

# Logic, Part II: Computer-Assisted Proofs

Pierre Castéran

October-December 2014

Slides written by Yves Bertot, Sandrine Blazy, Pierre Castéran, Pierre Letouzey, and Assia Mabhoubi for Summer Schools in Paris, Beijing, Suzhou, Shanghai and Shonan Village Center (Kanagawa).

## About Coq

*Awarded to an institution or individual(s) recognized for developing a software system that has had a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both. The Software System Award carries a prize of \$35,000. Financial support for the Software System Award is provided by IBM.*

[http://awards.acm.org/software\\_system/](http://awards.acm.org/software_system/)

Coq is a software tool for the [interactive development of formal proofs, which is a key enabling technology for certified software](#). It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. An open source product, Coq has played an influential role in formal methods, programming languages, program verification and formal mathematics. As certification gains importance in academic and industrial arenas, Coq plays a critical role as a primary programming and certification tool. Coq's first implementation was in 1985 ...

## Documentation

- ▶ <http://www.labri.fr/perso/casteran/FM/LogiqueM1>  
This course's page
- ▶ <http://coq.inria.fr>  
Coq web site (look at documentation index)
- ▶ <http://www.labri.fr/perso/casteran/CoqArt>  
Book by Yves Bertot and Pierre Castéran (french version freely available)
- ▶ <http://www.cis.upenn.edu/~bcpierce/sf>  
Software Foundations, by Benjamin Pierce.
- ▶ Interactive Theorem Proving and Program Development  
Coq'Art, The Calculus of Inductive Constructions  
Yves Bertot and Pierre Castéran, Springer, 2004.

- ▶ What is Coq ?
  - ▶ A programming language
  - ▶ A proof development tool
- ▶ Why do we use Coq ?
  - ▶ To develop software with few errors
  - ▶ To explore mathematical theories and/or proof theory
  - ▶ To use the computer to verify that all details are right
- ▶ How does one use Coq ?
  - ▶ The topic of this course.

## Describing the data

- ▶ Case-based
  - ▶ show all possible cases for the data
  - ▶ a finite number of different cases
- ▶ Structured
  - ▶ each case has all the components needed in the data
  - ▶ like a record
- ▶ Sometimes recursive
  - ▶ use the “divide-and-conquer” approach
  - ▶ recognize repetition to tame infinite datatypes
- ▶ Theoretical foundation : algebraic datatypes, term algebras, cartesian products, disjoint sums, least and greatest fixed points

## Describing the operations

- ▶ Functional programming : each operation is described as a function
- ▶ Map inputs to outputs, do not modify
- ▶ Programming guided by the cases from data-types
- ▶ Avoid undefined values
  - ▶ all cases must be covered
  - ▶ guaranteed termination of computations
- ▶ safer programming

## Describing the properties

- ▶ A predefined language of higher-order logic **and, or, forall, exists**.
- ▶ Possibility to express consistency between several functions
  - ▶ example *whenever  $f(x)$  is true,  $g(x)$  is a prime number*
- ▶ A general scheme to define new predicates : inductive predicates
  - ▶ example *the set of even numbers is the least set  $E$  so that  $0 \in E$  and  $x \in E \Rightarrow x + 2 \in E$*
  - ▶ foundation : least fixed points



## Proving properties of programs

- ▶ Decompose a logical formula into simpler ones
- ▶ Goal oriented approach, backward reasoning
- ▶ Consider a goal  $P(a)$
- ▶ Suppose there is a theorem  $\forall x, Q(x) \wedge R(x) \Rightarrow P(x)$
- ▶ By choosing to apply this theorem, get two new goals :  $Q(a)$  and  $R(a)$
- ▶ The system makes sure no condition is overlooked
- ▶ A collection for tools specialized for a variety of situations
- ▶ Handle equalities (rewriting), induction, numeric computation, function definitions, etc...

## A commented example on sorting : the data

```
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 l` to express that a list `l` 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 the 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 properties being proved
  - ▶  $\forall x l, \text{sorted } l \Rightarrow \text{sorted } (\text{insert } x l)$
  - ▶  $\forall x l, \text{permutation } (x: l) (\text{insert } x l)$

## 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

```
Eval compute in 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
```

Eval `compute` in `add3 2`.

```
= 5 : nat
```

Check `add3 (add3 2)`.

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

Eval `compute` in `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`