Programming in Haskell
Aug-Nov 2015

LECTURE 7

AUGUST 25, 2015

S P Suresh
Chennai Mathematical Institute

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

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 touppercase (filter is_vowel l)

is_vowel :: Char -> Char
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)
```

New lists from old

- Set comprehension

$$M = \{ x^2 \mid x \in L, \text{even}(x) \}$$

- 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

```
divisors n = [x | x <- [1..n],
                  (mod n x) == 0]
```

- Primes below n

```
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 ]
```

Translating list comprehensions

- List comprehension can be rewritten using `map`, `filter` and `concat`
- A list comprehension has the form

```
[e | q1, q2, ..., qN]
```

where each `qj` is either

- a boolean condition `b` or
- a generator `p <- l`, where `p` is a pattern and `l` is a list valued expression

Translating …

- A boolean condition acts as a filter.

```
[e | b,Q] = if b then [e | Q] else []
```

- Depends only on generators/qualifiers to its left

Translating …

- Generator `p <- l` produces a list of candidates
- Naive translation

```
[e | p <- l, Q] = map f l
  where
  f p = [e | Q]
  f _ = []
```

Translating ...

* `[n*n | n <- [1..7], mod n 2 == 0]`

  ⇛ `map f [1..7]`
  `  where`
  `  f n = [ n*n | mod n 2 == 0]`

⇛ `map f [1..7]`
`  where`
`  f n = if (mod n 2 == 0) then [n*n] else []`

⇛ `[[],[4],[],[16],[],[36],[]]`

Translating …

- Need an extra `concat` when translating `p <- l`
- Correct translation

```
[e | p <- l, Q] = concat map f l
  where
  f p = [e | Q]
  f _ = []
```

Translating ...

* `[n*n | n <- [1..7], mod n 2 == 0]`

  ⇛ `concat map f [1..7]`
  `  where`
  `  f n = [ n*n | mod n 2 == 0]`

⇛ `concat map f [1..7]`
`  where`
`  f n = if (mod n 2 == 0) then [n*n] else []`

⇛ `concat [[],[4],[],[16],[],[36],[]]`

⇛ `[4,16,36]`

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

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 …

The Sieve of Eratosthenes

- In Haskell,

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

The Sieve of Eratosthenes

```
primes ⇛ sieve [2..]

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

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

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

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