Introduction to the Coq system

Hugo Herbelin

Coq Tutorial at ITP'15, Nanjing, 27 August 2015 (revised 29 August 2015)

Coq: A proof assistant

A software providing an environment for interactively or semi-automatically developing mathematical proofs and certified programming.

Examples of similar softwares:

- Boyer-Moore's NqThm, now ACL2
- PVS (general purpose)
- HOL4
- Hol-Light
- Isabelle-HOL (general purpose)
- Mizar (set theory, mathematically oriented, large database of mathematics, controlled natural language)
- Agda (richly-typed programming-as-proving oriented)

Coq: specificities

Coq: general purpose, pretty mature, based on a formalism which is both a very expressive and natural logic and a richly-typed programming language (the Calculus of Inductive Constructions, CIC)

Three main components:

- a kernel ensuring correctness of proof certificates
- a concrete user language featuring high-level convenient features (type classes, implicit arguments, coercions, user notations, ...)
- a programmable multi-purpose proof language with a range of beginners-to-experts interactive and automated proof methods (tactics)

$+$ various extra features

- extraction of programs to OCaml, Haskell, ...
- libraries
- user interfaces

An overview of the Coq formalism (CIC)

Coq's logical formalism: The Calculus of Inductive Constructions

A formalism derived from a long-standing scientific history:

- Intuitionistic logic: a proof is a process which produces witnesses for existential statements, and effective proofs for disjunction (ruling out, say, $A \vee \neg A$, i.e. A `\/` `~`A in Coq notation, or

$$\exists x \forall y (P(x) \to P(y))$$

  i.e.

$$\texttt{exists}\ x, \texttt{forall}\ y, (Px \texttt{->}\ Py)$$

  in Coq notation)

- The proofs-as-programs, formulas-as-type correspondence (Curry 1958, Howard 1968) The language of proofs is a programming language. E.g. the proof of an implication A`->` B can be represented as a function `fun` $a : A$ `=>` *some proof of* B *depending on a proof of* A)

Coq's logical formalism: The Calculus of Inductive Constructions (continued)

A formalism derived from a long-standing scientific history:

- Martin-Löf's intuitionistic type theory (from 1975, proofs-as-λ-terms, propositions-as-sets, types are themselves *sorted*, inductive types, ...)

  In Coq's syntax, inductive types looks like

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

- Girard-Reynolds' System F (1971, impredicativity of propositions)

  E.g., in Coq, one can represent formulas of the form

$$\texttt{forall}\ A : \texttt{Prop}, A \texttt{->}\ A$$

Coq's logical formalism: The Calculus of Inductive Constructions (continued)

A formalism derived from a long-standing scientific history:

- Coquand's Calculus of Constructions (1984)

  The strength of higher-order logic, but no primitive inductive types

- Coquand-Paulin's Calculus of Inductive Constructions (1988)

  A merge of the Calculus of Constructions with Martin-Löf's type theory

- Coq V8.0 predicative's Calculus of Inductive Constructions (2004)

  A weakening of the logic so that it is compatible with classical logic and axiom of choice.

Coq's logical formalism: The Calculus of Inductive Constructions, syntax

A concise primitive language of expressions:

expr ::=	`Type` | `Set` | `Prop`	(sorts)
|	`forall` x : *expr* `,` *expr*	(universal quantification / dependent function type)
|	`fun` x : *expr* `=>` *expr*	(function abstraction over a variable)
|	`let` x `:=` $expr_1$ `in` $expr_2$	(local definitions)
|	x	(a name, referring either to a bound variable, a global constant, an inductive type or a constructor
|	$expr_1$ $expr_2$	(function application)
|	`match` *expr* `with`	(case analysis)
	`|` $C_1 x_{11} \ldots x_{1n_1}$ `=>` $expr_1$	
	$\vdots$	
	`|` $C_p x_{p1} \ldots x_{pn_p}$ `=>` $expr_p$	
	`end`	
|	`fix` f $(x_1 : expr_1) \ldots (x_n : expr_n) : expr := expr$	(well-founded recursion)
|	`cofix` f $(x_1 : expr_1) \ldots (x_n : expr_n) : expr := expr$	(guarded co-recursion)

and slight variants of them...

Coq's logical formalism: The Calculus of Inductive Constructions, syntax

Note: $\texttt{forall}\ x : \mathit{expr}_1\texttt{,}\ \mathit{expr}_2$ is also known as *dependent product*

All of $\texttt{forall}\ x : \mathit{expr}_1\texttt{,}\ \mathit{expr}_2$, $\texttt{fun}\ x : \mathit{expr}_1\ \texttt{=>}\ \mathit{expr}_2$ and $\texttt{let}\ x := \mathit{expr}_1\ \texttt{in}\ \mathit{expr}_2$ are *binding* x in expr_2. Conversely, the variable x is called bound in expr_2.

$\mathit{expr}_1 \mathit{expr}_1) \ldots \mathit{expr}_n$ is the same as $(\ldots(\mathit{expr}_0 \mathit{expr}_1)\ldots)\mathit{expr}_n$

$$\texttt{fun}\ (x_1 : \mathit{expr}_1)\ldots(x_n : \mathit{expr}_n)\ \texttt{=>}\ \mathit{expr}$$

is the same as

$$\texttt{fun}\ x_1 : \mathit{expr}_1\ \texttt{=>}\ \ldots\texttt{fun}\ x_n : \mathit{expr}_n\ \texttt{=>}\ \mathit{expr}$$

$$\texttt{forall}\ (x_1 : \mathit{expr}_1)\ldots(x_n : \mathit{expr}_n)\texttt{,}\ \mathit{expr}$$

is the same as

$$\texttt{forall}\ x_1 : \mathit{expr}_1\texttt{,}\ \ldots\texttt{forall}\ x_n : \mathit{expr}_n\texttt{,}\ \mathit{expr}$$

Coq's logical formalism: types

Any semantically well-formed expression has a *type*.

Types are themselves expressions, so any type has itself a type, which is a *sort*

Sorts are types and are hence themselves expressions.

The types form a subset of expressions, hereafter written *type*.

The sorts of the Calculus of Inductive Constructions

`Prop`: the sort of propositions

Examples: $\mathit{expr}_1 = \mathit{expr}_2$, $0 \leq 1$, `True`, `False`, `True-> False`, $0 = 0$ `/\` $1 \leq 2$, $0 = 0$ `\/` $1 \leq 2$, $0 = 0$ `<->` $1 \leq 2$,... are propositions (using names and notations defined in the initial state of Coq)

`Set`: the sort of "small" (data-)types

Examples: `nat`, `bool`, `list nat`, `option bool`, `nat-> bool`, ... are sets (using names defined in the initial state of Coq)

$\mathtt{Type}_1$: the sort of types, including `Prop` and `Set` seen themselves as types

$\mathtt{Type}_2$: the sort of types of level 2, including `Prop`, `Set` and $\mathtt{Type}_1$ seen themselves as types

...

$\mathtt{Type}_n$: the sort of types of level n

In practice: n is left implicit as it is inferred by Coq (one simply write `Type`). So, users only see `Prop`, `Set` and `Type`.

The general components of a Coq document

Gallina: A concise primitive language for expressing logical theories:

$$
\begin{array}{lll}
\textit{decl} & ::= & \texttt{Definition}\ c\ (x_1 : \textit{type}_1) \ldots (x_n : \textit{type}_n) : \textit{type} := \textit{expr}. \\
& | & \texttt{Axiom}\ c : \textit{type}. \\
& | & \texttt{Parameter}\ c : \textit{type}. \\
& | & \texttt{Theorem}\ c\ (x_1 : \textit{type}_1) \ldots (x_n : \textit{type}_n) : \textit{type}.\ \texttt{Proof.}\ \ldots\textit{proof script}\ldots\ \texttt{Qed.} \\
& | & \texttt{Inductive}\ I\ (x_1 : \textit{type}_1) \ldots (x_n : \textit{type}_n) : \textit{type} := C_1 : \textit{type}_1 \mid \ldots \mid C_p : \textit{type}_p \\
& | & \texttt{CoInductive}\ I\ (x_1 : \textit{type}_1) \ldots (x_n : \textit{type}_n) : \textit{type} := C_1 : \textit{type}_1 \mid \ldots \mid C_p : \textit{type}_p
\end{array}
$$

and variants (`Fixpoint`, `CoFixpoint`, `Record`, ...)

$\mathcal{L}_{tac}$: An extensive (and extensible) language of tactics to write proof scripts.

The vernacular: An extensive language of commands to manage the proof development environment (notations, implicit arguments, coercions, type classes, ...).

Inductive and coinductive types

A general scheme to introduce new types (i.e. sets, propositions, general types) by *constructors*.

Inductive types can be recursive if the recursion is strictly covariant (so-called *strict positivity* condition):

Dependency in types

Let us consider an expression `forall` x : *expr*$_1$, *expr*$_2$.

If x occurs in *expr*$_2$, one says that *expr*$_2$ depends on x, or, alternatively, that `forall` x : *expr*$_1$, *expr*$_2$ is a dependent function type.

When x is not dependent in *expr*$_2$, one writes *expr*$_1$`->` *expr*$_2$.

How to recognize sets, types and propositions?

The expression `forall` a : *expr*$_1$`,` *expr*$_2$ is a proposition (resp. set, type) whenever *expr*$_2$ is.

The expression *expr*$_1$`->` *expr*$_2$ is a proposition (resp. set, type) when *expr*$_1$ and *expr*$_2$ are.

When *expr*$_1$ is a type and *expr*$_2$ is `Prop`, *expr*$_1$`->` *expr*$_2$ denotes the types of predicates over the type *expr*$_1$.

Example: `nat-> Prop` is the type of predicates over a natural number.

For instance, `forall P:nat -> Prop, P 0 -> P 1` expresses that 0 and 1 are indistinguishable, in the sense that for any property, if the property holds for 0, it holds for 1 too.

Focusing on the sub-language which implements logic

Expressing logical connectives and quantifiers in Coq

Implication is expressed

```
A -> B
```

Universal quantification over domain T

```
forall x:T , A
```

Example:

```
forall x:nat, forall y:nat, x = y -> y = x
```

abbreviated

```
forall x y:nat, x = y -> y = x
```

(we shall see later on how the predicate = and the set `nat` are defined)

Expressing logical connectives and quantifiers in Coq (continued)

Note: on the contrary of common mathematical practice, in Coq, `forall` binds to the end of the expression. E.g.

```
forall A:Prop, A -> forall B:Prop, B -> False
```

means

```
forall A:Prop, (A -> forall B:Prop , (B -> False))
```

and not

```
(forall A:Prop, A) -> (forall B:Prop, B) -> False
```

The other connectives are defined

Falsity is defined inductively as a proposition with no constructor.

```
Inductive False : Prop := .
```

True is defined inductively as a proposition with a constructor with no argument.

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

Conjunction $A \wedge B$ is defined inductively as a parametric proposition with a constructor expecting a proof of A and a proof of B.

```
Inductive and (A B:Prop) : Prop := conj : A -> B -> A /\ B
where "A /\ B" := (and A B).
```

Disjunction $A \vee B$ is defined inductively as a parametric proposition with two constructors, one expecting a proof of A and the other a proof of B.

```
Inductive or (A B:Prop) : Prop :=
| or_introl : A -> A \/ B
| or_intror : B -> A \/ B
where "A \/ B" := (or A B).
```

The other connectives (continued)

Negation `~A` is defined as an abbreviation:

```
Definition not (A:Prop) := A -> False.
Notation "~ A" := (not A).
```

Existential quantification $\exists x : A.P(x)$ is defined inductively:

```
Inductive ex (A:Type) (P:A->Prop) : Prop :=
  ex_intro : forall x:A, P x -> exists x : A, P x
where "exist x : A , P" := (ex A (fun x => P)).
```

Equality $t = u$ is defined as an inductive predicate:

```
Inductive eq (A:Type) (a:A) : A -> Prop := refl : a = a
where "t = u" := (eq A t u).
```

Sets and types

The *unit* type is defined inductively:

```
Inductive unit : Set :=
| tt : unit.
```

The *Boolean* type is defined inductively:

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

The type of *natural number* is defined inductively:

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

Sets and types (continued)

The type of *list* is defined inductively with a parameter:

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

Similarly, the *option* type is defined:

```
Inductive option (A:Type) : Type :=
| None : list A
| Some : A -> list A.
```

The *function type* is given by `->` .

Dependent function types will be shown later.