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
- $\lambda$-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
Coq Reference manual:
http://coq.inria.fr/V8.1p13/refman/

Coq Library doc:
http://coq.inria.fr/library-eng.html

Course page:
http://www.cs.nott.ac.uk/~txa/mgs08/.

Logic: summary

- Propositional connectives ($P, Q : \text{Prop}$):
  
  \[
  P \land Q, P \rightarrow Q, P \lor Q, \text{True}, \text{False}
  \]

- Defined connectives:
  
  \[
  \sim P = P \rightarrow \text{False}
  
  P \leftrightarrow Q = (P \rightarrow Q) \land (Q \rightarrow P)
  \]

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

  \[
  \text{for all } x : A, P \quad \text{exists } x : A, P
  \]

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

  \[
  a = b : \text{Prop}
  \]
Basic tactics

- **Use an assumption:**
  assumption

- **Introduce an auxiliary proposition:**
  cut prop

<table>
<thead>
<tr>
<th>connective</th>
<th>Introduction</th>
<th>Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P \rightarrow Q )</td>
<td>intro(s)</td>
<td>apply ( Hyp )</td>
</tr>
<tr>
<td>( P \land Q )</td>
<td>split</td>
<td>elim ( Hyp )</td>
</tr>
<tr>
<td>True</td>
<td>split</td>
<td></td>
</tr>
<tr>
<td>( P \lor Q )</td>
<td>left, right</td>
<td>case ( Hyp )</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td>case ( Hyp )</td>
</tr>
<tr>
<td>forall ( x : A, P )</td>
<td>intro(s)</td>
<td>apply ( Hyp )</td>
</tr>
<tr>
<td>exists ( x : A, P )</td>
<td>exists ( wit )</td>
<td>elim ( Hyp )</td>
</tr>
<tr>
<td>( a = b )</td>
<td>reflexivity</td>
<td>rewrite ( Hyp )</td>
</tr>
</tbody>
</table>
**Rules**

$H : P \in \Gamma \quad $ assumption

$\Gamma \vdash P \quad \Gamma \vdash Q \quad $ cut $P$

$\Gamma, H : P \vdash Q \quad $ intro $H$

$\Gamma \vdash P \rightarrow Q \quad \Gamma \vdash Q \quad $ apply $H$

- The actual behaviour of apply is more subtle!
\[ \begin{array}{l}
\Gamma \vdash P \quad \Gamma \vdash Q \\
\hline
\Gamma \vdash P \land Q \quad \text{split}
\end{array} \]

\[ \begin{array}{l}
H : P \land Q \in \Gamma \\
\Gamma \vdash P \rightarrow Q \rightarrow R \\
\hline
\Gamma \vdash R \quad \text{elim}_H
\end{array} \]

\[ \begin{array}{l}
\Gamma \vdash P \\
\hline
\Gamma \vdash P \lor Q \quad \text{left}
\end{array} \]

\[ \begin{array}{l}
\Gamma \vdash Q \\
\hline
\Gamma \vdash P \lor Q \quad \text{right}
\end{array} \]

\[ \begin{array}{l}
H : P \lor Q \in \Gamma \\
\Gamma \vdash P \rightarrow R \\
\Gamma \vdash Q \rightarrow R \\
\hline
\Gamma \vdash R \quad \text{case}_H
\end{array} \]

\[ \begin{array}{l}
\Gamma \vdash \text{True} \\
\hline
\Gamma \vdash \text{split}
\end{array} \]

\[ \begin{array}{l}
H : \text{False} \in \Gamma \\
\hline
\Gamma \vdash R \quad \text{case}_H
\end{array} \]
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*
- Standard library (automatically loaded)
  basic logical notations and properties
  basic datatypes (e.g. bool, nat : Set) and operations +, *, −
  and relations <, ≤.

- Require Import Classic
  introduces classical logic axiomatically.
  classic : forall P : Prop, P ∨ ∼ P

- Require Import Arith
  algebraic laws, properties of orders,
  decidability of −, <, ≤
  enables ring tactic for nat, +, * (actually a semiring).

- Require Import List
  list library, basic functions and properties of lists.
Define inductive types, predicates and families using \texttt{Inductive}.

Define structurally recursive programs using \texttt{Fixpoint}. Mark the argument over which we do recursion using \texttt{struct}.

Use \texttt{match} for pattern matching.

Use the \texttt{induction} tactic to prove properties by induction over any inductive type.

Use the (experimental) \texttt{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.