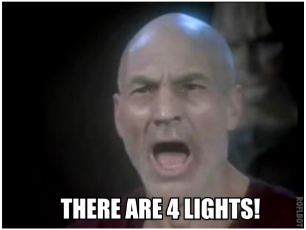
What's in a Proof?

Alex Vondrak

Cal Poly Pomona

March 24, 2013



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 “term”
- Returns subgoals—premises 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 "term"
- Returns subgoals—premises 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 “term”
- Returns subgoals—premises 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 “term”
- Returns subgoals—premises 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 (Pseudo)

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

Definition (Actual)

```
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`

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.
```

Predicative Calculus of (Co)Inductive Constructions

Problem

How can we apply `eq_refl.` to

```
@eq nat (plus (S (S O)) (S (S O)))
        (S (S (S (S O))))
```

when we need evidence of the form `eq A x x`?

Answer

The underlying formal language of Coq defines semantics for definitions, types, and (moreover) evaluation:

- δ-reduction
- ι-reduction
- β-reduction
- ζ-reduction (similar to δ)

δ-Reduction

Replaces definition names with their values

Example

```
@eq nat (plus (S (S O)) (S (S O)))
        (S (S (S (S O))))

Coq < cbv delta.

@eq nat
    ((fix plus (n m : nat) : nat :=
        match n return nat with
        | O ⇒ m
        | S p ⇒ S (plus p m)
        end) (S (S O)) (S (S O)))
    (S (S (S (S O))))
```

ι-Reduction

A specific conversion rule; for our purposes, expands **Fixpoint** definitions

Example

```
@eq nat ((fix plus (n m : nat) : nat :=
          match n return nat with
          | O ⇒ m
          | S p ⇒ S (plus p m)
          end) (S (S O)) (S (S O)))
        (S (S (S (S O))))

Coq < cbv iota.
```

ι-Reduction

A specific conversion rule; for our purposes, expands **Fixpoint** definitions

Example

```
@eq nat ((fun n m : nat ⇒
          match n return nat with
          | O ⇒ m
          | S p ⇒
              S ((fix plus (n0 m0 : nat) : nat :=
                   match n0 return nat with
                   | O ⇒ m0
                   | S p0 ⇒ S (plus p0 m0)
                   end) p m)
          end) (S (S O)) (S (S O)))
        (S (S (S (S O))))
```

β-Reduction

Applies **fun**ctions to their arguments

Example

```
@eq nat ((fun n m : nat ⇒
         match n return nat with
         | O ⇒ m
         | S p ⇒
             S ((fix plus (n0 m0 : nat) : nat :=
                   match n0 return nat with
                   | O ⇒ m0
                   | S p0 ⇒ S (plus p0 m0)
                   end) p m)
         end) (S (S O)) (S (S O)))
        (S (S (S (S O))))
```

β-Reduction

Applies **fun**ctions to their arguments

Example

```
Coq < cbv beta.

@eq nat (match S (S O) return nat with
         | O ⇒ S (S O)
         | S p ⇒
             S ((fix plus (n m : nat) : nat :=
                   match n return nat with
                   | O ⇒ m
                   | S p0 ⇒ S (plus p0 m)
                   end) p (S (S O)))
        end)
        (S (S (S (S O))))
```

ι-Reduction

Here, ι resolves the **match**

Example

```
@eq nat (match S (S O) return nat with
        | O ⇒ S (S O)
        | S p ⇒
            S ((fix plus (n m : nat) : nat :=
                  match n return nat with
                  | O ⇒ m
                  | S p0 ⇒ S (plus p0 m)
                  end) p (S (S O)))
        end)
        (S (S (S (S O))))
```

ι-Reduction

Here, ι resolves the **match**

Example

```
Coq < cbv iota.

@eq nat ((fun p : nat ⇒
          S ((fix plus (n m : nat) : nat :=
               match n return nat with
               | O ⇒ m
               | S p0 ⇒ S (plus p0 m)
               end) p (S (S O)))) (S O))
        (S (S (S (S O))))
```

β-Reduction

And β once again applies the **fun**ction...

Example

```
@eq nat ((fun p : nat ⇒
            S ((fix plus (n m : nat) : nat :=
                  match n return nat with
                  | O ⇒ m
                  | S p0 ⇒ S (plus p0 m)
                  end) p (S (S O)))) (S O))
        (S (S (S (S O))))
```

β-Reduction

And β once again applies the **fun**ction. . .

Example

```
Coq < cbv beta.

@eq nat (S ((fix plus (n m : nat) : nat :=
              match n return nat with
              | O ⇒ m
              | S p ⇒ S (plus p m)
              end) (S O) (S (S O))))
        (S (S (S (S O))))
```

Eventually…

ι resolves yet another **match**

Example

```
@eq nat (S (S (match O return nat with
               | O ⇒ S (S O)
               | S p ⇒
                 S ((fix plus (n m : nat) : nat :=
                      match n return nat with
                      | O ⇒ m
                      | S p0 ⇒ S (plus p0 m)
                      end) p (S (S O)))
               end)))
        (S (S (S (S O))))
```

Eventually. . .

ι resolves yet another **match**

Example

```
Coq < cbv iota.

@eq nat (S (S (S (S O))))
        (S (S (S (S O))))
```

Finally

So, what we really mean by

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

is

```
Theorem there_are_four_lights: 2 + 2 = 4.
Proof.
  cbv delta.
  cbv iota. cbv beta. cbv iota. cbv beta.
  cbv iota. cbv beta. cbv iota. cbv beta.
  cbv iota. cbv beta. cbv iota.
  apply eq_refl.
Qed.
```

none of which is really necessary!

Curry-Howard Isomorphism

In A Nutshell

$$\begin{aligned} \text{proofs} &\equiv \text{programs} \\ \text{propositions} &\equiv \text{types} \end{aligned}$$

(Coq can even export proofs as programs in other languages!)

Less about automated proofs, more about computer-assisted proofs

- To brute-force part of a solution: The Four Color Theorem
- To prove something deemed interesting: Robbins' Conjecture
- To better understand existing proofs: The Odd Order Theorem
- To rigorously validate software: CompCert