# Inductive properties (2)

Assia Mahboubi, Pierre Castéran, Yves Bertot
Paris, Beijing, Bordeaux, Suzhou

16 novembre 2011

# Inversion Techniques

Let us consider the following theorem.

```
Lemma le_n_0 : forall n:nat, n <= 0 -> n = 0.
Proof.
 intros n H; induction H.
 two subgoals :

  n : nat
  ============================
   n = n
reflexivity.
```

*1 subgoal*

```
  n : nat
  m : nat
  H : n <= m
  IHle : n = m (* P m *)
  ============================
   n = S m  (* P (S m) *)
```

```
Abort.
```

The induction H tactic call applied the induction principle le_ind with $P := fun\ m : nat \Rightarrow n = m$.

# How did we solve this problem in good old times ?

We could prove the following "inversion lemma" (a kind of reciprocal of the constructors).

```
Lemma le_inv : forall n p: nat,
  n <= p ->
  n = p \/ exists q:nat, p = S q /\ n <= q.
Proof.
  intros n p H; destruct  H.
```

*2 subgoals*

*n : nat*

*==========*

*n = n \/ (exists q : nat, n = S q /\ n <= q)*

```
left;reflexivity.
```

*1 subgoal*

*n : nat*
*m : nat*
*H : n <= m*
*==========*
*n = S m \/ (exists q : nat, S m = S q /\ n <= q)*

```
right; exists m;split;trivial.
Qed.
```

Note that le_inv is an expression of the minimality of le, with explicit equalities that can be used with injection and discriminate.

# Let's come back to our initial lemma

```
Lemma le_n_0_old_times : forall n:nat, n <= 0 -> n = 0.
Proof.
 intros n H;
  destruct (le_inv _ _ H) as [H0 | [q [Hq Hq0]]].
```

*2 subgoals*

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

*...*

```
assumption.
```

```
1 subgoal
 n : nat
  H : n <= 0
  q : nat
  Hq : 0 = S q
  Hq0 : n <= q
  ============================
   n = 0
 discriminate Hq.
Qed.
```

# The `inversion` tactic

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*. See also : inversion_ clear

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

# The inversion tactic

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*. See also : inversion_ clear

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

```
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.
```

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

*2 subgoals*

*n : nat*
*p : nat*
*H : S n <= S p*
*H1 : n = p*
*============================*
*p <= p*

*...*

```
constructor.
```

*1 subgoal*

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

```
Require Import Le.
apply le_trans with (S n); repeat constructor; assumption.
Qed.
```

# Comparison with other kinds of predicate definitions

Let us consider le again. Several other definitions can be given for this mathematical concept.

# Comparison with other kinds of predicate definitions

Let us consider le again. Several other definitions can be given for this mathematical concept.

First, we could use the `plus` function.

```
Definition Le (n p : nat) : Prop :=
     exists q:nat, q + n = p.
```

# Comparison with other kinds of predicate definitions

Let us consider le again. Several other definitions can be given for this mathematical concept.
First, we could use the `plus` function.

```
Definition Le (n p : nat) : Prop :=
     exists q:nat, q + n = p.
```

We can also give a recursive predicate :

```
Fixpoint LE (n p: nat): Prop :=
 match n, p with 0, _ => True
                 | S _, 0 => False
                 | S n', S p' => LE n' p'
 end.
```

Both definitions are equivalent to *Coq*'s le (exercise).

# Predicates and boolean functions

Let us consider the following function :

```
Fixpoint leb n m : bool :=
   match n, m with
   |0, _ => true
   |S i, S j => leb i j
   | _, _ => false
end.
```

# le or leb ?

```
Compute  leb 5 45.
```
*= true : bool*

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

# le or leb ?

```
Compute  leb 5 45.
```
*= true : bool*

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

Just try Print L5_45. !

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
reflexivity.
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.
```

# 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 is 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 ...*

```
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*

# 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.

```
Definition odd n := ~even n.

Inductive syracuse_steps : nat -> nat -> Prop :=
 done : syracuse_steps  1 1
|even_case : forall n p,even n ->
    syracuse_steps (div2 n) p ->
    syracuse_steps  n (S p)
|odd_case : forall n p , odd n ->
            syracuse_steps (S(n+n+n))  p ->
            syracuse_steps n (S p).
```

## Exercise

Prove the proposition syracuse_steps 5 6.

# 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.

# Specifying programs with inductive predicates

Let us consider a datatype for comparison w.r.t. some decidable total order. This type already exists in the Standard Library.

```
Inductive Comparison : Type := Lt | Eq | Gt.
```

We can easily specify whether some value of this type is consistent with an arithmetic inequality, through a three place predicate.

```
Inductive compare_spec (n p:nat) : Comparison -> Type :=
| lt_spec : forall Hlt : n < p, compare_spec n p Lt
| eq_spec : forall Heq : n = p, compare_spec  n p Eq
| gt_spec : forall Hgt : p < n, compare_spec n p Gt.
```

We can specify whether some comparison function is correct :

```
Definition cmp_correct (cmp : nat -> nat -> Comparison) :=
   forall n p, compare_spec n p (cmp n p).
```

In order to understand specifications like `compare_spec`, let us open a section :

```
Section On_compare_spec.
 Variable cmp :  nat -> nat -> Comparison.
 Hypothesis cmpP : cmp_correct cmp.
```

# How to use compare_spec

Let us consider a goal of the form $P\ n\ p\ (cmp\ n\ p)$ where $P$ : nat→nat→Comparison→Prop.

A call to the tactic destruct (cmpP n p) produces three subgoals :

```
Hlt : n < p
================================
P n p Lt
```

```
Heq : n = p
================================
P n p Eq
```

```
Hgt : p < n
================================
P n p Gt
```

# Example

Let us define functions for computing the greatest [lowest] of tho numbers :

```
Definition maxn n p :=
    match cmp n p with Lt => p | _ => n end.

Definition minn  n p :=
    match cmp n p  with Lt => n | _ => p end.
```

Proofs of properties of `maxn` and `minn` can use this pattern, which will give values to maxn $n$ $p$, and generate hypotheses of the form $n < p$, $n = p$, and $p < n$.

```
Lemma le_maxn: forall n p, n <= maxn n p.
Proof.
intros n p; unfold maxn;destruct (cmpP n p).
```

*3 subgoals*

*cmpP : cmp_correct cmp*
*...*
*Hlt : n < p*
*==========*
*n <= p*

*subgoal 2 is:*
*n <= n*
*subgoal 3 is:*
*n <= n*

Ecah one of the three subgoals is solved with auto with arith.

The following proofs use the same pattern :

```
Lemma maxn_comm :  forall n p, maxn n p = maxn p n.
Proof.
 intros n p; unfold maxn;
 destruct (cmpP n p), (cmpP p n); omega.
Qed.

Lemma maxn_le: forall n p q,
      n <= q -> p <= q -> maxn n p <= q.
Proof.
intros n p; unfold maxn; destruct (cmpP n p);
          auto with arith.
Qed.
```

```
Lemma min_plus_maxn : forall n p,
    minn n p + maxn n p = n + p.
Proof.
intros n p; unfold maxn, minn; destruct (cmpP n p);
      auto with arith.
Qed.
```

```
Definition compare_rev (c:Comparison) :=
 match c with
 | Lt => Gt
 | Eq => Eq
 | Gt => Lt
 end.


Lemma cmp_rev : forall n p,
   cmp n p = compare_rev (cmp p n).
Proof.
 intros n p; destruct (cmpP n p);destruct (cmpP p n) ;
 trivial;try discriminate;intros; elimtype False; omega.
Qed.
```

```
Lemma cmp_antiym : forall n p,
    cmp n p = cmp p n -> n = p.
Proof.
 intros n p;rewrite cmp_rev;
  destruct (cmpP p n);auto ;try discriminate.
Qed.
```

Notice that all the proofs above use only the *specification* of a comparison function and not a concrete definition.

We are now able to provide an implementation of a comparison function, and prove its correctness :

```
End On_compare_spec.

Fixpoint compare (n m:nat) : Comparison :=
 match n, m with | 0,0 => Eq
                 | 0, S _ => Lt
                 | S _, 0 => Gt
                 | S p, S q => compare p q
 end.
```

```
Lemma compareP : cmp_correct compare.
Proof.
 red;induction n;destruct p;simpl;auto;
 try (constructor;auto with arith).
 destruct (IHn p);constructor;auto with arith.
Qed.

Check maxn_comm _ compareP.
```

*: forall n p : nat, maxn compare n p = maxn compare p 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 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).
```

The third constructor is useless ! It may increase the size of the proofs by induction.

# 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 !

# A last example : The toy programming language

```
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.
Proof.
intros v s1 s2 H; inversion H. ...
```

We can also define a boolean function for testing equality on variables :

```
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.
```

We define a boolean test for the "assigned" property :

```
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.
  ...
```

# A small program

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

```
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)))))))).
```

```
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.
```