Haskell/Understanding monads

Monads are very useful in Haskell, but the concept is often difficult at first. Since they have so many applications, people often explain them from a particular point of view, and that can confuse your understanding of monads in their full glory.

Historically, monads were introduced into Haskell to perform input/output. A predetermined execution order is crucial for things like reading and writing files, and monadic operations follow an inherent sequence. We discussed sequencing and IO back in Simple input and output using the `do` notation. Well, `do` is actually just syntactic sugar over monads.

Monads are by no means limited to input and output. Monads support a whole range of things like exceptions, state, non-determinism, continuations, coroutines, and more. In fact, thanks to the versatility of monads, none of these constructs needed to be built into Haskell as a language; instead, they are defined by the standard libraries.

Definition

A *monad* is defined by three things:

- a type constructor `m`;
- a function `return`;[1]
- an operator `(>>=)` which is pronounced "bind".

The function and operator are methods of the `Monad` type class and have types

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

and are required to obey three laws that will be explained later on.

For a concrete example, take the `Maybe` monad. The type constructor is `m = Maybe`, while `return` and `(>>=)` are defined like this:

```
return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
```

```
            Just x  -> g x
```

`Maybe` is the monad, and `return` brings a value into it by wrapping it with `Just`. As for `(>>=)`, it takes a `m :: Maybe a` value and a `g :: a -> Maybe b` function. If `m` is `Nothing`, there is nothing to do and the result is `Nothing`. Otherwise, in the `Just x` case, `g` is applied to `x`, the underlying value wrapped in `Just`, to give a `Maybe b` result, which might be `Nothing`, depending on what `g` does to `x`. To sum it all up, if there is an underlying value in `m`, we apply `g` to it, which brings the underlying value back into the `Maybe` monad.

The key first step to understand how `return` and `(>>=)` work is tracking which values and arguments are monadic and which ones aren't. As in so many other cases, type signatures are our guide to the process.

Motivation: Maybe

To see the usefulness of `(>>=)` and the `Maybe` monad, consider the following example: Imagine a family database that provides two functions

```
    father :: Person -> Maybe Person
    mother :: Person -> Maybe Person
```

These look up the name of someone's father or mother. In case our database is missing some information, `Maybe` allows us to return a `Nothing` value instead of crashing the program.

Let's combine our functions to query various grandparents. For instance, the following function looks up the maternal grandfather:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom
```

Or consider a function that checks whether both grandfathers are in the database:

```
    bothGrandfathers :: Person -> Maybe (Person, Person)
    bothGrandfathers p =
        case father p of
            Nothing -> Nothing
            Just dad ->
                case father dad of
                    Nothing -> Nothing
                    Just gf1 ->                         -- found first grandfather
                        case mother p of
```

```
                        Nothing -> Nothing
                        Just mom ->
                            case father mom of
                                Nothing -> Nothing
                                Just gf2 ->          -- found second grandfather
                                    Just (gf1, gf2)
```

What a mouthful! Every single query might fail by returning `Nothing` and the whole function must fail with `Nothing` if that happens.

Clearly there has to be a better way to write that instead of repeating the case of `Nothing` again and again! Indeed, that's what the `Maybe` monad is set out to do. For instance, the function retrieving the maternal grandfather has exactly the same structure as the `(>>=)` operator, so we can rewrite it as:

```
   maternalGrandfather p = mother p >>= father
```

With the help of lambda expressions and `return`, we can rewrite the two grandfathers function as well:

```
   bothGrandfathers p =
      father p >>=
          (\dad -> father dad >>=
              (\gf1 -> mother p >>=   -- this line works as "\_ -> mother p", but naming gf1 allows lat
                  (\mom -> father mom >>=
                      (\gf2 -> return (gf1,gf2) ))))
```

While these nested lambda expressions may look confusing to you, the thing to take away here is that `(>>=)` releases us from listing all the `Nothing`s, shifting the focus back to the interesting part of the code.

To be a little more precise: The result of `father p` is a monadic value (in this case, either `Just dad` or `Nothing`, depending on whether `p`'s dad is in the database). As the `father` function takes a regular (non-monadic value), the `>>=` feeds `p`'s `dad` to it as a *non-monadic* value. The result of `father dad` is then monadic again, and the process continues.

So, `>>=` helps us pass non-monadic values to functions without leaving a monad. In the case of the `Maybe` monad, the monadic aspect is the qualifier that we don't know with certainty whether the value will be found.

Type class

In Haskell, the `Monad` type class is used to implement monads. It is provided by the Control.Monad (http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html) module and included in the Prelude. The class has the following methods:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a
```

Aside from return and bind, notice the two additional functions `(>>)` and `fail`.

The operator `(>>)` called "then" is a mere convenience and commonly implemented as

```
m >> n = m >>= \_ -> n
```

`>>` sequences two monadic actions when the second action does not involve the result of the first, which is common for monads like `IO`.

```
printSomethingTwice :: String -> IO ()
printSomethingTwice str = putStrLn str >> putStrLn str
```

The function `fail` handles pattern match failures in `do` notation. It's an unfortunate technical necessity and doesn't really have anything to do with monads. You are advised not to call `fail` directly in your code.

Notions of Computation

We've seen how `(>>=)` and `return` are very handy for removing boilerplate code that crops up when using `Maybe`. That, however, is not enough to justify why monads matter so much. We will continue our monad studies by rewriting the two-grandfathers function using `do` notation with explicit braces and semicolons. Depending on your experience with other programming languages, you may find this very suggestive:

```
bothGrandfathers p = do {
    dad <- father p;
    gf1 <- father dad;
    mom <- mother p;
    gf2 <- father mom;
    return (gf1, gf2);
  }
```

If this looks like a code snippet of an imperative programming language to you, that's because it is. In particular, this imperative language supports *exceptions* : `father` and `mother` are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole `do`-block will fail, i.e. terminate with

an exception.

In other words, the expression `father p`, which has type `Maybe Person`, is interpreted as a statement of an imperative language that returns a `Person` as result. This is true for all monads: a value of type `M a` is interpreted as a statement of an imperative language that returns a value of type `a` as result; and the semantics of this language are determined by the monad `M`.[2]

Under this interpretation, the bind operator `(>>=)` is simply a function version of the semicolon. Just like a `let` expression can be written as a function application,

```
  let x = foo in x + 3          corresponds to      (\x -> x + 3) foo
```

an assignment and semicolon can be written as the bind operator:

```
  x <- foo; return (x + 3)      corresponds to      foo >>= (\x -> return (x + 3))
```

The `return` function lifts a value `a` to `M a`, a full-fledged statement of the imperative language corresponding to the monad `M`.

Different semantics of the imperative language correspond to different monads. The following table shows the classic selection that every Haskell programmer should know. If the idea behind monads is still unclear to you, studying each of the examples in the following chapters will not only give you a well-rounded toolbox but also help you understand the common abstraction behind them.

Monad	Imperative Semantics	Wikibook chapter
`Maybe`	Exception (anonymous)	Haskell/Understanding monads/Maybe
`Error`	Exception (with error description)	Haskell/Understanding monads/Error
`State`	Global state	Haskell/Understanding monads/State
`IO`	Input/Output	Haskell/Understanding monads/IO
`[]` (lists)	Nondeterminism	Haskell/Understanding monads/List
`Reader`	Environment	Haskell/Understanding monads/Reader
`Writer`	Logger	Haskell/Understanding monads/Writer

Furthermore, these different semantics need not occur in isolation. As we will see

in a few chapters, it is possible to mix and match them by using monad transformers to combine the semantics of multiple monads in a single monad.

Monad Laws

In Haskell, every instance of the `Monad` type class (and thus all implementations of bind `(>>=)` and `return`) must obey the following three laws:

```
m >>= return     =  m                         -- right unit
return x >>= f   =  f x                       -- left unit

(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)  -- associativity
```

Return as neutral element

The behavior of `return` is specified by the left and right unit laws. They state that `return` doesn't perform any computation, it just collects values. For instance,

```
maternalGrandfather p = do
        mom <- mother p
        gf  <- father mom
        return gf
```

is exactly the same as

```
maternalGrandfather p = do
        mom  <- mother p
        father mom
```

by virtue of the right unit law.

Associativity of bind

The law of associativity makes sure that (like the semicolon) the bind operator `(>>=)` only cares about the order of computations, not about their nesting; e.g. we could have written `bothGrandfathers` like this (compare with our earliest version without `do`):

```
bothGrandfathers p =
   (father p >>= father) >>=
       (\gf1 -> (mother p >>= father) >>=
           (\gf2 -> return (gf1,gf2) ))
```

The associativity of the *then* operator `(>>)` is a special case:

```
(m >> n) >> o  =  m >> (n >> o)
```

Monadic composition

It is easier to picture the associativity of bind by recasting the law as

```
(f >=> g) >=> h  =  f >=> (g >=> h)
```

where **`(>=>)` is the monad composition operator**, a close analogue of the function composition operator `(.)`, only with flipped arguments. It is defined as:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >=> g = \x -> f x >>= g
```

We can also flip monad composition to go the other direction using `(<=<)`. The operation order of `(f . g)` is the same as `(f' <=< g')`.[3]

Monads and Category Theory

Monads originally come from a branch of mathematics called Category Theory. Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell. The definition of monads in Category Theory actually uses a slightly different presentation. Translated into Haskell, this presentation gives an alternative yet equivalent definition of a monad which can give us some additional insight.[4]

So far, we have defined monads in terms of `(>>=)` and `return`. The alternative definition, instead, starts with monads as functors with two additional combinators:

```
fmap   :: (a -> b) -> M a -> M b  -- functor

return :: a -> M a
join   :: M (M a) -> M a
```

(Reminder: as discussed in the chapter on the functor class, a functor `M` can be thought of as container, so that `M a` "contains" values of type `a`, with a corresponding mapping function, i.e. `fmap`, that allows functions to be applied to values inside it.)

Under this interpretation, the functions behave as follows:

- `fmap` applies a given function to every element in a container
- `return` packages an element into a container,
- `join` takes a container of containers and flattens it into a single container.

With these functions, the bind combinator can be defined as follows:

```
m >>= g = join (fmap g m)
```

Likewise, we could give a definition of `fmap` and `join` in terms of `(>>=)` and `return`:

```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

Is my Monad a Functor?

At this point we might, with good reason, conclude that all monads are by definition functors as well. That is indeed the case, both according to category theory and when programming in Haskell. When we presented the `Monad` methods just above, we omitted the following class constraint:

```
class Applicative m => Monad m where
    -- etc.
```

`Applicative` is an important class on its own merits, and we will study it at length soon. For now, all you need to know about it is that it inherits from `Functor`, and therefore so does `Monad` [5].

A final observation is that `Control.Monad` defines `liftM`, a function with a strangely familiar type signature...

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
```

As you might suspect, `liftM` is merely `fmap` implemented with `(>>=)` and `return`, just as we have done above. For a properly implemented monad with a matching `Functor` (that is, any *sensible* monad) `liftM` and `fmap` are interchangeable.

> *Note*
>
> While following the next few chapters, you will likely want to write instances of `Monad` and try them out, be it to run the examples in the book or to do other experiments you might think of. However, `Applicative` being a

superclass of `Monad` means that implementing `Monad` requires providing `Functor` and `Applicative` instances as well. At this point of the book, that would be somewhat of an annoyance, especially given that we have not discussed `Applicative` yet! As a workaround, once you have written the `Monad` instance you can use the functions in `Control.Monad` to fill in the `Functor` and `Applicative` implementations, as follows:

```
instance Functor Foo where
    fmap = liftM

instance Applicative Foo where
    pure = return
    (<*>) = ap
```

We will find out what `pure`, `(<*>)` and `ap` are in due course.

There is no need to do so right now, but if you are curious about what `Applicative` is you might want to have a brief look at the chapter about it. For the moment, stick to the first few sections ("Functor recap", "Application in functors" and "The Applicative class"), as what follows builds on the chapters about monads we are going to go through now.

Notes

1. This `return` function has nothing to do with the `return` keyword found in imperative languages like C or Java; don't conflate these two.
2. By "semantics", we mean what the language allows you to say. In the case of `Maybe`, the semantics allow us to express failure, as statements may fail to produce a result, leading to the statements that follow it being skipped.
3. Of course, the functions in regular function composition are non-monadic functions whereas monadic composition takes only monadic functions.
4. Deep into the Advanced Track, we will cover the theoretical side of the story in the chapter on Category Theory.
5. This important superclass relation was, thanks to historic accidents, only implemented quite recently (early 2015) in GHC (version 7.10). If you are using an older GHC version you might find the class constraint isn't there.

/Understanding_monads&oldid=2991875"
