Simple proofs about recursive functions

Yves Bertot

August 2009

At this point, we have seen how to program algorithms in Coq and how to perform basic proofs. We shall now concentrate on proving properties about the algorithms we were able to program. We will concentrate on :

- Specifying properties of functions
- Reasoning by cases on function inputs
- Eliminating inconsistent assumptions
- Using the injectivity of datatype constructors
- Using induction hypotheses

Properties of functions

We already know how to write formulas in higher-order logic

- We can quantify over input data, functions, predicates
- We can express relations between input and outputs
- We can restrict the conditions of application
- We can only assert consistency with other functions

Examples :

```
forall m n, n <= m -> (m - n) + n = m
```

```
forall m, 0 < n -> prime n = true ->
   ~exists x, exists y, 1 < x < n ∧ 1 < y < n
     ∧ n = x * y
```

Main guideline

Reason on executions of functions

- Perform the same case analyses as in the function
- Use induction when the function is recursive
  - Induction on the principal argument of the recursive function
  - Make the statement proved as general as possible

```
Print minus.
minus = fix minus (n m : nat) : nat :=
  match n with
  | O => n
  | S k =>
    match m with | O => n | S l => minus k l end
  end      : nat -> nat -> nat
```

Making statements as general as possible

Example on subtraction

```
Lemma ex1 : forall m n, n <= m -> (m - n) + n = m.
intros m n.

=====================
n <= m -> (m - n) + n = m
induction m.
focus on second goal
IHm : n <= m -> (m - n) + n = m
=====================
n <= S m -> (S m - n) + n = S m
```

The induction hypothesis IHn can only be used for one fixed value

Making statements as general as possible (2)

```
Lemma ex1 : forall m n, n <= m -> (m - n) + n = m.
intros m.

====================
forall n, n <= m -> (m - n) + n = m
induction m.
focus on second goal
IHm : forall n, n <= m -> (m - n) + n = m
======================
forall n, n <= S m -> (S m - n) + n = S m
```

Following the structure of functions

A proof by induction step usually takes care of one of the pattern-matching constructs
Other pattern-matching constructs must be followed in the proof too.
Use the `case` or `destruct` tactics for this.

More pattern-matching in subtraction

```
intros n; case n.
====================
0 <= S m -> (S m - 0) + 0 = S m

subgoal 2 is;
============================
forall n0, S n0 <= S m -> (S m - S n0) + S n0 = S m
```

Use the definition of functions

When enough pattern-matching is performed, we can give a value to a function call
The value can be stated precisely by hand using a tactic `change`
It can also be computed automatically by a tactic `simpl`

Computing by hand

```
====================
0 <= S m -> (S m - 0) + 0 = S m
change (S m - 0) with (S m).
====================
0 <= S m -> S m + 0 = S m
```

Accepts the change only if it is a consequence of the definition
The final statement can be proved automatically
We can also look for a theorem working on `S m + 0`

```
SearchRewrite (_ + 0).
plus_0_r: forall n : nat, n + 0 = 0
```

Completing one goal

```
==================
0 <= S m -> S m + 0 = S m
rewrite plus_0_r.
==================
0 <= S m -> S m = S m
reflexivity.
```

Solving a recursive case

```
====================
forall n0, S n0 <= S m -> S m - S n0 + S n0 = S m
intros p plem; simpl.
IHm : forall n, n <= m -> m - n + n = m
...
====================
m - p + S p = S m
SearchRewrite (_ + S _).
plus_n_Sm : forall n m, S (n + m) = n + S m
rewrite <- plus_n_Sm.
====================
S (m - p + p) = S m
```

Solving a recursive case (2)

```
rewrite IHm.
==================
S m = S m

subgoal 2 is:
p <= m
reflexivity.
SearchPattern (_ <= _).
...
le_S_n : forall n m, S n <= S m -> n <= m
apply le_S_n; exact plem.
```

Looking at the base case

```
forall n, n <= 0 -> 0 - n + n = 0
simpl.
forall n, n <= 0 -> n = 0
```

The statement we get does not mention subtraction, but it is still true

It can be proved by cases on `n`

```
SearchPattern (~S _ <= 0).
le_Sn_0 : forall n, ~ S n <= 0
intros n; case n.
==================
0 <= 0 -> 0 = 0
```

Absurd case

When `n` is non-zero the hypothesis `n <= 0` is contradictory.

```
==================
forall n0, S n0 <= 0 -> S n0 = 0
SearchPattern (~S _ <= 0).
le_Sn_0 : forall n, ~ S n <= 0
intros p Sple0; case (le_Sn_0 p).
Sple0 : S p <= 0
==================
S p <= 0
exact Sple0.
```

An example on lists

```
Lemma ex2 : forall A B (f:A->B) l l1 l2 y,
  map f l = l1++y::l2 ->
  exists l1', exists l2', exists x,
   l1 = map f l1' ∧ l2 = map f l2' ∧ y = f x.
intros A B f; induction l as [ | a l IHl].
====================
forall l1 l2 y, map f nil = l1++y::l2 -> exists ...
intros l1 l2 y; destruct l1 as [ | a' l1]; simpl.
====================
nil = y::l2 -> ...
intros H'; discriminate H'.
```

Second absurd case on data

```
====================
nil = a'::l1++y::l2 -> ...
intros H'; discriminate H'.
```

In general, `Discriminate H` succeeds as soon as `H` is an equality between two different constructors of a datatype.

Proof on lists continued

```
IHl : forall l1 l2 y,
    map f l = l1++y ::l2 -> exists l1' ...
====================
forall l1 l2 y, map f (a::l) = l1 ++ y :: l2 -> exists ...
intros l1 l2 y; destruct l1 as [ | a' l1]; simpl.
====================
f a :: map f l = y :: l2 ->
 exists l1', exists l2', exists x,
  nil = map f l1' ∧ l2 = map f l2' ∧ y = f x
```

We can choose `nil` for `l1'`, `l` for `l'2`, and `a` for `x`

Decomposing an equality

```
intros q; injection q.
```

```
====================
```

```
map f l = l2 -> f a = y ->
exists l1', exists l2', exists x,
  nil = map f l1' ∧ l2 = map f l2' ∧ y = f x
```

```
intros; exists nil; exists l; exists a; auto.
```

Proof on lists continued

```
IHl : forall l1 l2 y,
  map f l = l1 ++y::l2 ->
  exists l1', ..., l1 = map f l1' ∧ ...
====================
f a :: map f l = a' :: l1 ++ y :: l2 ->
exists l1', ...
intros q; injection q; intros ql qa.
ql : map f l = l1 ++ y :: l2
qa : f a = a'
====================
exists l1', exists l2', exists x, a'::l1 = map f l'1 ∧ ...
```

Using the induction hypothesis

```
IHl : forall l1 l2 y, map f l = l1 ++ y ::l2 ->
  exists l1', exists l2', exists x, ... ∧ ... ∧ ...
ql : map f l = l1 ++ y :: l2
destruct (IHl _ _ _ ql) as [l1' [l2' [x [q1 [q2 q3]]]]].
qa : f a = a'
q1 : l1 = map f l1'
...
=========================
exists l1'0, exists l2'0, exists x0,
  a' :: l1 = map f l1'0 ∧ ...
exist (a::l1'); exists l2'; exists x; subst; auto.
```

The Function command

The discipline for recursive programming imposes that recursive calls go down in the structure of terms

- Not easy for programming on integers : need to master the representation of numbers
- The `Function` command makes it possible to add artificial structure, but hide it from the user
- Popular approach : measuring with a natural number

Example using the Function command

```
Require Import ZArith Recdef.
Open Scope Z_scope.

Function fact (x:Z) measure Zabs_nat x :Z :=
  if Zle_bool x 0 then 1 else x * fact (x - 1).
forall x, Zle_bool x 0 = false ->
  Zabs_nat (x - 1) < Zabs_nat x
intros x xp.
assert (xp' : 1 <= x)
 by (apply Zle_bool_imp_le; apply Zone_min_pos; assumption)
apply Zabs_nat_lt.
Defined.

Eval compute in fact 3.
 = 6 : Z
```

Auxiliary theorems

The `Function` command generates a collection of auxiliary theorems and relations

- An equation to *unroll* the definition
- An induction principle on bouts of recursive execution
- Induction on trees of recursive calls instead of input data

Example functional induction

```
Lemma fact_pos : forall x, 0 < fact x.
intros x; functional induction fact x.
x : Z
e : Zle_bool x 0 = true
============================
0 < 1

subgoal 2 is:
 0 < x * fact (x - 1)
omega.
```

Functional induction continued

```
e : Zle_bool x 0 = false
IHz : 0 < fact (x - 1)
============================
0 < x * fact (x - 1)
assert (xp' : 1 <= x)
  by (apply Zle_bool_imp_le; apply Zone_min_pos; assumptior
apply Zmult_lt_0_compat; omega.
Proof completed.
Qed.
```