Monads in Haskell

Nathanael Schilling

December 12, 2014

Abstract

In Haskell, monads provide a mechanism for mapping functions of the type a `->` $m\ b$ to those of the type $m\ a$ `->` $m\ b$. This mapping is dependent on m, hence the behaviour of the resulting function can be made specific to m, allowing for a wide range of notions ranging from non-determenism to failure-handling to be modelled. Additionally, the way monads behave make it possible to use both functions with effects and those without effects in the same program whilst keeping them separated, letting Haskell preserve its functional aspect without losing the ability to run programs that have effects. Due to the utility of monadic programming (i.e. programming with monads), Haskell has a specific syntax (called 'do notation') specifically for programming with monads. This article describes what monads are in Haskell, and how to program with them. Examples of monads are given including the List, Maybe and IO monad.

1 Introduction

Initially having its roots in Category Theory, the concept of a monad was applied to the theory of programming languages by Moggi in 1989 [6, 10] leading to the inclusion of monadic features into Haskell soon after, such as monadic IO in 1996 [6]. Despite the fact that the concept of a monad is a purely functional one, monads are especially useful for modelling programs that depend on interaction with the external world (including IO) in the context of functional programming languages without violating the functional purity of the language itself. This is done by forcing any function application on external input to take place within a monad, making it impossible to write functions whose output depends on external factors. Section 2 of this article gives a definition of a monad (in the context of functional programming languages), along with the corresponding definition in Haskell. Section 3 contains examples of monads, and examples of how to use them. Section 4 looks at do notation, followed by Section 5 which briefly introduces some Monadic combinators. Section 6 contains a description of the IO Monad. Finally, a conclusion is given.

2 Definition of a Monad

2.1 The `Hask` Category and Functors

To understand monads, it can be helpful to compare them with functors. Both of these concepts are best understood by looking at Haskell in the context of Category Theory. Every type in Haskell can be seen as a node (or "object") in a directed multigraph, and (non-polymorphic, terminating) functions as labelled edges (or "arrows") between these objects. This, along with an identity arrow for each type and an associative rule for function composition, characterizes a Category [13]. The category consisting of Haskell types and arrows in the way described above will be referred to as `Hask` hereafter, and the set of objects of this Category (i.e. Haskell types) as Ob(`Hask`). Functions are considered equal if they give the same output on all values[1] [4].

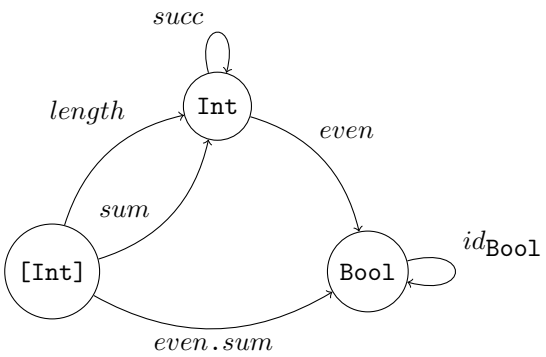

Figure 1: Some Objects and Arrows in `Hask`

Figure 1 introduces the following notation: id_a refers to the Haskell function `id::(a->a)`. The other functions in Figure 1 are hypothetical non-polymorphic Haskell functions of the given types. For example, the arrow *length* in Figure 1 could refer to the function `length::[Int]->Int` from Prelude. To distinguish between functions in the mathematical sense and Haskell functions, the following notation is used: $f : X \to Y$ refers to a mathematical function from X to Y. In contrast to this, a `->` b, where $a, b \in$ Ob(`Hask`) refers to the set of `Hask` arrows from a to b (which is identified with the `Hask` object of the same name). When the definition of a type of a function uses both notations and that function can be implemented in Haskell, all occurences of $\to$ can be replaced with `->`. If x is of the type a, i.e x is an element of an object of `Hask` (Note that in `Hask`, arrows also have types), we write $x :: a$. Function composition of two `Hask` functions f and g is written as f `.` g.

A *functor* is a structure preserving mapping from one category to another. Functors can be thought as the category-theoretic equivalent of homomorphisms

[1] Note that this is not intended as a rigorous definition of a Category. The framework of Category Theory is only used to provide a better understanding of monads and functors and not to prove any rigorous results.

in graph theory. We consider only functors from `Hask` to `Hask` in this article. Such a functor can be formally defined as a pair of functions f_0 and f_1 so that f_0 : Ob(`Hask`) $\rightarrow$ Ob(`Hask`) and f_1 : Hom(`Hask`) $\rightarrow$ Hom(`Hask`), where Hom(`Hask`) refers to the union of all the sets a `->` b, where $a, b \in$ Ob(`Hask`) so that the following holds [9]:

1. If $g :: a$ `->` b, then $f_1(g) :: f_0(a)$ `->` $f_0(b)$.
2. For all $a \in$ Ob(`Hask`), $f_1(id_a) = id_{f_0(a)}$.
3. If $g, h \in$ Hom(`Hask`) then $f_1(g$ `.` $h) = f_1(g)$ `.` $f_1(h)$.

An example of a functor is $f_0(a) = List\ a$, and $f_1(f) = map\ f$. It can be easily verified that this is indeed a functor.
Functors can be described in Haskell using the Functor typeclass. The function f_0 is, however, not a Haskell function, but is implicitly the function $a \mapsto m\ a$,where m is in the Functor typeclass.[2] The function f_1 is simply $fmap$, and the Haskell functor laws together with type of $fmap$ correspond the rules given in the definition above.

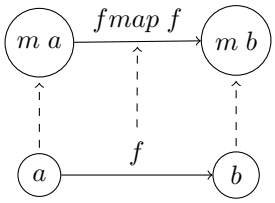

Figure 2: A Functor in `Hask`

2.1.1 Monads in `Hask`

Functors are useful if we want to lift 'simple' functions of the form a `->` b to 'higher' functions of the form $m\ a$ `->` $m\ b$ (Figure 2). However, it is often the case that we wish to not only lift 'simple' functions but that it would be useful to have a way to lift functions of the form a `->` $m\ b$ (Figure 3). For example, taking the definition of 'fmap' for lists, we would like to be able to lift Haskell functions like $f\ a = [a + 1]$ to the "List Level" in a way that does not simply create a list of lists. Monads provide exactly such a way of lifting functions. Formally speaking, a monad has the following definition [11, 14]:

A monad is a triple $(m, return, \star)$ where m : Ob(`Hask`) $\rightarrow$ Ob(`Hask`) and $return_a :: a$ `->` $m\ a$, and $\star : (a$ `->` $m\ b) \rightarrow (m\ a$ `->` $m\ b)$ satisfying:

1. $\star(return_a) = id_{m\ a}$
2. $\star(f)$ `.` $return_a = f$
3. $\star(g)$ `.` $\star(f) = \star(\star(g)$ `.` $f)$

where f and g are any Haskell functions composable in the way described above.

[2]The fact that f_0 is restricted this way means that non-injective functors are among the functors that cannot be described by the Functor typeclass.

Like f_0 for functors, m is simply the type constructor of the monad. More interesting are *return* and $\star$. As can be seen from the definition of *return*, it maps an element of a type to a corresponding element of the monadic type. The first monad rule implies that $\star(return_b)\ .\ \star(f) = \star(f)$, the second one is $\star(f)\ .\ return_a = f$ which show similarities to the concept of left and right identity respectively.
To give an example: for the list monad in Haskell, $return_a(x) = [x]$.

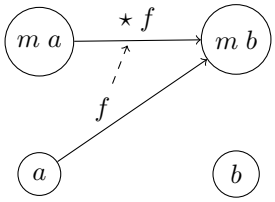

Figure 3: Monadic Lifting

The $\star$ function takes an element of a `->` $m\ b$ and maps it to an element of $m\ a$ `->` $m\ b$. What the three monad laws say is that the resulting functions should behave in a way that is somehow similar to what the original functions did. Applying $\star$ to $return_a$ should result in the identity function, and applying $return_a$ before $\star(f)$ should be equal to f. The latter statement is equivalent to saying any two "paths" through the following diagram are equal.

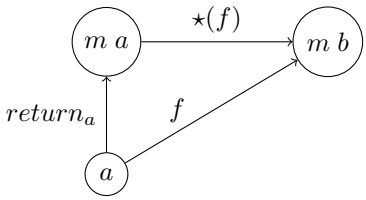

Figure 4: Illustration of the Second Monad Law

The third monad law deals with composition, and states that lifting the composition of an unlifted function with a lifted function is the same as combining the lifted version of both functions

2.2 The Monad Typeclass in Haskell

Monads are defined essentially the same way in Haskell as the formal definition given above, except for the fact that the equivalent of the $\star$ function in Haskell is called `>>=`, and can be defined in the following way in terms of $\star$:

$$x \texttt{ >>= } f = \star(f)\ x$$

With that in mind, the following gives the definition of the `Monad` typeclass in Haskell:

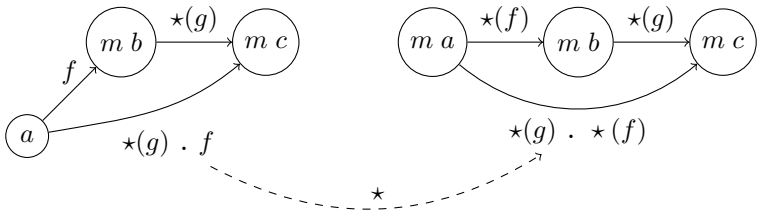

Figure 5: Illustration of the Third Monad Law.
The dashed arrow shows the application of the $\star$ function.

```
class Monad m where
    return :: a -> m a

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
```

The `>>=` and `return` functions have been described already. The `>>` function is useful for cases where we want to sequence two monadic actions whilst discarding the value of the first one. This function is described further in the examples given below.

An interesting function that is related to monads is the *join* function, which can be defined as $\star(id_{m\ a})$, meaning it has the type $m(\ m\ a)$ `->` $m\ a$. In short, *join* removes one layer of monadic structure. It is possible to give an alternative definition of a monad in terms of only *return*, *join* and *fmap* [14], but this is not discussed futher in this article.

The monad laws given earlier in Haskell form are [17]:

```
1. x >>= return  ==  x
2. return a >>= f  ==  f a
3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Often, the following law is also given which links monads to functors:

```
fmap f xs  ==  xs >>= return . f
```

Note that `==` in this case does not refer to the `==` function of the Eq typeclass, but to equality in the sense that both functions should give the same output on the same input.

3 Examples of Monads

3.1 Identity

The first example of a monad described here is that of a trivial monad that can be found in the *mtl* package [3].
If `f::a->b`, then $\star(f)$ for the `Identity` monad maps `Identity a` to `f a` The m

function is the type constructor. Return is equally simple: we set $return_a$ to be `Identity::(a->Identity a)`. From this, it follows that

```
return x = Identity x
Identity x >>= f = f x
```

The reader may verify that the monad laws hold for this trivial example.

3.2 Maybe

The Maybe monad is used for combining functions that can return some kind of "failure" result (called `Nothing`), with the intent that once a failure has occurred in a chain of such functions, the whole result should be `Nothing`. This article assumes that the reader is familiar with the definition of the `Maybe a` type. The monad instance declaration is fairly inituitive: we first imagine that we are only dealing with failureless functions (i.e those that do not return `Nothing`). Then we can inherit the definition of the Identity monad, but using the `Just` constructor instead of `Identity` for `return`. As the definition of `return` has thus been established, the only thing that still needs to be set is how `>>=` acts on `Nothing` values. Clearly, `Nothing >>= f = Nothing` – once we have some sort of "failure" result, this is not changed by trying to apply a function that acts on non-failure types. The monad instance declaration of Maybe hence consists of

```
return x = Just x
Nothing >>= f = Nothing
Just x >>= f  = f x
```

To verify that the monad laws hold for this example, it needs to be shown that

1. `x >>= return == x`
2. `return a >>= f == f a`
3. `(x >>= f) >>= g == x >>= (\y-> f y >>= g)`

The first law clearly holds, as can be shown by making a case distinction on `x`. If `x` is `Nothing`, then (by definition of `>>=`), `x >>= return` is also `Nothing`. The alternative is that `x` is of the form `Just a`. But then `x >>= return` is `return a` which is `Just a` (by definition of `return`).
The second law follows from the fact that `return a >>= f` is `Just a >>= f` which is by definition `f a`, proving that the second monad law holds.
The third law can also be proven by making a case distinction on `x`. If `x` is `Nothing`, then clearly both sides of the equation are `Nothing`. If `x` is of the form `Just a`, then we know that `(x >>= f)` is `f a`, meaning the left hand side can be re-written to `f a >>= g`. In the same way, the right side of the equation becomes `(\y -> f y >>= g) a`. Application results in (`f a >>= g`), proving that the third law holds.
An example for usage of the `Maybe` monad is a "safe" divide 12 by `x` function. If `x` is not 0, then the function should return `Just (div 12 x)`, otherwise it should return `Nothing`. In other words, `f x = if x == 0 then Nothing else Just(div 12 x)`. If we want to chain several of these function and apply them to a value, we could do:

```
return 2 >>= f >>= f >>= f
```

This would result in the value of `Just 6`. But for the following

```
return 13 >>= f >>= f >>= f
```

the result is `Nothing`, as the first application of `f` returns `Nothing`, meaning that the other `f`s do not affect the result.

3.3 List

The List monad is used to model composition of non-deterministic functions. In this context, the term "non-determinism" is used to refer to functions that return a list of results, which can be interpreted as the set of possible return values for a function [8, p. 233]. Composing two non-deterministic functions should therefore result in a function that returns a list consisting of the concatenation of all lists returned by the second function when applied to each of the results of the first function. More formally, if f :: a->[b] then $\star(f)$ :: [a]->[b], so that $\star(f)\ xs\ =$ `concat` (`map` $f\ xs$), which results in the definition of the `>>=`:

```
xs >>= f =  concat (map f xs)
```

The definition of *return* is so that *return* yields an element in a minimal List context, i.e.

```
return a = [a]
```

3.4 Functions as Monads

In Haskell, the type `(->) r` is a Monad. The relevant instance declaration is [8, p. 311]:

```
instance Monad ((->) r) where
        return x = \_ -> x
        h >>= f = \w -> f (h w) w
```

This Monad is also refered to as the *reader monad*, as the functions that are combined with this monad "read from a common source"[8, p.312]. To illustrate this, consider the following to functions as an example:

```
f = do
       x <- (+5)
       y <- (/2)
       return (x * y)
g x = (x + 5) * (x / 2)
```

Note that `f` and `g` describe the same function. The reader monad makes it possible to combine functions that are missing one parameter in a way that the resulting function passes its parameter as the parameter to each of the functions. This can be interpreted as combining functions "with context", where the context is the missing parameter[8, p. 312]

4 Do Notation

Due to the fact that certain indiomatic constructs often come up when programming with monads, Haskell provides the so called "do notation" as a syntactic construct specifically for working with monads. Do notation makes it simple to write functions consisting of nested structures of `>>=` with a lambda-expression as the second argument [12, Chapter 14]. For example, the following function could be used to calculate the cartesian product of two lists[3]:

```
cartesian_product xs ys = xs >>= (
                \x ->
                        (ys >>=
                        \y-> return (x,y)
                        )
                )
```

When written in do notation, each lambda term of the form `zs >>= (\z -> f(z))` is written as `z <- zs; f(z)` [15], where the semi-colon can also be replaced by a newline. Note that the expression `f(z)` is used here to denote an arbitary expression that may depend on `z`. The keyword `do` is used to mark the start of a block of do notation. Applying this rule to the `cartesian_product` function above leads to the following program, which is far shorter and also easier to read:

```
cartesian_product' xs ys = do
                x <- xs
                y <- ys
                return (x,y)
```

What is also interesting about this way to write the product function is the fact that using `return` instead of the `[]` type constructor results in a function that is not specific to just lists but that can be used on other monadic types. For example, we can use this product on `Maybe` types to obtain a function that gives back `Nothing` if either of the parameters is `Nothing`, but otherwise returns a pair of values wrapped in a `Just` constructor.
It should be noted that if an expression that was converted to do notation used `>>`, then expanding this using the default definition of `>>` can result in a line that does not depend on any of the other parameters. However, it is clear that such a line can still affect the output of the function, as can be seen in the following example of a function that returns the empty list:

```
empty_list xs ys = do
                x <- xs
                y <- ys
                []
                return (x,y)
```

Another way to write a function like this would be

```
empty_list' xs ys = do
                x <- xs
                y <- ys
```

[3]Recall that `return (x,y)` is the same as `[(x,y)]` for the list monad

```
        z <- []
        return (x,y)
```

This highlights the fact that "do notation" is *not* a way to write an imperative program in Haskell, but rather a way to write programs that make use of monads. Another example of do notation would be the following function, which divides the input `x` first by `y` and the result of that then by `z`, but returns `Nothing` if any divisor was zero. Note also the syntax for using `let` within do notation.

```
safediv x y = if y == 0 then Nothing else Just(div x y)
safedividetwice x y z = do
        let somename = x
        u <- safediv somename y
        q <- safediv u z
        Just q
```

5 Monadic Combinators

Haskell provides a wide range of functions for dealing with monads. Some (but not all) will briefly be introduced here, due to the fact that, like do notation, they make it easier to write succint monadic code. These functions can be found in the `Control.Monad` package of `base` [1].

5.1 liftM

The functions of the form `liftM`, `liftM2`, `liftM3`,... are used to *"Promote a function to a monad, scanning the monadic arguments from left to right"* [1]. `liftM` is the same as `fmap`[17]. `liftM2` is a generalisation of the `cartesian_product` function shown in the section above, its implementation is [2]:

```
liftM2 f m1 m2 = do
              x1 <- m1
              x2 <- m2
              return (f x1 x2)
```

The `cartesian_product` function described above can be rewritten to `product = liftM2 (:)`. The functions `liftM3`, `liftM4`, ... `liftM5` extend this to functions with more parameters.

5.2 sequence

The `sequence` function has the type `Monad m => [m a] -> m [a]`, which indicates that it turns a list of monadic values into a monadic value over a list. For example, `sequence [Just 1,Just 2,Just 3]` returns `Just [1,2,3]`, but `sequence [Just 1,Just 2,Just 3,Nothing]` gives `Nothing`. Its implementation is:

```
sequence [] = return []
sequence (m:ms) = do {x<-m; xs <- sequence ms; return (x:xs)}
```

6 The IO Monad

The `IO` monad makes it possible to combine actions that deal with input and output. Monads were only introduced for IO in 1996 [6], the other models for IO considered before that being *Streams* and *Continuations* [6], but it was monadic IO that was finally included in Haskell. `IO` can be used like any other monad. The type `IO a` can be viewed as a "computation", that when run, produces a value of the type `a`, and possibly has side-effects on the outside world [6]. Using `>>=` (or the equivalent do notation), makes it possible to sequence these computations in a way that allows the order in which they are performedi to be specified. This is important for IO, because unlike with regular functions, we do not want the compiler to reorder IO actions like it could with functions that do not have effects [5]. Due to the type signature of the `>>=` function, the type system also clearly separates those sections of a program that result in an IO action from the pure sections of the program[4]. A simple example of an IO program is one that reads in a line of text, and then outputs this line. Haskell provides the `getLine :: IO String` function for reading a line from standard input, and the `putStrLn :: String -> IO ()` for writing a line to standard output. Using monads, the two functions can be combined by writing `getLine >>= putStrLn`, giving the desired program. To convert the string to uppercase in between, `getLine >>= (\x -> putStrLn (map toUpper x))`. Using do notation makes this more readable:

```
do
    x <- getLine
    putStrLn $ map toUpper x
```

Due to the outward appearance of do notation, this program could be mistaken for an imperative program. However, it should be stressed that this is *not* the case, rather `IO` is simply an instance of `Monad`. Highlighting this is the fact that the monadic combinators described above can be used on `IO` like on any other monad. For example, `sequence $ replicate 5 getLine` creates an `IO` action that reads in 5 lines from standard input, and gives back a list of `String` (wrapped in `IO`) containing these lines.

7 Conclusion

In Conclusion, it can be seen that monads can be used in a variety of settings, ranging from nondeterminism to IO. Using Haskells do notation and monad combinators makes it possible to write programs that make use of the context provided by a monad, allowing the programmer to not have to explicitly write that which the monadic context does, making programs more readable. Monads can also be combined, though this is has not been addressed in this article. For example, *monad transformers* can be used to define monads that combine the features of two monads into one[16]. An alternative to using monad transformers that is less restrictive when dealing with the interaction of effects is the use of *extensible effects*[7].

[4]This excludes 'unsafe' functions such as `unsafePerformIO`

References

[1] Control.Monad. https://archive.today/X6pLI. Accessed: 2014-11-30.

[2] Control/Monad.hs. https://archive.today/Wh3lA. Accessed: 2014-11-30.

[3] Control.Monad.Identity. https://archive.today/kxW0Y. Accessed: 2014-11-29.

[4] Hask-HaskellWiki. https://www.haskell.org/haskellwiki/index.php?title=Hask&oldid=52908. Accessed: 2014-11-05.

[5] IO inside-HaskellWiki. https://www.haskell.org/haskellwiki/index.php?title=IO_inside&oldid=55993. Accessed: 2014-11-30.

[6] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy With Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007.

[7] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70. ACM, 2013.

[8] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011.

[9] Charles Wells Michael Barr. *Category Theory for Computing Science*. Prentice Hall International, 2 edition, 1995.

[10] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[11] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93:55–92, 1991.

[12] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. Real world haskell: Code you can believe in. http://book.realworldhaskell.org, 2008.

[13] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[14] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[15] Wikibooks. Haskell/do notation — wikibooks, the free textbook project. http://en.wikibooks.org/w/index.php?title=Haskell/do_Notation&oldid=2657865, 2014. [Online; accessed 29-November-2014].

[16] Wikibooks. Haskell/monad transformers — wikibooks, the free textbook project. http://en.wikibooks.org/w/index.php?title=Haskell/Monad_transformers&oldid=2695286, 2014. [Online; accessed 2-December-2014].

[17] Brent Yorgey. The typeclassopedia. *The Monad. Reader Issue 13*, page 17, 2009.