

Coq Summer School

Yves Bertot

Introduction

- ▶ Welcome!
- ▶ Coq from the practical side
 - ▶ But Theory has practical benefits, too.
- ▶ Start from what we expect you know: programming
 - ▶ Need to learn a new programming language!
- ▶ Learn to state assertions about programs
 - ▶ Need to learn a logical language
- ▶ Verify that the assertions do hold
- ▶ Need to learn how to prove statements

Week plan

- ▶ Today: basics about simple computations on numbers
- ▶ Tuesday: logical formulas and basic proofs
- ▶ Wednesday: more data-structures, starting with lists

All facets addressed, but small expressive power

- ▶ Thursday: Inductive predicates and dependent types
- ▶ Friday: dependent types in programming and recursion

Speakers and advisors

Speakers

- ▶ Assia Mahboubi: INRIA Researcher
 - ▶ mathematical proofs and proof automation
- ▶ Pierre Castéran: University Lecturer
 - ▶ Co-author of Coq'Art, formal methods
- ▶ Pierre Letouzey: University Lecturer
 - ▶ Derivation of programs from proofs, libraries
- ▶ Yves Bertot: INRIA Researcher
 - ▶ Co-author of Coq'Art, programming languages, geometry

Advisors for afternoon sessions

- ▶ Stéphane Glondu: PhD student
 - ▶ Derivation of programs from proofs
- ▶ Francesco Zappa Nardelli: INRIA Researcher
 - ▶ Programming languages

Interacting with Coq

There is no command called coq

- ▶ A command line interpreter : `coqtop`
 - ▶ Commands to define, evaluate expressions, or query the internal database
 - ▶ Outputs can be small data or complete listings
- ▶ User-interface support
 - ▶ Have a window where commands from the user are stored
 - ▶ Have one or two windows to display results of commands
 - ▶ Show the state by coloring commands (become read-only)
- ▶ User-interfaces: `coqide`, `Emacs/Proof-general`, `proofweb`
- ▶ A batch compiler `coqc`
 - ▶ Converts source files (suffixe `.v`) into pre-compiled files (`.vo`).

Expressions in Coq

- ▶ Programming in Coq: giving names to expressions
- ▶ Analogy in programming in C or Java
 - ▶ left-hand sides of assignments
 - ▶ arguments to procedure or method calls
- ▶ A command to verify if an expression is well-formed `Check`
 - ▶ `Check 3.`
`3 : nat`
 - ▶ `Check 3 + 5.`
`3 + 5 : nat`
 - ▶ `Check true.`
`true : bool`
 - ▶ `Check 3 + true.`
`Error: The term "true" has type "bool"`
`while it is expected to have type "nat".`

Finding functions

- ▶ Find functions by using Search.

- ▶ the argument is the name of the returned type.

- ▶ `Search nat.`

- `0: nat`

- `S: nat -> nat`

- `pred: nat -> nat`

- `plus: nat -> nat -> nat`

- `mult: nat -> nat -> nat`

- `minus: nat -> nat -> nat`

- ▶ Several arrows when the function has several arguments

- ▶ Functions with several arguments can be used with only one

- ▶ Implicit parentheses on the right

- `nat -> nat -> nat \equiv nat -> (nat -> nat)`

Using functions

- ▶ write the function on the left of the argument
- ▶ use parentheses only when necessary to avoid ambiguity
 - ▶ Check `plus 3`.
`plus 3 : nat -> nat`
 - ▶ Check `plus 3 (plus 4 5)`.
`3 + (4 + 5) : nat`
 - ▶ Implicit parentheses on the left
`plus 3 5` \equiv `(plus 3) 5`

Constructing functions

- ▶ The function that maps x to e written

`fun x => e`

- ▶ Examples

- ▶ Check `fun x => x + 3.`

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

- ▶ Check `(fun x => x + 3) 5.`

`(fun x : nat => x + 3) 5 : nat`

- ▶ Functions are values, like anything else.

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

`fun x : nat -> nat => x (x (x 3)) : nat -> nat`

Defining values and functions

- ▶ Keywords `Definition` and `:=`
- ▶ Give a name to a value, the value may be a function.
 - ▶ `Definition a_big_number := ((123 * 1000) + 456) * 1000 + 789.`
 - ▶ `Definition iter3on3 := fun x => x (x (x 3)).`
- ▶ Alternative syntax for functions
 - ▶ `Definition iter3on3 f := f (f (f 3)).`
`Definition iter3on3 (f : nat -> nat) :=`
 `f (f (f 3)).`

Local definitions

- ▶ Define intermediate results
- ▶ Forget after returning the main result
- ▶ Use a local name for some expression
- ▶ notation : `let x := ... in ...`
- ▶ Example

Check `let x := 3 in x * (x + x).`

Check `let x := 3 in x * (x + x) : nat`

Evaluating expressions

- ▶ Symbolic evaluation
 - ▶ `Eval vm_compute in iter3on3 (plus 3).`
`= 12 : nat`
 - ▶ `vm_compute` can be replaced with `lazy` and other reduction strategies
- ▶ Beware that Coq is only a symbolic evaluation engine, efficiency not guaranteed
- ▶ Other approach: derive an Ocaml program and compile it!
 - ▶ See Extraction
- ▶ Motto: *write your program in Coq, perform small tests (when possible) and proofs, then extract and obtain high-guarantee software*

Notations

- ▶ Nicer syntax for frequent constructs
- ▶ Same notation for different concepts
 - ▶ $A * B$: cartesian product, natural number multiplication, or integer multiplication
 - ▶ 5 : natural number $S (S (S (S (S 0))))$ or integer $Zpos (xI (x0 xH))$
 - ▶ `Check S (S (S 0)).`
`3 : nat`
- ▶ What is behind a notation : `Locate`.
 - ▶ `Locate "_ * _".`

Notation	Scope
<code>"x * y"</code>	<code>:= prod x y : type_scope</code>
<code>"n * m"</code>	<code>:= mult n m : nat_scope</code> (default interpretation)
 - ▶ `Locate "*".`

Predefined boolean type

- ▶ boolean value : true and false
- ▶ control structure : if ... then ... else ...
- ▶ functions andb, orb, negb
- ▶ Extra functions when loading the package Bool.
 - ▶ `Require Import Bool.`
 - ▶ Infix notations `&&`, `andb`, `||`, `orb`
- ▶ Find functions using the Search command.
- ▶ Beware: intuitive notations often not boolean
- ▶ Shows a distinction between *programming* and *logical reasoning*
 - ▶ `Check fun x y:nat => if x <= y then 0 else 1.`
Error: The term "x <= y" has type "Prop"
which is not a (co-)inductive type.

Natural numbers

- ▶ Simple, theoretical, representation, but inefficient
 - ▶ addition, +, subtraction, -, multiplication *
 - ▶ Unusual behavior for subtraction: $3 - 5 = 0$
- ▶ More functions after `Require Import Arith.`
 - ▶ `beq_nat`, `leb` (comparison)
- ▶ Examples
 - ▶ `Definition evenb x :=
 beq_nat (2 * Div2.div2 x) x.`
 - ▶ `Definition Collatz x :=
 if evenb x then Div2.div2 x else 3*x+1.`

Integers

- ▶ Positive and negative numbers, with better efficiency
- ▶ Available only after `Require Import ZArith.`
- ▶ addition, subtraction, multiplication, exponent $^$,
- ▶ Notations as for natural numbers after `Open Scope Z_scope.`
- ▶ `Zle_bool`, `Zlt_bool`, `Zeq_bool`, `Zeven_bool` division `/`, square root,
- ▶ An iterator: able to repeat any function from a type to itself from a given initial
 - ▶ `Definition ZCollatz (x : Z) :=
 if Zeven_bool x then x / 2 else 3 * x + 1.`
 - ▶ `Eval vm_compute in iter 10 Z ZCollatz 31.
 = 242 : Z`
- ▶ Note that the function's second argument is the type in which iterations occur.

pairs and tuples

- ▶ For any two types A B , $A * B$ is also a type
- ▶ Elements of the type are pairs, written (a, b) .
- ▶ Accessing elements of a pair is done with the following construct: `let (a, b) := ... in ...`
- ▶ The names a and b are local names
- ▶ the notation $(1, 2, 3)$ stands for $((1, 2), 3)$
- ▶ Example:
 - ▶ Definition `fact (x:Z) :=
 let (_, r) :=
 iter x (Z * Z)
 (fun p => let (n, r) := p in (n+1, n * r))
 (1, 1) in r.
 Eval vm_compute in fact 100.`

Lists

- ▶ Collections of data of the same type, to replace arrays
- ▶ `Require Import List.`
- ▶ Constructed from the empty list by adding elements in front of existing lists
- ▶ Accessed using `hd` and `nth`, with obligation to give a value for the default cases
- ▶ Notations: `1::2::3::nil`.
- ▶ Peculiarity of `nil`: empty list of a given type, which must be guessed from the context.
- ▶ `Check nil.`
`Error: Cannot infer the implicit parameter A of nil`
- ▶ `Check nil:list nat.`

Programming with lists

- ▶ pre-defined functions: `app` (`++`), `length`, `map`, `filter`, `seq`, `rev`, `combine`
- ▶ Iterators: `fold_left` and `fold_right`.

```
Require Import ZArith List.
```

```
Open Scope Z_scope.
```

```
Definition mx_row (M :list (list Z)) (n:nat) :=  
  nth n M nil.
```

```
Definition mx_col (M :list (list Z)) (n:nat) :=  
  map (fun row => nth n row 0) M.
```

Programming with lists

```
Definition vec_sum (v : list Z) :=  
  fold_right Zplus 0 v.
```

```
Definition pairwise_mult (V1 V2 : list Z) :=  
  map (fun (p : Z * Z) => let (x,y) := p in x*y)  
    (combine V1 V2).
```

```
Definition vec_prod (V1 V2 : list Z) :=  
  vec_sum (pairwise_mult V1 V2).
```

Programming with lists

```
Definition coord_mx (n m:nat) :=  
  map (fun i =>  
    map (fun j => (i, j)) (seq 0 m))  
    (seq 0 n).
```

```
Definition mx_prod (n m p : nat)  
(M N:list (list Z)) :=  
  map (map (fun t : nat*nat =>  
    let (i, j) := t in  
    vec_prod (mx_row M i)  
              (mx_col N j)))  
    (coord_mx n p).
```