Haskell/Understanding monads

There is a certain mystique about monads, and even about the word "monad" itself. While one of our goals of this set of chapters is removing the shroud of mystery that is often wrapped around them, it is not difficult to understand how it comes about. Monads are very useful in Haskell, but the concept is often difficult to grasp at first. Since monads have so many applications, people often explain them from a particular point of view, which can derail your efforts towards understanding them in their full glory.

Historically, monads were introduced into Haskell to perform input and output – that is, I/O operations of the sort we dealt with in the Simple input and output chapter and the prologue to this unit. A predetermined execution order is crucial for things like reading and writing files, and monadic operations lend themselves naturally to sequencing. However, monads are by no means limited to input and output. They can be used to provide a whole range of features, such as 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; rather, they are defined by the standard libraries.

In the Prologue chapter, we began with an example and used it to steadily introduce several new ideas. Here, we will do it the other way around, starting with a definition of monad and, from that, building connections with what we already know.

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. Note that this result may or may not be `Nothing`, depending on what `g` does to `x`. To sum it all up, if there is an *underlying value* of type `a` 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 relevant information, `Maybe` allows us to return a `Nothing` value to indicate that the lookup failed, rather than crashing the program.

Let's combine our functions to query various grandparents. For instance, the following function looks up the maternal grandfather (the father of one's mother):

```
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 >>=   -- gf1 is only used in the final return
               (\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 father 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 actually leaving a monad. In the case of the `Maybe` monad, the monadic aspect is the uncertainty about whether a 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 Applicative m => 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, there are two additional methods, `(>>)` and `fail`. Both of them have default implementations, and so you don't need to provide them when writing an instance.

The operator `(>>)`, spelled "then", is a mere convenience and has the default implementation

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

`(>>)` sequences two monadic actions when the second action does not involve the result of the first, which is a common scenario for monads such as `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.

Monad and Applicative

An important thing to note is that `Applicative` is a superclass of `Monad`.[2] That has a few consequences worth highlighting. First of all, every `Monad` is also a `Functor` and an `Applicative`, and so `fmap`, `pure`, `(<*>)` can all be used with monads. Secondly, actually writing a `Monad` instance also requires providing `Functor` and `Applicative` instances. We will discuss ways of doing that later in this chapter. Thirdly, if you have worked through the Prologue, the types and roles of `return` and `(>>)` should look familiar...

```
(*>) :: Applicative f => f a -> f b -> f b
(>>) :: Monad m => m a -> m b -> m b

pure :: Applicative f => a -> f a
return :: Monad m => a -> m a
```

The only difference between the types of `(*>)` and `(>>)` is that the constraint changes from `Applicative` to `Monad`. In fact, that is the only difference between the methods: if you are dealing with a `Monad` you can always replace `(*>)` and `(>>)`, and vice-versa. The same goes for `pure`/`return` – in fact, it is not even necessary to implement `return` if there is an independent definition of `pure` in the `Applicative` instance, as `return = pure` is provided as a default definition of `return`.

Notions of Computation

We have 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. Our next step towards that will be rewriting the two-grandfathers function in a quite different-looking style: 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 in 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, raising an exception instead; and when that happens, the whole `do`-block will *fail*, i.e. terminate with an exception (meaning, evaluate to `Nothing`, here).

In other words, the expression `father p`, which has type `Maybe Person`, is interpreted as a statement in an imperative *language* that returns a `Person` as the result, or fails.

This is true for all monads: a value of type `M a` is interpreted as a *statement* in an imperative language `M` that returns a value of type `a` as its result; and the semantics of this language are determined by the monad `M`.[3]

Under this interpretation, the *then* operator `(>>)` is simply an implementation of the semicolon, and `(>>=)` – of the semicolon and assignment (binding) of the result of a previous computational step. Just like a `let` expression can be written as a function application,

```
let x = foo in (x + 3)        corresponds to      foo  &  (\x -> id (x + 3))      -- v & f = f v
```

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

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

In case of functions, `&` and `id` are trivial; in case of a monad, `>>=` and `return` are substantial.

The `&` operator combines together two pure *calculations*, `foo` and `id (x + 3)`, while creating a new binding for the variable `x` to hold `foo`'s *value*, making `x` available to the second calculational step, `id (x + 3)`.

The bind operator `>>=` combines together two *computational* steps, `foo` and `return (x + 3)`, in a manner particular to the monad `M`, while creating a new binding for the variable `x` to hold `foo`'s *result*, making `x` available to the next computational step, `return (x + 3)`. In the particular case of `Maybe`, if `foo` will fail to produce a result, the second step is skipped and the whole combined computation will fail right away as well.

The function `return` lifts a plain value `a` to `M a`, a statement in the imperative language `M`, which statement, when executed / run, will result in the value `a` without any additional effects particular to `M`. This is ensured by Monad Laws, `foo >>= return === foo` and `return x >>= k === k x`; see below.

> *Note*
>
> The fact that `(>>=)`, and therefore `Monad`, lies behind the left arrows in `do`-blocks explains why we were not able to explain them in the Prologue, when we only knew about `Functor` and `Applicative`. `Applicative` would be enough to provide some, but not all, of the functionality of a `do`-block.

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
`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
`State`	Global state	Haskell/Understanding monads/State

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
```

There is also `(<=<)`, which is flipped version of `(>=>)`. When using it, the order of composition matches that of `(.)`, so that in `(f <=< g)` `g` comes first.[4]

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 on the `Monad` class.[5]

So far, we have defined monads in terms of `(>>=)` and `return`. The alternative definition, instead, treats 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
```

For the purposes of this discussion, we will use the functors-as-containers metaphor discussed in the chapter on the functor class. According to it, 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
```

`liftM` and Friends

Earlier, we pointed out that every `Monad` is an `Applicative`, and therefore also a `Functor`. One of the consequences of that was `return` and `(>>)` being monad-only versions of `pure` and `(*>)` respectively. It doesn't stop there, though. For one, `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 in the previous section. `liftM` and `fmap` are therefore interchangeable.

Another `Control.Monad` function with an uncanny type is `ap`:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

Analogously to the other cases, `ap` is a monad-only version of `(<*>)`.

There are quite a few more examples of `Applicative` functions that have versions *specialised* to `Monad` in `Control.Monad` and other base library modules. Their existence is primarily due to historical reasons: several years went by between the introductions of `Monad` and `Applicative` in Haskell, and it took an even longer time for `Applicative` to become a superclass of `Monad`, thus making usage of the specialised variants optional. While in principle there is little need for using the monad-only versions nowadays, in practice you will see `return` and `(>>)` all the time in other people's code – at this point, their usage is well established thanks to more than two decades of Haskell praxis without `Applicative` being a superclass of `Monad`.

> *Note*
>
> Given that `Applicative` is a superclass of `Monad`, the most obvious way of implementing `Monad` begins by writing the `Functor` instance and then moving down the class hierarchy:
>
> ```
> instance Functor Foo where
>     fmap = -- etc.
>
> instance Applicative Foo where
>     pure = -- etc.
>     (<*>) = -- etc.
>
> instance Monad Foo where
>     (>>=) = -- etc.
> ```
>
> 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, writing the instances in the manner shown above requires implementing `pure` and `(<*>)`, which is not a comfortable task at this point of the book as we haven't covered the `Applicative` laws yet (we will only do so at the applicative functors chapter). Fortunately, there is a workaround: implementing just `(>>=)` and `return`, thus providing a self-sufficient `Monad` instance, and then using `liftM`, `ap` and `return` to fill in the other instances:
>
> ```
> instance Monad Foo where
>     return = -- etc.
>     (>>=) = -- etc.
>
> instance Applicative Foo where
>     pure = return
>     (<*>) = ap
>
> instance Functor Foo where
>     fmap = liftM
> ```
>
> The examples and exercises in this initial series of chapters about monads will not demand writing `Applicative` instances, and so you can use this workaround until we discuss `Applicative` in detail.

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. This important superclass relationship was, thanks to historic accidents, only implemented quite recently (early 2015) in GHC (version 7.10). If you are using a GHC version older than that, this class constraint will not exist, and so some of the practical considerations we will make next will not apply.
3. 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.
4. Of course, the functions in regular function composition are non-monadic functions whereas monadic composition takes only monadic functions.
5. Deep into the Advanced Track, we will cover the theoretical side of the story in the chapter on Category Theory.
