Haskell Study Notes

Charlotte

April 29, 2021

Contents

1 Types

1.1 Basic types

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

```
"hello world" -- string
1234          -- integer
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 ¿

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

data Worker = Worker { name :: String, job :: String } deriving (Show)
data Student = Student { name :: String, school :: String } deriving (Show)

class Person a where
    getName :: a -> String
    getOccupation :: a -> String

instance Person Worker where
    getName x = name (x :: Worker)
    getOccupation x = "Working on " ++ job x

instance Person Student where
    getName x = name (x :: Student)
    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 1, and the *tail* is [2, 3, 4, 5]

The *splitAt* function splits a list into two parts at the element specified

```
Prelude> splitAt 5 [1..10]
([1,2,3,4,5],[6,7,8,9,10])
```

Lists can be indexed using the *!!* operator.

2.2 Tuples

Haskell has tuples, triples, and *n-tuples*. Tuples have a *fst* and *snd* function which respectively gets either the first or second item.

swap (defined in **Data.Tuple**) swaps the items in a *tuple*

```
("char", 20)

-- a triple
("char", 20, "turtles")
```

3 Conditionals

3.1 If expressions

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

```haskell
-- stolen from the haskell book
let x = 0
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*.

```haskell
pal xs =
  case y of
    True  -> "yes"
    False -> "no"
  where y = xs == reverse xs
```

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

```haskell
data Animal = Cat | Dog
speak a =
  case a of
    Cat -> "meow"
    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 like *function-name params* with each case on a new line starting with a pipe. Cases are written in the format:

"| *condition* = *value*."

Guards can also have fallback cases - marked as **otherwise**.

```
bloodNa :: Integer -> String
bloodNa x
```

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

```
isItOne :: Int -> Bool
isItOne 1 = True
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 <u>or</u> account number.

```
data User = User String Int

getName :: User -> String
getName (User name _) = name

getAccNumber :: User -> Int
getAccNumber (User _ acc) = acc
```

[1] [2]

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

[2] 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

```haskell
-- add takes two ints and returns an int
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 .

```haskell
add :: Int -> (Int -> Int)

addOne :: Int -> Int
addOne x = add 1

addOne 5
((add 1) 5)
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 $flip :: (a \to b \to c) \to b \to a \to c$ and it can be partially applied like so

```haskell
flipOne = flip 1
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**.

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

```
negate . sum $ [1,2,3,4,5]

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

[3]

[3]$ 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.

```haskell
fac :: Integer -> Integer
fac 0 = 1 -- base case
fac n = n * fac (n -1)
```

[4]

5.2 More examples of recursion

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

```haskell
-- Sums up the elements of a list using recursion
sum' [] = 0 -- base case
sum' (x:xs) = x + sum' xs

head' [] = error "empty list"
head' (x:xs) = x

-- Takes n elements from a list
take' n _
  | n <= 0 = [] -- if n is 0, give back an empty list
take' n [] = [] -- base case 2
take' n (x:xs) = x:take' (n-1) xs -- prepend x to the list

-- Maps a function over a list
map' f [] = [] -- base case, empty list
map' f (x:xs) = f x:map' f xs

-- foldl style recursion
```

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

5.3 Foldl vs foldr style recursion

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)))

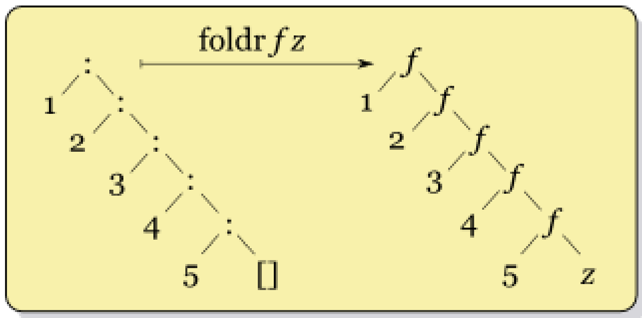

Figure 1: foldr recursion

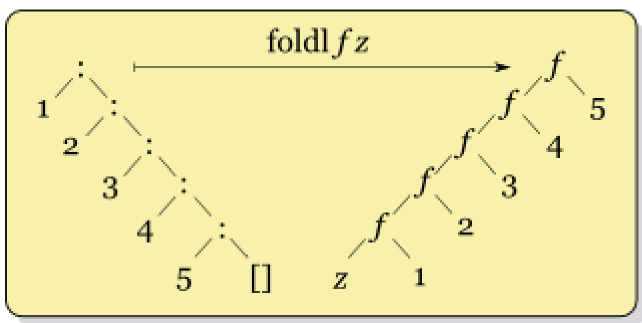

Figure 2: foldl recursion