

Programming with Coq

Yves Bertot

August 2009

In this class, we introduce the basic commands for defining new values and programs. We shall present :

- ▶ The defining commands
- ▶ The typing discipline : how to construct a well-formed value
- ▶ A collection of basic types and functions

The **Definition** command

Attach 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

Check $2 + 3$.

2 + 3 : nat

Check $5 + 3$.

5 + 3 : nat

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 1` is a function, like `add3`
- ▶ 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
```

Anonymous functions

Abstraction : remove a sub-expression from another expression,
make it a variable

Check $2 + 3$.

$2 + 3 : \text{nat}$

Check $\text{fun } (x : \text{nat}) \Rightarrow x + 3$.

$\text{fun } x : \text{nat} \Rightarrow x + 3 : \text{nat} \rightarrow \text{nat}$

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

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 `rep2`.

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

Check `add3`.

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

Check `rep2 add3`.

```
rep2 add3 : nat -> nat
```

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`

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, 0, S, S (S 0)).
```

```
(3, 0, S, 2) : nat * nat * (nat -> nat) * nat
```

```
Fixpoint fact (x : nat) :=
```

```
  match x with 0 => 1 | S p => x * fact p end.
```

```
fact is recursively defined (...)
```

```
Eval compute in fact 6.
```

```
= 720 : nat
```

Recursive programming with natural numbers

- ▶ Recursive definition keyword : `Fixpoint`
- ▶ Choice of a principal argument
- ▶ Recursive calls allowed only on numbers that are smaller than the initial argument
- ▶ Smaller numbers found by pattern-matching

```
Fixpoint fact (x:nat) : nat :=  
  match x with 0 => 1 | S p => x * fact p end
```

This simple schema only for natural numbers ! More complex schemas for other datatypes.

Focus on integers

```
Require Import ZArith.
```

```
Open Scope Z_scope.
```

```
Check (3, Z0, xH, xI, x0, Zpos (x0 xH)).
```

```
(3, 0, 1%positive, xI, x0, 2)
```

```
: Z * Z * positive * (positive -> positive) *  
  (positive -> positive) * Z
```

```
Eval compute in
```

```
  iter 6
```

```
    (Z*Z)
```

```
    (fun p => let (x,r) := p in (x+1, (x+1)*r))
```

```
    (0, 1).
```

```
= (6, 720) : Z * Z
```

Mathematical notations

- ▶ Depending on top opened scope
- ▶ Numbers read as natural numbers or integers
- ▶ Force the scope with `%nat` or `%Z`
- ▶ Infix notation for usual binary operations
- ▶ Understand notations with the command `Locate`

Understanding notations

```
Locate "_ * _".
```

```
Notation          Scope
```

```
"x * y" := prod x y : type_scope
```

```
"x * y" := Pmult x y : positive_scope
```

```
"n * m" := mult n m : nat_scope
```

```
"x * y" := Zmult x y : Z_scope (default interpretation)
```

```
"x * y" := Nmult x y : N_scope
```

Large collections of data

- ▶ If A is a datatype then `list A` is also a datatype
- ▶ Theoretically no size limit,
- ▶ Written `$a_1 :: a_2 :: \dots :: \text{nil}$`
- ▶ `$a_1 :: l$` adds the element a_1 in front of l
- ▶ `nil` is an empty list
- ▶ Need to load a package to use this, provides many functions

Examples using lists

```
Require Import List.
```

```
Check 1::2::3::nil.
```

```
1::2::3::nil : list Z
```

```
Eval compute in (1::2::3::nil) ++ (4::5::nil).
```

```
= 1::2::3::4::5::nil : list Z
```

```
Eval compute in length (1::2::nil).
```

```
= 2 : nat
```

```
Eval compute in map (fun x => x + 1) (1::2::3::nil).
```

```
= 2::3::4::nil : list Z
```

```
Eval compute in
```

```
  fold_right (fun x y => x + y) 0 (1::2::3::nil).
```

```
= 6 : Z
```

Boolean values

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