# Monads, Categories, and Computation

*Robert L. Bocchino Jr.*

Revised June 2024

**Monads** are widely used in functional programming (FP) languages as a way to assemble computations from sequences of sub-computations. The concept of a monad was originally developed in a branch of mathematics called **category theory**. However, the presentation of monads in the FP context differs from the standard category-theoretic presentation. Further, some of the category-theoretic motivation for the behavior of monads is lost in the standard FP presentation.

This paper explains the connection between FP monads and category-theoretic monads. It is hoped that this connection will provide additional motivation for and insight into the concept of a monad as it appears in programming languages. In particular, most FP discussions of monads emphasize the practical value of monads for programming, and they discuss the "monad laws" as an afterthought. This paper explains where the monad laws come from and why they exist, apart from their practical utility.

## 1. The Definition of a Monad

We start with the definition of a monad in an FP context. We will use Haskell-like notation, which is intentionally close to the category theory notation. However, the definition will differ slightly from the standard one given in Haskell. Later in the paper, we'll reconcile the two definitions.

A **monad** is a type $M\ t$ together with three functions that operate on this type. Here $t$ is a **type parameter**, also sometimes called a **generic parameter**. (In Scala, $M\ t$ would be written $M[T]$, in Java it would be written $M<T>$, and in Standard ML it would be written $'t\ M$.) The three functions are as follows:

1. A function of type $t \to M\ t$. We will call this function **inject**, because it injects a value of type $t$ into a monad of type $M\ t$. Haskell calls it **return**. Following standard notation in category theory, we will also refer to this function as $\eta_t$.
2. A function of type $(t \to t') \to (M\ t \to M\ t')$. We will call this function **map**, because it maps functions with non-monadic argument and return types to functions with monadic types. Haskell calls it **fmap**. Following standard notation in category theory, we will also refer to this function as $M$ (that is, we will give the map function the same name as the monad).
3. A function of type $M\ (M\ t) \to M\ t$. Following the mathematical notation, we will omit the parentheses and write $M\ M\ t \to M\ t$. We will call this function **flatten**, because it flattens two levels of monadic structure into one. Haskell calls it **join**. Following standard notation in category theory, we will also refer to this function as $\mu_t$.

These three functions must satisfy six combination laws, which we will explain below after introducing basic category theory.[1]

## 2. Examples of Monads

Here are several common examples of monads used in functional programming.

**The Option type or Maybe type.** The Option type (ML, Scala, Rust) or Maybe type (Haskell) is a type *Option t* that has the values *Some x* (for any value $x$ of type $t$) or *None*. Here are the three functions:

[1] Note that the concept of "a type with three functions" is expressed differently in different programming languages. In Haskell, one uses a type class. In Java, one uses an interface. In Scala, one uses a trait. In Standard ML, one uses a signature. Thus the details of how monads work vary from language to language. However, these details are below the level of abstraction that we will consider in this paper.

1. **Inject:** $x \mapsto \mathit{Some}\ x$.
2. **Map:** $f \mapsto \{\ \mathit{Some}\ x \mapsto \mathit{Some}\ f(x), \mathit{None} \mapsto \mathit{None}\ \}$.
3. **Flatten:** $\{\ \mathit{Some}\ (\mathit{Some}\ x) \mapsto \mathit{Some}\ x, \mathit{Some}\ \mathit{None} \mapsto \mathit{None}, \mathit{None} \mapsto \mathit{None}\ \}$.

As usual in functional programming, $p(X) \mapsto e(X)$ denotes the function $F$ that maps a pattern $p(X)$ to an expression $e(X)$, where $X = \{x_1, \ldots, x_n\}$ is a set of free variables. To evaluate $F(v)$, we (1) match the structure of $v$ with $p(X)$, deducing bindings $V = \{v_1, \ldots, v_n\}$ for the variables $X$; and (2) compute the value $e(V)$. This notation generalizes the mathematical notation $x \mapsto e(x)$, where $x$ is a single free variable, and $e(x)$ is an expression in that variable, as in the definition of *inject*.

In the definitions of *map* and *flatten*, the form $\{\, p_1(X_1) \mapsto e_1(X_1), p_2(X_2) \mapsto e_2(X_2), \ldots \}$ denotes the function that matches a value $v$ to one of the patterns $p_i(X_i)$ and then evaluates $F_i(v)$, where $F_i = p_i(X_i) \mapsto e_i(X_i)$. For example, *map* takes a function $f$ to a function $\mathit{map}(f)$, where we evaluate $\mathit{map}(f)(v)$ as follows:

- If $v$ has the form $\mathit{Some}\ v'$ for some value $v'$, then $\mathit{map}(f)(v)$ evaluates to $\mathit{Some}\ f(v')$.
- Otherwise $v = \mathit{None}$, and $\mathit{map}(f)(v)$ evaluates to $\mathit{None}$.

In functional programming languages, this kind of function is typically specified with alternative definitions, or with a single definition containing a *case* or *match* expression.

**The list type.** This is *List t*, the standard type of immutable lists. Here are the three functions:

1. **Inject:** $x \mapsto [x]$.
2. **Map:** $f \mapsto ([x, y, \ldots] \mapsto [f(x), f(y), \ldots])$.
3. **Flatten:** $[[x, \ldots], [y, \ldots], \ldots] \mapsto [x, \ldots, y, \ldots]$.

*inject* takes a value $x$ to the list $[x]$ containing just that value. *map* takes a function $f$ to the function that maps $f$ over each element of a list. *flatten* turns a list of lists $L$ into a list $L'$ by erasing the inner list structure.

**The state transformer type.** The type $M\ t$ is the type $s \to (t, s)$, where $s$ is a state type, and $t$ is a value type. The type represents functions that transform states of type $s$, producing a value of type $t$ in the process. Here are the three functions:

1. **Inject:** $x \mapsto (s \mapsto (x, s))$.
2. **Map:** $f \mapsto ((s \mapsto (x, s')) \mapsto (s \mapsto (f(x), s')))$.
3. **Flatten:** $(s \mapsto (f, s')) \mapsto (s \mapsto f(s'))$.

Here $x$ is a value of type $t$, and $s$ is a value of type $s$. *inject* takes a value $x$ to the state transformer that produces $x$, leaving the state $s$ unchanged. *map* takes a function $f$ to the function that applies $f$ to the result value of a state transformation. *flatten* propagates state transformations: it takes a state transformation $s \mapsto (f, s')$, where $f$ is another state transformation, and it yields the transformation $s \mapsto f(s')$.

### 3. Categories

A **category** formalizes the idea of a collection of objects with mappings (or arrows) between them. This concept is ubiquitous in mathematics and computer science. For example, in mathematics, the collection of all vector spaces (objects) and linear maps between them (arrows) is a category. In computer science, the collection of all types (objects) and functions from type to type (arrows) is a category.

Formally, a category $C$ is a directed graph consisting of objects and arrows. The graph must satisfy the following axioms.

**Existence of identity arrows.** For every object $c \in C$, there is a unique arrow $c \to c$ called the **identity arrow** or **identity** for $c$ and written $id_c$.

**Existence of composition.** For every triple of objects $c_1$, $c_2$, and $c_3$, and every pair of arrows $f_1 \colon c_1 \to c_2$ and $f_2 \colon c_2 \to c_3$, there is a unique arrow $c_1 \to c_3$. We call this arrow the **composition of $f_1$ and $f_2$** and write it $f_2 \circ f_1$. Note that following standard mathematical notation, the order of composition goes right to left: $g \circ f$ means "traverse $f$ and then traverse $g$." The reason is that arrows often represent functions, and functions appear in this order when they are applied in composition: $(g \circ f)(x) = g(f(x))$.

**Associativity of composition.** For every triple of composable arrows $f_1$, $f_2$, and $f_3$, the order of composition does not matter:

$$(f_3 \circ f_2) \circ f_1 = f_3 \circ (f_2 \circ f_1).$$

**Composition of identities.** For any function $f : c \to c'$, Composing an identity function with $f$ on the left or the right yields $f$:

$$id_{c'} \circ f = f \circ id_c = f.$$

It is easy to see that the examples given above (and many other examples, such as groups and group homomorphisms) satisfy these rules.

In the rest of this paper, let $P$ denote the **programming language category**: that is, the category with types $t$ as objects and functions $f : t \to t'$ as arrows.

## 4. The Monad Laws

Now we can present the six monad laws that we referred to in § 1.

### 4.1. $M$ Must Be a Functor from $P$ to $P$

In category theory, a **functor** $F$ is a mapping from the objects and arrows of one category $C$ to another category $C'$. $F$ takes objects $c$ of $C$ to objects $F\ c$ of $C'$, and it takes arrows $f$ of $C$ to arrows $F\ f$ of $C'$. In general, $C$ and $C'$ can be different categories. If they are the same ($C' = C$), then we say that $F$ is an **endofunctor**.

The first monad law is that $M$ must be an endofunctor from the programming language category $P$ to itself. The requirement that $M$ be a functor is the reason why we gave the *map* function of $M$ the name $M$ (and the reason why Haskell calls it *fmap* — the $f$ is for functor). The *map* function is the "arrow to arrow" part of $M$, interpreted as a functor.

In order for $M$ to be a functor, the following properties must hold:

1. For all types $t$, $M\ id_t = id_{M\ t}$.
2. For all pairs of composable arrows $f$ and $f'$, $M(f' \circ f) = (M\ f') \circ (M\ f)$.

It is easy to see that in all the examples given in § 2, these properties hold. For example, in the case of the option type:

1. $M\ id = \{\ Some\ x \mapsto Some\ id(x), None \mapsto None\ \} = id$.
2. $M(f' \circ f) = \{\ Some\ x \mapsto Some\ (f' \circ f)(x), None \mapsto None\ \}$.
3. $M\ f = \{\ Some\ x \mapsto Some\ f(x), None \mapsto None\ \}$.
4. $M\ f' = \{\ Some\ x \mapsto Some\ f'(x), None \mapsto None\ \}$.
5. The composition of the right-hand sides of (3) and (4) yields the right-hand side of (2).

Verifying the other examples is similar.

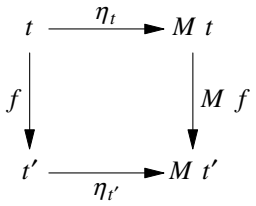
Finally, given that $M$ is a functor, it is straightforward to check that $M\ M$ and $M\ M\ M$ are also functors, where for any $t$, $M\ M\ t$ means $M(M\ t)$. We will need this fact below.

### 4.2. $\eta$ Must Be a Natural Transformation from the Identity Functor to $M$

In category theory, a **natural transformation** is a relationship between two functors $F$ and $F'$, both of which go from a category $C$ to another category $C'$. By convention, natural transformations are written with Greek letters, such as $\sigma$. For each object $c \in C$, $F$ and $F'$ in general map $c$ to two different objects $F\ c$ and $F'\ c$ in $C'$. So to specify a natural transformation $\sigma$ from $F$ to $F'$, we need to provide, for every $c \in C$, an arrow $\sigma_c : F\ c \to F'\ c$. The arrow $\sigma_c$ is called the $c$ **component** of the natural transformation $\sigma$. A natural transformation must also satisfy a commutativity relation (the "naturality") which we describe below.

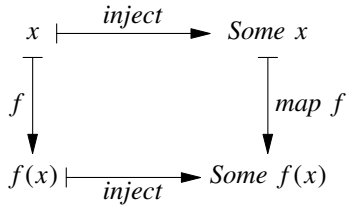
For any category $C$, the **identity functor** $I_C$ is the functor that takes each object of $C$ to itself and each arrow of $C$ to itself (that is, $I_C\ c = c$ and $I_C\ f = f$).

Recall that in the definition of a monad $M$, we specified for each type $t$ a function $\eta_t$. We want each $\eta_t$ to be the component of a natural transformation $\eta$ from $I_P$ to $M$, where we have already established (§ 4.1) that $M$ is a functor from $P$ to $P$. In order for the transformation $\eta$ to be natural, the following commutative diagram must be valid for each $t$, $t'$, and $f : t \to t'$:




Here we interpret traversal of successive arrows as composition. Commutativity means that all compositions that start and end at the same point in the diagram are identical. In this case, it means that $\eta_{t'} \circ f = (M\ f) \circ \eta_t$.

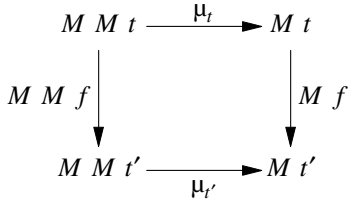
It is straightforward to check that for all the examples given in section § 2, $\eta$ is a natural transformation. For example, for the *Option* type, starting with a value $x$ in the upper left-hand corner, we can trace the flow of values around the diagram as follows:



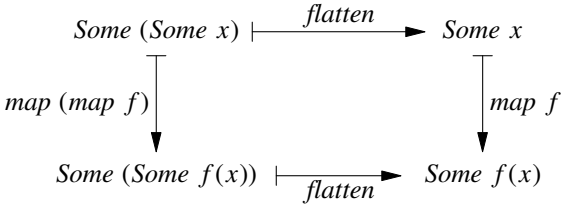

Then the commutativity is apparent. The other examples are similar.

### 4.3. μ Must Be a Natural Transformation from $M\ M$ to $M$

Recall that in the definition of a monad $M$, we specified for each type $t$ an arrow $\mu_t : M\ M\ t \to M\ t$. We want these arrows to be the components of a natural transformation from the functor $M\ M$ to the functor $M$. That is, we require the following diagram to commute for each $t$, $t'$, and $f : t \to t'$:




It is straightforward to check that for all the examples given in section § 2, μ is a natural transformation. For example, here is an object diagram for the *Option* type, starting with a value *Some* (*Some* $x$) in the upper left-hand corner:




The other examples are similar.

### 4.4. μ Must Satisfy an Associativity Law

Consider functions of the form $f : t \to M\ t'$. Call such functions **Kleisli functions** (after Heinrich Kleisli, who developed the theory of these functions). For any Kleisli function $f$, let $t_f$ and $t'_f$ be the types $t$ and $t'$ in the argument and return type of $f$. Call two Kleisli functions $f$ and $g$ **Kleisli composable** if $t'_f = t_g$. For example, $f : t_1 \to M\ t_2$ and $g : t_2 \to M\ t_3$ are Kleisli composable.

For any two Kleisli composable functions $f$ and $g$, define a **Kleisli composition operator** $\times$ as follows:

$$g \times f = \mu_{t'_g} \circ M\ g \circ f$$

(Here is the same definition annotated with expression types:

$$(g \times f)\colon t_f \to M\ t'_g = (\mu_{t'_g}\colon M\ M\ t'_g \to M\ t'_g) \circ ((M\ g)\colon M\ t_g \to M\ M\ t'_g) \circ (f\colon t_f \to M\ t'_f)$$

Try to convince yourself that the types are correct and that they compose correctly. Remember that $t'_f = t_g$. Working out the types in this way is often a useful exercise for understanding complex expressions in mathematics and functional programming.) Notice that $g \times f$ is again a Kleisli function, with $t_{g\times f} = t_f$ and $t'_{g\times f} = t'_g$.

We want Kleisli composition to be associative. Let $f$, $g$, and $h$ be Kleisli functions such that $(f, g)$ and $(g, h)$ are Kleisli composable pairs. Notice that $h \times (g \times f)$ and $(h \times g) \times f$ are legal Kleisli compositions. To show associativity, first expand $h \times (g \times f)$ as follows:

$$\begin{aligned} h \times (g \times f) &= h \times (\mu_{t'_g} \circ M\ g \circ f) \\ &= \mu_{t'_h} \circ M\ h \circ \mu_{t'_g} \circ M\ g \circ f \\ &= \mu_{t'_h} \circ M\ h \circ \mu_{t_h} \circ M\ g \circ f \\ &= \mu_{t'_h} \circ \mu_{M\ t'_h} \circ M\ M\ h \circ M\ g \circ f \end{aligned}$$

The second-to-last step is valid because $g$ and $h$ are Kleisli composable. The last step is valid because $\mu$ is a natural transformation from $M\ M$ to $M$. To see it, use the first commutative diagram in § 4.3, after applying the following substitutions: $f \mapsto h$, $t \mapsto t_h$, and $t' \mapsto M\ t'_h$.

Next, expand $(h \times g) \times f$:

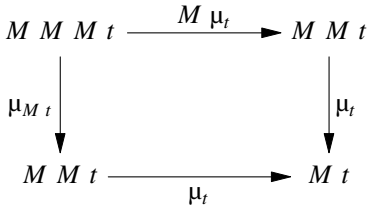
$$\begin{aligned} (h \times g) \times f &= \mu_{t'_{h\times g}} \circ M\ (h \times g) \circ f \\ &= \mu_{t'_h} \circ M\ (\mu_{t'_h} \circ M\ h \circ g) \circ f \\ &= \mu_{t'_h} \circ M\ \mu_{t'_h} \circ M\ M\ h \circ M\ g \circ f \end{aligned}$$

The last equation holds because $M$ is a functor.

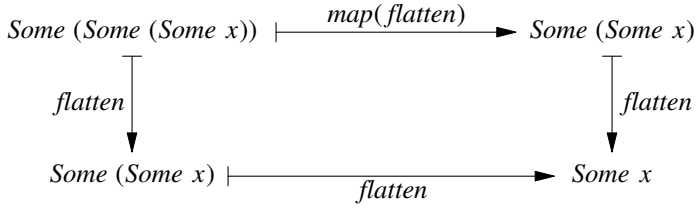
Comparing the two expansions above, we see that Kleisli composition is associative if the following holds for all $t$:

$$\mu_t \circ \mu_{M\ t} = \mu_t \circ M\ \mu_t$$

Written as a diagram, the rule looks like this:




Again, the verification of the examples in § 2 is straightforward. For example:



### 4.5. $\eta$ Must Be a Left Identity for Kleisli Composition

For each type $t$, $\eta_t : t \to M\ t$ is a Kleisli function. We want the following relation to hold for any Kleisli function $f$:

$$\eta_{t'_f} \times f = f$$

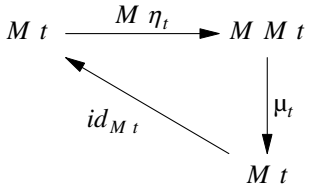
By definition we have this:

$$\eta_{t'_f} \times f = \mu_{t'_f} \circ M\ \eta_{t'_f} \circ f$$

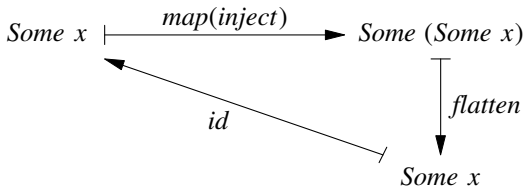
So we must require this for all $t$:

$$\mu_t \circ M\ \eta_t = id_{M\ t}$$

As a diagram:




Example of verification:




### 4.6. $\eta$ Must Be a Right Identity for Kleisli Composition

We want the following relation to hold for any Kleisli function $f$:

$$f \times \eta_{t_f} = f$$

By definition we have this:

$$f \times \eta_{t_f} = \mu_{t'_f} \circ M\ f \circ \eta_{t_f}$$

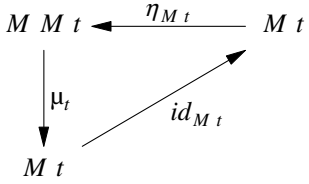
Because $\eta$ is a natural transformation (§ 4.2), we have this:

$$f \times \eta_{t_f} = \mu_{t'_f} \circ \eta_{M\ t'_f} \circ f$$

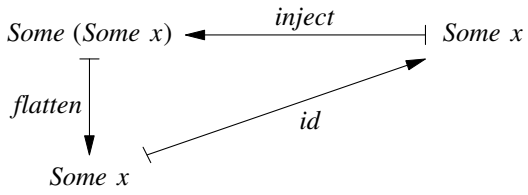
So we must require this for all $t$:

$$\mu_t \circ \eta_{M\ t} = id_{M\ t}$$

As a diagram:




Example of verification:




## 5. Monads and Computation

Monads are useful for programming because they provide an associative composition operator (Kleisli composition) that we can use to sequence computations.

First, let's look at Kleisli composition in a special case of a monad, the **identity monad** $I$, which we define as follows:

- $I\ t = t$.
- Each of the functions *inject*, *map*, and *flatten* is the identity function $id_t : t \to t$.

In computational terms, $I$ is the "no-op" monad: it is the unique monad whose operations all do nothing. If we plug this monad in to the definition of Kleisli composition, we see the following:

- Kleisli functions $f : t \to M\ t'$ become arbitrary functions $f : t \to t'$.
- Kleisli composition $g \times f$ becomes ordinary function composition $g \circ f$.

These observations provide some insight into the computational behavior of monads. Ordinary function composition provides a way to sequence functional computations. So does Kleisli composition. But Kleisli composition also provides some extra computation that we can specify by defining the monadic operations (type construction, *inject*, *map*, and *flatten*). If we specify that this extra computation does nothing (the identity monad), then we get back ordinary function composition. But by suitably defining a nontrivial monad $M$, we can use the extra computation associated with Kleisli composition in $M$ to perform side operations such as accumulating state, logging output, or aborting a computation when a *None* value occurs. There are at least two advantages to this style of programming:

1. The side computations are specified once and then applied many times, instead of being specified many times as they would be, for example, if the logging or error checking code were programmed directly.
2. The side computations are specified separately from the main computation, so the code is clearer and more modular than it would be if the side computation and the main computation were intermixed.

These benefits are well documented, with concrete examples, in the literature on languages such as Haskell and Scala that support programming with monads.

These observations may also provide insight into the idea of computation "inside a monad." Haskell programmers often refer to computations that occur "inside a monad" $M$. It is not immediately clear what this means, because monads are not necessarily containers that have computations (or anything else) "inside" them. It may mean that a computation is specified as a sequence of Kleisli functions $f : t \to M\ t'$ for some monad $M$. It may also mean that the results of computations have type $M\ t$, so they are hidden "inside" (or behind) the interface of $M\ t$, considered as an abstract data type, and may be used only in operations provided by that interface. Since a user typically specifies $t$, and a library typically specifies $M$, producing results of type $M\ t$ allows the library to restrict what the user can do with the values that it produces.

## 6. Haskell Monads

The discussion so far has followed the traditional approach in category theory. In Haskell, the standard definition of a monad looks slightly different. A monad is still a parameterized type $M\ t$. However, instead of functions *inject*, *map*, and *flatten*, a Haskell programmer typically provides the following functions when specifying a monad:

- A function $return : t \to M\ t$. This function is the equivalent of our *inject* function.
- A function $bind : M\ t \to (t \to M\ t') \to M\ t'$, written as an infix operator >>=. This function has no direct equivalent in our definition of a monad.

Notice that neither *map* nor *flatten* must be specified when defining a Haskell monad. However, every monad $M$ has a default implementation of *fmap* (equivalent to our *map*) and *join* (equivalent to our *flatten*). The default implementations use the functions *return* and *bind* provided by the programmer. We will now show that our definition of a monad and the Haskell definition are essentially equivalent.

### 6.1. Deriving *bind* from *map* and *flatten*

First, we show that given a monad $M$ as defined in § 1, we can use *map* and *flatten* to derive a Haskell-style *bind* function. As is often helpful in functional programming, we "follow the types." We have this:

- $map\colon (t \to t') \to (M\ t \to M\ t')$
- $flatten\colon M\ M\ t \to M\ t$

And we need this:

- $bind\colon M\ t \to (t \to M\ t') \to M\ t'$

This suggests that we do the following, assuming that $x\colon M\ t$ and $f\colon t \to M\ t'$ are the inputs to our *bind* function:

1. Apply *map* to $f$ to obtain a function $f'\colon M\ t \to M\ M\ t$.
2. Apply $f'$ to $x$ to obtain a value $x'\colon M\ M\ t$.
3. Apply *flatten* to $x'$ to obtain the result of type $M\ t'$.

In the category-theoretic notation from before, here is our definition of *bind*:

$$x \gg\!= f = (\mu_{t'} \circ M\ f)(x)$$

For example, if $x$ has type $List\ t$ and $f$ has type $t \to List\ t'$, then we compute $x \gg\!= f$ by (1) applying $f$ to each element of $x$ and (2) flattening the resulting list of lists into a list.

Notice that if $x = g(x')$ for some $x' \in t$ and $g\colon t \to M\ t$, then we have

$$\begin{aligned} x \gg\!= f &= (\mu_{t'} \circ M\ f)(g(x')) \\ &= (\mu_{t'} \circ M\ f \circ g)(x') \\ &= (f \times g)(x') \end{aligned}$$

So we have established the following identity:

$$(g \times f)(x) = f(x) \gg\!= g$$

We see that *bind* is the remainder of a Kleisli composition after applying the first function in the composition. Here is one expansion in terms of *bind* for a composition of three Kleisli functions $f$, $g$, and $h$:

$$\begin{aligned} (h \times g \times f)(x) &= f(x) \gg\!= (h \times g) \\ &= f(x) \gg\!= (x \mapsto (h \times g)(x)) \\ &= f(x) \gg\!= (x \mapsto (g(x) \gg\!= h)) \end{aligned}$$

This expansion is often used in Haskell. Because Kleisli composition is associative, this expansion is also valid:

$$\begin{aligned} (h \times g \times f)(x) &= (g \times f)(x) \gg\!= h \\ &= (f(x) \gg\!= g) \gg\!= h \end{aligned}$$

The equivalence of these expansions gives rise to one of the standard monad laws in Haskell (see § 6.4).

I assume that Haskell programmers prefer using *bind* to using Kleisli composition directly because it is more natural for specifying computations. For example, in both expansions, the order of function application is left to right, instead of right to left as it is in Kleisli composition. Also, in the first expansion, the first function in the composition, $f$, is at the outermost layer of parentheses. The second expansion is more compact; but in that expansion, $f$ is at the innermost layer.

It is also useful to observe that *bind* is a kind of **Kleisli application**. Just as Kleisli composition provides a way to compose two Kleisli functions $f: t \to M\ t'$ and $g: t' \to M\ t''$ to produce a function $g \times f: t \to M\ t''$, *bind* provides a way to apply a function $f: t \to M\ t'$ to a value $x: M\ t$ and produce a value $(x \gg\!= f): M\ t'$. Indeed, if we let $M$ be the identity monad in the definition of *bind*, we get $x \gg\!= f = f(x)$, i.e., *bind* becomes ordinary function application in this case. So $x \gg\!= f$ is to $f(x)$ as $g \times f$ is to $g \circ f$. This is why $\gg\!=$ is called *bind*: $x \gg\!= f$ binds $x$ to the argument of $f$ according to Kleisli application.

### 6.2. Deriving *flatten* from *bind*

Now we go in the other direction. First we show that given a Haskell-style *bind* function, we can derive our *flatten* function. Again we follow the types. We are given

$$bind: M\ t \to (t \to M\ t') \to M\ t'$$

for any $t$ and $t'$, and we want to derive

$$flatten: M\ M\ t \to M\ t$$

for any $t$. If we put $t = M\ t'$ in the definition of *bind*, and rename $t'$ to $t$, then we have

$$bind: M\ M\ t \to (M\ t \to M\ t) \to M\ t.$$

This has the type we want, except for the function argument of type $M\ t \to M\ t$. The obvious choice for that argument is the identity function $id_{M\ t}$. Indeed, if we put $id_{M\ t}$ in for $f$ in the definition of *bind* in § 6.1, then since $M$ is a functor and so $M\ id = id$, the right-hand side reduces to $\mu_t(x)$, which is what we wanted. So we have established the following:

$$\mu_t(x) = x \gg\!= id_{M\ t}$$

(Notice that $id_{M\ t}: M\ t \to M\ t$ is a Kleisli function $f: t_1 \to M\ t_2$ with $t_1 = M\ t$ and $t_2 = t$.)

### 6.3. Deriving *map* from *inject* and *bind*

Next we show how to use *inject* and *bind* to derive *map*. We are given

$$inject: t \to M\ t$$

$$bind: M\ t \to (t \to M\ t') \to M\ t'$$

We want to derive

$$map: (t \to t') \to (M\ t \to M\ t')$$

Assuming that $f$ is the input to *map*, we do the following:

1. Compose $f$ with *inject* to compute a function $f': t \to M\ t'$.
2. Construct the function $g$ that, for any argument $x: M\ t$, returns $x \gg\!= f'$.
3. Return $g$ as the value of $map(f)$.

Thus, we have the following provisional definition of *map*, written $M'$:

$$(M'\ f)(x) = x \gg\!= (\eta_{t'} \circ f)$$

It remains to prove that this definition is correct; we should have $M' = M$. Expanding the right-hand side according to the definition of *bind* in § 6.1 yields the following:

$$\begin{aligned} M'\ f &= \mu_{t'} \circ M(\eta_{t'} \circ f) \\ &= \mu_{t'} \circ M\ \eta_{t'} \circ M\ f \\ &= M\ f \end{aligned}$$

The second line holds because $M$ is a functor, and the third line holds by the left identity law (§ 4.5). This proves that the definition of *map* in terms of *inject* and *bind* is correct.

### 6.4. The Monad Laws in Haskell

In Haskell, it is common to write the monad laws as follows, for all $x: t$, $f: t \to M\ t'$, and $g: t' \to M\ t''$:

1. $(\eta_t(x) \gg\!= f) = f(x)$
2. $(x \gg\!= \eta_t) = x$
3. $((x \gg\!= f) \gg\!= g) = (x \gg\!= (x' \mapsto (f(x') \gg\!= g))$

It is straightforward to derive these identities from the previous discussion:

1. Rewrite the left-hand side as $(f \times \eta_t)(x)$ and observe that $\eta_t$ is a right identity for $\times$.
2. Rewrite the left-hand side as $(\mu_t \circ M\ \eta_t)(x)$ and use the left-identity monad rule (§ 4.5).
3. Use the two expansions of Kleisli composition in terms of *bind* given in § 6.1.

So we see that with respect to Kleisli composition, (1) is the right identity rule, (2) is the left identity rule, and (3) is the associativity rule, all written using $\gg\!=$ instead of $\times$.

When $M$ is the identity monad, so $(x \gg\!= f) = f(x)$ and $\eta_t = id_t$, the identities become the following:

1. $f(id_t(x)) = f(x)$
2. $id_t(x) = x$
3. $g(f(x)) = (x' \mapsto g(f(x'))(x)$

These identities are obviously true. The Haskell monad laws emphasize that these identities must hold in an arbitrary monad, when we replace $id_t$ with $\eta_t$ and function application with *bind*.

We can also write the third Haskell rule as

$$((x \gg\!= f) \gg\!= g) = (x \gg\!= (g \times f))$$

in general and

$$g(f(x)) = (g \circ f)(x)$$

in the identity case. This is nice because it illustrates in one formula that *bind* (a.k.a. Kleisli application) is analogous to function application, and Kleisli composition is analogous to function composition. It also states more clearly the essence of the rule: that applying $f$ and then applying $g$ should produce the same result as applying the composition of $f$ and $g$.

## 7. The Kleisli Category

We can use category theory to formalize the relationship between Kleisli and ordinary application and composition. Given the programming language category $P$ and a monad $M$ in $P$, we can construct a new category $K$ called the **Kleisli category**. Here is the construction:

- The objects $k$ of $K$ are in one-to-one correspondence with the objects $M\ t_k$ in $P$, where $t_k$ is any type in $P$.
- For each pair of objects $k$ and $k'$ in $K$, the arrows $f: k \to k'$ in $K$ are in one-to-one correspondence with the functions $f_P: t_k \to M\ t'_k$ in $P$. Each arrow $f$ in $K$ is itself a function with type $k \to k'$ in $K$: for each $x \in k = M\ t_k$, it maps $x$ to $(x \gg\!= f_P) \in k' = M\ _{k'}$.
- For each $k$ in $K$, the identity arrow $id_k: k \to k$ in $K$ is the arrow corresponding to $\eta_{k_t}: t_k \to M\ t_k$ in $P$.
- For each pair of arrows $f: k_1 \to k_2$ and $g: k_2 \to k_3$ in $K$, the composite arrow $g \circ f$ in $K$ corresponds to the function $g_P \times f_P: t_{k_1} \to M\ t_{k_3}$ in $P$.

Because Kleisli composition $\times$ is associative and $\eta$ is a left and right identity for it, $K$ is a category.

The Kleisli category may provide additional insight into the idea of computation "in a monad." When we compute "in a monad" $M$, we are computing in the Kleisli category $K$ associated with $M$, applying functions $f: k \to k' = f_P: t_k \to M\ t_{k'}$ to values $x: k = M\ t_k$, and composing functions $f: k \to k' = f_P: t_k \to M\ t_{k'}$ and $g: k' \to k'' = f_P: t_{k'} \to M\ t_{k''}$ to construct larger computations.

We can think of the category $K$ as a new functional programming language, defined by the monad $M$ in the base language, such that function application in $K$ is defined by $\gg\!=$ in $P$, and function composition in $K$ is defined by $\times$ in $P$. Thus $K$ is useful for constructing embedded domain-specific languages inside languages like Haskell that support programming with monads.

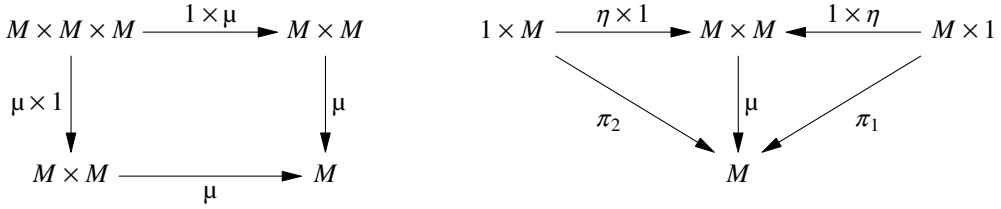
## 8. Why Is It Called a Monad?

Finally, we address the reason for the name "monad," which considered in isolation is an odd name for a programming language construct. The name comes from the association between a monad and the mathematical concept of a **monoid**. Both monad and monoid share the same root (the Greek word *monos*). The word monad also appears in philosophy and biology, where it means a simple or indivisible entity.

In mathematics, a monoid is a set together with a binary operation that is associative and has an identity element. For example, the integers under addition form a monoid with zero as the identity element, and the integers under multiplication form a monoid with 1 as the identity element. In category theory, there is a standard formulation of a monoid: a monoid is a set $M$ together with two functions

$$\mu\colon M \times M \to M \qquad \eta\colon 1 \to M$$

where $\times$ denotes the Cartesian product of sets, and 1 denotes the one-point set $\{0\}$. $\mu$ is the binary operation, and the $\eta$ selects an element of $M$, $\eta(0)$, to be the identity element of $M$. For $M$ to be a monoid, the following diagrams must commute:



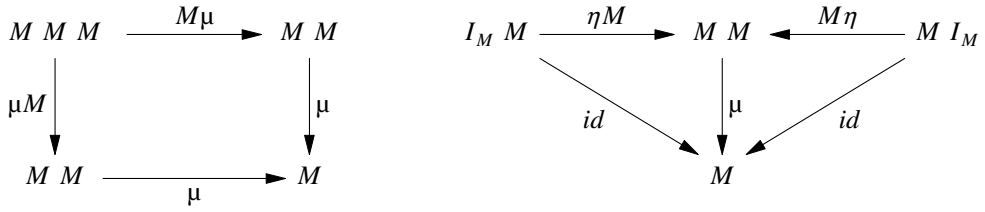

The first diagram states that $\mu$ is associative, i.e., for all $x$, $y$, and $z \in M$, $\mu(\mu(x, y), z) = \mu(x, (\mu(y, z))$. It uses the following notation:

- 1 denotes the identity function $M \to M$.
- $1 \times \mu\colon M \times (M \times M) \to M \times M$ takes $(x, (y, z))$ to $(x, \mu(y, z))$
- $\mu \times 1\colon (M \times M) \times M \to M$ takes $((x, y), z)$ to $(\mu(x, y), z)$.

The second diagram states that $\eta$ is a left and right identity for $M$, i.e., for all $x \in M$, $\mu(\eta(0), x) = \mu(x, \eta(0)) = x$. It uses the following notation:

- $\eta \times 1\colon 1 \times M \to M \times M$ takes $(0, x)$ to $(\eta(0), x)$.
- $1 \times \eta\colon M \times 1 \to M \times M$ takes $(x, 0)$ to $(x, \eta(0))$.
- $\pi_2\colon 1 \times M \to M$ (projection onto the second element) takes $(0, x)$ to $x$.
- $\pi_1\colon M \times 1 \to M$ (projection onto the first element) takes $(x, 0)$ to $x$.

These diagrams are similar to the ones shown in § 4.4 through § 4.6 in connection with the associativity and identity laws of a monad. We can rewrite those diagrams as follows:




These diagrams use the following notation:

- The nodes are functors, and the arrows are natural transformations.
- For each $t$, the natural transformation $\mu M$ has component $(\mu M)_t = \mu_{M\,t}$, and similarly for $\eta M$.
- For each $t$, the natural transformation $M\mu$ has component $(M\mu)_t = M\ \mu_t$, and similarly for $M\eta$.

The meaning of these diagrams is just that the diagrams in § 4.4 through § 4.6 must hold for each type $t$.

Notice that these diagrams are similar to the corresponding diagrams for monoids given above. The differences are as follows:

- Where the monoid diagrams use the Cartesian product of sets $M \times M$, the monad diagrams use the composition of functors $M\ M$.
- Where the monoid diagrams use the set 1, the monad diagrams use the identity functor $I_M$.
- Where the monoid diagrams use the Cartesian product of functions $1 \times$ or $\times 1$, the monad diagrams use the composition of natural transformations with $M$.
- Where the monoid diagrams use the projection functions $\pi$, the monad diagrams use the identity natural transformation *id* from $M$ to $M$.

The similarity between the diagrams motivates the name monad.

## 9. Conclusion

We have presented some basic category theory, and we have defined monads in the standard category-theoretic way. We have discussed programming language monads and how they relate to category-theoretic monads. We have discussed the relationship between monads and monoids in mathematics, and the motivation for the name "monad."

It is hoped that this discussion has provided some additional insight into the definition and behavior of a monad, apart from the usual assertions in programming language texts that the monad laws "just make sense" or "are required to make Haskell *do* blocks work as expected" (both of which, of course, are true).

## References

Hutton, G. *Programming in Haskell.* 2d ed. Cambridge University Press, 2016.

Mac Lane, S. *Categories for the Working Mathematician.* 2d ed. Springer, 1997.

https://en.wikibooks.org/wiki/Haskell/Category_theory

https://wiki.haskell.org/All_About_Monads