Building Verified Program Analyzers in Coq
A Tutorial

Lecture 2: Coq Crash Course

David Pichardie - INRIA Rennes / Harvard University

First steps with Coq

As any proof assistant, Coq provides

- a programming language to write executable programs
- a specification language to write properties about these programs
- a proof language to write interactive proofs (and each proof can be automatically checked by the system)

Coq's original approach:

- all these languages are in fact based on a same foundational core language: the Calculus of Inductive Constructions (CIC)

But what makes Coq original, makes it also hard to understand at first sight

- some newbies try to first understand CIC before actually using Coq
- a same keyword may be used to define (apparently) different notions

First steps with Coq

We first try to present the tool without first revealing its underlying foundations

We will assume you are familiar with a functional programming language as OCaml and show how to write similar programs in Coq.

As a running example, we will consider functional maps

- remember that in our verified analyser, we take care of the implementation of the analyser: good data structures are needed
- functional maps are the best we can do with mutable arrays (but provides far better memory sharing for free)

Our MAP signature in OCaml

```
module type MAP =
 sig
  (** the type of map keys *)
  type key
  (** the type of map elements *)
  type elt
  (** the type of maps *)
  type t
  (** [default] is the default element in a map *)
  val default : elt
  (** [get m k] returns the elements that is binded with key [k] *)
  val get : t -> key -> elt
  (** [empty] is a map that binds every keys to [default] *)
  val empty : t
  (** [set m k e] returns a new map that contains the same
      bindings as map [m] except for key [k] that is now binded with
      the element [e] *)
  val set : t -> key -> elt -> t
 end
```

Some tactics

A Coq (incomplete) survival kit

intros

```
                          intros        a : A
======================  ---------->  ======================
 forall (a : A), P                    P
```

```
                          intros        P : Prop
======================  ---------->  ======================
 P -> Q                               Q
```

```
                                        P : Prop
                          intros        Q : Prop
======================  ---------->  ======================
 P -> Q -> R                          R
```

intros

```
                                            a : A
                        intros
=====================  -------->  =====================
 forall (a : A), P                 P
```

You should think about a term of type **Prop** as a logical property

```
                                            P : Prop
                        intros
=====================  -------->  =====================
 P -> Q                            Q
```

```
                                            P : Prop
                                            Q : Prop
                        intros
=====================  -------->  =====================
 P -> Q -> R                       R
```

intros

intros

```
=====================
 forall (a : A), P
```

→

```
a : A
=====================
 P
```

You should think about a term of type **Prop** as a logical property

intros

```
=====================
 P -> Q
```

→

```
P : Prop
=====================
 Q
```

intros

```
=====================
 P -> Q -> R
```

→

```
P : Prop
Q : Prop
=====================
 R
```

In Coq, we often use the form $P \Rightarrow Q \Rightarrow R$ instead of $P \land Q \Rightarrow R$

destruct

on a term with an inductive type

```
n : nat
====================
P n
```

destruct n

first subgoal

```
====================
P 0
```

second subgoal

```
m : nat
====================
P (S m)
```

simpl

```
Definition pred (n:nat) :=
  match n with
    | O => O
    | S m => m
  end.
```

```
                             simpl
====================     ------------->     ====================
 ... pred 0 ...                               ... 0 ...

                             simpl
====================     ------------->     ====================
 ... pred (S n) ...                           ... n ...
```

But the behavior of the command is not always that simple...

(see recent improvements in Coq 8.4)

tac1 ; tac2

$$\overline{\text{goal0}} \xrightarrow{\textit{tac1}} \overline{\text{goal1}} \xrightarrow{\textit{tac2}} \overline{\text{goal2}}$$

$$\overline{\text{goal0}} \xrightarrow{\textit{tac1} \ ; \ \text{tac2}} \overline{\text{goal2}}$$

If *tac1* generates several subgoals, *tac2* is applied on each of them.

induction

on a term with an inductive type

```
n : nat
=====================
P n
```

induction n

first subgoal

```
=====================
P 0
```

second subgoal

```
m : nat
IH : P m
=====================
P (S m)
```

apply

```
H : P -> Q
====================
 Q
```

apply H →

```
H : P -> Q
====================
 P
```

```
H : forall x y, P y -> Q x y
============================
 Q a (f a)
```

apply H →

```
H : forall x y, P y -> Q x y
============================
 P (f a)
```

```
H : forall x y, P x y -> Q y
============================
 Q (f a)
```

apply H **with** a →

```
H : forall x y, P x y -> Q y
============================
 P a (f a)
```

apply

```
H : P -> Q
====================
 Q
```

apply H

```
H : P -> Q
====================
 P
```

```
H : forall x y, P y -> Q x y
============================
 Q a (f a)
```

apply H

```
H : forall x y, P y -> Q x y
============================
 P (f a)
```

Coq guesses how to instantiate the quantifiers

```
H : forall x y, P x y -> Q y
============================
 Q (f a)
```

apply H with a

```
H : forall x y, P x y -> Q y
============================
 P a (f a)
```

apply

```
H : P -> Q
=====================
 Q
```

apply H →

```
H : P -> Q
=====================
 P
```

```
H : forall x y, P y -> Q x y
============================
 Q a (f a)
```

apply H →

```
H : forall x y, P y -> Q x y
============================
 P (f a)
```

Coq guesses how to instantiate the quantifiers

```
H : forall x y, P x y -> Q y
============================
 Q (f a)
```

apply H → with a

```
H : forall x y, P x y -> Q y
============================
 P a (f a)
```

We have to help Coq and give him the missing instantiation

rewrite

```
H : a = b
=====================
 ... a ...
```

rewrite H →

```
H : b = b
=====================
 ... b ...
```

```
H : forall x y, f x y = x
=========================
 ... (f a b) ...
```

rewrite H →

```
H : forall x y, f x y = x
=========================
 ... a ...
```

rewrite

```
H : a = b
=====================
 ... a ...
```

rewrite H →

```
H : a = b
=====================
 ... b ...
```

```
H : forall x y, f x y = x
=========================
 ... (f a b) ...
```

rewrite H →

```
H : forall x y, f x y = x
=========================
 ... a ...
```

Coq guesses how to instantiate the quantifiers

Exercices

Complete the MAP interfaces with the following operators, their specifications and the corresponding proofs.

```
module type MAP =
  ...
  (** [remove m k] returns a new map that contains the same
      bindings as map [m] except for key [k] that is now binded with
      the default element *)
  val remove : t -> key -> t

  (** [map m f] returns a new map that binds each key [k] to an
      element [(f e)], assuming [k] was binded with [e] in [m].
      The function [f] must satisfy the property ... *)
  val map : t -> (elt -> elt) -> t

  (** [mapi m f] returns a new map that binds each key [k] to an
      element [(f k e)], assuming [k] was binded with [e] in [m].
      The function [f] must satisfy the property ... *)
  val mapi : t -> (key -> elt -> elt) -> t
 end
```

Inductive Predicates

Given a positive `p` and a nat `n`, we define the relation `(inf_log p n)` that holds iff `p` holds less then `n` bits.

$$\frac{\texttt{n:nat}}{\texttt{inf\_log xH (S n)}}$$

$$\frac{\texttt{p:positive} \quad \texttt{n:nat} \quad \texttt{inf\_log p n}}{\texttt{inf\_log (xO p) (S n)}}$$

$$\frac{\texttt{p:positive} \quad \texttt{n:nat} \quad \texttt{inf\_log p n}}{\texttt{inf\_log (xI p) (S n)}}$$

```
Inductive inf_log : positive -> nat -> Prop :=
| Inf_log_xH: forall n, inf_log xH (S n)
| Inf_log_xO: forall p n, inf_log p n -> inf_log (xO p) (S n)
| Inf_log_xI: forall p n, inf_log p n -> inf_log (xI p) (S n).
```

inv

on an inductive hypothesis

```
p: positive
n: nat
H : inf_log p n
=====================
P p n
```

inv H

first subgoal

```
n: nat
=====================
P xH (S n)
```

second subgoal

```
n : nat
p : positive
H : inf_log p n
=====================
P (xO p) (S n)
```

third subgoal

```
n : nat
p : positive
H : inf_log p n
=====================
P (xI p) (S n)
```

inv

on an inductive hypothesis

```
p: positive
n: nat
H : inf_log p n
====================
P p n
```

inv H

first subgoal

```
n: nat
====================
P xH (S n)
```

second subgoal

```
n : nat
p : positive
H : inf_log p n
====================
P (xO p) (S n)
```

third subgoal

```
n : nat
p : positive
H : inf_log p n
====================
P (xI p) (S n)
```

$$\frac{\texttt{n:nat}}{\texttt{inf\_log xH (S n)}}$$

$$\frac{\texttt{p:positive} \quad \texttt{n:nat} \quad \texttt{inf\_log p n}}{\texttt{inf\_log (xO p) (S n)}}$$

$$\frac{\texttt{p:positive} \quad \texttt{n:nat} \quad \texttt{inf\_log p n}}{\texttt{inf\_log (xI p) (S n)}}$$

induction

on an inductive hypothesis

```
p: positive
n: nat
H : inf_log p n
====================
P p n
```

induction H

first subgoal

```
n: nat
====================
P xH (S n)
```

second subgoal

```
n : nat
p : positive
H : inf_log p n
IH : P p n
====================
P (xO p) (S n)
```

third subgoal

```
n : nat
p : positive
H : inf_log p n
IH : P p n
====================
P (xI p) (S n)
```

Mixing programs with proofs

The type of (positive) binary numbers with at least n bits.

```
Inductive bin (n:nat) :=
| Build_bin (p:positive) (h:inf_log p n).
```

this is a proof!

Behind the curtain...

In fact every objects we have manipulated so far are just terms in the CIC.

CIC term

CIC term

```
Definition foo := T := def.
```

CIC term

```
Lemma foo_prop : P.
Proof.
  tac...
Qed.
```

this interactive script
builds also a CIC term

Some reading

The Coq Web site: software and documentation.
http://coq.inria.fr/

Coq in a hurry, Yves Bertot.
http://cel.archives-ouvertes.fr/inria-00001173/

Benjamin Pierce et al. Software Foundations.
http://www.cis.upenn.edu/~bcpierce/sf/

Interactive Theorem Proving and Program Development -- Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran: a comprehensive textbook on Coq.

Adam Chlipala, Certified Programming with Dependent Types. (advanced)
http://adam.chlipala.net/cpdt/