Coq Summer School, Session 2 : Basic programming with numbers and lists

Pierre Letouzey

Predefined data structures

- "Predefined" types are actually declared to Coq at load time [1]:

```
Inductive bool := true | false.

Inductive nat := O : nat | S : nat -> nat.

Inductive list A :=
 | nil : list A
 | cons : A -> list A -> list A.
```

- Nota: `a::b` is a notation for `(cons a b)`.

[1] see Init/Datatypes.v

Pattern matching

- We can analyse an expression and handle all possible cases:

```
Definition negb b :=
 match b with
  | true => false
  | false => true
 end.
```

- Most common situation: one pattern for each constructor.
- NB: for `bool`, an alternative syntactic sugar is
  `if b then false else true`.

Pattern matching

- Similarly, for numbers:

```
Definition pred x :=
 match x with
  | S x => x
  | O => O
 end.

Definition iszero x :=
 match x with
  | O => true
  | S _ => false
 end.
```

More complex pattern matching

- We can use deeper patterns, combined matchings, as well as wildcards:

```
Definition istwo x :=
 match x with
  | S (S O) => true
  | _ => false
 end.

Definition andb b1 b2 :=
 match b1, b2 with
  | true, true => true
  | _, _ => false
 end.
```

- These matchings are not atomic, but rather expansed internally into nested matchings (use `Print` to see how many).

Recursion

- When using `Fixpoint` instead of `Definition`, recursive sub-calls are allowed (at least some of them).

```
Fixpoint div2 n :=
 match n with
  | S (S n') => S (div2 n')
  | _ => 0
 end.
```

- Here, `n'` is indeed a *structural sub-term* of the inductive argument `n`.
- This way, termination of computations is (syntactically) ensured.
- Example of rejected recursive functions:

```
Fixpoint loop n := loop (S n).
Fixpoint div2_ko n :=
 if leb n 1 then 0 else S (div2_ko (n-2)).
```

Some other recursive functions over nat

```
Fixpoint plus n m :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.

Fixpoint minus n m := match n, m with
  | S n', S m' => minus n' m'
  | _, _ => n
 end.

Fixpoint beq_nat n m := match n, m with
  | S n', S m' => beq_nat n' m'
  | O, O => true
  | _, _ => false
 end.
```

Recursion over lists

- With recursive functions over lists, the main novelty is *polymorphism* :

```
Fixpoint length A (l : list A) :=
 match l with
  | nil => 0
  | _ :: l' => S (length l')
 end.

Fixpoint app A (l1 l2 : list A) : list A :=
 match l1 with
  | nil => l2
  | a :: l1' => a :: (app l1' l2)
 end.
```

- NB: `(app l1 l2)` is noted `l1++l2`.
- NB: we use here *Implicit Arguments* to avoid writing type parameters such as `A` again and again when applying functions.

Fold on the right

- With `fold_right`, computation starts at the end of the list:
  `fold_right f init (a::b::nil) = (f a (f b init))`
- The code:

```
Fixpoint fold_right A B (f:B->A->A)(init:A)(l:list B)
 : A :=
 match l with
  | nil => init
  | x :: l' => f x (fold_right f init l')
 end.

Eval vm_compute in fold_right plus 0 (1::2::3::nil).
  ==> (1+(2+(3+0))) ==> 6

Eval vm_compute in
 fold_right (fun x l => x::l) nil (1::2::3::nil).
  ==> 1::2::3::nil
```

Fold on the left

- With `fold_left`, computation starts at the top of the list:
  `fold_left f (a::b::nil) init = (f (f init a) b)`
- The code:

```
Fixpoint fold_left A B (f:A->B->A)(l:list B)(init:A)
 : A :=
 match l with
  | nil => init
  | x :: l' => fold_left f l' (f init x)
 end.

Eval vm_compute in fold_left plus (1::2::3::nil) 0.
  ==> (((0+1)+2)+3) ==> 6

Eval vm_compute in
 fold_left (fun l x => x::l) nil (1::2::3::nil).
  ==> 3::2::1::nil
```

A complete example: mergesort

- Part I: splitting a list in two
- Part II: merging two sorted lists into one
- Part III: iterating the process...

Mergesort I : splitting

```
Fixpoint split A (l : list A) : list A * list A :=
 match l with
  | nil => (nil, nil)
  | a::nil => (a::nil, nil)
  | a::b::l' => let (l1, l2) := split l' in (a::l1, b::l2)
 end.

Eval vm_compute in split (1::2::3::4::5::nil).
  ==> (1::3::5::nil, 2::4::nil)
```

Mergesort II : merging

- A new syntax construct: `fix` for local fixpoints

Mergesort II : merging

- A new syntax construct: `fix` for local fixpoints
-

```
Definition merge A (less:A->A->bool)
 : list A -> list A -> list A :=
 fix merge l1 := match l1 with
   | nil => (fun l2 => l2)
   | x1::l1' =>
      (fun l2 => match l2 with
        | nil => l1
        | x2::l2' =>
           if less x1 x2 then x1 :: merge l1' l2
                         else x2 :: merge l1 l2'
      end)
 end.
```

Mergesort II : merging

- A new syntax construct: `fix` for local fixpoints
-
```
Definition merge A (less:A->A->bool)
 : list A -> list A -> list A :=
 fix merge l1 := match l1 with
   | nil => (fun l2 => l2)
   | x1::l1' =>
      (fun l2 => match l2 with
        | nil => l1
        | x2::l2' =>
           if less x1 x2 then x1 :: merge l1' l2
                         else x2 :: merge l1 l2'
      end)
 end.
```
- No structurally decreasing argument: `Rejected!`

Mergesort II : merging

- A new syntax construct: `fix` for local fixpoints
-
  ```
  Definition merge A (less:A->A->bool)
   : list A -> list A -> list A :=
   fix merge l1 := match l1 with
     | nil => (fun l2 => l2)
     | x1::l1' =>
        (fix merge_l1 l2 := match l2 with
          | nil => l1
          | x2::l2' =>
             if less x1 x2 then x1 :: merge l1' l2
                           else x2 :: merge_l1 l2'
        end)
   end.
  ```
- Trick of the specialized internal fix: `Accepted!`

Mergesort III : iterating

- We need to recursively sort sub-lists produced by `split`.
  No direct solution, we use here a auxiliary *counter* `n`,

Mergesort III : iterating

- We need to recursively sort sub-lists produced by `split`.
  No direct solution, we use here a auxiliary *counter* `n`,
- ```
  Definition mergeloop A (less:A->A->bool) :=
   fix loop (l:list A) (n:nat) :=
    match n with
     | O => nil
     | S n => match l with
                | nil => l
                | _::nil => l
                | _ => let (l1,l2) := split l in
                  merge less (loop l1 n) (loop l2 n)
               end
    end.
  ```
  ```

Mergesort III : iterating

- We need to recursively sort sub-lists produced by `split`.
  No direct solution, we use here a auxiliary *counter* `n`,
-
```
Definition mergeloop A (less:A->A->bool) :=
 fix loop (l:list A) (n:nat) :=
  match n with
   | O => nil
   | S n => match l with
              | nil => l
              | _::nil => l
              | _ => let (l1,l2) := split l in
                 merge less (loop l1 n) (loop l2 n)
            end
  end.
```
- Invariant: `n` $\geq$ `length l`

Mergesort III : iterating

- We need to recursively sort sub-lists produced by `split`.
  No direct solution, we use here a auxiliary *counter* `n`,
- ```
  Definition mergeloop A (less:A->A->bool) :=
   fix loop (l:list A) (n:nat) :=
    match n with
     | O => nil
     | S n => match l with
                | nil => l
                | _::nil => l
                | _ => let (l1,l2) := split l in
                  merge less (loop l1 n) (loop l2 n)
              end
    end.
  ```
- Invariant: `n` $\geq$ `length l`
- ```
  Definition mergesort A less (l:list A) :=
    mergeloop less l (length l).
  ```