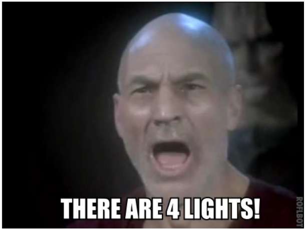
What's in a Proof?

Alex Vondrak

Cal Poly Pomona

November 14, 2012



An Anecdote

The Reason

```
Theorem a: 2 + 2 = 4.
```

The Reason

```
Theorem a: 2 + 2 = 4.
```

```
Proof.
  trivial.
Qed.
```

Coq

- An interactive theorem prover started in 1984
- Provides a formal language and environment for mathematical definitions, algorithms, theorems, and machine-checked proofs
- Language based on a derivative of the calculus of constructions (CoC)

Example

```
Theorem two_and_two_make_four: 2 + 2 = 4.
Proof.
  trivial.
Qed.
```

Coq

- An interactive theorem prover started in 1984
- Provides a formal language and environment for mathematical definitions, algorithms, theorems, and machine-checked proofs
- Language based on a derivative of the calculus of constructions (CoC)

Example

```
Theorem two_and_two_make_four: 2 + 2 = 4.
Proof.
  auto 1.
Qed.
```

Proof Automation

Rough Algorithm

```
auto n =
  if no more subgoals then
    success
  if n == 0 then
    failure
  foreach term in hypotheses ∪ hints :
    try
      apply term.
      foreach subgoal generated :
        auto (n - 1) on that subgoal
```

apply term.

- Tries to unify the goal with the conclusion of the type of "term"
- Returns subgoals—premises of the type of "term"

Example (At the Coq Top-Level)

```
Coq < Example ex: (1=2 → 2=1) → (2=1 → 1=2) → 1=2.
1 subgoal

  ============================
   (1 = 2 -> 2 = 1) -> (2 = 1 -> 1 = 2) -> 1 = 2

ex <
```

apply term.

- Tries to unify the goal with the conclusion of the type of “term”
- Returns subgoals—premises of the type of “term”

Example (At the Coq Top-Level)

```
  ============================
   (1 = 2 -> 2 = 1) -> (2 = 1 -> 1 = 2) -> 1 = 2

ex < intros.
1 subgoal

  H : 1 = 2 -> 2 = 1
  H0 :  2 = 1 -> 1 = 2
  ============================
   1 = 2
```

apply term.

- Tries to unify the goal with the conclusion of the type of “term”
- Returns subgoals—premises of the type of “term”

Example (At the Coq Top-Level)

```
  H : 1 = 2 -> 2 = 1
  H0 :  2 = 1 -> 1 = 2
  ============================
   1 = 2
```

```
ex < apply H.
Toplevel input, characters 6-7:
> apply H.
>       ^
Error:  Impossible to unify "2 = 1" with "1 = 2".
```

`apply term.`

- Tries to unify the goal with the conclusion of the type of "term"
- Returns subgoals—premises of the type of "term"

Example (At the Coq Top-Level)

```
ex < apply H0.
1 subgoal

  H : 1 = 2 -> 2 = 1
  H0 :  2 = 1 -> 1 = 2
  ============================
   2 = 1

ex <
```

Hints and Hypotheses

```
Coq < Theorem two_and_two_make_four: 2 + 2 = 4.
1 subgoal

  ============================
   2 + 2 = 4

two_and_two_make_four < Print Hint.

Applicable Hints :
[...]
In the database core:
  apply mult_n_O(0) apply mult_n_Sm(0) apply plus_n_O(0)
  apply eq_refl(0) apply plus_n_Sm(0)
  apply eq_add_S ; trivial(1) apply eq_sym ; trivial(1)
  apply f_equal (A:=nat)(1) apply f_equal2 mult(2)
  apply f_equal2 (A1:=nat) (A2:=nat)(2)
[...]
```

Hints and Hypotheses

Which Hint?

```
two_and_two_make_four < Proof.

two_and_two_make_four < info trivial.
 == apply eq_refl.

Proof completed.

two_and_two_make_four < Qed.
info trivial.

two_and_two_make_four is defined

Coq <
```

Equality

Definition

```
Inductive eq (A:Type) (x:A) : A → Prop :=
  eq_refl : eq A x x.
```

- `eq_refl` is a constructor of a proposition
  - Given evidence that `eq A x x`, ...
  - ...`eq_refl` allows us to conclude the proposition is true
- `eq_refl` "is a proof of" `eq A x x`
- It's the only way to prove something of type eq—thus, this defines the smallest reflexive relation

How Does That Help?

As it turns out, Coq tricks us a little. . .

```
Coq < Set Printing All.

Coq < Print two_and_two_make_four.
two_and_two_make_four =
@eq_refl nat (S (S (S (S O))))
     : @eq nat (plus (S (S O)) (S (S O)))
               (S (S (S (S O))))
```

Note

The `@`-sign has to do with making implicit arguments explicit for a particular function application (`2 + 2 = 4` leaves the type `nat` implicit)

Peano Arithmetic

Definition

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

Definition

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m)
  end

where "n + m" := (plus n m) : nat_scope.
```