Programming in Haskell: Lecture 9

S P Suresh

September 9, 2019

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible

Computation as rewriting

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

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

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`

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`	multiplication
`---> 9`	multiplication

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`
  - `(8+3)*(5-3) ---> (8+3)*2 ---> 11*2 ---> 22`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`
  - `(8+3)*(5-3) ---> (8+3)*2 ---> 11*2 ---> 22`
- Two ways of computing `power (5+2) (4-4)`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`
  - `(8+3)*(5-3) ---> (8+3)*2 ---> 11*2 ---> 22`
- Two ways of computing `power (5+2) (4-4)`
  - `power (5+2) (4-4) ---> power 7 (4-4) ---> power 7 0 ---> 1`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`
  - `(8+3)*(5-3) ---> (8+3)*2 ---> 11*2 ---> 22`
- Two ways of computing `power (5+2) (4-4)`
  - `power (5+2) (4-4) ---> power 7 (4-4) ---> power 7 0 ---> 1`
  - `power (5+2) (4-4) ---> power (5+2) 0 ---> 1`

Order of evaluation

- Two ways of computing `(8+3) * (5-3)`
  - `(8+3)*(5-3) ---> 11*(5-3) ---> 11*2 ---> 22`
  - `(8+3)*(5-3) ---> (8+3)*2 ---> 11*2 ---> 22`
- Two ways of computing `power (5+2) (4-4)`
  - `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 (3 `div` 0) 0 `` return?

Lazy evaluation

- Any Haskell expression is of the form `f e`

Lazy evaluation

- Any Haskell expression is of the form `f e`
  - `f` is the outermost function `e` is the expression to which it is applied.

Lazy evaluation

- Any Haskell expression is of the form `f e`
  - `f` is the outermost function `e` is the expression to which it is applied.
- In `head (2:reverse [1..5])`

Lazy evaluation

- Any Haskell expression is of the form `f e`
  - `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]`

Lazy evaluation

- Any Haskell expression is of the form `f e`
  - `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

Lazy evaluation

- The argument is not evaluated if the function definition does not force it to be evaluated
- `head (2:reverse [1..5]) ---> 2`

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

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 (3 `div` 0) 0` return?

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

Lazy evaluation

- What would `power (3 `div` 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`

Lazy evaluation

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

Lazy evaluation

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

Lazy evaluation

- If all simplifications are possible, order of evaluation does not matter, same answer
- One order may terminate, another may not

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**

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

```
      infList :: [Integer]
      infList = infFrom 0
      infFrom :: Integer -> [Integer]
      infFrom n = n: infFrom (n+1)

      infList ---> [0,1,2,3,4,5,6,7,8,9,10,11,12,...]

      head infList
---> head (infFrom 0)
---> head (0:infFrom (0+1))
---> 0
```

Infinite lists

```
     infList = infFrom 0
     infFrom n = n: infFrom (n+1)


     take 2 infList
---> take 2 (infFrom 0)
---> take 2 (0:infFrom (0+1))
---> 0:take 1 (infFrom (0+1))
---> 0:take 1 (infFrom 1)
---> 0:take 1 (1:infFrom (1+1))
---> 0:1:take 0 (infFrom (1+1))
---> 0:1:[]
```

Infinite lists

- Range notation extends to infinite lists

Infinite lists

- Range notation extends to infinite lists
- `[m..] = [m, m+1, m+2, ...]`

Infinite lists

- Range notation extends to infinite lists
- `[m..] = [m, m+1, m+2, ...]`
- `[m, m+d..] = [m, m+d, m+2d, m+3d, ...]`

Infinite lists

- Range notation extends to infinite lists
- `[m..] = [m, m+1, m+2, ...]`
- `[m, m+d..] = [m, m+d, m+2d, m+3d, ...]`
- Using infinite lists often simplifies programs

Functions and types

- Consider these definitions

```
myLength []      = 0
myLength (x:xs)  = 1 + myLength xs

myReverse []     = []
myReverse (x:xs) = myReverse xs ++ [x]

myInit [x]       = []
myInit (x:xs)    = x:myInit xs
```

Functions and types

- Consider these definitions

```
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

Functions and types

- Consider these definitions

```
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

Polymorphism

- Functions that work across multiple types
- Use type variables to denote flexibility

Polymorphism

- Functions that work across multiple types
- Use type variables to denote flexibility
- `a`, `b`, `c` are place holders for types

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 elements of type `a`

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 elements of type `a`
- Types for our list functions

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 elements of type `a`
- Types for our list functions
    - `myLength :: [a] -> Int`

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 elements of type `a`
- Types for our list functions
  - `myLength :: [a] -> Int`
  - `myReverse :: [a] -> [a]`

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 elements of type `a`
- Types for our list functions
  - `myLength :: [a] -> Int`
  - `myReverse :: [a] -> [a]`
  - `myInit :: [a] -> [a]`

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 elements of type `a`
- Types for our list functions
  - `myLength :: [a] -> Int`
  - `myReverse :: [a] -> [a]`
  - `myInit :: [a] -> [a]`
- All `a`'s in the type should be instantiated in the same way

Higher-order functions

- Most functions produce a function as result

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`
- Hence `apply :: (a -> b) -> a -> b`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`
- Hence `apply :: (a -> b) -> a -> b`
- Same as the built-in `($)`

The built-in function `map`

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

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list

The built-in function `map`

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

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list
- Built in function `map` achieves this

The built-in function **map**

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

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list
- Built in function **`map`** achieves this
- `map f [x0, x1, ..., xk] ---> [f x0, f x1, ..., f xk]`

The built-in function `map`

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

The built-in function **map**

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

- Given a list of lists, sum the lengths of inner lists

```
sumLength:: [[Int]] -> Int
sumLength []     = 0
sumLength (x:xs) = length x + sumLength xs
```

The built-in function `map`

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

- 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 built-in function **map**

- The function **map**

```
map f []     = []
map f (x:xs) = f x: map f xs
```

The built-in 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]
```

The built-in function **filter**

- Select all even numbers from a list

```
allEvens :: [Int] -> [Int]
allEvens []                  = []
allEvens (x:xs) | even x     = x: allEvens xs
                | otherwise = allEvens xs
```

The built-in function `filter`

- Select all even numbers from a list

```
allEvens :: [Int] -> [Int]
allEvens []                  = []
allEvens (x:xs) | even x     = x: allEvens xs
                | otherwise = allEvens xs
```

- Abstract pattern:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                  = []
filter p (x:xs) | p x        = x: filter p xs
                | otherwise = filter p xs
allEvens = filter even
```

Combining **map** *and* **filter**

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

Combining **map** *and* **filter**

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

- Extract all vowels in a string and capitalize them

```
capVows :: String -> String
capVows = map toUpper . filter isVow
isVow c = c `elem` "aeiou"
```

Combining `map` *and* `filter`

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

- Extract all vowels in a string and capitalize them

```
capVows :: String -> String
capVows = map toUpper . filter isVow
isVow c = c `elem` "aeiou"
```

- `(.)` denotes function composition: `(f . g) e = f (g e)`