Programming In Haskell Chapter 9

CS 1JC3

Reasoning About Programs

How do we understand what a function does?

- Evaluate it in for particular inputs in GHCi
- Do the same thing by hand, performing a line by line calculation
- We can try to reason about how the program behaves in general

Reasoning About Programs

- Consider a simple function definition by pattern matching

```haskell
sum :: [Integer] -> Integer
sum []     = 0            -- def sum.1
sum (x:xs) = x + sum xs   -- def sum.2
```

Reasoning About Programs

- Consider a simple function definition by pattern matching

```haskell
sum :: [Integer] -> Integer
sum []     = 0              -- def sum.1
sum (x:xs) = x + sum xs     -- def sum.2
```

- We can reason about how the function should behave by defining a property it should hold

```haskell
sumOneProp :: Integer -> Bool
sumOneProp x = sum [x] == x
```

- and test it by running it with QuickCheck

```haskell
quickCheck sumOneProp
```

Alternative: Test with QuickCheck

- Alternatively, let us try and *prove* a certain property by reasoning about case definitions, for example

- ```
      -- Proof
  ```
  ```

Alternative: Test with QuickCheck

- Alternatively, let us try and *prove* a certain property by reasoning about case definitions, for example

- ```
  -- Proof
  sum [x]
    = sum (x:[])     -- by def of a list
    = x + sum []     -- by def sum.2
    = x + 0          -- by def sum.1
    = x              -- integer arithmetic
  ```
  ```

Definedness & Termination

Consider the function for computing factorials defined as

```haskell
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

Try entering the following into GHCi, what happens?

```haskell
fact (-1)
True || fact (-1)
```

Definedness & Termination

Consider the function for computing factorials defined as

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

Try entering the following into GHCi, what happens?

```
fact (-1)
True || fact (-1)
```

Definition: We say a program is *defined* on inputs for which it terminates

A Bit of Logic

Implication ($\to$) - the logical if-then
(i.e $a \to b$ specifies if a is True then b must also be True)

- Definition 1:

```haskell
a ==> b = case (a,b) of
            (True,False) -> False
            _            -> True
```

- Definition 2:

```haskell
a ==> b = (not a) || b
```

Testing Properties with Assumptions

Consider the following QuickCheck property for fact

```
factProp :: Int -> Bool
factProp n = (fact n) `div` n == fact (n-1)
```

What happens if you call quickCheck factProp?

Challenge: create a property that tests fact only when it is defined

Testing Properties with Assumptions

Consider the following QuickCheck property for fact

```
factProp :: Int -> Bool
factProp n = (fact n) `div` n == fact (n-1)
```

What happens if you call quickCheck factProp?
Challenge: create a property that tests fact only when it is defined

```
factProp :: Integer -> Bool
factProp n = (n > 0) ==>
                ((fact n) `div` n == fact (n-1))
```

Proving factProp

Now that we've done a "sanity check", prove that factProp holds when defined

For reference

```
fact :: Integer -> Integer
fact 0 = 1                     -- def fact.1
fact n = n * fact (n-1)        -- def fact.2
```

Proving factProp

Now that we've done a "sanity check", prove that factProp holds when defined

For reference

```
fact :: Integer -> Integer
fact 0 = 1                      -- def fact.1
fact n = n * fact (n-1)         -- def fact.2
```

```
-- Proof
(fact n) `div` n
    = (n * fact (n-1)) `div` n -- by def fact.2
    = fact (n-1)               -- integer arithmetic
```

Structural Induction On Lists

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

- **Base Case** Prove $P([])$ outright
- **Induction Step** Prove $P(xs) \rightarrow P(x : xs)$

Note: $P(xs)$ is known as the *Induction Hypothesis*, i.e we assume it to be true and prove $P(x : xs)$

Example: Structural Induction On Lists

- Consider the reverse function, defined as

```haskell
reverse :: [a] -> [a]
reverse []     = []                       -- def rev.1
reverse (x:xs) = reverse xs ++ [x]        -- def rev.2
```

- We might reason about reverse with the QuickCheck property

```haskell
reverseProp :: [Integer] -> Bool
reverseProp x = reverse [x] == [x]
```

- Then we can construct a pretty straight forward proof,

Example: Structural Induction On Lists

- Consider the reverse function, defined as

```haskell
reverse :: [a] -> [a]
reverse []     = []                       -- def rev.1
reverse (x:xs) = reverse xs ++ [x]        -- def rev.2
```

- We might reason about reverse with the QuickCheck property

```haskell
reverseProp :: [Integer] -> Bool
reverseProp x = reverse [x] == [x]
```

- Then we can construct a pretty straight forward proof,

```haskell
-- Proof
  reverse [x]
   = reverse (x:[])      -- by def list
   = [x] ++ reverse []   -- by rev.2
   = [x]  ++ []          -- by rev.1
   = [x]                 -- eval (++)
```

Example: Structural Induction On Lists

Let's more thoroughly test this implementation with the QuickCheck property

```haskell
reverseProp2 :: [Integer] -> Bool
reverseProp2 xs = reverse (reverse xs) == xs
```

We prove our reverseProp2 using Structural Induction, we'll start with the Base Case

```haskell
-- Base Case:
```

Example: Structural Induction On Lists

Let's more thoroughly test this implementation with the QuickCheck property

```haskell
reverseProp2 :: [Integer] -> Bool
reverseProp2 xs = reverse (reverse xs) == xs
```

We prove our reverseProp2 using Structural Induction, we'll start with the Base Case

```haskell
-- Base Case:

  reverse (reverse [])
   = reverse []          -- by rev.1
   = []                  -- by rev.1
```

Example: Structural Induction On Lists

Now lets prove the Inductive Step, this is much more difficult, and is reasonably done by taking some liberties

To simplify the proof, lets use the following lemma

```
-- Lemma:
   reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Example: Structural Induction On Lists

Now lets prove the Inductive Step, this is much more difficult, and is reasonably done by taking some liberties

To simplify the proof, lets use the following lemma

```
-- Lemma:
   reverse (xs ++ ys) == reverse ys ++ reverse xs
```

```
-- Inductive Step:

 reverse (reverse (x:xs))
  = reverse (reverse xs ++ [x])          -- by rev.2
  = reverse [x] ++ reverse (reverse xs)  -- by lemma
  = [x] ++ reverse (reverse xs)          -- by prev proof
  = [x] ++ xs                            -- by induc. hyp.
  = x:xs                                 -- eval (++)
```

Example: Structural Induction On Lists Continued

Consider the following functions,

```
length :: [a] -> Int
length []     = 0                 -- def len.1
length (x:xs) = 1 + length xs     -- def len.2
```

and

```
take :: Integer -> [a] -> [a]
take _ [] = []                          -- def take.1
take n (x:xs)
    | n > 0     = x : take (n-1) xs     -- def take.2
    | otherwise = []                    -- def take.3
```

Example: Structural Induction On Lists Continued

Consider the following functions,

```
length :: [a] -> Int
length []     = 0                -- def len.1
length (x:xs) = 1 + length xs    -- def len.2
```

and

```
take :: Integer -> [a] -> [a]
take _ [] = []                       -- def take.1
take n (x:xs)
    | n > 0     = x : take (n-1) xs  -- def take.2
    | otherwise = []                 -- def take.3
```

We can reason about them together using the QuickCheck property

```
takeLenProp xs = take (length xs) xs == xs
```

Example: Structural Induction on Lists Continued

Once again, lets now prove our function using Structural Induction, starting with the Base Case

Example: Structural Induction on Lists Continued

Once again, lets now prove our function using Structural Induction, starting with the Base Case

```
-- Base Case
take (length []) []
  = []              -- by take.1
```

And then the Inductive Step

Example: Structural Induction on Lists Continued

Once again, lets now prove our function using Structural Induction, starting with the Base Case

```
-- Base Case
take (length []) []
  = []              -- by take.1
```

And then the Inductive Step

```
-- Inductive Step
take (length (x:xs)) (x:xs)
  = take (1 + length xs) (x:xs)       -- by len.2
  = x : take (1 + length xs -1) xs)   -- by take.2
  = x : take (length xs) xs           -- integer arith.
  = x : xs                            -- induc. hyp.
```

Tips on Reasoning, Proving Programs

- Evaluating Integer arithmetic is ok, but it's NOT OK TO DO THE SAME FOR FLOATING POINT ARITHMETIC

Tips on Reasoning, Proving Programs

- Evaluating Integer arithmetic is ok, but it's NOT OK TO DO THE SAME FOR FLOATING POINT ARITHMETIC
- If something is trivially true, but a proof for it is not immediately obvious, assume it as a lemma and then perhaps try to prove it later. Even though you really don't have a proof until the lemma is verified, it stills helps with reasoning about your program

Tips on Reasoning, Proving Programs

- Evaluating Integer arithmetic is ok, but it's NOT OK TO DO THE SAME FOR FLOATING POINT ARITHMETIC
- If something is trivially true, but a proof for it is not immediately obvious, assume it as a lemma and then perhaps try to prove it later. Even though you really don't have a proof until the lemma is verified, it stills helps with reasoning about your program
- Test properties with QuickCheck before trying to prove them or assuming something as a lemma. You can consider this a "sanity check" that will help prevent you from trying to prove un-valid properties

Exercise 1

For the following definition of $(++)$

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys                  -- def ++.1
(x:xs) ++ ys = x : (xs ++ ys)      -- def ++.2
```

Provide a proof for the following property

```
listPlusProp x xs = [x] ++ xs == x:xs
```

Solution 1

```
-- Proof
  [x] ++ xs
   = (x:[]) ++ xs      -- by def of a list
   = x : ([] ++ xs)    -- by ++.2
   = x : xs            -- by ++.1
```

Note: you can refer previous proof obligations that simply claimed "eval (++)" to this

Exercise 2

Consider the function

```
average :: [Integer] -> Integer
average xs = sum xs `div` length xs
```

Write a QuickCheck property that tests average for when it is defined

Solution 2

```haskell
averageProp :: [Integer] -> Bool
averageProp xs = not (length xs > 0) ||
                 (average xs * length xs <= sum xs)
```

Recall: the definition of implication

```haskell
(==>) :: Bool -> Bool -> Bool
a ==> b = (not a) || b
```

Exercise 3

Recall the function

```
sum :: Num a => [a] -> a
sum []     = 0            -- def sum.1
sum (x:xs) = x + sum xs -- def sum.2
```

Prove the following property

```
sumProp2 :: ([Integer],[Integer]) -> Bool
sumProp2 (xs,ys) =  sum (xs ++ ys) == sum xs + sum ys
```

You're allowed to use the following

```
-- Lemma
   x : (xs ++ ys) == (x:xs ++ ys)
```

Solution 3

```
-- Base Case
  sum [] + sum ys
   = 0 + sum ys       -- by sum.1
   = sum ys           -- integer arith
   = sum ([] ++ ys) -- by ++.1
```

Solution 3

```
-- Base Case
  sum [] + sum ys
   = 0 + sum ys       -- by sum.1
   = sum ys           -- integer arith
   = sum ([] ++ ys) -- by ++.1

-- Inductive Step
  sum (x:xs) + sum ys
    = x + sum xs + sum ys   -- by sum.2
    = x + sum (xs ++ ys)    -- induc. hyp.
    = sum (x : (xs ++ ys)) -- by sum.2
    = sum (x:xs ++ ys)      -- by lemma
```

Exercise 4

Recall the function

```
length :: [a] -> Integer
length []     = 0              -- def len.1
length (x:xs) = 1 + length xs -- def len.2
```

Prove the QuickCheck Property

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

You're allowed to use the same lemma as last slide

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

Solution 4

```
-- Base Case
 length [] + length ys
  = 0 + length ys        -- by len.1
  = length ys            -- integer arithm.
  = length ([] ++ ys)  -- by ++.1
```

Solution 4

```
-- Base Case
 length [] + length ys
  = 0 + length ys        -- by len.1
  = length ys            -- integer arithm.
  = length ([] ++ ys) -- by ++.1

-- Inductive Step
  length (x:xs) + length ys
    = 1 + length xs + length ys -- by len.2
    = 1 + length (xs ++ ys)      -- induc. hyp.
    = length (x : (xs ++ ys))    -- by len.2
    = length (x:xs ++ ys)        -- by lemma
```