Haskell Study Notes

Charlotte

April 30, 2021

Contents

1 Types

1.1 Basic types

Haskell has many primitive types such as strings, characters, integers and floating point numbers.

```
1 "hello world"  -- string
2 1234            -- integer
3 3.14            -- float
```

1.2 Typeclasses

Typeclasses are a way of sharing specific functionality between types. We can either implement our own instances of typeclasses, or let haskell *derive* them automatically.

They are kind of like Java interfaces.

Num is the generic base typeclass that all numbers derive from.

1. Integral numbers (*Integral* typeclass)

   - *Int*: fixed precision integer with a min and maximum size

   - *Integer*: supports **very** large integers

2. Floating point numbers (*Fractional* typeclass)

   - *Float*: single precision floating point number

   - *Double*: double precision floating point number

1.3 More on typeclasses

Anything that derives from

- *Show* can be printed

- *Read* can be read as a value

- *Eq* can be compared for equality with $==$ and $/=$

- *Ord* can be compared and ordered with ¡ and ¿

```
1  {-# LANGUAGE DuplicateRecordFields #-}
2
3  data Worker = Worker { name :: String , job :: String }
       deriving (Show)
4  data Student = Student { name :: String , school :: String }
       deriving (Show)
5
6  class Person a where
7      getName :: a -> String
8      getOccupation :: a -> String
9
10 instance Person Worker where
11     getName x = name (x :: Worker)
12     getOccupation x = "Working on " ++ job x
13
14 instance Person Student where
15     getName x = name (x :: Student)
16     getOccupation x = "Studying at " ++ school x
```

Figure 1: Demonstration of a custom typeclass for Person

2 Lists and Tuples

2.1 Lists

Haskell lists are represented as linked lists. A node in a linked list can either be *Nil* or a pointer to the *next node.* Lists have a *head* and a *tail.* Because of Haskell being lazily evaluated, lists can be infinite. *take* takes n items from a list, and *drop* drops n items from a list.

list = [1, 2, 3, 4, 5]
In the above list, the *head* is $\underline{1}$, and the *tail* is $[\underline{2, 3, 4, 5}]$
    The *splitAt* function splits a list into two parts at the element specified. Lists can be indexed using the *!!* operator.

2.1.1 List Comprehensions

List comprehensions are <u>similar</u> to how they work in Python. They must have at least one list that is the generator, which provides the input.
    [*operation* | $x \leftarrow$ *list*]
    An example of a list comprehension:

```
[x ^ 2 | x <- [1..10]]
```

List comprehensions can have predicates (conditions)

```
[x ^ 2 | x <- [1..10], x `mod` 2 == 0]
```

... and they can pull from multiple generators/inputs too

```
[(x, y) | x <- [1,2,3], y <- ['a', 'b']]
```

List comprehensions can also be <u>bound</u> to *variables.*

```
1 let square' = [x ^ 2 | x <- [1..10]]
        1
```

2.2 Tuples

Haskell has tuples, triples, and *n-tuples.* Tuples have a *fst* and *snd* function which respectively gets either the first <u>or</u> second item.
    swap (defined in **Data.Tuple**) swaps the items in a *tuple*

```
1 ("char", 20)
2
3 -- a triple
4 ("char", 20, "turtles")
```

---

[1]When we say variables, the mathematical meaning of variables is meant. This means we can bind values to identifiers, but we can **never** change the values of those bindings.

3 Conditionals

3.1 If expressions

Haskell doesnt have if statements, however it has if expressions instead.

```
1  -- stolen from the haskell book
2  let x = 0
3  if (x + 1 == 1) then "AWESOME" else "wut"
```

3.2 Case expressions

Case expressions are similar to switch-case from languages like Java, and C++. Case expressions begin with `case x of` and their body contains all the different cases in the format *value → return-value*.

```
1  pal xs =
2    case y of
3      True  -> "yes"
4      False -> "no"
5    where y = xs == reverse xs
```

It can also be used to *pattern match* against data types

```
1  data Animal = Cat | Dog
2  speak a =
3    case a of
4      Cat -> "meow"
5      Dog -> "bork"
```

3.3 Guards

There are also guards which can provide a nicer way of pattern matching instead of writing if-else expressions or case blocks. Guard blocks are written as a series of cases, along with a fallback case called `otherwise`

Cases are written in the format: "| *condition = value.*"

```
1 abs n
2   | n < 0      = -n
3   | otherwise = n
```

3.4 Pattern matching

Pattern matching is very common in Haskell. It allows great flexibility since you can pattern match against anything to extract values etc.

_ or x means a catch-all/fallback pattern to match on if all else fails

```
1 isItOne :: Int -> Bool
2 isItOne 1 = True
3 isItOne _ = False
```

3.4.1 Matching against data constructors

Pattern matching against data constructors is possible. The example defines two functions called `getName` and `getAccNumber` that pattern match on the *User* data constructor to extract either the name or account number.

```
1 data User = User String Int
2
3 getName :: User -> String
4 getName (User name _) = name
5
6 getAccNumber :: User -> Int
7 getAccNumber (User _ acc) = acc
```

2 3

---

[2] If we don't use a field, we can use an _ just like pattern matching with functions to signify the field is ignored

[3] Pattern matching against data constructors can get real old if you have a lot of parameters. So an alternative to this that will be covered later is record types.

4 Functions

Functions are defined like add x y = x + y. All functions are pure, unless stated otherwise which means they cannot modify state, or perform side effects like input output, writing to a database, etc

4.1 Currying

By default, all functions are curried in Haskell. Given the following type signature

```
1 -- add takes two ints and returns an int
2 add :: Int -> Int -> Int
```

we would say that the function **add** takes two integers, and returns an integer. However, because of how currying works it actually means.

- **add** takes a *single* integer parameter

- and returns a function that takes another integer parameter, and eventually will return an integer itself.

Currying is useful because it means we can create new functions by *partially applying* functions to parameters.

addOne 5 evaluates to *((add 1) 5)* which is then further reduced to the normal form .

```
1 add :: Int -> (Int -> Int)
2
3 addOne :: Int -> Int
4 addOne x = add 1
5
6 addOne 5
7 ((add 1) 5)
8 6
```

4.2 Higher-order functions

Higher order functions are functions that can accept functions as parameters **flip** is an example of a higher-order function.

Its type signature is *flip :: (a → b → c) → b → a → c* and it can be partially applied like so

```
1 flipOne = flip 1
2 partialApply = flipOne 2 -- ((flip 1) 2)
```

4.3   Function composition

Composing functions is like a *right to left* pipeline. **f . g** can be read as **f** after **g**. In Haskell . is used since ∘ is not a valid ASCII character.

So (f ∘ g) x = f (g x)

1. first applies g

2. then applies f to the result of applying g

3. and makes a new function which takes a parameter x

Example of function composition in action

```
1 negate . sum $ [1,2,3,4,5]
2
3 -- which is equivalent to
4 negate (sum [1,2,3,4,5])
5 negate (15)
```

[4]

---

[4]$ is used since function application has the highest precedence, so Haskell will think we mean negate . 15. Alternatively, you can wrap it in brackets like **negate . (sum [1,2,3,4,5])**

5 Recursion

5.1 Factorial

A classic way of demonstrating recursion is factorial. 4! means 4 * 3 * 2 * 1. The broken down steps can be seen below.

```
4! = 4 * 3 * 2 * 1
        12  * 2 * 1
             24 * 1
                  24
4! = 24
```

The *factorial* function can be defined in Haskell. If there is no base case defined, the recursion will go on forever and will never stop.

```
1 fac :: Integer -> Integer
2 fac 0 = 1 -- base case
3 fac n = n * fac (n -1)
      5
```

5.2 More examples of recursion

Some more examples of recursion from *work/recursion.hs*.

```
1  -- Sums up the elements of a list using recursion
2  sum' [] = 0 -- base case
3  sum' (x:xs) = x + sum' xs
4
5  head' [] = error "empty list"
6  head' (x:xs) = x
7
8  -- Takes n elements from a list
9  take' n _
10   | n <= 0 = [] -- if n is 0, give back an empty list
11 take' n [] = [] -- base case 2
12 take' n (x:xs) = x:take' (n-1) xs -- prepend x to the list
13
14 -- Maps a function over a list
15 map' f [] = [] -- base case, empty list
16 map' f (x:xs) = f x:map' f xs
```

---

[5] *fac 0 = 1* is the base case since it will return 1, and stop the recursion.

5.3   Fold Styles

foldr folds a list from right to left, whereas *foldl* folds a list from left to right.

   foldl stacks parentheses on the left where as *foldr* stacks them on the right
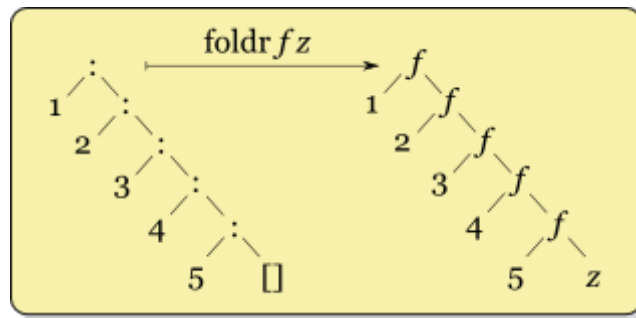$(((0+1)+2)+3)$ vs $(1+(2+(3+0)))$
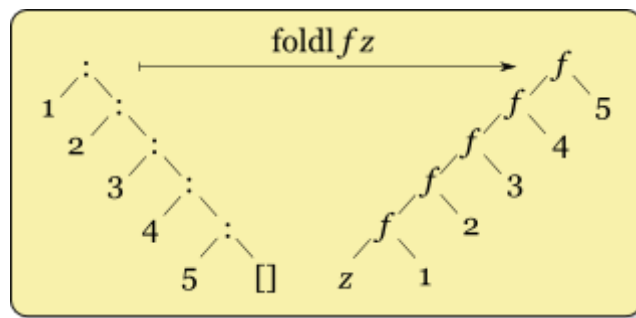

Figure 1: foldr recursion

Figure 2: foldl recursion

6 Weak head normal form

Values in Haskell are reduced to Weak head normal form (WHNF). Weak head normal form is a type of normal form which contains both the possibility the expression has been fully evaluated/reduced (normal form) as well as the possibility that the expression has been evaluated to the point of arriving at a data constructor/lambda waiting for an argument.

These expressions **are** in WHNF:

```
1 -- https://stackoverflow.com/questions/6872898/
2 (1, 1 + 1)          -- outermost part is the data constructor (,)
3 \x -> 2 + 2         -- outermost part is a lambda abstraction
4 'h' : ("e" + "llo") -- outermost part is the data constructor (:)
```

These examples **are not** in WHNF

```
1 1 + 2               -- the outermost part here is an application of (+)
2 (\x -> x + 1) 2     -- the outermost part is an application of (\x -> x +
   1)
3 "he" ++ "llo"       -- the outermost part is an application of (++)
```

We can use `:sprint` in GHCI to print out the representation of the value in memory. Due to laziness, polymorphic types are unevaluated (thunked) until we use them. This is marked with an underscore (_).

```
Prelude> let nums = [1..5]

Prelude> :sprint nums
nums = _

Prelude> take 2 nums
Prelude> :sprint nums
nums = 1 : 2 : _
```

⁶

---

[6]The examples on WHNF were taken directly from the amazing Haskell Book