FUNCTIONAL PROGRAMMING

Vasco T. Vasconcelos

Lecture 8 – Reasoning About Programs

Testing

How many elements does a singleton list contain?
Let us test on hugs!

```
> length [7]
1
> length [-3]
1
> length [`a`]
1
> length [isDigit]
1
```

How many more tests do we need?

Manual evaluation

We can also perform manual evaluation.

```
length [7] = length (7:[]) =
1 + length [] = 1 + 0 = 1
```

```
length [-3] = length (-3:[]) =
1 + length [] = 1 + 0 = 1
```

How many more evaluations do we need?

Reasoning

We can instead do a little reasoning, and solve our problem <u>for all inputs</u>.

```
length [x] = length (x:[]) =
1 + length [] = 1 + 0 = 1
```

We use a variable, x, to denote any input value.

Simple proofs

Prove the law

```
[z] ++ zs = z:zs
```

Resorting to the definition

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

We have

```
[z] ++ zs = (z:[]) ++ zs = z : [] ++ zs = z : zs
```

Expand abbreviation

Eq 2 above

Eq 1 above

Show that

```
reverse [x] = [x]
```

Resorting to the definition

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

We have

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

Recall that [] ++ ys = ys.

More simple proofs

Prove the law

```
[] ++ zs = zs
```

resorting to the definition

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

It follows directly from equation 1 in the definition!

Now show that

```
zs ++ [] = zs
```

Looking at the definition of ++ we conclude that we must analyse the structure of zs.

A. If zs is the empty list, we use Eq 1

```
[] ++ [] = []
```

B. If zs is of the form z:zs' , we use Eq 2

```
(z:zs') ++ [] = z : (zs' ++ [])
```

How do I go from here to the result (z:zs')? If I could assume that zs' ++ [] = zs' , I would continue

```
z : (zs' ++ []) = z : zs'
```

As required!

Structural induction

The equation

```
zs'++ [] = zs'
```

is similar to

```
zs ++ [] = zs
```

only that zs' is "smaller"

Recall that zs = z : zs'

The principle of Structural Induction allows us to make the assumption, thus validating the proof.

Principle of structural induction for lists

In order to prove that a logical property P(xs) holds for all finite lists xs we have to do two things.

A. Base case: Prove P([]) outright.
B. Induction step: Prove P(x:xs) on the assumption that P(xs) holds.

The P(xs) in the induction step is called the induction hypothesis

The law of length and ++

Show that

```
length (xs ++ ys) = length xs + length ys
```

A. Base case: xs is []

(++) Eq 1

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

There is nothing more we can do on the left-hand-side; let us try the right-hand-side

```
length [] + length ys =
0 + length ys =
length ys
```

length Eq 1

arithmetic

Done!

B. Induction step: xs is x:xs’

```
length (x:xs’ ++ ys) =
length (x : (xs’ ++ ys)) =
1 + length (xs’ ++ ys) =
1 + length xs’ + length ys
```

(++) Eq 2

length Eq 2

induction hypothesis

Move to the right-hand-side

```
length (x:xs’) + length ys =
1 + length xs’ + length ys
```

length Eq 2

Done!

Correctness

Function `last` in the Prelude returns the last element of a list.

```
last:: [a] -> a
```

For example:

```
> last [‘a‘..‘z‘]
‘z‘
```

How can we be sure that `last` does what it is supposed to do?

A specification for last

We would like to assert that "last xs returns the element at position length xs - 1", that is:

```
last xs = xs !! (length xs – 1)
```

Spec only: visits xs twice!

Surely one can only compute the last element of a non-empty list. We add the condition:

```
length xs > 0
```

xs' != []

which means that xs is either [x] or (x:xs').

Proving last correct

Recall the Prelude definition for last

```
last            :: [a] -> a
last [x]        = x
last (_:xs) = last xs
```

Implementation: visits xs once!

And for (!!)

```
(!!)            :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

A. Case xs is [x]

```
last [x] = x                              -- last Eq 1
```

```
[x] !! (length [x]-1) =                   -- length
[x] !! 0 =                                -- (!!) Eq 1
x
```

B. Case xs is x:xs’

```
last (x:xs’) = last xs’                   -- last Eq 2
```

```
(x:xs’) !! (length (x:xs’)-1) =  -- length Eq 2
(x:xs’) !! (length xs’) =        -- (!!) Eq 2
xs’ !! (length xs’ - 1)          -- IH
last xs’
```

Done!

A specification for zip

Function zip in the Prelude produces a list of pairs from a pair of lists. We would like to say that "the resulting pair at a given position is obtained from the elements at the same position in the original lists".

```
(zip xs ys) !! i = (xs !! i, ys !! i)
```

We must say which are the valid indices i. Inspecting the right hand side we conclude:

```
0 <= i < min (length xs) (length ys)
```

The equation above is not enough!
An implementation of zip could place extraneous pairs at the back of the list. We must also specify the length of the resulting list.

```
length (zip xs ys) =
         min (length xs) (length ys)
```

Implementation of zip and (!!)

```
zip                  :: [a] -> [b] -> [(a,b)]
zip []     _          = []
zip _      []         = []
zip (x:xs) (y:ys) = (x,y) : zip xs yz
```

```
(!!)          :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

Recall our equation:

```
(zip xs ys) !! i = (xs !! i, ys !! i)
```

(!!) is defined by pattern-matching on n.
We need <u>structural induction for natural numbers</u>!

Principle of structural induction for natural numbers

In order to prove that a logical property P(n) holds for all finite numbers n we have to do two things.

A. Base case: Prove P(0) outright.
B. Induction step: Prove P(n+1) on the assumption that P(n) holds.

Once again, the P(n) in the induction step is called the induction hypothesis.

Proving zip correct (part 1)

xs is x:xs'
ys is y:ys' Why?

A. Base case: i is 0

```
(zip xs ys) !! 0 =              -- zip Eq 3
((x,y):zip xs' ys') !! 0 =      -- (!!) Eq 1
(x,y) =                         -- (!!) Eq 1
(xs !! 0, ys !! 0)
```

B. Induction step: i is k+1

```
(zip xs ys) !! (k+1) =           -- zip Eq 3
((x,y):zip xs' ys') !! (k+1) = -- (!!) Eq 2
(zip xs' ys') !! k =             -- IH
(xs' !! k, ys' !! k) =           -- (!!) Eq 2
(xs !! i, ys !! i)
```

Done!

Proving zip correct (part 2)

Zip is defined by pattern-matching on its first argument. For the second equation, use induction on xs!

A. Case xs is []

```
length (zip [] ys) =                -- zip Eq 1
length [] =                         -- length Eq 1
0
```

```
min (length []) (length ys) =  -- length Eq 1
min 0 (length ys)               -- a law
0
```

B. Case xs is x:xs'

B1. Case ys is []. Similar to A.

B2. Case ys is y:ys'

Case analysis on ys!

```
length (zip xs ys) =                -- zip Eq 3
length ((x,y):zip xs' ys') =        -- length Eq 2
1 + length (zip xs' ys') =          -- IH
1 + min (length xs') (length ys')
```

```
min (length xs) (length ys) =      -- length Eq 2
min (1 + length xs') (1 + length ys') = -- a law
1 + min (length xs') (length ys')
```

Done!

A law for map

Recall map

```
map            :: (a -> b) -> [a] -> [b]
map f []        = []
map f (x:xs) = f x : map f xs
```

Show that

```
map id = id where id = \x -> x
```

The equation compares two functions.
When can we say that two functions are equal?

Principle of extensionality

"Two functions are equal if they have the same value at every argument."

To show that

```
map id = id
```

we show that

```
map id xs = id xs
```

for all finite lists xs.

The map law again

A. Base case: xs is []

```
map id [] = []                          -- map Eq 1
```

```
id [] = []                              -- id Eq 1
```

B. Induction step: xs is x:xs'

```
map id (x:xs') =                        -- map Eq 2
id x : map id xs' =                     -- id Eq 1
x : map id xs' =                        -- IH
x : id xs' =                            -- id Eq 1
x:xs'
```

```
id (x:xs') = x:xs'                      -- id Eq 1
```

Done!

A law for reverse

Recall that

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

Show that

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

A. Base case: xs is []

```
reverse (reverse []) =          -- reverse Eq 1
reverse [] =                    -- reverse Eq 1
[]
```

B. Induction step: xs is x:xs’

```
reverse (reverse (x:xs’)) =   -- reverse Eq 2
reverse (reverse xs’ ++ [x])
```

There is nothing more we can do to the lhs; let us turn to the rhs

```
x : xs’ =                       -- IH
x : reverse (reverse xs’)
```

We are left with proving that

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

where we choose to write xs instead of xs’

Trying to prove the result by induction does not work. Generalise the equation; show that

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

A. Base case: ys is []

```
reverse ([] ++ [x]) =  -- (++) Eq 1
reverse [x] =           -- abbreviation
reverse (x:[]) =        -- reverse Eq 2
reverse [] ++ [x] =     -- reverse Eq 1
[] ++ [x] =             -- (++) Eq 1
[x]
```

```
x : reverse [] =        -- reverse Eq 1
x : [] =                -- abbreviation
[x]
```

B. Induction step: ys is y:ys'

```
reverse ((y:ys') ++ [x]) =        -- (++) Eq 2
reverse (y : (ys' ++ [x])) =      -- reverse Eq 2
reverse (ys' ++ [x]) ++ [y] =     -- IH
(x : reverse ys') ++ [y] =        -- (++) Eq 2
x : (reverse ys' ++ [y])
```

```
x : reverse (y:ys') =             -- reverse Eq 2
x : (reverse ys' ++ [y])
```

Done!

A method

1. Decide on the induction variable. Inspect the definitions of the functions involved.
For example, to prove that

```
length (xs ++ ys) = length xs + length ys
```

choose xs, because ++ is defined by pattern matching on its first argument:

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

2. For each case (base and step), expand each side of the equation (4 in total), to reach a common expression.
For example, to show that

```
map id = id
```

show that

```
map id [] = …       = []
id [] = …           = []
map id (x:xs’) = … = x:xs’
id (x:xs’) = …     = x:xs’
```

3. For the induction step, look for an opportunity to use the induction hypothesis.

```
map id (x:xs') = id x : map id xs'
```

IH here!

4. If you cannot use the IH, look for an auxiliary result (a law) that would allow you to progress.

5. Concentrate on your goal: “To make the two sides of the equation equal”.

6. In either case (base, step) you may have to proceed by case analysis on a variable, other than the induction variable. For example, in:

```
length (zip xs ys) =
        min (length xs) (length ys)
```

The induction step needs a case analysis on ys.

7. Present your proofs, justifying each step.

```
map id (x:xs') =          -- map Eq 2
id x : map id xs' =        -- IH
id x : id xs' =            -- id Eq 1
x : id xs' =               -- id Eq 1
x : xs'
```

```
id (x:xs') =               -- id Eq 1
x : xs'
```

Possible justifications are:

- Equations, laws
- Induction hypothesis
- Arithmetic, abbreviations

Exercises

(1) Write a specification for ++, based on !! and on length. Prove that ++ is correct with respect to your specification.

(2) Prove the following laws

```
map f . reverse = reverse . map f
drop m . drop n = drop (m + n)
```

(3) Show that n! = factp n 1 where

```
factp:: Int -> Int -> Int
factp 0 p = p
factp n p = factp (n-1) (n*p)
```