Simple proofs about recursive functions

Yves Bertot

August 2009

Reasoning about programs and behavior

- Proofs by induction to cover the whole input type
- Reasoning by cases on function inputs
- Getting rid of inconsistent assumptions
- Using the injectivity of datatype constructors
- Using specialized induction principles

Proofs by induction

- Goal of the form `C x`, where `x` is an integer
- Tactic : `induction n as [ | p IHp]`
- The system creates two goals corresponding to cases
  - `C 0`
  - `C (S p)`
- In the second goal, a fact `IHp` is added to the context with statement `C p`

Main guideline

Reason on executions of functions

- Reason by induction when working on a recursive function
- Induction on the argument where recursion occurs

```
Fixpoint fact (n:nat) : nat :=
  match n with
  | 0 => 1
  | S p => S p * fact p
  end.

Lemma factp : forall n, 0 < fact n.
```

Proof by induction on `fact`

Two cases for the input of `fact` : 0 or `S p`

- In second case, recursive call on `p`

Proof by induction makes the cases appear

- in step case where `n = S p`, induction hypothesis on `p`

```
Induction n as [ | p IHp].
  ======================
   0 < fact 0

Subgoal 2 is:
0 < fact (S p)
```

Force computation of recursive functions

Tactic `simpl` : compute the function but respect the recursive structure

```
  ====================
   0 < fact 0
simpl.
```

Force computation of recursive functions

Tactic `simpl` : compute the function but respect the recursive structure

```
  ====================
   0 < fact 0
simpl.

  ====================
   0 < 1
omega.
```

```
1 subgoal
  IHp : 0 < fact p
  ====================
    0 < fact (S p)
simpl.
```

```
1 subgoal
  IHp : 0 < fact p
  ====================
    0 < fact (S p)
simpl.
...
    0 < fact p + p * fact p
```

Completing the example

```
apply lt_le_trans with (fact p).
```

Completing the example

```
apply lt_le_trans with (fact p).
...
  IHp : 0 < fact p
  ======================
   0 < fact p

Subgoal 2 is:
  factp <= fact p + p * fact p
```

Completing the example

```
apply lt_le_trans with (fact p).
...
  IHp : 0 < fact p
  ======================
   0 < fact p

Subgoal 2 is:
  factp <= fact p + p * fact p
SearchPattern (?x <= ?x + _).
le_plus_l : forall n m, n <= n + m
```

Completing the example

```
apply lt_le_trans with (fact p).
...
  IHp : 0 < fact p
  ======================
   0 < fact p

Subgoal 2 is:
  factp <= fact p + p * fact p
SearchPattern (?x <= ?x + _).
le_plus_l : forall n m, n <= n + m
apply le_plus_l.
Qed.
```

Induction on lists

Induction on lists is like induction on natural numbers

- base case : the empty list
- step case : the list with an element at the head and another list at the tail
- the tail can be handled by recursive calls and induction hypotheses

An example on lists

```
Require Import List.

Fixpoint rev1 (A : Type) (l1 l2 : list A) :=
 match l1 with
 | nil => l2
 | a::t1 => rev1 A t1 (a::l2)
 end.

Fixpoint rev (A : Type) (l : list A) :=
 match l with
 | nil => nil
 | a::tl => rev A tl ++ a::nil
 end.
```

Proof on rev

```
Lemma rev1_rev : forall A (l1 l2 : list A),
   rev1 A l1 l2 = rev A l1 ++ l2.
intros A; induction l1 as [ | a t1 IHt1].
```

Proof on rev

```
Lemma rev1_rev : forall A (l1 l2 : list A),
   rev1 A l1 l2 = rev A l1 ++ l2.
intros A; induction l1 as [ | a t1 IHt1].
  ============================
   forall l2, rev1 A nil l2 = rev A nil ++ l2
intros l2; reflexivity.
```

Proof on rev

```
Lemma rev1_rev : forall A (l1 l2 : list A),
   rev1 A l1 l2 = rev A l1 ++ l2.
intros A; induction l1 as [ | a t1 IHt1].
  ============================
   forall l2, rev1 A nil l2 = rev A nil ++ l2
intros l2; reflexivity.
  IHt1 : forall l2 : list A,
         rev1 A t1 l2 = rev A t1 ++ l2
  ============================
   forall l2 : list A,
        rev1 A (a :: t1) l2 = rev A (a :: t1) ++ l2
```

Finishing the proof on `rev`

```
intros l2; simpl.
```

Finishing the proof on rev

```
intros l2; simpl.
  IHt1 : forall l2, rev1 A t1 l2 = rev A t1 ++ l2
  ======================
  rev1 A t1 (a::l2) = (rev A tl ++ a::nil) ++ l2
rewrite IHt1, app_ass; simpl; reflexivity.
Qed.
```

Reasoning by cases

- Reasoning by cases is already provided by the tactic `induction`
- But induction adds induction hypotheses
- The tactics `case`, `case_eq`, `destruct` are more lightweight
  - `case` *e* replaces all instances of *e* in the conclusion with possible cases
  - `case_eq` *e* the same and adds an equality to remember the case
  - `destruct` *e* replaces all instances in conclusion and hypotheses of the goal

Example of case, case_eq, and destruct

```
Definition max m n := if leb m n then n else m.

Lemma maxge1 : forall m n, m <= max m n.
intros m n; unfold max.
assert (t1 := leb_complete m n).
assert (t2 := leb_complete_conv n m).
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
  m <= if leb m n then n else m
```

Example of case_eq

```
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
  m <= if leb m n then n else m
case_eq (leb m n).
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
  leb m n = true -> m <= n
intros t; apply t1; exact t.
```

Example of destruct

```
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
  m <= if leb m n then n else m
destruct (leb m n)).
  t1 : true = true -> m <= n
  t2 : true = false -> n < m
  ============================
   m <= n
apply t1; reflexivity.
```

Example of case

```
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
  m <= if leb m n then n else m
case (leb m n).
  t1 : leb m n = true -> m <= n
  t2 : leb m n = false -> n < m
  ============================
   m <= n
Abort.
```

Controlling execution

The tactic `simpl` performs computation, but sometimes it goes too far

- When you know what value to aim for use `change` e_1 `with` e_2
- The values e_1 and e_2 have to be obviously the same (for Coq)
- Use `replace` e_1 `and` e_2 : it gives you more work, but is more supple
- Use `change` C' to change the whole goal conclusion
- Use `unfold f` to only unfold the definition of `f`

Getting rid of inconsistent cases

An equality between two different constructors is an inconsistency

- to be handled with `discriminate` or `discriminate H`

```
  H : a::l = nil
  ======================
   C
discriminate.
Proof completed.
```

Decomposing equalities of constructors

An equality between two terms with the same constructor

- Components must be equal : constructors are injective
- The tactic is `injection`

```
  H : a :: l = b :: l'
  ====================
   C
injection H.
  H : a :: l = b :: l'
  ====================
   l = l' -> a = b -> C
```

Specialized induction principles

- General approach is to follow the structure of functions
- This can be expressed in a theorem
- Theorem generated by `Functional Scheme`
- Theorem then used by `functional induction`

A last example

```
Fixpoint even (n:nat) : bool :=
  match n with
  | 0 => true
  | 1 => false
  | S (S p) => even p
  end.

Functional Scheme even_ind :=
  Induction for even Sort Prop.
```

Proof by specialized induction

```
Lemma even_double : forall n, even n = true ->
  exists p, n = 2 * p.
intros n; functional induction even n.
3 subgoals
 ============================
   true = true -> exists p : nat, 0 = 2 * p

subgoal 2 is:
 false = true -> exists p : nat, 1 = 2 * p
subgoal 3 is:
 even p = true -> exists p0 : nat, S (S p) = 2 * p0
```

Finishing the proof for even_double

```
exists 0; reflexivity.
intros; discriminate.
  IHb : even p = true -> exists p0 : nat, p = 2 * p0
  ============================
   even p = true -> exists p0 : nat, S (S p) = 2 * p0
intros t; destruct (IHb t) as [p' qp']; rewrite qp'.
exists (S p'); ring.
```