Inductive properties

Assia Mahboubi, Pierre Castéran
Paris, Beijing, Bordeaux

31 août 2010

We have already seen how to define new datatypes by the mean of inductive types.
During this session, we shall present how *Coq*'s type system allows us to define specifications using inductive declarations.

```
Inductive even : nat -> Prop :=
|  even0 : even 0
|  evenS : forall p:nat, even p -> even (S (S p)).

Require Import List.
Set Implicit Arguments.
Inductive is_repetition(A:Type) : list A -> Prop :=
| is_rep_nil : is_repetition nil
| is_rep_single : forall a, is_repetition (a::nil)
| is_rep_cons: forall a l, is_repetition (a::l) ->
                           is_repetition (a::a::l).
```

Inductive predicates

Let us consider again our little programming language. We can define the predicate "the variable *v* appears in the expression *e*" :

```
Inductive Occurs  (v:toy_Var): toy_Exp -> Prop :=
|Occ_var : Occurs v (variable v)
|Occ_op1 : forall op e1 e2, Occurs v e1 ->
                            Occurs v (toy_op op e1 e2)
|Occ_op2 : forall op e1 e2, Occurs v e2 ->
                            Occurs v (toy_op op e1 e2).
```

Constructors are displayed in red.

Likewise, "The variable *v* may be modified by an execution of the statement *s*".

```
Inductive Assigned_in (v:toy_Var): toy_Statement->Prop :=
| Assigned_assign : forall e, Assigned_in v (assign v e)
| Assigned_seq1 : forall s1 s2,
                    Assigned_in v s1 ->
                    Assigned_in v (sequence s1 s2)
| Assigned_seq2 : forall s1 s2,
                    Assigned_in v s2 ->
                    Assigned_in v (sequence s1 s2)
| Assigned_loop : forall e s,
                    Assigned_in v s ->
                    Assigned_in v (simple_loop e s).
```

For proving that some given variable is assigned in some given statement, just apply (a finite number of times) the constructors.

```
Lemma Y_assigned : Assigned_in  Y factorial_Z_program.
Proof.
 unfold factorial_Z_program.
 constructor 3 (* apply Assigned_seq2 *).
 constructor 2 (* apply Assigned_seq1 *) .
 constructor 1 (* apply Assigned_assign *).
Qed.
```

```
(* Using hints and auto *)
Hint Constructors Assigned_in.

Lemma X_assigned : Assigned_in X factorial_Z_program.
Proof.
 unfold factorial_Z_program;auto.
Qed.
```

```
(* Using hints and auto *)
Hint Constructors Assigned_in.

Lemma X_assigned : Assigned_in X factorial_Z_program.
Proof.
 unfold factorial_Z_program;auto.
Qed.

Lemma Z_unassigned : ~(Assigned_in Z factorial_Z_program).
intro H.
```

1 subgoal

H : Assigned_in Z factorial_Z_program

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

False

```
(* Using hints and auto *)
Hint Constructors Assigned_in.

Lemma X_assigned : Assigned_in X factorial_Z_program.
Proof.
 unfold factorial_Z_program;auto.
Qed.

Lemma Z_unassigned : ~(Assigned_in Z factorial_Z_program).
intro H.
1 subgoal
  H : Assigned_in Z factorial_Z_program
  ============================
   False
(* ???????????? *)
Abort.
```

A relation already used in previous lectures

The $\leq$ relation on `nat` is defined by the means of an inductive predicate :

```
Inductive le (n : nat) : nat -> Prop :=
   | le_n : le n n
   | le_S : forall m : nat, le n m -> le n (S m)
```

The proposition `(le n m)` is denoted by `n <= m`.

Reasoning with inductive predicates

Use constructors as introduction rules.

```
Lemma le_n_plus_pn : forall n p: nat, n <= p + n.
Proof.
 induction p;simpl.
```

2 subgoals

$n : nat$

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

$n <= n$

subgoal 2 is:

$n <= S\ (p + n)$

```
 constructor 1.
```

1 subgoal

$n : nat$
$p : nat$
$IHp : n <= p + n$
$==========================$
$n <= S\ (p + n)$

```
constructor 2;assumption.
Qed.
```

The induction principle for le

```
le_ind
    : forall (n : nat) (P : nat -> Prop),
      P n ->
      (forall m : nat, n <= m -> P m -> P (S m)) ->
      forall p : nat, n <= p -> P p
```

In order to prove that for every $p \geq n$, $P\ p$, prove :

- $P\ n$
- for any $m \geq n$, if $P\ m$ holds, then $P\ (\mathtt{S}\ m)$ holds.

Use induction or destruct as elimination tactics.

```
 Lemma le_plus : forall n m, n <= m ->
                  exists p:nat, p+n = m
                  (* P n *).
Proof.
 intros n m H.
```

1 subgoal

$n : nat$

$m : nat$

$H : n <= m$

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

$exists\ p : nat,\ p + n = m$

induction H.

2 subgoals

n : nat
============================
exists p : nat, p + n = n
(P n *)*

subgoal 2 is:
exists p : nat, p + n = S m

```
exists 0;trivial.
```

1 subgoal

$n : nat$
$m : nat$
$H : n <= m$
$IHle : exists\ p : nat,\ p + n = m$ (* $P\ m$ *)
==============================
$exists\ p : nat,\ p + n = S\ m$ (* $P\ (S\ m)$ *)

```
 destruct IHle as [q Hq]; exists (S q);
   simpl;rewrite Hq;trivial.
Qed.
```

The `inversion` tactic

How to prove that :

```
Lemma foo : ~(1 <= 0).
```

The `inversion` tactic

How to prove that :

```
Lemma foo : ~(1 <= 0).
Proof.
intro h.
inversion h.
Qed.
```

The `inversion` tactic derives all the necessary conditions to an inductive hypothesis. If no condition can realize this hypothesis, the goal is proved by *ex falso quod libet*.

```
Lemma le_n_0 : forall n, n <= 0 -> n = 0.
Proof.
 intros n H;inversion H.
```

1 subgoal

n : nat
H : n <= 0
H0 : n = 0
============================
0 = 0

```
trivial.
Qed.
```

Some other examples

```
Lemma Assigned_inv1 : forall v w e,
   Assigned_in v (assign w e) ->
   v=w.
Proof.
 intros v w e H; inversion H. ...

Lemma Assigned_inv2 : forall v s1 s2,
   Assigned_in v (sequence s1 s2) ->
   Assigned_in v s1 \/ Assigned_in v s2.
Proo.
intros v s1 s2 H; inversion H. ...
```

Inductive predicates and functions

Let us consider again our last failure :

```
Lemma Z_unassigned :
   ~(Assigned_in Z factorial_Z_program).
intro H.
```

1 subgoal

H : Assigned_in Z factorial_Z_program

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

False

Inductive predicates and functions

Let us consider again our last failure :

```
Lemma Z_unassigned :
   ~(Assigned_in Z factorial_Z_program).
intro H.
1 subgoal
  H : Assigned_in Z factorial_Z_program
  ============================
   False
inversion H ???????????
```

Inductive predicates and functions

Let us consider again our last failure :

```
Lemma Z_unassigned :
   ~(Assigned_in Z factorial_Z_program).
intro H.
```

1 subgoal
  H : Assigned_in Z factorial_Z_program
  ============================
   False

```
inversion H ??????????? NO !
(gives two unreadable subgoals)
```

```
Require Import Bool.
Definition var_eqb (v w : toy_Var) :=
match v,w with  X, X => true
              | Y, Y => true
              | Z, Z => true
              | _, _ => false
end.

Fixpoint assigned_inb (v:toy_Var)(s:toy_Statement) :=
 match s with
            | assign w _ => var_eqb v w
            | sequence s1 s2 =>
                assigned_inb v s1 || assigned_inb v s2
            | simple_loop e s => assigned_inb v s
 end.
```

Bridge lemmas

```
Lemma Assigned_In_OK : forall v s,
  Assigned_in v s ->
  assigned_inb v s = true.
Proof.
 intros v s H;induction H;simpl;...

Lemma Assigned_In_OK_R :
forall v s, assigned_inb v s = true ->
            Assigned_in v s.
Proof.
 induction s;simpl.
  ...
```

```
Lemma Z_unassigned : ~(Assigned_in Z factorial_Z_program).
Proof.
intro H;assert(H0 :=  Assigned_In_OK  _ _ H).
```

1 subgoal

H : Assigned_in Z factorial_Z_program
H0 : assigned_inb Z factorial_Z_program = true
============================
False

```
simpl in H0;discriminate H0.
Qed.
```

Let us consider again two aspects of $\leq$:

```
Inductive le (n : nat) : nat -> Prop :=
   | le_n : n <= n
   | le_S : forall m : nat, le n m -> le n (S m)
```

```
The term  (le n m) is denoted by  n <= m.
```

```
Fixpoint leb n m : bool :=
   match n, m with
   |O, _ => true
   |S i, S j => leb i j
   | _, _ => false
end.
```

```
Eval compute in leb 5 45.
```

= true : bool

```
Lemma L5_45 : 5 <= 45.
Proof.
 repeat constructor.
Qed.
```

```
Eval compute in leb 5 45.
```

$= \mathit{true} : \mathit{bool}$

```
Lemma L5_45 : 5 <= 45.
Proof.
 repeat constructor.
Qed.
```

Just try Print L5_45. !

```
Lemma le_trans :
  forall n p q, n <= p -> p <= q -> n <= q.
Proof.
```

```
Lemma le_trans :
  forall n p q, n <= p -> p <= q -> n <= q.
Proof.
```

We recognize the scheme :

```
p <= q -> P q where  P q is n <= q.
```

Thus, the base case is n $<=$ p and the inductive step is

```
forall q, p <= q -> n <= q  -> n <= S q.
```

```
intros n p q H H0;induction H0.
```

2 subgoals

$n : nat$

$p : nat$

$H : n <= p$

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

$n <= p \ldots$

```
assumption.
```

1 subgoal

n : nat
p : nat
H : $n <= p$
m : nat
$H0$: $p <= m$
$IHle$: $n <= m$
============================
$n <= S\ m$

```
constructor;assumption.
Qed.
```

The tactic constructor tries to make the goal progress by applying a constructor. Constructors are tried in the order of the inductive type definition.

```
Lemma le_Sn_Sp_inv:  forall n p,  S n <= S p ->  n <= p.
intros n p H;inversion H.
```

2 subgoals

$n : nat$

$p : nat$

$H : S\ n <= S\ p$

$H1 : n = p$

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

$p <= p \ldots$

```
constructor.
```

1 subgoal

```
n : nat
p : nat
H : S n <= S p
m : nat
H1 : S n <= p
H0 : m = p
============================
 n <= p
```

```
apply le_trans with (S n);
  [repeat constructor|assumption].
```

`le` or `leb` ?

We can build a bridge between both aspects by proving the following theorems :

```
Lemma le_leb_iff : forall n p, n <= p <-> leb n p=true.

Lemma lt_leb_iff : forall n p, n < p <-> leb p n = false.
(* Proofs left as exercise *)
```

```
Lemma L: 0 <= 47.
Proof.
 rewrite  le_leb_iff.
```

1 subgoal

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

leb 0 47 = true

```
trivial.
Qed.
```

```
Lemma leb_Sn_n : forall n p, leb  n (n + p)= true.
Proof.
 intros n p;rewrite <- le_leb_iff.
```

1 subgoal

n : *nat*
p : *nat*
============================
$n <= n + p$

```
 SearchPattern (_ <= _ + _).
 apply le_plus_l;auto.
Qed.
```

Inductive definitions and functions

It is sometimes very difficult to represent a function `f : A -> B` as a *Coq* function, for instance because of the :

- Undecidability (or hard proof) of termination
- Undecidability of the domain characterization

This situation often arises when studying the semantic of programming languages.

In that case, describing functions as inductive relations is really efficient.

What you think is not what you get

An odd alternative definition of le :

```
Inductive alter_le (n : nat) : nat -> Prop :=
| alter_le_n : alter_le n n
| alter_le_S : forall m : nat, alter_le n m ->
                                alter_le n (S m)
| alter_dummy : alter_le n (S n).
```

What you think is not what you get

An odd alternative definition of le :

```
Inductive alter_le (n : nat) : nat -> Prop :=
| alter_le_n : alter_le n n
| alter_le_S : forall m : nat, alter_le n m ->
                                alter_le n (S m)
| alter_dummy : alter_le n (S n).
```

What about the proof of :

```
Lemma alter_le_trans :  forall x y z,
  alter_le x y -> alter_le y z -> alter_le z z.
```

What you think is not what you get

An odd alternative definition of le :

```
Inductive alter_le (n : nat) : nat -> Prop :=
| alter_le_n : alter_le n n
| alter_le_S : forall m : nat, alter_le n m ->
                              alter_le n (S m)
| alter_dummy : alter_le n (S n).
```

What about the proof of :

```
Lemma alter_le_trans :  forall x y z,
  alter_le x y -> alter_le y z -> alter_le z z.
```

Well, it does not behave as nicely as expected.

A more abstract example

```
Section transitive_closures.
 Definition relation (A : Type) := A -> A -> Prop.
 Variables (A : Type)(R : relation A).

(* the transitive closure of R is the least
relation ... *)
Inductive clos_trans : relation A :=
  (* ... that contains R *)
  | t_step : forall x y : A, R x y -> clos_trans x y
  (* ... and is transitive *)
  | t_trans : forall x y z : A,
    clos_trans x y -> clos_trans y z
                   -> clos_trans x z.
```

If some relation R is transitive, then its transitive closure in included in R :

```
Hypothesis Rtrans :
   forall x y z, R x y -> R y z -> R x z.

Lemma trans_clos_trans :  forall a1 a2,
                           clos_trans a1 a2 -> R a1 a2.
Proof.
intros a1 a2 H; induction H.
```

2 subgoals

$x : A$

$y : A$

$H : R\ x\ y$

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

$R\ x\ y\ \ \ldots$

```
exact H.
```

```
x : A
y : A
z : A
H : clos_trans x y
H0 : clos_trans y z
IHclos_trans1 : R x y
IHclos_trans2 : R y z
============================
 R x z
apply Rtrans with y;  assumption.
Qed.
```

```
End transitive_closures.
Check trans_clos_trans.
```

trans_clos_trans
: forall (A : Type) (R : relation A),
(forall x y z : A, R x y -> R y z -> R x z) ->
forall a1 a2 : A, clos_trans A R a1 a2 -> R a1 a2

```
End transitive_closures.
Check trans_clos_trans.
```

trans_clos_trans
: forall (A : Type) (R : relation A),
(forall x y z : A, R x y -> R y z -> R x z) ->
forall a1 a2 : A, clos_trans A R a1 a2 -> R a1 a2

```
Implicit Arguments clos_trans [A].
Implicit Arguments trans_clos_trans [A].
Check (trans_clos_trans le le_trans).
```

trans_clos_trans nat le le_trans
: forall a1 a2 : nat, clos_trans le a1 a2 -> a1 <= a2

Advice for crafting useful inductive definitions

- Constructors are “axioms” : they should be intuitively true...
- Constructors should as often as possible deal with mutually exclusive cases, to ease proofs by induction ;
- When an argument always appears with the same value, make it a parameter
- Test your predicate on negative and positive cases !

Logical connectives as inductive definitions

Most logical connectives are defined using inductive types :

- Conjunction $\wedge$
- Disjunction $\vee$
- Existential quantification $\exists$
- Equality
- Truth and False

Notable exceptions : implication, negation.

Let us revisit the 3th and 4th lectures.

Logical connectives : conjunction

Conjunction is a pair :

```
Inductive and (A B : Prop) := conj : A -> B -> and A B.
```

- Term `(and A B)` is denoted `(A` $\wedge$ `B)`.
- Prove a conjunction goal with the `split` tactic (generates two subgoals).
- Use a conjunction hypothesis with the `destruct as [...]` tactic.

Logical connectives : disjunction

Disjunction is a two constructors inductive :

```
Inductive or (A B : Prop) : Prop :=
|or_introl : A -> or A B | or_intror : B -> or A B.
```

- Term `(or A B)` is denoted `(A` $\vee$ `B)`.
- Prove a disjunction with the `left`, `right` tactics (choose the side to prove).
- Use a conjunction hypothesis with the `case` or `destruct as [...|...]` tactics.

Logical connectives : existential quantification

Existential quantification is a pair :

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P.
```

- The term ex A (fun x => P x) is denoted exists x, P x.
- Prove an existential goal with the `exists` tactic.
- Use an existential hypothesis with the `destruct as [...]` tactic.

Equality

The built-in (predefined) equality relation in *Coq* is a parametric inductive type :

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
   refl_equal : eq A x x.
```

- Term `eq A x y` is denoted $(x = y)$
- The induction principle is :

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
P x -> forall y : A, x = y -> P y
```

Equality

- Use an equality hypothesis with the `rewrite [<-]` tactic (uses `eq_ind`)
- Remember equality is computation compliant !

  ```
  Goal 2 + 2 = 4. apply refl_equal. Qed.
  ```

  Because + is a program.
- Prove trivial equalities (modulo computation) using the `reflexivity` tactic.

Truth

The "truth" is a proposition that can be proved under any assumption, in any context. Hence it should not require any argument or parameter.

```
Inductive True : Prop := I : True.
```

Its induction principle is :

```
True_ind : forall P : Prop, P -> True -> P
```

which is not of much help...

Falsehood

Falsehood should be a proposition of which no proof can be built (in empty context).
In *Coq*, this is encoded by an inductive type with no constructor :

```
Inductive False : Prop :=
```

coming with the induction principle :

```
False_ind : forall P : Prop, False -> P
```

often referred to as *ex falso quod libet*.

- To prove a False goal, often apply a negation hypothesis.
- To use a H : False hypothesis, use destruct H.

Specifying programs with inductive predicates

Programs are computational objects.
Inductive types provide structured specifications.
How to get the best of both worlds ?

Specifying programs with inductive predicates

Programs are computational objects.
Inductive types provide structured specifications.
How to get the best of both worlds ?
By combining programs with inductive specifications.