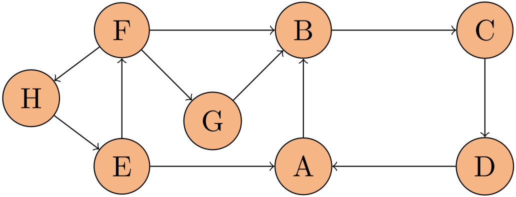
Let $\mathcal{G} := (\mathcal{V}, \mathcal{E})$ be a directed graph and $\mathcal{C} \subseteq \mathcal{V}$. $\mathcal{C}$ is a SCC of $\mathcal{G}$ if:

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

## 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:

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



## 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:

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