Introduction to the Coq proof assistant

Frédéric Blanqui (INRIA)

16 March 2013

These notes have been written for a 7-days school organized at the Institute of Applied Mechanics and Informatics (IAMA) of the Vietnamese Academy of Sciences and Technology (VAST) at Ho Chi Minh City, Vietnam, from Tuesday 12 March 2013 to Tuesday 19 March 2013. The mornings were dedicated to theoretical lectures introducing basic notions in mathematics and logic for the analysis of computer programs [2]. The afternoons were practical sessions introducing the OCaml programming language (notes in [3]) and the Coq proof assistant (these notes).

Coq is a proof assistant, that is, a tool to formalize mathematical objects and help make proofs about them, including pure functional programs like in OCaml. Its development started in 1985 [5] and has been improved quite a lot since then. It is developed at INRIA, France. You can find some short history of Coq on its web page.

1 Before starting

I suggest to use a computer running the Linux operating system, *e.g.* the Ubuntu distribution (but Coq can also be used under Windows). Then, you have to install the following software:

- emacs: a text editor
- proofgeneral: an Emacs mode for Coq (and other proof assistants)

Alternatively, you can use Coq's own GUI CoqIDE.

In Ubuntu, the installation is easy: run the "Ubuntu software center", search for these programs and click on "Install".

See the companion paper on OCaml [3] for a list of useful Emacs shortcuts. There are also shortcuts specific to ProofGeneral: see the Emacs menus for ProofGeneral and Coq.

2 Defining inductive types and functions in Coq

All the things that we have previously done in OCaml (see [3]) can be done in Coq as well, although with a slightly different syntax.

```
(* Type of finite sequences of natural numbers. *)

Inductive seq :=
| Empty : seq
| Add : nat -> seq -> seq.

(* Examples of lists. *)

Definition s123 := Add 1 (Add 2 (Add 3 Empty)).
Definition s456 := Add 4 (Add 5 (Add 6 Empty)).

(* Length of a list. *)

Fixpoint length l :=
  match l with
    | Empty => 0
    | Add x l' => 1 + length l'
  end.

Eval compute in s123.

(* Concatenation function. *)

Fixpoint concat l1 l2 :=
  match l1 with
    | Empty => l2
    | Add x l1' => Add x (concat l1' l2)
  end.

Eval compute in (concat s123 s456).

(* Reverse function. *)

Fixpoint add_at_the_end x l :=
  match l with
    | Empty => Add x Empty
    | Add y l' => Add y (add_at_the_end x l')
  end.

Fixpoint reverse l :=
  match l with
```

```
    | Empty => Empty
    | Add x l' => add_at_the_end x (reverse l')
  end.

Eval compute in (reverse s123).

(* Comparison function on [nat]. In Coq, [>] is a notation for the
predicate [gt]. [gt] is a relation, i.e. a function into the type
[Prop] of propositions, and not a function into the type [bool] of the
boolean values [true] and [false]. We cannot define a function using
[>], but we can define a boolean function [bool_gt] implementing [>]
as follows instead. *)

Check gt.

(* We need to use the Coq standard library on arithmetic. *)

Require Import Arith.

Definition bool_gt x y :=
  match le_gt_dec x y with
    | left _ => (* x <= y *) false
    | right _ => (* x > y *) true
  end.

Check bool_gt.

(* Sorting function. *)

Fixpoint insert x l :=
  match l with
    | Empty => Add x Empty
    | Add y l' =>
      if bool_gt x y
      then (* x > y *) Add y (insert x l')
      else (* x <= y *) Add x l
  end.

Fixpoint sort l :=
  match l with
    | Empty => Empty
    | Add x l' => insert x (sort l')
  end.

Eval compute in (sort (concat s456 s123)).
```

3 Proving theorems in Coq

To prove some properties of these functions, we are going to use the following tactics:

Shape of the goal	Tactic	Effect
`forall x, P`	`intro y`	Introduction rule for universal quantification (see [2]): replace the current goal by $\texttt{P}\{\texttt{x} \mapsto \texttt{y}\}$.
`forall l : seq, P`	`induction l`	Application of induction principle on lists (see [2]: replace the current goal by two new goals $\texttt{P}\{\texttt{l} \mapsto \texttt{Empty}\}$ and $\texttt{P}\{\texttt{l} \mapsto \texttt{Add n l}'\}$.
any goal `P`	`simpl`	Simplify the current goal by unfolding function definitions.
any goal `P`	`rewrite e`	Where `e` is any term of type `forall x1...xn, t = u`. Replace every subterm of `P` matching `t` with substitution σ by σ(`u`) (see [2]).
`t = t`	`reflexivity`	Close the goal.

```
Lemma length_concat : forall l1 l2,
  length (concat l1 l2) = length l1 + length l2.

Proof.
  induction l1.
  (* case Empty *)
  intro l2. simpl. reflexivity.
  (* case Add *)
  intro l2. simpl. rewrite IHl1. reflexivity.
Qed.

Print length_concat.

Lemma length_add_at_the_end : forall x l,
  length (add_at_the_end x l) = S (length l).

Proof.
  intro x. induction l.
  (* case Empty *)
  simpl. reflexivity.
  (* case Add *)
```

```
  simpl. rewrite IHl. reflexivity.
Qed.

Lemma length_reverse : forall l, length (reverse l) = length l.

Proof.
  induction l.
  (* case Empty *)
  simpl. reflexivity.
  (* case Add *)
  simpl. rewrite length_add_at_the_end. rewrite IHl. reflexivity.
Qed.

Lemma length_insert : forall x l, length (insert x l) = S (length l).

Proof.
  intro x. induction l.
  (* case Empty *)
  simpl. reflexivity.
  (* case Add *)
  simpl. destruct (bool_gt x n).
  simpl. rewrite IHl. reflexivity.
  simpl. reflexivity.
Qed.

Lemma length_sort : forall l, length (sort l) = length l.

Proof.
  induction l.
  simpl. reflexivity.
  simpl. rewrite length_insert. rewrite IHl. reflexivity.
Qed.
```

4 Going further

You can find more material on Coq on its web page `http://coq.inria.fr` including:

- The reference manual.
- Tutorials.
- The standard library.
- Links towards other Coq libraries.

You can also read the following books: [1, 4, 6].

There is also a mailing list coq-club@inria.fr where you can ask questions and follow discussions about Coq and its applications.

References

[1] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[2] F. Blanqui. Elements of mathematics and logic for the analysis of computer programs. Lecture notes available on `https://who.rocq.inria.fr/Frederic.Blanqui/`, March 2013.

[3] F. Blanqui. Introduction to the OCaml programming language. Lecture notes available on `https://who.rocq.inria.fr/Frederic.Blanqui/`, March 2013.

[4] A. Chlipala. Certified programming with dependent types. `http://adam.chlipala.net/cpdt/`, 2013.

[5] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *Proceedings of the European Conference on Computer Algebra*, Lecture Notes in Computer Science 203, 1985.

[6] B. Pierce *et al.* Software foundations. `http://www.cis.upenn.edu/~bcpierce/sf/`, 2012.