An Introduction to Rocq and the Hydra Battle

A Journey into Interactive Theorem Proving and Termination Proofs

John Alejandro González-González[1]
Andrés Sicard-Ramírez[2]

[1] *Mathematical Engineering, Universidad EAFIT, jagonzale4@eafit.edu.co*

[2] *School of Applied Sciences and Engineering, Universidad EAFIT, asr@eafit.edu.co*

May, 2025

Outline

What is Rocq?

Rocq (previously known as Coq) is an interactive theorem prover based on constructive type theory. It embodies the **Curry–Howard correspondence**, where:

- *Types* correspond to logical propositions.
- *Terms* (i.e., programs) correspond to proofs.

As a consequence:

- *Type checking* serves as proof verification.

Rocq supports defining functions, stating theorems, and interactively constructing proofs using tactics.

Sets, Types, and Propositions

In Rocq's type theory (inspired by Martin-Löf), we distinguish between three main categories:

- `Type` A general notion used to classify all meaningful constructs.
- `Set` A type for computational data, like `nat`, `list`, or `tree`. Used for defining programs and data structures.
- `Prop` A type for logical propositions, where inhabitants are proofs rather than data.

Both `Set` and `Prop` are `Type`, but they serve different purposes and are not comparable.

Boolean Logic

Booleans can be defined as non-recursive type. Logic operators can be encoded functionally:

```
Inductive bool : Set := true | false.
```

The type `bool` represents the set {true, false}, which models the propositions {*True*, *False*} in classical logic. We can then define logical operations like `not` as functions:

```
  Definition negate (b : bool) : bool :=
  match b with
  | true  => false
  | false => true
  end.
```

Coq Notation for Readability

Coq allows you to introduce custom notation to make proofs clearer and closer to mathematical style. These notations are defined with the `Notation` command and help avoid repeatedly writing constructors.

Example definitions:

```
Notation "0" := O.
Notation "1" := (S O).
Notation "2" := (S (S O)).
Notation "3" := (S (S (S O))).

Notation "x + y" := (plus x y)
  (at level 50, left associativity).
```

This feature is part of Coq's syntax extensions. For convenience, the standard library (in `Coq.Init.Nat`) already provides notations up to arbitrarily large numerals (‘0‘, ‘1‘, ‘2‘, . . .) and the familiar infix ”+” for addition.

Peano Natural Numbers

Numbers in Rocq can be defined from scratch:

```
Inductive nat : Set :=
  | O
  | S (n : nat).

Fixpoint plus (m n : nat) : nat :=
  match m with
  | O     => n
  | S m'  => S (plus m' n)
  end.
```

Peano numbers:

$$\mathbb{N} = \{0, S(0), S(S(0)), \ldots\}$$

where $S(n)$ is the successor function. *The recursive plus function models addition.*

Proof by Simplification

Some equalities can be shown directly by simplification:

```
Example plus_1_2 : plus 1 2 = 3.
Proof. simpl. reflexivity. Qed.
```

Simple use of Rocq's simplification engine.

Classical View:
In Peano Arithmetic:

$$S(O) + S(S(O)) = S(S(S(O))) \quad \Rightarrow \quad 1 + 2 = 3$$

This is a computation using the recursive definition of addition:

$$\begin{aligned} 1 + 2 &= S(0) + S(S(0)) \\ &= S(S(S(0))) = 3 \end{aligned}$$

Verified by unfolding the definition of $+$ on natural numbers.

Proof by Induction

Classical View:
Let $+$ be defined recursively:

$$\begin{cases} 0 + n = n & \text{(base case)} \\ S(m') + n = S(m' + n) & \text{(recursive case)} \end{cases}$$

We want to prove by induction:

$$\forall n \in \mathbb{N},\ n + 0 = n$$

Base case: $0 + 0 = 0$
Inductive step: Assume $n + 0 = n$, then:

$$S(n') + 0 = S(n' + 0) = S(n')$$

Therefore, the property holds for all $n \in \mathbb{N}$.

Proof by Induction

For universally quantified properties, use structural induction:

```
Theorem plus_n_O : forall n : nat, n + 0 = n.
Proof.
  induction n as [| n' IH].
  - reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.
```

Key pattern: base case + inductive hypothesis.

Using Lemmas

Given previously proven lemmas:

```
Lemma plus_n_O  : forall n   : nat, n = n + 0.
Lemma plus_n_Sm : forall m n : nat, S (m + n) = m + S n.
```

We can simplify proofs:

```
Theorem plus_comm : forall m n : nat, m + n = n + m.
Proof.
  intros m n.
  induction m as [| m' IH].
  - simpl. rewrite <- plus_n_O. reflexivity.
  - simpl. rewrite IH.
    rewrite <- plus_n_Sm. reflexivity.
Qed.
```

plus_n_O: $\forall n \in \mathbb{N}, \quad n + 0 = n$

plus_n_Sm: $\forall n, m \in \mathbb{N}, \quad S(n + m) = n + S(m)$

plus_comm: $\forall n, m \in \mathbb{N}, \quad n + m = m + n$

Lists in Rocq

Rocq provides a built-in list type, defined inductively:

```
Inductive list (A : Type) : Type :=
  | nil  : list A
  | cons : A -> list A -> list A.
```

We can then define functions over lists, for example, concatenation:

```
Fixpoint app {A : Type} (l1 l2 : list A) : list A :=
  match l1 with
  | nil      => l2
  | cons x t => cons x (app t l2)
  end.
```

Notation for convenience:

```
Notation "x :: l"   := (cons x l)      (at level 60, right associativity).
Notation "[]"       := nil.
Notation "l1 ++ l2" := (app l1 l2)    (at level 60, right associativity).
```

Now you can write expressions like: `1 :: 2 :: [] ++ [3; 4]`.

Rose Trees and Height in Rocq

Define a rose (multi-way) tree type:

```
Inductive RoseTree : Type :=
  | Node : nat -> list RoseTree -> RoseTree.
```

Measure its *height* (max depth of nodes):

```
Fixpoint height (t : RoseTree) : nat :=
  match t with
  | Node _ children =>
      1 + fold_right max 0 (map height children)
  end.
```

Rose Tree Example with Height

Consider this rose tree of height 3:

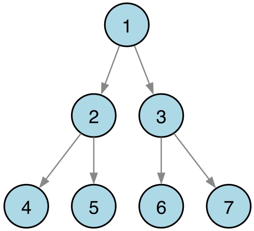

Figure: A binary tree with 7 nodes and height 3.

Its Coq representation as a rose tree:

```
Definition example_rose : RoseTree :=
  Node 1 [
    Node 2 [Node 4 []; Node 5 []];
    Node 3 [Node 6 []; Node 7 []]
  ].
Compute height example_rose. (* = 3 *)
```

Well-Foundedness and Accessibility

To prove termination in Rocq, we use the concept of **well-founded relations**.

Definition (Well-founded): A relation R on a set A is **well-founded** if there is no infinite descending chain:

$$a_0 \succ a_1 \succ a_2 \succ \cdots$$

where each $a_{i+1} \; R \; a_i$.

Definition (Accessibility):
An element $a \in A$ is *accessible* with respect to R if every R-smaller element of a is also accessible.

In Rocq:

- `Acc R a` means a is accessible under relation R.
- A relation is well-founded if all elements are accessible: `well_founded R := forall a, Acc R a`.

Example: < on Natural Numbers is Well-Founded

The usual less-than relation on `nat` is well-founded:

Fact: `lt` (i.e., $<$) on $\mathbb{N}$ is well-founded.

```
Require Import Coq.Arith.Wf_nat.

Check lt_wf.
(* lt_wf : well_founded lt *)
```

What does this mean?

- `lt_wf` proves: for every n, there are no infinite descending chains:

$$n > n_1 > n_2 > \cdots$$

- Therefore, every n is accessible with respect to $<$:

```
Acc lt n
```

- Enables defining recursive functions that decrease on n

The Hydra Battle

The Hydra battle is a famous history from Greek mythology, where Hercules faces the learnean Hydra, a serpent-like creature with multiple heads. Each time a head is cut off, two more grow back in its place.

Figure: Hercules and the Hydra of Lerna (1876). Oil on canvas, 179.3 x 154 cm. Art Institute of Chicago. Gustave Moreau (1826-1898).

Looking Ahead

To Explore:

1. **Modeling the Hydra Battle:** Formalizing the cutting–growing rules as inductive definitions and transition relations in Rocq.
2. **Exploring Variant Dynamics:** Analyzing how changing growth factors, cut rules, or tree structures affects termination behavior.
3. **Establishing Termination Proofs:** Constructing well-founded measures (lexicographic, ordinals, etc.) that decrease with every step.

Further Reading / Resources

- The Rocq (formerly Coq) development team, "Rocq Prover," `https://rocq-prover.org/`
- B. Pierce *et al.*, "Basics," in *Software Foundations*, `https://softwarefoundations.cis.upenn.edu/lf-current/Basics.html`
- A. Chlipala, "Universes," in *Certified Programming with Dependent Types*, `http://adam.chlipala.net/cpdt/html/Universes.html`
- YouTube: "The Hydra vs. Hercules – Numberphile," `https://www.youtube.com/watch?v=prURA1i8Qj4`
- P. Casteran, "Hydras&Co," `https://rocq-community.org/hydra-battles/doc/hydras.pdf`
- L. Kirby and J. Paris, "Accessible independence results for Peano arithmetic," *Bull. London Math. Soc.*, vol. 14, pp. 285–293, 1982.