UPPSALA
UNIVERSITET

IT 19 050
Examensarbete 15 hp
September 2019

Monads in Haskell and Category Theory

Samuel Grahn

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Monads in Haskell and Category Theory

Samuel Grahn

The monad is a mathematical concept, used by Haskell to describe — among other things — Input/Output. Many are intimidated by it since it stems from abstract mathematics — namely Category Theory. However, the mathematics required to use and understand the monad is straight forward and intuitive, and can be explained through incremental definitions and proofs. This paper intends to construct and explain the monad from the ground up and show some example uses for it.

Handledare: Justin Pearson
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Johannes Borgström
IT 19 050
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

1.1 Motivation

Through the development of computer science, mathematics has been a very important tool. From the basic logical circuits the transistor enabled to the computability, complexity and formalization of the Turing Machine. Abstractions — through mathematics — have moved the knowledge requirements for writing software from the hardware level with electronics and circuits, to machine language, to assembly, and on towards higher level languages.

One useful class of abstractions are those of the functional languages; languages in which functions are treated as first-class objects. As a way to eliminate unpredictable behaviour, functional languages generally disallow *mutation*, changing the value of a variable. This has the effect that a function will always return the same value, when passed the same arguments, preventing small issues from creating a butterfly effect at unrelated parts of the program.

There is one problem that needs to be solved however, and that is the issue of Input/Output.

If no function is allowed to alter the global state — as this would cause the next function call to have different effects, how is a program supposed to output data to the console? How to read a file, whose contents might change between reads?

One answer to this, the answer chosen by Haskell[4], is to abstract the changes away; to wrap them in a separate object, an object that keeps track of the outside world. This object is called the *IO monad*[4, Chapter 41], and while the category theoretic concept of a monad[1, p.137] has many other uses than IO[4, p.156], it is a good motivation for exploring it. This thesis aims to provide the reader with an understanding of the monad and its prerequisite category theoretic concepts, both of the mathematical objects and of the practical uses in functional programming.

1.2 Method

In order to explain the monad, we need some previous knowledge. The monad is a *monoid in the category of endofunctors*. While this sounds vague and complicated, it is just a description of how and what it consists of. After explaining each of these concepts in turn, the monad will be a simple extension.

We will begin by stating any prerequisite knowledge, closely followed by a incremental construction of category theory, along with code that shows its practical uses.

The first thing we need to know, is our programming language of choice; Haskell.

2 Haskell

Haskell is a pure, functional language which makes heavy use of monads and other abstractions from category theory. It has a lot of features, but we will only be using a select few in order to keep the code examples as close to our mathematical notation as possible.

```
-- Comments begin with two dashes

-- Declare the type of a variable
a :: Int
-- And its value
a = 1

-- Or a function
double :: Int -> Int
-- And its definition
double x = x * 2

-- Functions can be passed as arguments
apply :: (Int -> Int) -> Int -> Int
-- apply takes a function f, and an int x, and applies f to x.
apply f x = f x
-- or equivalently using lambda notation
apply = \f x -> f x

-- When the program is run, it prints the number 2
main = print (apply double a)
```

We will be using some syntactic sugar to help make code more readable.

1. `($) :: (a -> b) -> a -> b`, can be used instead of parentheses to specify argument locations. For example, the expression `f $ x + y` evaluates to the same result as `f (x + y)`.

2. `\x -> x + 2`, an anonymous function. The function itself is equivalent to a function f defined as `f x = x + 2`, but it is nameless and is sometimes easier to express.

3. `[a]`, shorthand for the type of list with elements of type a, equivalent to `[] a`

We will be using two more features of Haskell, namely custom types and classes.

A type is declared by stating its name and its possible values, each of which may have parameters.

A *class*, not to be confused with classes from object oriented programming, is a promise that a type supports a set of functions; in order to implement a class for a type, it is required to provide implementations for each of the

functions the class promises to deliver. Some of these functions can have default implementations. These implementations may use any of the constraints placed upon the class itself, as well as the other functions within the class, but no other assumptions can be made about the types.

```
-- Simple datatypes without parameters
data Language = Swedish | English
data Animal = Cat | Cow

-- Both of the above datatypes can be viewed as social in some way,
-- so we define a common behaviour
class Social a where
  greet :: a -> String

-- And implement this behaviour
instance Social Language where
  greet Swedish = "Hej"
  greet English = "Hello"

instance Social Animal where
  greet Cat = "Meow"
  greet Cow = "Moo"

-- A function can take general types as parameters (a -> b),
-- and can impose restrictions on these types (Social a).
interaction :: (Social a, Social b) => a -> b -> String
interaction a b =
  (greet a) ++ (greet b)

-- This simply prints "MooHello" into the console
main = print (interaction Cow English)
```

These classes are used to model mathematical objects. The most simple of which - the *monoid*, is our first building block.

3 Monoid

Mathematically speaking, a monoid is a simple construction, consisting only of a set of objects C, and a way to combine two objects of C into an object of C; the operator $\cdot : C \times C \to C$, subject to two rules [3, p.3];

1. Associativity: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
2. Identity: There exists a $1_{\mathbf{C}} \in C$ such that $1_{\mathbf{C}} \cdot x = x = x \cdot 1_{\mathbf{C}}$, for all $x \in C$.

These rules are of course still present in Haskell, albeit not enforced or checked by the compiler. All built-in instances of monoids satisfy these condi-

tions, and any user-implemented instances are encouraged to satisfy them as well, simply for mathematical consistency.

In Haskell, the operation $(\cdot)$ on a monoid is `mappend` (monoid-append), which is named after the list monoid. The identity element, $1_{\mathbf{C}}$, the element with which every other element simply composes into itself is also named after the list monoid, `mempty` (monoid-empty). More abstract examples include functions `a -> a`, with function composition as operation and the identity function `id` as identity.

The typeclass `Monoid` is a subclass of `Semigroup`, meaning each monoid is also a semigroup. This means that to instantiate a monoid, you first need to instantiate a semigroup. The difference between the two is that in a semigroup there is no requirement of identity, and the combination operator is the operator `<>`. Note that `mappend` has a default implementation `mappend = <>`. Trivial examples can be implemented as follows

```
-- The nonnegative integers
-- represented as a list data structure
-- | Zero is 0, Seq (a) is the number after a.
-- | Seq (Seq Zero) = 2
data PosInt = Zero | Seq (PosInt)

instance Semigroup PosInt where
  Zero <> x = x
  y <> Zero = y
  (Seq x) <> Seq y = Seq (Seq (x <> y))

-- The nonnegative integers form a monoid with addition
instance Monoid PosInt where
  mempty = Zero


-- The booleans with the or-operator form a monoid
instance Semigroup Bool where
  False <> False = False
  _ <> _ = True

instance Monoid Bool where
  mempty = False
```

The monoid is a useful construction, and its structure often reappears in mathematical objects. It is, however, an important structure that will reappear as we delve into *Category Theory*.

4 Categories

Categories are the basic construct of category theory. The mathematical object upon which all others are based.

Formally, a category **C** consists of [3, p.582]

1. a collection of *objects*. We use a collection instead of the more traditional set, in order to effectively allow the *Category of Categories*. For the purpose of this thesis, however, it is sufficient to consider them roughly equivalent to sets.

2. a collection of *morphisms*, also known as *arrows*, between objects. Each morphism has a source and a target object, which for a morphism f we write as $f : a \to b$. Further, by $hom(a, b)$ we mean the collection of all morphisms from a to b.

3. a binary operation $\circ : hom(a, b) \times hom(b, c) \to hom(a, c)$, for all objects a, b, c, called composition. Further, this operation requires that the following axioms hold

   (a) Associativity; if $f : a \to b$, $g : b \to c$, $h : c \to d$
   then $h \circ (g \circ f) = (h \circ g) \circ f$.

   (b) Identity: For every object x there exists a morphism $1_x : x \to x$ such that composition with another morphism simply yields that other morphism.

4.1 Commutative Diagrams

The equations and laws of category theory often get complex and difficult to understand when using only the standard mathematical notation. Therefore, category theorists developed a different way to represent categories and their relations – using commutative diagrams. In fact, through rigorous definition of diagrams, one could define categories using diagrams only.

For instance, a category **C** with objects A, B and morphisms $f : A \to B$, $g : B \to A$ can be drawn as

$$A \underset{g}{\overset{f}{\rightleftarrows}} B$$

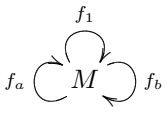
The identity morphisms are typically not drawn, since they always exist, and would mainly contribute clutter to the diagram. Relations on categories can also be represented as diagrams.

4.2 Monoid as a Category

Given any monoid M we can consider the following functions.

$$f_a : M \to M,\ f_a(x) = a \cdot x$$

These functions can be considered morphisms of a single object category $\mathbf{C}_M$. We can inspect the axioms for a category, and state that $f_1(x) = 1_M \cdot x$ is the identity morphism for the single object of $\mathbf{C}_M$, and since the monoid operation is accosiative we have $f_a \circ (f_b \circ f_c) = a \cdot (b \cdot c) = (a \cdot b) \cdot c = (f_a \circ f_b) \circ f_c$, so $\mathbf{C}_M$ is indeed a category, as represented by the following diagram.



Conversely, given any category $\mathbf{C}$ with only one object x, we can construct a monoid M by letting each morphism $f : x \to x$ map to an object f in M, and composition of morphisms be represented as $f \circ g = f \cdot g$. Thus, any monoid is a Category, and any category with a single object is a monoid (with morphisms representing the elements of the monoid).

4.3 The Category Hask

Now that we know what a category is, we can start connecting the dots to Haskell. The most interesting category in our case is the category **Hask**, the base of Haskell's type system.

In **Hask**, the objects are the types of Haskell (Int, Double, et.c.), and the morphisms are Haskell functions $f : A \to B$, where A and B are Haskell types. The composition of two morphisms is the operator (.) in Haskell, defined as[4, Chapter 9]

```
(.) :: (B -> C) -> (A -> B) -> (A -> C)
f . g = \x -> f (g x)
```

In **Hask**, two functions, f, g are considered to be the same morphism if for all x, $f(x) = g(x)$, despite them being different functions in Haskell. The identity morphism in **Hask** is the function[4, Chapter 9]

```
id :: A -> A
id x = x
```

4.4 Subcategories

If you have a category and disregard some of its objects, and all morphisms either to or from these objects, you have a subcategory. For instance, the right category is a subcategory of the left category in the following diagrams.

In **Hask**, the subcategories are commonly viewed as the types wrapped into data types. For instance, `[a]`, the type of the linked list, is a subcategory of Hask. The elements within that list follow the same structure as if they were not in a list, while the list itself provides an additional structure.

In order to actually use the concept of categories in Haskell, we introduce the simplest operation upon them: the functor.

5 Functors

Functors are the next building block of category theory. It is a mapping between two categories; a way to turn one category into another.

When talking about functors in Haskell, however, one usually refers to some kind of container. This is a good way to get a basic understanding for some of its use cases.

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

-- The functor instance for list
instance Functor [a] where
  fmap f [] = []
  fmap f (x : xs) = (f x) : (fmap xs)

-- Or equivalently
instance Functor [a] where
  fmap = map

data Maybe a = Just a | Nothing
-- The functor instance for Maybe
instance Functor Maybe a where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

If we look closer at these definitions, we can see the notion of containers quite clearly. The type `Maybe a`, for instance, which may or may not contain a value of type a. Regarding functors as containers in Haskell allows you to use most of them without any further knowledge of category theory. However, the functor is more abstract than just containerization, and can apply to many other use cases. Those, however, require more mathematics.

5.1 Functors in Category Theory

Formally [3, p.586], a functor $F : \mathbf{C} \to \mathbf{D}$ is a mapping from the category $\mathbf{C}$, to the category $\mathbf{D}$. It associates each object $X \in \mathbf{C}$ to an object $F(X) \in \mathbf{D}$, and

each morphism $\phi : X \to Y$ in $\mathbf{C}$ to a morphism $F(\phi) : F(X) \to F(Y)$ in $\mathbf{D}$.

$$\begin{array}{ccc} X & \xrightarrow{F} & F(X) \\ {\scriptstyle\phi}\downarrow & & \downarrow{\scriptstyle F(\phi)} \\ Y & \xrightarrow[F]{} & F(Y) \end{array}$$

Subject to two conditions:

1. $F(1_X) = 1_{F(X)}$ for every X in $\mathbf{C}$.
2. $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$, $g : Y \to Z$.

$$\begin{array}{ccccc} X & \xrightarrow{f} & Y & \xrightarrow{g} & Z \\ \downarrow{\scriptstyle F} & & \downarrow{\scriptstyle F} & & \downarrow{\scriptstyle F} \\ F(X) & \xrightarrow[F(f)]{} & F(Y) & \xrightarrow[F(g)]{} & F(Z) \end{array}$$

So a functor preserves identity morphisms and composition of morphisms.

5.2 Functors in Haskell

When referring to functors in Haskell, what is usually meant is actually only a subset of possible functors, the functors from a category unto itself, the *endofunctors*. In **Hask**, the endofunctors are represented as a typeclass with a single function, fmap, which takes a function `a -> b`, and lifts it into a function `f a -> f b`, that performs that same computation on the wrapped objects. We frequently use the infix operator `<$>`, which is a synonym to `fmap`, defined as `f <$> x = fmap f x`.

5.2.1 Functor Laws in Haskell

Implementing each function in the Functor typeclass is not enough to be a proper mathematical functor. Satisfying the laws of the functor object is the task of the programmer when implementing `fmap`. The first law — preserving of identity — is equivalent to the statement

```
fmap id = id
```

The second law, preserving of composition, is equivalent to the statement

```
fmap (g . h) = (fmap g) . (fmap h)
```

5.3 List is a Functor

The built-in functors of Haskell satisfy these laws. As an example, we can view the functor instance for lists, and note that

```
  fmap id (x:xs)
= (id x) : fmap xs
= x : fmap xs
```

Which inductively gives us `fmap id (x:xs) = (x:xs)`, satisfying the first law.

We can also see that

```
  (fmap g) . (fmap h) (x : xs)
= fmap g (fmap h (x : xs))
= fmap g ((h x) : fmap h xs)
= (g . h x) : fmap g (fmap h xs)
```

which inductively gives us the second law, meaning the list functor is indeed a functor. All instances of Functor in Haskells' standard library satisfy these laws [4].

6 Applicative Functors

Haskell consists of several data types that have more in common than just being functors, but that don't quite match all requirements for being a monad. For this reason, Haskell has another subclass between the functor and the monad: the applicative functor. An applicative functor is the last stepping stone in our construction of the monad. Its type class has three functions

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c

  (<*>) = liftA2 id
  liftA2 g x y = g <$> x <*> y
```

where the default implementations of `(<*>)` and `liftA2` means only one of them needs manual implementation, along with the implementation of `pure`.

From the types we can see that `pure` takes an arbitrary value, and turns it into an applicative — and into a functor, since each applicative is also a functor — containing this value. The other functions, `(<*>)` and `liftA2`, are about sequencing.

To make the uses more clear, assume you have a function `foo :: a -> b -> c`, and variables `x :: f a` and `y :: f b`, where `f` is an applicative functor. You

can supply `foo` with the contents of `x,y` through the following two equivalent expressions.

```
pure foo <*> x <*> y
liftA2 f x y
```

The most trivial example is that of the Maybe Applicative, where we from

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _ <*> _ = Nothing

foo x y = x + y
a = Just 1
b = Just 2

c = pure foo <*> a <*> b
```

get the result `c = Just 3`. Another example is the list applicative [4, Chapter 9].

```
instance Applicative [a] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

This implementation, is using list comprehension to make it easier to read. For every `f` in `fs`, and every `x` in `xs`, the list consists of the values of `f x`. This means that for every value in the list `xs`, each function in the list `fs` is applied. For instance `[(+1), (+2)] <*> [5,10] = [6,11, 7, 12]`. While the applicative typeclass have no significant purpose in category theory, it is useful as an abstraction in Haskell. Using this final subclass of the monad, we can finally approach the monad in Haskell.

7 Monads in Haskell

The monad instance is defined as follows [4, Chapter 9]

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

   m >> k = m >>= \_ -> k

   return      :: a -> m a
   return      = pure
```

The primary use case of the monad is to emulate sequential actions, much in the same way that semicolons operate in C-like languages. This chaining of

operations, wrapped in a separate functor container, gives us the tools to deal with many of the issues functional programming provides.

7.1 Do-notation

In order to simplify the syntax for sequencing monadic operations, Haskell provides what is called *do-notation*. This is a compiler feature, replacing matching syntax with the monadic operations. Using this syntax, the following function definitions of `doPutStr` and `putStr` are equivalent, as are the definitions of `doBindop` and `bindOp`.

```
doPutStr = do {
  putStr "Hello "
  putStr "World"
  }

putStr = putStr "Hello " >> putStr "World"

doBindOp = do
  s1 <- getLine
  s2 <- getLine
  putStr (s1 ++ s2)

bindOp = getStrLn >>= (\s1 -> getStrLn >>= (\s2 -> putStr (s1 ++ s2)))
```

7.2 Maybe

Using the Maybe monad we can deal with error handling, and other types of functions that not always have a value to return [4, Chapter 9].

```
instance Monad Maybe where
  return = pure
  (Just x) >>= f = f x
  Nothing >>= f = Nothing
```

Some examples of its usefulness are

```
-- Prevents runtime errors when dividing by zero
(/.) :: Integral a => a -> a -> Maybe a
x /. y = if y == 0 then Nothing else Just $ x `div` y

-- Examples
a = (3 + 2) /. 1
-- a = Just 5

b = (3 + 2) /. 0
-- b = Nothing
```

```
-- Get element at nth position of a list.
-- If it's not there, return Nothing
at :: [a] -> Int -> Maybe a
at (x : xs) 0 = Just x
at [] _ = Nothing
at (_ : xs) n = at xs (n-1)
```

7.3 State

One of the useful applications for monads is the State monad. It emulates the notion of a state by having a hidden variable wrapped in a monad. When this monad is used to call functions, this state can change, and the results of the functions can depend upon it.

In the Haskell library, the State monad is defined as

```
newtype StateT s m a = StateT { runState :: s ->  m (a,s) }

newtype State s = StateT s Identity
```

For our purposes, however, we will ignore the abstraction of StateT, and define[2]

```
newtype State s = State { runState :: s -> (a,s) }
```

We can then instantiate the state monad from the ground up as

```
instance Functor (State s) where
  fmap f s = State $ \w ->
    let (na, ns) = runState s w
    in (f na, ns)

instance Applicative (State s) where
  pure x = State $ \s -> (x,s)
  sf <*> sv = State $ \w ->
    let
      (v, s1) = runState sv w
      (f, s2) = runState sf s1
    in (f v, s2)

instance Monad (State s) where
  return = pure

  p >>= k = State $ \s0 ->
    let (x, s1) = runState p s0
    in runState (k x) s1
```

The state monad is accompanied by some helper functions that enable for changing the state itself, namely using `put` and `get`.

```
get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put x = State $ \s -> ((), x)

evalState :: State s a -> s -> a
evalState a = fst . runState a

execState :: State s a -> s -> s
execState a = snd . runState a
```

`get` leaves the state unchanged and sets its value to the state itself, and `put` replaces the state value, leaving the value as the unit. `evalState` and `execState` are used to evaluate a sequence of state operations and return the value or the state itself, respectively.

7.3.1 State-based Game

An example use is the following trivial game.

```
playGame :: String -> State (Int, Int) Int
-- Plays a game using a string
  -- + adds 1 to value
  -- - subtracts 1 from value
  -- x stores value into memory
  -- * multiplies value by the value in memory

-- The goal is to provide a string computing
-- a certain value according to some constraint.
-- (Example: shortest string computing the number 42)


playGame [] = do
  (_, score) <- get
  return score

playGame (x:xs) = do
  (mem, score) <- get
  case x of
    '+' -> put (mem, score +1)
    '-' -> put (mem, score -1)
    '*' -> put (mem, score * mem)
    'x' -> put (score, score)
  playGame xs
```

The state monad is used as a container for the current game state; the value in memory and the current score. At each step, the game checks the next character, changes its state accordingly, and proceeds to the next character.

7.3.2 Tree Relabeling

Another example is that of relabeling the nodes of a (in this example, binary) tree.

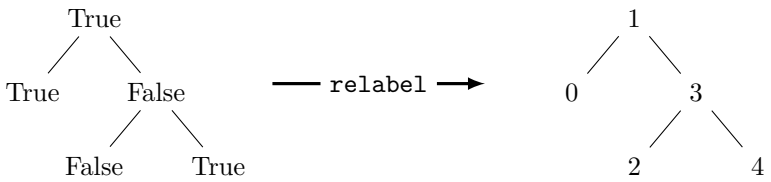
```
import Control.Monad.Trans.State

data Tree a = Empty | Node a (Tree a) (Tree a) deriving Show

relabel::Tree a -> Tree Int
relabel t = evalState (relabel1 t) 0

relabel1:: Tree a -> State Int (Tree Int)
relabel1 Empty = do return Empty
relabel1 (Node _ l r) =
  do l1 <- relabel1 l
     label <- get
     put (label + 1)
     r1 <- relabel1 r
     return (Node label l1 r1)
```

The function `relabel1` takes a tree, and traverses it, starting by descending the left branch, followed by the current node and finally the right branch at each node, giving each node the value of the current number of nodes already traversed. Given a binary search tree, for instance, the function assigns to each node the position of itself in an ordered list, i.e. how many nodes are *smaller* than it.



Note that the values of the nodes in the source tree does not matter, only the structure of the tree does.

If we look at the function `relabel1`, we can see that it traverses the tree depth first, left to right.

7.3.3 Unfolding Tree Relabeling

The functionality the relabling example provides can of course be implemented without monads as well. For instance, the following program is (computationally) equivalent

```
relabel' t = fst . relabel1' t 0

relabel1' :: Tree a -> Int -> (Tree a, Int)
relabel1' Empty n = (Empty, n)
relabel1' (Node _ l r) n =
  let
    (l1, n1) = relabel1' l n
    label = n1
    n2 = label + 1
    (r1, n3) = relabel1' r n2
  in
    (Node label l1 r1 n3)
```

We can show this by considering the base case and the recursive case indepently. For the base case, we have

```
relabel' Empty = fst . relabel1' Empty 0
               = fst . (Empty, 0)
               = Empty
```

and

```
relabel Empty = evalState (relabel1 Empty) 0
              = fst . runState (relabel1 Empty) 0
              = fst . runState (return Empty) 0
              = fst . runState (State $ \s -> (Empty, s)) 0
              = fst ((\s -> (Empty, s)) 0)
              = fst (Empty, 0)
              = Empty
```

The recursive case can be done similarly, but is left as an exercise to the reader. This conversion of monadic programs to regular programs can help convince us that the monad is not magic.

8 Monads in Category Theory

The basic understanding of monads as a tool in Haskell can be further evolved by exploring the category theory behind it. To do this, we need to understand the concept of *natural transformations*.

8.1 Natural Tranformations

Given two functors, F, G between categories $\mathbf{C}, \mathbf{D}$, a natural transformation $\psi : F \to G$ is a collection of morphisms in $\mathbf{D}$ [3, p.417]. The morphisms must satisfy the following conditions

1. For every object x in $\mathbf{C}$, there is a morphism $\psi_x : F(x) \to G(x)$, called the component of ψ at x.
2. Components must satisfy that for every morhpism $f : X \to Y$ in $\mathbf{C}$, we have $\psi_y \circ F(f) = G(f) \circ \psi_x$.

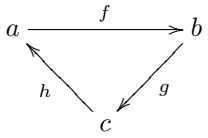
$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \psi_x}\downarrow & & \downarrow{\scriptstyle \psi_y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

That is, a natural transformation can be seen as a way to transform one functor into another. Further, if each ψ_x is bijective, it is said that ψ is a natural isomorphism, and that F and G are isomorphic. Note that while the notation hints at ψ being a function between F and G, F and G are not sets, so the notation simply means that ψ transforms a functor F to another functor G.

8.1.1 Example

In order to make the concept of natural transformations more clear, we will construct a simple one.

Let $\mathbf{C}$ be the category defined by the diagram



Let F be a functor $F : \mathbf{C} \to \mathbf{C}$ with $F(a) = F(b) = F(c) = a$ and $F(f) = F(g) = F(h) = 1_a$, and let I be the identity functor on $\mathbf{C}$ that maps each object to itself, and each morphism to itself.

Let $\varphi : I \to F$ be a natural transformation with components

$$\begin{aligned} \varphi_a &= h \circ g \circ f = 1_a \\ \varphi_b &= h \circ g \\ \varphi_c &= h \end{aligned}$$

We note that every $x \in \mathbf{C}$ there is a morphism φ_x, so the first law is satisfied. Next, we examine the morphisms, and show that

$$\begin{aligned}\varphi_b \circ I(f) = h \circ g \circ f = (h \circ g \circ f) \circ 1_a = F(f) \circ \varphi_a \\ \varphi_c \circ I(g) = h \circ g = 1_a \circ (h \circ g) = F(g) \circ \varphi b \\ \varphi_a \circ I(h) = h \circ f \circ g \circ h = F(h) \circ \varphi_c\end{aligned}$$

So F is a natural transformation, transforming $\mathbf{C}$ into the subcategory

8.2 Composition of Natural Transformations

If $\alpha : F \to G$ and $\beta : G \to H$ are natural transformations between functors $F, G, H : \mathbf{C} \to \mathbf{D}$, we can construct a natural transformation $\beta \circ \alpha : F \to H$, by componentwise defining $(\beta \circ \alpha)_x = \beta_x \alpha_x$. Next, given a natural transformation $\xi : F \to G$ between functors $F, G : \mathbf{C} \to \mathbf{D}$, and functors $H_1 : \mathbf{B} \to \mathbf{C}$, $H_2 : \mathbf{D} \to \mathbf{E}$, we can define natural transformations $H_1\xi : H_1F \to H_1G$ and $\xi H_2 : FH_2 \to GH_2$ as

$$\begin{aligned}(H_1\xi)_x = H_1\xi_x \\ (\xi H_2)_x = \xi_{H_2(x)}\end{aligned}$$

There is a third way to compose natural transformations which is irrelevant for our purposes, so we can ignore it.

These compositions of natural transformations will become useful in the following definition.

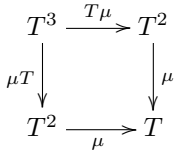
8.3 Monads

A monad on a category $\mathbf{C}$ consists of an endofunctor $T : \mathbf{C} \to \mathbf{C}$ together with two natural transformations [1, p. 137]:

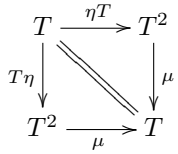
- $\eta : 1_{\mathbf{C}} \to T$, where $1_{\mathbf{C}}$ is the identity functor on C
- $\mu : T \circ T \to T$

Subject to the following criterion [1] [2]

1. $\mu \circ T\mu = \mu \circ \mu T$



2. $\mu \circ T\eta = \mu \circ \eta T = 1_T$, where 1_T is the identity morphism from T to T.



8.4 Power Set Monad

A simple example of a monad is the *power set monad*.

Let **Set** be the category of all sets, with morphisms as functions between them. Let $T : \mathbf{Set} \to \mathbf{Set}$ be an endofunctor with $T(A) = \{X \in \mathbf{Set} | X \subseteq A\}$ for objects A of **Set**, i.e. the power set of A, and for morphisms $f : A \to B$, $T(f) : T(A) \to T(B)$, for each subset A_i of A, let $T(f)(A_i) = \{f(x) \mid x \in A_i\}$, in other words, map each subset of $T(A)$ to its image in $T(B)$.

First, we need to prove that T is indeed a functor. The identity morphism on each object $A \in \mathbf{Set}$ is the function $1_A(x) = x$. For each A_i of $T(A)$ we get $T(1_A)(A_i) = \{1_A(x) \mid x \in A_i\} = \{A_i\}$, i.e. the identity morphism on A_i, so the first functor law is satisfied.

As for the second functor law, consider $f : A \to B$, $g : B \to C$. We have for each $A_i \in A$, that

$$\begin{aligned}
(T(g) \circ T(f))(A_i) &= T(g)(T(f)(A_i)) \\
&= T(g)(\{f(a) \in T(B) \mid a \in A_i\}) \\
&= \{g(b) \mid b \in \{f(a) \in T(B) \mid a \in A_i\}\} \\
&= \{g(f(a)) \mid a \in A_i\} \\
&= \{(g \circ f)(a) \mid a \in A_i\} \\
&= T(g \circ f)(A_i)
\end{aligned}$$

so the second functor law also holds, meaning T is indeed a functor.

Define the natural transformation $\eta : 1_{\mathbf{Set}} \to T$ with components for each $A \in \mathbf{Set}$, $\eta_A : A \to T(A)$ as $\eta_A(a) = \{a\}$. Note that for a morphism $f : A \to B$ in **Set**, we have, for any $a \in A$

$$\begin{aligned}
(\eta_B \circ 1_{\mathbf{Set}}(f))(a) &= \eta_B((1_{\mathbf{Set}}(f))(a)) \\
&= \eta_B(f(a)) \\
&= \{f(a)\}
\end{aligned}$$

and

$$\begin{aligned}
(T(f) \circ \eta_A)(a) &= T(f)(\eta_A(a)) \\
&= T(f)(\{a\}) \\
&= \{f(x) \mid x \in \{a\}\} \\
&= \{f(a)\}
\end{aligned}$$

so η is a natural transformation.

Define the natural transformation $\mu : T \circ T \to T$ componentwise as the function

$$\mu_A(A) = \bigcup A = \{a \in X \mid X \in A\}$$

In other words, let μ map a set of sets into their union. We have

$$\begin{aligned}(\mu_B \circ T(T(f)))(A) &= \mu_B(T(T(f(A)))) \\ &= \mu_B(T(\{f(a) \mid a \in A\})) \\ &= \mu_B(\{X \mid X \subseteq \{f(a) \mid a \in A\}\}) \\ &= \bigcup\{X \mid X \subseteq \{f(a) \mid a \in A\}\} \\ &= \{f(a) \mid a \in A\}\end{aligned}$$

and

$$\begin{aligned}(T(f) \circ \mu_A)(A) &= T(f)(\mu_A(A)) \\ &= T(f)(\bigcup A) \\ &= \{f(a) \mid a \in \bigcup A\} \\ &= \{f(a) \mid a \in A\}\end{aligned}$$

so μ is a natural transformation.

Now that we have an endofunctor T, along with two natural transformations η, μ, we only need to show the monad laws are satisfied.

As for the first law, denote $A = \{B_i\}$ with $B_i = \{C_i\}$ and check

$$\begin{aligned}(\mu \circ T\mu)(A) &= \mu(T(\mu)(A)) \\ &= \mu(\{\mu(A_i) \mid B_i \in A\}) \\ &= \mu(\{C_i \mid \exists B_i : C_i \in B_i \in A\}) \\ &= \{c \mid \exists C_i, B_i : c \in C_i \in B_i \in A\}\end{aligned}$$

and

$$\begin{aligned}(\mu \circ \mu T)(A) &= \mu(\mu(T(A))) \\ &= \mu(\mu(\{B \mid B \subseteq A\})) \\ &= \mu(\{C_i \mid \exists B_i : C_i \in B_i \in A\}) \\ &= \{c \mid \exists C_i, B_i : c \in C_i \in B_i \in A\}\end{aligned}$$

satisfying the first monad law.

We have

$$\begin{aligned}(\mu \circ T\eta)(A) &= \mu(T(\eta)(A)) \\ &= \mu(\{\eta(a) \mid a \in A\}) \\ &= \mu(\{\{a\} \mid a \in A\}) \\ &= \{x \mid \exists a \in A : x \in \{a\}\} \\ &= A\end{aligned}$$

and

$$\begin{aligned}(\mu \circ \eta T)(A) &= \mu(\eta(T(A))) \\ &= \mu(\eta(\{X \mid X \subseteq A\})) \\ &= \mu(\{\{X \mid X \subseteq A\}\}) \\ &= \{x \mid \exists X | x \in X \subseteq A\} \\ &= A\end{aligned}$$

We can now conclude that (T, η, μ) is a monad on **Set**.

8.5 Connection to Haskell

Since we have $\mu : T^2 \to T$, we can view this transformation at an object a as
`join :: T (T a) -> T a`
which in Haskell is defined as
`join :: (Monad m) =>m (m a) -> m a`
`join x = x >>= id`.
This means that a proper implementation of `>>=` gives us a proper implementaton of μ.

As for $\eta : 1_{\mathbf{C}} \to T$, we note that $1_{\mathbf{C}}(a) = a$, so we get $\eta_a : a \to Ta$, giving us the type of
`return :: a -> T a` from the monad typeclass

Note, however, that while both of the natural transformations are of the correct type, neither criterion is automatically satisfied. The programmer implementing `return` and `>>=` is responsible for making sure these criterion are satisfied. Not fulfilling these criterion will remove some of the mathematical consistency behind the monads, and possibly even render them useless. All monads that is implemented in Haskells standard library satisfy the criterion [4, Chapter 9].

8.5.1 Monad Laws in Haskell

In order to effectively discuss the monad laws for Monad instances, we need to reformulate the laws to make sense in Haskell. The laws Haskell uses are equivalent to the laws of category theory, but this is not obvious since they are

simplified and separated into several smaller pieces. The monad laws of Haskell are [4, Chapter 13]

$$\texttt{(return a >>= k)} = \texttt{k a} \tag{1}$$
$$\texttt{(m >>= return)} = \texttt{m} \tag{2}$$
$$\texttt{x >>= (return . f)} = \texttt{fmap f x} \tag{3}$$
$$\texttt{(m >>= (\x -> k x >>= h))} = \texttt{(m >>= k) >>= h} \tag{4}$$

In order to show that these laws are equivalent, we assume that these laws hold for some monad, and show that it follows that the laws from category theory hold as well.

We begin with the first law from category theory, $\mu_x T\mu_x = \mu_x \mu_{T(x)}$, or in Haskell terms

`join . (fmap join) = join . join . (fmap id)`

Since each Monad is also a functor, we can use the functor law `fmap id = id` to transform the above into `join . fmap join = join . join`. Transforming this using the Haskell laws, we get

$$\begin{aligned}
LHS &= \texttt{join . (fmap join)} \\
[\text{def. (.)}] &= \texttt{(\x -> join (fmap join x))} \\
[(3)] &= \texttt{(\x -> join (x >>= (return . join)))} \\
[\text{def. join}] &= \texttt{(\x -> (x >>= (return . join)) >>= id)} \\
[(4)] &= \texttt{(\x -> x >>= (\y -> ((return . join) y) >>= id))} \\
[\text{def. (.)}] &= \texttt{(\x -> x >>= (\y -> (return (join y)) >>= id))} \\
[(1)] &= \texttt{(\x -> x >>= (\y -> id (join y)))} \\
[\text{def. id}] &= \texttt{(\x -> x >>= (\y -> join y))} \\
[\text{def. join}] &= \texttt{(\x -x >>= (\y -> y >>= id))} \\
[\text{def. id}] &= \texttt{(\x -> x >>= (\y -> id y >>= id))} \\
[(4)] &= \texttt{(\x -> (x >>= id) >>= id)} \\
[\text{def. join}] &= \texttt{(\x -> join (x >>= id))} \\
[\text{def. join}] &= \texttt{(\x -> join (join x))} \\
[\text{def. (.)}] &= \texttt{join . join} \\
&= RHS
\end{aligned}$$

The second law, $\mu_x \eta_{T(x)} = \mu_x T\eta_x = id$, translates into

`join . return = join . fmap return = id`

which we split into two equations

`join . return = id`
`join . fmap return = id`

Once again we transform using the haskell laws, starting with the first equation

$$\begin{aligned}
LHS_1 &= \texttt{join . return} \\
[\text{def. } \texttt{(.)}] &= \texttt{(\textbackslash x -> join (return x))} \\
[\text{def. } \texttt{join}] &= \texttt{(\textbackslash x -> return x >>= id)} \\
[(1)] &= \texttt{(\textbackslash x -> id x)} \\
[\text{eta-conversion}] &= \texttt{id} = RHS_1
\end{aligned}$$

And for the second equation we get

$$\begin{aligned}
LHS_2 &= \texttt{join . fmap return} \\
[(3)] &= \texttt{(\textbackslash x -> (x >>= return . return) >>= id)} \\
[\text{eta-conversion}] &= \texttt{(\textbackslash x -> (x >>= (\textbackslash y -> (return (return y)) >>= id)))} \\
[(1)] &= \texttt{(\textbackslash x -> (x >>= (\textbackslash y -> id (return y))))} \\
[\text{reverse eta-conversion}] &= \texttt{(\textbackslash x -> (x >>= return))} \\
&= \texttt{(\textbackslash x -> x)} = \texttt{id} = RHS_2
\end{aligned}$$

Thus we have that given an instance of the Monad typeclass that satisfies the Haskell monadic laws, it is indeed a monad on Hask. To show equivalence between the two sets of laws, one needs to show implication in the other direction. However, for the purposes of this paper, this is not needed.

8.5.2 Maybe is a Monad

We know that Maybe is a functor on Hask, so to show that it is a monad on Hask, we only need to show the monadic laws.

Let the Maybe monad be defined as previously stated, and consider

$$\begin{aligned}
LHS_1 &= \texttt{(return a >>= k)} \\
[\text{def. } \texttt{return} \text{ (p. 13)}] &= \texttt{(pure a >>= k)} \\
[\text{def. } \texttt{pure} \text{ (p. 12)}] &= \texttt{(Just a >>= k)} \\
[\text{def. } \texttt{>>=} \text{ (p. 13)}] &= \texttt{k x} = RHS_1
\end{aligned}$$

$$\begin{aligned}
LHS_2 &= \texttt{(m >>= return)} \\
[\text{def. } \texttt{return, pure}] &= \texttt{(m >>= Just)} \\
[\text{Split into cases for m}] &= \begin{cases} \texttt{Nothing >>= Just} & \texttt{m = Nothing} \\ \texttt{Just x >>= Just} & \texttt{m = Just x} \end{cases} \\
[\text{def. } \texttt{>>=}] &= \begin{cases} \texttt{Nothing} & \texttt{m = Nothing} \\ \texttt{Just x} & \texttt{m = Just x} \end{cases} \\
&= \texttt{m} = RHS_2
\end{aligned}$$

$$
\begin{aligned}
RHS_3 &= \texttt{fmap f x} \\
[\text{Split into cases for x}] &= \begin{cases} \texttt{fmap f Nothing} & \texttt{x = Nothing} \\ \texttt{fmap f (Just y)} & \texttt{x = Just y} \end{cases} \\
[\text{def. } \texttt{fmap} \text{ (p. 9)}] &= \begin{cases} \texttt{Nothing} & \texttt{x = Nothing} \\ \texttt{Just (f y)} & \texttt{x = Just y} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
LHS_3 &= \texttt{x >>= (return . f)} \\
[\text{def. } \texttt{(.)}] &= \texttt{x >>= (\a -> return (f a))} \\
[\text{def. } \texttt{return, pure}] &= \texttt{x >>= (\a -> Just (f a))} \\
[\text{Split into cases for x}] &= \begin{cases} \texttt{Nothing >>= (\a -> Just (f a))} & \texttt{x = Nothing} \\ \texttt{Just y >>= (\a -> Just (f a))} & \texttt{x = Just y} \end{cases} \\
[\text{def. } \texttt{>>=}] &= \begin{cases} \texttt{Nothing} & \texttt{x = Nothing} \\ \texttt{Just (f y)} & \texttt{x = Just y} \end{cases}
\end{aligned}
$$

Which gives us $LHS_3 = RHS_3$.

As for the fourth law, we have two cases, either `m = Nothing` or `m = Just x`. For the first case, we get

$$
\begin{aligned}
LHS_4 &= \texttt{Nothing >>= (\x -> k x >>= h)} \\
[\text{def. } \texttt{>>=}] &= \texttt{Nothing}
\end{aligned}
$$

and

$$
\begin{aligned}
RHS_4 &= \texttt{(Nothing >>= k) >>= h} \\
[\text{def. } \texttt{>>=}] &= \texttt{Nothing >>= h} \\
[\text{def. } \texttt{>>=}] &= \texttt{Nothing}
\end{aligned}
$$

So $LHS_4 = RHS_4$. As for the second case, we have

$$
\begin{aligned}
LHS_4 &= \texttt{Just x >>= (\x -> k x >>= h)} \\
[\text{def. } \texttt{>>=}] &= \texttt{k x >>= h}
\end{aligned}
$$

and

$$
\begin{aligned}
RHS_4 &= \texttt{(Just x >>= k) >>= h} \\
[\text{def. } \texttt{>>=}] &= \texttt{k x >>= h}
\end{aligned}
$$

So, again, $LHS_4 = RHS_4$, giving us that Maybe is indeed a monad.

8.5.3 List is a Monad

Recall that we have already shown that the list functor in Haskell is indeed a functor on Hask.

The Monad instance can be defined as

```
instance Monad [] where
  xs >>= f = (concat' (fmap f xs))
  return x = [x]


concat' :: [[a]] -> [a]
concat' [] = []
concat' ([] : xs) = concat' xs
concat' ((x : xs) : ys) = x : concat' (xs : ys)
```

where `concat'` takes a list of lists as argument, and concatenates its elements.

First, we state that List is indeed an endofunctor, since it is a functor from Hask to Hask. It is also equipped with the natural transformations η, μ as `return`, `join`.

We consider the Haskell equivalents of the monadic laws and state that

$$\begin{aligned} LHS_1 =& \texttt{return x >>= f} \\ [\text{def. } \texttt{return}] =& \texttt{[x] >>= f} \\ [\text{def. } \texttt{>>=}] =& \texttt{concat' [f x]} \\ =& \texttt{f x} \\ =& RHS_1 \end{aligned}$$

and

$$\begin{aligned} LHS_2 =& \texttt{m >>= return} \\ [\text{annotate the list } \texttt{m}] =& \texttt{[m1,m2,...] >>= return} \\ [\text{def. } \texttt{>>=}] =& \texttt{concat' (fmap return [m1,m2,...])} \\ [\text{def. } \texttt{fmap} \text{ (p. 9)}] =& \texttt{concat' [[m1],[m2],[...]]} \\ [\text{apply } \texttt{concat'}] =& \texttt{[m1,m2,...]} \\ =& \texttt{m} \\ =& RHS_2 \end{aligned}$$

meaning both unit laws hold.

Examining the third law gives us

$$
\begin{aligned}
LHS_3 &= \texttt{(x >>= (return . f))} \\
[\text{def. } \texttt{>>=}] &= \texttt{concat' (fmap (return . f) x)} \\
[\text{annotate the list } \texttt{x}] &= \texttt{concat' (fmap (return . f) [x1, x2, ...])} \\
[\text{def. } \texttt{fmap}] &= \texttt{concat' [[f x1], [f x2], ...]} \\
[\text{apply } \texttt{concat'}] &= \texttt{[f x1, f x2, ...]} \\
[\text{def. } \texttt{fmap}] &= \texttt{fmap f x} \\
&= RHS_3
\end{aligned}
$$

As for the associative law, note the following corollary

$$
\begin{aligned}
\texttt{m >>= f} &= \texttt{[m1, m2, ...] >>= f} \\
[\text{def. } \texttt{>>=}] &= \texttt{concat' (fmap f [m1, m2, ...])} \\
[\text{def. } \texttt{fmap}] &= \texttt{concat' [f m1, f m2, ...]} \\
[\text{def. } \texttt{concat'}] &= \texttt{f m1 ++ f m2 ++ ...}
\end{aligned}
$$

i.e. a list consisting of the concatenated results of applying f to the elements of m. Using this we get

$$
\begin{aligned}
LHS_4 &= \texttt{(m >>= f) >>= g} \\
[\text{corollary}] &= \texttt{(f m1 ++ f m2 ++ ...) >>= g} \\
[\text{def. } \texttt{>>=}] &= \texttt{concat' (fmap g (f m1 ++ f m2 ++ ...))} \\
[\text{distribute } \texttt{fmap} \text{ over } \texttt{++}] &= \texttt{concat'(fmap g (f m1) ++ fmap g (f m2) ++ ...)} \\
[\text{distribute } \texttt{concat'} \text{ over } \texttt{fmap}] &= \texttt{concat' (fmap g (f m1)) ++ concat' (fmap g (f m2)) ++ ...}
\end{aligned}
$$

$$
\begin{aligned}
RHS_4 &= \texttt{m >>= (\x -> f x >>= g)} \\
[\text{corollary}] &= \texttt{f m1 >>= g ++ f m2 >>= g ++ ...} \\
[\text{def. } \texttt{>>=}] &= \texttt{concat' (fmap g (f m1)) ++ concat' (fmap g (f m2)) ++ ...}
\end{aligned}
$$

So $LHS_4 = RHS_4$, and the associative law is satisfied. With this we can conclude that the list monad in Haskell is indeed a monad on Hask.

9 Conclusion

After exploring these concepts, we can see that there is a lot of room for mathematical abstractions in the field of computer science. Understanding these abstractions will make working in functional languages easier, and will provide more legible code — given that your reader also understands these concepts.

Due to the simple syntax of do-notation in Haskell, the monad also provides a point of entry for those who are not well versed in functional programming. By using this simplified syntax, and thinking of it imperatively, you can write code first, learn the simpler concepts of currying, higher order functions and so on, and later develop an understanding of the underlying structure of the functor, the applicative and the monad, should you choose to do so.

10 Related Work

The approach to understanding Haskell and category theory in parallel is tried and tested by many authors. The Haskell wiki - as an example - uses the same approach, but with a larger focus on Haskell. The category theory explained is contained in much the same form of presentation in the referenced books; which focus less - if at all - on Haskell.

11 Future Work

There is much to learn about both category theory and the Haskell abstractions. The next step in category theory is to be more thorough, following a strictly mathematical and rigorous textbook. The textbooks [1] and [3] are good examples.

Further reading on Haskell abstractions is available in multiple textbooks, and the next concept is likely that of lenses, that allow for easy modification of components of composite data types[5]

12 Analysis

As previously mentioned, exploring category theory and Haskell in parallell is not a new concept. However, finding a middle ground — with practical examples along with abstract theories — the goal was to make accessible introductions for those who are equally versed in mathematics and computer science. This goal has been adequatly fulfilled, by explaining the needed theory compactly and without unneccessary details.

References

[1] Mac Lane, Saunders *Categories for the Working Mathematician* Springer ISBN 0-521-47249-0

[2] Philip Wadler *Monads for functional programming*

[3] Pierre Antoine Grillet *Abstract Algebra* Springer ISBN 978-0-387-71567-4

[4] Simon Marlow *Haskell 2010 Language Report*
https://www.haskell.org/definition/haskell2010.pdf, September 25, 2019

[5] Hackage (Haskell Package Database)
https://hackage.haskell.org/package/lens
September 25, 2019

A Monads as Monoids

A.1 Product Categories

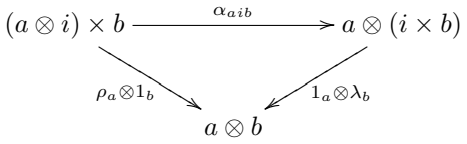
Let C, D be categories. The *product category* $C \times D$ consists of the objects (a, b), where $a \in C, b \in D$, and morphisms (f, g) where f is a morphism on C, and g is a morphism on D.

The composition on $C \times D$ is the composition of the components, i.e. $(f, g) \circ (\alpha, \beta) = (f \circ \alpha, g \circ \beta)$. The identity morphisms are the pair of identitiy morphisms of objects in C, D, i.e. $(1_c, 1_d)$, for $c \in C, d \in D$.

A.2 Monoidal Category

A Monoidal category [1, p.161] is a category C along with

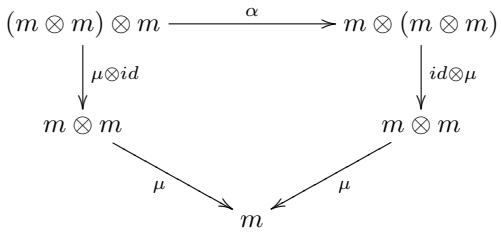
1. A bifunctor — a functor whose domain is a product category — called *tensor product* $\otimes : C \times C \to C$, where $\times$ is the cartesian product from set theory.
2. An object $i \in C$ called the *unit object*.
3. Three natural isomorphisms, that together demand that the tensor operation satisfies
   (a) Associativity, through the isomorphism $\alpha_{abc} :: (a \otimes b) \otimes c \to a \otimes (b \otimes c)$, often called the *associator*
   (b) Identity, shown by two isomorphisms;
      i. The *Left unitor* $\lambda_a : i \otimes a \to a$
      ii. The *Right unitor* $\rho_a : a \otimes i \to a$.



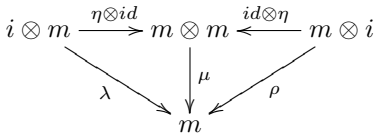
A.3 Monoid in a monoidal category

Let C be a monoidal category with tensor product $\otimes$. In order to create a monoid in C, we start by picking an object $m \in C$. Pick two morphisms on C:
$\mu : m \otimes m \to m$
$\eta : i \to m$
where i is the unit object of $\otimes$. Note that the source for the morphism μ is the result of applying the tensor product $m \otimes m$, which is an object in C. Further,

these morphisms must satisfy associativity



and unit laws



Remember the definition for a monoid, that requires a set of objects M, and a way to combine said objects, $\cdot$. If we take a monoidal category C, with tensor product $\otimes$, we can let $i \in C$, the unit object of $\otimes$, be the identity object in a monoid M. The tensor product can then act as the combination operator $\cdot$, and the monoid behaves as expected, due to the laws set on $\otimes$ [1, p.170].

A.4 Monoid in the Category of Endofunctors

Next, we look at the category of endofunctors on C, the category commonly denoted $[C, C]$. A monoid in this category requires a tensor product — to turn the category into a monoidal category. The tensor product is of the form $\otimes : [C, C] \times [C, C] \to [C, C]$, meaning it takes two endofunctors and combines them into one. The most intuitive way to do said combining is by endofunctor composition.

Take two functors $F, G \in [C, C]$. Their composition, $F \circ G$, is also an endofunctor on C, so intuitively, we could use endofunctor composition as a tensor product. However, in order to do so, we must formally prove that it satisfies the conditions.

First, we need to show that it is a bifunctor. The type signature of endomorphism composition is $\circ : [C, C] \times [C, C] \to [C, C]$, which shows that if it is a functor, it is also a bifunctor. The first condition for functors requires that $F \circ G(1_X) = 1_{F \circ G(X)}$ for all X in C. Since both F and G are functors, it follows that

$$F \circ G(1_X) = F(G(1_X)) = F(1_{G(X)}) = 1_{F(G(X))} = 1_{F \circ G(X)}$$

Next, morphism composition must be preserved, and we have

$$F \circ G(fg) = F(G(fg)) = F(G(f)(G(g))) = F(G(f))F(G(f)) = F \circ G(f)F \circ G(g)$$

meaning that $\circ$ is indeed a bifunctor on $[C, C]$.

We can use the identity functor $1_{\mathbf{C}}$ as the unit object on $[C, C]$, which means that we need no natrual isomorphisms λ, ρ to force identity, since $1_{\mathbf{C}}$ already satisfies it. As for associativity, we have

$$F \circ (G \circ H) = (F \circ G) \circ H$$

Meaning we have no need for the associator.

We can thus conclude that the category of endofunctors on a category is a monoidal category with functor composition as tensor product. What needs to be stated now is what a monoid in this category is.

We need to pick an object $T \in C$ with morphisms $\mu : T \otimes T \to T$, $\eta : i \to T$. In the category of endofunctors this translates to selecting an endofunctor and two natural transformations.

Since our tensor product is endofunctor composition, and our identity is $1_{\mathbf{C}}$, the identity functor of C, we get $\mu : T \circ T \to T$ and $\eta : 1_{\mathbf{C}} \to m$ Note that these type signatures correspond exactly to those of the monad.

The conditions set upon a monoid in a monoidal category require associativity, which when applying our endofunctor notation transforms into

$$\begin{array}{ccc} T^3 & \xrightarrow{T(\mu)} & T^2 \\ {\scriptstyle \mu}\downarrow & & \downarrow{\scriptstyle T\mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

Note the lack of the α arrow. Since endofunctor composition is associative, we can simply state that $T \circ (T \circ T) = (T \circ T) \circ T$. Also note that this law corresponds exactly to the first law set upon a monad [1, p. 137].

Note that $T = 1_{\mathbf{C}} \circ T = T \circ 1_{\mathbf{C}}$, so we have for the unit law diagram

$$\begin{array}{ccccc} 1_{\mathbf{C}} \circ T & \xrightarrow{\eta} & T \circ T & \xleftarrow[\eta]{} & T \circ 1_{\mathbf{C}} \\ & {\scriptstyle \lambda}\searrow & \downarrow{\scriptstyle \mu} & \swarrow{\scriptstyle \rho} & \\ & & T & & \end{array} \quad \Leftrightarrow \quad \begin{array}{ccc} T & \xrightarrow{\eta \circ id} & T^2 \\ {\scriptstyle id \circ \eta}\downarrow & \searrow & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow[\mu]{} & T \end{array}$$

Which corresponds exactly to the second law of the monad. From this we can conclude that — as previously suggested — the monad is a monoid in the category of endofunctors.