

Coq: What, Why, How?

Who: Yves Bertot

When: July 2012

- ▶ What is Coq ?
 - ▶ A programming language
 - ▶ A proof development tool
- ▶ Why do we use Coq ?
 - ▶ To develop software without errors (CompCert)
 - ▶ To develop mathematical proofs (Four Colors Theorem)
 - ▶ 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
 - ▶ The topic of this week-long course.

Describing the data

- ▶ Case-based
 - ▶ show all possible cases for the data
 - ▶ a finite number of different cases (`bool`, `disjoint sum`)
- ▶ Structured
 - ▶ each case has all the components needed in the data (`product`)
- ▶ Sometimes recursive
 - ▶ recognize repetition to tame infinite datatypes (`list`)
- ▶ 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
 - ▶ Computation must be guaranteed to terminate
- ▶ safer programming

Describing the properties

- ▶ A predefined language for 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

- ▶ First state an objective, then use logical steps to make it simpler
 - ▶ 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 listZ : Type :=  
  nilZ | consZ (hd : Z) (tl : listZ).
```

```
Notation "hd :: tl" := (consZ hd tl).
```

The operations

```
Fixpoint insert (x : Z) (l : listZ) :=  
  match l with  
  | nilZ => x::nilZ  
  | hd::tl =>  
    if Zle_bool x hd then x::l else hd::insert x tl  
end.
```

```
Fixpoint sort l :=  
  match l with  
  | nilZ => nilZ  
  | hd::tl => insert hd (sort tl)  
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 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

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 3.

$3 : nat$

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

Function with one argument that return a function.

Check s3 2.

s3 2 : nat -> nat -> nat

Check s3 2 1.

s3 2 1 : nat -> 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`

This is called an **abstraction**

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

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`

Compute `s3 2 1`.

`= fun z : nat => S (S (S z)) : nat -> nat`

Function arguments

- Functions can also expect functions as argument (higher order)

```
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
```

Programming by cases

- ▶ Datatypes are described by giving disjoint cases
- ▶ A single case can group several components
- ▶ When defining a function, all cases must be covered
pattern-matching construct
 - ▶ `match ... with`
 `Case1 => ... | Case2 a b => ... | ...`
 `end`
- ▶ For example the data-type of lists
 - ▶ Two cases : empty list (called `nil`) or object with two components (called `cons`, notation `a :: l'`)

Example datatype : lists

Print list.

```
Inductive list (A : Type) : Type :=  
  nil : list A | cons : A -> list A -> list A
```

(* Example function that adds the first two elements of a list *)

```
Definition addf2 (l : list nat) : nat :=  
  match l with  
  | nil => 0  
  | a::nil => a  
  | a::b::l' => a + b  
  end.
```

Recursion on lists

Three kinds of recursive functions in Coq, all restricted
We have time to see only the first kind

- ▶ Keyword : `Fixpoint`
- ▶ Recursive call must occur on sub-lists
- ▶ Easiest way to recognize sublists : variables from pattern-matching
- ▶ Already seen with the `sorting` example
- ▶ Avoid non-terminating computation

Example recursion

```
Fixpoint dl (l : list nat) (n : nat) : list nat :=  
  match l with  
  | nil => nil  
  | a::l' => n * a :: dl l' (n + 1)  
end.
```


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
```

Recursion on natural numbers

- ▶ natural numbers are either 0 or the successor of another
- ▶ Two cases for programming, recursion on the only component

Print nat.

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

```
Fixpoint mult2 (n : nat) : nat :=  
  match n with 0 => 0 | S p => S (S (mult2 p)) end.
```

Compute mult2 3.

```
= 6 : 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 bool`
- ▶ Or `Search (nat -> nat -> bool)`