Inductive properties

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

July 2012

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.

Simple inductive definitions

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

- The first line expresses that we are defining a predicate
- The second and third lines give ways to prove instances of this predicate
- `even0` and `evenS` can be used like theorems
  - They are called *constructors*
- `even`, `even0`, `evenS` and `even_ind` are defined by this definition

Using constructors as theorems

```
Check evenS.
evenS : forall p : nat, even p -> even (S (S p))

Lemma four_even : even 4.
apply evenS.
  ============================
   even 2
apply evenS.
  ============================
   even 0
apply even0
Proof completed
```

Meaning of constructors

- The arrow in constructors is an implication
- Goal-directed proof works by backward chaining
- the operational meaning in proofs walks the arrow backwards
  - Unlike the symbol => in function definitions
  - premises of constructors should be "simpler" than conclusions

Meaning of the inductive definition

- Not just any relation so that the constructors are verified
- The smallest one
- For all other predicate P so that formulas similar to constructors hold, the inductive predicate implies P

```
forall P : nat -> Prop,
  (P 0) -> (* as in even0 *)
  (forall n : nat, P n -> P (S (S n))) -> (* as in evenS *)
  forall k : nat, even k -> P k
```

- This is expressed by `even_ind`

Meaning of the inductive definition

```
forall P : nat -> Prop,
  (P 0) -> (* as in even0 *)
  (forall n : nat, P n -> P (S (S n))) -> (* as in evenS *)
  forall k : nat, even k -> P k
```

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

Example proof with induction principle

```
Lemma even_double :
   forall n, even n -> exists k, n = 2 * k.
intros n H.
  ============================
   exists k, 0 = 2 * k
induction H as [|n H' IHn].
```

- Patterned after constructors
- Induction hypotheses for premises that are instances of the inductive predicate

Goals of proof by induction

```
============================
 exists k : nat, 0 = 2 * k

H' : even n
IHeven : exists k : nat, n = 2 * k
============================
 exists k : nat,  S (S n) = 2 * k
(* rest of proof left as an exercise. *)
```

- hypothesis `H` was `even n`
- three copies of `exist k, n = 2 * k` have been generated
  - `n` has been replaced by `0`, `n`, and `S (S n)`
  - values taken from the constructors of even

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`.
`n` is called a *parameter* of the previous definition.
It is used in a stable manner throughout the definition: every occurrence of `le` has `n` as first argument

Reasoning with inductive predicates

Use constructors as introduction rules.

- To *prove* facts based on the predicate

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

  n : nat
  ============================
   n <= 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.

- To *use* facts based on the predicate

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

```
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

...

```
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.

Constructing induction principles

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

- Parameterless arity : `nat -> Prop`
- Parameter-bound predicate : `le n`
- quantify over parameters, then a predicate with parameterless arity
  `forall n : nat, forall P : nat -> Prop,`
- Process each constructor, add an epilogue

Process each constructor

- Abstract over the parameter-bound predicate
  - for `le_n : le n n`
    fun X : nat -> Prop => X n
  - for `le_S : forall n, le n m -> le n (S m)`
    fun X => forall n, X m -> X (S m)
- Duplicate instances of `X` in premises, with a new variable
  - for `le_n : le n n`
    fun X Y : nat -> Prop => X n
  - for `le_S : forall n, le n m -> le n (S m)`
    fun X Y => forall n, Y m -> X m -> X (S m)
- Instanciate `X` with `P`, `Y` with `le n` (the parameter-bound predicate)

Adding an epilogue

- Express that every object that satisfies the parameter-bound predicate also satisfies the property `P`
- `forall m:nat, le n m -> P m`

Logical connectives as inductive definitions

Most logical connectives are defined using inductive types:

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

Notable exceptions: implication, negation.

Let us revisit the 3rd and 4th lectures.

Logical connectives: conjunction

Conjunction is a pair:

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

and_ind : forall A B P : Prop,
   (A -> B -> P) -> and A B -> P
```

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

Logical connectives: disjunction

Disjunction is a two constructor inductive predicate:

```
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)`.
- `left` and `right` tactic apply `or_introl` or `or_intror`
- Use a conjunction hypothesis with the `case` or `destruct as [...|...]` tactics.
- Two goals correspond to two constructors

Logical connectives: existential quantification

Existential quantification is also 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.
- `exists` applies `ex_intro`
- 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.

A toy programming language

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 (ident : nat).
```

or

```
Require Import String.
Inductive toy_Var : Set := toy_Var (ident: 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 expr := Econst (i:nat) |
                  Evar (v:toy_Var) |
                  Eop (op:toy_Op) (e1 e2: expr)
```

Statements

```
Inductive cmd :=
  | (* x = e *)
     Cassign (v:toy_Var)(e:expr)
  | (* s ; s1 *)
     Cseq (s s1: cmd)
  | (* execute e repetitions of s *)
    Csimple_loop (e:expr)(s : cmd).
```

```
Definition factorial_Z_program :=
Cseq (Cassign X (Econst 0))
 (Cseq
   (Cassign Y (Econst 1))
   (Csimple_loop (Evar Z)
    (Cseq
      (Cassign X
         (toy_op toy_plus (Evar X) (Econst 1)))
      (Cassign Y
         (toy_op toy_mult (Evar Y) (Evar X)))))).
```

We can define the predicate "the variable v appears in the expression e":

```
Inductive occurs_in  (v:toy_Var): expr -> Prop :=
|Occ_var : occurs_in v (Evar v)
|Occ_op1 : forall op e1 e2, occurs_in v e1 ->
                             occurs_in v (toy_op op e1 e2)
|Occ_op2 : forall op e1 e2, occurs_in v e2 ->
                             occurs_in v (toy_op op e1 e2).
```

Constructor names are displayed in red.

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

```
Inductive Assigned_in (v:toy_Var): cmd->Prop :=
| Assigned_assign : forall e, Assigned_in v (Cassign v e)
| Assigned_seq1 : forall s1 s2,
                    Assigned_in v s1 ->
                    Assigned_in v (Cseq s1 s2)
| Assigned_seq2 : forall s1 s2,
                    Assigned_in v s2 ->
                    Assigned_in v (Cseq s1 s2)
| Assigned_loop : forall e s,
                    Assigned_in v s ->
                    Assigned_in v (Csimple_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.
```