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 -> b` be a function, with domain `a` and codomain `b`, then `(fmap g) :: f a -> 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:

```
fmap id = id
fmap (g . h) = (fmap g) . (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` and then `g` over a functor is the same as mapping the composite `g . 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:

```
fmap g x = pure g <*> x
```

In addition, applicative functors must also satisfy several laws:

- Identity: `pure id <*> v = v`

[1] Kmett [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 such that any expression written with `pure` and `<*>` can be transformed into an expression using `pure` only once at the beginning, and left associative[3] occurences 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 encoded in the typeclass definitions. Indeed, this pattern is a generalisation up to n terms for `liftA2` and `liftA3` which respectively operate on 2 and 3 terms.

`<$>` is also provided in Haskell as an infix version of `fmap`; we can apply the applicative idiom by rewriting the first equation to `g <$> x = pure g <*> x`. This pattern seems to occur very frequently; for example, when sequencing effectful computations [McBride and Paterson 2008], or efficient context-free parsing [Röjemo 1995].

Applicative functors offer more power than a regular functor, enabling 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 defines an additional operator:[4]

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

`>>=`[5] allows a value to be taken out of a monadic context, and then applied to some function which lifts it into the same monad, before returning that monadic value.

`>>` is a convenience function, defined by

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

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
(=<<) = flip (>>=)
```

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

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

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

If we view `m a` as a computation, then `(a -> m b)` can operate on the result of that computation, and give us a new `m b` computation to be run. But, we can choose any `m b` we want based on the value of `a`! This means that monadic action sequencing allows for dependencies on previous monadic values in the sequence, with `>>=` as our sequencing operator. We will see later that this allows us to define list comprehensions, and context-sensitive parsers[6] .

Laws

Monads must adhere to three laws:

```
pure x >>= g = g x
x >>= pure = x
(x >>= g) >>= h = x >>= (\v -> g v >>= 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* $\mathbf{C}$ is defined to be a collection of objects, $\mathrm{ob}\,(\mathbf{C})$, and a collection of morphisms (or arrows) $\hom(\mathbf{C})$, where

- if f is a morphism, there exist objects
$$\mathrm{dom}\,(f) \text{ and } \mathrm{cod}\,(f)$$
in $\mathrm{ob}\,(\mathbf{C})$ called the *domain* and *codomain* of f; we write
$$f\colon A \to B$$
to indicate that $\mathrm{dom}\,(f) = A$ and $\mathrm{cod}\,(f) = B$,
- given morphisms $f\colon A \to B$ and $g\colon B \to C$, there exists a morphism
$$g \cdot f\colon A \to C$$

[3]As function application associates to the left, such an expression does not need to be bracketed.

[4]The actual definition in `ghc` contains more functions, like `return`, `>>`, and `fail`. However, `return` is always equivalent 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 fixivity declarations as they are clear in reading.

[5]`>>=` is pronounced 'bind'.

[6]Due to general recursion and laziness in Haskell, we can actually 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 *composite* of f and g,

- for each object A, there exists a morphism

$$\mathrm{id}_A \colon A \to A$$

called the *identity morphism* of A,

and the following axioms hold:

- composition is associative:

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

for all $f \colon A \to B$, $g \colon B \to C$, and $h \colon C \to D$,

- composition has left and right identities:

$$\mathrm{id}_B \cdot f = f = f \cdot \mathrm{id}_A$$

for all $f \colon A \to B$.

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

- for every A in $\mathrm{ob}\,(\mathbf{S})$, its corresponding identity morphism id_A is in $\mathrm{hom}\,(\mathbf{S})$,
- for every morphism $f \colon A \to B$ in $\mathrm{hom}\,(\mathbf{S})$, both $\mathrm{dom}\,(f)$ and $\mathrm{cod}\,(f)$ are in $\mathrm{ob}\,(\mathbf{S})$,
- for every pair of morphisms f and g in $\mathrm{hom}\,(\mathbf{S})$, the composite $f \cdot g$ is in $\mathrm{hom}\,(\mathbf{S})$ whenever it is defined.

Note that $\mathbf{S}$ is just $\mathbf{C}$ with some of its objects and morphisms removed.

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

Definition 3. **Hask** forms a category:[8]

- Every Haskell function has a domain and codomain which can be encoded as Haskell types,
- For a morphism `f :: a -> b`, and a morphism `g :: b -> c`, the composite `(g . f) :: a -> c` exists,
- For each type `a`, the identity morphism exists as `id :: a -> a`.

Furthermore,

- `(.)` is associative,
- with `f :: a -> b`, and by instantiating `id` with a monomorphic type, we have

```
(id :: b -> b) . f = f = f . (id :: a -> a)
```

for all types `a` and `b`.

Functors

Definition 4. A *functor*

$$F \colon \mathbf{C} \to \mathbf{C}'$$

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

- F preserves domains and codomains:

$$F(f \colon A \to B) = F(f) \colon F(a) \to F(b),$$

- F preserves identities:

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)},$$

- F distributes over composition of morphisms:

$$F(f \cdot g) = F(f) \cdot F(g).$$

Definition 5. An *endofunctor* is a functor which maps a category to itself.

Definition 6. The Haskell `Functor` typeclass specifies functors from **Hask**. Given a type constructor `f :: * -> *` and a higher-order function `fmap :: (a -> b) -> (f a -> f b)`, define **func** as a subcategory of **Hask** such that

$$\mathrm{ob}\,(\mathbf{func}) \coloneqq \text{types of the form } \texttt{f a},$$
$$\mathrm{hom}\,(\mathbf{func}) \coloneqq \text{functions with the signature } \texttt{f a -> f b}.$$

Then it is clear that the pair (`f`, `fmap`) forms a functor from **Hask** to **func**.

For example, the **List** subcategory of **Hask** contains only list types `[a]` as objects, and functions of type `[a] -> [b]` as morphisms, where (`[]`, `map`) forms a functor from **Hask** to **List**.

The functor laws described before are just the axioms for functors in Category theory transcribed into Haskell![9]

Monads

Definition 7. A *monad* is an endofunctor $M \colon \mathbf{C} \to \mathbf{C}$, with two morphisms for each object X in $\mathrm{ob}\,(\mathbf{C})$,

$$\eta \colon X \to M(X),$$

and

$$\mu \colon M\,(M(X)) \to M(X),$$

such that the following axioms must also hold:[10]

$$\begin{aligned}
\mu \cdot M(\mu) &= \mu \cdot \mu, \\
\mu \cdot M(\eta) &= \mu \cdot \eta = \mathrm{id}_X, \\
\eta \cdot f &= M(f) \cdot \eta, \\
\mu \cdot M\,(M(f)) &= M(f) \cdot \mu,
\end{aligned}$$

where f is a morphism $f \colon A \to B$ for A and B in $\mathrm{ob}\,(\mathbf{C})$.

Definition 8. Monads can be formulated in **Hask** as follows:

```
class Functor m => Monad m where
  unit :: a -> m a
  join :: m (m a) -> m a
```

with $\eta =$ `unit` and $\mu =$ `join`.

Note that `unit` is identical to `pure` from our previous monad definition.

The monad axioms transcribed into **Hask**:

```
join . fmap join = join . join
join . fmap unit = join . unit = id
unit . g = fmap g . unit
join . fmap (fmap g) = fmap g . join
-- where g :: a -> b
```

THEOREM 1. *The function* `>>=` *is equivalent to* `fmap` *and* `join`*.*

---

[8] **Hask** is not a real category, due to `undefined` ($\perp$), but for our purposes this can be safely ignored [Danielsson et al. 2006].

[9] Functors preserving domains and codomains are guaranteed by the type of `fmap`.

[10] η and μ are usually interpreted as natural transformations instead of morphisms in other literature about Category theory.

PROOF. For equivalence, it is necessary to show that a function of the type of `>>=` can be constructed by an expression using only `fmap` and `join` and vice versa. Furthermore, it is also necessary to show that the axioms for monads in the category **Hask** imply the monad laws and vice versa.

PART 1. *`>>=` can be written in terms of `fmap` and `join`*[11].*

```
x >>= y = join (fmap y x)
```

PART 2. *The monad laws can be derived with the axioms for monads in the category* **Hask**.

1. *First monad law:*

```
pure x >>= g
= -- pure = unit
(unit x) >>= g
= -- definition of (>>=)
join (fmap g (unit x))
= -- composition
join ((fmap g . unit) x)
= -- third monad axiom
join ((unit . g) x)
= -- composition
(join . unit . g) x
= -- second monad axiom
(id . g) x
= -- id is left identity of composition
g x
```

2. *Second monad law:*

```
x >>= pure
= -- pure = unit
x >>= unit
= -- definition of (>>=)
join (fmap unit x)
= -- composition
(join . fmap unit) x
= -- second monad axiom
id x
= -- id is the identity morphism
x
```

3. *Third monad law:*

```
(x >>= g) >>= h
= -- definition of (>>=)
join (fmap h (x >>= g))
= -- definition of (>>=)
join (fmap h (join (fmap g x)))
= -- composition
join ((fmap h . join) (fmap g x))
= -- fourth monad axiom
join ((join . fmap (fmap h)) (fmap g x))
= -- composition
join ((join . fmap (fmap h) . fmap g) x)
= -- second functor law
join ((join . fmap (fmap h . g)) x)
= -- composition
(join . join . fmap (fmap h . g)) x
= -- first monad axiom
(join . fmap (join) . fmap (fmap h . g)) x
= -- second functor law
(join . fmap (join . fmap h . g)) x
= -- composition
join (fmap (join . fmap h . g) x)
= -- definition of (>>=)
x >>= (join . fmap h . g)
= -- construct lambda
x >>= (\v -> join (fmap h (g v)))
= -- definition of (>>=)
x >>= (\v -> g v >>= h)
```

PART 3. *`join` can be written in terms of `>>=`*.*

```
join x = x >>= id
```

`fmap` can be written in terms of `>>=`.*

```
fmap g x = x >>= (pure . g)
```

PART 4. *The monad axioms in the category* **Hask** *can be derived from the monad laws.*

1. *First monad axiom:*

```
(join . fmap join) x
= -- composition
join (fmap join x)
= -- definition of fmap
join (x >>= (pure . join))
= -- definition of join
(x >>= (pure . join)) >>= id
= -- third monad law
x >>= (\v -> (pure . join) v >>= id)
= -- composition
x >>= (\v -> pure (join v) >>= id)
= -- second monad law
x >>= (\v -> id (join v))
= -- id is the identity morphism
x >>= (\v -> join v)
= -- definition of join
x >>= (\v -> v >>= id)
= -- id is the identity morphism
x >>= (\v -> id v >>= id)
= -- third monad law
(x >>= id) >>= id
= -- definition of join
(join x) >>= id
= -- definition of join
join (join x)
= -- composition
(join . join) x
```

2. *Second monad axiom:*

```
(join . unit) x
= -- composition
join (unit x)
= -- definition of join
(unit x) >>= id
= -- pure = unit
(pure x) >>= id
= -- first monad law
id x

(join . fmap unit) x
= -- composition
join (fmap unit x)
= -- definition of fmap
join (x >>= (pure . unit))
= -- pure = unit
join (x >>= (pure . pure))
= -- definition of join
(x >>= (pure . pure)) >>= id
= -- third monad law
x >>= (\v -> (pure . pure) v >>= id)
= -- composition
x >>= (\v -> pure (pure v) >>= id)
= -- first monad law
```

[11] Equations marked with a * are presented again in the appendix with additional type annotations to aid the reader.

```
x >>= (\v -> id (pure v))
= -- id is the identity morphism
x >>= (\v -> pure v)
= -- deconstruct lambda
x >>= pure
= -- second monad law
x
= -- id is the identity morphism
id x
```

3. *Third monad axiom:*

```
(unit . g) x
= -- first monad law
pure x >>= (unit . g)
= -- pure = unit
pure x >>= (pure . g)
= -- definition of fmap
fmap g (pure x)
= -- pure = unit
fmap g (unit x)
= -- composition
(fmap g . unit) x
```

4. *Fourth monad axiom:*

```
(join . fmap (fmap g)) x
= -- composition
join (fmap (fmap g) x)
= -- definition of fmap
join (x >>= (pure . fmap g))
= -- definition of join
(x >>= (pure . fmap g)) >>= id
= -- third monad law
x >>= (\v -> (pure . fmap g) v >>= id)
= -- composition
x >>= (\v -> pure (fmap g v) >>= id)
= -- first monad law
x >>= (\v -> id (fmap g v))
= -- id is the identity morphism
x >>= (\v -> fmap g v)
= -- definition of fmap
x >>= (\v -> v >>= (pure . g))
= -- id is the identity morphism
x >>= (\v -> id v >>= (pure . g))
= -- third monad law
(x >>= id) >>= (pure . g)
= -- definition of join
(join x) >>= (pure . g)
= -- definition of fmap
fmap g (join x)
= -- composition
(fmap g . join) x
```

Thus, `>>=` sufficiently defines `fmap` and `join` and vice versa. □

Kleisli categories of Hask

Definition 9. For any subcategory of **Hask** with morphisms of type `a -> b`, define its *Kleisli category* to have the same objects, but morphisms of type `Monad m => a -> m b` where `m` is a monad. The identity morphisms are given by instantiating `unit :: a -> m a` with the appropriate type, and composition is given by*:

```
(<=<) :: Monad m =>
          (b -> m c) -> (a -> m b) -> a -> m c
f <=< g = join . fmap f . g
```

THEOREM 2. *`<=<` is a suitable composition operator for a Kleisli category.*

PROOF. For `<=<` to be considered a suitable composition operator, we must show that is is associative and has left and right identities.

PART 1. *`<=<` is associative:*

```
(f <=< (g <=< h)) x
= -- definition of <=<
(f <=< (join . fmap g . h)) x
= -- definition of <=<
(join . fmap f . (join . fmap g . h)) x
= -- composition
(join . fmap f) ((join . fmap g . h) x)
= -- composition
(join . fmap f) ((join . fmap g) (h x))
= -- composition
(join . fmap f) (join (fmap g (h x)))
= -- theorem 1
(join . fmap f) (h x >>= g)
= -- composition
join (fmap f (h x >>= g))
= -- theorem 1
(h x >>= g) >>= f
= -- third monad law
h x >>= (\v -> g v >>= f)
= -- theorem 1
h x >>= (\v -> join (fmap f (g v)))
= -- composition
h x >>= (\v -> (join . fmap f) (g v))
= -- composition
h x >>= (\v -> (join . fmap f . g) v)
= -- deconstruct lambda
h x >>= (join . fmap f . g)
= -- theorem 1
join (fmap (join . fmap f . g) (h x))
= -- composition
(join . fmap (join . fmap f . g)) (h x)
= -- composition
(join . fmap (join . fmap f . g) . h) x
= -- definition of <=<
((join . fmap f . g) <=< h) x
= -- definition of <=<
((f <=< g) <=< h) x
```

PART 2. *`<=<` has left identity:*

```
unit <=< f
= -- definition of <=<
join . fmap unit . f
= -- second monad axiom
id . f
= -- id is the identity morphism in Hask
f
```

PART 3. *`<=<` has right identity:*

```
(f <=< unit) x
= -- definition of <=<
(join . fmap f . unit) x
= -- composition
join . fmap f (unit x)
= -- composition
join (fmap f (unit x))
= -- theorem 1
unit x >>= f
= -- pure = unit
pure x >>= f
= -- first monad law
```

$f\ x$

Thus, `<=<` 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 `join` and `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:

```
instance Functor [] where
  fmap g [] = []
  fmap g (x:xs) = g x : fmap g xs

instance Applicative [] where
  pure x = [x]
  [] <*> _ = []
  (g:gs) <*> xs = (g <$> xs) ++ (gs <*> xs)

instance Monad [] where
  xs >>= h = concatMap h xs
```

The reader is encouraged to try to derive an equivalent expression which matches list comprehensions using only `fmap`/`<$>`, `pure`, `<*>` and `>>=` — 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

```
-- let f :: a -> b, xs :: [a], yss :: [[a]]
1. fmap f xs
2. [ [ f y | y <- ys ] | ys <- yss ]

-- solutions
1. [ f x | x <- xs ]
2. fmap (fmap f) yss
```

The functor instance allows lists to be mapped over.

Mapping multiple functions with multiple arguments over lists

```
-- let g :: a -> b -> c,
--     fs :: [a -> b], gs :: [a -> b -> c],
--     xs :: [a], ys :: [b]
1. [ g x y | x <- xs, y <- ys ]
2. fs <*> xs
3. [ g x y | g <- gs, x <- xs, y <- ys ]

-- solutions
1. g <$> xs <*> ys
2. [ f x | f <- fs, x <- xs ]
3. gs <*> xs <*> 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

```
-- let f :: a -> b, p :: a -> Bool, xs :: [a]
1. [ f x | x <- xs, p x ]
2. [ y | x <- xs, y <- f x ]
3. [1..] >>= (\x -> [1..x] >>= (\y -> pure (x, y)))

-- solutions
1. xs >>= (\x -> if p x then pure (f x) else [])
2. xs >>= f
3. [ (x, y) | x <- [1..], y <- [1..x] ]
```

Thus, monads allow the values drawn to depend on previously drawn values, and we can apply functions to values as they are drawn.

Choice

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

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)

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

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

Now we can take another look at this embodiment of 'choice' provided only by monads; consider the function of type:

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
```

which satisfies equations:

```
ifM (pure True) t e = t
ifM (pure False) t e = e
```

This function can be defined as follows:

```
ifM mx t e = mx >>= (\x -> if x then t else e)
```

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

```
ifA :: Applicative f => f Bool -> f a -> f a -> f a,
```

one might see that*

```
ifA mx t e =
  (\x y z -> if x then y else z) <$> mx <*> t <*> e
```

typechecks.

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

```
ifM (Just True) (Just ()) Nothing
= -- definition of ifM
(Just True) >>=
  (\x -> if x then (Just ()) else Nothing)
= -- definition of >>=
if True then (Just ()) else Nothing
= -- if expression True branch
Just ()

ifA (Just True) (Just ()) Nothing
= -- definition of ifA
```

```
(\x y z -> if x then y else z) <$>
  (Just True) <*> (Just ()) <*> Nothing
= -- <*> left associative
((\x y z -> if x then y else z) <$>
  (Just True) <*> (Just ())) <*> Nothing
= -- definition of <*>
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 `<*>`, 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.

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 `IO`, and that plenty of common - yet pure - code is monadic already.

Understanding monads is only the beginning, and their utility and elegance is only realised in Haskell by becoming fluent with instances of the `Monad` typeclass. When writing code that sequences actions 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:

Table 1:

Monad	Purpose
IO	Impure actions sequenced for effectful computation.
List	Models non-deterministic computation.
Maybe	Computations which may or may not succeed.
Writer	Collecting effects inside a monoid. Used for logging.
Reader	Allows values to be queried from state ('environment').
State	Similar to a combination of Writer and Reader.
ST	Like escapable IO; allows for arbitrary mutable state.
STM	Provides memory-safe concurrency via transactions.
Cont	Computations which can be interrupted and resumed.

APPENDIX

DO NOTATION

`do` notation is a syntax sugar for monadic sequences using `>>=`. 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:

```
do { a <- f ; m }   f >>= \a -> do { m }
-- bind f to a, proceed to desugar m

do { f ; m }   f >> do { m }
-- evaluate f, then proceed to desugar m

do { m }   m
```

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

TYPE ANNOTATIONS

```
x >>= y = join (fmap y x)
-- y :: a -> m b
-- (fmap y) :: m a -> m (m b)
-- (fmap y x) :: m (m b)
-- (join (fmap y x)) :: m b
join x = x >>= id
-- x :: m (m a)
-- (x >>=) :: (m a -> m b) -> m b
-- (x >>= id) :: m a
fmap g x = x >>= (pure . g)
-- x :: m a
-- g :: a -> b
-- (pure . g) :: a -> m b
-- (x >>= (pure . g)) :: m b
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
f <=< g = join . fmap f . g
-- (f) :: b -> m c
-- (g) :: a -> m b
-- (fmap f) :: m b -> m (m c)
-- (fmap f . g) :: a -> m (m c)
-- (join . fmap f . g) :: a -> m c
ifA mx t e =
  (\x y z -> if x then y else z) <$> mx <*> t <*> 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

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *ACM sigplan notices*. ACM, 206–217.

Gabriel Gonzalez. 2013. Comonads are objects. (2013). Retrieved October 6, 2016 from http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html

Martin Grabmüller. 2006. Monad transformers step by step. *Draft paper, October* (2006).

Paul Hudak. 2007. A regressive view of support for imperative programming in haskell. (2007). Retrieved October 5, 2016 from https://mail.haskell.org/pipermail/haskell-cafe/2007-August/030178.html

John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1 (2000), 67–111.

Edward Kmett. 2015. The free theorem for fmap. (2015). Retrieved September 26, 2016 from https://www.schoolofhaskell.com/user/edwardk/snippets/fmap

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (January 2008), 1–13. DOI:http://doi.org/10.1017/S0956796807006326

Dan Piponi. 2008. From monoids to monads. (2008). Retrieved October 6, 2016 from http://blog.sigfpe.com/2008/11/from-monoids-to-monads.html

Niklas Röjemo. 1995. *Garbage collection, and memory efficiency, in lazy functional languages*, Department of Computer Science, Chalmers University of Technology,

The Glasgow Haskell Team. 2016. Haskell ghc prelude source code. (2016). Retrieved October 5, 2016 from https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#line-1093

Philip Wadler. 1989. Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture*. ACM, 347–359.

Edward Z. Yang. 2012. Applicative functors. (2012). Retrieved September 8, 2016 from http://blog.ezyang.com/2012/08/applicative-functors

Brent Yorgey. 2012. Parsing context-sensitive languages with applicative. (2012). Retrieved September 27, 2016 from https://byorgey.wordpress.com/2012/01/05/parsing-context-sensitive-languages-with-applicative

Brent Yorgey. 2009. The typeclassopedia. *The Monad. Reader* 13 (2009), 17–68.

Bibliography

Steve Awodey. 2010. *Category theory*, OUP Oxford.
Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *ACM sigplan notices*. ACM, 206–217.
Stephen Diehl. 2013. Monads made difficult. (2013). Retrieved October 3, 2016 from
Yann Esposito. 2012. Category theory & programming. (2012). Retrieved September 26, 2016 from http://yogsototh.github.io/Category-Theory-Presentation/categories.html
Gabriel Gonzalez. 2013. Comonads are objects. (2013). Retrieved October 6, 2016 from http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html
Martin Grabmüller. 2006. Monad transformers step by step. *Draft paper, October* (2006).
Paul Hudak. 2007. A regressive view of support for imperative programming in haskell. (2007). Retrieved October 5, 2016 from https://mail.haskell.org/pipermail/haskell-cafe/2007-August/030178.html
John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1 (2000), 67–111.
Stefan Klinger. 2005. The haskell programmer's guide to the io monad — don't panic. (2005).
Edward Kmett. 2015. The free theorem for fmap. (2015). Retrieved September 26, 2016 from https://www.schoolofhaskell.com/user/edwardk/snippets/fmap
Miran Lipovaca. 2011. *Learn you a haskell for great good!: A beginner's guide*, no starch press.
Alfio Martini. 1999. *Elements of basic category theory*,
Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (January 2008), 1–13. DOI:http://doi.org/10.1017/S0956796807006326
Dan Piponi. 2008. From monoids to monads. (2008). Retrieved October 6, 2016 from http://blog.sigfpe.com/2008/11/from-monoids-to-monads.html
Dan Piponi. 2006a. Monads, a field guide. (2006). Retrieved October 2, 2016 from
Dan Piponi. 2006b. You could have invented monads! (And maybe you already have.). (2006). Retrieved September 8, 2016 from
Niklas Röjemo. 1995. *Garbage collection, and memory efficiency, in lazy functional languages*, Department of Computer Science, Chalmers University of Technology,
The Glasgow Haskell Team. 2016. Haskell ghc prelude source code. (2016). Retrieved October 5, 2016 from https://hackage.haskell.org/package/base-4.9.0.0/docs/src/GHC.Base.html#line-1093
Philip Wadler. 1995. Monads for functional programming. In *International school on advanced functional programming*. Springer, 24–52.
Philip Wadler. 1989. Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture*. ACM, 347–359.
Haskell Wiki. 2016. Do notation considered harmful. (2016). Retrieved October 5, 2016 from https://wiki.haskell.org/index.php?title=Do_notation_considered_harmful&oldid=60636
Wikibooks. 2016. Haskell/category theory. (2016). Retrieved September 26, 2016 from https://en.wikibooks.org/w/index.php?title=Haskell/Category_theory&oldid=3122870
Edward Z. Yang. 2012. Applicative functors. (2012). Retrieved September 8, 2016 from http://blog.ezyang.com/2012/08/applicative-functors
Brent Yorgey. 2012. Parsing context-sensitive languages with applicative. (2012). Retrieved September 27, 2016 from https://byorgey.wordpress.com/2012/01/05/parsing-context-sensitive-languages-with-applicative
Brent Yorgey. 2009. The typeclassopedia. *The Monad. Reader* 13 (2009), 17–68.