Programming in Coq

Yves Bertot

September 2019

The plan

- Write simple programs
- Write properties of the programs
- Prove that we did not lie

Illustration: algorithm, property, proof

Describe an algorithm

- add all numbers between 0 and n

Describe a property

- if x_n is the result value, then $2x_n = n(n+1)$

Prove the property

- By induction on n, if $n = 0$ both sides are 0
- if we assume $2x_n = n(n+1)$ we have

$$2x_{n+1} = 2(x_n + (n+1)) = 2x_n + 2(n+1) = n(n+1) + 2(n+1)$$

$$\ldots = (n+1) * ((n+1)+1)$$

Let's start easy

Two ways to develop software in Coq

- Describe algorithms inside Coq, Execute outside
  - More extended programming language (+)
  - Lighter runtime environment (+)
- Do everything inside Coq
  - Simpler programming language (-)
  - Use Coq as an interpreter (-)
  - Instant feedback (+)
- We will mostly show the latter

Basic data structures

- numbers 1, 42, 1024
- boolean values `true`, `false`
- pairs `(1, true)`
- lists of things `1 :: 2 :: 3 :: 4 :: nil`
- functions `fun x => x`

  `fun x => (x, true)`

  `fun x => x :: x :: 3 :: 4 :: nil`

Data structures under the hood

natural numbers consist of *symbolic expressions* of two kinds

- `O`
- `S` n
  where n is already a natural number

For instance `S (S (S O))` is a natural number
Display machinery will print it as 3
Lists consists of *symbolic expressions* of two kinds

- `nil`
- `cons` a l
  where a is an element and l is a list of elements

For instance `cons (S O) (cons (S (S O)) nil)` is a list
Display machinery will print it as `1 :: 2 :: nil`

More about functions

- Always use initial line
  `Require Import Arith ZArith List Bool Psatz.`
- 2-argument operations on numbers `+`, `*`, `/`, `mod`, `-`
- boolean relations on numbers `<?`, `=?`, `<=?`
- 2-argument operations on boolean values `&&`, `||`
- boolean negation `negb`
- test on boolean value `if then else`
- projections on pairs `fst`, `snd`
- more complex programming structure for lists (to be given later)

Defining and using your own functions

- Give a name to a value : `Definition name := value.`
  - Give a name to a function :
    `Definition fname := fun x : nat => x.`
  - Alternative : `Definition fname (x : nat) := x.`
- Use a function: write the name before the argument
  write fname (fname 1)
  - parentheses not always needed
- Check your own formulas using the `Check` command.
- Compute your examples using the `Compute` command.
- Know what is defined using the `Print` command.

Examples

```
Require Import Arith ZArith List Bool Psatz.

Definition add2 x := x + 2.

Check add2 3.
add2 3 : nat

Compute add2 3.
= 5 : nat

Print add2.
add2 = fun x : nat => x + 2
     : nat -> nat
```

Examples (2)

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

Compute twice add2 1.
= 5 : nat

Compute twice (twice add2) 1.
= 9 : nat
```

Comments on the examples

- `twice` is a function with two arguments
  - the syntax is really different from `C, java, etc.`
- parentheses are needed around `f x` in the definition of twice
- **No parentheses** around the two arguments in the use of `twice`
- `twice` can also be used with only one argument
  the value is a function

Functions about data-structures

- components of a pair : `fst`, `snd`
- Fetching elements of a list

Programming with pairs

```
Definition pair_to_sum1 (p : nat * nat) := fst p + snd p.

Definition pair_to_sum2 (p : nat * nat) :=
  match p with (a, b) => a + b end.

Compute pair_to_sum2 (3, 5).
= 8 : nat
```

Programming with lists

```
Definition headplus1 (l : list nat) :=
  match l with
    a :: l1 => a + 1
  | nil => 0
  end.

Compute headplus1 (3 :: 2 :: nil).
```

Programming with lists

```
Definition headplus1 (l : list nat) :=
  match l with
    a :: l1 => a + 1
  | nil => 0
  end.

Compute headplus1 (3 :: 2 :: nil).
= 4 : nat
```

Recursive programming with lists

- A list has a sub-component that is itself a list
- A recursive program can call itself on that sub-component

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

Fixpoint my_filter {T : Type} (p : T -> bool)
    (l : list T) : list T :=
match l with
  nil => nil
| a :: l1 =>
  if p a then a :: my_filter p l1 else my_filter p l1
end.
```

Comments on list programming

- Lists and pairs are *polymorphic* data structures
- You don't need to know the type of elements for many operations
- Types actually are function arguments
  - You can choose for the type argument to be implicit
- No undefined behavior: all functions must cover all cases of the data-structure

animated recursive computing

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

grow_nat (0 :: 1 :: nil)
```

animated recursive computing

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

2 * 0 :: 2 * 0 + 1 :: grow_nat (1 :: nil)
```

animated recursive computing

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

2 * 0 :: 2 * 0 + 1 :: 2 * 1 :: 2 * 1 + 1 :: grow_nat nil
```

animated recursive computing

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

2 * 0 :: 2 * 0 + 1 :: 2 * 1 :: 2 * 1 + 1 :: nil
```

Initial example : sum of n first natural numbers

```
Fixpoint sumn (n : nat) :=
  match n with 0 => 0 | S p => sumn p + (p + 1) end.

Compute sumn 10.
 = 55 : nat
```

Dependently typed functions

- Coq notation for dependent product: `forall`
- Polymorphic identity has type : `forall A, A -> A`
  - It is a 2-argument function
- All branches of pattern matching return in the same type by default
- Dependently typed pattern matching has a `return` clause

Strongly typed programming

- Including proofs in data
- Including specifications in types
- Example: `forall x y:  nat, {x = y} + {x <> y}`
- Such a function returns more information than just `true` or `false`
- the type is the full specification
- This style is not exploited much in this tutorial

Advanced topic: type classes

- What you write is not what you get
- Implicit arguments are used extensively
  - Polymorphism
  - Overloading
- Coq works for you: type inference
- You can guide type inference

illustrating implicit arguments

```
Check length.
length : forall A : Type, list A -> nat
Check length (1 :: nil).
length (1 :: nat) : nat
Set Printing Implicit.
Check length (1 :: nil).
@length nat (1 :: nat) : nat
```

The value of the first argument is guessed by Coq

Arguments can be made implicit: 3 approaches

- By default `Set Implicit Arguments.`
- At declaration time `Definition id {A:Type} A -> A.`
- Later using a command called `Arguments`.

Type classes: ad-hoc type inference

- Type classes: added properties
- Declaration for each type
- Users can set up proof search for type inference

Example type classes (thx Software foundations)

```
Require Import String.

Open Scope string_scope.

Class Show (T : Type) := {show : T -> string}.

Instance showBool : Show bool :=
  { show := fun b => if b then "true" else "false" }.
```

```
Fixpoint show_list_aux {T : Type} `{Show T} (l : list T) :
  string :=
  match l with
    nil => "]"
  | e :: l' =>
     "; " ++ show e ++ show_list_aux l'
  end.
```

```
Definition show_list {T : Type} `{Show T} (l : list T) :=
  match l with
    nil => "[]"
  | a :: l' => "[" ++ show a ++ show_list_aux l'
  end.

Instance showList {T : Type} `{Show T} : Show (list T) :=
 {show := show_list}.

Compute
  show ((true::false::nil)::(true::nil)::nil::nil).
 = "[[true; false]; [true]; []]" : string
Compute show (1:: 2:: nil).
(let (show) := ?Show in show)(1 :: 2 :: nil) : string
```

The risks of type classes

- You can program type class resolution to perform proof search
- Proof search is undecidable
- You may force type inference to never terminate

Arbitrary computation through type classes

```
Definition cstep (x : Z) : Z :=
  if Z.even x then x / 2 else (3 * x + 1).

Class C_index (x : Z) (y : Z).

Instance C_1 : C_index 1 0.

Instance C_o {x n : Z} `{C_index (cstep x) n} :
   C_index x (n + 1).

Definition ctrigger (x : Z) {y : Z} `{C_index x y} := y.

Compute ctrigger 2223.
= 182 : Z
```