# A Short Introduction to Coq

Lecture Interactive Proof Tools
Summer Term 2003

Hans de Nivelle, Patrick Maier



## Coq - Basic Facts and Features

Coq is an interactive proof assistant based on intuitionsitic higher-order logic with inductive types:

- Formulas are types in a typed $\lambda$-calculus (with inductive datatypes).
- A proof $p$ of a formula $F$ is well-formed $\lambda$-term of type $F$.
  $\rightsquigarrow$ Possibly executable proofs.
- For every type $T$ exists a term $t$ of type $T$ ($T$ is *inhabited*).
  $\rightsquigarrow$ Only provable formulas are well-formed.
- Deduction is performed in a natural deduction calculus.

Some non-logical features of Coq:

- Less automated than PVS.
- Open source.



## A note on inhabitation of types:

It is possible to define inductive types which are not inhabited. An example is Coq's inductive type `False` which has no constructors and thus no way to construct inhabitants. Thus, well-formed uninhabited types do exist in Coq. However, if we exclude uninhabited inductive types from the family of well-formed types then all well-formed types must be inhabited. Informally, the reason for this is that basic (non-inductive) type constructor is the dependent product $(x : T)U$, where $x$ is a variable and $T$ and $U$ are types. If $T$ and $U$ are inhabited then $(x : T)U$ is also inhabited. This is easy to see for the special case that type $U$ is a formula. Then $x$ occurs as a free variable (of type $T$) in $U$, and $U$ inhabited means that there is a proof of $U$. Thus by $\forall$-introduction there is a proof of the formula $(x : T)U$, which means that $(x : T)U$ is inhabited.



## Terms, Types and Sorts I

Syntax of *terms*/*types*:

- $c$ declared constant $\rightsquigarrow$ $c$ term/type (depending on declaration).
- $x$ variable $\rightsquigarrow$ $x$ term/type (depending on context).
- $x$ variable, $T$, $U$ types $\rightsquigarrow$ *dependent product* $(x : T)U$ type. If $x$ does not occur in $U$, then write $T \to U$.
- $x$ variable, $T$ type, $U$ term $\rightsquigarrow$ *$\lambda$-abstraction* $[x : T]U$ term.
- $T$ term, $U$ term/type $\rightsquigarrow$ *application* $(T\,U)$ term/type (depending on $T$ and $U$).
- $x$ variable, $T$, $U$ terms/types $\rightsquigarrow$ *let-binding* $[x := T]U$ term/type (depending on $T$ and $U$).

## Terms, Types and Sorts II

Examples of terms and types:

- `0: nat`, `S: nat -> nat` (constants `nat`, `0` and `S` declared in initial library).
- `(S (S 0)): nat`
- `[x: nat] (S (S x)): nat -> nat`
- `[x, y: Prop] [f: x] [g: y] f: (A, B: Prop) A -> B -> A`
- `[x: nat] [y := (S x)] (S y): [T := [A: Set] A -> A] (T nat)`



## Terms, Types and Sorts III

*Sorts* are the types of types.

- `Prop`: sort of types that correspond to formulas. Roughly, `Prop` is the sort of everything that has an (intuitionistic) proof.
- `Set`: sort of types that correspond to (inductively defined) objects. E. g.,, `bool`, `nat`, `nat * nat`, `(list nat)`, ...
- `Type`: super sort of `Prop` and `Set`. Arises by building products of types.

**Note:** Coq treats sorts as if they were types (see examples on the previous slide).



## Proofs in Propositional Logic I

```
Section PropExamples.
  Variables A, B, C: Prop.

  Goal A /\ B -> B /\ A.
    Intro Hyp. Elim Hyp. Intro. Intro. Split.
      Assumption.
      Assumption.
  Save conj_comm.

  Goal A \/ B -> B \/ A.
   Intro Hyp. Elim Hyp.
      Intro. Right. Assumption.
      Intro. Left. Assumption.
  Save disj_comm.
End PropExamples.
```



## Proofs in Propositional Logic II

```
  Goal (A -> B -> C) -> (B -> A -> C).
    Intro Hyp. Intro. Intro. Apply Hyp.
      Assumption.
      Assumption.
  Save imp_left_comm.

  Goal A \/ False -> A /\ True.
    Intro Hyp. Split.
      Elim Hyp.
        Intro. Assumption.
        Intro. Contradiction.
      Exact I.
  Save false_true_neutral.
End PropExamples.
```

## Tactics Used in Propositional Proofs

Tactics corresponding to introduction/elimination rules:

- `Intro`: $\rightarrow$-introduction
- `Split`: $\wedge$-, $\leftrightarrow$-introduction
- `Left`, `Right`: $\vee$-introductions
- `Apply`: $\rightarrow$-elimination
- `Elim`: $\wedge$-, $\vee$-, $\neg$-, $\leftrightarrow$-elimination (variants!)

Tactics for terminating proof branches:

- `Assumption`: conclusion in assumptions
- `Contradiction`: one assumption equivalent to $\bot$
- `Exact`: directly provides proof term



## More Propositional Examples

See exercise 3.1:

1. $(A \vee A) \leftrightarrow (A \vee \bot)$
2. $(B \wedge A) \leftrightarrow ((A \wedge B) \wedge A)$
3. $(A \rightarrow B) \rightarrow \neg(A \wedge \neg B)$
4. $\neg(A \wedge \neg A)$
5. $(\neg\neg\neg A) \rightarrow \neg A$
6. $(A \rightarrow B \rightarrow C) \leftrightarrow (A \wedge B \rightarrow C)$



## A Proof Involving Quantifiers

```
Variable X: Set.
Variable Y: Set.
Variable R: X -> Y -> Prop.

Goal (EX y: Y | (x: X) (R x y)) -> (x: X) (EX y: Y | (R x y)).
  Intros ex_y. Intro x. Elim ex_y. Intro y. Intro R_y.
  Exists y. Apply R_y.
Qed.
```

Tactics used:

- $\forall$-introduction/-elimination: `Intro`, `Apply` (on dep. products)
- $\exists$-elimination: `Elim`
- $\exists$-introduction: `Exists`



## A Proof Involving Equality

```
Variable Nat: Set.
Variable s: Nat -> Nat.
Variable plus: Nat -> Nat -> Nat.
Variables x, y: Nat.

Goal (plus x y) = (plus y x) ->
     ((x, y: Nat) (plus x (s y)) = (s (plus x y))) ->
     ((x, y: Nat) (plus (s x) y) = (s (plus x y))) ->
     (plus x (s y)) = (plus (s y) x).
  Intros. Rewrite H0. Rewrite H1. Rewrite H. Reflexivity.
Qed.
```

Tactics used:

- `Reflexivity`: equality reflexivity axiom
- `Rewrite`: $\forall$-elimination + equality replacement rule

## References


1. Coq Reference Manual, in particular Chapters 4 and 7.
2. Gerard Huet, Gilles Kahn, Christine Paulin-Mohring. The Coq Proof Assistant. A Tutorial.