An Introduction to Coq/Rocq

Thierry Martinez

31 January 2025

What is Coq/Rocq?

- A Proof Assistant (to perform machine-checked proofs, relying on the Curry-Howard correspondance)
- A Programming Language
- An Implementation of the Calculus of Inductive Constructions CIC (with extensions: perhaps pCuIC, the Predicative Calculus of Cumulative Inductive Constructions, is the closest formalization)
- A Compiler and a Type-Checker (both at the same time, since types can compute)

Coq development started in 1984.

Coq will soon be renamed into *The Rocq Prover* (testimony of the former Inria research center in Rocquencourt where Coq is born).

A Proof Assistant

As many programming languages, Coq provides a command (‘coqc‘) to compile (and type-check) programs.

But programs are nearly impossible to write without interacting with Coq, that is what we call a Proof Assistant. The interactive loop is provided by the command ‘coqtop‘.

But ‘coqtop‘ itself remains very cumbersome to use directly. We generally use an interface on top of it: either CoqIDE, ProofGeneral, VSCoq.

A Programming Language

The Coq language has several layers:

- The Vernacular language: the top-level commands that compose the programs, to introduce `Definition`, `Theorem`, `Proof`, `Qed`, `Print`, `Check`, `Import`, etc.
- The Gallina specification language: the CIC calculus itself
- The tactics language (there are some variants: Ltac, Ltac2, SSReflect, elpi, ...)

These languages cohabit in the same program script: vernacular commands introduce definitions and theorems written in Gallina and proofs written with tactics.

These languages interleave: we can use Gallina in tactics (`exact`, `refine`), and we can use tactics to write Gallina terms (`(ltac:...)`).

An Implementation of the Calculus of Inductive Constructions CIC (or pCuIC?)

An extension of the λ-calculus:

- variable `x`
- abstraction `fun x => e` (traditionnally written as $\lambda x.e$, or more commonly in other branchs of mathematics $x \mapsto e$)
- application `f e` (or, equivalently, `f(e)`, but the parentheses are superfluous; in mathematics, we often write $\sin \alpha$)!)

with the rule called β-reduction: $(\lambda x.e_1)e_2 \rightarrow_\beta e_1[e_2/x]$, where $e_1[e_2/x]$ means e_1 where every occurrence of x has been replaced by e_2.

η-reduction: $\lambda x.f\ x \rightarrow_\eta f$, also useful backwards (η-expansion).

Expressions can be computed:

```
Coq < Eval cbv in (fun x => x + 1) 2.
     = 3
     : nat
```

Typed λ-calculus

Every variable and every expression have types (but there is a partial type inference, types can be ommited in simple cases – full type inference for CIC is indecidable).

Theorem (Strong normalization)

Every computation terminates, i.e. there is no infinite chains of β-reduction.

For instance, λ-terms such as $\Delta\Delta$ is rejected by Coq (where Δ is the λ-term $\lambda x.xx$: in pure λ-calculus, we have the chain $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \ldots$).
To perform loops, we have fixpoints (*aka* guarded recursion):

```
Fixpoint fact (n: nat) {struct n}: nat :=
  match n with
  | 0 => 1
  | S m => n * fact m
  end.
```

Types can compute

```
Fixpoint nary (n: nat): Type :=
  match n with
  | 0 => nat
  | S m => nat -> nary m
  end.

Fixpoint bigsum (n: nat) (acc: nat): nary n :=
  match n with
  | 0 => acc
  | S m => fun x => bigsum m (acc + x)
  end.

Eval cbv in bigsum 4 0 1 2 3 4.
```

Note that the branches of this `match` are heterogeneous:

- the branch `0` has type `nary 0`,
- the branch `S m` has type `nary (S m)`.

Inductive types

Types can be defined inductively. For instance, the type `nat` is defined as the smallest type that contains `O` and is closed by applications of `S` (Peano numbers).

```
Inductive nat :=
| O: nat
| S: nat -> nat.
```

`match` and **`fix`** allow us to define the recurrence principle, and **`Inductive`** defines it for us automatically.

```
Coq < Print nat_ind.
nat_ind =
fun (P: nat -> Prop) (f: P O) (f0: forall n: nat, P n -> P (S n)) =>
fix F (n: nat): P n :=
  match n as n0 return (P n0) with
  | O => f
  | S n0 => f0 n0 (F n0)
  end
    : forall P: nat -> Prop,
      P O -> (forall n: nat, P n -> P (S n)) -> forall n: nat, P n
```

Logical Conjunction as Inductive Type

```
Coq < Print and.
Inductive and (A B : Prop) : Prop :=
    conj : A -> B -> A /\ B.

Lemma and_comm': forall A B, A /\ B -> B /\ A.
Proof.
  exact (fun A B a_b =>
    match a_b with
    | conj a b => conj b a
    end
  ).
Qed.
```

`A /\ B` is an infix notation for `and A B`.

Note: we can define our own infix notations with
`Infix "/\" := and.`

Logical Disjunction as Inductive Type

```
Coq < Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B.

Lemma or_comm': forall A B, A \/ B -> B \/ A.
Proof.
  exact (fun A B a_b =>
    match a_b with
    | or_introl a => or_intror a
    | or_intror b => or_introl b
    end
  ).
Qed.
```

`\/` is an infix notation for `or`.

Proofs as Programs

Three ways of defining the same object:

```
Definition id: forall A: Prop, A -> A :=
  fun A => fun (x : A) => x.

Lemma id': forall A, A -> A.
Proof.
  exact (fun A => fun (x : A) => x).
Qed.

Lemma id'': forall A, A -> A.
Proof.
  intro A. intro x. apply x.
Qed.
```

Tactics construct proof trees

```
Lemma id'': forall A, A -> A.
Proof.
  intro A. intro x. apply x.
Qed.
```

$$\dfrac{\dfrac{\dfrac{}{\texttt{A: Prop, x: A} \vdash \texttt{x} : \texttt{A}}(\texttt{apply x})}{\texttt{A: Prop} \vdash \texttt{fun x => x} : \texttt{A -> A}}(\texttt{intro x})}{\vdash \texttt{fun A => fun x => x} : \texttt{forall A: Prop, A -> A}}(\texttt{intro A})$$

context $\vdash$ proof term to construct : goal

As we do by hand, we usually know the context and the goal of the bottom of the proof tree and we construct the proof tree from bottom to top by applying tactics (*aka* logic rules), and the proof term is constructed gradually, from the outside to the inside (this is a term with holes).

Basic tactics

$$\text{context} \vdash \text{proof term to construct} : \text{goal}$$

We already saw `exact e` that type-checks `e` against the goal.

```
Lemma id': forall A, A -> A.
Proof.
  exact (fun A => fun (x : A) => x).
Qed.
```

More generally, `refine e` takes a term `e` with holes (_), and the types of the holes become subgoals to prove.

```
Lemma id''': forall A, A -> A.
Proof.
  refine (fun A => _).
  refine (fun x => _).
  refine x.
Qed.
```

`intro x` is a shorthand for `refine (fun x => _)`.

Tactics with multiple subgoals

```
Lemma and_comm': forall A B, A /\ B -> B /\ A.
Proof.
  intros A B a_b. destruct a_b as [a b]. split.
  - apply b.
  - apply a.
Qed.
```

Bullets are optional, but enable Coq to check layout consistency.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{a\_b: A /\textbackslash\ B}, \texttt{a: A}, \texttt{b: B} \vdash \texttt{b} : \texttt{B}} \qquad \dfrac{}{\texttt{a\_b: A /\textbackslash\ B}, \texttt{a: A}, \texttt{b: B} \vdash \texttt{a} : \texttt{A}}}{\texttt{a\_b: A /\textbackslash\ B}, \texttt{a: A}, \texttt{b: B} \vdash \texttt{conj \_ \_} : \texttt{B /\textbackslash\ A}}}{\texttt{a\_b: A /\textbackslash\ B} \vdash \texttt{match a\_b with conj a b => \_ end} : \texttt{B /\textbackslash\ A}}}{\vdash \texttt{fun A B a\_b => \_} : \texttt{forall A B, A /\textbackslash\ B -> B /\textbackslash\ A}}$$

```
Lemma and_comm'': forall A B, A /\ B -> B /\ A.
Proof.
  refine (fun A B a_b => _).
  refine (match a_b with conj a b => _ end).
  refine (conj _ _). - refine b. - refine a.
Qed.
```

Tactics for and/or

We have already seen that:

- `destruct` is a shorthand for `refine (match ...)`,
- `split` is a shorthand for `refine (conj _ _)`.

We also have:

- `left`, a shorthand for `refine (or_introl _)`,
- `right` is a shorthand for `refine (or_intror _)`.

```
Lemma or_comm': forall A B, A \/ B -> B \/ A.
Proof.
  intros A B a_b. destruct a_b as [a | b].
  - right. apply a.
  - left. apply b.
Qed.
Lemma or_comm'': forall A B, A \/ B -> B \/ A.
Proof.
  refine (fun A B a_b => _).
  refine (match a_b with or_introl a => _ | or_intror b => _ end).
  - refine (or_intror _). refine a.
  - refine (or_introl _). refine b.
Qed.
```

Tactics for inductive reasoning

```
Lemma nat_ind':
  forall P, P 0 -> (forall n, P n -> P (S n)) -> forall n, P n.
Proof.
  intros P P_0 H n.
  induction n as [|n IHn].
  - apply P_0.
  - apply H. apply IHn.
Qed.

Lemma nat_ind'':
  forall P, P 0 -> (forall n, P n -> P (S n)) -> forall n, P n.
Proof.
  refine (fun P P_0 H n => _).
  refine (nat_rect _ _ _ n).
  - refine P_0.
  - clear n. refine (fun n IHn => _). refine (H _ _). refine IHn.
Qed.
```

`clear n` forgets the hypothesis `n` in the context.

`nat_rect` is a generalization of `nat_ind` for any sort of types.

A Type Hierarchy

```
Coq < Check O.
O
     : nat

Coq < Check nat.
nat
     : Set

Coq < Check Set.
Set
     : Type

Coq < Check Type.
Type
     : Type
```

Implicit Universes

```
Coq < Set Printing Universes.

Coq < Check Set.
Set
     : Type@{Set+1}

Coq < Check Type.
Type@{Top.5}
     : Type@{Top.5+1}
(* {Top.5} |=  *)

Coq < Check (forall x, x).
forall x : Type@{Top.7}, x
     : Type@{Top.7+1}
(* {Top.7} |=  *)
```

Prop Impredicativity

```
Coq < Check (forall (P: Prop), P -> P).
forall P : Prop, P -> P
     : Prop

Coq < Check (forall (P: Type), P -> P).
forall P : Type@{Top.13}, P -> P
     : Type@{Top.13+1}
(* {Top.13} |=  *)

Coq < Check (forall (P: Set), P -> P).
forall P : Set, P -> P
     : Type@{Set+1}
```

Inductive Types with Parameters

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

Coq < Check (cons 0 nil).
(0 :: nil)%list
     : list nat

Coq < Check (cons False (cons True nil)).
(False :: True :: nil)%list
     : list Prop

Require Import List.
Import ListNotations.

Fixpoint length {A} (l: list A): nat :=
  match l with
  | [] => 0
  | _hd :: tl => 1 + length tl
  end.
```

Inductive Types with Indices

```
Inductive vect {A}: nat -> Type :=
| nil: vect 0
| cons: forall {n}, A -> vect n -> vect (S n).

Fixpoint map {A B n} (f: A -> B) (v: @vect A n):
  @vect B n :=
  match v with
  | nil => nil
  | cons hd tl => cons (f hd) (map f tl)
  end.
```

`@vect` turns implicit arguments explicit again.
This definition of `map` ensures that the length is preserved!

Typing rule for `match`

$$\texttt{Inductive}\ I : \vec{i}\ \texttt{->}\ s\ \texttt{:=}$$
$$|\ C_i : \vec{t_i}\ \texttt{->}\ I\ \vec{p_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : I\ \vec{v} \\ \Gamma, \vec{u} : \vec{i}, y : I\ \vec{u} \vdash T : s' \qquad \Gamma, \vec{a_i} : \vec{t_i} \vdash e_i : T[(C_i \vec{a_i})\vec{p_i}/y\vec{u}] \\ I\ \vec{v} : s \text{ and } s \text{ can be eliminated into } s'\end{array}}{\Gamma \vdash \texttt{match}\ x\ \texttt{as}\ y\ \texttt{in}\ I\ \vec{u}\ \texttt{return}\ T\ \texttt{with}\ C_i\ \vec{a_i}\ \texttt{=>}\ e_i\ \texttt{end} : T[x\vec{v}/y\vec{u}]}$$

```
Inductive vect {A}: nat -> Type :=
| nil: vect 0
| cons: forall {n}, A -> vect n -> vect (S n).

Fixpoint map {A B n} (f: A -> B) (v: @vect A n):
  @vect B n :=
  match v in vect A m return @vect A m with
  | nil => nil (* : @vect A 0 *)
  | @cons m hd tl => cons (f hd) (map f tl) (* : @vect A (S m) *)
  end.
```

Elimination condition: Prop must be eliminated into Prop

```
Coq < Print or.
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B.

Coq < Print sum.
Inductive sum (A B : Type) : Type :=
    inl : A -> A + B | inr : B -> A + B.
```

There is an injection from `A + B` to `A \/ B`, but we cannot define the reverse injection, since a pattern-matching over a term of type `or` can only return a `Prop`.

```
Coq < Print sig.
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x : A | P x}.
```

There is an injection from `{x : A | P x}` to `exists x : A, P x`, but we cannot define the reverse injection.

bigsum again

```
Fixpoint nary (n: nat): Type :=
  match n with
  | 0 => nat
  | S m => nat -> nary m
  end.

Fixpoint bigsum (n: nat) (acc: nat): nary n :=
  match n as n0 return nary n0 with
  | 0 => acc (* : nary 0 *)
  | S m => fun x => bigsum m (acc + x) (* : nary (S m) *)
  end.
```

Coq Equality is an Inductive Type

```
Coq < Print eq.
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : x = x.
```

`x = y` is an infix notation for `eq x y`.

$$\frac{\Gamma \vdash e : x = y \qquad \Gamma, y' : A \vdash f\ y' : s' \qquad \Gamma \vdash e : f\ x}{\Gamma \vdash \texttt{match}\ e\ \texttt{in}\ \_ = y'\ \texttt{return}\ f\ y'\ \texttt{with eq\_refl} \Rightarrow e\ \texttt{end} : f\ y}$$

This typing rule is a type cast: we can convert terms of type `x` into terms of type `y`.

```
Definition type_cast {A B} (e: A = B) (v: A): B :=
  match e in _ = t return t with
  | eq_refl => v
  end.
```

Coq Equality is Reflexive

```
Lemma eq_refl': forall A (x: A), x = x.
Proof.
  intros. reflexivity.
Qed.

Lemma eq_refl'': forall A (x: A), x = x.
Proof.
  refine (fun A x => _). refine eq_refl.
Qed.
```

Note: these are η-expansions of `eq_refl`.

Coq Equality is Symmetric

```
Lemma eq_sym': forall {A} {x y: A}, x = y -> y = x.
Proof.
  intros A x y H. symmetry. apply H.
Qed.

Lemma eq_sym'': forall A (x y: A), x = y -> y = x.
Proof.
  refine (fun A x y H => _).
  refine (
    match _ in _ = y' return y' = x with
    | eq_refl => eq_refl
    end
  ).
  refine H.
Qed.
```

Coq Equality is Transitive

```
Lemma eq_trans':
  forall A (x y z: A), x = y -> y = z -> x = z.
Proof.
  intros A x y z x_y y_z. transitivity y.
  - apply x_y.
  - apply y_z.
Qed.
Lemma eq_trans'':
  forall A (x y z: A), x = y -> y = z -> x = z.
Proof.
  refine (fun A x y z x_y y_z => _).
  refine (
    match eq_sym (y := y) _ in _ = y' return y' = z with
    | eq_refl => _
    end
  ). - refine x_y. - refine y_z.
Qed.
```

Coq Equality is equivalent to Leibniz Equality

Definition (Leibniz Equality)

Two terms are equal if they satisfy the same properties (for all properties).

```
Lemma eq_leibniz:
  forall A (x y: A), x = y <-> forall P: A -> Prop, P y -> P x.
Proof.
  intros A x y. split; intro H.
  - intro P. intro HP. rewrite H. apply HP.
  - apply H. reflexivity.
Qed.
Lemma eq_leibniz':
  forall A (x y: A), x = y <-> forall P: A -> Prop, P y -> P x.
Proof.
  refine (fun A x y => _).
  refine (conj _ _); refine (fun H => _).
  - refine (fun P => _). refine (fun HP => _).
    refine (match eq_sym H with eq_refl => _ end).
    refine HP.
  - refine (H (fun _ => _) _). refine eq_refl. Qed.
```

Tactic composition

t ; t' executes t and then executes t' on each subgoal generated by t.

t ; [t_1 | $\cdots$ | t_n] executes t and then executes t_i on the ith subgoal generated by t.

t + t' executes t and, if t fails, executes t'.
`try` t is a shorthand for t + `idtac`.

n:{...} focuses on the nth subgoal (interactively).

Pattern-matching with absurd cases

```
Inductive vect {A}: nat -> Type :=
| nil: vect O
| cons: forall {n}, A -> vect n -> vect (S n).
Definition tail {A n} (v: @vect A (S n)): @vect A n :=
  match v with
  | cons _ tl => tl
  end.
Definition tail' {A n} (v: @vect A (S n)): @vect A n :=
  match v in @vect _ m return
    match m with
    | O => unit
    | S n' => @vect A n'
    end
  with
  | nil => tt
  | cons _ tl => tl
  end.
```

Pattern-matching with convoy pattern

`match` with non-linear dependencies between types (`n` occurs several times in the context).

convoy pattern : η-expansion around `match`.

```
Fixpoint map2 {A B C n} (f: A -> B -> C)
  (u: @vect A n) (v: @vect B n): @vect C n :=
  match u in @vect _ n' return @vect B n' -> _ with
  | nil => fun v =>
    match v with
    | nil => nil
    end
  | cons uh ut => fun v =>
    match v with
    | cons vh vt => fun ut =>
      cons (f uh vh) (map2 f ut vt)
    end ut
  end v.
```