Intorduction to the Rocq Programming Language

Learn to Code, Week 7 HT25

Tadayoshi Kamegai

Oxford Compsoc

March 8, 2025

Motivation

We want to generate proofs for programs, rather than test them.
$\rightarrow$ Provide a gurantee that it works on any input

What kind of things do we want to be able to verify?

- Write code with gurantees
  $\rightarrow$ (e.g., Prove correctness of sorting algorithms)
- Prove mathematical theorems
  $\rightarrow$ (e.g., Four Colour Theorem)
- Verify software and hardware
  $\rightarrow$ (e.g., A verified C compiler)

All of these can / have been done in Rocq!

Quick History

1969 : Howard, William A. "The formulae-as-types notion of construction"

- Curry-Howard Correspondence : Correspondence between proof systems and models of computation
- (simply) propositions are types, and proofs are terms of those types

1989 : Coq's Initial Release

- An interactive theorem prover implemented using OCaml.
  - Proof is built interactively by the user
  - Proof is automatically checked by the type system (CIC)
- Can trust proof is correct if one trusts the Coq Kernel

2005 : Georges Gonthier formalizes the four colour theorem in Coq.
2023 : Coq renames itself to "The Rocq Prover"

Getting a Rocq Environment

Find a suitable download for your device from https://coq.inria.fr/download

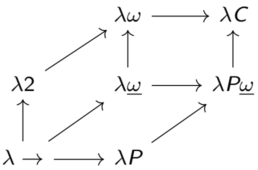
CoqIDE is the legacy IDE for Rocq. VSCode extensions are also avaialble.

Goal of Today's Talk

- Give a brief outline of the language syntax
- Outline how to write inductive proofs
- Understand the importance of design choices

What we won't cover (but definitely worth learning)

- Proof automation
- Type Theory and Dependent Types
- General theory behind the language



Computation in Rocq

Example

- We can explicitly compute computable functions by using the "Compute" function.

```
Compute (2 + 3).                 = 5
                                 : nat

Compute (10 - 3).                = 7
                                 : nat

Compute (andb true false).       = false
                                 : bool
```

Types and Expressions

Every expression in Rocq has a **type**. The type system prevents runtime errors.

Example

- `nat` : The natural numbers
- `bool` : The booleans
- `Prop` : Propositions (for proofs)
- `Set` : Computable data

Coq has a builtin "Check" function which returns the type.

Example

`Check 5.`	`5` `: nat`
`Check (negb false).`	`negb false` `: bool`

If types do not match, Rocq will return an error.

Example

```
Check (true + 3).

The term "true" has type "bool" while it is expected to have type
   "nat".
```

What would the types for the following be?

Rocq Code

```
Check (3 = 3).
Check (forall x : nat, x + 0 = x).
```

Key Point : Equality **is not** a boolean, it is a Proposition.

Functional Programming in Rocq - Functions (1)

Main Idea : We can define functions using `Definition` and `Fixpoint`.

Rocq Code

```
  Definition name (parameter) : return_type := expression.

  Definition double (n : nat) : nat := n * 2.

Compute double 4.
  = 8
  : nat
```

Functional Programming in Rocq - Functions (2)

Main Idea : We can define functions using `Definition` and `Fixpoint`.

Rocq Code

```
Fixpoint name (parameter) : return_type := expression.

Fixpoint factorial (n : nat) : nat :=
  match n with
    | 0    ⇒ 1
    | S n' ⇒ n * factorial n'
  end.

Compute factorial 5.
  = 120
  : nat
```

Fixpoints need to terminate to be well-defined. There needs to be some explicit decreasing argument, which in this case is `n`.

Functional Programming in Rocq - Functions (3)

Idea : We use pattern matching just like in Haskell.

Rocq Code

```
Definition is_true (b : bool) : bool :=
  match b with
    | true  ⇒ true
    | false ⇒ false
  end.

Fixpoint is_even (n : nat) : bool :=
  match n with
    | 0    ⇒ true
    | S m' ⇒ negb (is_even m')
  end.
```

Functional Programming in Rocq - Constructors

Rocq Code

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.

Inductive list (A : Type) : Type :=
  | nil  : list A
  | cons : A → list A → list A
```

Interactive Theorem Proving

When writing a proof in Rocq, the IDE will give you the **proof context** and **goal** that you are trying to prove.

A **proof context** is a multiset Γ of formulas, and a **goal** F is a formula that one tries to derive from the context.

Rocq Code

Given a pair (Γ, F), writing $\Gamma = \{F_1, F_2, \ldots, F_n\}$, the Rocq IDE will display this as

```
H1 : F1
H2 : F2

Hn : Fn
______________________________________(1/1)
F
```

From here, we use elements of the context to constructively give an element of `F`. If such a proof exists, we write $\Gamma \vdash F$.

Natural Deduction

Rocq has a variety of builtin tactics, which we use in proofs.

Reasoning with natural deduction have corresponding tactics.

Example

Axiom Rule :

```
...
x : A
____________(1/1)
A
```

```
assumption. (or exact x.)
```

Example

Intro → :

```
Γ
____________(1/1)
A → B
```

```
Γ
H : A
____________(1/1)
B
```

```
intro H. (or intros H.)
```

Example

Elim $\rightarrow$:

```
Γ
____________(1/1)
B
```

```
Γ
____________(1/2)
A
```

```
Γ
H : A
____________(1/2)
B
```

```
assert A as H.
```

Example

Elim $\rightarrow$ 2 :

```
Γ
H : A → B
____________(1/1)
B

apply H.
```

```
Γ
H : A → B
____________(1/1)
A
```

Example

Intro ∧ :

```
Γ
____________(1/1)
A ∧ B
```

```
Γ
____________(1/2)
A

Γ
____________(2/2)
B
```

```
split.
```

Example

Intro ∨ :

```
Γ
____________(1/1)
A ∨ B
```

```
Γ
____________(1/1)
A
```

or

```
Γ
____________(1/1)
B
```

`left. (or right.)`

Example

Intro ∀ :

```
Γ
__________________(1/1)
forall (x : A), B
```

```
Γ
x : A
____________(1/1)
B
```

```
intro x.
```

The variable to be introduced must be free in Γ. You can alternatively just write "intro" and Rocq will give a free name to that variable.

Example

Intro ∃ :

```
Γ
__________________(1/1)
exists (x : A), B

exists t.
```

```
Γ
____________(1/1)
B[t/x]
```

If the existential is bound in B, Rocq will again automatically rename bound variables (in De Bruijn Indices fashion).

Example

```
Γ
H : A ∨ B
__________________(1/1)
C
```

```
Γ
H : A
____________(1/2)
C

Γ
H : B
____________(2/2)
C
```

```
destruct H.
```

- You can do the same with conjunctions, existentials, or on elements (splits them into constructors).
- The "as" clause lets you put a name on hypothesis, otherwise Rocq will automatically generate them.

Specialize Tactic

Example

```
Γ
t : A
H : forall (x : A), F(x)
__________________(1/1)
B

specialize(H(t)).
```

```
Γ
t : A
H : F(t)
____________(1/1)
B
```

A simple Tautology

We first illustrate proofs in Rocq with the simplest example – a tautology.

Rocq Code

```
Theorem truth : True.
  Proof.
    exact I.
  Qed.
```

The statement says that we can always derive True.
In Rocq terms, this means we can find an element in `True`.
What is `True`?

Rocq Code

```
Inductive True : Prop :=
  | I : True.
```

So the proof is straight forward. We constructively prove this by saying that `I` is an element of `True`.

Proofs with Booleans

Rocq Code

```
Theorem double_negation : forall (b : bool), negb (negb b) = b.
Proof.
  intros b.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

Tactic "simpl" unfold definitions and simplifies them (reduces them).

Aside on Equality

What's equality?

Rocq Code

```
Inductive eq (A : Type) (x : A) : A → Prop :=
  | eq_refl : x = x.
```

Key Idea : Proofs are equivalent to finding elements of the object, and this analogy is consistent even for equalities.

Proofs by Induction

Rocq Code

```
Lemma plus_0_n : forall n : nat, 0 + n = n.
Proof.
  intros n.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.
```

Proofs by Induction - continued

Rocq Code

```
Lemma plus_n_Sm : forall (n m : nat), n + S m = S (n + m).
Proof.
  induction n.
  - intros. simpl. reflexivity.
  - intros. simpl.
    specialize(IHn m). rewrite IHn.
    reflexivity.
Qed.
```

Proofs by Induction - continued

Rocq Code

```
Theorem plus_comm : forall n m, n + m = m + n.
Proof.
  intros.
  induction m.
  - simpl. apply plus_O_n.
  - simpl.
    specialize(plus_n_Sm n m). intros.
    rewrite H. rewrite IHm. reflexivity.
Qed.
```

Design Choices

Design choices alter proof methods / difficulty.

Let's revisit addition.
The original definition looks like

Rocq Code

```
Fixpoint add_nat (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S p ⇒ S (add p m)
  end.
```

Design Choices

Design choices alter proof methods / difficulty.

Let's revisit addition.
Alternatively, consider the following :

Rocq Code

```
Fixpoint add_nat (n m : nat) : nat :=
  match (n, m) with
    | (0, 0)  => 0
    | (0, _m) => _m
    | (_n, 0) => _n
    | (S _n, S _m) => S (S (add_nat _n _m))
  end.
```

This makes symmetry straightforward to prove.

Questions?