Inductive data types

Assia Mahboubi

16 novembre 2011

In this class, we shall present how *Coq*'s type system allows us to define data types using inductive declarations.

Inductive declarations

An arbitrary type as assumed by :

```
Variable T : Type.
```

gives no a priori information on the nature, the number, or the properties of its inhabitants.

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

```
Print bool.
```

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

```
Print bool.
```

Inductive bool : Set := true : bool | false : bool.

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

```
Print bool.
```

Inductive bool : Set := true : bool | false : bool.

```
Print nat.
```

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

```
Print bool.
```

Inductive bool : Set := true : bool | false : bool.

```
Print nat.
```

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

Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule :

```
Print bool.
```

Inductive bool : Set := true : bool | false : bool.

```
Print nat.
```

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

Each such rule is called a constructor.

Inductive declarations in *Coq*

Inductive types in *Coq* can be seen as the generalization of similar type constructions in more common programming languages.

They are in fact an extremely rich way of defining data-types, operators, connectives, specifications,...

They are at the core of powerful programming and reasoning techniques.

Enumerated types

Enumerated types are types which list and name exhaustively their inhabitants.

```
Inductive bool : Set := true : bool | false : bool.
```

Enumerated types

Enumerated types are types which list and name exhaustively their inhabitants.

```
Inductive bool : Set := true : bool | false : bool.

Inductive color:Type :=
| white | black | yellow | cyan | magenta
| red | blue | green.

Check cyan.
```

cyan : color

Labels refer to distinct elements.

Enumerated types : program by case analysis

Inspect the enumerated type inhabitants and assign values :

```
Definition my_negb (b : bool) :=
  match b with true => false | false => true.
```

Enumerated types : program by case analysis

Inspect the enumerated type inhabitants and assign values :

```
Definition my_negb (b : bool) :=
  match b with true => false | false => true.

Definition is_prevert_animal (x : prevert_enum) : bool :=
  match x with
  | dozen_of_oysters => true
  | an_other_racoon => true
  | _ => false
  end.
```

Enumerated types : program by case analysis

Inspect the enumerated type inhabitants and assign values :

```
Definition my_negb (b : bool) :=
  match b with true => false | false => true.

Definition is_prevert_animal (x : prevert_enum) : bool :=
  match x with
  | dozen_of_oysters => true
  | an_other_racoon => true
  | _ => false
  end.

Eval compute in (is_prevert_animal one_stone).
```

Enumerated types : program by case analysis

Inspect the enumerated type inhabitants and assign values :

```
Definition my_negb (b : bool) :=
  match b with true => false | false => true.

Definition is_prevert_animal (x : prevert_enum) : bool :=
  match x with
  | dozen_of_oysters => true
  | an_other_racoon => true
  | _ => false
  end.

Eval compute in (is_prevert_animal one_stone).
                                             = false
                                             : bool
```

Enumerated types : reason by case analysis

Inspect the enumerated type inhabitants and build proofs :

```
Lemma bool_case : forall b : bool, b = true ∨ b = false.
Proof.
intro b.
case b.
  left; reflexivity.
right; reflexivity.
Qed.
```

Enumerated types : reason by case analysis

Inspect the enumerated type inhabitants and build proofs :

```
Lemma is_prevert_animalP : forall x : prevert_enum,
  is_prevert_animal x = true ->
  x = dozen_of_oysters \/ x = an_other_racoon.
Proof.
(* Case analysis + computation *)
intro x; case x; simpl; intro e.
(* In the three first cases: e: false = true *)
  discriminate e.
  discriminate e.
  discriminate e.
(* Now: e: true = true *)
  left; reflexivity.
  right; reflexivity.
Qed.
```

Enumerated types : reason by case analysis

Two important tactics, not specific to enumerated types :

- `simpl` : makes computation progress (pattern matching applied to a term starting with a constructor)
- `discriminate` : allows to use the fact that constructors are distincts :
  - `discriminate H` : closes a goal featuring a hypothesis `H` like `(H : true = false)` ;
  - `discriminate` : closes a goal like `(O <> S n)`.

Recursive types

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

Recursive types

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

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

Recursive types

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

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

Base case constructors do not feature self-reference to the type.
Recursive case constructors do.

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
```

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
```

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
| Node : nat ->
```

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
| Node : nat -> natBinTree -> natBinTree
```

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
| Node : nat -> natBinTree -> natBinTree -> natBinTree
```

Recursive types

Let us craft new inductive types :

```
Inductive natBinTree : Set :=
| Leaf  : nat -> natBinTree
| Node  : nat -> natBinTree -> natBinTree -> natBinTree

Inductive term : Set :=
|Zero : term
|One : term
|Plus : term -> term -> term
|Mult : term -> term -> term.
```

An inhabitant of a recursive type is built from a finite number of constructor applications.

Recursive types : program by case analysis

We have already seen some examples of such pattern matching :

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

Recursive types : program by case analysis

We have already seen some examples of such pattern matching :

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

Definition is_single_nBT (t : natBinTree) :=
match t with
|Leaf _ => true
|_ => false
end.
```

Recursive types : proofs by case analysis

```
Lemma is_single_nBTP : forall t,
  is_single_nBT t = true -> exists n : nat, t = Leaf n.
Proof.
```

Recursive types : proofs by case analysis

```
Lemma is_single_nBTP : forall t,
  is_single_nBT t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
  while introducing *)
intros [ nleaf | nnode t1 t2] h.
```

Recursive types : proofs by case analysis

```
Lemma is_single_nBTP : forall t,
  is_single_nBT t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
  while introducing *)
intros [ nleaf | nnode t1 t2] h.
(* First case: we use the available label *)
  exists nleaf.
  reflexivity.
```

Recursive types : proofs by case analysis

```
Lemma is_single_nBTP : forall t,
  is_single_nBT t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
  while introducing *)
intros [ nleaf | nnode t1 t2] h.
(* First case: we use the available label *)
  exists nleaf.
  reflexivity.
(* Second case: the test evaluates to false *)
simpl in h.
discriminate.
```

Recursive types : proofs by case analysis

```
Lemma is_single_nBTP : forall t,
  is_single_nBT t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
  while introducing *)
intros [ nleaf | nnode t1 t2] h.
(* First case: we use the available label *)
  exists nleaf.
  reflexivity.
(* Second case: the test evaluates to false *)
simpl in h.
discriminate.
Qed.
```

Recursive types

Constructors are injective :

```
  Lemma inj_leaf : forall x y, Leaf x = Leaf y -> x = y.
  Proof.
  intros x y hLxLy.
  injection hLxLy.
  trivial.
Qed.
```

Recursive types : structural induction

Let us go back to the definition of natural numbers :

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

Recursive types : structural induction

Let us go back to the definition of natural numbers :

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

The Inductive keyword means that at definition time, this system geneates an induction principle :

```
nat_ind
     : forall P : nat -> Prop,
       P 0 ->
```

Recursive types : structural induction

Let us go back to the definition of natural numbers :

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

The Inductive keyword means that at definition time, this system geneates an induction principle :

```
nat_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Recursive types : structural induction

To prove that for `P : term -> Prop`, the theorem `forall t : term, P t` holds, it is sufficient to :

Recursive types : structural induction

To prove that for `P : term -> Prop`, the theorem `forall t : term, P t` holds, it is sufficient to :

- Prove that the property holds for the base cases :
  - `(P Zero)`
  - `(P One)`

Recursive types : structural induction

To prove that for `P : term -> Prop`, the theorem `forall t : term, P t` holds, it is sufficient to :

- Prove that the property holds for the base cases :
  - `(P Zero)`
  - `(P One)`
- Prove that the property is transmitted inductively :
  - ```
    forall t1 t2 : term,
    P t1 -> P t2 -> P (Plus t1 t2)
    ```
  - ```
    forall t1 t2 : term,
    P t1 -> P t2 -> P (Mult t1 t2)
    ```

Recursive types : structural induction

To prove that for `P : term -> Prop`, the theorem `forall t : term, P t` holds, it is sufficient to :

- Prove that the property holds for the base cases :
  - `(P Zero)`
  - `(P One)`
- Prove that the property is transmitted inductively :
  - `forall t1 t2 : term,`
    `P t1 -> P t2 -> P (Plus t1 t2)`
  - `forall t1 t2 : term,`
    `P t1 -> P t2 -> P (Mult t1 t2)`

The type `term` is the smallest type containing `Zero` and `One`, and closed under `Plus` and `Mult`.

Recursive types : structural induction

The induction principles generated at definition time by the system allow to :

- Program by recursion (`Fixpoint`)
- Prove by induction (`induction`)

Recursive types : program by structural induction

We can compute some information on the size of a term :

```
Fixpoint height (t : natBinTree) : nat :=
  match t with
    |Leaf _ => 0
    |Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.
```

Recursive types : program by structural induction

We can compute some information on the size of a term :

```
Fixpoint height (t : natBinTree) : nat :=
  match t with
    |Leaf _ => 0
    |Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.

Fixpoint size (t : natBinTree) : nat :=
  match t with
```

Recursive types : program by structural induction

We can compute some information on the size of a term :

```
Fixpoint height (t : natBinTree) : nat :=
  match t with
    |Leaf _ => 0
    |Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.

Fixpoint size (t : natBinTree) : nat :=
  match t with
  |Leaf _ => 1
```

Recursive types : program by structural induction

We can compute some information on the size of a term :

```
Fixpoint height (t : natBinTree) : nat :=
  match t with
    |Leaf _ => 0
    |Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.

Fixpoint size (t : natBinTree) : nat :=
  match t with
  |Leaf _ => 1
  |Node _ t1 t2 => (size t1) + (size t2) + 1
  end.
```

Recursive types : program by structural induction

We can access some information contained in a term :

```
Require Import List.
Fixpoint label_at_occ (dflt : nat)
               (t : natBinTree)(u : list bool) :=
match u, t with
|nil, _ =>
  (match t with Leaf n => n | Node n _ _ => n end)
|b :: tl, t =>
  match t with
    |Leaf _ => dflt
    | Node _ t1 t2 =>
      if b then label_at_occ t2 tl dflt
      else label_at_occ t1 tl dflt
  end
end.
```

Recursive types : proofs by structural induction

We have already seen induction at work on nats and lists.
Here its goes on binary trees :

```
Lemma le_height_size : forall t : natBinTree,
           height t <= size t.
Proof.
induction t; simpl.
  auto.
apply plus_le_compat_r.
apply max_case.
  apply (le_trans _ _  _ IHt1).
  apply le_plus_l.
  apply (le_trans _ _  _ IHt2).
  apply le_plus_r.
Qed.
```

Option types

A polymorphic (like `list`) non recursive type :

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

Option types

A polymorphic (like `list`) non recursive type :

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

Use it to lift a type to a copy of this type but with a default value :

```
Fixpoint olast (A : Type)(l : list A) : option A :=
  match l with
    |nil => None
    |a :: nil => Some a
    |a :: l => olast A l
  end.
```

Pairs & co

A polymorphic (like `list`) pair construction :

```
Print pair.
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> A * B
```

Pairs & co

A polymorphic (like `list`) pair construction :

```
Print pair.
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> A * B
```

The notation `A * B` denotes `(prod A B)`.
The notation `(x, y)` denotes `(pair x y)` (implicit argument).

```
Check (2, 4).     : nat * nat
Check (true, 2 :: nil).     : bool * (list nat)
```

Fetching the components :

```
Eval compute in (fst (0, true)).
 = 0 : nat
Eval compute in (snd (0, true)).
 = true : bool
```

Pairs & co

Pairs can be nested :

```
Check (0, 1, true).
  : nat * nat * bool
Eval compute in (fst (0, 1, true)).
 = (0, 1)
                 : nat * nat
```

This can also be adapted to polymorphic n-tuples :

```
Inductive triple (T1 T2 T3 : Type) :=
  Triple T1 -> T2 -> T3 -> triple T1 T2 T3.
```

Record types

A record type bundles pieces of data you wish to gather in a single type.

```
Record admin_person := MkAdmin {
id_number : nat;
date_of_birth : nat * nat * nat;
place_of_birth : nat;
gender : bool}
```

They are also inductive types with a single constructor !

Record types

You can access to the fields :

```
Variable t : admin_person.
Check (id_number t).

  : nat
Check id_number.

fun a : admin_person =>
        let (id_number, _, _, _) := a in id_number
   : admin_person -> nat
```

In proofs, you can break an element of record type with tactics `case`/`destruct`.

Warning : this is pure functional programming...