Datatypes, pattern-matching, and recursion

Yves Bertot

August 2009

In this class, we discuss how to introduce new datatypes and how to program with them. We shall present :

- Enumerated datatypes
- Structure datatypes
- Fetching components from a structure
- Repeated behavior : recursion

Enumerated datatypes

You define a datatype by stating what are its element

```
Inductive month : Type :=
  Jan | Feb | Mar | Apr | May | Jun
| Jul | Aug | Sep | Oct | Nov | Dec.

Check Jan.
Jan : month
```

The various names `Jan`, `Feb`, etc, are called constructors.

Defining values by cases

When a datatype is *inductive*, you can compute values according to which element you are looking at

```
Definition nbdays (m:month) :=
 match m with
   Apr => 30 | Jun => 30 | Sep => 30 | Nov => 30
 | Feb => 28 | _ => 31
 end.

Eval compute in nbdays Jul.
= 31 : nat
```

On the left hand side of =>, one must find a constructor name, or a variable, or the anonymous variable _.

- beware of typographical errors !

Record types

A plain record type packs together several objects

```
Inductive i_plane : Type :=
  point (x y : Z).
```

```
Check point.
point : Z -> Z -> i_plane
```

Here again, we enumerated all possible cases, but we used variables to capture infinite possibilities

Fetching components in records

Again use the pattern matching construct to look at the value being manipulated

```
Definition point_x (p : i_plane) : Z :=
  match p with point x _ => x end.
```

- All cases must be covered, all fields must have a variable
- Here the second field has an anonymous variable

Several variants and several components

Constructors still cover all cases

```
Inductive t1 : Type :=
  c1t1 (n : Z)(s : string)
| c2t1 (n : nat).

Definition ft1 (v : t1) : Z :=
  match v with
    c1t1 a s => a
  | c2t1 n => Z_of_nat n
  end.
```

Functions defined by pattern-matching still have to cover all cases

well-formed pattern-matching

- `match` v `with` p_1 `=>` e_1 `| ...|` p_k `=>` e_k `end`
- v must be well-formed and its type must be an inductive type t
- p_1, ..., p_k must be patterns built with the constructors of t
- e_1, e_k must be well-formed and all share the same type t'
- e_1 can use the variables appearing in p_1 with the corresponding type
- The type of the whole expression is t'

Well-formed pattern-matching on an example

```
Inductive t1 : Type :=
  c1t1 (n : Z)(s : string) | c2t1 (n : nat).

Definition ft1 (v : t1) : Z :=
  match v with
    c1t1 a s => a
  | c2t1 n => Z_of_nat n
  end.
```

Recursive types

- When the current type appears in the component types
- Allows for data of arbitrary size
- Typical example : natural numbers

```
Inductive nat : Set := O | S (n:nat).

Check (O, S O, S (S O)).
(0, 1, 2) : nat * nat * nat
```

- Pattern matching works as usual

Recursive programming is not free

- Provided only for functions with input in an inductive type
- Strict rules on well-formed recursive calls
  - Choice of a principal argument
  - Recursive calls only on variables
  - Variables for recursive calls obtained by pattern-matching from principal
- Guarantee of termination (Weak normalization)

Examples of recursive functions on nat

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

Fixpoint minus (n m : nat) : nat :=
  match n, m with
    S p, S q => minus p q
  | n, _ => n
  end.
```

Example : binary trees with integer labels

```
Inductive btz : Type :=
  Nbtz (x : Z) (t1 t2 : btz) | Lbtz.

Fixpoint btz_size (t : btz) :=
  match t with
    Nbtz _ t1 t2 => 1 + btz_size t1 + btz_size t2
  | Lbtz => 1
  end.
```

Exercise : write a function that adds all the integer values in a binary tree

Polymorphic recursive types

- The type of some components for some constructors can be given by a variable
- This type becomes an extra argument for the constructors
- Technically, not one type is defined, but a family of types
- Implicit arguments can help recover a *polymorphic* style

Polymorphic binary trees

```
Inductive bt (A : Type) : Type :=
  Nbt (x : A) (t1 t2 : bt A) | Lbt.

Implicit Arguments Nbt [A].
Implicit Arguments Lbt [A].
```

Thanks to implicit arguments declarations, the `A` argument to `Nbt` and `Lbt` is never written, but guessed from the `x` argument. `Nbt` has 4 arguments, but can be used as if it had 3.

```
Check Nbt 1 Lbt Lbt.
Nbt 1 Lbt Lbt : bt Z
```

To force implicit arguments, one uses the notation `@Lbt`

Pattern-matching with parameters

Parameters do not appear in pattern-matching construct

```
Fixpoint bt_size (A:Type)(t : bt A) : Z :=
  match t with
    Nbt _ t1 t2 => 1 + bt_size t1 + bt_size t2
  | Lbt => 1
  end.
```

Beware : this is not related to implicit arguments.

Polymorphic lists

Lists as provided in Coq are a polymorphic recursive datatype

```
Inductive list (A : Type) : Type :=
  nil | cons (a : A) (l : list A).
```

The argument `A` is implicit for both `nil` and `cons`.
The notation `a :: l` is for `cons a l` $\equiv$ `@cons _ a l`

Programming with lists

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

Eval compute in dispatch Z (1::2::3::4::5::nil).
(1::3::5::nil, 2::4::nil) : list Z*list Z
```

Pair type as a polymorphic datatype

The type of pairs is also a polymorphic inductive type

```
Inductive prod (A B : Type) : Type :=
  pair (a : A) (b : B).
```

For pair, arguments `A` and `B` are implicit the notation `A * B` stands for `prod A B` (when a type is expected).
There also exists a `sum` type.

Option type as a polymorphic datatype

All functions in Coq are total
When modeling a partial function, it is useful to describe it as a total function to a type with an extra value

```
Inductive option (A : Type) : Type :=
  Some : A -> option A
| None : option A
```

The argument `A` in `Some` and `None` is implicit