Stephen Diehl

Monads Made Difficult

This is a short, fast and analogy-free introduction to Haskell monads derived from a categorical perspective. This assumes you are familiar with Haskell and basic category theory.

If you aren't already comfortable with monads in Haskell, please don't read this. It will confuse your intuition even more.

Categories

We have a abstract **category** $\mathcal{C}$ which consists of objects and morphisms.

Objects : $\bullet$

Morphisms : $\bullet \to \bullet$

For each object there is an identity morphism `id` and a composition rule $(\circ)$ for combining morphisms associatively. We can model this with the following type class in Haskell with kind polymorphism.

```
-- Morphisms
type (a ~> b) c = c a b

class Category (c :: k -> k -> *) where
  id :: (a ~> a) c
  (.) :: (y ~> z) c -> (x ~> y) c -> (x ~> z) c
```

In Haskell we call this cateogry *Hask*, over the type constructor `(->)` of function types between Haskell types.

```
type Hask = (->)

instance Category Hask where
  id x = x
  (f . g) x = f (g x)
```

The constructor `(->)` is sometimes confusing to read in typeclass signatures as a typelevel operator since it's first argument usually appears to the left of it in infix form and the second to the right. For example the following are equivalent.

```
(->) ((->) a b) ((->) a c)
```

```
(a -> b) -> (a -> c)
```

Functors

Between two categories we can construct a **functor** denoted T, which maps between objects and morphisms of categories that preserves morphism composition and identities.

Objects : $T(\bullet)$

Morphisms : $T(\bullet \to \bullet)$

Represented in Haskell by:

```
class (Category c, Category d) => Functor c d t where
    fmap :: c a b -> d (t a) (t b)
```

With the familiar functors laws:

```
fmap id ≡ id
fmap (a . b) ≡ (fmap a) . (fmap b)
```

The identity functor $1_{\mathcal{C}}$ for a category $\mathcal{C}$ is a functor mapping all objects to themselves and all morphisms to themselves.

```
newtype Id a = Id a

instance Functor Hask Hask Id where
  fmap f (Id a) = Id (f a)

instance Functor Hask Hask [] where
  fmap f [] = []
  fmap f (x:xs) = f x : (fmap f xs)

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

An **endofunctor** is a functor from a category to itself, i.e. ($T : \mathcal{C} \to \mathcal{C}$).

```
type Endofunctor c t = Functor c c t
```

The composition of two functors is itself a functor as well. Convincing Haskell of this fact requires some trickery with constraint kinds and scoped type variables.

```
newtype FComp g f x = C { unC :: g (f x) }
newtype Hom (c :: * -> Constraint) a b = Hom (a -> b)

instance (Functor a b f, Functor b c g, c ~ Hom k) => Functor a c (FComp g f) where
  fmap f = (Hom C) . (fmapg (fmapf f) . (Hom unC))
    where
      fmapf = fmap :: a x y -> b (f x) (f y)
      fmapg = fmap :: b s t -> c (g s) (g t)
```

The repeated composition of an endofunctor over a category is written with exponential notation:

$$
\begin{aligned}
T^2 &= TT : \mathcal{C} \to \mathcal{C} \\
T^3 &= TTT : \mathcal{C} \to \mathcal{C}
\end{aligned}
$$

The category of small categories $\mathbf{Cat}$ is a category with categories as objects and functors as morphisms between categories.

Natural Transformations

For two functors F, G between two categories $\mathcal{A}, \mathcal{B}$:

$$
\begin{aligned}
F &: \mathcal{A} \to \mathcal{B} \\
G &: \mathcal{A} \to \mathcal{B}
\end{aligned}
$$

We can construct a mapping called a **natural transformation** η which is a mapping between functors $\eta : F \to G$ that associates every object X in $\mathcal{A}$ to a morphism in $\mathcal{B}$:

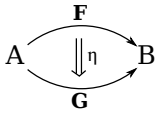
$$\eta_X : F(X) \to G(X)$$

Such that the following *naturality condition* holds for any morphism $f : X \to Y$. Shown as a *naturality square*:

$$\eta_Y \circ F(f) = G(f) \circ \eta_X$$

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \downarrow{\scriptstyle \eta_X} & & \downarrow{\scriptstyle \eta_Y} \\ G(X) & \xrightarrow{G(f)} & G(Y) \end{array}$$

The natural transformation itself is shown diagrammatically between two functors as:



This is expressible in our general category class as the following existential type:

```
type Nat c f g = forall a. c (f a) (g a)
```

In the case of *Hask* we a family of polymorphic functions with signature:

```
type NatHask f g = forall a. (f a) -> (g a)
```

With the naturality condition as the following law for a natural transformation (h), which happens to be a free theorem in Haskell's type system.

```
fmap f . h ≡ h . fmap f
```

The canonical example is the natural transformation between the List functor and the Maybe functor (where f = `List`, g = `Maybe`).

```
headMay :: forall a. [a] -> Maybe a
headMay []     = Nothing
headMay (x:xs) = Just x
```

Either way we chase the diagram we end up at the same place.

```
fmap f (headMay xs) ≡ headMay (fmap f xs)
```

Run through each of the cases of the naturality square for `headMay` if you need to convince yourself of this.

```
fmap f (headMay [])
= fmap f Nothing
= Nothing

headMay (fmap f [])
= headMay []
= Nothing
```

```
fmap f (headMay (x:xs))
= fmap f (Just x)
= Just (f x)

headMay (fmap f (x:xs))
= headMay [f x]
= Just (f x)
```

Functor Categories

A natural transformation $\eta : C \to D$ is itself a morphism in the **functor category** $\mathbf{Fun}(\mathcal{C}, \mathcal{D})$ of functors between $\mathcal{C}$ and $\mathcal{D}$. The category $\mathbf{End}$ is the category of endofunctors between a category and itself.

```
-- Functor category
newtype Fun f g a b = FNat (f a -> g b)

-- Endofunctor category
type End f = Fun f f

instance Category (End f) where
  id = FNat id
  (FNat f) . (FNat g) = FNat (f . g)
```

Monads

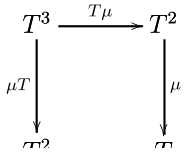
We can finally define a *monad* over a category $\mathcal{C}$ to be a triple (T, η, μ) of:

1. An endofunctor $T : \mathcal{C} \to \mathcal{C}$
2. A natural transformation $\eta : 1_{\mathcal{C}} \to T$
3. A natural transformation $\mu : T^2 \to T$

```
class Endofunctor c t => Monad c t where
  eta :: c a (t a)
  mu  :: c (t (t a)) (t a)
```

With an associativity square:

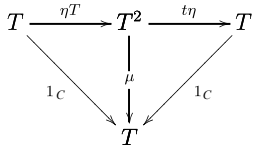
$$\mu \circ T\mu = \mu \circ \mu T$$



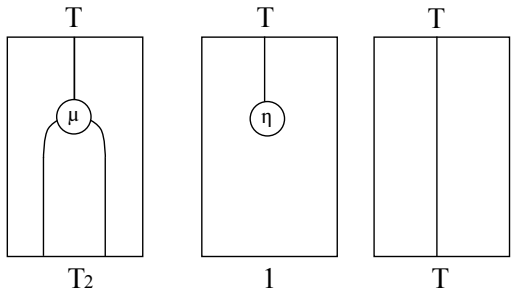
$$T \xrightarrow[\mu]{} T$$

And a triangle equality:

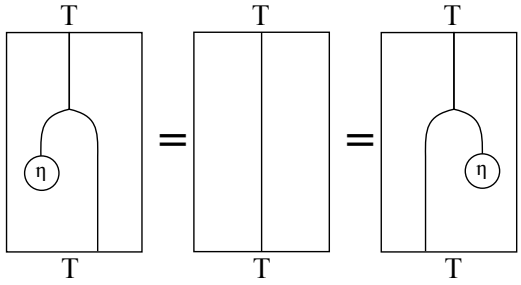
$$\mu \circ T\eta = \mu \circ \eta T = 1_C$$

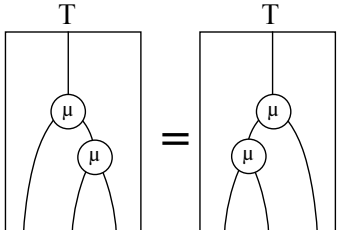


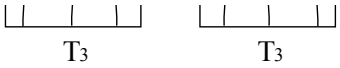
Alternatively we can express our triple as a series of **string diagrams** in which we invert the traditional commutative diagram of lines as morphism and objects as points and morphisms as points and objects as lines. In this form the monad laws have a nice geometric symmetry.



With the coherence conditions given diagrammatically:







Bind/Return Formulation

There is an equivalent formulations of monads in terms of two functions (`(>>=), return`) which can be written in terms of `mu, eta`)

In Haskell we define a bind `(>>=)` operator defined in terms of the natural transformations and `fmap` of the underlying functor. The `join` and `return` functions can be defined in terms of `mu` and `eta`.

```
(>>=) :: (Monad c t) => c a (t b) -> c (t a) (t b)
(>>=) f = mu . fmap f

return :: (Monad c t) => c a (t a)
return = eta
```

In this form equivalent naturality conditions for the monad's natural transformations give rise to the regular monad laws by substitution with our new definitions.

```
fmap f . return  ≡  return . f
fmap f . join    ≡  join . fmap (fmap f)
```

And the equivalent coherence conditions expressed in terms of bind and return are the well known Monad laws:

```
return a >>= f   ≡  f a
m >>= return     ≡  m
(m >>= f) >>= g  ≡  m >>= (\x -> f x >>= g)
```

Kleisli Category

The final result is given a monad we can form a new category called the *Kleisli category* from the monad. The objects are embedded in our original `c` category, but our arrows are now Kleisli arrows `a -> T b`. Given this class of "actions" we'd like to write an operator which combined these morphisms just like we combine functions in our host category.

$$(b \to Tc) \to (a \to Tb) \to (a \to Tc)$$
$$(b \to c) \to (a \to b) \to (a \to c)$$

In turns we out can for a specific function `(<=<)` expressed in terms of μ and the underlying functor, which gives associative composition operator of Kleisli arrows. The Kleisli category models "composition of actions" and form a very general model of computation.

The mapping between a Kleisli category formed from a category $\mathcal{C}$ is that:

1. Objects in the Kleisli category are objects from the underlying category.
2. Morphisms are Kleisli arrows of the form : $f : A \to TB$
3. Identity morphisms in the Kleisli category are precisely η in the underlying category.
4. Composition of morphisms $f \circ g$ in terms of the host category is defined by the mapping:

$$f \circ g = \mu(Tf)g$$

```
-- Kleisli category
newtype Kleisli c t a b = K (c a (t b))

-- Kleisli morphisms ( c a (t b) )
type (a :~> b) c t = Kleisli c t a b

-- Kleisli morphism composition
(<=<) :: (Monad c t) => c y (t z) -> c x (t y) -> c x (t z)
f <=< g = mu . fmap f . g

instance Monad c t => Category (Kleisli c t) where
  -- id :: (Monad c t) => c a (t a)
  id = K eta

  -- (.) :: (Monad c t) => c y (t z) -> c x (t y) -> c x (t z)
  (K f) . (K g) = K ( f <=< g )
```

In the case of Hask where `c = (->)` we see the usual instances:

```
-- Kleisli category
newtype Kleisli m a b = K (a -> m b)

-- Kleisli morphisms ( a -> m b )
type (a :~> b) m = Kleisli m a b

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

instance Monad m => Category (Kleisli m) where
  id            = K return
  (K f) . (K g) = K (f <=< g)
```

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

class Functor t => Monad t where
  eta :: a -> (t a)
  mu  :: t (t a) -> (t a)

(>>=) :: Monad t => t a -> (a -> t b) -> t b
ma >>= f = join . fmap f
```

Stated simply that the monad laws above are just the category laws in the Kleisli category, specifically the monad laws in terms of the Kleisli category of a monad `m` are:

```
(f >=> g) >=> h ≡ f >=> (g >=> h)
return >=> f ≡ f
f >=> return ≡  f
```

For example, `Just` is just an identity morphism in the Kleisli category of the `Maybe` monad.

```
Just >=> f = f
f >=> Just = f
```

```
just :: (a :~> a) Maybe
just = K Just
```

```
left :: forall a b. (a :~> b) Maybe -> (a :~> b) Maybe
left f = just . f

right :: forall a b. (a :~> b) Maybe -> (a :~> b) Maybe
right f = f . just
```

Haskell Monads

For instance the **List monad** would have:

1. η returns a singleton list from a single element.
2. μ turns a nested list into a flat list.
3. `fmap` applies a function over the elements of a list.

```
instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap f (x:xs) = f x : fmap f xs

instance Monad [] where
  -- eta :: a -> [a]
  eta x = [x]

  -- mu :: [[a]] -> [a]
  mu = concat
```

The **Maybe monad** would have:

1. η is the Just constructor.
2. μ combines two levels of Just constructors yielding the inner value or Nothing.
3. `fmap` applies a function under the Just constructor or nothing for Nothing.

```
instance Functor [] where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing

instance Monad Maybe where
  -- eta :: a -> Maybe a
  eta x = Just x

  -- mu :: (Maybe (Maybe a)) -> Maybe a
  mu (Just (Just x)) = Just x
  mu (Just Nothing) = Nothing
```

The **IO monad** would intuitively have the implementation:

1. η returns a pure value to a value within the context of the computation.
2. μ turns a sequence of IO operation into a single IO operation.
3. `fmap` applies a function over the result of the computation.

```
instance Functor IO where
  -- fmap :: (a -> b) -> IO a -> IO b
```

```haskell
instance Monad Maybe where
  -- eta :: a -> IO a
  -- mu :: (IO (IO a)) -> IO a
```

References

functor

natural transformation

monad

kleisli category

computational trinitarianism