Coq Sessions

Polymorphism and Higher-Order Functions

Jeroen Goudsmit
Universiteit Utrecht

monday september 25th 2011

Contents

f_equal

$$\frac{t_1 = s_1, \ldots, t_n = s_n}{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)}$$

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal.
  now apply IH.
Qed.
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal.
  now apply IH.
Qed.
```

```
Coq < Lemma snoc_app: forall (ls : natlist) (l : nat),
Coq <   snoc ls l = ls ++ [l].
Coq < 1 subgoal
============================================
 forall (ls : natlist) (l : nat), snoc ls l = ls ++ [l]
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal.
  now apply IH.
Qed.
```

```
2 subgoals
l : nat
==================================================
 snoc [] l = [] ++ [l]
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity. •
  simpl.
  f_equal.
  now apply IH.
Qed.
```

```
1 subgoal
x : nat
xs : natlist
IH : forall l : nat, snoc xs l = xs ++ [l]
l : nat
======================================================
 snoc (x :: xs) l = (x :: xs) ++ [l]
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal.
  now apply IH.
Qed.
```

```
1 subgoal
x : nat
xs : natlist
IH : forall l : nat, snoc xs l = xs ++ [l]
l : nat
============================================
 x :: snoc xs l = x :: xs ++ [l]
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal. •
  now apply IH.
Qed.
```

```
1 subgoal
x : nat
xs : natlist
IH : forall l : nat, snoc xs l = xs ++ [l]
l : nat
========================================================
 snoc xs l = xs ++ [l]
```

f_equal in action

```
Lemma snoc_app: ∀ (ls : natlist) (l : nat),
  snoc ls l = ls ++ [l].
Proof.
  induction ls as [| x xs IH]; intro l.
  reflexivity.
  simpl.
  f_equal.
  now apply IH.
Qed.
```

```
Proof completed.
```

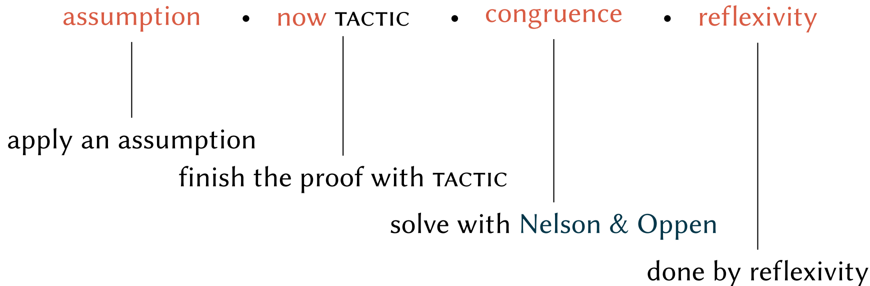
Closing Tactics

Ensure proof-flow.

assumption • now TACTIC • congruence • reflexivity

Closing Tactics

Ensure proof-flow.



Inversion

```
Lemma length_inversion: ∀ (X : Type) (l : list X),
  length l = 0 → l = [].
Proof.
  destruct l; simpl; intro eq.
    reflexivity.
    now inversion eq.
Qed.
```

Refactoring

Theorem *split_combine*: $\forall$ (*X Y*: *Type*) (l_1 : *list X*) (l_2 : *list Y*) ,
  length l_1 = *length* l_2 → split (*combine* l_1 l_2) = (l_1, l_2).

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  intros X Y; induction l1; intros l2 len; simpl in len.
  rewrite ← length_inversion with (l := l2); try congruence.
  reflexivity
  simpl; destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  intros X Y; induction l1; intros l2 len; simpl in len.
  rewrite ← length_inversion with (l := l2) by congruence.
  reflexivity
  simpl; destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; intros l2 len; simpl in len.
  rewrite ← length_inversion with (l := l2) by congruence.
  reflexivity
  simpl; destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len; simpl in len.
  rewrite ← length_inversion with (l := l2) by congruence.
  reflexivity
  simpl; destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len.
  rewrite ← length_inversion with (l := l2) by congruence.
  reflexivity
  destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len.
    f_equal; apply length_inversion.
  destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len.
    now f_equal; apply length_inversion.
  destruct l2; simpl in len; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len.
    now f_equal; apply length_inversion.
  destruct l2; inversion len.
  simpl.
  rewrite IHl1; congruence.
Qed.
```

Refactoring

```
Theorem split_combine: ∀ (X Y: Type) (l1 : list X) (l2 : list Y) ,
  length l1 = length l2 → split (combine l1 l2) = (l1, l2).
Proof.
  induction l1; simpl; intros l2 len.
    now f_equal; apply length_inversion.
  destruct l2; inversion len; simpl.
  rewrite IHl1; congruence.
Qed.
```

Reflection

Idea: prove some proposition via a computation.

Reflection of equality on $\mathbb{N}$

Lemma. *beqnat_reflection*: $\forall\ n\ m : nat$,
  $beq_nat\ n\ m = true \rightarrow n = m$.

Reflection of equality on ℕ

```
Lemma. beqnat_reflection: ∀ n m : nat,
  beq_nat n m = true → n = m.
Proof.
induction n; destruct m; simpl; intro eq.
  reflexivity.
  discriminate.
  discriminate.
  rewrite IHn with (m := m).
    reflexivity.
    assumption.
Qed.
```

Reflection of equality on ℕ

```
Lemma. beqnat_reflection: ∀ n m : nat,
  beq_nat n m = true → n = m.
Proof.
induction n; destruct m; simpl; intro eq.
  reflexivity.
  discriminate.
  discriminate.
  rewrite IHn with (m := m) by assumption.
    reflexivity.
Qed.
```

Reflection of equality on ℕ

```
Lemma. beqnat_reflection: ∀ n m : nat,
  beq_nat n m = true → n = m.
Proof.
induction n; destruct m; simpl; intro eq.
  reflexivity.
  discriminate.
  discriminate.
  erewrite IHn by eassumption.
    reflexivity.
Qed.
```

Reflection of equality on ℕ

```
Lemma. beqnat_reflection: ∀ n m : nat,
  beq_nat n m = true → n = m.
Proof.
induction n; destruct m; simpl; intro eq.
  reflexivity.
  discriminate.
  discriminate.
   auto.
Qed.
```

Reflection of equality on ℕ

Lemma. *beqnat_reflection*: ∀ *n m* : *nat*,
  beq_nat n m = *true* → *n* = *m*.
Proof.
induction *n*; destruct *m*; simpl; intro eq.
  reflexivity.
  discriminate.
  discriminate.
  now auto.
Qed.

Reflection of equality on ℕ

Lemma. *beqnat_reflection*: ∀ *n m* : *nat*,
  beq_nat n m = *true* → *n* = *m*.
Proof.
induction *n*; destruct *m*; simpl; intro eq.
  now auto.
  discriminate.
  discriminate.
  now auto.
Qed.

Reflection of equality on ℕ

```
Lemma. beqnat_reflection: ∀ n m : nat,
  beq_nat n m = true → n = m.
Proof.
induction n; destruct m; simpl; intro eq;
  now auto || discriminate.
Qed.
```

Inequality

Corollary *beqnat_nonequal*: $\forall\ n\ m : nat$,
$n \neq m \rightarrow beq_nat\ n\ m = false$.

Inequality

```
Corollary beqnat_nonequal: ∀ n m : nat,
  n ≠ m → beq_nat n m = false.
Proof.
  intros ? ? H.
  remember (beq_nat n m) as beq.
  destruct beq .
  Case "beq = true".
    destruct H.
    apply beqnat_reflection.
    symmetry; assumption.
  Case "beq = false".
    reflexivity.
Qed.
```

Inequality

```
Corollary beqnat_nonequal: ∀ n m : nat,
  n ≠ m → beq_nat n m = false.
Proof.
  intros ? ? H.
  remember (beq_nat n m) as beq.   ←—— make sure we don't forget!
  destruct beq .
  Case "beq = true".
    destruct H.
    apply beqnat_reflection.
    symmetry; assumption.
  Case "beq = false".
    reflexivity.
Qed.
```

Inequality

```
Corollary beqnat_nonequal: ∀ n m : nat,
  n ≠ m → beq_nat n m = false.
Proof.
  intros ? ? H.
  remember (beq_nat n m) as beq.
  destruct beq .
  Case "beq = true".
    destruct H.
    apply beqnat_reflection.
    symmetry; assumption.
  Case "beq = false".
    reflexivity.
Qed.
```

Inequality

```
Corollary beqnat_nonequal: ∀ n m : nat,
  n ≠ m → beq_nat n m = false.
Proof.
  intros ? ? H.
  remember (beq_nat n m) as beq.
  destruct beq ; try reflexivity.
    apply beqnat_reflection.
    symmetry; assumption.
Qed.
```

Reflection of being even

```
Fixpoint evenb (n : nat) : bool :=
  match n with
    |          0  ⇒  true
    | S (S n')    ⇒  evenb n'
    |          _  ⇒  false
  end.
```

Reflection of being even

```
Fixpoint evenb (n : nat) : bool :=
  match n with
    |          0  ⇒  true
    | S (S n')    ⇒  evenb n'
    |          _  ⇒  false
  end.

Definition double (n : nat) : nat
  := n + n.

Definition even (n : nat) : Prop
  := ∃ m : nat, n = double m.
```

Even things are computed even

```
Corollary evenb_even:
  ∀ n : nat, even n → evenb n = true.
Proof.
  intros n evenn.
  destruct evenn.
  subst.
  now apply double_evenb.
Qed.
```

Even things are computed even

```
Corollary evenb_even:
  ∀ n : nat, even n → evenb n = true.
Proof.
  intros n evenn.
  destruct evenn.
  subst.
  now apply double_evenb.
Qed.

Lemma double_evenb:
  ∀ n : nat, evenb (double n) = true.
Proof.
  induction n.
    reflexivity.
  rewrite double_S; simpl.
  assumption.
Qed.
```

Conversely,...?

Lemma *even_evenb*:
$\forall\ n : nat,\ evenb\ n = true \rightarrow even\ n.$

Conversely,...?

Lemma *even_evenb*:
$\forall\ n : nat,\ evenb\ n = true \rightarrow even\ n.$

Usual induction gets us stuck!

Wellfounded induction

Lemma *even_wf*:
  $\forall\ n : nat, (\forall\ m : nat, m < n \to evenb\ m = true \to even\ m)$
    $\to evenb\ n = true \to even\ n.$

Wellfounded induction

```
Lemma even_wf:
  ∀ n : nat, (∀ m : nat, m < n → evenb m = true → even m)
    → evenb n = true → even n.

Lemma even_evenb:
  ∀ n : nat, evenb n = true → even n.
Proof.
  intro n.
  eapply well_founded_ind with (A := nat)
    (P := fun n : nat, evenb n = true → even n).
  now apply naturals_wellfounded.
  now apply even_wf.
Qed.
```

Finally, Reflection!

```
Example even_test: even 1026.
Proof.
  apply even_evenb.
  reflexivity.
Qed.
```

Finally, Reflection!

```
Example even_test: even 1026.
Proof.
  apply even_evenb.
  reflexivity.
Qed.
```

```
1 subgoal
===================
 even 1026
```

Finally, Reflection!

```
Example even_test: even 1026.
Proof.
  apply even_evenb.
  reflexivity.
Qed.
```

```
1 subgoal
==================
 evenb 1026 = true
```

Finally, Reflection!

```
Example even_test: even 1026.
Proof.
  apply even_evenb.
  reflexivity. •
Qed.
```

```
Proof Completed
```

Moreover: Creating Proof-Objects

```
Definition evenproof (n : nat) : option (even n).
  remember (evenb n) as b.
  destruct b.
  constructor.
  apply even_evenb; congruence.
  exact None.
Defined.
```

Moreover: Creating Proof-Objects

```
Definition evenproof (n : nat) : option (even n).
  remember (evenb n) as b.
  destruct b.
  constructor.
  apply even_evenb; congruence.
  exact None.
Defined.

Eval compute in evenproof 459.
```

Moreover: Creating Proof-Objects

```
Definition evenproof (n : nat) : option (even n).
  remember (evenb n) as b.
  destruct b.
  constructor.
  apply even_evenb; congruence.
  exact None.
Defined.

Eval compute in evenproof 459.
  = None
  : option (even 459)
```

Moreover: Creating Proof-Objects

```
Definition evenproof (n : nat) : option (even n).
  remember (evenb n) as b.
  destruct b.
  constructor.
  apply even_evenb; congruence.
  exact None.
Defined.

Eval compute in evenproof 918.
```

Moreover: Creating Proof-Objects

```
Definition evenproof (n : nat) : option (even n).
  remember (evenb n) as b.
  destruct b.
  constructor.
  apply even_evenb; congruence.
  exact None.
Defined.
```

```
Eval compute in evenproof 918.
  = Some (even_evenb 918 (eq_refl true))
   option (even 918)
```

Sources

Split & Combine • Reflection

Interesting:
Reflection in “Certified Programming with Dependent Types”.