


# Coq Introduction




张昱

Department of Computer Science and Technology  
University of Science and Technology of China

September, 2008

Yu Zhang, USTC


# Coq Introduction



- ❖ A proof assistant for a logical framework known as the **Calculus of Inductive Constructions**.
- ❖ Allows:
  - to define functions or predicates,
  - to state mathematical theorems and software specifications,
  - to develop interactively formal proofs of these theorems,
  - to check these proofs by a relatively small certification "kernel".
- ❖ Version
  - The current stable version of Coq is the **8.1**. It is available for Unix and Windows 95/98/NT/XP systems.

Yu Zhang, USTC Coq Introduction 2


# Useful Links



- Coq Home page  
<http://coq.inria.fr/>
  - The Coq Proof Assistant - A Tutorial v8.1
  - The Coq Proof Assistant - Reference Manual v8.1
- Coq Art Home page  
<http://www.labri.fr/perso/casteran/CoqArt/index.html>
- Coq Tutorial in POPL08  
<http://www.cis.upenn.edu/~plclub/popl08-tutorial/>

Yu Zhang, USTC Coq Introduction 3


# Coq Tools-1



- ❖ Coq IDE  
<http://coq.inria.fr/distrib-eng.html>
- ❖ Proof General (PG)  
<http://proofgeneral.inf.ed.ac.uk/>
  - a generic front-end for *proof assistants* (also known as *interactive theorem provers*), based on the customizable text editor **Emacs**

Yu Zhang, USTC Coq Introduction 4

# Coq Tools-2





To get started ... (e.g. using PG and for Windows)

- Install **Coq**  
Notice: don't select installing **coqide** and **GDK**
- Install **Emacs** & **PG**
- Modify the environment variables
  - Add
    - **COQBIN** = "(coq install dir)\coq\bin"
    - **COQLIB** = "(coq install dir)\coq\lib"
    - **HOME** = "(where PG is installed)"
  - Append the bin dir of coq and emacs to the value of **Path**
- Type **coqtop.opt** in command line to check ...
- Run **Emacs/bin/runemacs.exe**

Yu Zhang, USTC Coq Introduction 5

# Coq Tools-3

The screenshot shows the Emacs editor window titled 'emacs#GATES'. The menu bar includes 'File Edit Options Buffers Tools Coq Proof-General Help'. The main text area contains a Coq script defining a set of days and an inductive type:

```
(** Let's start with a very simple example. The following
definition tells Coq that we are defining a new set of
data values. The set is called {day} and its members
are {monday}, {tuesday}, etc. The lines of the
definition can be read "{monday} is a {day}, {tuesday}
is a {day}, etc." *)
Inductive day : Set :=
| monday : day
| tuesday : day
| wednesday : day
| thursday : day
| friday : day
| saturday : day
| sunday : day
```

Yu Zhang, USTC Coq Introduction 6



## What does Coq system provide?

- A specification language named **Gallina**
  - consists in a sequence of *declarations* and *definitions*.
  - Its terms can represent **programs** as well as **properties** of these programs and **proofs** of these properties.
  - Using **Curry-Howard isomorphism**, **programs**, **properties** and **proofs** are formalized in the same language called **GIC**, that is a  $\lambda$ -calculus with a rich type system.
  - All **logical judgments** in COQ are **typing judgments**.
- **Type-checker**
  - checks the **correctness of proofs**, in other words that checks that a program complies to its specification.
- **The proof engine**
  - provides an interactive proof assistant to **build proofs** using specific programs called **tactics**.



## Gallina - Declarations

- ❖ **Declarations**
  - A **declaration** associates a **name** with a **specification**.
    - Declared objects play the role of **axioms** or **parameters** in mathematics.
      - Axiom**  $ident : term$
      - Parameter**  $ident_1 \dots ident_n : term$
  - **Specifications**
    - logical propositions: **Prop**
    - mathematical collections: **Set**
    - abstract types: **Type**
  - Every valid expression **e** in Gallina is associated with a **specification**, itself a valid expression, called its **type**  $\tau(E)$ .  $e : \tau(E)$



## Gallina - Inductive Definitions

### ❖ Inductive Definitions

#### ➤ Simple inductive types

**Inductive**  $ident : sort :=$   
|  $ident_1 : type_1$   
| ...  
|  $ident_n : type_n$ .

**ident**: the name of the inductively defined type

**sort**: the universes where it lives

**ident<sub>1</sub>, ..., ident<sub>n</sub>**: the names of its constructors

**type<sub>1</sub>, ..., type<sub>n</sub>**: the types of its constructors



## Gallina - Inductive Definitions

#### ➤ Simple inductive types

```
Inductive yesno : Set :=
| yes : yesno
| no : yesno.
```

defines the following four objects at once:

```
yesno is defined
yesno_rect is defined
yesno_ind is defined
yesno_rec is defined
```

**Check** yesno.

```
yesno
: Set      A new Set is declared, with name yesno.
```



## Gallina - Inductive Definitions

#### ➤ Simple inductive types

```
Inductive yesno : Set :=
| yes : yesno
| no : yesno.
```

defines the following four objects at once:

```
yesno is defined
yesno_rect is defined
yesno_ind is defined
yesno_rec is defined
```

**Check** yesno\_rect.

```
yesno_rect
: forall P : yesno -> Type, P yes -> P no -> forall y : yesno, P y
Destructor (Elimination principle on Type): expresses structural
induction/recursion principle over objects of yesno.
```



## Gallina - Inductive Definitions

#### ➤ Simple inductive types

```
Inductive yesno : Set :=
| yes : yesno
| no : yesno.
```

defines the following four objects at once:

```
yesno is defined
yesno_rect is defined
yesno_ind is defined
yesno_rec is defined
```

**Check** yesno\_ind.

```
yesno_ind
: forall P : yesno -> Prop, P yes -> P no -> forall y : yesno, P y
Destructor (Elimination principle on Prop): expresses structural
induction/recursion principle over objects of yesno.
```



## Gallina - Inductive Definitions

### Simple inductive types

```
Inductive yesno : Set :=
| yes : yesno
| no : yesno.
```

defines the following four objects at once:

```
yesno is defined
yesno_rect is defined
yesno_ind is defined
yesno_rec is defined
Check yesno_rec.
```

```
yesno_rec
: forall P : yesno -> Set, P yes -> P no -> forall y : yesno, P y
```

Destructor (Elimination principle on Set): expresses structural induction/recursion principle over objects of `yesno`.



## Gallina - Inductive Definitions

### Simple inductive types

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.
```

defines the following four objects at once:

```
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
Check nat_ind.
```

```
nat_ind
: forall P : nat -> Prop,
P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Destructor (Elimination principle on Prop): expresses structural induction/recursion principle over objects of `nat`.



## Gallina - Inductive Definitions

### Simple annotated inductive types

```
Inductive evenP : nat -> Prop :=
| even_0 : evenP 0
| even_SS : forall n:nat, evenP n -> evenP (S (S n)).
```

defines the following two objects at once:

```
evenP is defined
evenP_ind is defined
Check evenP_ind.
```

```
evenP_ind
: forall P : nat -> Prop,
P 0 ->
(forall n : nat, evenP n -> P n -> P (S (S n))) ->
forall n : nat, evenP n -> P n
```



## Gallina - Inductive Definitions

### Simple annotated inductive types

```
Inductive eventT : nat -> Type :=
| eventT_0 : eventT 0
| eventT_SS : forall n:nat, eventT n -> eventT (S (S n)).
```

defines the following four objects at once:

```
eventT is defined
eventT_rect is defined
eventT_ind is defined
eventT_rec is defined
Check eventT_ind.
```

```
eventT_ind
: forall P : forall n : nat, eventT n -> Prop,
P 0 eventT_0 ->
(forall (n : nat) (e : eventT n), P n e -> P (S (S n)) (eventT_SS n e)) ->
forall (n : nat) (e : eventT n), P n e
```



## Gallina - Inductive Definitions

### Mutually defined inductive types

```
Inductive evenM : nat -> Prop :=
| evenM_0 : evenM 0
| evenM_S : forall n, oddM n -> evenM (S n)
with oddM : nat -> Prop :=
| oddM_S : forall n, evenM n -> oddM (S n).
```

defines the following four objects at once:

```
evenM, oddM are defined
evenM_ind is defined
oddM_ind is defined
Check evenM_ind.
```

```
evenM_ind
: forall P : nat -> Prop,
P 0 ->
(forall n : nat, oddM n -> P (S n)) -> forall n : nat, evenM n -> P n
```



## Gallina - Inductive Definitions

### Mutually defined inductive types

```
Inductive evenM : nat -> Prop :=
| evenM_0 : evenM 0
| evenM_S : forall n, oddM n -> evenM (S n)
with oddM : nat -> Prop :=
| oddM_S : forall n, evenM n -> oddM (S n).
```

defines the following four objects at once:

```
evenM, oddM are defined
evenM_ind is defined
oddM_ind is defined
Check oddM_ind.
```

```
oddM_ind
: forall P : nat -> Prop,
(forall n : nat, evenM n -> P (S n)) -> forall n : nat, oddM n -> P n
```



## Gallina - Definitions

### ❖ Definitions

- A definition gives a name to a term (definition).

Definition  $ident[(ident_1 : term_1) \dots (ident_n : term_n)] : term_0 := term$

```
Definition swap_yesno (b:yesno) : yesno :=
  match b with
  | yes => no
  | no => yes
  end.

Check swap_yesno.
swap_yesno
  : yesno -> yesno
```



## Gallina - Definitions

### ❖ Definitions

```
Definition both_yes (b1:yesno) (b2:yesno) : yesno :=
  match b1 with
  | yes => b2
  | no => no
  end.
```

Check both\_yes.

```
both_yes
  : yesno -> yesno -> yesno
```



## Gallina - Definition of recursive functions

### ❖ Definition of recursive functions

Fixpoint  $ident\ params\{\text{struct } ident_0\} : type_0 := term_0$

```
Fixpoint plus (m : nat) (n : nat) (struct m) : nat :=
  match m with
  | 0 => n
  | S m' => S (plus m' n)
  end.
```

plus is recursively defined[]

Check plus.

```
plus
  : nat -> nat -> nat[]
```



## Gallina - Definition of recursive functions

### ❖ Definition of recursive functions

Fixpoint  $ident\ params\{\text{struct } ident_0\} : type_0 := term_0$

```
Fixpoint even (n:nat) (struct n) : yesno :=
  match n with
  | 0      => yes
  | S 0   => no
  | S (S n') => even n'
  end.
```

Check even.[]

```
even
  : nat -> yesno[]
```

See [bnat\\_1.v](#) and make exercises.



## Gallina - Statement and proofs

### ❖ Statement

- A statement claims a goal of which the proof is then interactively done using tactics.

Theorem  $ident : type.$       Lemma  $ident : type.$

Proposition  $ident : type.$       Fact  $ident : type.$

Definition  $ident : type.$

- After a statement, Coq needs a proof.

Proof. ... Qed.

A proof starts by the keyword **Proof**. Then Coq enters the proof editing mode until the proof is completed.

Proof. ... Admitted.

Turns the current conjecture into an axiom and exits editing of current proof.



## Proof engine - Common Commands

### ❖ Displaying [Coq RM 6]

- **Print *qualid*.**    e.g. **Print even.**
- **Print All.**

### ❖ Requests to the environment [Coq RM 6]

- **Check *term*.**    displays the type of term.

### ❖ Compiled files [Coq RM 6]

- **Require *dirpath*.** , **Require Export *qualid*.**
- .....



## Proof engine - Atomic Tactics - 1

See Chapter 8 in Coq Reference Manual for detail.

- ❖ **Explicit proof as a term [Coq RM 8.2]**
  - **exact term.** gives directly the exact proof term of the goal.
- ❖ **Basics [Coq RM 8.3]**
  - **intro [ident]** introduce the premise of the goal.
  - **intros [ident<sub>1</sub> ... ident<sub>n</sub>].** introduce all premises of the goal.
  - **apply [term].** tries to match the current goal against the conclusion of the type of term.
    - E.g. goal:  $Q \rightarrow (apply\ P \rightarrow Q) \rightarrow Q$  goal:  $P$
    - apply term in term<sub>p</sub>.**
    - E.g. goal:  $Q \rightarrow x : P \rightarrow (apply\ P \rightarrow Q\ in\ x) \rightarrow Q$  goal:  $Q \rightarrow x : Q$
  - **assumption.** It looks in the local context for a hypothesis which type is equal to the goal.

Yu Zhang, USTC

25



## Proof engine - Atomic Tactics - 2

- ❖ **Basics [Coq RM 8.3]**
  - **split.** splits conjunction  $A \wedge B$  into  $A$  and  $B$ .
  - **left, right.** applies upon the left/right of disjunction  $A \vee B$ , and then they are respectively equivalent to  $A$  and  $B$ . See [prop.v](#) for detail and make excises.
- ❖ **Conversion tactics [Coq RM 8.5]**
  - **simpl.** applies  $\beta\eta$ -reduction rule.

**RULE 8 (BETA REDUCTION)**

The following transformation is called  $\beta$ -reduction:

$$((\lambda x.M)N) \xrightarrow{\beta} M[x \leftarrow N]$$

**RULE 8 (ETA REDUCTION)**

The following transformation of a lambda expression is called  $\eta$ -reduction.

$$(\lambda x.M(x)) \xrightarrow{\eta} M,$$

where  $x$  may not be a free variable in  $M$ .

Yu Zhang, USTC

Coq Introduction

26



## Proof engine - Atomic Tactics - 3

- ❖ **Conversion tactics [Coq RM 8.5]**
  - **unfold qualid.** qualid must denote a defined transparent constant or local definition. 展开目标中出现的qualid, 在替换时执行 $\beta\eta$ 归约
  - **red.** applies to a goal which has the form  $\text{forall } (x:T_1) \dots (x_k:T_k), c\ t_1 \dots t_n$  where  $c$  is a constant. If  $c$  is transparent then it replaces  $c$  with its definition (say  $t$ ) and then reduces  $(t\ t_1 \dots t_n)$  according to  $\beta\eta$ -reduction rules.
  - **fold term.** term is reduced using the red tactic.

Yu Zhang, USTC

Coq Introduction

27



## Proof engine - Atomic Tactics - 4

- ❖ **Introductions [Coq RM 8.6]**
  - **constructor num.** applies to a goal such that the head of its conclusion is an inductive constant (say  $I$ ). The argument  $num$  must be less or equal to the numbers of constructor(s) of  $I$ .

Let  $ci$  be the  $i$ -th constructor of  $I$ , then

**constructor i**  
is equivalent to  
**intros; apply ci.**

Yu Zhang, USTC

Coq Introduction

28



## Proof engine - Atomic Tactics - 5

- ❖ **Equality [Coq RM 8.8]**
  - **reflexivity.** applies to a goal which has the form  $t = u$ . It checks that  $t$  and  $u$  are convertible and then solve the goal.
  - **symmetry.** applies to a goal which has the form  $t = u$  and changes it into  $u = t$ .
  - **rewrite term.** is equivalent to **rewrite -> term.** the type of **term** must have the form  $\text{term1} = \text{term2}$ . the tactic replaces every **term1** by **term2** in the goal.
  - **rewrite <- term.** uses  $\text{term1} = \text{term2}$  from right to left.

Yu Zhang, USTC

Coq Introduction

29



## Proof engine - Atomic Tactics - 6

- ❖ **Elimination [Coq RM 8.7]**
  - **induction term.** the type of **term** must be an inductive constant. Then, the tactic generates subgoals, one for each possible form of **term**.
  - **destruct term.** its behavior is similar to induction except that no induction hypothesis is generated. 不产生归纳项
  - **decompose [qualid<sub>1</sub> ... qualid<sub>n</sub>] term.** recursively decompose a complex proposition in order to obtain atomic ones. e.g. **decompose [ and or ] H.**

Yu Zhang, USTC

Coq Introduction

30



## Proof engine - Atomic Tactics - 7

### ❖ Elimination [Coq RM 8.7]

- `elim term`. more basic `induction` tactic. 根据目标的类型选择合适的destructor并应用之; 它不影响目标的假设, 也不会引入归纳假设.



## Proof engine - Atomic Tactics - 8

### ❖ Inversion [Coq RM 8.10]

- `inversion ident`. let the type `ident` in the local context be  $(I \ t)$ , where  $I$  is an inductive predicate. The tactic derives for each possible constructor `ci` of  $(I \ t)$ , all the necessary conditions that should hold for the instance  $(I \ t)$  to be proved by `ci`.

### ❖ Contradictory

- `intros contra`. introduce contradictory assumptions  
introduce `true = false`



## Proof engine -Tactic macros

### ❖ Ltac

- Ltac `Solve := simpl; intros; auto`.
- `(util.v)`  
Ltac `move_to_top x' :=`  
  `match reverse goal with`  
  `| H : _ |- _ => try move x' after H`  
  `end.`  
Ltac `Case s' := let c' := fresh "case" in set (c' := s');`  
  `move_to_top c'.`



## User extensions - Syntax extensions

### ❖ Notations[Coq RM 11.1]

- Notation "`A /\ B`" := (and A B).
- Notation "`A /\ B`" := (and A B) (at level 80, right associativity).
- Notation "`( x , y )`" := (@pair \_\_ x y) (at level 0).
- Notation "`A /\ B`" := (and A B) : type\_scope.



## Tools - Coq commands

### ❖ Coq commands [Coq RM 12]

- Interactive use: `coqtop.opt`  
  `coqtop.opt -help` show the help of usage  
  If we have an `util.v` to be compiled, we can execute  
  `coqtop.opt -compile util`  
  then `util.vo` is generated.
- Batch compilation: `coqc.opt`  
  `coqc.opt util.v`



Thanks!