Inductive properties

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

16 novembre 2011

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`

Minimality and induction principle

- The induction principle can be derived from minimality
  - Tip : proving `P n /\ even n` using minimality give induction
- For every true statement of `even n`, there exists a proof done solely with constructors
- The induction principle can be use to establish consequences from the inductive predicate

Example proof with induction principle

```
Lemma even_double :
   forall n, even n -> exists k, n = 2 * k.
intros n H.
```

$$\overline{}$$

exists k, $0 = 2 * k$

```
induction H.
```

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

Goals of proof by induction

$$\overline{\overline{\hspace{20em}}}$$

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

n : nat
*IHeven : exists k : nat, n = 2 * k*

$$\overline{\overline{\hspace{20em}}}$$

*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 * n` 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.

```
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 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

$n : nat$

$p : nat$

$H : n <= p$

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

$n <= p \ldots$

```
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].
```

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 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.
```

- 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.

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 (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)
```

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)))))).
```

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.
```