CS720

Logical Foundations of Computer Science

Lecture 10: Inductive propositions

Tiago Cogumbreiro

Summary

- How is Coq being used in research
- Exercises on inductive propositions
- Proofs by reflection

Projects that use Coq

- Coq Proof of the Four Color Theorem (Georges Gonthier, 2008) (Proposed in 1852, first proof in 1976 by Appel and Haken, proved in Coq in 2005). Four colors suffice to color any flat map.
- CompCert (2009): "CompCert is the first commercially available optimizing compiler that is formally verified, using machine assisted mathematical proofs, to be free from mis-compilation."
- Programming language formalization: Rust (2015), Haskell (2018)
- Verdi (2015): Verdi is a framework from the University of Washington to implement and formally verify distributed systems.
- A Formal Proof of the Expressiveness of Deep Learning (2017): A Formal Proof of the Expressiveness of Deep Learning.
- Coq: The world's best macro assembler (2013)

Projects that use Coq @ UMB

- Deadlock Avoidance in Parallel Programs with Futures (2017): formalized a task parallel programming model and the result that Data-Race-Freedom implies Deadlock-Freedom.
- Dynamic Deadlock Verification for General Barrier Synchronisation (2019): formalized phaser semantics and the notion of deadlock
- Checking Data-Race Freedom of GPU Kernels, Compositionally (2021): formalized GPU program semantics and our data-race-freedom analysis
- Formalizing the Introduction to the Theory of Computation (unpublished): decidability/undecidability results (eg, halting problem, Rice's theorem, etc). Rice's Theorem was proved by Kleopatra Ginji, an undergraduate student here at UMB.

Proofs are code

Proofs by induction

Derivation versus data

Recall the definition on even numbers

```
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => evenb n'
  end.

Inductive ev : nat -> Prop :=
| ev_0 : ev 0
| ev_SS : forall n : nat, ev n -> ev (S (S n)).
```

Let us prove that these two propositions are equivalent

```
Theorem evenb_to_ev:
  forall n,
  evenb n = true →
  ev n.
  (* Hint: use [even_bool_prop]; no need for induction. *)

Theorem ev_to_evenb:
  forall n,
  ev n →
  evenb n = true.

Theorem ev_iff_evenb:
  forall n,
  ev n ↔ evenb n = true.
```

Proofs by reflection

Reflection

We say that a proposition is reflected by a boolean value according to the following definition.

```
Inductive reflect (P : Prop) : bool -> Prop :=
| ReflectT : P -> reflect P true
| ReflectF : ~ P -> reflect P false.

Theorem iff_reflect : forall P b, (P <-> b = true) -> reflect P b.
Theorem reflect_iff : forall P b, reflect P b -> (P <-> b = true). (* Homework*)
```

Let us prove that `ev n` reflects `evenb n`.

```
Lemma ev_reflect : forall n, reflect (ev n) (evenb n).
```

Recall proving that 6 is even

It is much easier to compute that 6 is even, than to derive a proposition for it.

```
Theorem ev_6: ev 6.
Proof.
  apply ev_SS, ev_SS, ev_SS, ev_0.
Qed.

Theorem evenb_6: evenb 6 = true.
  reflexivity.
Qed.
```

Prove that 6 is even with reflection

```
Lemma reflect_true:
  forall P,
  reflect P true →
  P.
Proof.
  intros.
  inversion H.
  apply H0.
Qed.

Theorem ev_6_reflect: ev 6.
Proof.
  apply (reflect_true (ev 6) (ev_reflect 6)).
Qed.
```

Proof by Reflection

> The term reflection applies because we will need to **translate Gallina propositions into values of inductive types** representing syntax, so that Gallina programs may analyze them, and **translating such a term back to the original form** is called reflecting it.

– Certified Programming with Dependent Types

A bit more than what we have seen so far...

Reflecting the Logical And

```
Lemma reflect_and:
  forall P b1 Q b2,
  reflect P b1 →
  reflect Q b2 →
  reflect (P /\ Q) (andb b1 b2).
```

Reflecting the Logical Or

```
Lemma reflect_or:
  forall P b1 Q b2,
  reflect P b1 ->
  reflect Q b2 ->
  reflect (P \/ Q) (orb b1 b2).
```

A mini-language of expressions

```
Inductive Lang :=
| Eq: nat → nat → Lang        (* x = n  *)
| Even: nat → Lang            (* ev n   *)
| And: Lang → Lang → Lang     (* P /\ Q *)
| Or: Lang → Lang → Lang.     (* P \/ Q *)
```

Evaluate our mini-language

```
Fixpoint eval (exp:Lang) :=
  match exp with
  | Eq n m => beq_nat n m
  | Even n => evenb n
  | And l r => andb (eval l) (eval r)
  | Or l r => orb (eval l) (eval r)
  end.

Goal eval (Or (Even 3) (Eq 3 3)) = true.
  reflexivity.
Qed.
```

Generate a proposition

```
Fixpoint as_prop (exp:Lang) :=
  match exp with
  | Eq n m => n = m
  | Even n => ev n
  | And l r => as_prop l /\ as_prop r
  | Or l r => as_prop l \/ as_prop r
  end.

Goal as_prop (Or (Even 3) (Eq 3 3)).
  (* ev 3 \/ 3 = 3 *)
  simpl.
  right.
  reflexivity.
Qed.
```

Show that our language is reflective

```
Lemma reflect_lang:
  forall p,
  reflect (as_prop p) (eval p).

Goal ev 3 \/ 3 = 3.
  assert (H:=reflect_lang (Or (Even 3) (Eq 3 3))).
  apply reflect_true, H.
Qed.
```

Automating the translation

```
Ltac trans P :=
  match P with
    | ?P1 /\ ?P2 ⇒
      let t1 := trans P1 in
      let t2 := trans P2 in constr:(And t1 t2)
    | ev ?x ⇒ constr:(Even x)
    | ?P1 \/ ?P2 ⇒
      let t1 := trans P1 in
      let t2 := trans P2 in constr:(Or t1 t2)
    | ?x = ?y ⇒ constr:(Eq x y)
  end.

Goal ev 3 \/ 3 = 3.
  let t := trans (ev 3 \/ 3 = 3) in
  assert (H:= reflect_lang t).
```

Automating the translation

```
Ltac solve :=
  match goal with
    | [ |- ?P ] ⇒
      let t := trans P in
      let H := fresh "H" in
      assert (H := reflect_lang t);
      apply reflect_true, H
  end.

Goal ev 3 \/ 3 = 3.
  solve.
Qed.
```

Summary on Proof by Reflection

- Reflection establishes a deep connection between a proposition and the function that decides it
- We can leverage `Ltac` to automate trivial operations and build solvers ***(Not covered in this course.)***

Exercises on Less-Than

Prove that

1. `<` is transitive
2. `<` is irreflexive
3. `<` is asymmetric
4. `<` is decidable

Summary

- We looked at Coq being used in research
- Exercises on inductive propositions
- A deep dive in proofs by reflection