Informatics 1

Functional Programming Lecture 9

Tuesday 27 October 2009

Proofs

Willem Heijltjes

University of Edinburgh

Formal Proofs

Why do proofs?

- Safety-critical systems
  (autopilots, internet banking, theorem provers)

Why not QuickCheck?

- What are the ‘right’ tests?

- How many test cases are ‘enough’?

What is a proof?

A (possible) definition:

"A formal argument showing the truth of a proposition"

A more helpful description is perhaps:

"A stepwise analysis of a proposition leading to basic statements (axioms), where each step is obviously correct, and each axiom is obviously true"

When is something 'obvious'?

This is 'socially' determined!

Pythagoras again

```
isTriple a b c = a*a + b*b == c*c
leg1 x y = x*x - y*y
leg2 x y = 2 * x * y
hyp  x y = x*x + y*y

prop_triple x y = isTriple (leg1 x y) (leg2 x y) (hyp x y)
```

Unfolding definitions

```
isTriple (leg1 x y) (leg2 x y) (hyp x y)

isTriple (x*x - y*y) (2 * x * y) (x*x + y*y)
```

Unfolding definitions

```
isTriple (leg1 x y) (leg2 x y) (hyp x y)

isTriple (x*x - y*y) (2 * x * y) (x*x + y*y)

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

Arithmetic

```
isTriple (leg1 x y) (leg2 x y) (hyp x y)

isTriple (x*x - y*y) (2 * x * y) (x*x + y*y)

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

Law: `(a+b)*(c+d) == a*c + a*d + b*c + b*d`

Arithmetic

```
isTriple (leg1 x y) (leg2 x y) (hyp x y)

isTriple (x*x - y*y) (2 * x * y) (x*x + y*y)

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

Law: `(a+b)*(c+d) == a*c + a*d + b*c + b*d`

```
x*x*x*x - 2*x*x*y*y + y*y*y*y + 4*x*x*y*y
                   == x*x*x*x + 2*x*x*y*y + y*y*y*y

x*x*x*x + 2*x*x*y*y + y*y*y*y
                   == x*x*x*x + 2*x*x*y*y + y*y*y*y
```

Induction

Suppose we want to prove something about lists.

- We don't know how long an arbitrary list is.
- But we may assume it ends somewhere. (Sometimes we don't!)
- And we know what it looks like:

  A list is either empty, or has a head and a tail

Induction

How induction works:

To prove that a property `p :: [Int] -> Bool` holds for any list, we must show:

- `p []`
- if `p xs` then `p (x:xs)` (for any `x :: Int`)

The first is called the *base case*,

the second the *induction step*, and

the statement `p xs` is the *induction hypothesis*.

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

The base case:

```
length [] >= 0
```

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

The base case:

```
length [] >= 0
```

Unfolding definitions (`length [] = 0`):

```
0 >= 0
```

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

The base case:

```
length [] >= 0
```

Unfolding definitions (`length [] = 0`):

```
0 >= 0
```

The induction step:

if `length xs >= 0` **then** `length (x:xs) >= 0`

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

The base case:

```
length [] >= 0
```

Unfolding definitions (`length [] = 0`):

```
0 >= 0
```

The induction step:

if `length xs >= 0` **then** `length (x:xs) >= 0`

Unfolding definitions (`length (x:xs) = 1 + length xs`):

if `length xs >= 0` **then** `length xs + 1 >= 0`

An easy example

We want to show that for every list `xs`:

```
length xs >= 0
```

The base case:

```
length [] >= 0
```

Unfolding definitions (`length [] = 0`):

```
0 >= 0
```

The induction step:

if `length xs >= 0` then `length (x:xs) >= 0`

Unfolding definitions (`length (x:xs) = 1 + length xs`):

if `length xs >= 0` then `length xs + 1 >= 0`

Mathematical induction

Induction can be done over the natural numbers as well.

To prove a property `p` we need to show:

- `p 0` (base case)
- if `p n` then `p (n+1)` (induction step).

The statement `p n` is the induction hypothesis.

Another example

Suppose we want to show:

```
even x || even (x+1)
```

Another example

Suppose we want to show:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Another example

Suppose we want to show:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Induction step:

if `even x || even (x+1)`

then `even (x+1) || even ((x+1)+1)`

Another example

Suppose we want to show:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Induction step:

if `even x || even (x+1)`

then `even (x+1) || even ((x+1)+1)`

Case distinction:

1. Suppose `even x`

2. Suppose `even (x+1)`

Another example

Suppose we want to show:

```
even x || even (x+1)
```

Base case:

```
even 0 || even (0+1)
```

(obvious)

Induction step:

if `even x || even (x+1)`

then `even (x+1) || even ((x+1)+1)`

Case distinction:

1. Suppose `even x`
   Then (obviously) `even (x+2)`
2. Suppose `even (x+1)`
   (We're done)