# An introduction to that which shall not be named

Harley Eades III
Computer Science
Augusta University
heades@augusta.edu

September 2016

## 1 A quick overview from programming

Consider a C program with signature `int f(int)`. Now describe all possible computations this program can do. This is a difficult task, because this function could do a lot, like, prompt the user for input, send packets across the network, modify global state, and much more, but then eventually return an integer.

Now consider a purely functional programming language like Haskell [1] and list all the possible computations the function `f :: Int -> Int` can do. The list is a lot smaller. We know without a doubt that this function must take an integer as an input, and then do integer computations, and finally return an integer. No funny business went on inside this function. Thus, reasoning about pure programs is a lot easier.

However, a practical programmer might now be asking, "How do we get any real work done in a pure setting?". From stage left enters the monad. These allow for a programmer to annotate the return types of functions to indicate which side effects the function will use. For example, say we wanted `f` to use a global state of integers, then its type would be `f :: Int -> State [Int] Int` to indicate that `f` will take in an integer input, then during computation will use a global state consisting of a list of integers, but then eventually return an integer. Thus, the return type of `f` literally lists the side effects the function will use. Now while reasoning about programs we know exactly which side effects to consider.

In full generality a monad is a type constructor `m : * -> *` where `*` is the universe of types. Given a type `a` we call the type `m a` the type of computations returning values of type `a`. Thus, a function `f : a -> m b` is a function that takes in values of type `a`, and then returns a computation that will eventually return a value of type `b`.

Suppose we have `f :: a -> m b` and `g :: b -> m c`, and we wish to apply `g` to the value returned by `f`. This sounds perfectly reasonable, but ordinary composition `g.f` does not suffice, because the return type of `f` is not identical to the input type of `g`. Thus, we must come up with a new type of composition.

To accomplish this in Haskell we first need a new operator called *bind* which is denoted by `>>= :: m b -> (b -> m c) -> m c`. Then composition of `f` and `g` can be defined by `\x -> ((f x) >>= g) :: a -> m c`.

So we have a composition for functions whose type has the shape `a -> m b`, but any self respecting composition has an identity. This implies that we need some function `id :: a -> m a`, such that, `(\x -> (f x) >>= id) = f` and `(\x -> (id x) >>= g) = g`. This identity is denoted by `return :: a -> m a` in Haskell, and has to be taken as additional structure, because it cannot be defined in terms of bind.

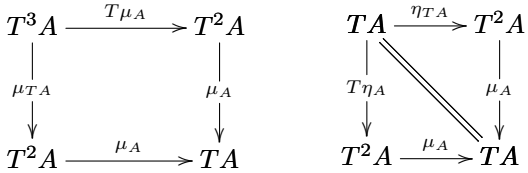
Using `bind` and `return` in combination with products, sum types, and higher-order functions a large number of monads can be defined, but what are monads really?

## 2 What is a monad really?

Monad's first arose in category theory and go back to Eilenberg and MacLane [3], but Moggi was the first to propose that they be used to model effectful computation in a pure setting [4]. After learning about Moggi's work Wadler pushed for their adoption by the functional programming community [2, 5, 6, 7]. This push resulted in the adoption of monads as the primary means of effectful programming in Haskell.

In the most general sense a monad is defined as follows:

**Definition 1.** *Suppose $\mathcal{C}$ is a category. Then a* ***monad*** *is a functor $T : \mathcal{C} \longrightarrow \mathcal{C}$ equipped with two natural transformations $\eta_A : A \longrightarrow TA$ and $\mu_A : T^2A \longrightarrow TA$ such that the following diagrams commute:*




From a computational perspective one should think of an object $A$ as being the type of values, and the object $TA$ as the type of computations. Then $\eta_A : A \longrightarrow TA$ says that all values are computations that eventually yield a value of type $A$, and $\mu_A : T^2A \longrightarrow TA$ says forming the type of computations that eventually yield a computation of type $TA$ is just as good as a computation of type $TA$. We can also think of $TA$ as capturing all of the possible computations where $T^2A$ really does not add anything new. We will see that join allows for a very interesting form of composition to be defined.

The diagrams above tell us how $\eta$ and $\mu$ interact together. For example, inserting a computation of type $TA$ into the type of computations of type $T^2A$, and then joining $T^2A$ to yield $TA$ does not do anything to the input. That is, $\eta_{TA}; \mu_A = \mathsf{id}_{TA}$. These diagrams will ensure that program evaluation behaves correctly in the model.

Consider an example. The functor $\mathcal{P} : \mathsf{Set} \longrightarrow \mathsf{Set}$ defined as $\mathcal{P}(X) = \{S \subseteq X\}$. First, we need to check to make sure this is an endofunctor so suppose $f : A \longrightarrow B$ is a function, then we can define $\mathcal{P}(f)(X \subseteq A) = \{f(x) \mid x \in X\} : \mathcal{P}(A) \longrightarrow \mathcal{P}(B)$. Suppose $f : A \longrightarrow B$ and $g : B \longrightarrow C$. Then composition is preserved $\mathcal{P}(f;g) = \mathcal{P}(f);\mathcal{P}(g) : \mathcal{P}(A) \longrightarrow \mathcal{P}(C)$. We can also see that $\mathcal{P}(\mathsf{id}_A) = \mathsf{id}_{\mathcal{P}(A)} : \mathcal{P}(A) \longrightarrow \mathcal{P}(A)$.

It turns out that this functor is indeed a monad:

$$\eta_A(x) = \{x\} : A \longrightarrow \mathcal{P}(X)$$
$$\mu_A(X) = \bigcup_{S \in X} S : \mathcal{P}(\mathcal{P}(A)) \longrightarrow \mathcal{P}(A)$$

Now we must verify that the diagrams for a monad commute:

$$\begin{array}{ccc} \mathcal{P}^3(A) & \xrightarrow{\mathcal{P}\mu_A} & \mathcal{P}^2(A) \\ \downarrow{\scriptstyle \mu_{\mathcal{P}(A)}} & & \downarrow{\scriptstyle \mu_A} \\ \mathcal{P}^2(A) & \xrightarrow{\mu_A} & \mathcal{P}(A) \end{array} \qquad \begin{array}{ccc} \mathcal{P}(A) & \xrightarrow{\eta_{\mathcal{P}(A)}} & \mathcal{P}^2(A) \\ \downarrow{\scriptstyle \mathcal{P}\eta_A} & \searrow & \downarrow{\scriptstyle \mu_A} \\ \mathcal{P}^2(A) & \xrightarrow{\mu_A} & \mathcal{P}(A) \end{array}$$

Using diagram chasing[1] it is easy to see that the diagrams on the right commute. The left most diagram commutes by the following equational reasoning:

$$\begin{array}{rcl} \mu_A(\mathcal{P}(\mu_A)(S \in \mathcal{P}^3(A))) & = & \mu_A(\{\mu_A(S') \mid S' \in S\}) \\ & = & \bigcup_{S'' \in (\{\mu_A(S') \mid S' \in S\})} S'' \\ & = & \bigcup_{S''' \in \bigcup_{S'' \in S} S''} S''' \\ & = & \bigcup_{S''' \in \mu_{\mathcal{P}(A)}(S)} S''' \\ & = & \mu_A(\mu_{\mathcal{P}(A)}(S)) \end{array}$$

Functions with a type of the form $A \longrightarrow \mathcal{P}(B)$ have a special place in computer science, because they model non-determinism.

# 3 Jumping inside a monad

Suppose $(T : \mathcal{C} \longrightarrow \mathcal{C}, \eta, \mu)$ is a monad. Then we can think of $\mathcal{C}$ as the pure world, and the world inside $T$ as the impure world, or the universe of computations. Given such a monad can we define exactly what this impure world is? In turns out we can by constructing the underlying category of the monad. There happens to be two such categories, but they are related.

## 3.1 The Kleisli category

Suppose we have a monad $(T, \eta, \mu)$ on some category $\mathcal{C}$. Then the **Kleisli category** of the monad $T$, denoted $\mathcal{C}_T$, has as objects the objects $\mathcal{C}$, and as morphisms all the morphisms of $\mathcal{C}$ of the form $f : A \longrightarrow TB$.

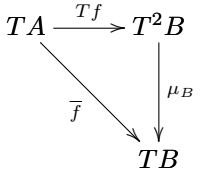
[1] Chasing an element in $\mathcal{P}(A)$ across the top path, and then across the bottom path should echo back what we started with.

It is common to see the later as being defined as if $f : A \longrightarrow TB$ is a morphism of $\mathcal{C}$, then $\hat{f} : A \longrightarrow B$ is a morphism of $\mathcal{C}_T$. However, this can often lead to confusion, and so for this lecture we will be explicit about the form of $f$, and use the former definition.

Before we can do anything we must first show that $\mathcal{C}_T$ is indeed a category.

**Lemma 2.** *Suppose $(T, \eta, \mu)$ is a monad on $\mathcal{C}$. Then the Kleisli construction $\mathcal{C}_T$ is a category.*

*Proof.* Suppose $f : A \longrightarrow TB$ is a morphism. Then the **Kleisli lifting** of $f$ is the morphism $\overline{f}$:

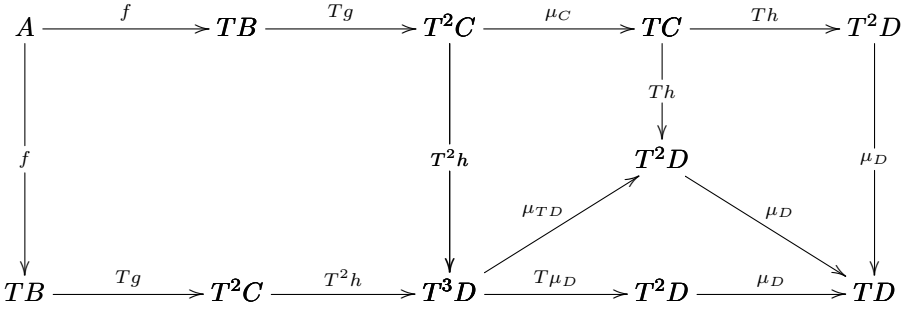



**Composition.** Suppose $f : A \longrightarrow TB$ and $g : B \longrightarrow TC$ are two morphisms in $\mathcal{C}_T$. Then their composition is defined by $f; \overline{g}$. Thus, composition in $\mathcal{C}_T$ is composition in $\mathcal{C}$ where the second morphism is lifted.

We have to prove that this composition is associative. Suppose $f : A \longrightarrow TB$, $g : B \longrightarrow TC$, and $g : C \longrightarrow TD$ are morphisms of $\mathcal{C}_T$. Then we must show that:

$$(f; \overline{g}); \overline{h} = f; \overline{(g; \overline{h})}$$

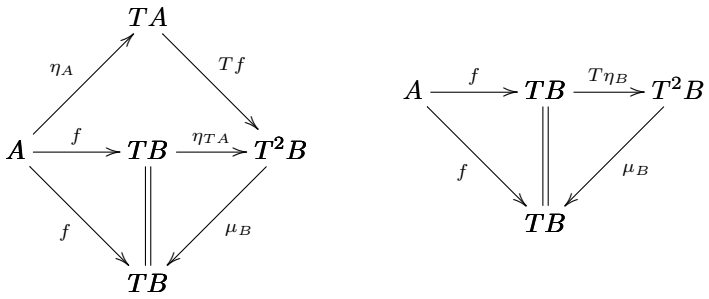
We can prove this by showing that the following diagram commutes:




The left square trivially commutes, the left-most upper-right square commutes by naturality of $\mu$, the right-most upper-right square trivially commutes, and the right-lower triangle commutes by the monad laws.

**Identities.** Suppose $A$ is an object of $\mathcal{C}_T$. Then we need to show there there exists a morphism $\mathsf{id}_A : A \longrightarrow TA$ such that for any morphism $f : A \longrightarrow TB$ we have $\mathsf{id}_A; \overline{f} = f = f; \overline{\mathsf{id}_B}$. The only option we have is $\eta_A : A \longrightarrow TA$, because it is the only morphism with the required form that we know always exists. The

following commutative diagrams imply our desired property:




The left diagram commutes, because the upper triangle commutes by naturality of $\eta$, and the lower-left triangle commutes by the monad laws. The right diagram commutes, because the right-most diagram commutes by the monad laws. □

Notice that the previous proof explicitly defines the notion of Kleisli lifting of a morphism. The astute reader will notice that we have seen this before. Consider monads from a functional programming perspective, we can see that `return : A -> m A` corresponds to $\eta_A : A \longrightarrow TA$, but what does bind, `>>= :: m b -> (b -> m c) -> m c`, correspond to? Surely it is not $\mu_A : T^2A \longrightarrow TA$. Consider the following equivalent form of bind obtained by currying:

```
>>=' :: (b -> m c) -> (m b -> m c)
```

This looks a lot like the Kleisli lifting:

$$\mathsf{Hom}_{\mathcal{C}}(B, TC) \longrightarrow \mathsf{Hom}_{\mathcal{C}}(TB, TC)$$

In fact, it is! One of the most important realizations that Moggi had was that programming in a monad amounts to programming in the Kleisli category of the monad.

### 3.2 The Eilenberg-Moore category

## 4 Categorical Model of $\lambda_T$

## 5 Monads are modular, right?

## References

[1] The haskell programming language. Online: `http://www.haskell.org`.

[2] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93. Citeseer, 1993.

[3] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[4] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.

[5] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[6] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[7] Philip Wadler. *Monads for functional programming*, pages 24–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.