Proof and Programs

Informatics 1
Functional Programming Lecture 17

Don Sannella

University of Edinburgh

25 November 2013

Tutorials

Last tutorial this week, usual time/place

Revision tutorials this week:
Monday 1–2pm in AT 5.07
Wednesday 2–3pm in AT 5.05

Revision tutorials after this week:
Wednesday 2–3pm in AT 5.05 on 4 Dec
Wednesday 2–3pm in AT 5.05 on 11 Dec?
Wednesday 2–3pm in AT 5.05 on 18 Dec?

What is a Proof?

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

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

```
*Main> quickCheck prop_squares
+++ 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

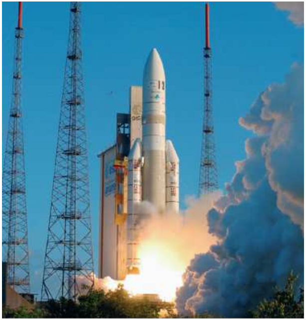
What about...

Convince a computer

What is a Proof?

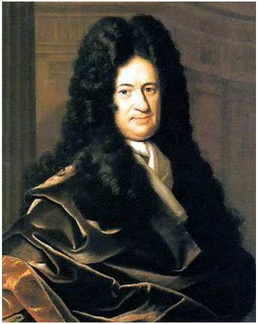
- $A \vee (B \wedge C), A \rightarrow D \vdash C \vee D$
  - $A, A \rightarrow D \vdash C \vee D$
    - $A, A \rightarrow D \vdash D$
      - $A, A \rightarrow D \vdash A$
        - $A \in \{A, A \rightarrow D\}$
      - $A, A \rightarrow D \vdash A \rightarrow D$
        - $A \rightarrow D \in \{A, A \rightarrow D\}$
  - $B \wedge C, A \rightarrow D \vdash C \vee D$
    - $B \wedge C, A \rightarrow D \vdash C$
      - $B, C, A \rightarrow D \vdash C$
        - $C \in \{B, C, A \rightarrow D\}$

Why do Proof?

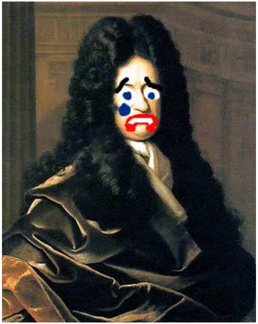


Rule of Leibniz

- 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) == x * x + 2 * x * y + y * 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

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

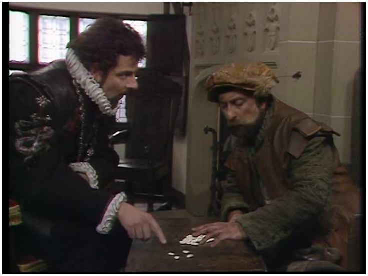
```
x*x + (2*(x*y) + y*y)
  = x*x + ((1+1) * (x*y) + y*y)                 -- Commut.
  = x*x + ((x*y) * (1+1) + y * y)               -- Distrib.
  = x*x + (((x*y) * 1 + (x*y) * 1) + y*y)       -- Id.
  = x*x + ((x*y + (x*y) * 1) + y*y)             -- Id.
  = x*x + ((x*y + x*y) + y*y)                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

Natural Numbers

```
data Nat = Zero
         | Succ Nat

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

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

one = Succ Zero
two = Succ one
three = Succ two
four = Succ 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 + Succ y = Succ (x + y)

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

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

Cutting-Edge Mathematics

Prove that:

```
Zero + x = x
```

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

(*) :: Nat -> Nat -> Nat
x * Zero = Zero
x * Succ 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 Succ n.

Suppose

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

Then

```
p n = True
```

Identity of Addition

Prove that:

```
Zero + x = x
```

Base case:

```
Zero + Zero = Zero
```

Step case:
Supposing that

```
Zero + x = x
```

We have

```
Zero + Succ x = Succ (Zero + x)
              = Succ 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 + Succ y = Succ (x + y)
Succ y + x =
```

Uh-oh! We need a lemma.

Commutativity Lemma

Prove that:

```
Succ y + x = Succ (y + x)
```

Base case:

```
Succ y + Zero = Succ y
Succ (y + Zero) = Succ y
```

Step case:
Supposing that

```
Succ y + x = Succ (y + x)
```

We have

```
Succ y + Succ x
  = Succ (Succ y + x)
  = Succ (Succ (y + x))
Succ (y + Succ x) =
  Succ (Succ (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 + Succ y = Succ (x + y)
Succ y + x = Succ (y + x)
           = Succ (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 induction
5. $2 + 2 = 4!$