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, $A \lor \neg A$, i.e. $A \nlor \sim A$ in Coq notation, or

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

  i.e.

  exists $x$, forall $y$, ($Px \rightarrow Py$)

  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 $A \rightarrow B$ can be represented as a function

  \[
  \text{fun } a : A \rightarrow \text{some proof of } B \text{ depending on a proof of } A
  \]
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-\(\lambda\)-terms, propositions-as-sets, types are themselves sorted, inductive types, ...)
  In Coq’s syntax, inductive types looks like

  \[
  \text{Inductive nat : Type :=}
  \]
  \[
  \mid 0 : \text{nat} \\
  \mid S : \text{nat -> nat}.
  \]

• Girard-Reynolds’ System F (1971, impredicativity of propositions)
  E.g., in Coq, one can represent formulas of the form

  \[
  \text{forall } A : \text{Prop}, A \rightarrow A
  \]
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:

\[
expr ::= \text{Type} \mid \text{Set} \mid \text{Prop} \quad \text{(sorts)}
\]
\[
\mid \forall x : expr, expr \quad \text{(universal quantification / dependent function type)}
\]
\[
\mid \text{fun } x : expr \Rightarrow expr \quad \text{(function abstraction over a variable)}
\]
\[
\mid \text{let } x := expr_1 \text{ in } expr_2 \quad \text{(local definitions)}
\]
\[
\mid x \quad \text{(a name, referring either to a bound variable,}
\]
\[
\text{a global constant, an inductive type or a constructor}
\]
\[
\mid expr_1 \ expr_2 \quad \text{(function application)}
\]
\[
\mid \text{match } expr \text{ with}
\]
\[
\mid \quad C_1x_{1_1}\ldots x_{1_{n_1}} \Rightarrow expr_1
\]
\[
\quad \ldots
\]
\[
\mid \quad C_p x_{p_1}\ldots x_{p_{n_p}} \Rightarrow expr_p
\]
\[
\text{end}
\]
\[
\mid \text{fix } f (x_1 : expr_1) \ldots (x_n : expr_n) : expr := expr
\quad \text{(well-founded recursion)}
\]
\[
\mid \text{cofix } f (x_1 : expr_1) \ldots (x_n : expr_n) : expr := expr
\quad \text{(guarded co-recursion)}
\]

and slight variants of them...
Coq's logical formalism: The Calculus of Inductive Constructions, syntax

Note: forall $x : expr_1$, $expr_2$ is also known as dependent product

All of forall $x : expr_1$, $expr_2$, fun $x : expr_1 \Rightarrow expr_2$ and let $x := expr_1$ in $expr_2$
are binding $x$ in $expr_2$. Conversely, the variable $x$ is called bound in $expr_2$.

$expr_1 \ldots expr_n$ is the same as $(\ldots (expr_0 expr_1) \ldots) expr_n$

fun $(x_1 : expr_1) \ldots (x_n : expr_n) \Rightarrow expr$

is the same as

fun $x_1 : expr_1 \Rightarrow \ldots \text{fun } x_n : expr_n \Rightarrow expr$

forall $(x_1 : expr_1) \ldots (x_n : expr_n)$, $expr$

is the same as

forall $x_1 : expr_1$, $\ldots$forall $x_n : expr_n$, $expr$
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: $expr_1 = expr_2$, $0 \leq 1$, True, False, $\text{True} \rightarrow \text{False}$, $0 = 0 \lor 1 \leq 2$, $0 = 0 \lor 1 \leq 2$, $0 = 0 \leftrightarrow 1 \leq 2$, ... 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)

**Type$_1$**: the sort of types, including Prop and Set seen themselves as types

**Type$_2$**: the sort of types of level 2, including Prop, Set and Type$_1$ seen themselves as types

... 

**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:

\[
\text{decl ::= } \begin{align*}
\text{Definition} & \ c(x_1 : \text{type}_1) \ldots (x_n : \text{type}_n) : \text{type} := \text{expr}. \\
\text{Axiom} & \ c : \text{type}. \\
\text{Parameter} & \ c : \text{type}. \\
\text{Theorem} & \ c(x_1 : \text{type}_1) \ldots (x_n : \text{type}_n) : \text{type}. \text{ Proof. } \ldots \text{proof script... Qed.} \\
\text{Inductive} & \ I(x_1 : \text{type}_1) \ldots (x_n : \text{type}_n) : \text{type} := C_1 : \text{type}_1 \mid \ldots \mid C_p : \text{type}_p \\
\text{CoInductive} & \ I(x_1 : \text{type}_1) \ldots (x_n : \text{type}_n) : \text{type} := C_1 : \text{type}_1 \mid \ldots \mid C_p : \text{type}_p
\end{align*}
\]

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

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

\textbf{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 $expr_1 \rightarrow expr_2$. 
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 $expr_1 \rightarrow expr_2$ is a proposition (resp. set, type) when $expr_1$ and $expr_2$ are.

When $expr_1$ is a type and $expr_2$ is Prop, $expr_1 \rightarrow expr_2$ denotes the types of predicates over the type $expr_1$.

Example: $\text{nat} \rightarrow \text{Prop}$ is the type of predicates over a natural number.

For instance, $\forall P : \text{nat} \rightarrow \text{Prop}, P \, 0 \rightarrow P \, 1$ 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

\[ A \rightarrow B \]

*Universal quantification over domain* \( T \)

\[ \forall x : T \ , \ A \]

Example:

\[ \forall x : \text{nat}, \forall y : \text{nat}, \ x = y \rightarrow y = x \]

abbreviated

\[ \forall x \ y : \text{nat}, \ x = y \rightarrow y = x \]

(we shall see later on how the predicate \( = \) and the set \( \text{nat} \) 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.

\[
\text{Inductive False : Prop := .}
\]

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

\[
\text{Inductive True : Prop := I : True.}
\]

*Conjunction* \(A \land B\) is defined inductively as a parametric proposition with a constructor expecting a proof of \(A\) and a proof of \(B\).

\[
\text{Inductive and (A B:Prop) : Prop := conj : A -> B -> A \land B}
\]

where "A \land B" := (and A B).

*Disjunction* \(A \lor B\) is defined inductively as a parametric proposition with two constructors, one expecting a proof of \(A\) and the other a proof of \(B\).

\[
\text{Inductive or (A B:Prop) : Prop :=}
\]

\[
| \text{or_introl : A -> A \lor B}
\]

\[
| \text{or_intror : B -> A \lor B}
\]

where "A \lor B" := (or A B).
The other connectives (continued)

*Negation* \( \neg A \) is defined as an abbreviation:

Definition not \((A:\text{Prop}) := A \rightarrow \text{False.} \)
Notation "\( \neg A " := (\text{not } A) .

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

Inductive ex \((A:\text{Type}) (P:A\rightarrow\text{Prop}) : \text{Prop} := \)
\( \text{ex}_\text{intro} : \forall x:A, P x \rightarrow \exists x : A, P x \)
where "\( \exists x : A , Q " := (\text{ex } A (\text{fun } x \Rightarrow Q)).

*Equality* \( t = u \) is defined as an inductive predicate:

Inductive eq \((A:\text{Type}) (a:A) : A \rightarrow \text{Prop} := \text{refl} : a = a \)
where "\( t = u " := (\text{eq } A t u).
The *unit* type is defined inductively:

\[
\text{Inductive unit : Set := } \\
| \text{tt : unit.}
\]

The *Boolean* type is defined inductively:

\[
\text{Inductive bool : Set := } \\
| \text{true : bool } \\
| \text{false : bool.}
\]

The type of *natural number* is defined inductively:

\[
\text{Inductive nat : Set := } \\
| \text{O : nat } \\
| \text{S : nat \rightarrow nat.}
\]
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 : option A
| Some : A -> option A.
```

The *function type* is given by \( \rightarrow \).

Dependent function types will be shown later.