Programming in Haskell
Aug-Nov 2015

LECTURE 6

AUGUST 20, 2015

S P Suresh
Chennai Mathematical Institute

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
  - An “answer” is an expression that cannot be further simplified
- Built-in simplifications

```
3+5 ⇛ 8
```

```
True || False ⇛ True
```

Computation as rewriting

- Simplifications based on user defined functions

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```

Computation as rewriting

- power 3 2

  ⇛ 3 * (power 3 (2-1)) user definition

  ⇛ 3 * (power 3 1) built in simplification

  ⇛ 3 * (3 * (power 3 (1-1))) user definition

  ⇛ 3 * (3 * (power 3 0)) built in simplification

  ⇛ 3 * (3 * 1) user definition

  ⇛ 3 * 3 built in simplification

  ⇛ 9 built in simplification

Order of evaluation

- `(8+3)*(5-3)` ⇛ `11*(5-3)` ⇛ `11*2` ⇛ `22`

  `(8+3)*(5-3)` ⇛ `(8+3)*2` ⇛ `11*2` ⇛ `22`

- `power (5+2) (4-4)` ⇛ `power 7 (4-4)`
  ⇛ `power 7 0 = 1`

  `power (5+2) (4-4)` ⇛ `power (5+2) 0` ⇛ `1`

- What would `power (div 3 0) 0` return?

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- In `head (2:reverse [1..5])`
  - `f` is head
  - `e` is `(2:reverse [1..5])`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

Lazy evaluation …

- The argument is not evaluated if the function definition does not force it to be evaluated.
  - `head (2:reverse [1..5])` ⇛ `2`
- Argument is evaluated if needed
  - `last (2:reverse [1..5))` ⇛
    `last (2:[5,4,3,2,1])` ⇛ `1`

Lazy evaluation …

- What would `power (div 3 0) 0` return?

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```

- First definition ignores value of `x`

- `power (div 3 0) 0` returns `1`

Lazy evaluation …

- If all simplifications are possible, order of evaluation does not matter, same answer

- One order may terminate, another may not

- Lazy evaluation expands arguments by “need”

  - Can terminate with an undefined sub-expression if that expression is not used

Infinite lists

```
infinite_list :: [Int]
infinite_list = inflistaux 0
    where
    inflistaux :: Int -> [Int]
    inflistaux n = n:(inflistaux (n+1))
```

- infinite_list ⇛ [0,1,2,3,4,5,6,7,8,9,10,12,…]
- head (infinite_list) ⇛ head(0:inflistaux 1) ⇛ 0
- take 2 (infinite_list) ⇛
  take 2 (0:inflistaux 1) ⇛
  0:(take 1 (inflistaux 1)) ⇛
  0:(take 1 (1:inflistaux 2)) ⇛ [0,1]

Infinite lists

- Range notation extends to infinite lists
  - `[m..]` ⇛ `[m,m+1,m+2,…]`
  - `[m,m+d..]` ⇛ `[m,m+d,m+2d,m+3d,…]`
- Sometimes infinite lists simplify function definition

Summary

- In functional programming, computation is rewriting
- Haskell uses lazy evaluation — simplifies outermost expression first
- Lazy evaluation allows us to work with infinite lists

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 of type `a`

Polymorphism …

- Types for our list functions

```
mylength :: [a] -> Int

myreverse :: [a] -> [a]

myinit :: [a] -> [a]
```

- All `a`'s 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 its 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 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)
```

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