Inductive properties

Assia Mahboubi

9 juin 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 predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
```

Inductive predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :
```

Inductive predicates

“The friend of my friend is my friend” : who’s friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :  forall x, is_friend_of x x
```

Inductive predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :  forall x, is_friend_of x x
|No_hypocrisy :
```

Inductive predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :  forall x, is_friend_of x x
|No_hypocrisy :  forall x y,
         is_friend_of x y -> is_friend_of y x
```

Inductive predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :  forall x, is_friend_of x x
|No_hypocrisy :  forall x y,
         is_friend_of x y -> is_friend_of y x
|Proverb :
```

Inductive predicates

"The friend of my friend is my friend" : who's friend with whom ?

```
Variable People : Type.
Inductive is_friend_of : People -> People -> Prop :=
|I_am_my_own_friend :  forall x, is_friend_of x x
|No_hypocrisy :  forall x y,
         is_friend_of x y -> is_friend_of y x
|Proverb :  forall x y z,
     is_friend_of x y -> is_friend_of y z ->
     is_friend_of x z.
```

Inductive predicates

```
Variable MrCat MrDog : People.
Hypothesis not_friends : ~(is_friend_of MrCat MrDog).

Lemma no_common_friend : forall p : People,
  is_friend_of p MrDog -> ~(is_friend_of p MrCat).
Proof.
intros p hpMrDog hpMrCat.
apply not_friends.
apply Proverb with p.
  apply No_hypocrisy.
  trivial.
trivial.
Qed.
```

Inductive predicates

The same construction is useful to define closures.
Here is the transitive closure of a relation :

```
Definition relation (A : Type) := A -> A -> Prop.

Variables (A : Type)(R : relation A).

Inductive clos_trans : relation A :=
  | t_step : forall x y : A, R x y -> clos_trans x y
  | t_trans : forall x y z : A,
    clos_trans R x y -> clos_trans y z
                              -> clos_trans x z.
```

Inductive predicates

Reason by induction on these inductive predicates.

```
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.
  exact H.
apply Rtrans with y.
  assumption.
assumption.
Qed.
```

A relation already used in previous lectures

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

```
Print le.
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`.

Inductive definitions and functions

The `le` predicate can be seen as the inductive description of the boolean test :

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

Functional an inductive predicates have respective assets and drawbacks that should be evaluated at formalization time.

As usual, the choice of data structures matters !

Inductive definitions and functions

However, 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.

The constructor tactic

```
Lemma le_trans : forall x y z,
  x <= y -> y <= z -> x <= z.
Proof.
move=> x y z hxy hyz.
induction hyz.
  assumption.
constructor.
assumption.
Qed.
```

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

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.

What you think is not what you get

Be careful at definition time, you might otherwise soon be stuck.

```
Inductive alter_clos_trans :
                        (relation A) -> (relation A) :=
  | alt_t_step : forall (R : relation A) x y,
                       R x y -> alter_clos_trans R x y
  | alt_t_trans : forall (R : relation A) x y z,
    alter_clos_trans R x y -> alter_clos_trans R y z
                          -> alter_clos_trans R x z.
```

What you think is not what you get

Be careful at definition time, you might otherwise soon be stuck.

```
Inductive alter_clos_trans :
                        (relation A) -> (relation A) :=
  | alt_t_step : forall (R : relation A) x y,
                       R x y -> alter_clos_trans R x y
  | alt_t_trans : forall (R : relation A) x y z,
    alter_clos_trans R x y -> alter_clos_trans R y z
                          -> alter_clos_trans R x z.
```

What about the proof of :

```
Lemma alter_trans_clos_trans :  forall a1 a2,
  alter_clos_trans R a1 a2 -> R a1 a2.
```

What you think is not what you get

Be careful at definition time, you might otherwise soon be stuck.

```
Inductive alter_clos_trans :
                        (relation A) -> (relation A) :=
  | alt_t_step : forall (R : relation A) x y,
                       R x y -> alter_clos_trans R x y
  | alt_t_trans : forall (R : relation A) x y z,
    alter_clos_trans R x y -> alter_clos_trans R y z
                          -> alter_clos_trans R x z.
```

What about the proof of :

```
Lemma alter_trans_clos_trans :  forall a1 a2,
  alter_clos_trans R a1 a2 -> R a1 a2.
```

Well, it does not behave as nicely as expected.

What you think is not what you get

One more odd alternative definition :

```
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 (
| alter_dummy : alter_le n (S n).
```

What you think is not what you get

One more odd alternative definition :

```
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 (
| 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

One more odd alternative definition :

```
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 (
| 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.

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 4th lecture.

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 ∧ 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 ∨ 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.
```

- Term `(ex_intro A P x Px)` 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.
```

  Beacuse + 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 `elim H`.

Specifying programs with inductive predicates

Programs are computational objects.

Specifying programs with inductive predicates

Programs are computational objects.
Inductive types provide structured specifications.

Specifying programs with inductive predicates

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

Specifying programs with inductive predicates

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

Specifying programs with inductive predicates

To program a function `maxn`, computing the maximum of two `nat`, you might consider writing something like :

```
Definition maxn n m := if (ltb m n) then n else m.
```

Specifying programs with inductive predicates

To program a function `maxn`, computing the maximum of two `nat`, you might consider writing something like :

```
Definition maxn n m := if (ltb m n) then n else m.
```

and then prove :

```
Lemma add_sub_maxn : forall m n, m + (n - m) = maxn m n.
```

since on natural numbers, $n - m = 0$ when $m > n$.

Specifying programs with inductive predicates

To program a function maxn, computing the maximum of two nat, you might consider writing something like :

```
Definition maxn n m := if (ltb m n) then n else m.
```

and then prove :

```
Lemma add_sub_maxn : forall m n, m + (n - m) = maxn m n.
```

since on natural numbers, $n - m = 0$ when $m > n$.

We propose a way to reason comfortably on programs written using these boolean tests.

Specifying programs with inductive predicates

Here is how the boolean comparison can be programmed :

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

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

Specifying programs with inductive predicates

Here is how the boolean comparison can be programmed :

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

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

They satisfy :

```
Lemma ltb_lebn : forall n m, ltb n m = negb (leb m n).
```

which is shown by induction on the first argument.

Specifying programs with inductive predicates

We can specify the respective values that `ltb` and `leb` can take by defining the inductive specification :

```
Inductive leb_xor_gtb (m n : nat): bool -> bool -> Type :=
  | LebNotGtb : (leb m n = true) ->
                              leb_xor_gtb m n true false
  | GtbNotLeb : (ltb n m = true) ->
                              leb_xor_gtb m n false true.
```

and proving the lemma :

```
Lemma lebP : forall m n,
                leb_xor_gtb m n (leb m n) (ltb n m).
```

Specifying programs with inductive predicates

Now let us see how this specification is used. The script :

```
Lemma add_sub_maxn : forall m n, m + (n - m) = maxn m n.
Proof.
intros m n; unfold maxn.
```

generates the subgoal

```
m : nat
n : nat
============================
m + (n - m) = (if ltb n m then m else n)
```

Specifying programs with inductive predicates

Now the tactic :

```
case (lebP m n); intros h.
```

generates the two subgoals :

```
m : nat
n : nat
============================
 leb m n = true -> m + (n - m) = n
```

and :

```
m : nat
n : nat
============================
 ltb n m = true -> m + (n - m) = m
```