

Logic

Pierre Castéran, Géraud Sénizergues

September-December 2011

This course contains two main (independent) parts :

- ▶ A : Introduction to proof assistants, using the Coq system.
 - ▶ This part shows some implementation principles of the Coq proof assistant, based on typed lambda-calculus. This tool is mainly used for proving program correctness and/or representing mathematical concepts and proving theorems about these concepts.
 - ▶ Slides from summer schools held in Beijing, Suzhou (China) and Paris (France) by Yves Bertot, Pierre Castéran, Assia Mabhoubi and Pierre Letouzey will be available. Lab sessions will be organized on computer.

This course contains two main (independent) parts :

- ▶ A : Introduction to proof assistants, using the Coq system.
 - ▶ This part shows some implementation principles of the Coq proof assistant, based on typed lambda-calculus. This tool is mainly used for proving program correctness and/or representing mathematical concepts and proving theorems about these concepts.
 - ▶ Slides from summer schools held in Beijing, Suzhou (China) and Paris (France) by Yves Bertot, Pierre Castéran, Assia Mabhoubi and Pierre Letouzey will be available. Lab sessions will be organized on computer.
- ▶ B : Proof theory, by Gérard Sénizergues.

This course contains two main (independent) parts :

- ▶ A : Introduction to proof assistants, using the Coq system.
 - ▶ This part shows some implementation principles of the Coq proof assistant, based on typed lambda-calculus. This tool is mainly used for proving program correctness and/or representing mathematical concepts and proving theorems about these concepts.
 - ▶ Slides from summer schools held in Beijing, Suzhou (China) and Paris (France) by Yves Bertot, Pierre Castéran, Assia Mabhoubi and Pierre Letouzey will be available. Lab sessions will be organized on computer.
- ▶ B : Proof theory, by Gérard Sénizergues.

This year, part A will be given during the first 6 weeks of the semester, part B will take place during the last 6 weeks.

- ▶ What is Coq ?
 - ▶ A programming language
 - ▶ A proof development tool
 - ▶ This tool is based on a typed λ -calculus called *The Calculus of Inductive Constructions*.
- ▶ Why do we use Coq ?
 - ▶ To develop software with few errors
 - ▶ To use the computer to verify that all details are right
- ▶ How does one use Coq ?
 - ▶ Describe four components : the data, the operations, the properties, the proofs

Documentation

- ▶ This course's page
www.labri.fr/perso/casteran/FM/Logique
- ▶ Book by Bertot and Castéran (French version) available on this page
- ▶ English and Chinese version are under copyright.
- ▶ Coq's official site : coq.inria.fr

A commented example on sorting : the data

```
(* this is a comment. *)
```

```
Inductive list (A : Type) : Type :=  
  nil | cons (a : A) (l : list A).
```

```
Implicit Arguments nil [A].
```

```
Implicit Arguments cons [A].
```

```
Notation "a :: l" := (cons a l).
```

The operations

```
Fixpoint insert (x : Z) (l : List Z) :=
  match l with
  | nil => x::nil
  | a::l' =>
    if Zle_bool x a then x::a::l' else a::insert x l'
  end.
```

```
Fixpoint sort l :=
  match l with
  | nil => nil
  | a::l' => insert a (sort l')
  end.
```


The properties

- ▶ Have a property `sorted` to express that a list is sorted
- ▶ Have a property `permutation l1 l2`

```
Definition permutation l1 l2 :=  
  forall x, count x l1 = count x l2.
```

- ▶ assuming the existence of a function `count`

Proving properties

Two categories of statements :

- ▶ General theory about the properties (statements that do not mention the algorithm being proved) :

$$\forall x \ y \ l, \text{ sorted } (x::y::l) \Rightarrow x \leq y$$

transitive permutation

Proving properties

Two categories of statements :

- ▶ General theory about the properties (statements that do not mention the algorithm being proved) :

$$\forall x y l, \text{sorted } (x::y::l) \Rightarrow x \leq y$$

transitive permutation

- ▶ Specific theory about the algorithms `insert` and `sort` :

$$\forall x l, \text{sorted } l \Rightarrow \text{sorted}(\text{insert } x l)$$

$$\forall x l, \text{permutation } (x::l) (\text{insert } x l)$$

$$\forall l, \text{let } l' := \text{sort } l \text{ in} \\ \text{permutation } l l' \wedge \text{sorted } l'$$

First steps in Coq

Write a comment “open parenthesis-star”, “star-close parenthesis”

```
(* This is a comment *)
```

Give a name to an expression

```
Definition three := 3.
```

```
three is defined
```

Verify that an expression is well-formed

```
Check three.
```

```
three : nat
```

Compute a value

```
Compute three.
```

```
= 3 : nat
```

Defining functions

Expressions that depend on a variable

Definition add3 (x : nat) := x + 3.

add3 is defined

The type of values

The command **Check** is used to verify that an expression is well-formed

- ▶ It returns the **type** of this expression
- ▶ The type says in which context the expression can be used

Check $2 + 3$.

$2 + 3 : nat$

Check 2 .

$2 : nat$

Check $(2 + 3) + 3$.

$(2 + 3) + 3 : nat$

The type of functions

The value `add3` is not a natural number

Check `add3`.

```
add3 : nat -> nat
```

The value `add3` is a **function**

- ▶ It expects a natural number as **input**
- ▶ It **outputs** a natural number

Check `add3 + 3`.

```
Error the term "add3" has type "nat -> nat"
while it is expected to have type "nat"
```

Applying functions

Function application is written only by juxtaposition

- ▶ Parentheses are not mandatory

Check `add3 2`.

```
add3 2 : nat
```

Compute `add3 2`.

```
= 5 : nat
```

Check `add3 (add3 2)`.

```
add3 (add3 2) : nat
```

Compute `add3 (add3 2)`.

```
= 8 : nat
```


Functions with several arguments

At definition time, just use several variables

Definition s3 (x y z : nat) := x + y + z.

s3 is defined

Check s3.

s3 : nat -> nat -> nat -> nat

Functions with one argument that return functions.

Check s3 2.

s3 2 : nat -> nat -> nat

Check s3 2 1.

s3 2 1 : nat -> nat

Functions are values

- ▶ The value `add3 2` is a natural number,
- ▶ The value `s3 2` is a function,
- ▶ The value `s3 2 1` is a function, like `add3`

Function arguments

- ▶ Functions can also expect functions as argument

```
Definition rep2 (f : nat -> nat)(x:nat) := f (f x).
```

rep2 is defined

```
Check rep2.
```

```
rep2 : (nat -> nat) -> nat -> nat
```

```
Definition rep2on3 (f : nat -> nat) := rep2 f 3.
```

```
Check rep2on3.
```

```
rep2on3 : (nat -> nat) -> nat
```

Type verification strategy (function application)

Function application is well-formed if types match :

- ▶ Assume a function f has type $A \rightarrow B$
- ▶ Assume a value a has type A
- ▶ then the expression $f a$ is well-formed and has type B

Check rep2on3. *rep2on3 : (nat -> nat) -> nat*

Check add3. *add3 : nat -> nat*

Check rep2 add3. *rep2on3 add3 : nat*

Anonymous functions

Functions can be built without a name

Construct well-formed expressions containing a variable, with a header

Check `fun (x : nat) => x + 3`.

```
fun x : nat => x + 3 : nat -> nat
```

The new expression is a function, usable like `add3` or `s3 2 1`

Check `rep2on3 (fun (x : nat) => x + 3)`.

```
rep2on3 (fun x : nat => x + 3) : nat
```

This is called an **abstraction**

Type verification strategy (abstraction)

An anonymous function is well-formed if the body is well formed

- ▶ add the assumption that the variable has the input type
- ▶ add the argument type in the result
- ▶ Example, `verify : fun x : nat => x + 3`
- ▶ `x + 3` is well-formed when `x` has type `nat`, and has type `nat`
- ▶ Result : `fun x : nat => x + 3` has type `nat -> nat`

A few datatypes

- ▶ An introduction to some of the pre-defined parts of Coq
- ▶ Grouping objects together : tuples
- ▶ Natural numbers and the basic operations
- ▶ Boolean values and the basic tests on numbers

Putting data together

- ▶ Grouping several pieces of data : tuples,
- ▶ fetching individual components : pattern-matching,

Check (3,4).

```
(3, 4) : nat * nat
```

Check

```
fun v : nat * nat =>  
  match v with (x, y) => x + y end.  
fun v : nat * nat => let (x, y) := v in x + y  
  : nat * nat -> nat
```


Numbers

As in programming languages, several types to represent numbers

- ▶ natural numbers (non-negative), relative integers, more efficient representations
- ▶ Need to load the corresponding libraries
- ▶ Same notations for several types of numbers : need to choose a scope
- ▶ By default : natural numbers
 - ▶ Good properties to learn about proofs
 - ▶ Not adapted for efficient computation

Focus on natural numbers

```
Require Import Arith.
```

```
Open Scope nat_scope.
```

```
Check 3.
```

```
3 : nat
```

```
Check S.
```

```
S : nat -> nat
```

```
Check S 3.
```

```
4 : nat
```

```
Check 3 * 3.
```

```
3 * 3 : nat
```

Boolean values

- ▶ Values true and false
- ▶ Usable in `if .. then .. else ..` statements
- ▶ comparison function provided for numbers
- ▶ To find them : use the command Search