The magic of Haskell

Evan Misshula

2017-02-14

Outline

Haskell is statically typed

- Haskell uses Damas–Hindley–Milner type system
  - We can write a number and Haskell will infer a type
  - We can see that type by using the ':t' command in the repl:

```
 :t 'a'
 :t True
 :t "HELLO!"
 :t (True, 'a')
 :t 4 == 5
1==1
```

```
'a' :: Char
True :: Bool
"HELLO!" :: [Char]
(True, 'a') :: (Bool, Char)
4 == 5 :: Bool
```

How to read a type signature

- The symbol "::' is read as "has type of".
- Both data and functions have types
- It is considered good practice to give functions of any length types

```
:set +m
let removeNonUppercase :: [Char] -> [Char]; removeNonUppercase
    [ c | c <- st, c ‘elem‘ [’A’..’Z’]]
1==1
```

What if we take more than one parameter

- The parameters are separated by an '->' symbol
- No distinction between parameter and return type
- If you are not sure use ':t'

```
:set +m
let addThree :: Int -> Int -> Int -> Int; addThree x y z = x +
  1==1
```

Review of basic types

- There are two types of Integers, Int and Integer
- Int is bounded
  - The lower bound is guaranteed to be -2^{29}
  - The upper bound is guaranteed to be $2^{29} - 1$
- This is for 32 bit machines are usually bigger or smaller
- On my machine, the bounds are $-2^{63}, 2^{63} - 1$
- Integer has only the bound of your machine but it is less efficient

What does single precision mean?

Single-precision floating-point format is a computer number format that occupies 4 bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point. In IEEE 754-2008 the 32-bit base-2 format is officially referred to as binary32. (Wikipedia)

Bool

- is a boolean type
- it can only have two values
  - True
  - False

Char

- represents a character
- denoted by single quotes
- a list of char's is a string

Tuples are also types

- Most basic tuple is an empty ()

What about functions that work on different types

What about functions that work on different types

- These are polymorphic functions
  - examples

```
head [1,2,3]
head "Evan"
fst ("Evan","Misshula")
snd (1,3)
1==1
```

```
1
'E'
Evan
3
```

We can still examine the type

- :t examines the type
- by convention single characters are type variables

```
:t fst
1==1
```

```
fst :: (a, b) -> a
```

Typeclass defines a behavior

- Let's look at the the type of '=='

```
:t (==)
1==1
```

```
(==) :: Eq a => a -> a -> Bool
```

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
  - The declaration we can read says:

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
    - The declaration we can read says:
- The equality function takes two variables of the same type and returns a Bool

(==) :: Eq a => a -> a -> Bool

- Typeclass constraint
  - The declaration we can read says:
- The equality function takes two variables of the same type and returns a Bool
- The new part 'Eq a =>' says:
- The type must be part of Eq typeclass
  - This is called the class constraint

The Eq typeclass provides an interface for testing for equality

- Eq is used for types that support equality testing
  - Its members implement both:
    - '=='
    - '/='

```
5==5
5/=5
'a' == 'a'
"Ho Ha" == "Ho Ha"
3.4 == 3.4
  1==1
```

```
True
False
True
True
True
```

Ord is for types that have an ordering

- We can see the type of '>' comparison
- We can see some functions which rely on being in the ord typeclass

```
:t (>)
"Abc"< "Zev"
compare "Abc" "Zev"
5 >= 2
compare 5 3
1==1
```

```
(>) :: Ord a => a -> a -> Bool
True
LT
True
GT
```

Everything except function has been part of show

- It works like Java or Ruby's toString methods
- Mostly we use it to examine a value

```
show 3
show 5.334
show True
1==1
```

```
3
5.334
True
```

Read is the inverse of show

- It works reads a string and returns a type which supports the interface Read

Read is the inverse of show

- It works reads a string and returns a type which supports the interface Read
- You can use it to create Javascript like craziness

```
read "True" || False
read "8.2" + 3.8
read "5" - 2
read "[1,2,3,4]" ++ [3]
1==1

True
12.0
3
[1,2,3,4,3]
```

Let's look at a type error

```
read 4
1==1

<interactive>:654:6:
    No instance for (Num String) arising from the literal ‘4’
    Possible fix: add an instance declaration for (Num String)
    In the first argument of ‘read’, namely ‘4’
    In the expression: read 4
    In an equation for ‘it’: it = read 4
```

- GHCI is saying it does not know what type to return
  - Do you want an Float or an Integer?

We can specify a type

- We just add '::<Type>' and read will work

```
read "5" :: Int
read "5" :: Float
(read "5" :: Int) * 4
read "[1,2,3,4]" :: [Int]
read "(3,'a')" :: (Int, Char)
1==1
```

```
5
5.0
20
[1,2,3,4]
(3,'a')
```

Sequentially ordered types

- Being *sequentialy ordered* means that they can be counted in order
- This property is also called being *enumerable*
- We can use them in list ranges
    - they each have a predecessor which you can get with 'pred'
    - they each have a successor which you can get with 'succ'

```
['a'..'e']
[LT .. GT]
[3..7]
succ 'B'
1==1
```

```
abcde
[LT,EQ,GT]
[3,4,5,6,7]
'C'
```

Bounded type class has concrete types

- with maximum and minimum elements
  - minBound and maxBound are functions with polymorphic type
    - (Bounded a) => a

```
minBound :: Int
maxBound :: Char
maxBound :: Bool
minBound :: Bool
i==i
```

```
-9223372036854775808
'\1114111'
True
False
```

Numeric types can be operated on mathematically

- Let's look at this type

```
:t (*)
(5 :: Int) * (6 :: Integer)
(5 :: Int) * 6
i==i
```

```
(*) :: Num a => a -> a -> a
<interactive>:677:15:
    Couldn't match expected type 'Int' with actual type 'Intege
    In the second argument of '(*)', namely '(6 :: Integer)'
    In the expression: (5 :: Int) * (6 :: Integer)
    In an equation for 'it': it = (5 :: Int) * (6 :: Integer)
30
```

Integral and Floating types

- The Integral typeclass only includes Integer and Int
- The Floating typeclass only includes floats and double

```
:t fromIntegral
fromIntegral (length [1,2,3,4]) + 3.2
i==i
```

```
fromIntegral :: (Integral a, Num b) => a -> b
7.2
```