

0.1 Definition 1–Using natural transformations

Let \mathcal{C} be a category.

A monad on \mathcal{C} consists of:

- a functor $T : \mathcal{C} \rightarrow \mathcal{C}$ (endofunctor)
- a natural transformation η called the unit:

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{id} & \mathcal{C} \\ \eta \Downarrow & & \\ \mathcal{C} & \xrightarrow{T} & \mathcal{C} \end{array}$$

- a natural transformation μ called multiplication:

$$\begin{array}{ccccc} \mathcal{C} & \xrightarrow{T} & \mathcal{C} & \xrightarrow{T} & \mathcal{C} \\ & & \mu \Downarrow & & \\ \mathcal{C} & \xrightarrow{\quad T \quad} & \mathcal{C} & & \end{array}$$

satisfying three properties:

- the right identity law

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ & \searrow id & \downarrow \mu \\ & & T \end{array}$$

- the left identity law

$$\begin{array}{ccc} T & \xrightarrow{T\eta} & T^2 \\ & \searrow id & \downarrow \mu \\ & & T \end{array}$$

- the associativity law

$$\begin{array}{ccc} T^3 & \xrightarrow{\mu T} & T^2 \\ T\mu \downarrow & & \downarrow \mu \\ T^2 & \xrightarrow{\mu} & T \end{array}$$

It is similar to a monoid, so the name is not a coincidence:

- set X
- element $e \in X$
- function $* : X^2 \rightarrow X$

Also satisfying three equations:

- $\forall x \in X. x * e = x$
- $\forall x \in X. e * x = x$
- $\forall x, y, z \in X. (x * y) * z = x * (y * z)$

0.2 Definition 2—As a Kleisli triple

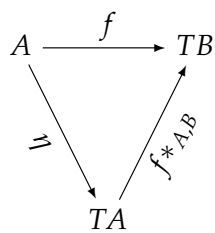
Let \mathcal{C} be a category.

A Kleisli triple on \mathcal{C} consists of:

- for each object A , we have an object TA and a morphism $\eta_A : A \rightarrow TA$.
- for objects A, B and morphism $f : A \rightarrow TB$, we have a morphism $f* : TA \rightarrow TB$.

such that the following equations are satisfied:

- left identity law:



analogous to:

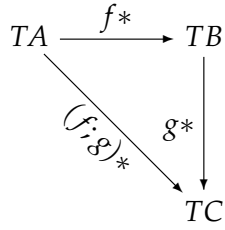
$a : A, f : A \rightarrow TB \vdash \text{return } a \gg = f = f a$
in Haskell.

- right identity law: $\eta_B *_{B,B} = id_{TB}$

analogous to:

$p : TB \vdash p \gg = \lambda x. \text{return } x = p : TB$
in Haskell.

- associativity law:



analogous to:

$p : TA, f : A \rightarrow TB, g : B \rightarrow TC$
 $\vdash (p >>= f) >>= g = p >>= (f >>= g)$
 in Haskell.

Even though the Kleisli triple makes no mention of naturality, it is equivalent to the previous definition given.

0.3 Example: Maybe and Exception monad

$Maybe\ X = X + 1 = \{inl x | x \in X\} \cup \{inr()\}$

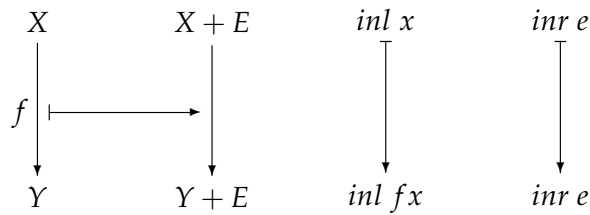
$Exc_E\ X = X + E$ where E is a set of behaviours.

The Exc (Exception monad) generalises Id (Identity monad) and $Maybe$ monad:

- $Id: E = \emptyset$
- $Maybe: E = 1 = \{()\}$

In **Set**, where morphisms are functions, let's check that Exc_E is indeed a monad.

1. First we map functions to Exc_E :

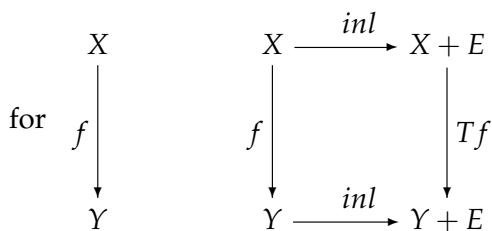


2. Unit at X (the return) is

$$X \longrightarrow X + E$$

$$x \longmapsto inl\ x$$

We can check that this is natural (as in definition 1):



therefore we have a single case

$$\begin{array}{ccc}
 x & \xrightarrow{inl} & inl\ x \\
 \downarrow f & & \downarrow Tf \\
 fx & \xrightarrow{inl} & inl\ fx
 \end{array} \text{ which does commute}$$

$$\begin{array}{ccc}
 X & \xrightarrow{\eta_X} & TX \\
 \downarrow f & & \downarrow Tf \\
 Y & \xrightarrow{\eta_Y} & TY
 \end{array}$$

so

3. Multiplication $(T^2X \xrightarrow{\mu_X} TX)$ μ is:

$$\mu_X : (X + E) + E \rightarrow X + E$$

$$inl\ inl\ x \mapsto inl\ x$$

$$inl\ inr\ e \mapsto inr\ e$$

$$inr\ e \mapsto inr\ e$$

Once again, we can check it is natural:

$$\begin{array}{ccccc}
 X & & (X + E) + E & \xrightarrow{\mu_X} & X + E \\
 \downarrow f & & \downarrow (f + E) & & \downarrow f + E \\
 Y & & (Y + E) + E & \xrightarrow{\mu_Y} & Y + E
 \end{array}$$

for

we have to check 3 cases:

$$\begin{array}{ccc}
 inl\ inl\ x & \xrightarrow{\mu_X} & inl\ x \\
 \downarrow f & & \downarrow f \\
 inl\ inl\ fx & \xrightarrow{\mu_X} & inl\ fx
 \end{array}$$

(a)

$$\begin{array}{ccc}
 inl\ inr\ e & \mapsto & inr\ e \\
 \downarrow & & \downarrow \\
 inl\ inr\ e & \mapsto & inr\ e
 \end{array}$$

(b)

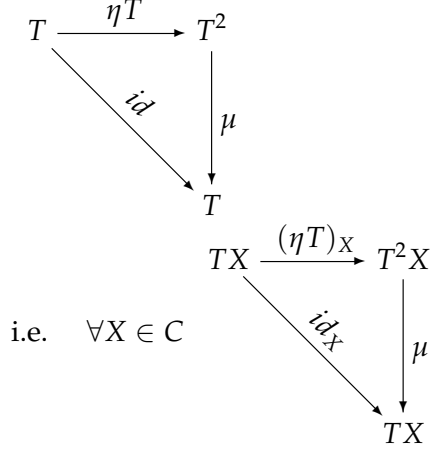
$$\begin{array}{ccc}
 inr\ e & \mapsto & inr\ e \\
 \downarrow & & \downarrow \\
 inr\ e & \mapsto & inr\ e
 \end{array}$$

(c)

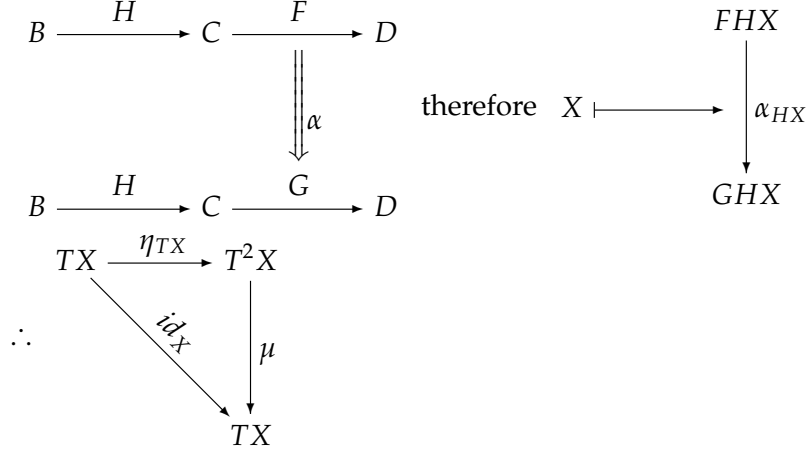
which all commute.

4. Finally, we have to check the three properties:

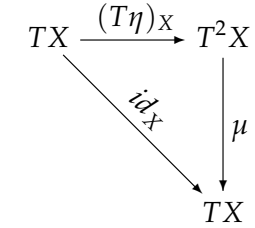
(a) left identity law:



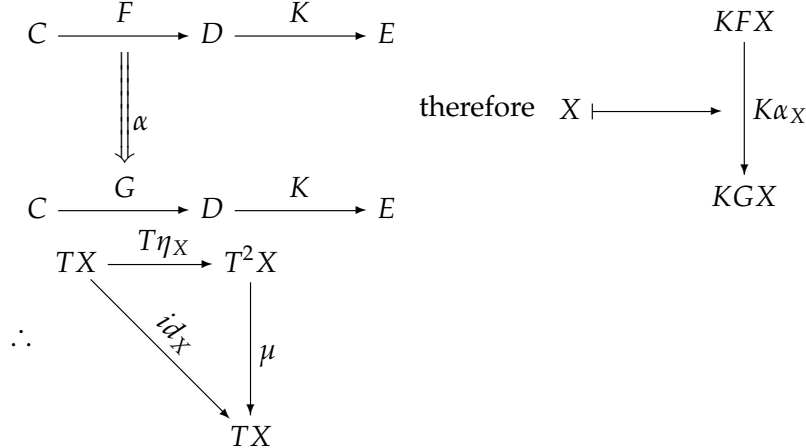
This can be defined using left-whiskering:



(b) right identity law:



This can be defined using right-whiskering:



(c) associativity law:

$$\begin{array}{ccc}
T^3X & \xrightarrow{\mu_{TX}} & T^2X \\
\downarrow T\mu_X & & \downarrow \mu_X \\
T^2X & \xrightarrow{\mu_X} & TX
\end{array}$$

Now the actual proof that the properties indeed hold:

- (a) There are two cases to show the left-identity law; $inl\ x$ and $inr\ e$:

$$\begin{array}{ccc}
inl\ x \xrightarrow{inl} inl\ inl\ x & & inr\ e \xrightarrow{inl} inl\ inr\ e \\
\searrow id \quad \downarrow \mu & \text{and} & \searrow id \quad \downarrow \mu \\
& inl\ x & inr\ e
\end{array}$$

- (b) There are two cases to show the right-identity law; $inl\ x$ and $inr\ e$:

$$\begin{array}{ccc}
inl\ x \xrightarrow{T\ inl} inl(inl\ x) & & inr\ e \xrightarrow{T\ inl} inl(inr\ e) \\
\searrow id \quad \downarrow \mu & \text{and} & \searrow id \quad \downarrow \mu \\
& inl\ x & inr\ e
\end{array}$$

- (c) There are four cases to show the associativity law; $inl\ inl\ inl\ x$, $inl\ inl\ inr\ e$, $inl\ inr\ e$, and $inr\ e$:

$$\begin{array}{ccc}
inl\ inl\ inl\ x \xrightarrow{\mu_{X+E}} inl\ inl\ x & & inl\ inl\ inr\ e \xrightarrow{\mu_{X+E}} inl\ inr\ e \\
\downarrow \mu_X + E \quad \downarrow \mu_X & & \downarrow \mu_X + E \quad \downarrow \mu_X \\
inl(inl\ x) \xrightarrow{\mu_X} inl\ x & & inl\ inl\ e \xrightarrow{\mu_X} inr\ e \\
\downarrow \mu_X + E \quad \downarrow \mu_X & & \downarrow \mu_X + E \quad \downarrow \mu_X \\
inl\ inl\ e \xrightarrow{\mu_{X+E}} inr\ e & & inr\ e \xrightarrow{\mu_{X+E}} inr\ e \\
\downarrow \mu_X + E \quad \downarrow \mu_X & & \downarrow \mu_X + E \quad \downarrow \mu_X \\
inl\ inr\ e \xrightarrow{\mu_X} inr\ e & & inr\ e \xrightarrow{\mu_X} inr\ e
\end{array}$$

With this, we have shown that the properties hold, and that Exc_E is indeed a monad.

0.4 Kleisli extension

Monads are many times implemented using the Kleisli extension. As with Haskell.

$$(-)^* : hom(X, TY) \rightarrow hom(TX, TY)$$

Given a monad $\langle T, \eta, \mu \rangle$ over category C and a morphism $f : X \rightarrow TY$:

$$f^* : TX \rightarrow TY$$

$$f^* = \mu_Y \circ Tf$$

In other words:

$f : X \rightarrow TY$ is any function that gives us the monad type in Haskell. This also matches η , which is equivalent to the `return` function.

$f^* : TX \rightarrow TY$ is analogous to the `bind` function in Haskell. The difference is that arguments are in different order.

For example, if we apply this to the Exc_E monad:

$$X \xrightarrow{f} Y + E$$

$$X + E \xrightarrow{f^*} Y + E$$

so

$$inl\ x \mapsto f\ x$$

$$inr\ e \mapsto inr\ e$$

0.5 Comparison with Haskell

Monads can be defined in the standard mathematical way or through the Kleisli triple. We can either have:

- $\langle T, \eta, \mu \rangle$, where T gives us the type of the monad, η is the `return` and μ is the `join`. Note that T is a functor, so in Haskell, this would be a pair $(T, fmap)$, which maps objects and functions.
- $\langle T, \eta, (-)^* \rangle$, where T gives us the type, again a pair with `fmap`, η is the `return`, and $(-)^*$ is equivalent to `bind` (`=>`).

This can be seen more clearly with the types:

`join` and μ (multiplication):

$$\text{join} :: T\ T\ A \rightarrow T\ A$$

$$\mu : TTA \rightarrow TA$$

`bind` and $(-)^*$ (Kleisli extension operator):

$$\text{>=} :: T\ A \rightarrow (A \rightarrow T\ B) \rightarrow T\ B$$

$$(-)^* : (A \rightarrow TB) \rightarrow (TA \rightarrow TB)$$

We can show that a Kleisli triple on C is equivalent to the standard mathematical definition of a monad on C because:

- μ and η give you $\gg=$, i.e. $(-)^*$
- $(-)^*$ and η give you `join`, i.e. μ

0.6 Exercise: State monad

Let S be a set (for some state, i.e. the type of the state/store).

We can define a monad on **Set** with $T A = S \rightarrow (S \times A)$

In programming, $T A$ would be a stateful program that returns a value of type A .

For instance, if $A = \mathbb{N}$, then $T \mathbb{N} = S \rightarrow (S \times \mathbb{N})$.

We could write a program that uses this state:

$$\lambda b : \text{bool}. (\text{not } b, b) : T \text{ bool}$$

This program would take the original state, invert it, and then output the original state.

With that context, we can define the monad either the standard way, or as a Kleisli triple. Defining it as a Kleisli triple:

Let *State* be the monad:

- $\text{State } X = S \rightarrow (S \times X)$
- The Unit (η)
 $\eta : X \rightarrow (S \rightarrow (S \times X))$
 $\eta = x \mapsto (s \mapsto (s, x))$ where $s : S$ and $x : X$
- The Kleisli operator (equivalent to bind):
 Given $g : X \rightarrow (S \rightarrow (S \times Y))$
 $g^* : (S \rightarrow (S \times X)) \rightarrow (S \rightarrow (S \times Y))$
 $g^* = f \mapsto s \mapsto (g(\pi_2 f s)) (\pi_1 f s)$

This is analogous to the Haskell state monad. Given *State* is analogous to our functor T :

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

return ::  a -> State s a
return x = State ( \s -> (x, s) )

(>>=) ::  State s a -> (a -> State s b) -> State s b
(State h) >>= f
= State ( \s -> let (a,new_sate) = h s in f a new_state )
```

Alternatively:


```
type State s a = s → (a,s)
return x = λ s → (x,s)
f >>= g   = λ s → case f s of (x,s') → g x s'
```