1 Introduction

In this note I want to connect monads in category theory with monads in functional programming. The starting point is the category theoretic claim by Mac Lane that

"A monad in $\mathcal{C}$ is a monoid in the category of endofunctors of $\mathcal{C}$."

- What are the objects and morphisms in this category?
- How does this particular "monoid" provide an associative binary operator and an identity element?
- How does the monoidal construction translate to functional programming (e.g. Haskell, OCaml) monads (with `return` and `>>=`) and satisfy the corresponding rules?

2 Defining the monad

A monad needs a single object: the endofunctor T. By definition, T maps some underlying category $\mathcal{C}$ to itself. As Mac Lane claims, T belongs to the so-called category of endofunctors of $\mathcal{C}$, where endofunctors like T are the objects and morphisms are the natural transformations between them. As a useful example for later, if T is an endofunctor, then $T^2, T^3 \dots$ are also endofunctors and we might be able to go between them. For monads, the two important natural transformations are:

- $\eta : 1_C \to T$
- $\mu : T^2 \to T$

Thus a monad is properly a triple $\langle T, \eta, \mu \rangle$, or just T for short. The properties that η and μ satisfy are precisely those that qualify T as a monoid, albeit slightly informally.

3 Why are monads also monoids?

As a category, each monoid M contains an identity element I and an associative morphism op from $M \times M$ to M. In abstract algebra, M is just a set of elements, $I \in M$, and no morphisms (unary functions) need to be defined on M[1].

In our case, replace M with T and let op map two endofunctors to another endofunctor. For convenience, we allow η as an argument by identifying it with T, since $\eta(1_C(X)) = T$. However, bear in mind that η maps only objects but not morphisms, unlike T. Thus, $\eta(T) = \eta \circ T$, and $T(\eta) \neq \eta(T)$ *a priori*.

For appropriate endofunctors T_1, T_2 (possibly including η) such that $T_1 T_2 = T^n$, we define $op(T_1, T_2) := \mu \circ T_1(T_2)$, and the output is another endofunctor that sends $T_1 T_2(X)$ to $\mu_{T^{n-2}}(X)$, for some object X in $\mathcal{C}$. The notation means that μ leaves the innermost $n-2$ layers untouched.

Informally, we want η to be our (two-sided) identity. This means η must satisfy the following:

$$\begin{aligned} op(T, \eta) &= \mu \circ T(\eta)^{(*)} && = T && {}^*T \text{ transforms } \eta \\ op(\eta, T) &= \mu \circ \eta(T) = \mu \circ \eta_T^{(*)} && = T && {}^*\eta_T = \eta(T) \text{ acts on } TX \end{aligned}$$

What does it mean for op to be associative? First, note that $op(T, T) = \mu \circ T^2$. Now, we have to show that the following are equal:

$$\begin{aligned} op(T, op(T, T)) &= \mu \circ T(\mu \circ T^2) && =^{(*)} \mu \circ T(\mu) \circ T[T^2] && {}^*\text{Functor } T \text{ is distributive} \\ op(op(T, T), T) &= \mu \circ (\mu \circ T^2)T && = \mu \circ \mu_T^{(*)} \circ T^2[T] && {}^*\mu_T = \mu(T) \text{ acts on } TX \end{aligned}$$

[1]Alternatively, we can imagine a dummy object with elements as endomorphisms, and composition of morphisms given by op. However, it might be less confusing to let objects be objects.

Translating these two monoid laws into category theory, i.e. into commutative diagrams, we have

Identity	Associativity
$\begin{tikzcd} T[X] = [TX] \arrow[r, "\eta_{TX}"] \arrow[d, "T(\eta_X)"] \arrow[dr, equal] & T^2X \arrow[d, "\mu_X"] \\ T[TX] \arrow[r, "\mu_X"] & TX \end{tikzcd}$	$\begin{tikzcd} T[T^2X] = T^2[TX] \arrow[r, "\mu_{TX}"] \arrow[d, "T(\mu_X)"] & T[TX] \arrow[d, "\mu_X"] \\ T[TX] \arrow[r, "\mu_X"] & TX \end{tikzcd}$

These diagrams will be used to provide the mapping to functional programming.

4 Kleisli Category

The properties of T, η, and μ are useful because they can transform the underlying category $\mathcal{C}$ into a new category $\mathcal{C}_T$ known as the **Kleisli category**. While not originally developed for functional programming (see Section 6), the Kleisli category augments $\mathcal{C}$ much like side effects augment computation. Thus, it is the final step before we jump from category-theoretic monads to real-life monads.

In short, $\mathcal{C}_T$ wraps everything in $\mathcal{C}$ with T. Objects are wrapped (e.g. $x \mapsto Tx$) by morphisms that produce wrapped objects (e.g. $f : X \to TY$). With the help of μ, we can compose $f : X \to TY$ and $g : Y \to TZ$ as follows:

$$g \circ f = \mu_Z \circ Tg \circ f$$

Furthermore, we define the identity morphism for each object X to be $\eta_X : X \to TX$, the component-wise morphism of the natural transformation η.

The following diagrams summarizes the interaction between different objects and morphisms. The solid line shows how we can chain evaluations together, whereas the up-arrows labelled as η_Y and η_Z are dashed because the commutativity of the square will only be established later.

$$\begin{tikzcd} X \arrow[r, "f"] & TY \arrow[r, "Tg"] & T^2Z \arrow[d, "\mu_Z"] & \\ & Y \arrow[u, dashed, "\eta_Y"] \arrow[r, "g"] & TZ \arrow[r] & \dots \\ & & Z \arrow[u, dashed, "\eta_Z"] \arrow[r] & \dots \end{tikzcd}$$

Strictly speaking, $\mathcal{C}_T$ should share the same objects as $\mathcal{C}$ (i.e. the leftmost diagonal) since the domain of every morphism is some unwrapped X. Thus, we have to pretend that TX and X are the same, much like algebraic embeddings (e.g. $\mathbb{Z} \hookrightarrow \mathbb{Q}$). Besides, we could let morphisms be morphisms and not worry about changing the codomain of a "function". The most important thing is the commutative diagrams.

5 Functional Programming

With the Kleisli category defined, we now interpret $\mathcal{C}$ as the cartesian-closed category that underpins lambda calclus, $x, y, z \dots$ as variables of types (i.e. objects) $X, Y, Z \dots$, the endofunctor T as both the object and function wrapper, and morphisms $f : X \to TY, g : Y \to TZ \dots$ as well-defined functions.

Then, with η and μ, we will define the corresponding monad functions as follows, with signature/ declaration on the left (using ML notation `'a t` and `'b t` for arbitrary types TX) and definitions in (pseudo-) lambda calculus on the right:

- `return:  'a` $\rightarrow$ `'a t` $= \lambda$`x.`$\eta_{\mathtt{a}}$`x`[2]
- `>>=:  'a t` $\rightarrow$ `('a` $\rightarrow$ `'b t)` $\rightarrow$ `'b t` $= \lambda$`Tx.(`λ`f.(`$\mu_{\mathtt{b}}\circ$`Tf)(Tx))`

Given the commutativity of natural transformations and the monoid laws on T, we have to show that our construction satisfies the following monad rules in functional programming:

1. `(return x) >>= f` $\longleftrightarrow$ `f(x)`
2. `Tx >>= return` $\longleftrightarrow$ `Tx`
3. `(Tx >>= f) >>= g` $\longleftrightarrow$ `Tx >>=` λ`x.(fx >>= g)`

The proof is mostly by commutative diagrams, where type X has been replaced by variable x:

Rule	Functional Programming	Category Theory
Left Identity	$$\begin{tikzcd} Ty \arrow[r, dashed, "\eta_{Ty}"] \arrow[dr, equal, dashed] & T^2y \arrow[d, "\mu_y"] & Tx \arrow[l, "Tf"'] \\ & Ty & x \arrow[l, "f"] \arrow[u, "\eta_X"'] \end{tikzcd}$$	$$\begin{tikzcd} Tx \arrow[r, "\eta_{Tx}"] \arrow[dr, equal] & T^2x \arrow[d, "\mu_x"] \\ & Tx \end{tikzcd}$$
Right Identity	$$\begin{tikzcd} Tx \arrow[r, "T(\eta_X)"] & T^2x \arrow[r, "\mu_x"] & Tx \end{tikzcd}$$	$$\begin{tikzcd} Tx \arrow[d, "T(\eta_X)"] \arrow[dr, equal] & \\ T^2x \arrow[r, "\mu_x"] & Tx \end{tikzcd}$$
Associativity	$$\begin{tikzcd} Tx \arrow[r, "Tf"] & T^2y \arrow[r, "T^2g"] \arrow[d, "\mu_y"] & T^3z \arrow[r, "T(\mu_z)"] \arrow[d, dashed, "\mu_{Tz}"] & T^2z \arrow[d, "\mu_z"] \\ & Ty \arrow[r, "Tg"] & T^2z \arrow[r, "\mu_z"] & Tz \end{tikzcd}$$	$$\begin{tikzcd} T^2[Tx] = T[T^2x] \arrow[r, "T(\mu_x)"] \arrow[d, "\mu_{Tx}"] & T[Tx] \arrow[d, "\mu_x"] \\ T[Tx] \arrow[r, "\mu_x"] & Tx \end{tikzcd}$$

Some notes:

1. Dashed lines are properties used for proofs rather than the rules themselves. In this case,

`fx` $= \mu_{\mathtt{y}} \circ$ `T(fx)`	use category theory's triangle
$= \mu_{\mathtt{y}} \circ$ `Tf(Tx)`	η is a natural transformation from $1_{\mathcal{C}}$ to T

2. No proof needed.
3. To check associativity, the two possible evaluation paths in functional programming are represented by solid arrows in the same figure. The one that goes from $T^2(y)$ to $T^3(z)$ comes from evaluating `Tx >>=` λ`x.(fx >>= g)`: first expand out the lambda `fx >= g`: `x`$\rightarrow$ `Ty` $\rightarrow$ `T`^{2}`z` $\rightarrow$ `Tz`, then 'lift' the whole expression by T, and apply μ_z at the end.

   If one is evaluating `(Tx >>= f) >>= g`, the sequence is `Tx` $\rightarrow$ `T`^{2}`y` $\rightarrow$ `Ty` $\rightarrow$ `T`^{2}`z` $\rightarrow$ `Tz`. Now

$\mu_{\mathtt{z}} \circ$ `Tg` $\circ\, \mu_{\mathtt{y}} = \mu_{\mathtt{z}} \circ \mu_{\mathtt{Tz}} \circ$ `T`^{2}`g`	μ is a natural transformation from T^2 to T
$= \mu_{\mathtt{z}} \circ$ `T(`$\mu_{\mathtt{z}}$`)` $\circ$ `T`^{2}`g`	use category theory's associativity

4. Surprisingly, we used the naturalness of both η and μ (alongisde the monoid laws) to prove the three rules. According to this article, the two sets of rules are equivalent and we can go from functional programming back to category theory i.e. define μ and η from `>>=` and `return`.

[2] $\eta_{\mathtt{a}}$ is sometimes just written as the function `T(·)`.

6 Why do monads exist?

In category theory, one can show that all monads are created from adjoints. Furthermore, the adjunctions that induce a monad T form a category, whose initial object (i.e. some 'special' recipe for T) involves passing from C to $\mathcal{C}_T$ and then back to C.

Somewhere down the road, functional programmers probably realized that monads (in their `return` and `>>=` form) can be used for stateful computation, and monads became the buzzword for FP evangelists. According to this comment, monads is a contraction of "monoidal triad", an incredibly descriptive term!

7 References for Programmers

- OCaml perspective: Monads
- Haskell perspective: IO Monad, State Monad, More IO etc
- Others: Pictures, nLab