The Rocq Proof Assistant

Semantics and applications to verification

Xavier Rival

École Normale Supérieure

March 6th, 2026

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 Rocq and principle

Rocq is a programming language

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

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

Rocq 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

Rocq: 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 Rocq (or vernacular) language

2. Write and prove **intermediate lemmas**
   - a theorem is defined by a formula in the Rocq 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 Rocq, everything is a term

- The **core of Rocq** 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 Rocq, 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
```

Curry-Howard correspondence

The core principle of Rocq

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

Similarity between **typing** rules and **proof** rules:

$$\frac{\Gamma, x : P \vdash u : Q}{\Gamma \vdash \lambda x \cdot u : P \longrightarrow Q}\ \textit{fun} \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Longrightarrow Q}\ \textit{implic}$$

$$\frac{\Gamma \vdash u : P \longrightarrow Q \quad \Gamma \vdash v : P}{\Gamma \vdash u\ v : Q}\ \textit{app} \qquad \frac{\Gamma \vdash P \Longrightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\ \textit{mp}$$

Correspondance:

program	proof
type	theorem

Searching a proof of P
$\equiv$ searching u of type P

Defining a term

Two ways:

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

```
Definition zero: nat := 0.
Definition incr (n: nat): nat := n + 1.
```

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 Rocq, **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 Rocq, 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:**

```
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):**

```
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:

```
a : Prop
b : Prop
H : a /\ b
H0 : a
H1 : b
============================
 a
```

- above the bar: **current assumptions**
- below the bar: **current subgoal** (there may be several goals)
- **at the end:** displays `No more subgoals.`
- command `Save.` stores the term.

Progression towards a finished proof:

- based on commands called **tactics**
- in the background, Rocq **constructs the proof term**

A few tactics, and their effect

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

- **Introduction of an assumption** (proof tree and term):

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Longrightarrow Q} \qquad \frac{\Gamma, x : P \vdash u : Q}{\Gamma \vdash \lambda x \cdot u : P \longrightarrow Q}$$

- **Application of an implication:**

$$\frac{\Gamma \vdash P \Longrightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \qquad \frac{\Gamma \vdash u : P \longrightarrow Q \quad \Gamma \vdash v : P}{\Gamma \vdash u\ v : Q}$$

- **Immediate conclusion of a subgoal:**

$$\frac{}{\Gamma, P \vdash P} \qquad \frac{}{\Gamma, x : P \vdash x : P}$$

Automation in Rocq

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

Examples:

- **Tauto:** decides propositional logic
- **Omega:** solves a class of numeric (in)-equalities (see manual)

Language of tactics:
more advanced users can combine tactics to build their own

Proof by reflection: prove decision procedures, and invoke them...

Summary of a few useful tactics (1/3)

Tactics for basic connectors manipulations:

Tactic	Effect
`intro.`	Introduce one assumption
`intros.`	Introduce as many assumptions as possible
`apply H.`	Applies assumption `H` (should be of the form `A->B`)
`trivial.`	Conclude immediately very simple proofs
`exact t.`	Provides a proof term for current sub-goal
`cut H.`	Add `H` as a proof step.
`exfalso.`	Replace the current goal with `False`
`split.`	Decompose the proof of a conjunction into two proofs
`left.`	Select the left disjunct of a disjunctive goal
`right.`	Select the right disjunct of a disjunctive goal
`tauto.`	Decision procedure in propositional logic

Summary of a few useful tactics (2/3)

Tactics for equality and computation:

Tactic	Effect
`reflexivity.`	Prove a goal of the form `e = e`
`simpl.`	Compute as much as possible in the current goal
`simpl in H.`	Compute as much as possible in assumption `H`.
`f_equal.`	Turn a proof of `C e0=C e1` into one of `e0=e1` where `C` is an inductive constructor
`rewrite H.`	Rewrite assumption `H` (should be of the form `t0=t1`)
`rewrite H0 in H1.`	Rewrite assumption `H0` (should be of the form `t0=t1`) into `H1`

Summary of a few useful tactics (3/3)

Tactics for induction and for desctructuring inductive assumptions:

Tactic	Effect
`induction t.`	Perform induction proof over term `t` (which must be of inductive type)
`elim H.`	Decomposes assumption `H`

Do not hesitate to look at the online manual !

A few useful commands (for the toplevel)

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

Command	Meaning
`Check t.`	Prints the type of term `t`
`Print t.`	Prints the type and definition of term `t`
`Definition u: t := [term].`	Full definition of term `u`

`Lemma t.`
`Theorem t.`
`Definition t.`
	Start a proof of term `t`

`Qed.`	Exit proof mode and save proof term
`Require Import Arith.`	Start a proof of term `Arith` (other useful module: `List`)
`Search text.`	Search `text` in library lemmas