Haskell Tutorial: Functors, Applicatives, Monads

November 26, 2019

[1]:
```
:opt no-lint
import Control.Applicative
import Control.Monad
```

1 Functors

A very basic concept of Haskell is the application of a function to a value.

[2]:
```
(*5) 2
```

```
10
```

Sometimes, we have values that are contained in a certain context. For example, `Just 2` is the value 2 contained in the context `Just`. Another example is the value 2 contained in a list: `[2]`. We cannot directly apply the function `(*5)` to `Just 2` or `[2]` since we first have to obtain the value 2 from its context. You can think of the context as a box that contains a value. For lists, we already know that we can apply a function to all list elements by using the `map` function. A generalization of `map` is the function `fmap :: (a -> b) -> f a -> f b`.

[3]:
```
map  (*5) [2]
fmap (*5) [2]
fmap (*5) (Just 2)
fmap (*5) Nothing
```

```
[10]
```

```
[10]
```

```
Just 10
```

```
Nothing
```

Types that implement `fmap` are called functors. A functor is formally defined as a type class:

[4]:
```
-- class Functor f where
--     fmap :: (a -> b) -> f a -> f b
```

Types implementing the `Functor` type class, i.e., types that are instances of the `Functor` type class, implement `fmap` in a way that is consistent with the nature of the context.

[5]:
```
fmap' :: (a -> b) -> [a] -> [b]
fmap' _ [] = []
fmap' f (x:xs) = f x : map f xs

fmap' :: (a -> b) -> Maybe a -> Maybe b
fmap' f (Just x) = Just $ f x
fmap' f Nothing = Nothing
```

Haskell commonly defines the special operator <$> for `fmap`.

[6]:
```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap

fmap (*5) [2]
(*5) <$> [2]
fmap (*5) (Just 2)
(*5) <$> (Just 2)
```

```
[10]
```

```
[10]
```

```
Just 10
```

```
Just 10
```

Interestingly, functions are functors as well. Hence, we can apply `fmap` to functions, giving us function composition.

[7]:
```
f = fmap (*5) (+5)
g = (*5) <$> (+5)
h = (*5) . (+5)

f 0
g 0
h 0
```

```
25
```

```
25
```

```
25
```

Types implementing the `Functor` type class have to implement `fmap` such that the following functor laws hold (`id` is the identity function):

[8]:
```
-- fmap id = id
-- fmap (f . g) = fmap f . fmap g
```

2 Applicative

The `Applicative` type class extends the `Functor` type class. While the `Functor` type class assumes that the values are wrapped in a context (a box), `Applicative` also supports functions wrapped in a context (a box). The `Applicative` type class is defined as follows:

[9]:
```
-- class Functor f => Applicative f where
--    pure  :: a -> f a
--    (<*>) :: f (a -> b) -> f a -> f b
```

The function `pure` takes a value and puts it into the context (a box). The operator `(<*>)` applies a function in a context to a value in a context and it returns a value in a context.

[10]:
```
Just (*5) <*> Just 2
[(*5)] <*> [2]

[(*1),(*2),(*3)] <*> [1..3]
```

```
Just 10
```

```
[10]
```

```
[1,2,3,2,4,6,3,6,9]
```

The following example combines `<$>` (fmap) with `<*>`. The first parenthesis applies the `(*)` function to `Just 5`, which givs us the function `Just (*5)`. This function is then applied to `Just 2`, which gives us `Just 10`.

[11]:
```
((*) <$> (Just 5)) <*> (Just 2)
```

```
Just 10
```

Types implementing the `Applicative` type class have to implement the `<*>` operator and the function `pure` such that the following applicative laws hold (`id` is the identity function):

[12]:
```
-- pure id <*> v = v
-- pure f <*> pure x = pure (f x)
-- u <*> pure y = pure ($ y) <*> u
-- u <*> (v <*> w) = pure . <*> u <*> v <*> w
```

The last applicative law says that `<*>` is associative.

3 Monads

The `Monad` type class extends the `Applicative` type class. It defines a function `return` that puts a value into a monad and a function `(>>=)` called *bind* that takes a value in a monad (a box), a function that takes a value and returns a value in a possibly different monad, and returns the later monadic value.

[13]:
```
-- class Applicative m => Monad m where
--     return :: a -> m a
--     (>>=)  :: m a -> (a -> m b) -> m b
```

To illustrate this idea, we use the type `Maybe`, which happens to be an instance of the Monad type class. Lets start with a function that takes and `Integer` and returns a `Maybe Integer`.

[14]:
```
half :: Integer -> Maybe Integer
half x = if even x
         then Just (x `div` 2)
         else Nothing

half 8
```

```
Just 4
```

Unfortunately, we cannot compose `half` with itself since `half` takes an `Integer` but returns a boxed value. This is where the bind operator can help. (Note that `return` is a rather confusing name for what the function does.) There is another operator `(=<<)`, which swaps the first two arguments, which is sometimes handy. Note that the operators kind of indicate how the value is flowing through a sequence of functions.

[15]:
```
Just 8 >>= half
Just 4 >>= half
Just 2 >>= half
Just 1 >>= half

return 8 >>= half >>= half >>= half >>= half
half =<< half =<< half =<< half =<< return 8
```

```
Just 4
```

```
Just 2
```

```
Just 1
```

```
Nothing
```

```
Nothing
```

```
Nothing
```

Types implementing the `Monad` type class have to implement the `>>=` operator and the function `return` such that the following monad laws hold (`id` is the identity function):

[16]:
```
-- return a >>= f = f a
-- m >>= return = m
-- (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Monds play an important role in Haskell since they can be used to encapsulate side effects. For example, the `IO` Monad takes care of input and output operations. The `getLine :: IO String` function takes no arguments and returns an IO action to read a string from the input. The `putStrLn :: String -> IO String` takes a string and returns an IO action to print it. These functions can be chained together:

[17]:
```
getLine >>= putStrLn
```

```
some input typed in here
```

In case the result of a chained function is not needed, one can use the *then* (`>>`) operator.

[18]:
```
putStr "Hello" >> putStr " " >> putStrLn "World"
```

```
Hello World
```

Haskell has a special notation for monads, the `do` notation. Using the do notation, code manipulating monads starts to look like imperative code. Hence, some people call monads *programmable semicolons* since semicolons are often used to sequence statements in imperative languages. However, in Haskell, the behaviour of monads is programmable.

[19]:
```
do
    putStr "Hello"
    putStr " "
    putStrLn "World"

do
    line <- getLine
    putStrLn line
```

```
Hello World
```

```
some more input typed in here
```

4 Summary

Below is a summary of the operators defined by the type classes discussed here. Recall that a `Monad` type is also `Applicative` and that an `Applicative` type is also a `Functor`.

[20]:
```
-- (<$>) ::   (a -> b) -> f a -> f b    -- Functor
-- (<*>) :: f (a -> b) -> f a -> f b    -- Applicative
-- (=<<) :: (a -> m b) -> m a -> m b    -- Monad
-- (>>=) :: m a -> (a -> m b) -> m b    -- Monad
```