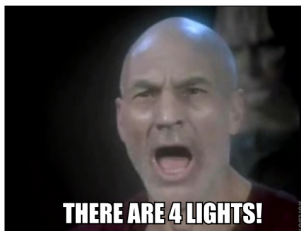


What's in a Proof?

Alex Vondrak

Cal Poly Pomona

February 14, 2013



An Anecdote



The Reason

Theorem a: $2 + 2 = 4$.

The Reason

Theorem a: $2 + 2 = 4$.

Proof.
trivial.
Qed.



- 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.



- 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  $\boxed{\text{hypotheses} \cup \text{hints}}$  :  
    try  
       $\boxed{\text{apply term.}}$   
    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)

=====

$(1 = 2 \rightarrow 2 = 1) \rightarrow (2 = 1 \rightarrow 1 = 2) \rightarrow 1 = 2$

```
ex < intros.
```

```
1 subgoal
```

H : $1 = 2 \rightarrow 2 = 1$

H0 : $2 = 1 \rightarrow 1 = 2$

=====

$1 = 2$

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_0(0) apply mult_n_Sm(0) apply plus_n_0(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 0))))  
      : @eq nat (plus (S (S 0)) (S (S 0)))  
          (S (S (S (S 0))))
```

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 :=  
  | 0 : nat  
  | S : nat → nat.
```

Definition

```
Fixpoint plus (n m:nat) : nat :=  
  match n with  
    | 0 ⇒ 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 0)) (S (S 0)))  
        (S (S (S (S 0))))
```

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 0)) (S (S 0)))  
        (S (S (S (S 0)))))
```

```
Coq < cbv delta.
```

```
@eq nat  
  ((fix plus (n m : nat) : nat :=  
    match n return nat with  
    | 0  $\Rightarrow$  m  
    | S p  $\Rightarrow$  S (plus p m)  
    end) (S (S 0)) (S (S 0)))  
  (S (S (S (S 0)))))
```

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  
    | 0 => m  
    | S p => S (plus p m)  
    end) (S (S 0)) (S (S 0)))  
  (S (S (S (S 0))))
```

```
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
  | 0 => m
  | S p =>
    S ((fix plus (n0 m0 : nat) : nat :=
      match n0 return nat with
      | 0 => m0
      | S p0 => S (plus p0 m0)
      end) p m)
  end) (S (S 0)) (S (S 0)))
(S (S (S (S 0))))
```

β -Reduction

Applies **functions** to their arguments

Example

```
@eq nat ((fun n m : nat =>
  match n return nat with
  | 0 => m
  | S p =>
    S ((fix plus (n0 m0 : nat) : nat :=
      match n0 return nat with
      | 0 => m0
      | S p0 => S (plus p0 m0)
      end) p m)
  end) (S (S 0)) (S (S 0)))
(S (S (S (S 0))))
```

β -Reduction

Applies **functions** to their arguments

Example

```
Coq < cbv beta.
```

```
@eq nat (match S (S 0) return nat with
| 0 => S (S 0)
| S p =>
    S ((fix plus (n m : nat) : nat :=
        match n return nat with
        | 0 => m
        | S p0 => S (plus p0 m)
        end) p (S (S 0)))
end)
(S (S (S (S 0))))
```

ι -Reduction

Here, ι resolves the **match**

Example

```
@eq nat (match S (S 0) return nat with
| 0  $\Rightarrow$  S (S 0)
| S p  $\Rightarrow$ 
    S ((fix plus (n m : nat) : nat :=
        match n return nat with
        | 0  $\Rightarrow$  m
        | S p0  $\Rightarrow$  S (plus p0 m)
        end) p (S (S 0)))
end)
(S (S (S (S 0))))
```

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
                | 0 => m
                | S p0 => S (plus p0 m)
                end) p (S (S 0)))) (S 0))
(S (S (S (S 0))))
```


β -Reduction

And β once again applies the **function**...

Example

```
@eq nat ((fun p : nat =>
  S ((fix plus (n m : nat) : nat :=
    match n return nat with
    | 0 => m
    | S p0 => S (plus p0 m)
    end) p (S (S 0)))) (S 0))
(S (S (S (S 0))))
```

β -Reduction

And β once again applies the **function**...

Example

```
Coq < cbv beta.
```

```
@eq nat (S ((fix plus (n m : nat) : nat :=  
              match n return nat with  
                | 0  $\Rightarrow$  m  
                | S p  $\Rightarrow$  S (plus p m)  
              end) (S 0) (S (S 0))))  
  (S (S (S (S 0))))
```

Eventually...

ι resolves yet another **match**

Example

```
@eq nat (S (S (match 0 return nat with
  | 0  $\Rightarrow$  S (S 0)
  | S p  $\Rightarrow$ 
    S ((fix plus (n m : nat) : nat :=
      match n return nat with
        | 0  $\Rightarrow$  m
        | S p0  $\Rightarrow$  S (plus p0 m)
      end) p (S (S 0)))
    end)))
(S (S (S (S 0))))
```

Eventually...

ι resolves yet another **match**

Example

```
Coq < cbv iota.
```

```
@eq nat (S (S (S (S 0))))  
        (S (S (S (S 0))))
```

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 Correspondence

In A Nutshell

propositions \equiv types
proofs \equiv programs

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

Strong future in automating proofs:

- Not just results that humans can't achieve on their own
 - Though this **has** happened (e.g., Robbins' conjecture)
 - Ask a math student about this sometime!
- End of the day: proof **assistants** (e.g., 4-color theorem, Compcert)