Programming in Haskell

S P Suresh

http://www.cmi.ac.in/~spsuresh

Lecture 8

September 4, 2017

Functions and types

- ```
  mylength [] = 0
  mylength (x:xs) = 1 + mylength xs
  ```
- ```
  myreverse [] = []
  myreverse (x:xs) = (myreverse xs) ++ [x]
  ```
- ```
  myinit [x] = []
  myinit (x:xs) = x:(myinit xs)
  ```
- None of these functions look into the elements of the list
  - Will work over lists of any type!
  ```

Polymorphism

- Functions that work across multiple types
- Use type variables to denote flexibility
  - `a`, `b`, `c` are place holders for types
  - `[a]` is a list whose elements are of type `a`

Polymorphism ...

- Types for our list functions
- `mylength :: [a] -> Int`
- `myreverse :: [a] -> [a]`
- `myinit :: [a] -> [a]`
- All occurrences of `a` in a type definition must be instantiated in the same way

Functions and operators

- `+`, `-`, `/`, … are operators — infix notation
  - `3+5`, `11-7`, `8/9`
- `div`, `mod` … are functions —prefix notation
  - `div 7 5`, `mod 11 3`
- Use operators as functions: `(+)`, `(-)` …
  - `(+) 3 5`, `(-) 11 7`, `(/) 8 9`
- Use (binary) functions as operators: `` `div` ``, `` `mod` ``
- `` 7 `div` 5 ``, `` 11 `mod` 3 ``

Functions and operators …

- `plus :: Int -> Int -> Int`
  `plus m n = m + n`

- `(plus m) :: Int -> Int` adds `m` to its argument

- Likewise, `m + n` is the same as `(+) m n`

- Hence `(+ m)` and `(m +)`, like `(plus m)` adds `m` to the argument

- `(+17) 7 = 24`
  `(17+) 7 = 24`

Functions and operators …

- ```
  (5*) 3 = 15
  (*5) 3 = 15
  ```
- ```
  (5/) 3 = 1.666..
  (/5) 3 = 0.6
  ```
- ```
  (5-) 3 = 2
  (-5) 3 = ??
  ```
- ```
  subtract :: Int -> Int -> Int
  subtract m n = n - m
  ```
- Use `(subtract 5) 3` instead

Higher order functions

- Can pass functions as arguments
- `apply f x = f x`
  - Applies first argument to second argument
- What is the type of `apply`?
  - A generic function `f` has type `f :: a -> b`
  - Argument `x` and output must be compatible with `f`
- `apply :: (a -> b) -> a -> b`

Higher order functions

- Sorting a list of objects
  - Need to compare pairs of objects
  - What quantity is used for comparison?
  - Ascending, descending?
- Pass a comparison function along with the list to the sort function

Summary

- Haskell functions can be polymorphic
  - Operate on values of more than one type
- Notation to use operators as functions and vice versa
- Higher order functions
  - Arguments can themselves be functions

Applying a function to a list

- ```
  touppercase :: String -> String
  touppercase ""     = ""
  touppercase (c:cs) = (capitalize c):
                          (touppercase cs)
  ```

- ```
  sqrlist :: [Int] -> [Int]
  sqrlist [] = []
  sqrlist (x:xs) = sqr x : (sqrlist xs)
  ```

- Apply a function `f` to each member in a list

- Built in function `map`

  - `map f [x0,x1,…,xk]` ⇛ `[(f x0),(f x1),…,(f xk)]`

Examples

- `map (+ 3) [2,6,8] = [5,9,11]`
- `map (* 2) [2,6,8] = [4,12,16]`
- Given a list of lists, sum the lengths of inner lists
- ```
  sumLength:: [[Int]] -> Int
  sumLength [] = 0
  sumLength (x:xs) = length x + (sumLength xs)
  ```
- Can be written using `map` as:
- `sumLength l = sum (map length l)`
  ```

The function `map`

- The function `map`
- ```
  map f [] = []
  map f (x:xs) = (f x):(map f xs)
  ```
- What is the type of `map`?
- `map :: (a -> b) -> [a] -> [b]`
  ```

Selecting elements in a list

- Select all even numbers from a list

- ```
  even_only :: [Int] -> [Int]
  even_only [] = []
  even_only (x:xs)
    | is_even x = x:(even_only xs)
    | otherwise = even_only xs
    where
      is_even :: Int -> Bool
      is_even x = (mod x 2) == 0
  ```
  ```

Filtering a list

- `filter` selects all items from list `l` that satisfy property `p`

- ```
  filter p [] = []
  filter p (x:xs)
    | (p x)     = x:(filter p xs)
    | otherwise = filter p xs
  ```

- `filter :: (a -> Bool) -> [a] -> [a]`

- `even_only l = filter is_even l`
  ```

Combining `map` *and* `filter`

- Extract all the vowels in the input and capitalize them
- `filter` extracts the vowels, `map` capitalizes them
- ```
  cap_vow :: [Char] -> [Char]
  cap_vow l = map toUpper (filter is_vowel l)
  ```
- ```
  is_vowel :: Char -> Bool
  is_vowel c =  (c=='a') || (c=='e') ||
                (c=='i') || (c=='o') ||
                (c=='u')
  ```

Combining `map` *and* `filter`

- Squares of even numbers in a list
- ```
  sqr_even :: [Int] -> [Int]
  sqr_even l = map sqr (filter is_even l)
  ```
  ```

Summary

- `map` and `filter` are higher order functions on lists
- `map` applies a function to each element
- `filter` extracts elements that match a property
- `map` and `filter` are often combined to transform lists

New lists from old

- Set comprehension
  - $M = \{ x^2 \mid x \in L, even(x) \}$
- Generates a new set M from a given set L
- Haskell allows this almost verbatim
  - `[ x*x | x <- l, is_even(x) ]`
- List comprehension, combines `map` and `filter`

Examples

- Divisors of *n*
- ```haskell
  divisors n = [x | x <- [1..n],
                    (mod n x) == 0]
  ```
- Primes below *n*
- ```haskell
  primes n = [x | x <- [1..n],
                  (divisors x == [1,x])]
  ```

Examples ...

- Can use multiple generators
- Pairs of integers below *10*
  - `[(x,y) | x <- [1..10], y <- [1..10]]`
- Like nested loops, later generators move faster
  - `[(1,1), (1,2),..., (1,10), (2,1), ..., (2,10), ..., (10,10)]`

Examples ...

- The set of Pythogorean triples below *100*

    - ```
      [(x,y,z) | x <- [1..100],
                 y <- [1..100],
                 z <- [1..100],
                 x*x + y*y == z*z]
      ```

- Oops, that produces duplicates.

    - ```
      [(x,y,z) | x <- [1..100],
                 y <- [(x+1)..100],
                 z <- [(y+1)..100],
                 x*x + y*y == z*z]
      ```

Examples …

- The built-in function **`concat`**
  - **`concat l = [x | y <- l, x <- y]`**

Examples ...

- Given a list of lists, extract all even length non-empty lists
- ```
  even_list l =
      [ (x:xs) | (x:xs) <- l,
                 (mod (length (x:xs)) 2) == 0 ]
  ```
- Given a list of lists, extract the head of all the even length non-empty lists
- ```
  head_of_even l =
      [ x | (x:xs) <- l,
            (mod (length (x:xs)) 2) == 0 ]
  ```

The Sieve of Eratosthenes

- Start with the (infinite) list *[2,3,4,…]*
- Enumerate the left most element as next prime
- Remove all its multiples from the list
- Repeat the above with this list

The Sieve of Eratosthenes

In Haskell,

```
primes = sieve [2..]
  where
  sieve (x:xs) =
     x:(sieve [y | y <- xs, y `mod` x /= 0])
```

The Sieve of Eratosthenes

```
primes ⇛ sieve [2..]

⇛ 2:(sieve [ y | y <- [3..] , y `mod` 2 /= 0])

⇛ 2:(sieve (3:[y | y <- [4..], y `mod` 2 /= 0])

⇛ 2:(3:(sieve [z |
     z <- [y | y <- [4..], y `mod` 2 /= 0] |
                                 z `mod` 3 /= 0])

⇛ 2:(3:(5:(sieve [w |
     w <- [z |
        z <- [y | y <- [4..], y `mod` 2 /= 0] |
                             z `mod` 3 /= 0] |
                                 w `mod` 5 /= 0])

⇛ ...
```

Summary

- List comprehension is a succinct, readable notation for combining map and filter
- Can translate list comprehension in terms of `concat`, `map`, `filter` (next class)