Inductive data types

Assia Mahboubi, Pierre Castéran
Paris, Beijing, Bordeaux

31 août 2010

Inductive declarations

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

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

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

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 (just like bool).

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

Check cyan.
```

cyan : color

```
Lemma red_diff_black : red <> black.
Proof.
 discriminate.
Qed.
```

Enumerated types : program by case analysis

Inspect the enumerated type inhabitants and assign values :

```
Definition is_bw (c : color) : bool :=
  match c with
  |  black => true
  |  white => true
  | _ => false
  end.

Eval compute in (is_bw red).
```

$=$ *false : bool*

Enumerated types : reason by case analysis

Inspect the enumerated type inhabitants and build proofs :

```
Lemma is_bw_cases : forall c : color,
  is_bw c = true ->
  c = white ∨ c = black.
Proof.
(* Case analysis + computation *)
intro c; case c; simpl; intro e.
```

8 subgoals

c : color

e : true = true

============================

white = white ∨ white = black

. . .

```
left;reflexivity.
```

7 subgoals

$c : color$

$e : true = true$

$============================$

$black = white \lor black = black$

subgoal 2 is:

$red = white \lor red = black$

$\ldots$

```
right;trivial.
```

```
6 subgoals
```

$c : color$

$e : false = true$

$==============================$

$red = white \vee red = black$

$\ldots$

```
discriminate e.
```

5 subgoals. . .

A shorter proof :

```
Lemma is_bw_cases' : forall c, is_bw c = true ->
   c=white \/ c=black.
Proof.
 intros c H; destruct c; try discriminate H ;
                            ((left;reflexivity) ||
                             (right;reflexivity)).
Qed.
```

Note : The tactic case_eq *t* behaves like case or destruct, but introduces the equality $t = c$ in the context associated to every constructor C (see the reference manual).

Enumerated types : reason by case analysis

Two important tactics :

- `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 : red = blue)`;
  - `discriminate` : closes a goal whose conclusion is `(red <> black)`.

Recursive types

Remember nat and list A :

```
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 :=
| Leaf : natBinTree
| Node (n:nat)(t1 t2 : natBinTree).

Definition t0 : natBinTree :=
  Node 5 (Node 3 Leaf Leaf)
         (Node 8 Leaf Leaf).
```

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

Programming with recursive types : pattern matching and recursivity

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

Fixpoint mirror (t: natBinTree) : natBinTree :=
match t with
| Leaf => Leaf
| Node n t1 t2 => Node n (mirror t2) (mirror t1)
end.
```

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

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

```
Require Import List.

Fixpoint labels (t: natBinTree) : list nat :=
match t with
| Leaf => nil
| Node n t1 t2 => labels t1 ++ (n :: labels t2)
end.

Eval compute in labels (Node 9 t0 t0).
```

$= 3 :: 5 :: 9 :: 3 :: 5 :: nil$

$: list\ nat$

Recursive types : proofs by case analysis

```
Lemma tree_decompose : forall t, tree_size t <> 1 ->
                     exists n:nat,  exists t1:natBinTree,
                     exists t2:natBinTree,
                      t = Node n t1 t2.
Proof.
 intros t H; destruct t as [ | i t1 t2].
```

Recursive types : proofs by case analysis

```
Lemma tree_decompose : forall t, tree_size t <> 1 ->
                          exists n:nat,  exists t1:natBinTree,
                          exists t2:natBinTree,
                           t = Node n t1 t2.
Proof.
 intros t H; destruct t as [ | i t1 t2].
```

2 subgoals:
H : tree_size Leaf <> 1
==============================
exists n : nat,
exists t1 : natBinTree,
exists t2 : natBinTree,
Leaf = Node n t1 t2

```
 destruct H;reflexivity.
```

```
1 subgoal

  i : nat
  t1 : natBinTree
  t2 : natBinTree
   H : tree_size (Node i t1 t2) <> 1
  ============================
   exists n : nat,
     exists t3 : natBinTree,
       exists t4 : natBinTree, Node i t1 t2 = Node n t3 t4
 exists i;exists t1;exists t2;reflexivity.
Qed.
```

Recursive types

Constructors are injective :

```
Lemma Node_inj : forall n p t1 t2 t3 t4,
                 Node n t1 t2 = Node p t3 t4 ->
                 n = p ∧ t1 = t3 ∧ t2 = t4.
Proof.
 intros n p t1 t2 t3 t4 H;injection H.
```

Recursive types

Constructors are injective :

```
Lemma Node_inj : forall n p t1 t2 t3 t4,
                 Node n t1 t2 = Node p t3 t4 ->
                 n = p ∧ t1 = t3 ∧ t2 = t4.
Proof.
 intros n p t1 t2 t3 t4 H;injection H.
```

1 subgoal:

n : nat ...

t3 : natBinTree

t4 : natBinTree

H : Node n t1 t2 = Node p t3 t4

============================

t2 = t4 -> t1 = t3 -> n = p -> n = p ∧ t1 = t3 ∧ t2 = t4

```
auto.
Qed.
```

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 generates 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 : natBinTree -> Prop`, the theorem `forall t : term, P t` holds, it is sufficient to :

Recursive types : structural induction

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

- Prove that the property holds for the base case :
  - `(P Leaf)`

Recursive types : structural induction

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

- Prove that the property holds for the base case :
  - `(P Leaf)`
- Prove that the property is transmitted inductively :
  - `forall (n :nat) (t1 t2 : natBinTree),`
    `P t1 -> P t2 -> P (Node n t1 t2)`

Recursive types : structural induction

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

- Prove that the property holds for the base case :
  - `(P Leaf)`
- Prove that the property is transmitted inductively :
  - `forall (n :nat) (t1 t2 : natBinTree), P t1 -> P t2 -> P (Node n t1 t2)`

The type `natBinTree` is the smallest type containing `Leaf`, and closed under `Node`.

```
Check natBinTree_ind.
natBinTree_ind
     : forall P : natBinTree -> Prop,
       P Leaf ->
       (forall (n : nat) (t1 : natBinTree),
        P t1 -> forall t2 : natBinTree, P t2 -> P (Node n t1 t2)) ->
       forall n : natBinTree, P n
```

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 : 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,
          tree_height t <= tree_size t.
Proof.
induction t; simpl.
```

2 subgoals:

==============================

1 <= 1

subgoal 2 is:

S (max (tree_height t1) (tree_height t2)) <=
S (tree_size t1 + tree_size t2)

```
auto with arith.
```

```
n : nat
t1 : natBinTree
t2 : natBinTree
IHt1 : tree_height t1 <= tree_size t1
IHt2 : tree_height t2 <= tree_size t2
============================
 S (max (tree_height t1) (tree_height t2)) <=
 S (tree_size t1 + tree_size t2)
```

```
Require Import Omega.
Search About max.
```

```
max_case
   : forall (n m : nat) (P : nat -> Type), P n -> P m -> P (max n m)
```

```
apply max_case;omega.
Qed.
```

A more concrete example

Let us consider a *toy* (very small) programming language. You will see a bigger language next friday.
We want to be able to write and analyze programms lile below :

```
X = 0 ;
Y = 1 ;
do Z times {
   X = X + 1;
   Y := Y * X
}
```

- Only three variables : X, Y and Z
- 2 operations : addition and multiplication
- simple for loop

A type for the variables

```
Inductive toy_Var : Set := X | Y | Z.
```

Note : If you wanted an infinite number of variables, you would have written :

```
Inductive toy_Var : Set := toy_Var (label : nat).
```

or

```
Require Import String.
Inductive toy_Var : Set := toy_Var (name: string).
```

Expressions

We associate a constructor to each way of building an expression :

- integer constants
- variables
- application of a binary operation

```
Inductive toy_Op := toy_plus | toy_mult.

Inductive toy_Exp := const (i:nat) |
                     variable (v:toy_Var) |
                     toy_op (op:toy_Op) (e1 e2: toy_Exp)
```

Don't be mistaken !

```
Lemma toy_plus_inj : forall e1 e2 e3 e4,
  toy_op  toy_plus e1 e2 = toy_op toy_plus  e3 e4 ->
   e1 = e3 /\ e2 = e4.
Proof.
 intros e1 e2 e3 e4 H;injection H;auto.
Qed.
```

Don't be mistaken !

```
Lemma toy_plus_inj : forall e1 e2 e3 e4,
  toy_op  toy_plus e1 e2 = toy_op toy_plus  e3 e4 ->
   e1 = e3 /\ e2 = e4.
Proof.
 intros e1 e2 e3 e4 H;injection H;auto.
Qed.

Lemma plus_not_inj : ~(forall  n p q r:nat, n+p=q+r ->
                       n = q /\ p = r).
Proof.
 intro H;destruct (H 2 2 3 1) as [H0 H1].
 trivial.
 discriminate H0.
Qed.
```

Statements

```
Inductive toy_Statement :=
  | (* x = e *)
     assign (v:toy_Var)(e:toy_Exp)
  | (* s ; s1 *)
     sequence (s s1: toy_Statement)
  | (* for i := e to n do s *)
    simple_loop (e:toy_Expr)(s : toy_Statement).
```

```
Definition factorial_Z_program :=
sequence (assign X (const 0))
 (sequence
   (assign Y (const 1))
   (simple_loop (variable Z)
    (sequence
      (assign X
         (toy_op toy_plus (variable X) (const 1)))
      (assign Y
         (toy_op toy_mult (variable Y) (variable X)))))).
```

```
Inductive toy_State : Set :=
state (val_X val_Y val_Z : nat).

Definition update (v:toy_Var)(s: toy_State)(val : nat):=
match v,s with |X, state _ y z => state val y z
               |Y, state x _ z => state x val z
               |Z, state x y _ => state x y val
end.
```

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 version with 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*

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;
sex : 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).
```
id_number t : nat
```
Check id_number.
```
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...