Proofs about programs

Yves Bertot

Proofs about computation

- Reason about functional correctness
- State properties about computation results
  - Show consistency between several computations
- Use the same tactics as for usual logical connectives
- Add tactics to control computations and observation of data
- Follow the structure of functions
  - Proving is akin to symbolic debugging
  - A proof is a guarantee that all cases have been covered

Controlling execution

- Replace formulas containing function with other formulas
- Manually with direct Coq control:
  - `change` f_1 `with` f_2
  - Really checks that f_1 and f_2 are the same modulo computation
- Manually with indirect control
  - `replace` f_1 `with` f_2
  - Produces a side goal with the equality $f_1 = f_2$
- Unfold recursive functions, keeping readable output
  - `simpl`, `simpl` f
  - Sometimes computes too much (so the output is not so readable!)
- Simply expand definitions
  - `unfold f`, `unfold f at 2`

Reason on other functions

- Each function comes with theorems about it
- In this course, sometimes called companion theorems
- Usable directly through apply when the goal's conclusion fits
- Otherwise, can be brought in the context using `assert`
  `assert (H : th a b c H').`
- Can be moved from the context to the goal using `revert`.

Example reasoning on functions

```
Parameters (f g : nat -> nat) (P Q R : nat -> nat -> Prop).

Axiom Pf : forall x, P x (f x).
Axiom Qg : forall y, Q y (g y).
Axiom PQR : forall x y z, P x y -> Q y z -> R x z.

Definition h (x:nat) := g (f x).

Lemma exfgh: forall x, R x (h x).
intros x; apply PQR with (y:= f x).
  x : nat
  ============================
   P x (f x)
apply Pf.
```

Example (continued)

```
  x : nat
  ============================
   Q (f x) (h x)
change (h x) with (g (f x)).
  x : nat
  ============================
   Q (f x) (g (f x))
apply Qg.
Proof completed.
Qed.
```

Reasoning about pattern-matching constructs

- Pattern-matching typically describes alternative behaviors
- Reason by covering all cases
- `case` is the basic constructs
  - generates one goal per constructor
  - the expression is replaced by constructor-values, in the conclusion
  - the argument to `S` becomes a universally quantified variable
- `destruct` is more advanced and covers the context
  - like `case`, but nesting is authorized
  - the context is also modified
- `case_eq` remembers in which case we are
  - the context is not modified (as in `case`)
  - remembering can be crucial

Example on cases

```
Definition pred (x:nat) :=
  match x with O => x | S p => p end.

Lemma S_pred : forall x, x <> 0 -> S (pred x) = x.
intros x; unfold pred.
  x : nat
  ============================
   x <> 0 ->
   S match x with | O => x | S p => p end = x
```

Example on cases (continued)

```
case x.
2 subgoals
  x : nat
  ============================
   0 <> 0 -> 1 = 0

subgoal 2 is:
 forall n : nat, S n <> 0 -> S n = S n
```

Example on cases (continued)

```
case x.
2 subgoals
  x : nat
  ============================
   0 <> 0 -> 1 = 0

subgoal 2 is:
 forall n : nat, S n <> 0 -> S n = S n

intros n0; case n0.
  ============================
   0 = 0
reflexivity.
intros; reflexivity.
Qed.
```

Example using companion theorems

```
Require Import Arith.
Check beq_nat_true.
beq_nat_true:
    forall x y : nat, beq_nat x y = true -> x = y
Definition pre2 (x : nat) :=
    if beq_nat x 0 then 1 else pred x.

Lemma pre2pred : forall x, x <> 0 -> pre2 x = pred x.
intros x; unfold pre2.
  x : nat
  ============================
   x <> 0 ->
   (if beq_nat x 0 then 1 else pred x) = pred x
```

Companion theorems (continued)

```
case_eq (beq_nat x 0).
2 subgoals
  x : nat
  ============================
   beq_nat x 0 = true -> x <> 0 -> 1 = pred x

subgoal 2 is:
 beq_nat x 0 = false -> x <> 0 -> pred x = pred x
intros test; assert (x0 := beq_nat_true _ _ test).
  test : beq_nat x 0 = true
  x0 : x = 0
  ============================
   x <> 0 -> 1 = pred x
intros xn0; case xn0; exact x0.
intros; reflexivity.
Qed.
```

How to find Companion theorems

- `SearchAbout` is your friend
- In general `Search` commands are your friends
  - `Search`: use a predicate name
    `Search le.`
  - `SearchRewrite`: use patterns of expressions
    `searchRewrite (_ + 0).`
  - `SearchPattern`: use a pattern of a theorem's conclusion (type `Prop`, usually)
    `SearchPattern (_ * _ <= _ * _).`

Recursive functions and induction

- When a function is recursive, calls are usually made on direct subterms
- Companion theorems do not already exist
- Induction hypotheses make up for the missing theorems
- The structure of the proof is imposed by the data-type

A trick to control recursion

- Add one-step unfolding theorems to recursive functions
- Associate any definition
  `Fixpoint f x1 ...xn := body`
  with a theorem
  `forall x1 ...xn, f x1 ...xn := body`
- Use `rewrite` instead of `change`, `replace`, or `simpl`
- More concise than `replace` or `change`
- Better control than `simpl`
- `unfold` is not well-suited for recursive functions

Example proof on a recursive function

```
Fixpoint add n m :=
    match n with O => m | S p => add p (S m) end.

Lemma addnS : forall n m, add n (S m) = S (add n m).
induction n.
2 subgoals
  ============================
   forall m : nat, add 0 (S m) = S (add 0 m)

subgoal 2 is:
 forall m : nat, add (S n) (S m) = S (add (S n) m)
```

Example proof on a recursive function

```
Fixpoint add n m :=
    match n with 0 => m | S p => add p (S m) end.

Lemma addnS : forall n m, add n (S m) = S (add n m).
induction n.
2 subgoals
  ============================
   forall m : nat, add 0 (S m) = S (add 0 m)

subgoal 2 is:
 forall m : nat, add (S n) (S m) = S (add (S n) m)
intros m; simpl.
  ============================
   S m = S m
reflexivity.
```

Recursive function (continued)

```
  n : nat
  IHn : forall m : nat, add n (S m) = S (add n m)
  ============================
   forall m : nat, add (S n) (S m) = S (add (S n) m)
intros m; simpl.
  ============================
   add n (S (S m)) = S (add n (S m))
apply IHn.
Proof completed.
Qed.
```

Functional schemes

- The tactic induction assumes a simple form of recursion
  - direct pattern-matching on the main variable
  - recursive calls on direct subterms
- Coq recursion allows deeper recursive calls
- Need for specialized induction principles
- Provided by `Functional Scheme`.
  - Exhibits the true pattern-matching structure from the function
  - Provides induction hypotheses suited for recursive calls.

Example functional scheme

```
Fixpoint div2 (x : nat) : nat :=
  match x with S (S p) => S (div2 p) | _ => 0 end.

Functional Scheme div2_ind :=

  Induction for div2 Sort Prop.

Lemma div2_le : forall x, div2 x <= x.
intros x; induction x using div2_ind.
3 subgoals
0 <= 0

0 <= 1

S (div2 p) <= S (S p)
```

Functional scheme (continued)

```
  e : x = S n
  p : nat
  e0 : n = S p
  IHn : div2 p <= p
  ============================
   S (div2 p) <= S (S p)
info auto with arith.
 == simple apply le_S; simple apply gt_le_S;
      change (div2 p < S p);
      simple apply le_lt_n_Sm; exact IHn.

Proof completed.
Qed.
```

Proofs on functions on lists

- Tactics `case`, `destruct`, `case_eq` also work
  - values `a` and `tl` in `a::tl` are universally quantified in `case` and `case_eq`, added to the context in `destruct`
- Induction on lists works like induction on natural numbers
- `nil` plays the same role as `0`: base case of proofs by induction
- `a::tl` plays the same role as `S`
  - Induction hypothesis on `tl`
  - Fits with recursive calls on `tl`

Example proof on lists

```
Require Import List.

Print rev.
fun A : Type => fix rev (l : list A) : list A :=
  match l with
  | nil => nil
  | x :: l' => rev l' ++ x :: nil
  end : forall A : Type, list A -> list A

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

Implicit Arguments rev_app.
```

Example proof on lists (continued)

```
Lemma rev_appP : forall A (l1 : list A),
   rev_app l1 nil = rev l1.
intros A l1.
  A : Type
  l1 : list A
  ===========================
  rev_app l1 nil := rev l1
assert (tmp: forall l2, rev_app l1 l2 = rev l1 ++ l2);
  [ | rewrite tmp, <- app_nil_end; reflexivity].
```

Example proof on lists (continued)

```
forall l2 : list A, rev_app l1 l2 = rev l1 ++ l2
induction l1; intros l2.
2 subgoals

  A : Type
  l2 : list A
  ============================
   rev_app nil l2 = rev nil ++ l2

subgoal 2 is:
 rev_app (a :: l1) l2 = rev (a :: l1) ++ l2
simpl; reflexivity.
```

proof on lists (continued)

```
  IHl1 : forall l2 : list A, rev_app l1 l2 = rev l1 ++ l2
  l2 : list A
  ============================
   rev_app (a :: l1) l2 = rev (a :: l1) ++ l2
simpl.
   rev_app l1 (a :: l2) = (rev l1 ++ a :: nil) ++ l2
SearchRewrite ((_ ++ _) ++ _).
app_ass:
    forall A (l m n:list A), (l ++ m) ++ n = l ++ m ++ n
rewrite app_ass; apply IHl1.
Proof completed.
Qed.
```