Haskell Overview II (2A)

Young Won Lim
9/26/16

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Haskell Tutorial, Medak & Navratil
ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

Yet Another Haskell Tutorial, Daume
https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf

Counting functions

```
l1 []          = 0
l1 (x:xs)      = 1 + l1 xs
```

Pattern Matching

```
l2 xs = if xs == [] then 0 else 1 + l2 (tail xs)
```

if-then-else

```
l3 xs | xs == []   = 0
      | otherwise  = 1 + l3 (tail xs)
```

guard notation

```
l4    = sum . map (const 1)
```

replace and sum

```
l5 xs = foldl inc 0 xs
    where inc x _ = x+1
```

local function, local counter

```
l6    = foldl' (\n _ -> n + 1) 0
```

lambda expression

Guard Notation

https://www.haskell.org/tutorial/patterns.html

```
sign x | x >  0 = 1
       | x == 0 = 0
       | x <  0 = -1
```

if (x > 0) *sign = +1;*
else if (x == 0) *sign = 0;*
else *sign= -1*

Anonymous Function

Input

Prompt> (\x -> x + 1) 4
5 :: Integer

Prompt> (\x y -> x + y) 3 5
8 :: Integer

addOne = \x -> x + 1

$$\lambda x \rightarrow x^2$$

\ x -> x*x

In Lambda Calculus
Lambda Expression
Lambda Abstraction
Anonymous Function

https://www.haskell.org/tutorial/patterns.html

Naming a Lambda Expression

inc x = x + 1

add x y = x + y

Input

inc = \x -> x + 1

add = \x y -> x + y

Lambda Expression

https://www.haskell.org/tutorial/patterns.html

Lambda Calculus

The lambda calculus consists of a language of **lambda terms**,
which is defined by a certain formal syntax,
and a set of transformation rules,
which allow manipulation of the lambda terms.

These transformation rules can be viewed
as an equational theory or as an operational definition.

All functions in the lambda calculus are anonymous functions,
having no names.

They only accept one input variable,
with currying used to implement functions with several variables.

Composite Function (1)

https://www.haskell.org/tutorial/patterns.html

```
(.)         ::  (b -> c) -> (a -> b) -> (a -> c)
                   f           g         f(g(x))


f . g       =  \ x -> f (g x)

(f . g) x   =  f (g x)
```

Composite Function (2)

```
p1 = (1.0,2.0,1.0) :: (Float, Float, Float)
p2 = (1.0,1.0,1.0) :: (Float, Float, Float)
ps = [p1,p2]
newPs       = filter real ps
rootsOfPs   = map roots newPs
```

```
RootsOfPs2 = (map roots . filter real) ps
```

Local Variables

```
lend amt bal = let  reserve    = 100
                    newBal = bal - amt
               in  if bal < reserve
                   then Nothing
                   else Just newBal
```

```
lend2 amt bal = if amt < reserve * 0.5
                then Just newBal
                else Nothing
    where reserve    = 100
          newBal = bal - amt
```

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Local Function

```
pluralise :: String -> [Int] -> [String]
pluralise word counts = map plural counts
    where plural 0 = "no " ++ word ++ "s"
          plural 1 = "one " ++ word
          plural n = show n ++ " " ++ word ++ "s"
```

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Local Function

roots :: (Float, Float, Float) -> (Float, Float)

type PolyT = (Float, Float, Float)
type RootsT = (Float, Float)

roots :: PolyT -> RootsT

typedef

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Infix operators as functions

Input

(x+) = \y -> x + y

Input

(+y) = \x -> x + y

partial application of an infix operator

Inputs

(+) = \x y -> x + y

https://www.haskell.org/tutorial/functions.html

Infix operators as function values

```
inc    =    (+1)

add    =    (+)


map  (+)  [1,2,3]

                     [(+1),(+2),(+3)]
```

https://www.haskell.org/tutorial/functions.html

Sections

```
(+), (*) :: Num a => a -> a -> a
```

```
(+) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
```

a belongs to the Num type class

```
? 3 + 4

? (+) 3 4
```

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Sections

```
(3 +) :: Num a => a -> a

? map (3 +) [4, 7, 12]



? filter (==2) [1,2,3,4] [2]
? filter (<=2) [1,2,3,4] [1,2]


? filter (2<) [1,2,3,4] [3,4]
? filter (<2) [1,2,3,4] [1]
? filter (2>) [1,2,3,4] [1]
? filter (>2) [1,2,3,4] [3,4]
```

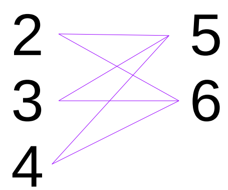
http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

List Comprehensions

```
[x | x <- [0..100], odd x]
```

$$\{x \mid x \in [0, 100], odd\ x\}$$

```
? f [2,3,4] [5,6] where f xs ys = [x*y | x <- xs, y <- ys]
[10,12,15,18,20,24] :: [Integer]
```



http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

List Comprehensions

```
[ (i,j) | i <- [1,2],
          j <- [1..3] ]
```

i=1 j=1
i=2 j=2
j=3

[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]

```
[h| (h:t) <- [[1, 2, 3], [4, 5, 6]]]
```

(h:t) = (1:[2,3])
(h:t) = (4:[5,6])

h = 1, 4

[1, 4]

```
[t | (h:t) <- [[1, 2, 3], [4, 5, 6]]]
```

(h:t) = (1:[2,3])
(h:t) = (4:[5,6])

t = [2,3], [5,6]

[[2,3], [5,6]]

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

List function map & filter as a list comprehension

```
map f xs = [ f x | x <- xs ]

filter p xs = [x | x <- xs, p x ]
```

additional condition
to be satisfied

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Currying

f x y = blabla

f x = \y -> blabla

f = \x -> (\y -> blabla)

f is a function with one argument, **x**,
and returns another function with one argument, **y**,
and then returns the actual result blabla.
This is known as currying.

(f x) y

http://stackoverflow.com/questions/3794371/haskell-currying-need-further-explanation

Curry & Uncurry

f :: a -> b -> c   curried form

Currying is the process of transforming a **function** that takes multiple arguments into a **function** that takes just a single argument and returns another function if any arguments are still needed.

g :: (a -> b) -> c   uncurried form

f = curry g
g = uncurry f

f x y	←	g (x, y)	currying
g(x, y)	←	f x y	uncurrying

f x y = g (x, y)

https://wiki.haskell.org/Currying

Functional & Imperative Programming

```
c := 0
for i:=1 to n do
    c := c + a[i] * b[i]
```

a belongs to the Num type class

uncurried form

inn2 :: Num a => ([a], [a]) -> a

Inn2 = foldr (+) 0 . map (uncurry (*)) . uncurry zip

(1) uncurry zip
(2) map (uncurry (*))
(3) foldr (+) 0

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Simple List Functions – zip and unzip functions

zip [a1, a2, a3] [b1,b2,b3]

[(a1,b1),(a2,b3),(a3,b3)]

f x y ⟵ g (x, y)		currying
g(x, y) ⟵ f x y		uncurrying

uncurry zip ([a1, a2, a3], [b1,b2,b3])

[(a1,b1),(a2,b2),(a3,b3)]

Simple List Functions – zip and unzip functions

map (uncurry (*)) [(a1,b1),(a2,b2),(a3,b3)]

[a1*b1, a2*b2, a3*b3]

map foldr (+) 0 [a1*b1, a2*b2, a3*b3]

a1*b1 + a2*b2 + a3*b3

Functional & Imperative Programming

a belongs to the Num type class

uncurried form

```
inn2 :: Num a => ([a], [a]) -> a

inn2 =  foldr (+) 0  .  map (uncurry (*))  .  uncurry zip

inn2 x = (foldr (+) 0   .   map (uncurry (*))   .   uncurry zip) x
inn2 x = (foldr (+) 0      (map (uncurry (*))      (uncurry zip x)))

testvec =  inn2 ([1,2,3], [4,5,6])
```

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

foldr, foldl, fold1

foldr (-) 1 [4,8,5]

4 - (foldr (-) 1 [8,5])
4 - (8 - foldr (-) 1 [5])
4 - (8 - (5 - foldr (-) 1 []))
4 - (8 - (5 - 1))
4 - (8 - 4)
4 - 4
0

(4 (8 (5)))
(4 (8 (5)))
(4 (8 (5)))

foldl (-) 1 [4,8,5]

(foldl (-) 1 [4,8]) - 5
((foldl (-) 1 [4]) - 8) – 5
((1 – 4) – 8) – 5
((-3) – 8) – 5
-11 -5
-16

(((4) 8) 5)
(((4) 8) 5)
(((4) 8) 5)

foldl (+) 0 [4,8,5]

foldl1 (+) [4,8,5]

foldl (*) 1 [4,8,5]

foldl1 (*) [4,8,5]

No starting value argument

Fold function applications

ft = foldr (*) 7 [2,3]
ut = uncurry (+) (5,6)
zt = zip "Haskell" [1,2,3,4,5,6,7]

(2 (3 * 7))
(2 * 3 *7)

(+) 5 6

uncurry (+) (5, 6)

innXa, innXb : [[Integer]] -> Integer

innXa = foldr (+) 0 . map (foldr (*) 1) . transpose

innXb = foldr1 (+) . map (foldr1 (*)) . transpose

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

Transpose

transpose :: [[a]] -> [[a]]

transposes the rows and columns of its argument.

transpose [[a,b,c],[1,2,3]] == [[a,1],[b,2],[c,3]]

elements may be skipped

transpose [[10,11],[20,],[],[30,31,32]] == [[10,20,30],[11,31],[32]]

[[10,11, -],[20, - , -],[- , - , -],[30,31,32]]
[[10,20, - , 30], [11, - , - , 31], [- , - , - , 32]]
[[10,20, 30], [11, 31], [32]]

https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-List.html

Prepend (cons) Operator :

1 : 2 : 3 : 4 : []

is the list [1, 2, 3, 4].

4 : []	(cons 4 to the empty list)
3 : [4]	(cons 3 onto the list containing 4)
2 : [3,4]	(cons 2 onto the list containing 3, 4)
1 : [2,3,4]	(cons 1 onto the list containing 2,3,4)

http://stackoverflow.com/questions/1696751/what-does-the-infix-operator-do-in-haskell

Transpose

```
transpose :: [[a]] -> [[a]]

transpose []              = []
transpose ([]:xss)        = transpose xss
transpose ((x:xs) : xss)  = (x : [h | (h:t) <- xss]) :
                               transpose (xs : [t | (h:t) <- xss])
```

```
((a:[a]) : [[a]])
((x:xs) : xss)

((x:xs) : (h:t))
((a:[a]) :([a]:[[a]])

(x:h) : transpose (xs:t)
(a:[a]) :             ([a]:[[a]])
```

List Comprehension
[x | x <- [0..100], odd x]
$\{x \mid x \in [0,100], odd\ x\}$

https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-List.html

Transpose Example (1)

```
[[1,2,3], [4,5,6],[1,1,1]]

[1,2,3] : [[4,5,6],[1,1,1]]
(1 : [2,3])  : ([4,1] : [[5,6],[1,1]])

(1 : [4,1]) : transpose ( [2,3] : [[5,6],[1,1]])
[1,4,1]  transpose ([[2,3],[5,6],[1,1]])
```

```
((a:[a]) : [[a]])
((x:xs) : xss)

[h| (h:t) <- xss]
[t | (h:t) <- xss]

((x:xs) : (h:t))
((a:[a]) :([a]:[[a]])

(x:h) : transpose (xs:t)
(a:[a]) :             ([a]:[[a]])
```

https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-List.html

Transpose Example (2)

([[2,3],[5,6],[1,1]]

[2,3] : [[5,6],[1,1]]
(2 : [3]) : ([5,1] : [[6,1]])

(2 : [5,1]) : transpose ([3] : [[6,1]])
[2,5,1] : transpose ([[3,6,1]])

((a:[a]) : [[a]])
((x:xs) : xss)

[h| (h:t) <- xss]
[t | (h:t) <- xss]

((x:xs) : (h:t))
((a:[a]) :([a]:[[a]])

(x:h) : transpose (xs:t)
(a:[a]) : ([a]:[[a]])

https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-List.html

Inner Product

innXa, innXb : [[Integer]] -> Integer

innXa = foldr (+) 0 . map (foldr (*) 1) . transpose

innXb = foldr1 (+) . map (foldr1 (*)) . transpose

[[1, 2, 3], [10, 20, 30]]

[[1, 10], [2, 20], [3, 30]]

[1*10, 2*20, 3*30]

1*10 + 2*20 + 3*30

http://book.realworldhaskell.org/read/defining-types-streamlining-functions.html

1. Terse Syntax

Recognize blocks

- By indentation (prefer to use space than tab)
- By newline

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

2. Function Calls

a b c d e f

Function name : a
Function arguments : b, c, d, e, and f

a (b, c, d, e, f)

If parenthesis is used, use also comma : tuple representation

Function name : a
Function argument : one tuple (b, c, d, e, f)

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

3. Function Definitions

a b c = d e f

Function name : a
Function arguments : b and c

The body of function a is
a function call d with argument of e and f

a (b, c) = d e f

Formal parameters in parentheses : patterns contained

f (leaf x) = …
f (node, left, right) = …

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

4 .Currying

a b c d e f
function with 5 arguments

can be called with fewer arguments
a 1.0 2.0
will return a new function say g d e f

Partial application
is possible because any function
with multiple arguments can be curried

```
f :: a -> b -> c
f x = g

f x y = g y
```

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

5. statement-like expression

a function body : expression (return value)
No return statement in Haskell
But return library function exists

If-then-else structure : expression

Local variable definitions : expression

```
let x = 5 in
    x * x
```

Pattern matching constructs : expression

```
case tree of
    Leaf x -> ...
    Node left right -> ...
```

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

6. No Loop

map, filter, foldl, and foldr

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

7. Function application precedence

A b c d + e f g

a b c d
e f g

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

8. Data Types : algebraic, pattern matching

data Bool = True | False

'OR' : the role of addition
'AND' : the role of multiplication

Data constructors used as patterns to match

data Tree = Leaf Int | Node Tree Tree

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

9. No order

Order doesn't matter in the definition

```
len x y = sqrt (sq x + sq y)
  where
    sq a = a * a
```

can define local functions or variables
after the code that calls them

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

10. Order in Do

```
do
    a <- giveMeAnA
    b <- giveMeAB
    return (a + b)
```

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

Tree Example

```
data Tree = Leaf Int | Node Tree Tree

g (Leaf x) = x
g (Node left right) = g left + g right

f tree =
    case tree of
        Leaf x -> x
        Node left right -> f left + f right
```

https://www.fpcomplete.com/blog/2012/09/ten-things-you-should-know-about-haskell-syntax

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf