COQ : a quick introduction

Thorsten Altenkirch

School of Computer Science
University of Nottingham

April 17, 2008

What is COQ?

- COQ: a Proof Assistant based on the *Calculus of Inductive Constructions*
- Developed in France since 1989.
- Growing user community.
- Big proof developments:
  - Correctness of a C-compiler
  - 4 colour theorem

Why formal proofs?

- Avoid holes in paper proofs.
- Provide additional evidence that the construction is correct.
- Aid understanding.
- Formal certification of programs.

What this course is **not** about:

- The Calculus of Inductive Constructions
- Proof Theory
- λ-calculus
- Type Theory

Metatheory of formal proofs

What this course **is** about:

- Formalizing proofs using COQ
- Developing and verifying programs in COQ
- Formalize mathematics using COQ
- Use dependent types in programs

Using COQ

- Download COQ from `http://coq.inria.fr/`
- Runs under MacOS, Windows, Linux
- coqtop : command line interface
- coqide : graphical user interface
- proof general : emacs interface

For reference

- Coq Reference manual:
  `http://coq.inria.fr/V8.1pl3/refman/`
- Coq Library doc:
  `http://coq.inria.fr/library-eng.html`
- Course page:
  `http://www.cs.nott.ac.uk/~txa/mgs08/`.
- *Coq'Art*, the book by Yves Bertot and Pierre Casteran (2004).

Logic: summary

- Propositional connectives ($P, Q : \mathrm{Prop}$):

$$P \wedge Q, P \to Q, P \vee Q, \mathrm{True}, \mathrm{False}$$

- Defined connectives:

$$\begin{aligned} \sim P &= P \to \mathrm{False} \\ P \leftrightarrow Q &= (P \to Q) \wedge (Q \to P) \end{aligned}$$

- Quantifiers (where $A : \mathrm{Set}$)

$$\mathrm{forall}\ x : A, P \qquad \mathrm{exists}\ x : A, P$$

- Equality ($a, b : A : \mathrm{Set}$)

$$a = b : \mathrm{Prop}$$

Basic tactics

- Use an assumption:
  `assumption`
- Introduce an auxilliary proposition:
  `cut` *prop*

connective	Introduction	Elimination
$P \to Q$	`intro(s)`	`apply` *Hyp*
$P \wedge Q$	`split`	`elim` *Hyp*
True	`split`	
$P \vee Q$	`left,right`	`case` *Hyp*
False		`case` *Hyp*
forall $x : A, P$	`intro(s)`	`apply` *Hyp*
exists $x : A, P$	`exists` *wit*	`elim` *Hyp*
$a = b$	`reflexivity`	`rewrite` *Hyp*

Rules

$$\frac{H : P \in \Gamma}{\Gamma \vdash P}\ \texttt{assumption}$$

$$\frac{\Gamma \vdash P \to Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\ \texttt{cut}\ \mathrm{P}$$

$$\frac{\Gamma, H : P \vdash Q}{\Gamma \vdash P \to Q}\ \texttt{intro}\ \mathrm{H}$$

$$\frac{\begin{array}{l} H : P \to Q \in \Gamma \\ \Gamma \vdash P \end{array}}{\Gamma \vdash Q}\ \texttt{apply}\ \mathrm{H}$$

- The actual behaviour of `apply` is more subtle!

Rules

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \texttt{split} \qquad \frac{\begin{array}{c} H : P \wedge Q \in \Gamma \\ \Gamma \vdash P \to Q \to R \end{array}}{\Gamma \vdash R}\ \texttt{elim}\,\mathrm{H}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q}\ \texttt{left} \quad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q}\ \texttt{right} \quad \frac{\begin{array}{c} H : P \vee Q \in \Gamma \\ \Gamma \vdash P \to R \quad \Gamma \vdash Q \to R \end{array}}{\Gamma \vdash R}\ \texttt{case}\,\mathrm{H}$$

$$\frac{}{\Gamma \vdash \mathrm{True}}\ \texttt{split} \qquad \frac{H : \mathrm{False} \in \Gamma}{\Gamma \vdash R}\ \texttt{case}\,\mathrm{H}$$

Rules

$$\frac{\Gamma, x : D \vdash P(x)}{\Gamma \vdash \forall x : D, P(x)}\ \texttt{intro}\ \mathbf{x}$$

$$\frac{H : \forall x : D, P(x) \in \Gamma \quad \Gamma \vdash d : D}{\Gamma \vdash P(d)}\ \texttt{elim}\ \mathbf{H}$$

$$\frac{\Gamma \vdash d : D \quad \Gamma \vdash P(d)}{\Gamma \vdash \exists x : D, P(x)}\ \texttt{exists}\ \mathbf{d}$$

$$\frac{H : \exists x : D, P(x) \in \Gamma \quad \Gamma \vdash \forall x : D, P(x) \to R}{\Gamma \vdash R}\ \texttt{elim}\ \mathbf{H}$$

$$\frac{\Gamma d : D}{\Gamma \vdash d = d}\ \texttt{reflexivity}$$

$$\frac{H : d = e \in \Gamma \quad \Gamma \vdash P(e)}{\Gamma \vdash P(d)}\ \texttt{rewrite}\ \mathbf{H}$$

- Assumption of the form $d : D$ are checked automatically.

Automatisation

- `auto`
  PROLOG style inference, solves trivial goals
  can be extended (Hint).
- `tauto`
  complete for (intuitionistic) propositional logic.
- `firstorder`
  incomplete for 1st order (intuitionistic) predicate logic.
- `ring`
  solves equations for *rings* and *semirings*

Libraries

- Standard library (automatically loaded)
  basic logical notations and properties
  basic datatypes (e.g. $\mathrm{bool}, \mathrm{nat} : \mathrm{Set}$) and operations $+, *, -$
  and relations $<, \leq$.
- `Require Import Classic`
  introduces classical logic axiomatically.
  $\mathrm{classic} : \mathrm{forall}\ P : \mathrm{Prop}, P \vee \sim P$
- `Require Import Arith`
  algebraic laws, properties of orders,
  decidability of $-, <, \leq$
  enables ring tactic for $\mathrm{nat}, +, *$ (actually a semiring).
- `Require Import List`
  list library, basic functions and properties of lists.

Writing programs

- Define inductive types, predicates and families using `Inductive`.
- Define structurally recursive programs using `Fixpoint`. Mark the argument over which we do recursion using `struct`.
- Use `match` for pattern matching.
- Use the `induction` tactic to prove properties by induction over any inductive type.
- Use the (experimental) `Program` feature to implement programs with dependent types and subsets.

Projects

- Formalize basic category theory.
  - Assume extensionality as an axiom.
  - Show that the categories of sets and functions is cartesian closed.
  - Use records to define an abstract notion of category and define functors, natural transformations,. . .
- Formalize Kleene algebras.
  - Assume the axioms of Kleene algebra.
  - Define test algebras.
  - Use `autorewrite` to simplify the proofs.
- Formalize constructive ordinals.
  - Implement `Omega` like in Haskell.
  - Define addition, multiplication, exponentiation.
  - Define an order and an equality on ordinals.
  - Show basic laws of ordinal arithmetic.