The Coq Proof Assistant
Semantics and applications to verification

Xavier Rival

École Normale Supérieure
What is a proof assistant?

A tool to **formalize** and **verify** proofs

The key word is **assistant**: it assists the user in

- defining the proof goals formally;
- setting up the structure of the proofs;
- making the proof steps;
- checking the overall consistency of the proof, at the end.

Some steps are more assisted than others:

- formalization is done with respect to the knowledge of the user, it is **error prone**
- key structural arguments (induction hypotheses and such) are very hard to get right in general
- checking a series of proof steps is easier to mechanize...
Purpose of Coq and principle

Coq is a programming language

- We can **define data-types** and **write programs** in Coq
- Language similar to a **pure functional language**
- **Very expressive** type system (more on this later)

- Programs can be ran inside Coq
- Programming language of the year ACM Award in 2014...

Coq is a proof assistant

- It allows to **express theorems** and **proofs**
- It can **verify** a proof
- It can also **infer some proofs** or **proof steps**

- Proof search is usually mostly manual and takes most of the time
Main proof assistants

**Coq:** the topic of this lecture

**Isabelle / HOL:** a higher order logic framework
- syntax is closer to the logics
- proof term underneath...

**ACL2:** A Computational Logic for Applicative Common Lisp
- a framework for automated reasoning
- based on functional common lisp

**PVS:** Prototype Verification System
- kernel extends Church types
- less emphasis on the notion of proof term, more emphasis on automation
Overall workflow

1. **Define the objects** properties need be proved about
   Data-structures, base types, programs written in the Coq (or vernacular) language

2. Write and prove **intermediate lemmas**
   ▶ a theorem is defined by a formula in the Coq language.
   ▶ a proof requires a sequence of **tactics applications**
     tactics are described as part of a separate language.
   ▶ at the end of the proof, a **proof term** is constructed and verified.

3. Write and prove the **main theorems**

4. If needed, **extract** programs

**Two languages:** one for **definitions/theorems/proofs**, one for **tactics**
In Coq, everything is a term

- **The core of Coq** is defined by a language of **terms**
- **Commands** are used in order to manipulate terms

Examples of terms:

- **base values:** 0, 1, true...
- **types:** nat, bool, but also Prop...
- **functions:** fun (n: nat) => n + 1
- **function applications:** (fun (n: nat) => n + 1) 8
- **logical formulas:**
  
  exists p: nat, 8 = 2 * p,
  forall a b: Prop, a /\ b -> a

- **complex functions** (more on this one later):
  
  fun (a b : Prop) (H : a /\ b) =>
  and_ind (fun (H0 : a) (_, b) => H0) H
In Coq, every term has a type

- As observed, **types are terms**
- Every term also **has a type**, denoted by `term : type`

- `0 : nat`
- `nat : Set`
- `Set : Type`
- `Type : Type (caveat: not quite the same instance)`

```
(fun (n : nat) => n + 1) : nat -> nat
```

- More complex types get interesting:

```
fun (a b : Prop) (H : a /\ b) =>
  and_ind (fun (H0 : a) (_ : b) => H0) H
: forall a b : Prop, a /\ b -> a
```
The core principle of Coq

- A proof of $P$ can be viewed a term of type $P$
- A proof of $P \implies Q$ can be viewed a function transforming a proof of $P$ into a proof of $Q$, hence, a function of type $P \to Q$...

Similarity between typing rules and proof rules:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : P \vdash u : Q$</td>
<td>$\Gamma \vdash \lambda x \cdot u : P \to Q$</td>
<td>fun</td>
</tr>
<tr>
<td>$\Gamma \vdash u : P \to Q$</td>
<td>$\Gamma \vdash v : P$</td>
<td>app</td>
</tr>
<tr>
<td>$\Gamma \vdash u , v : Q$</td>
<td>$\Gamma, P \vdash Q$</td>
<td>mp</td>
</tr>
<tr>
<td>$\Gamma \vdash Q$</td>
<td>$\Gamma \vdash P \implies Q$</td>
<td>implic</td>
</tr>
</tbody>
</table>

Correspondance:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>proof</td>
</tr>
<tr>
<td>type</td>
<td>theorem</td>
</tr>
</tbody>
</table>

Search a proof of $P$ 
$\equiv$ search $u$ of type $P$
Defining a term

Two ways:

1. **Define it fully**, with its type and its definition

   Definition zero: nat := 0.

2. **Provide only its type** and search for a proof of it

   Lemma lzero: nat.
   exact 0.
   Save.
   Definition lincr: forall n: nat, nat.
   intro. exact (n + 1).
   Save.

- **Definition**: Definition name u: t := def.
- **Proof**: Definition name u: t. or Lemma name u: t.
Inductive definition

- A very powerful mechanism
- In Coq, almost everything is actually an inductive definition
  ... examples: integers, booleans, equality, conjunction...

**Syntax:**

Inductive tree : Set :=
  | leaf : tree
  | node : tree -> tree -> tree.

**Induction principles** automatically provided by Coq, and to use in induction proofs:

tree_ind : forall P : tree -> Prop,
  P leaf
  -> (forall t : tree, P t -> forall t0 : tree, P t0
      -> P (node t t0))
  -> forall t : tree, P t
Recursive functions

- Very natural to work with inductive definitions
- **Caveat:** must provably terminate
  - this is usually checked with a **strict sub-term condition**

**Syntax:**

```coq
Fixpoint size (t: tree) : nat :=
  match t with
  | leaf => 0
  | node t0 t1 => 1 + (size t0) + (size t1)
end.
```

**Ill formed definition, rejected by the system (termination issue):**

```coq
Fixpoint f (t: tree): nat :=
  match t with
  | leaf | node leaf leaf => 0
  | node _ _ => f (node leaf leaf)
end.
```
Proving a term

View in proof mode:

\[ \begin{align*}
    a & : \text{Prop} \\
    b & : \text{Prop} \\
    H : a \land b \\
    H0 : a \\
    H1 : b \\
\end{align*} \]

\[ \begin{align*}
    \text{above the bar: current assumptions} \\
    \text{below the bar: current subgoal} \\
    \text{(there may be several goals)} \\
    \text{at the end: displays} \\
    \text{No more subgoals.} \\
    \text{command Save. stores the term.} \\
\end{align*} \]

Progression towards a finished proof:

- based on commands called tactics
- in the background, Coq constructs the proof term
A few tactics, and their effect

- Each tactic performs a basic operation on the current goal
- In the background, Coq constructs the proof term
- At the end, the term is independantly checked (very reliable !)

- **Introduction of an assumption** (proof tree and term):
  \[
  \Gamma, P \vdash Q \\
  \frac{\Gamma \vdash P \Rightarrow Q}{\Gamma \vdash P = \Rightarrow Q}
  \quad \\
  \frac{\Gamma, x : P \vdash u : Q}{\Gamma \vdash \lambda x \cdot u : P \rightarrow Q}
  \]

- **Application of an implication**:
  \[
  \Gamma \vdash P \Rightarrow Q \\
  \frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}
  \quad \\
  \frac{\Gamma \vdash u : P \rightarrow Q \quad \Gamma \vdash v : P}{\Gamma \vdash u \cdot v : Q}
  \]

- **Immediate conclusion of a subgoal**:
  \[
  \Gamma, P \vdash P \\
  \frac{\Gamma, x : P \vdash x : P}{\Gamma, x : P \vdash x : P}
  \]
So far, we have considered fairly manual tactics...

There are also **automated tactics**, that typically call an external program to try to solve a goal, and then constructs a proof term:

- either verify the proof term afterwards...
- ... or call a function proved once and for all to build it

**Tauto**: decides propositional logic

**Omega**: solves a class of numeric (in)-equalities (see manual)
A glimpse at the tactic language

**Most common tactics:**

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro.</td>
<td>Introduce one assumption</td>
</tr>
<tr>
<td>intros.</td>
<td>Introduce as many assumptions as possible</td>
</tr>
<tr>
<td>apply H.</td>
<td>Applies assumption $H$ (should be of the form $A \rightarrow B$)</td>
</tr>
<tr>
<td>elim H.</td>
<td>Decomposes assumption $H$</td>
</tr>
<tr>
<td>exact t.</td>
<td>Provides a proof term for current sub-goal</td>
</tr>
<tr>
<td>trivial.</td>
<td>Conclude immediately very simple proofs.</td>
</tr>
<tr>
<td>induction t.</td>
<td>Perform induction proof over term $t$</td>
</tr>
<tr>
<td>rewrite H.</td>
<td>Rewrite assumption $H$ (should be of the form $t_0 = t_1$)</td>
</tr>
<tr>
<td>tauto.</td>
<td>Decision procedure in propositional logic</td>
</tr>
</tbody>
</table>

Do not hesitate to look at the online manual!
A glimpse at the command language

**Most common tactics** (should be enough for a TD):

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check ( t. )</td>
<td>Prints the type of term ( t )</td>
</tr>
<tr>
<td>Print ( t. )</td>
<td>Prints the type and definition of term ( t )</td>
</tr>
<tr>
<td>Definition ( u: t := \text{[term]} ).</td>
<td>Full definition of term ( u )</td>
</tr>
<tr>
<td>Lemma ( t. )</td>
<td>Start a proof of term ( t )</td>
</tr>
<tr>
<td>Theorem ( t. )</td>
<td></td>
</tr>
<tr>
<td>Definition ( t. )</td>
<td></td>
</tr>
<tr>
<td>Save.</td>
<td>Exit proof mode and save proof term</td>
</tr>
</tbody>
</table>