Understanding Monads
From types to categories to analogy
Nick Hu

ABSTRACT
Monads are often a mysterious topic in functional programming, partly due to their abstract nature and how they appear in many seemingly unrelated areas. Furthermore, there is a plethora of material to guide one to ‘understanding monads’ which make liberal use of analogy and skip mathematical reasoning. Such material can leave a reader baffled, and contributes to the wide opinion that monads are mysterious.

I address this by presenting monads from their definitions in Haskell, with commentary on what the laws intuitively mean. Then, I explore monads from a mathematical perspective, introducing basic Category theory, and showing how this allows us to reason about monads in Haskell. Finally, I examine some of the broader applications of monads by looking at the Maybe and list monads.

INTRODUCTION
Monads have existed in Haskell for a very long time now, and their utility pervades almost every corner of modern and useful Haskell code. Every Haskell program uses at least one monad, as the entrypoint is

```
main :: IO ()
```

However, monads themselves are about so much more than just IO, and you may already be using them without realising it.

IO is indeed a monad instance, but not a very nice one - the compiler treats it specially [Team 2016], and it is not very nice to reason about it - instead, we shall explore some monads which do not put into question Haskell’s purity. In learning Haskell, lasting understanding comes from understanding the types, so we shall build up from the ground up. The reader is expected to have a basic understanding of Haskell, including understanding the typeclass mechanism and the relation between types and kinds.

MONADS VIA APPLICATIVE FUNCTORS
Monad is a typeclass in Haskell, and is a subclass of Applicative, which in turn is a subclass of Functor.

Each of these typeclasses has laws, which are not enforced by the compiler, but are necessary to preserve their relationships to the mathematical objects they represent.

Functor
From the Haskell Prelude, we have the typeclass Functor (of kind * -> *):

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

We can think of a functor as something that can be mapped over, like a container; Haskell lists ([]) are functors with map as fmap. This is even clearer when the types are lined up:

```
fmap :: (a -> b) -> f a -> f b
map :: (a -> b) -> [a] -> [b]
```

Observe that the type of fmap can also be written as (a -> b) -> (f a -> f b).

If we let \( g :: a \rightarrow b \) be a function, with domain \( a \) and codomain \( b \), then \((fmap g) :: f a \rightarrow f b\) is a function with domain \( f a \) and codomain \( f b \); in other words, the domain and codomain of \( g \) has been ‘lifted’ into the functor \( f \).

Laws
fmap must satisfy the functor laws\(^1\), defined as:

\[
\text{fmap id } x = x
\]

\[
\text{fmap (g . h) } x = (\text{fmap g}) \circ (\text{fmap h})
\]

Intuitively, this can be thought of as limiting fmap to not change the structure of the functor, but only its value. The second functor law states that mapping \( h \) after \( g \) over a functor is the same as mapping the composite \( g \circ h \) over that functor, which generalises fusion over all functors.

Applicative
An applicative functor, captured by the Applicative typeclass, is a special kind of functor:

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

pure allows any value to be ‘lifted’ into the applicative functor, and <*>\(^2\) can apply functions lifted into the functor on values inside that functor.

Laws
Applicative functors must maintain their relation to functors such that:

\[
\text{fmap g } x = \text{pure g } x
\]

In addition, applicative functors must also satisfy several laws:

- Identity: \( \text{pure id } x = x \)

\[^1\]Knott [2015] argues that the second functor law can be derived from the first as a free theorem [Wadler 1989], but this does not yet seem to have become universally accepted in the Haskell community.

\[^2\] <*> is pronounced ‘ap’ as in ‘apply’.
Homomorphism: pure g ** pure x = pure (g x)

Interchange: x ** pure y = pure ($ y) ** x  
   = ($ y) is syntactically equivalent to ($ g -> g y).

Composition: 
   x ** (y ** z) = pure (.). (**) x ** y ** z

These laws facilitate a form of normalisation, specifically 
that any expression written with pure and ** can be 
transformed into an expression using pure only once at the 
beginning, and left associative\(^3\) occurrences of ** [McBride 
and Paterson 2008].

This introduces the notion of the applicative idiom in Haskell 
code, whereby a chain of applicative and non-applicative 
values can be applied to a non-applicative function:

g :: t1 -> t2 -> t3 -> ... -> tm -> tn 
x :: f t1 
y :: t2 
(pure y) :: f t2 
...

z :: f tm

(pure g) :: f (t1 -> t2 -> t3 -> ... -> tm -> tn)

(pure g ** x) :: f (t2 -> t3 -> ... -> tm -> tn)

(pure g ** x ** pure y) :: f (t3 -> ... -> tm -> tn)

(pure g ** x ** pure y ... ** z) :: f tn

We can interpret this as a sequence of 'actions', delimited by **. It is also no coincidence that fmap = liftA = liftM, 
which is where the terminology for 'lifting' comes from, and 
in fact this is a relic of older versions of ghc where the 
Functor-Applicative-Monad hierarchy had not been explicitly 
coded in the typeclass definitions. Indeed, this pattern is 
a generalisation up to n terms for liftA\(^2\) and liftA\(^3\) 
which respectively operate on 2 and 3 terms.

\(<\)\(\) is also provided in Haskell as an infix version of fmap; we can 
apply the applicative idiom by rewriting the first equa-
tion to g <$> x = pure g <$> x. This pattern seems to 
very frequently; for example, when sequencing effect-
ful computations [McBride and Paterson 2008], or efficient 
context-free parsing [Rójemo 1995].

Applicative functors offer more power than a regular functor, 
allowing the sequencing of applicative actions and injection 
of non-applicative values into applicative context, but none 
of the actions in sequence can depend upon previous actions.

**Monad**

The typeclass Monad is a subclass of Applicative, and de-
finest an additional operator:

\class\ (Applicative m) \Rightarrow\ Monad m where 

\(\Rightarrow\)\(^\text{a}\) allows a value to be taken out of a monadic 
case, and then applied to some function which lifts it into the 
same monad, before returning that monadic value.

\(\Rightarrow\) is a convenience function, defined by 

\(\Rightarrow\) :: Monad m \Rightarrow m a \Rightarrow m b 

\(x \Rightarrow y = x \Rightarrow (\_ \Rightarrow y)\)

and is used to sequence monadic actions when the value can 
be discarded; for example, some actions in the IO monad 
don't produce useful values, but running the action itself is 
useful (e.g. putStrLn :: String -> IO ()).

Consider the Prelude function:

\(\Rightarrow\) :: Monad m \Rightarrow (a -> m b) -> m a -> m b

\(\Rightarrow\) is \(\text{flip} \ (\Rightarrow)\)

It is interesting to line it up against the type of ** from 
Applicative:

\(\Rightarrow\) :: Applicative m \Rightarrow m (a -> b) -> m a -> m b

So, it is clear that Monad allows the responsibility of lifting 
to the monad to be delegated to the function that is sup-
plied to it, rather than a non-monadic function applied to 
pure.

If we view m a as a computation, then (a -> m b) can op-
erate on the result of that computation, and give us a new 
b computation to be run. But, we can choose any m b we 
want based on the value of a! This means that monadic ac-
tion sequencing allows for dependancies on previous monadic 
values in the sequence, with >> as our sequencing operator. 
We will see later that this allows us to define list comprehen-
sions, and context-sensitive parsers\(^6\).

**Laws**

Monads must adhere to three laws:

pure x >>= g = g x

x >>= pure = x

(x >>= g) >>= h = x >>= (\_ \rightarrow g \_ \rightarrow h)

The laws are difficult to reason about in this form\(^7\), but we 
will soon see an alternative and equivalent construction.

Note that the >>= operator looks a bit like composition, 
and the monad laws look a bit like laws describing left identity, 
right identity and associativity...

**MONADS VIA CATEGORIES**

Monads themselves were originally formulated in a theory 
of mathematical structure called Category theory; many 
monad tutorials gloss over Category theory entirely, but it 
is an indispensable tool to understanding them, and can 
be considered the basis for equational reasoning in Haskell 
[Danielsson et al. 2006].

**Definitions**

**Definition 1.** A category C is defined to be a collection 
of objects, ob (C), and a collection of morphisms (or arrows) 
\(\text{hom} (C), \) where

\(\bullet\) if f is a morphism, there exist objects 

\(\text{dom} (f) \text{ and cod} (f)\)

in ob (C) called the domain and codomain of f; we 
write 

\(f: A \rightarrow B\)

to indicate that dom (f) = A and cod (f) = B,

\(\bullet\) given morphisms \(f: A \rightarrow B \text{ and } g: B \rightarrow C, \)

there exists a morphism 

g \cdot f: A \rightarrow C

\(^3\)As function application associates to the left, such an ex-
pression does not need to be bracketed.

\(^4\)The actual definition in ghc contains more functions, like 
return, >>, and fail. However, return is always equiva-

tent to pure and exists from pre-Functor-Applicative-Monad 
hierarchy, and fail is a shim to allow for partial pattern 
matching. I have also omitted fixity declarations as they 
are clear in reading.

\(^5\)>> is pronounced ‘bind’.

\(^6\)Due to general recursion and laziness in Haskell, we can ac-
tually do context-sensitive parsing with Applicative [Yorgey 
2012].

\(^7\)The reason this form is used is to facilitate do notation, 
which can enable one to write something that 'looks like' 
imperative code whilst maintaining purity in Haskell.
called the \textit{composite} of \( f \) and \( g \),

\begin{itemize}
\item for each object \( A \), there exists a morphism
\[
\text{id}_A : A \rightarrow A
\]
called the \textit{identity morphism} of \( A \),

\begin{itemize}
\item and the following axioms hold:
\begin{itemize}
\item composition is associative:
\[
h \cdot (g \cdot f) = (h \cdot g) \cdot f
\]
for all \( f : A \rightarrow B \), \( g : B \rightarrow C \), and \( h : C \rightarrow D \),
\item composition has left and right identities:
\[
\text{id}_B \cdot f = f = f \cdot \text{id}_A
\]
for all \( f : A \rightarrow B \).
\end{itemize}
\end{itemize}
\end{itemize}

Definition 2. A \textit{subcategory} \( S \) of category \( C \) is given by a subcollection of objects of \( C \), \( \text{ob}(S) \), and a subcollection of morphisms of \( C \), \( \text{hom}(S) \), such that

\begin{itemize}
\item for every \( A \) in \( \text{ob}(S) \), its corresponding identity morphism \( \text{id}_A \) is in \( \text{hom}(S) \),
\item for every morphism \( f : A \rightarrow B \) in \( \text{hom}(S) \), both \( \text{dom}(f) \) and \( \text{cod}(f) \) are in \( \text{ob}(S) \),
\item for every pair of morphisms \( f \) and \( g \) in \( \text{hom}(S) \), the composite \( f \cdot g \) is in \( \text{hom}(S) \) whenever it is defined.
\end{itemize}

Note that \( S \) is just \( C \) with some of its objects and morphisms removed.

Many things across many disciplines form categories (most obviously, the category \textit{Set} with mathematical sets as objects and functions as morphisms), but the category we are interested in is \texttt{Hask}, where the objects are Haskell types and the morphisms are Haskell functions, and its subcategories.

Definition 3. \texttt{Hask} forms a category:

\begin{itemize}
\item Every Haskell function has a domain and codomain which can be encoded as Haskell types,
\item For a morphism \( f : a \rightarrow b \), and a morphism \( g : b \rightarrow c \), the composite \( (g \cdot f) : a \rightarrow c \) exists,
\item For each type \( a \), the identity morphism exists as \( \text{id}_a : a \rightarrow a \).
\end{itemize}

Furthermore,

\begin{itemize}
\item \( (\cdot) \) is associative,
\item with \( f : a \rightarrow b \), and by instantiating \( \text{id} \) with a monomorphic type, we have
\[
(\text{id} : b \rightarrow b) \cdot f = f = f \cdot (\text{id} : a \rightarrow a)
\]
for all types \( a \) and \( b \).
\end{itemize}

\textbf{Functors}

Definition 4. A \textit{functor}

\[
F : C \rightarrow C'
\]

between categories \( C \) and \( C' \) maps \( \text{ob}(C) \) to \( \text{ob}(C') \) and \( \text{hom}(C) \) to \( \text{hom}(C') \) such that the following axioms hold:

\begin{itemize}
\item \( F \) preserves domains and codomains:
\[
F(f : A \rightarrow B) = F(f) : F(a) \rightarrow F(b),
\]
\item \( F \) preserves identities:
\[
F(\text{id}_A) = \text{id}_{F(A)}.
\]
\end{itemize}

\texttt{Hask} is not a real category, due to \texttt{undefined} (\_), but for our purposes this can be safely ignored [Danielsson et al. 2006].

\begin{itemize}
\item \( F \) distributes over composition of morphisms:
\[
F(f \cdot g) = F(f) \cdot F(g).
\]
\end{itemize}

Definition 5. An \textit{endofunctor} is a functor which maps a category to itself.

Definition 6. The Haskell \texttt{Functor} typeclass specifies functors from \texttt{Hask}. Given a type constructor \( f :: \star \rightarrow \star \) and a higher-order function \texttt{fmap} : \( (a \rightarrow b) \rightarrow (f a \rightarrow f b) \), define \texttt{func} as a subcategory of \texttt{Hask} such that

\[
\text{ob(func)} \equiv \text{types of the form} f a,
\]
\[
\text{hom(func)} \equiv \text{functions with the signature} f a \rightarrow f b.
\]

Then it is clear that the pair \( (f, \text{fmap}) \) forms a functor from \texttt{Hask} to \texttt{func}.

For example, the \texttt{List} subcategory of \texttt{Hask} contains only list types \([a]\) as objects, and functions of type \([a] \rightarrow [b]\) as morphisms, where \([1] \mapsto \text{map}\) forms a functor from \texttt{Hask} to \texttt{List}.

The functor laws described before are just the axioms for functors in Category theory transcribed into Haskell!\footnote{Functors preserving domains and codomains are guaranteed by the type of \texttt{fmap}.
\texttt{eta} and \( \mu \) are usually interpreted as natural transformations instead of morphisms in other literature about Category theory.}

\textbf{Monads}

Definition 7. A \textit{monad} is an endofunctor \( M : C \rightarrow C \), with two morphisms for each object \( X \) in \( \text{ob}(C) \),

\[
\eta : X \rightarrow M(X),
\]
and

\[
\mu : M(M(X)) \rightarrow M(X),
\]
such that the following axioms must also hold:\footnote{\texttt{eta} and \( \mu \) are usually interpreted as natural transformations instead of morphisms in other literature about Category theory.}

\[
\mu \cdot M(\mu) = \mu \cdot \mu,
\]
\[
\mu \cdot M(\eta) = \mu \cdot \eta = \text{id}_X,
\]
\[
\eta \cdot f = M(f) \cdot \eta,
\]
\[
\mu \cdot M(M(f)) = M(f) \cdot \mu,
\]
where \( f \) is a morphism \( f : A \rightarrow B \) for \( A \) and \( B \) in \( \text{ob}(C) \).

Definition 8. Monads can be formulated in \texttt{Hask} as follows:

\begin{verbatim}
class Monad m where  
  unit :: a -> m a  
  join :: m (m a) -> m a  
with unit = \eta and \mu = \join.
Note that unit is identical to \texttt{pure} from our previous monad definition.

The monad axioms transcribed into \texttt{Hask}:
join \cdot fmap join = \join \cdot \join
join \cdot fmap unit = \join \cdot unit = \text{id}
unit \cdot g = fmap g \cdot unit
join \cdot fmap (fmap g) = fmap g \cdot \join
where g :: a \rightarrow b

theorem 1: The function \texttt{>>=} is equivalent to \texttt{fmap} and \texttt{join}.
\end{verbatim}
Proof. For equivalence, it is necessary to show that a function of the type of \( \gg \gg \) can be constructed by an expression using only \texttt{fmap} and \texttt{join} and vice versa. Furthermore, it is also necessary to show that the axioms for monads in the category \texttt{Hask} imply the monad laws and vice versa.

Part 1. \( \gg \gg \) can be written in terms of \texttt{fmap} and \texttt{join} and vice versa. Furthermore, it is also necessary to show that the axioms for monads in the category \texttt{Hask} imply the monad laws and vice versa.

\begin{itemize}
  \item \textbf{Part 1.} \( \gg \gg \) can be written in terms of \texttt{fmap} and \texttt{join}.
  \begin{align*}
    x >> y &= \texttt{join} (\texttt{fmap} \ y \ x) \\
  \end{align*}
  \item \textbf{Part 2.} The monad laws can be derived with the axioms for monads in the category \texttt{Hask}.
  \begin{enumerate}
    \item \textbf{First monad law:}
      \begin{align*}
        \texttt{pure} x >> g &= \texttt{unit} x >> g \\
      \end{align*}
    \item \textbf{Second monad law:}
      \begin{align*}
        x >> \texttt{pure} &= x >> \texttt{unit} \\
      \end{align*}
    \item \textbf{Third monad law:}
      \begin{align*}
        (x >> g) >> h &= x >> (g \gg \gg \texttt{join} \gg \gg \texttt{fmap} \ h \gg \gg g) \\
      \end{align*}
  \end{enumerate}
  \item \textbf{Fourth monad law:}
    \begin{align*}
      (\texttt{join} \gg \gg \texttt{fmap} \ h \gg \gg g) x &= x >> (\texttt{join} \gg \gg \texttt{fmap} \ h \gg \gg g) \\
    \end{align*}
\end{itemize}

\textsuperscript{1}Equations marked with a * are presented again in the appendix with additional type annotations to aid the reader.
Kleisli categories of Hask

Definition 9. For any subcategory of Hask with morphisms of type \( a \rightarrow b \), define its Kleisli category to have the same objects, but morphisms of type \( \text{Monad } m \Rightarrow a \rightarrow m b \) where \( m \) is a monad. The identity morphisms are given by instantiating \( \text{unit} : a \rightarrow m a \) with the appropriate type, and composition is given by*:

\[
(f <<< g) (x) = (f <<< (g <<< h)) x
\]

"<<<" is a suitable composition operator for a Kleisli category.

Proof. For "<<<" to be considered a suitable composition operator, we must show that it is associative and has left and right identities.

Part 1. "<<<" is associative:

\[
(f <<< (g <<< h)) x = (f <<< (g <<< h)) x
\]

Part 2. "<<<" has left identity:

\[
\text{unit} <<< f = \text{unit} <<< f
\]

Part 3. "<<<" has right identity:

\[
(f <<< \text{unit}) x = (f <<< \text{unit}) x
\]
\[ f \ x \]

Thus, \(<\left<\right>\) is suitable. □

**Theorem 3.** The monad laws are equivalent to the properties of the Kleisli composition operator.

**Proof.** From theorem 2, we establish that the monad laws imply the existence of the Kleisli composition operator. If we assume that the Kleisli composition operator is well-defined, because it is constructed entirely from \(\text{join}\) and \(\text{fmap}\), they must be well defined also, and therefore the existence of the monad laws is implied by theorem 1. □

So it is clear now that monads provide a generalisation of function composition! The laws required by monads are merely the properties of this special type of composition.

**MONADS BY EXAMPLE**

Now we shall examine how monads can be hidden behind the syntax sugar of list comprehensions, and how **Monad** allows us to do more than **Applicative**

**List monad**

Let’s focus on the list monad, and see how it is equivalent to list comprehensions.

Define the instances for the list monad:

\[
\text{instance} \quad \text{Functor} \quad [] \quad \text{where} \\
\quad \text{fmap} \quad g \quad = \quad [] \\
\quad \text{fmap} \quad (x:xs) \quad = \quad g \ x \ : \ \text{fmap} \ g \ xs
\]

\[
\text{instance} \quad \text{Applicative} \quad [] \quad \text{where} \\
\quad \text{pure} \quad x \quad = \quad [x] \\
\quad (f <\left<\right>) \quad = \quad [] \\
\quad (g:gs) \quad <\left<\right> \quad xs \quad = \quad (g <\left<\right> \ xs) \ + \ (gs \ <\left<\right> \ xs)
\]

\[
\text{instance} \quad \text{Monad} \quad [] \quad \text{where} \\
\quad (x \ +\ +) \quad = \quad \text{concatMap} \ b \ xs
\]

The reader is encouraged to try to derive an equivalent expression which matches list comprehensions using only \(\text{fmap}/<\left<\right>\), \(\text{pure}/<\left<\right>\) and \(\text{>>=}\) — or similarly, an expression which matches expressions built only from those functions using only list comprehensions — before looking at the example solution.

**Mapping over lists**

\[
\text{-- let } f :: \ a \rightarrow b, \ xs :: [a], \ ys :: [[a]] \quad \text{where} \\
\quad \text{1. } f \text{ map } f \text{ xs} \\
\quad \text{2. } [ \{ f \ y \mid y \left<\left<\right> \ ys \} \mid ys \left<\left<\right> \ ys \} \\
\quad \text{-- solutions} \\
\quad \text{1. } [ f \ x \mid x \left<\left<\right> \ xs ] \\
\quad \text{2. } \text{fmap} \ (f \text{ map } f) \text{ ys}
\]

The functor instance allows lists to be mapped over.

**Mapping multiple functions with multiple arguments over lists**

\[
\text{-- let } g :: \ a \rightarrow b \rightarrow c, \quad \text{-- } f :: [a \rightarrow b], \quad g :: [a \rightarrow b \rightarrow c], \quad \text{-- } xs :: [a], \quad ys :: [b] \\
\quad \text{1. } [ g \ x \ y \mid x \left<\left<\right> \ xs, \ y \left<\left<\right> \ ys ] \\
\quad \text{2. } f \text{ map } f \text{ xs} \\
\quad \text{3. } [ g \ x \ y \mid g \left<\left<\right> \ gs, \ x \left<\left<\right> \ xs, \ y \left<\left<\right> \ ys ] \\
\quad \text{-- solutions} \\
\quad \text{1. } g \ <\left<\right> \ xs \ <\left<\right> \ ys
\]

2. \( [ f \ x \mid f \left<\left<\right> \ fs, \ x \left<\left<\right> \ xs ] \\
3. \ g \ <\left<\right> \ xs \ <\left<\right> \ ys
\]

With the applicative functor instance, we can specify multiple lists to draw from on the right side of the list comprehension, and we can apply multiple arguments to a function.

**Filtering and depending on previous values**

\[
\text{-- let } f :: \ a \rightarrow b, \ p :: a \rightarrow \text{Bool}, \ xs :: [a] \\
\quad \text{1. } [ f \ x \mid x \left<\left<\right> \ xs, \ p \ x ] \\
\quad \text{2. } [ y \mid x \left<\left<\right> \ xs, \ y \left<\left<\right> \ f \ x ] \\
\quad \text{3. } [ \{ \} \mid x \left<\left<\right> \ [\{\} \mid x \left<\left<\right> \ ] \}
\]

**Choice**

Firstly, define the instances for the **Maybe** monad:

\[
\text{instance} \quad \text{Functor} \quad \text{Maybe} \quad \text{where} \\
\quad \text{fmap} \quad g \quad = \quad \text{Nothing} \quad = \quad \text{Nothing} \\
\quad \text{fmap} \quad (f \text{ Just } x) \quad = \quad \text{Just} \ (f \ x)
\]

\[
\text{instance} \quad \text{Applicative} \quad \text{Maybe} \quad \text{where} \\
\quad \text{pure} \quad = \quad \text{Just} \\
\quad (\text{Just} \ f) \quad \text{get\ } \ (\text{Just} \ x) \quad = \quad \text{Just} \ (f \ x) \\
\quad (\text{pure} \ False) \quad = \quad \text{Nothing}
\]

\[
\text{instance} \quad \text{Monad} \quad \text{Maybe} \quad \text{where} \\
\quad (\text{Just } x) \quad \text{get\ } \ (\text{Just } f) \quad = \quad \text{Just} \ (f \ x) \\
\quad (\text{pure} \ True) \quad t \quad e \quad = \quad t \\
\quad (\text{pure} \ False) \quad t \quad e \quad = \quad e
\]

This function can be defined as follows:

\[
\text{ifM} \quad mx \quad t \quad e \quad = \quad m \ x \quad >>\quad \text{if} \ (x \quad \text{then} \quad t \quad \text{else} \quad e)
\]

However, this cannot be defined using just an applicative instance; if we try to come up with an expression for

\[
\text{ifA} :: \ \text{Applicative} \quad f \quad \text{Bool} \quad \text{get\ } \ f \quad a \quad \text{get\ } \ f \quad a \quad f \quad a, \quad \text{one might see that}^{*}
\]

\[
\text{ifA} \quad mx \quad t \quad e \quad = \quad \text{if} \ (x \quad \text{then} \quad y \quad \text{else} \quad z) \quad \text{ifA} \quad mx \quad t \quad e \quad = \quad \text{if} \ (x \quad \text{then} \quad y \quad \text{else} \quad z)
\]

But when we try to use each function, except for example in the **Maybe** monad, we see that **ifA** works as expected, but the intended semantics do not hold for **ifA**:

\[
\text{ifM} \quad \text{(Just True)} \quad (\text{Just } () \quad \text{Nothing} \quad = \quad \text{-- definition of ifM} \\
\quad (\text{Just True}) \quad >>\quad (x \quad \text{then} \quad \text{if} \ (x \quad \text{then} \quad (\text{Just } () \quad \text{else} \quad \text{Nothing}) \quad = \quad \text{-- definition of get\ } \quad \text{ifA} \\
\quad \text{if True then} \quad (\text{Just } () \quad \text{else} \quad \text{Nothing} \quad = \quad \text{-- if expression True branch} \\
\quad \text{Just } ()
\]

\[
\text{ifA} \quad \text{(Just True)} \quad (\text{Just } () \quad \text{Nothing} \quad = \quad \text{-- definition of ifA}
\]
CONCLUSION

We have seen how monads are built from both a Haskell and a mathematical perspective, and how the two derivations are related. Furthermore, we have explained how the laws of monads work from three different structures. We have also seen that monads are not just used for monadic sequences over containers, one might be tempted to think: 'Is my functor a monad?'

FURTHER READING

- Monads can be understood in terms of monoids [Piponi 2008].
- Applicative functors also have an alternative formation in category theory as lax monoidal functors [Yang 2012].
- Monad transformers allow monads to be combined into a single monad to combine several effects [Grabmüller 2006].
- mtl monad classes provide typeclasses to generalise over monads which provide the same effects.
- Lenses provide highly generic abstractions for getters, setters traversals and folds over data types.
- Monads can be generalised into arrows [Hughes 2000].
- MonadZero, MonadPlus and Alternative allow for monads to fail, monads to encode choice, and applicative functors to have choice respectively by adding monoidal properties [Yorgey 2009].
- Foldable and Traversable typeclasses provide generalisations over data types which can be folded and sequenced respectively [Yorgey 2009].
- Comonads provide abstractions which can be viewed as objects in the sense of object oriented programming [Gonzalez 2013], or streams.

Some useful monad instances:

\[
(x y z \rightarrow \text{if } x \text{ then } y \text{ else } z) \Downarrow
\]

(Just True) \Downarrow (Just () ) \Downarrow \text{Nothing}

= \bot

\[
(x y z \rightarrow \text{if } x \text{ then } y \text{ else } z) \Downarrow
\]

(Just True) \Downarrow (Just () ) \Downarrow \text{Nothing}

= \Downarrow \text{ definition of } \bot

Nothing

We need monads to be able to 'short-circuit' on actions in the sequence, so they give us more choice - applicative functors must run all of the actions. However, this means that with applicative functors we can split up a sequence into chunks which can be ran in parallel; in fact, the applicative functor laws guarantee a property that is a bit like associativity for \(\Downarrow\), with the caveat that functions must be fully applied. As a final exercise, the reader is encouraged to show that the laws for each instance of Maybe and [] hold.

APPENDIX

DO NOTATION

do notation is a syntax sugar for monadic sequences using \(\Downarrow\). It allows for the writing of programs which look very much like imperative code, but is arguably harder to reason about [Hudak 2007]; in particular, the order of statements in a do block is not the same as the order of evaluation, which is unlike any imperative language. Plenty of Haskell code does use do notation, and as long the writer of a Haskell program understands what the do notation is doing, it is not dangerous. Fortunately, the rules for do notation are rather simple:

\[
\text{do } \{ a \leftarrow f ; m \} \Downarrow \{ \text{a } \rightarrow \text{do } \{ m \} \}
\]

-- bind f to a, proceed to desugar m

\[
\text{do } \{ f ; m \} \Downarrow \text{do } \{ m \}
\]

-- evaluate f, then proceed to desugar m

\[
\text{do } \{ m \} \Downarrow
\]

As per usual, a block delimited by semicolons and curly braces can be written over multiple lines with appropriate indentation.

TYPE ANNOTATIONS

\[
x \Downarrow y = \text{join } (\text{fmap } y \ x)
\]

\[
= y :: x \rightarrow m b
\]

-- (\text{fmap } y \ x) :: m a \rightarrow m (m b)

-- (\text{fmap } y \ x) :: m b

-- (\text{join } (\text{fmap } y \ x)) :: m b

\text{join } x = x \Downarrow id

-- x :: m (m a)

-- (x \Downarrow id) :: m a

\text{fmap } g \ x = x \Downarrow (\text{pure } . \ g)

-- x :: m a

-- g :: a \rightarrow b

-- (\text{pure } . \ g) :: a \rightarrow m b

-- (x \Downarrow (\text{pure } . \ g)) :: m b

\[
((\text{join } . \text{fmap } y) . g) :: a \rightarrow m c
\]

\[
\text{f } \Downarrow g = \text{join } . \text{fmap } f . g
\]

-- (f) :: b \rightarrow m c

-- (g) :: a \rightarrow m b

-- (\text{fmap } f) :: m b \rightarrow m (m c)

-- (\text{fmap } f . g) :: a \rightarrow m (m c)

-- (\text{join } . \text{fmap } f . g) :: a \rightarrow m c

\text{if\forall } t e =

\[
(x y z \rightarrow \text{if } x \text{ then } y \text{ else } z) \Downarrow
\]

\[
\text{mx } \Downarrow t \Downarrow e
\]
-- (\x y z -> if x then y else z)
-- :: Bool -> a -> a -> a
-- (fmap (\x y z -> if x then y else z))
-- :: f Bool -> f (a -> a -> a)
-- ((\x y z -> if x then y else z) <$> mx)
-- :: f (a -> (a -> a))
-- ((\x y z -> if x then y else z) <$> mx <*> t)
-- :: f (a -> a)
-- ((\x y z -> if x then y else z) <$> mx <*> t <*> f)
-- :: f a

References


Bibliography