Think more effectively with Haskell

Evan Misshula

2017-02-14

Outline

These slides are based on

- Learn You Haskell for Great Good
- Haskell Tutorials from
    - Programming Languages by Peter Drake
- Videos from the New York and Boston Haskell Meetups
    - Special thanks to NYC's:
        - Gershon Bazerman
        - Evie Borthwick
        - Ryan Trinkle
        - Brian Hurt
        - Cat Chuang
    - Boston:
        - Edward Kmet

These slides are based on

- Learn You Haskell for Great Good
- Haskell Tutorials from
  - Programming Languages by Peter Drake
- Videos from the New York and Boston Haskell Meetups
  - Special thanks to NYC's:
    - Gershon Bazerman
    - Evie Borthwick
    - Ryan Trinkle
    - Brian Hurt
    - Cat Chuang
  - Boston:
    - Edward Kmet
    - I have watched his videos so many times I think we have met
- Bartoz Milewski's
  - Category Theory blog
  - Video Tutorials
- The brilliant and funny Eugenia Cheng

Help

- Internet Relay Chat
    - Freenode
    - #Haskell
    - Or our gmail group

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
  - In Haskell computations is *lazy*.
  - In Haskell a variables is *immutable*.
  - In Haskell a variable is never implicitly coerced into another type.

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
    - In Haskell computations is *lazy*.
    - In Haskell a variables is *immutable*.
    - In Haskell a variable is never implicitly coerced into another type.

    - immutability means you can build Single Page [Web] Apps (SPA)

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
  - In Haskell computations is *lazy*.
  - In Haskell a variables is *immutable*.
  - In Haskell a variable is never implicitly coerced into another type.
  - immutability means you can build Single Page [Web] Apps (SPA)
  - immutability means you can transpile into JavaScript and build native mobile apps without coding in Java, Objective-C or Swift

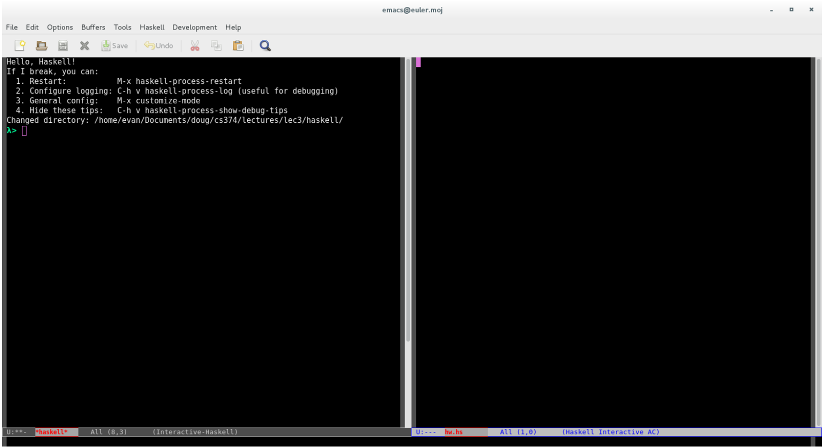
What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
  - In Haskell computations is *lazy*.
  - In Haskell a variables is *immutable*.
  - In Haskell a variable is never implicitly coerced into another type.
  - immutability means you can build Single Page [Web] Apps (SPA)
  - immutability means you can transpile into JavaScript and build native mobile apps without coding in Java, Objective-C or Swift
  - Immutability, functional and lazy gives you code you can run in parallel by default

What do you need?

- You need an editor and the Haskell compiler. We will install Haskell stack, Emacs and code completion.

let's open Emacs



ready, set, go

```
  2 + 15
  49 * 100
  1892 - 1472
  5 / 2
1=1

17
4900
420
2.5
```

Compound arithmetic computations

```
   (50 * 100) - 4999
   50 * 100 - 4999
   50 * (100 - 4999)
1==1

1
1
-244950
```

Watch out for negative numbers

- `5 * -3` doesn't work

Watch out for negative numbers

- `5 * -3` doesn't work
- `5 * (-3)` does

usual rules with Boolean variables

- && == Boolean *and*
- || == Boolean *or*
- not == negation

testing for equality

```
 True && False
 True && True
 False || True
 not False
 not (True && True)
1=1

False
True
True
True
False
```

Bad addition

```
5+"llama"
1==1

<interactive>:252:2:
    No instance for (Num [Char]) arising from a use of ‘+’
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 5 + "llama"
    In an equation for ‘it’: it = 5 + "llama"
```

Bad addition

```
5+"llama"
1==1

<interactive>:252:2:
    No instance for (Num [Char]) arising from a use of ‘+’
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 5 + "llama"
    In an equation for ‘it’: it = 5 + "llama"
```

result

Bad test

```
  5 == True
1==1

<interactive>:255:1:
    No instance for (Num Bool) arising from the literal ‘5’
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘(==)’, namely ‘5’
    In the expression: 5 == True
    In an equation for ‘it’: it = 5 == True
```

Bad test

```
  5 == True
1==1

<interactive>:255:1:
    No instance for (Num Bool) arising from the literal ‘5’
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘(==)’, namely ‘5’
    In the expression: 5 == True
    In an equation for ‘it’: it = 5 == True
```

result

```
<interactive>:340:1:
    No instance for (Num Bool) arising from the literal ‘5’
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘(==)’, namely ‘5’
    In the expression: 5 == True
    In an equation for ‘it’: it = 5 == True
```

GHCI can tell that the types don't match

- '+' expects left and right to be number
- '==' expects two things that are of the same type

infix and *prefix* functions

infix and *prefix* functions

- '+' is *infix* because it goes between its arguments

infix and *prefix* functions

- '+' is *infix* because it goes between its arguments
- 'succ' is a *prefix* function because it goes before its argument

prefix demo

```
succ 8
min 9 10
min 3.4 3.2
max 100 101
1=1
```

```
9
9
3.2
101
```

Haskell relies on precedence

- Many Lisp/Scheme/Clojure programmers put in more parenthesis than is idiomatic in Haskell
- These two statements are the same

```
succ 9 + max 5 4 + 1
(succ 9) + (max 5 4) + 1
1==1
```

```
16
16
```

imperative steps

imperative steps

1. open Emacs
2. create the following director in Documents
   `~/Documents/cs374/haskell/`
3. create the file baby.hs
4. start the GHCI

imperative steps

1. open Emacs
2. create the following director in Documents `~/Documents/cs374/haskell/`
3. create the file baby.hs
4. start the GHCI

1. type `doubleMe x = 2*x`
2. load the file into ghci

take integers or floats

take integers or floats

1. type `doubleUS x y = 2*x + 2*y` in baby.hs

take integers or floats

1. type `doubleUS x y = 2*x + 2*y` in baby.hs
1. type `doubleUS x y = x + x + y + y` in baby.hs

take integers or floats

1. type `doubleUS x y = 2*x + 2*y` in baby.hs
1. type `doubleUS x y = x + x + y + y` in baby.hs
1. type `doubleUS x y = doubleMe x + doubleMe y` in baby.hs

piece-wise functions

piece-wise functions

1. type the following

```
let doubleSmallNumber x = if (x > 100) then x else 2*x
doubleSmallNumber 54
doubleSmallNumber 103
1==1
```

piece-wise functions on one line

piece-wise functions on one line

1. type the following

```
doubleSmallNumber' x = (if x > 100 then x else 2*x) + 1
```

character functions

1. type the following

character functions

1. type the following

1. can't capitalize the name

character functions

1. type the following
1. can't capitalize the name

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

naming lists

1. type the following `let lostNumbers = [4,8,15,16,23,42]`

naming lists

1. type the following `let lostNumbers = [4,8,15,16,23,42]`
1. '++' is the concatenate operator

naming lists

1. type the following `let lostNumbers = [4,8,15,16,23,42]`

1. '++' is the concatenate operator

```
[1,2,3,4] ++ [9,10,11,12]
"hello" ++ " " ++ "world"
['w','o'] ++ ['o','t']
1=1
```

```
[1,2,3,4,9,10,11,12]
hello world
woot
```

naming lists

1. ':' adds an element to the front of a list O(1) time (cons)

naming lists

1. ':' adds an element to the front of a list O(1) time (cons)

1. '++' works in O(n)

naming lists

1. ':' adds an element to the front of a list O(1) time (cons)

1. '++' works in O(n)

1. [1,2,3] is really 1:2:3:[]

naming lists

1. ':' adds an element to the front of a list O(1) time (cons)
1. '++' works in O(n)
1. [1,2,3] is really 1:2:3:[]
1. '!!' takes element out of a list

naming lists

1. ':' adds an element to the front of a list O(1) time (cons)
1. '++' works in O(n)
1. [1,2,3] is really 1:2:3:[]
1. '!!' takes element out of a list

```
"Steve Buscemi" !! 6
[9.4,33.2,96.2,11.2,23.25] !! 1
```

```
1==1
```

```
'B'
33.2
```

Four O(1) list functions

1. 'head' takes a list and returns only its first element

Four O(1) list functions

1. 'head' takes a list and returns only its first element
1. 'tail' takes a list and returns everything except its first element

Four O(1) list functions

1. 'head' takes a list and returns only its first element
1. 'tail' takes a list and returns everything except its first element
1. 'last' takes a list and returns only its last element

Four O(1) list functions

1. 'head' takes a list and returns only its first element
1. 'tail' takes a list and returns everything except its first element
1. 'last' takes a list and returns only its last element
1. 'init' takes a list and returns everything except its last element

Four O(1) list functions

1. 'head' takes a list and returns only its first element
1. 'tail' takes a list and returns everything except its first element
1. 'last' takes a list and returns only its last element
1. 'init' takes a list and returns everything except its last element

```
head "Steve Buscemi"
head [9.4,33.2,96.2,11.2,23.25]
tail "Steve Buscemi"
tail [9.4,33.2,96.2,11.2,23.25]
last "Steve Buscemi"
last [9.4,33.2,96.2,11.2,23.25]
init "Steve Buscemi"
init [9.4,33.2,96.2,11.2,23.25]
1==1
```

```
'S'
9.4
```

Matrices can be represented as nested lists

- We use nested lists
- We can also apply the list functions multiple times to access parts of the list

```
let firstMat = [[9,8,7],[6,5,4],[3,2,1]]
firstMat !! 1
(firstMat !! 1) !! 2
1==1
```

```
Prelude|
<interactive>:298:1: parse error on input ‘firstMat’
<interactive>:298:2: Not in scope: ‘firstMat’
```

Length examples

- 'length' returns the length of a list

```
length [3,9,3]
length [1,2]
length [6,6,6]
  1==1
```

```
3
2
3
```

Null examples

- 'null' let's us know if a list is empty

```
null [1,2,3]
null []
  1==1
```

```
False
True
```

Functions to reduce a list

- 'drop' returns the list without the first n elements

```
drop 1 [3,9,3]
drop 5 [1,2]
drop 0 [6,6,6]
  1==1
```

```
[9,3]
[]
[6,6,6]
```

Take examples

- 'take' returns the first n elements of a list

```
take 1 [3,9,3]
take 5 [1,2]
take 0 [6,6,6]
  1==1
```

```
[3]
[1,2]
[]
```

There is a built in function to reverse a list

- 'reverse' doesn't reduce
    - returns a list in the reverse order
    - can't be bound to the same variable

```
reverse [5,4,3,2,1]
1==1
```

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

There are a built in function to return extremes of a list

- 'minimum' returns the minimum of a list
- 'maximum' returns the maximum of a list

```
minimum [5,4,3,2,1]
maximum [5,4,3,2,1]
1==1
```

```
1
5
```

There are a built in functions to sum and multiply elements

- 'sum' returns the sum of the elements of a list
- 'product' returns the product of the elements of a list

```
sum [5,4,3,2,1]
product [5,4,3,2,1]
1==1
```

```
15
120
```

There is a built in function to check membership

- 'elem' returns the sum of the elements of a list
- the backtick key '‘' allows you to use a *prefix* function as *infix*

```
4 `elem` [3,4,5,6]
10 `elem` [3,4,5,6]
1==1
```

```
True
False
```

Enumeration and range

- If we can enumerate all the elements of a set we can use ranges:
  - numbers
  - letters
  - capital letters

```
[1..20]
['a'..'z']
['X'..'Z']
1==1
```

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
abcdefghijklmnopqrstuvwxyz
XYZ
```

ranges can have steps

- We can get even the even numbers
- We can get the multiples of three

```
[2,4..20]
[3,6..20]
1==1
```

```
[2,4,6,8,10,12,14,16,18,20]
[3,6,9,12,15,18]
```

Here are functions that produce infinite data structures

- First 24 multiples of 13

Here are functions that produce infinite data structures

- First 24 multiples of 13
- take 24 [13,26..]

Here are functions that produce infinite data structures

- First 24 multiples of 13
- take 24 [13,26..]
- Haskell is lazy so it won't compute until you ask it for something

Here are functions that produce infinite data structures

- First 24 multiples of 13
- take 24 [13,26..]
- Haskell is lazy so it won't compute until you ask it for something
- 'cycle' takes a finite list and makes it infinite

```
take 24 [13,26..]
take 10 (cycle [1,2,3])
take 12 (cycle "LOL ")
1==1
```

```
[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,2
[1,2,3,1,2,3,1,2,3,1]
LOL LOL LOL
```

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- take 10 (repeat 5)

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- take 10 (repeat 5)
- 'replicate' produces a list of a given size of all a given number

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- take 10 (repeat 5)
- 'replicate' produces a list of a given size of all a given number
- 'cycle' takes a finite list and makes it infinite

```
take 10 (repeat 5)
replicate 3 10
1==1
```

```
[5,5,5,5,5,5,5,5,5,5]
[10,10,10]
```

Just like Python, Haskell has its own twist on list comprehensions

Definition (In set notation)

- $S = \{2 \cdot x | x \in \mathbb{N}, x \leq 10\}$
  - $2 \cdot x$ the output function
  - 'x' is the variable
  - N is the input set
  - x $\leq$ 10 is the predicate

```
[x*2 |x<-[1..100]  ,x <=10, x>=6]
[ x | x <- [50..100], mod x 7==3]
1==1

[12,14,16,18,20]
[52,59,66,73,80,87,94]
```

We can combine functions and list comprehensions

- We can even get product lists

```
let boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <-
boomBangs [7..13]
[ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[ x*y | x <- [2,5,10], y <- [8,10,11]]
[ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
1==1

Prelude|
<interactive>:359:1: parse error on input ‘boomBangs’
[10,11,12,14,16,17,18,20]
[16,20,22,40,50,55,80,100,110]
[55,80,100,110]
```

We can string functions and list comprehensions

- We can even get combine lists of adjectives and nouns for some laughs

```
let nouns = ["hobo","frog","pope"]
let adjectives = ["lazy","grouchy","scheming"]
[adjective ++ " " ++ noun | adjective <- adjectives, noun <- no
```

We can even use projections

- If we keep track of how many elements we throw away we can redefine the 'length' function
- '_' means we don't care what the element is

```
let length' xs = sum [1 | _ <- xs]
i==i
```

We can even use membership to keep only a certain items

- Here we keep only capital letters
- Let's write a function that keeps only lower case

```
let removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
removeNonUppercase "Hahaha! Ahahaha!"
removeNonUppercase "IdontLIKEFROGS"
let removeNonLowercase st = [ c | c <- st, c `elem` ['a'..'z']]
removeNonLowercase "i love AND HATE haskell"
i==i
```

```
Prelude|
<interactive>:373:1: parse error on input ‘removeNonUppercase’
<interactive>:373:1: Not in scope: ‘removeNonUppercase’
Prelude|
<interactive>:376:1: parse error on input ‘removeNonLowercase’
```

Nested lists can be used to remove odd numbers in a list of lists

- We could do this across several lines

```
let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6
    [[ x | x <- xs, even x ] | xs <- xxs]
i==i
```

Ordered pairs in Haskell

- Tuples in Haskell have stronger type checking

```
[[1,2],[8,11,5],[4,5]] --doesn't throw an error
[(1, 2), (8, 11, 5), (4, 5)]
i==i

[[1,2],[8,11,5],[4,5]]
<interactive>:383:10:
    Couldn't match expected type ‘(t0, t1)’
                with actual type ‘(t2, t3, t4)’
    In the expression: (8, 11, 5)
    In the expression: [(1, 2), (8, 11, 5), (4, 5)]
    In an equation for ‘it’: it = [(1, 2), (8, 11, 5), (4, 5)]
```

A 2-tuple is also called an ordered pair

- There are two special functions which return the components

```
fst ("One", 1)
snd ("Two", 2)
i==i
```

```
One
2
```

Like Python, Haskell has a zip operation

- We can zip two lists into a list of tuples
- The two lists do not need to be the same type
- One list can be shorter than the other
- We can zip finite lists with infinite ones (lazy)

```
zip [1,2,3,4,5] [5,5,5,5,5]
zip [1 .. 5] ["one", "two", "three", "four", "five"]
zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
zip [1..] ["apple", "orange", "cherry", "mango"]
i==i
```

```
[(1,5),(2,5),(3,5),(4,5),(5,5)]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
[(5,"im"),(3,"a"),(2,"turtle")]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg
- generate all tuples of sides less than 10

```
length ([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10]])
i==i

1000
```

Let's add constraints

Let's add constraints

- make b $<$ c

Let's add constraints

- make b < c
- make only right triangles
- Here is the first

```
length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z])
i==i
```

```
450
```

Let's add constraints

Let's add constraints

- make $a^2 + b^2 = c^2$

Let's add constraints

- make $a^2 + b^2 = c^2$
- the perimeter equal 24
- $a + b + c = 24$

```
length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z,(x^2 + y
i==i
```

```
4
```

Let's add constraints

- the perimeter equal 24
- $a + b + c = 24$

```
:set +m

length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],
y<z,
x+y+z==24,
(x^2 + y^2 == z^2)])
[(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z,
  x+y+z==24,
  (x^2 + y^2 == z^2)]
i==i


Prelude> Prelude| Prelude| Prelude| 2
Prelude| Prelude| [(6,8,10),(8,6,10)]
```