Haskell continuation: Monads

Functions

Type class

Type class is sort of an interface that defines some behavior. A type can be made an instance of a typeclass if it supports that behavior. Example: The Int type is an instance of the Eq typeclass because the Eq typeclass defines behavior for stuff that can be equated.

In Haskell, the most common standard classes are the following ones:

**Num** numeric types

Contains types whose values are numeric, and as such can be processed using the following six methods: +, -, \*, negate, abs, signum.

**Integral** – integral types

Contains numeric types whose values are integers and as such support the methods of integer division and integer remainder: div and mod

**Fractional** – fractional types

Contains numeric types whose values are decimal/fractions and as such support the methods of fractional division and fractional reciprocation: / and recip

**Eq** equality types

Contains types whose values can be compared for equality and inequality using the following two methods == (is equal) and /=(is not equal).

All the basic types Bool, Char, String, Int, Integer, Float, and Double are instances of the Eq class, as are list and tuple types, provided that their element and component types are instances

**Ord** ordered types

Contains types that are ordered, and as such can be compared using these methods: <, <=, >, and >= and the functions min, max, and compare.

All the basic types Bool, Char, String, Int, Integer, Float, and Double are instances of the Ord class, as are list types and tuple types, provided that their element and component types are instances.

**Enum** is used in enumerations and lets you use syntax such as [Red .. Yellow].

**Show** showable types

Contains types whose values can be converted into strings of characters using the method show, show :: a -> String.

**Read** readable types

Contains types whose values can be converted from strings of characters using the following method: read :: String -> a

example:

> read "False" :: Bool

False

The use of :: in these examples resolves the type of the result, which would otherwise not be able to be inferred by GHCi. In practice, however, the necessary type information can usually be inferred automatically from the context

**Functor typeclass**

So far, we've encountered a lot of the typeclasses in the standard library. And now, we're going to take a look at the Functor typeclass, which is basically for things that can be mapped over.

Functors abstract programming patterns by mapping functions over specific types.

Example:

inc :: [Int] -> [Int]

inc [] = []

inc (x:xs) = x+1 : inc xs -- we increment the first element of the list and recursively deal with the rest of the elements in the same way.

sqr :: [Int] -> [Int]

sqr [] = []

sqr (x:xs) = x^2 : sqr xs -- we square the first element of the list and recursively deal with the rest of the elements in the same way.

-- These two functions have similar patterns and can be abstracted by the map function which applies a function to each element of a list in turns.

map' :: (a->b) -> [a] -> [b]

map' f [] = []

map' f (x:xs) = f x : map' f xs

-- We can define the inc and sqr function in terms of map as:

inc' = map (+1)

sqr' = map (n^2)

**Generalizing further**

The technique of mapping functions is not limited to list you can map to other data structures like trees, Maybe types

This is the purpose of functors.

-- Functor class definition:

class Functor f where

    fmap :: (a -> b) -> fa -> fb

For a type f to be an instance of the class Functor it must support the fmap method.

-- f represents the type constructor of parameterized type like lists [a], Maybe a and trees that take one type parameter.

Types that implement functors must support the fmap method.

-- Functor generalizes mapping for any parameterized type f

**fmap** takes a function from one type to another and a functor applied with one type and returns a functor applied with another type.

It is almost similar to map which takes a function from one type to another and a list of one type and returns a list of another type.

In fact, map is just a fmap that works only on lists. Here's how the list is an instance of the Functor typeclass.

We see that the f has to be a type constructor that takes one type. [a] is already a concrete type (of a list with any type inside it), while [] is a type constructor that takes one type and can produce types such as [Int], [String] or even [[String]].

Types that can act like a box can be functors. A more correct term for what a functor is would be **computational context**. The context might be that the computation can have a value or it might have failed (Maybe and Either a) or that there might be more values (lists), stuff like that.

If you mentally replace the **f**s with **Maybe**s, **fmap** acts like a **(a -> b) -> Maybe a -> Maybe b** for this particular type, which looks OK. But if you replace **f** with **(Maybe m)**, then it would seem to act like a **(a -> b) -> Maybe m a -> Maybe m b**, which doesn't make any damn sense because **Maybe** takes just one type parameter.