Proof

Informatics 1
Functional Programming

Phil Scott

University of Edinburgh

October 25, 2011

Where to find Proof?

- Mathematics?
- Science?
- Alcohol?
- Divine Revelation?

What is a Proof?

```
squares :: Integer -> Integer
squares x = x * x

squares_prop :: Integer -> Integer -> Bool
squares_prop x y =
  squares (x + y) == x * x + 2 * x * y + y * y
```

```
*Main> quickCheck squares_prop
+++ OK, passed 100 tests.
*Main>
```

What is a Proof?

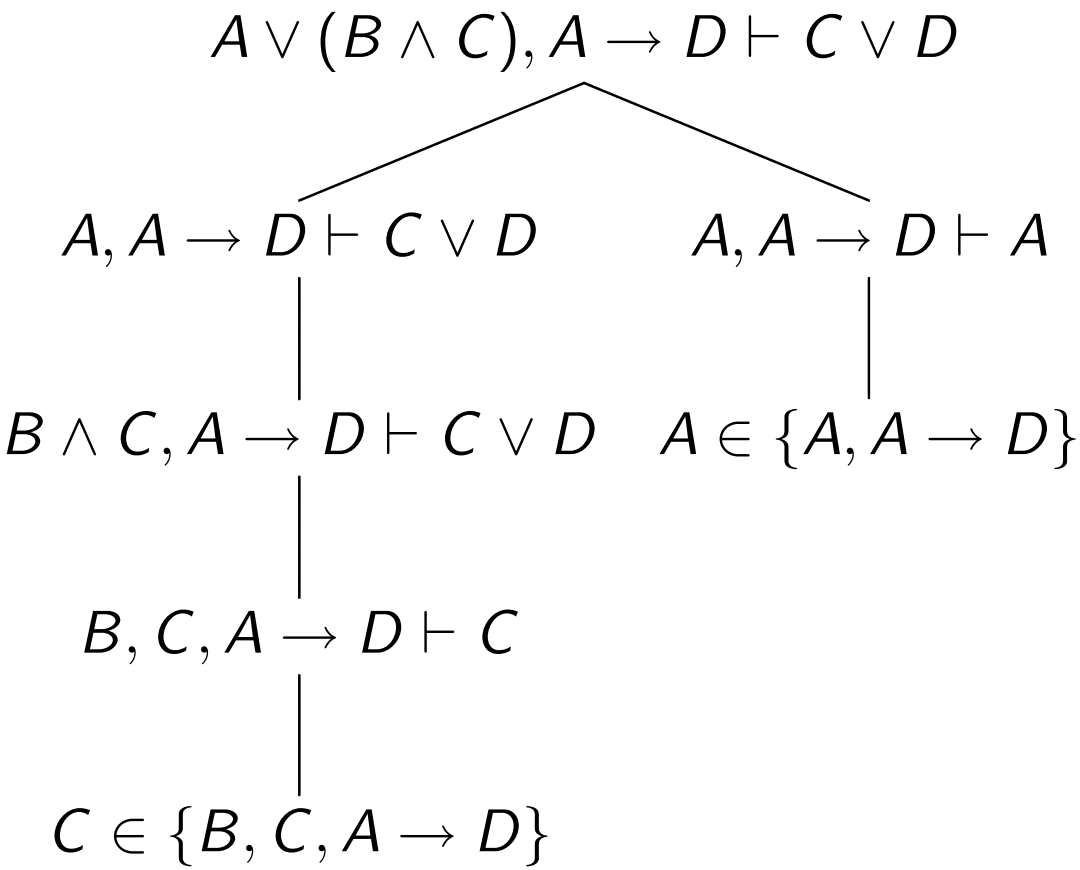
According to *Thinking Mathematically* (1982)

1. Convince yourself
2. Convince a friend
3. Convince an enemy

What about...

Convince a computer

What is a Proof?



Why do Proof?

Rule of Leibniz

- Indiscernability of Identity
- Identity of Indiscernables
- Equality is reflexive:
  $x = x$
- Equals may be substituted for equals

Retreat of Leibniz

The number I am thinking of *now* is not the number I am thinking of *now*.

```
i++ != i++
```

1. President of the United States = Barack Obama
2. Abraham Lincoln was President of the United States in 1861
3. Abraham Lincoln was Barack Obama in 1861

A Simple Function

```
square :: Integer -> Integer
square x = x * x

prop_squares :: Integer -> Integer -> Bool
prop_squares x y =
    square (x + y) == square x + 2 * x * y
                              + square y
```

Algebra

```
x + 0 = x
x * 1 = x
x + y = y + x
x * y = y * x
(x + y) + z = x + (y + z)
(x * y) * z = x * (y * z)
x * (y + z) = x * y + x * z

2 = 1 + 1
```

Algebraic Proof

```
squares (x + y) = x * x + (2 * (x * y) + y * y)

squares  (x + y)

    =  (x + y) * (x + y)                    -- Distrib.

    =  (x + y) * x  + (x + y) * y           -- Commut.

    = x * (x + y) +  (x + y) * y            -- Commut.

    =  x * (x + y)  + y * (x + y)           -- Distrib.

    = (x * x + x * y) +  y * (x + y)        -- Distrib.

    =  (x * x + x * y) + (y * x + y * y)    -- Assoc.

    = x * x + (x * y + ( y * x  + y * y))   -- Commut.

    = x * x +  (x * y + (x * y + y * y))
```

Algebraic Proof

```
x*x + ( 2 *(x*y) + y*y)
  = x*x + ( (1+1) * (x*y)  + y*y)              -- Commut.
  = x*x + ( (x*y) * (1+1)  + y * y)            -- Distrib.
  = x*x + (( (x*y) * 1  + (x*y) * 1) + y*y)    -- Id.
  = x*x + ((x*y +  (x*y) * 1 ) + y*y)          -- Id.
  = x*x +  ((x*y + x*y) + y*y)                 -- Assoc.
  = x * x + (x * y + (x * y + y * y))
```

Natural Numbers

```
data Nat = Zero
         | Suc Nat

(+) :: Nat -> Nat -> Nat
x + Zero = x
x + Suc y = Suc (x + y)

(*) :: Nat -> Nat -> Nat
x * Zero = Zero
x * Suc y = x + (x * y)

one = Suc Zero
two = Suc one
three = Suc two
four = Suc three
```

Advanced Mathematics

If I have two beans, and
I add two more beans,
what do I have?

Proof!

```
(+) :: Nat -> Nat -> Nat
x + Zero = x
x + Suc y = Suc (x + y)

(*) :: Nat -> Nat -> Nat
x * Zero = Zero
x * Suc y = x + (x * y)

 two + two
      = Suc (Suc Zero) + Suc (Suc Zero)
      = Suc (Suc (Suc Zero) + Suc Zero)
      = Suc (Suc (Suc (Suc Zero + Zero)))
      = Suc (Suc (Suc (Suc Zero)))
      = four
```

Cutting-Edge Mathematics

Prove that:

```
Zero + x = x
```

```
(+) :: Nat -> Nat -> Nat
x + Zero = x
x + Suc y = Suc (x + y)

(*) :: Nat -> Nat -> Nat
x * Zero = Zero
x * Suc y = x + (x * y)
```

Uh-oh! Our rules aren't enough!

Induction

To prove that a statement is true for all natural numbers:

1. Prove it is true for Zero;
2. Assuming it is true for n, show it is true for Suc n.

Suppose

```
p :: Nat -> Bool
p Zero = True
p (Suc n) | p n = True
```

Then

```
p n = True
```

```
p Zero = ...
p (Suc n) | p n = ...
```

Identity of Addition

Prove that:

```
Zero + x = x
```

Base case:

```
Zero + Zero = Zero
```

Step case:
Supposing that

```
Zero + x = x
```

We have

```
Zero + Suc x = Suc (Zero + x)
             = Suc x
```

Commutativity

Prove that:

```
x + y = y + x
```

Base case:

```
x + Zero = x
Zero + x = x
```

Step case:
Supposing that

```
x + y = y + x
```

We have

```
x + Suc y = Suc (x + y)
Suc y + x =
```

Uh-oh! We need a lemma.

Commutativity Lemma

Prove that:

```
Suc y + x = Suc (y + x)
```

Base case:

```
Suc y + Zero = Suc y
Suc (y + Zero) = Suc y
```

Step case:
Supposing that

```
Suc y + x = Suc (y + x)
```

We have

```
Suc y + Suc x
  = Suc (Suc y + x)
  = Suc (Suc (y + x))
Suc (y + Suc x) =
  Suc (Suc (y + x))
```

Commutativity again

Prove that:

Base case:

```
x + Zero = x
Zero + x = x
```

Step case:

Supposing that

```
x + y = y + x
```

We have

```
x + Suc y = Suc (x + y)
Suc y + x = Suc (y + x)
          = Suc (x + y)
```

Lists

```
data [a] = []
         | a : [a]

(++) : [a] -> [a] -> [a]
[] ++ xs       = xs
(x : xs) ++ ys = x : (xs ++ ys)

reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

Associativity of append

Prove that:

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

- Base case

  ```
  [] ++ (ys ++ zs) = ys ++ zs
  ([] ++ ys) ++ zs = ys ++ ys
  ```

- Step case

  Supposing that

  ```
  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
  ```

  We have

  ```
  (x : xs) ++ (ys ++ zs)
      = x : (xs ++ (ys ++ zs))

  ((x : xs) ++ ys) ++ zs
      = (x : (xs ++ ys)) ++ zs
      = x : ((xs ++ ys) ++ zs)
      = x : (xs ++ (ys ++ zs))
  ```

Reversing Append: Base case

Prove that:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

```
reverse ([] ++ ys) = reverse ys
reverse ys ++ reverse [] = reverse ys ++ []
                         =
```

Reversing Append: Step case

Supposing that

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

We have

```
reverse ((x : xs) ++ ys)
  = reverse (x : (xs ++ ys))
  = reverse (xs ++ ys) ++ [x]
  = (reverse ys ++ reverse xs) ++ [x]
  = reverse ys ++ (reverse xs) ++ [x])
```

```
reverse ys ++ reverse (x : xs)
  = reverse ys ++ (reverse xs ++ [x])
```

Double-Reverse

Prove that:

```
reverse (reverse xs) = xs
```

Summary

1. Proof is challenging, mechanical
2. Proof shows our programs are correct rigorously.
3. Haskell allows equational proof
4. Haskell recursion requires mathematical induction
5. $2 + 2 = 4$!