BAHUG 101 - Lecture 1

9th September 2015

Outline of the Course

- Meet every two weeks, same time, same place.
- Using Piazza (piazza.com/bay_area_haskell_users_group/fall2015/bahug101, code is “alonzo”) for class discussion and announcements.
- Go up to week 8 in the CS194 course (Monads)
- “Final project” is to contribute to an open source Haskell project.

Thanks to the UPenn guys

The material is from the CIS194 course taught at UPenn by Richard Eisenberg, Brent Yorgey, and Noam Zolberstein. The vast majority of this course (including the outline and code in this lecture) is their cirriculum. Thanks!

The CIS194 course material can be found at `http://www.seas.upenn.edu/~cis194/`. We will be following the Spring 2015 course.

Outline of Today's Lecture

- What makes Haskell, Haskell
- Some basic syntax
- Lists
- Functions

Properties of Haskell

Haskell is a Functional language

What do we mean by this?

Haskell is a Functional language

What do we mean by this?

- Functions are *first class*, which means that functions can be used in a way variables are used in other languages.
  - Functions can be passed to functions and returned from other functions.

Haskell is a Functional language

What do we mean by this?

- Functions are *first class*, which means that functions can be used in a way variables are used in other languages.
  - Functions can be passed to functions and returned from other functions.

- We think of programs by the evaluation of expressions to reach a solution, not a sequence of explicit steps to get to the expressions.

Haskell is a Pure language

When we think of a function in math

$$y = f(x)$$

we can say that whenever we give the input x, we will always get out y. This is called *referential transparency*. This implies the following properties.

- Functions cannot cause mutations
- There are no "side effects".
- Calling a function with the same input leads to the same output.

Equational Reasoning

Take the following Python program.

```
a = ...
b = a

c = someFunc(a)
d = someFunc(b)

if c == d:
    print(”c and d are the same!”)
else:
    print(”c and d are _not_ the same!”)
```

Which `if` case is printed?

Equational Reasoning

Another python example

```
a = {'a': 1, 'b': 2}
b = a

c = someFunc(a)
d = someFunc(b)

if a == {'a': 1, 'b': 2}
    print("a is {'a': 1, 'b': 2}")
else:
    print("a is something else.")
```

which `if` case is printed?

Equational Reasoning

Same concept in Haskell

```
someFunc :: (Eq a, Eq b) => a -> b

a = ... -- some value
b = a

c = someFunc(a)
d = someFunc(b)

inputs_equal = a == b -- What is this value?
output_equal = c == d -- What is this value?
```

Haskell is a Lazy language

Haskell does not evaluate expressions until their results are needed. This allows for some interesting things.

- *Infinite* data structures.
- Compositional programming style

As a downside, it can be challenging to reason about the space and time a functional program/algorithm will take.[1]

[1]See Chris Okasaki's "Purely Functional Data Structures" for more details

Let's write some Haskell!

```haskell
-- x is a variable (or expression) of type Int
x :: Int

-- the variable (or expression) x has the value 3
x = 3
```

Let's write some Haskell!

```haskell
-- x is a variable (or expression) of type Int
x :: Int

-- the variable (or expression) x has the value 3
x = 3
```

What happens if I add this to the file?

```haskell
x = 4
```

Let's write some Haskell!

```haskell
-- x is a variable (or expression) of type Int
x :: Int

-- the variable (or expression) x has the value 3
x = 3
```

What happens if I add this to the file?

```haskell
x = 4
```

```
Multiple declarations of 'x'
Declared at: file.hs:2:1
             file.hs:3:1
```

Let's write some Haskell!

Can we do this?

```haskell
y :: Int
y = y + 1
```

[2]for example, by printing it

Let's write some Haskell!

Can we do this?

```haskell
y :: Int
y = y + 1
```

Since Haskell is *lazy*, we can define y. But if we try to evaluate it[2], we will get an exception.

[2] for example, by printing it

Basic Types

Integers

We can have standard integers.

```haskell
i :: Int
i = -78
```

There is no guarantee that an integer has a certain size, only that it has to accommodate up to 2^{29} (536,870,912). For example, mine is 2^{63} (a very large number).[3]

[3] Can get by running `maxBound :: Int` in GHCi

Infinitely large integers!

These integers only bound by the amount of memory you have.

```haskell
n :: Integer
n = 123456789012345678901234567890

reallyBig :: Integer
reallyBig = 2^(2^(2^(2^2))) -- 2^65536

numDigits :: Int
numDigits = length (show reallyBig) -- 19,729 digits
```

Floating Point

Standard single precision floating point

```
f1, f2 :: Float
f1 = 4.556345
f2 = -45.34e-6
```

and double precision floating point

```
d1, d2 :: Double
d1 = 4.556345
d2 = -45.34e-6
```

Booleans

Not much to say, there are only two values.

```
b1, b2 :: Bool
b1 = True
b2 = False
```

Chars and String

`Char` is the type for Unicode characters.

```
c1, c2, c3 :: Char
c1 = 'x' -- Note the single quotes
c2 = 'λ'
c3 = 'א'
```

Strings are list of characters.

```
s1 :: String
s2 :: [Char]

s1 = "Hello World!" -- Note the double quote
s2 = s1
```

How to use GHCi and FP Complete

Arithmetic

- `(+)`, `(-)`, `(*)` for any number
- `(/)` for floating point, `div` for integer numbers.
- Powers
  - `(**)` :: Floating a => a -> a -> a)
  - `(^^)` :: (Fractional a, Integral b) => a -> b -> a
  - `(^)` :: (Integral b, Num a) => a -> b -> a
- Negative nubers must be in braces.
- `fromIntegral` to convert integers to floats.

Boolean Logic

The boolean operators are (&&) "and" and (||) "or". Haskell includes the **not** unitary operator.

```
ex11 = True && False
ex12 = not (False || True)
ex12b = not False || True
```

We can determine equality using (==) and (/=), and compare using the standard (<), (>), (<=), and (>=).

```
ex13 = 'a' == 'a'
```

Boolean Logic

The boolean operators are (&&) “and” and (||) “or”. Haskell includes the **not** unitary operator.

```
ex11 = True && False
ex12 = not (False || True)
ex12b = not False || True
```

We can determine equality using (==) and (≠), and compare using the standard (<), (>), (≤), and (≥).

```
ex13 = 'a' == 'a'

ex13b = 'a' == "a"
```

Boolean Logic

The boolean operators are (`&&`) "and" and (`||`) "or". Haskell includes the **`not`** unitary operator.

```haskell
ex11 = True && False
ex12 = not (False || True)
ex12b = not False || True
```

We can determine equality using (`==`) and (`/=`), and compare using the standard (`<`), (`>`), (`<=`), and (`>=`).

```haskell
ex13 = 'a' == 'a'

ex13b = 'a' == "a"

ex14 = 16 /= 3
ex15 = (5 > 3) && ('p' <= 'q')
ex16 = "Haskell" > "C++" -- What kind of ordering is this?
```

If expressions

Haskell has the **`if`** control structure with a twist. It *requires both branches to return the same type*.

```
ex_a = if (9 > 7) then "Math works!" else "We broke the universe"
```

For example, the following will not compile!

```
ex_b = if (9 > 7) then "I want to end early" -- Won't compile
```

Functions

Defining basic functions

Here is a basic function.

```haskell
-- Compute the sum of the integers from 1 to n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0                            -- Base case
sumtorial n = n + sumtorial (n - 1) -- Recurse case
```

Guards

We can use guards when our function needs to discriminate on a value.

```
hailstone :: Integer -> Integer
hailstone n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Also known as the Collatz conjecture, it says that for any positive input we start with, we will always reach 1.

Where Clause

Where clauses allow for sub-expressions.

```
hailstone' :: Integer -> Integer
hailstone' n
  | isEven    = n `div` 2
  | otherwise = 3*n + 1
  where
    isEven = n `mod` 2 == 0
```

Note that `isEven` is local to `hailstone'`; it cannot be accessed outside of `hailstone'`.

Let Clause

Let clauses provide a similar facility.

```
hailstone'' :: Integer -> Integer
hailstone'' n = let isEven = n `mod` 2 == 0 in
  if isEven then n `div` 2
            else 3*n + 1
```

Note that `isEven` is local to `hailstone''`; it cannot be accessed outside of `hailstone''`.

Guards and pattern matching together

Guards and pattern matching can be combined together.

```
bizarro :: Integer → Integer
bizarro 0 = 1
bizarro 1
  | (3 + 4) > 4 = 0
  | otherwise   = 2
bizarro n
  | n < 0 = fromIntegral (maxBound :: Int)
  | n > 1 = fromIntegral (minBound :: Int)
```

Pairs

To create pairs of values use the `(,)` function.

```
p :: (Int, Char)
p = (3, 'x')

p' :: (Int, Char)
p' = (,) 3 'x'
```

Elements of a pair can be extracted with pattern matching.

```
sumPair :: (Int, Int) -> Int
sumPair (x, y) = x + y
```

Functions of multiple arguments

Functions can take in multiple arguments by adding more arrows.

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
```

Functions of multiple arguments

Functions can take in multiple arguments by adding more arrows.

```haskell
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
```

As a preview, functions can be partially applied.

```haskell
f3 :: Int -> Int -> Int -- Note the one less `Int`
f3 = f 3
```

Lists

Lists allow for any number of values *of the same type* to be grouped together.

```
nums, range, range2, range3 :: [Integer]
nums = [1, 2, 3, 19]
```

Lists

Lists allow for any number of values *of the same type* to be grouped together.

```
nums, range, range2, range3 :: [Integer]
nums = [1, 2, 3, 19]

range = [1..100]
```

Lists

Lists allow for any number of values *of the same type* to be grouped together.

```haskell
nums, range, range2, range3 :: [Integer]
nums = [1, 2, 3, 19]

range = [1..100]

range2 = [2,4..100]
```

Lists

Lists allow for any number of values *of the same type* to be grouped together.

```
nums, range, range2, range3 :: [Integer]
nums = [1, 2, 3, 19]

range = [1..100]

range2 = [2,4..100]

range3 = [8,6..0]
```

Constructing Lists

To create an empty list, just use empty brackets.

```
emptyList = []
```

To add to the beginning of a list, use the `(:)` "cons" operator.

```
ex18 = 1 : []
ex19 = 3 : (1 : [])
ex20 = 1 : 2 : 3 : 4 : []

ex21 = [2, 3, 4] == 2 : 3 : 4 : []
```

Constructing list by appending

The (++) operator combines two lists, and can be used to append an element to a list.

```
exAppend1 = [1, 2, 3] ++ [4, 5, 6]
exAppend2 = [1, 2, 3] ++ [4]
exAppend3 = [1] ++ [2, 3, 4] == 1 : [2, 3, 4]
```

Note that (++) is particularly inefficient: it takes time proportional to the length of the first list.

List constructing example function

Also known as the Collatz conjecture, it says that for any positive input we start with, we will always reach 1.

```
hailstoneSeq 1 = [1]
hailstoneSeq n = n : hailstoneSeq (hailstone n)
```

Functions on Lists

Using the same idea as pairs, we can write functions on lists by pattern matching.

```
-- Compute the length of a list of Integers
intListLength :: [Integer] -> Integer
intListLength [] = 0
intListLength (x : xs) = 1 + intListLength xs
```

Functions on Lists

Using the same idea as pairs, we can write functions on lists by pattern matching.

```
-- Compute the length of a list of Integers
intListLength :: [Integer] -> Integer
intListLength [] = 0
intListLength (x : xs) = 1 + intListLength xs
```

We can replace the `x` with `_` since we do not use it.

```
-- Compute the length of a list of Integers
intListLength' :: [Integer] -> Integer
intListLength' [] = 0
intListLength' (_ : xs) = 1 + intListLength xs
```

Functions on Lists Extended

We can nest pattern matches

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo [] = []
sumEveryTwo (x : []) = [x]
sumEveryTwo (x : (y : zs)) = (x + y) : sumEveryTwo zs
```

Functions on Lists Extended

We can nest pattern matches

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo [] = []
sumEveryTwo (x : []) = [x]
sumEveryTwo (x : (y : zs)) = (x + y) : sumEveryTwo zs
```

We can also do different types of pattern matches

```
sumPairs :: [(Integer, Integer)] -> [Integer]
sumPairs [] = []
sumPairs ((x1, x2) : xs) = (x1 + x2) : sumPairs xs
```

Combining Functions

Functions can be combined to make more complex functions.

```
-- The number of hailstone steps needed to reach 1
-- from a starting number.
hailstoneLen :: Integer -> Integer
hailstoneLen n = intListLength (hailstoneSeq n) - 1
```

Error Messages

Haskell's error messages are scary, but really Haskell just likes being verbose.

```
<interactive>:31:1:
    Couldn't match expected type ‘[Char]’ with actual type ‘Char’
    In the first argument of ‘(++)’, namely ‘'x'’
    In the expression: 'x' ++ "foo"
    In an equation for ‘it’: it = 'x' ++ "foo"
```