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?

*These lecture notes were for the Computational Logic Center Seminar at the University of Iowa.

2 What is a monad really?

Monad's first arose in category theory and go back to Eilenberg and MacLane [7], but Moggi was the first to propose that they be used to model effectful computation in a pure setting [11]. After learning about Moggi's work Wadler pushed for their adoption by the functional programming community [4, 13, 14, 15]. 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:*

$$
\begin{tikzcd}
T^3A \arrow[r, "T\mu_A"] \arrow[d, "\mu_{TA}"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[d, "T\eta_A"'] \arrow[dr, equal] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"] & TA
\end{tikzcd}
$$

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 η and μ 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:

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

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

$$
\begin{tikzcd}
\mathcal{P}^3(A) \arrow[r, "\mathcal{P}\mu_A"] \arrow[d, "\mu_{\mathcal{P}(A)}"'] & \mathcal{P}^2(A) \arrow[d, "\mu_A"] \\
\mathcal{P}^2(A) \arrow[r, "\mu_A"] & \mathcal{P}(A)
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{P}(A) \arrow[r, "\eta_{\mathcal{P}(A)}"] \arrow[d, "\mathcal{P}\eta_A"'] \arrow[dr, equal] & \mathcal{P}^2(A) \arrow[d, "\mu_A"] \\
\mathcal{P}^2(A) \arrow[r, "\mu_A"] & \mathcal{P}(A)
\end{tikzcd}
$$

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:

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

[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.

Note that in addition to the previous diagrams we would also need to show that η and μ are natural transformations, but we leave this to the reader. 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, η, μ) on some category $\mathcal{C}$. The **Kleisli category** of the monad T is denoted $\mathcal{C}_T$. The objects of $\mathcal{C}_T$ are the objects of $\mathcal{C}$, and the morphisms of $\mathcal{C}_T$ from an object A to an object B are the morphisms of $\mathcal{C}$ from A to TB. That is, $\mathcal{C}_T(A, B) = \mathcal{C}(A, TB)$. We will denote the morphisms in $\mathcal{C}_T$ as $\hat{f}$.

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

Lemma 2. *Suppose (T, η, μ) 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 $\overline{f}$:

$$
\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[dr, "\overline{f}"'] & T^2B \arrow[d, "\mu_B"] \\
& TB
\end{tikzcd}
$$

Composition. Suppose $\hat{f} : A \longrightarrow B$ and $\hat{g} : B \longrightarrow C$ are two morphisms in $\mathcal{C}_T$. These are equivalent to the morphisms $f : A \longrightarrow TB$ and $g : B \longrightarrow TC$ in $\mathcal{C}$. Then their composition, $\hat{f};\hat{g} : A \longrightarrow C$ in $\mathcal{C}_T$ is defined by $f;\overline{g}$ in $\mathcal{C}$. 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 $\hat{f} : A \longrightarrow B$, $\hat{g} : B \longrightarrow C$, and $\hat{h} : C \longrightarrow D$ are morphisms of $\mathcal{C}_T$. These are all equivalent to $f : A \longrightarrow TB$, $g : B \longrightarrow TC$, and $h : C \longrightarrow TD$ from $\mathcal{C}$.

It suffices to show that:

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

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

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dd, "f"] & TB \arrow[r, "Tg"] & T^2C \arrow[r, "\mu_C"] \arrow[dd, "T^2h"] & TC \arrow[r, "Th"] \arrow[d, "Th"] & T^2D \arrow[dd, "\mu_D"] \\
& & & T^2D \arrow[dr, "\mu_D"] & \\
TB \arrow[r, "Tg"] & T^2C \arrow[r, "T^2h"] & T^3D \arrow[r, "T\mu_D"] \arrow[ur, "\mu_{TD}"] & T^2D \arrow[r, "\mu_D"] & TD
\end{tikzcd}
$$

The left square trivially commutes, the left-most upper-right square commutes by naturality of μ, 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 $\hat{\mathsf{id}}_A : A \longrightarrow A$ such that for any morphism $\hat{f} : A \longrightarrow B$ we have $\hat{\mathsf{id}}_A; \hat{f} = \hat{f} = \hat{f}; \hat{\mathsf{id}}_B$. The only option we have is $\eta_A : A \longrightarrow TA$, because it is the only morphism from $\mathcal{C}$ with the required form that we know always exists. Thus, $\hat{\mathsf{id}}_A = \eta_A$. The following commutative diagrams imply our desired property:

$$
\begin{tikzcd}
 & TA \arrow[dr, "Tf"] & \\
A \arrow[ur, "\eta_A"] \arrow[r, "f"] \arrow[dr, "f"'] & TB \arrow[r, "\eta_{TA}"] \arrow[d, equal] & T^2B \arrow[dl, "\mu_B"] \\
 & TB &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "f"'] & TB \arrow[r, "T\eta_B"] \arrow[d, equal] & T^2B \arrow[dl, "\mu_B"] \\
 & TB &
\end{tikzcd}
$$

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

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. As we can see `bind` and μ are not completely unrelated, and one can actually define each of them in terms of the other. So monads could be defined in terms of bind and then we could derive μ, but we leave the details to the reader.

3.2 The Eilenberg-Moore category

A second more general category that corresponds to the universe inside of a monad is called the Eilenberg-Moore category. Suppose (T, η, δ) is a monad on the category $\mathcal{C}$. Then a T-**algebra** is a pair (A, h_A) of an object A of $\mathcal{C}$ and a morphism, called the structure map, $h_A : TA \longrightarrow A$ such that the following diagrams commute:

$$
\begin{tikzcd}
T^2A \arrow[r, "Th_A"] \arrow[d, "\mu_A"'] & TA \arrow[d, "h_A"] \\
TA \arrow[r, "h_A"] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, equal] & TA \arrow[d, "h_A"] \\
 & A
\end{tikzcd}
$$

A morphism $f : (A, h_A) \longrightarrow (B, h_B)$ between T-algebras is a morphism $f : A \longrightarrow B$ of $\mathcal{C}$ such that the following diagram commutes:

$$\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[d, "h_A"] & TB \arrow[d, "h_B"] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

The **Eilenberg-Moore category** $\mathcal{C}^T$ of a monad (T, η, μ) has as objects all the T-algebras and as morphisms all of the T-algebras morphisms. The categorical structure of $\mathcal{C}^T$ is inherited from the underlying category $\mathcal{C}$ as the following result shows.

Lemma 3. *Suppose (T, η, μ) is a monad on a category $\mathcal{C}$. Then $\mathcal{C}^T$ is a category.*

Proof. Suppose (T, η, μ) is a monad on a category $\mathcal{C}$.

Composition. Suppose $f : (A, h_A) \longrightarrow (B, h_B)$ and $g : (B, h_B) \longrightarrow (C, h_C)$ are two T-algebra morphisms. The composition $f; g : A \longrightarrow C$ is a T-algebra morphism between T-algebras (A, h_A) and (C, h_C) because the following diagram commutes:

$$\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[d, "h_A"] & TB \arrow[r, "Tg"] \arrow[d, "h_B"] & TC \arrow[d, "h_C"] \\
A \arrow[r, "f"] & B \arrow[r, "g"] & C
\end{tikzcd}$$

Each square commutes by the respective diagram for each morphism. Associativity holds trivially, because it holds in $\mathcal{C}$.

Identities. Given a T-algebra, (A, h_A), we must define an identity morphism $\mathsf{id} : (A, h_A) \longrightarrow (A, h_A)$, but we can simply take $\mathsf{id}_A : A \longrightarrow A$ as this morphism, because the following diagram commutes:

$$\begin{tikzcd}
TA \arrow[r, "T\mathsf{id}_A"] \arrow[d, "h_A"'] & TA \arrow[d, "h_A"] \\
A \arrow[r, "\mathsf{id}_A"'] & A
\end{tikzcd}$$

This diagram commutes, because we know $T\mathsf{id}_A = \mathsf{id}_{TA}$, because T is an endofunctor on $\mathcal{C}$. Composition will respect identities, because composition in $\mathcal{C}$ does. □

The Eilenberg-Moore category is related to the Kleisli category in the following way. Define the category $\mathsf{Free}(\mathcal{C}^T)$ to be the full subcategory of $\mathcal{C}^T$ with objects the free T-algebras of the form (TA, μ_A) the diagram making this a T-algebra is the monad law for μ. Morphisms in $\mathsf{Free}(\mathcal{C}^T)$ are all the T-algebras morphisms between free T-algebras.

Lemma 4. *Suppose (T, η, μ) is a monad on the category $\mathcal{C}$. Then the category $\mathcal{C}_T$ is a full subcategory of $\mathcal{C}^T$.*

Proof. The full proof of this is out of scope of this short lecture note, but it is possible to show that $\mathcal{C}_T$ is equivalent to $\mathsf{Free}(\mathcal{C}^T)$, and hence, we obtain our result. □

The benefit of the Eilenberg-Moore category is that it is often easier to prove properties about it than the Kleisli category. Since the Kleisli category is a full subcategory of the Eilenberg-Moore category any property that holds on the Eilenberg-Moore category also holds for the Kleisli category.

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x : A}\text{var} \qquad \frac{}{\Gamma \vdash \mathsf{triv} : 1}1_i \qquad \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash (t_1, t_2) : A \times B}\times_i \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A}\times_{e_1}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\, t : B}\times_{e_2} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\lambda_i \qquad \frac{\Gamma \vdash t_2 : A \quad \Gamma \vdash t_1 : A \to B}{\Gamma \vdash t_1\, t_2 : B}\lambda_e \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\, t : T\, A}T_i$$

$$\frac{\Gamma \vdash t_1 : T\, A \quad \Gamma, x : A \vdash t_2 : T\, B}{\Gamma \vdash \mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2 : T\, B}T_e$$

Figure 1: Typing Rules for λ_T

$$\frac{}{(\lambda x : A.t_2)\, t_1 \rightsquigarrow [t_1/x]t_2}\text{R\_BETA} \qquad \frac{}{\mathsf{fst}\,(t_1, t_2) \rightsquigarrow t_1}\text{R\_FIRST} \qquad \frac{}{\mathsf{snd}\,(t_1, t_2) \rightsquigarrow t_2}\text{R\_SECOND}$$

$$\frac{}{\mathsf{let}\, x \leftarrow \mathsf{return}\, t_1\, \mathsf{in}\, t_2 \rightsquigarrow [t_1/x]t_2}\text{R\_BIND}$$

Figure 2: Reduction Rules for λ_T

4 Categorical Model of λ_T

At this point we have introduced the basics of monads categorically. In this section we show how to categorically model a simple type theory with monads called λ_T. We can view λ_T as the smallest typed functional programming language with monads, but by extending this language with more features one can study monads incrementally.

The syntax for λ_T is as follows:

$$\begin{array}{ll}
\text{(types)} & A, B, C := 1 \mid T\, A \mid A \times B \mid A \to B \\
\text{(terms)} & t := x \mid \mathsf{triv} \mid (t_1, t_2) \mid \mathsf{fst}\, t \mid \mathsf{snd}\, t \mid \lambda x : A.t \mid t_1\, t_2 \mid \mathsf{return}\, t \mid \mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2 \\
\text{(contexts)} & \Gamma := \cdot \mid x : A \mid \Gamma_1, \Gamma_2
\end{array}$$

We can see that this is an extension of the simply typed λ-calculus. We add a new type $T\, A$ which represents an arbitrary monad, and new terms for return and bind denoted $\mathsf{return}\, t$ an $\mathsf{let}\, x \leftarrow t_1\, \mathsf{in}\, t_2$ respectively. This language is very similar to Moggi's metalanguage [11].

The typing rules for λ_T can be found in Figure 1, and the reduction rules are in Figure 2. The reduction rules are rather simplistic, but are advanced enough for the purpose of this note. Congruence rules are omitted in the interest of brevity. There are also more monadic rules that one might one, for example a commuting conversion of bind, but we leave these out.

The main question of this section is, what is the categorical model of λ_T? We know we can interpret λ_T excluding the monadic bits into a cartesian closed category. Thus, the model of full λ_T must be some extension of a cartesian closed category with a monad. Is it enough to simply take a cartesian closed category $\mathcal{C}$ with a monad (T, η, μ) on $\mathcal{C}$?

Suppose $(\mathcal{C}, 1, \times, \to)$ is a cartesian closed category, and (T, η, μ) is a monad on $\mathcal{C}$. Types are interpreted into this model as follows:

$$\begin{array}{lcl}
[\![1]\!] & = & 1 \\
[\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!] \\
[\![T\, A]\!] & = & T\, [\![A]\!]
\end{array}$$

Contexts $\Gamma = x_1 : A_1, \ldots, x_i : A_i$ will be interpreted into $\mathcal{C}$ by $[\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_i]\!]$. To make the syntax less cluttered we will drop the interpretation brackets from the interpretation of types.

We will interpret each typing judgment $\Gamma \vdash t : A$ as a morphism $[\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$ by induction on the form of the typing judgment. Now consider the two monadic typing rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\, t : T\,A}\,T_i \qquad\qquad \frac{\Gamma \vdash t_1 : T\,A \quad \Gamma, x : A \vdash t_2 : T\,B}{\Gamma \vdash \mathsf{let}\, x \leftarrow t_1 \,\mathsf{in}\, t_2 : T\,B}\,T_e$$

Consider the left rule, and suppose we have a morphism $t : \Gamma \longrightarrow A$ in $\mathcal{C}$. Then we must construct a morphism of the form $\Gamma \longrightarrow T\,A$, but this is easily done by $t;\eta_A : \Gamma \longrightarrow T\,A$. Thus, the interpretation of $[\![\mathsf{return}\, t]\!]$ is $[\![t]\!];\eta_A$.

Now consider the rule T_e, and suppose we have morphisms $t_1 : \Gamma \longrightarrow T\,A$ and $t_2 : \Gamma \times A \longrightarrow T\,B$. We are expecting to Kleisli lift t_2 to $\overline{t_2} = (T\,t_2);\mu_B : T(\Gamma \times A) \longrightarrow TB$, and then compose $\langle \mathsf{id}_\Gamma, t_1\rangle : \Gamma \longrightarrow \Gamma \times T\,A$ with $\overline{t_2}$, but the types do not match! If we had a natural transformation $\mathsf{st}_{A,B} : A \times TB \longrightarrow T(A \times B)$ then we could finish the job by interpreting $\Gamma \vdash \mathsf{let}\, x \leftarrow t_1 \,\mathsf{in}\, t_2 : T\,B$ by $\langle \mathsf{id}_\Gamma, t_1\rangle;\mathsf{st}_{\Gamma,A};\overline{t_2} : \Gamma \longrightarrow T\,B$. Therefore, an arbitrary monad does not have enough structure to model the bind rule in the presence of multiple hypotheses. Instead we need a strong monad.

Definition 5. *A monad (T,η,μ) on a category $\mathcal{C}$ with all finite products is **strong** if there exists a natural transformation $\mathsf{st}_{A,B} : A \times TB \longrightarrow T(A \times B)$ called the **tensorial strength** of the monad. In addition, the following diagrams must commute:*

$$\begin{tikzcd}
1 \times TA \arrow[r, "\mathsf{st}_{1,A}"] \arrow[dr, "\rho_{TA}"'] & T(1 \times A) \arrow[d, "T\rho_A"] \\
& TA
\end{tikzcd}
\qquad
\begin{tikzcd}
& A \times B \arrow[dl, "\mathsf{id}_A \times \eta_B"'] \arrow[dr, "\eta_{A \times B}"] & \\
A \times TB \arrow[rr, "\mathsf{st}_{A,B}"] & & T(A \times B) \\
A \times T^2B \arrow[u, "\mathsf{id}_A \times \mu_B"] \arrow[r, "\mathsf{st}_{A,TB}"] & T(A \times TB) \arrow[r, "T\mathsf{st}_{A,B}"] & T^2(A \times B) \arrow[u, "\mu_{A \times B}"']
\end{tikzcd}$$

$$\begin{tikzcd}
(A \times B) \times C \arrow[rr, "\mathsf{st}_{A \times B, C}"] \arrow[d, "\alpha_{A,B,C}"'] & & T((A \times B) \times C) \arrow[d, "T\alpha_{A,B,C}"] \\
A \times (B \times C) \arrow[r, "\mathsf{id}_A \times \mathsf{st}_{B,C}"] & A \times T(B \times C) \arrow[r, "\mathsf{st}_{A, B \times C}"] & T(A \times T(B \times C))
\end{tikzcd}$$

Adopting strong monads instead of arbitrary ones yields a sound and complete model.

Definition 6. *A λ_T **model** consists of a cartesian closed category $\mathcal{C}$ equipped with a strong monad (T,η,μ) on $\mathcal{C}$.*

Finally, we have the following:

Theorem 7 (Soundness)**.** *Suppose $(T : \mathcal{C} \longrightarrow \mathcal{C}, \eta, \mu)$ is a λ_T model. Then if $\Gamma \vdash t_1 : A$ and $t_1 \rightsquigarrow t_2$, the $[\![t_1]\!] \cong [\![t_2]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$ in $\mathcal{C}$.*

5 Monads are modular, right?

The most important concept of category theory, logic, and functional programming is composition. In practice, it is very common to need several different types of side effects. Naturally, some programs will use different ones, and others may use all of them. So given monads (T_1,η_1,μ_1) and (T_2,η_2,μ_2) on a category $\mathcal{C}$

can we compose these together and obtain a monad (T_3, η_3, μ_3) that encompasses the side effects of both T_1 and T_2?

One might think that the question is obviously true. This is category theory, right? Functors compose, and so we should be able to compose monads, but in what order? It turns out that we cannot even define join of this composition. If we take $T_1; T_2 : \mathcal{C} \longrightarrow \mathcal{C}$ to be the composition, then notice that we can easily obtain a natural transformation $\eta_3 = A \xrightarrow{\eta_2} T_2 A \xrightarrow{\eta_1} T_1(T_2 A)$, but notice that we cannot define $\mu_3 : T_1(T_2(T_1(T_2 A))) \longrightarrow T_1(T_2 A)$ in terms of $\mu_1 : T_1^2 \longrightarrow T_1$ and $\mu_2 : T_2^2 \longrightarrow T_2$. So this will not work.

Composition of monads has been a hot topic since their conception. In fact, Moggi spent a lot of time thinking about this; see [12]. Papers on this concept pop up pretty consistently each year since monads where introduced to the functional programming community. However, most of these papers exclude a categorical model. This is rather upsetting, because the model allows us to decide on the approaches merits. Monads are a categorical concept after all, and when we extend their use we should provide an elegant categorical model.

In this section we will cover some of the most popular ways of composing monads. We focus on the categorical models, but we will give brief descriptions of how these can be added to a functional programming language.

5.1 Distributive Laws

Recall that we were unable to define $\mu_3 : T_1(T_2(T_1(T_2 A))) \longrightarrow T_1(T_2 A)$ in terms of $\mu_1 : T_1^2 \longrightarrow T_1$ and $\mu_2 : T_2^2 \longrightarrow T_2$. But, if we could first commute $T_2 T_1$ in the source of μ_3, then we could. This is exactly what distributive laws give us.

Given two monads (T_1, η_1, μ_1) and (T_2, η_2, μ_2) on a category $\mathcal{C}$, a **distributive law** of T_2 over T_1 is a natural transformation $\mathsf{dist} : T_1 T_2 \longrightarrow T_2 T_1$ subject to the following commutative diagrams:

$$
\begin{tikzcd}
T_1 \arrow[r, "T_1\eta_2"] \arrow[ddr, "\eta_2 T_1"'] & T_1 T_2 \arrow[dd, "\mathsf{dist}"] & T_1 T_2 T_2 \arrow[l, "T_1\mu_2"'] \arrow[d, "\mathsf{dist} T_2"] \\
 & & T_2 T_1 T_2 \arrow[d, "T_2 \mathsf{dist}"] \\
 & T_2 T_1 & T_2 T_2 T_1 \arrow[l, "\mu_2 T_1"']
\end{tikzcd}
\qquad
\begin{tikzcd}
T_2 \arrow[r, "\eta_1 T_2"] \arrow[ddr, "T_2\eta_1"'] & T_1 T_2 \arrow[dd, "\mathsf{dist}"] & T_1 T_1 T_2 \arrow[l, "\mu_1 T_2"'] \arrow[d, "T_1 \mathsf{dist}"] \\
 & & T_1 T_2 T_1 \arrow[d, "\mathsf{dist} T_1"] \\
 & T_2 T_1 & T_2 T_1 T_1 \arrow[l, "T_2 \mu_1"']
\end{tikzcd}
$$

Distributive laws are due to Beck [2], and were extensively studented by Manes and Mulry [8, 9]. Please see the latter for further references on the subject.

We now have the following result:

Theorem 8. *Suppose (T_1, η_1, μ_1) and (T_2, η_2, μ_2) are two monads on $\mathcal{C}$, and $\mathsf{dist} : T_1 T_2 \longrightarrow T_2 T_1$ is a distributive law. Then the endofunctor $T_2 T_1$ is a monad on $\mathcal{C}$.*

Proof. It suffices to define $\eta_3 : A \longrightarrow T_2 T_1 A$ and $\mu_3 : T_2 T_1 T_2 T_1 A \longrightarrow T_2 T_1 A$, and show they satisfy the monad laws (Definition 1).

We have the following definitions:

$$
\begin{aligned}
\eta_3 &= A \xrightarrow{\eta_2} T_2 A \xrightarrow{T_2 \eta_1} T_2 T_1 A \\
\mu_3 &= T_2 T_1 T_2 T_1 A \xrightarrow{T_2 \mathsf{dist}_{T_1 A}} T_2^2 T_1^2 A \xrightarrow{T_2^2 \mu_1} T_2^2 T_1 A \xrightarrow{\mu_2} T_2 T_1 A
\end{aligned}
$$

Now we show that these definitions satisfy the monad laws:

Case.

$$\begin{tikzcd}
(T_2T_1)^3A \arrow[r, "T_2T_1\mu_3"] \arrow[d, "\mu_3T_2T_1"'] & (T_2T_1)^2A \arrow[d, "\mu_3"] \\
(T_2T_1)^2A \arrow[r, "\mu_3"] & T_2T_1A
\end{tikzcd}$$

This case follows from the fact that the following diagram commutes:

$$\begin{tikzcd}
T_2T_1T_2T_1T_2T_1A \arrow[r, "T_2T_1T_2\mathsf{dist}"] \arrow[d, "T_2\mathsf{dist}T_1T_2T_1"'] \arrow[dr, phantom, "(1)"] & T_2T_1T_2^2T_1^2A \arrow[r, "T_2T_1T_2^2\mu_1"] \arrow[d, "T_2\mathsf{dist}T_2T_1^2"] \arrow[dr, phantom, "(2)"] & T_2T_1T_2^2T_1A \arrow[r, "T_2T_1\mu_2"] \arrow[d, "T_2\mathsf{dist}T_2T_1"] & T_2T_1T_2T_1A \arrow[dd, "T_2\mathsf{dist}T_1"] \\
T_2^2T_1^2T_2T_1A \arrow[r, "T_2^2T_1\mathsf{dist}T_1"] \arrow[dd, "T_2^2\mu_1T_2T_1"'] & T_2^2T_1T_2T_1^2A \arrow[r, "T_2^2T_1T_2\mu_1"] \arrow[d, "T_2^2\mathsf{dist}T_1^2"] \arrow[dr, phantom, "(5)"] & T_2^2T_1T_2T_1A \arrow[d, "T_2\mathsf{dist}T_1"] \arrow[r, phantom, "(3)"] & {} \\
{} \arrow[r, phantom, "(4)"] & T_2^3T_1^3A \arrow[r, "T_2^3T_1\mu_1"] \arrow[d, "T_2^3\mu_1T_1"] \arrow[dr, phantom, "(6)"] & T_2^3T_1^2A \arrow[r, "T_2\mu_2T_1^2"] \arrow[d, "T_2^3\mu_1"] \arrow[dr, phantom, "(7)"] & T_2^2T_1^2A \arrow[d, "T_2^2\mu_1"] \\
T_2^2T_1T_2T_1A \arrow[r, "T_2^2\mathsf{dist}T_1"] \arrow[d, "\mu_2T_1T_2T_1"'] \arrow[dr, phantom, "(8)"] & T_2^3T_1^2A \arrow[r, "T_2^3\mu_1"] \arrow[d, "\mu_2T_2T_1^2"] \arrow[dr, phantom, "(9)"] & T_2^3T_1A \arrow[r, "T_2\mu_2T_1"] \arrow[d, "\mu_2T_2T_1"] \arrow[dr, phantom, "(10)"] & T_2^2T_1A \arrow[d, "\mu_2T_1"] \\
T_2T_1T_2T_1A \arrow[r, "T_2\mathsf{dist}T_1"] & T_2^2T_1^2A \arrow[r, "T_2^2\mu_1"] & T_2^2T_1A \arrow[r, "\mu_2T_1"] & T_2T_1A
\end{tikzcd}$$

Diagrams one, two, and five commute by naturality of dist, diagrams seven, eight, and nine commute by naturality of μ_2, diagrams six and ten commute by the monad laws, and diagrams three and four commute by the distributive laws.

Case.

$$\begin{tikzcd}
T_2T_1A \arrow[r, "\eta_3"] \arrow[dr, equal] & (T_2T_1)^2A \arrow[d, "\mu_3"] \\
& T_2T_1A
\end{tikzcd}$$

This diagram commutes because the following one does:

$$\begin{tikzcd}
T_2T_1A \arrow[r, "\eta_2T_2T_1"] \arrow[dd, equal] & T_2^2T_1A \arrow[r, "T_2\eta_1T_2T_1"] \arrow[dr, "T_2^2\eta_1T_1"] \arrow[dd, equal] & T_2T_1T_2T_1A \arrow[d, "T_2\mathsf{dist}T_1"] \\
& & T_2^2T_1^2A \arrow[d, "T_2^2\mu_1"] \\
T_2T_1A & T_2^2T_1A \arrow[l, "\mu_2T_1"'] \arrow[r, equal] & T_2^2T_1A
\end{tikzcd}$$

The left and the right lower squares commute by the monad laws, and the right triangle commutes by the distributive laws.

Case.

$$
\begin{tikzcd}
T_2T_1A \arrow[d, "T_2T_1\eta_3"'] \arrow[dr, equal] & \\
(T_2T_1)^2A \arrow[r, "\mu_3"] & T_2T_1A
\end{tikzcd}
$$

This is case is similar to the previous one.

□

Many concrete monads have the benefit that we can define the distributive laws, and hence, can be composed. As an example suppose we wanted to compose the maybe monad and the list monad. In Haskell, we can define the following distributive law[2] (writing `List a` instead of `[a]` for readability):

```
dist :: List (Maybe a) -> Maybe (List a)
dist [] = Just []
dist (Nothing:xs) = Nothing
dist (Just x:xs) = dist xs >>= (\l -> return (x:l))
```

This definition shows that applying `dist` to a list where every element is of the form `Just x` for some `x` of type `a` results in `Just l` where `l` contains all of the elements like `x`. If `Nothing` ever appears then `dist` returns `Nothing`. This fits well with how the maybe monad operates.

Using `dist` we can define the monad `Maybe (List a)`:

```
returnML :: a -> Maybe (List a)
joinML :: Maybe (List (Maybe (List a))) -> Maybe (List a)
bindML :: Maybe (List a) -> (a -> Maybe (List a)) -> Maybe (List a)
```

There are many more example use cases. See the work of Jones and Duponcheel [3] for some variations of this type of composition in Haskell.

One of the negatives of distributive laws is that a distributive law may not be definable for a particular pair of monads. For example, in Haskell it is not possible to define a distributive law between the I/O monad and the state monad.

5.2 Monad Transformers

Moggi [12] was perhaps the first to realize that the success of the monadic approach depends on the need for monadic models to be modular. So he spent quite sometime studying a means of composing monads called **monad transformers.**

Suppose $\mathcal{C}$ is a category with products. Then define the category $\mathsf{Mon}(\mathcal{C})$ to have as objects strong monads (Definition 5) and as morphisms strong-monad morphisms.

Definition 9. *A* ***strong-monad morphism*** *is a natural transformation, $\alpha : T_1 \longrightarrow T_2$, where $(T_1, \eta_1, \mu_1, \mathsf{st}_1)$ and $(T_2, \eta_2, \mu_2, \mathsf{st}_2)$ are strong monads. Furthermore, the following diagrams must commute:*

[2]For the entire implementation please see `https://github.com/heades/intro-monads/blob/master/MaybeList.hs`.

$$
\begin{tikzcd}
A \arrow[r, "\eta_1"] \arrow[d, equal] & T_1 A \arrow[d, "\alpha_A"] & T_1^2 A \arrow[l, "\mu_1"'] \arrow[d, "(\alpha;\alpha)_A"] \\
A \arrow[r, "\eta_2"] & T_2 A & T_2^2 A \arrow[l, "\mu_2"']
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times T_1 B \arrow[r, "\mathsf{st}_1"] \arrow[d, "\mathsf{id}_A \times \alpha_B"] & T_1(A \times B) \arrow[d, "\alpha_{A \times B}"] \\
A \times T_2 B \arrow[r, "\mathsf{st}_2"] & T_2(A \times B)
\end{tikzcd}
$$

Now a **monad transformer** is an endofunctor in $\mathsf{Mon}(\mathcal{C})$.

Consider the following example due to Moggi [12]. Given a monad $(T, \eta, \mu, \mathsf{st})$ over a category $\mathcal{C}$ with products and coproducts, then the monad T_E of T**-computations with exceptions** is defined as follows:

- $T_E(A) = T(A + E)$
- $\eta^E = \mathsf{in}_l; \eta_{A+E}$
- $\mu^E = \overline{[\mathsf{id}_{T(A+E)}, \mathsf{in}_r; \eta_{A+E}]}$

In addition, a monad morphism $\alpha : T' \longrightarrow T$ induces a monad morphism $\sigma = \alpha_{A+E} : S_E \longrightarrow T_E$. Finally, for every monad $(T, \eta, \mu, \mathsf{st})$ there are two monad morphism $\eta_E : T \longrightarrow T_E$ and $\mu_E : T_{EE} \longrightarrow T_E$ making monad transformers monads in the category $\mathsf{Mon}(\mathcal{C})$.

As presented here monad transformers are Moggi's monad constructors [12], but in practice they were found to not allow for many commonly used monads to be composed. Liang et al. [5] extended the approach arriving at a more general notion of monad transformer that has been adopted in Haskell. However, this improved the practical side, but without an elegant categorical model. In fact, to the knowledge of the author the only account of a categorical semantics for the monad transformers in Haskell is detailed in the note by Oleksandr Manzyuk [10], but it is very ad hoc.

5.3 Coproducts

Distributive laws nor monad transformers are general solutions to the composition of monads problem. Thus, the question is still very much open. A more recent approach that is more general than both distributive laws and monads transformers which is based in category theory was proposed by Lüth and Ghani [6]. In fact, distributive laws arrive as special cases of their model.

The main idea is when given two monads (T_1, η_1, μ_1) and (T_2, η_2, μ_2) one can construct the coproduct monad $(T_1 + T_2, \eta_+, \mu_+)$ that encompasses the side effects of both monads. The general construction turns out to be quite complex. The interested reader should see their paper [6].

References

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

[2] Jon Beck. *Distributive laws*, pages 119–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 1969.

[3] Mark Jones and Luc Duponcheel. Composing monads. Yaleu/dcs/rr-1004, Yale University, December 1993.

[4] 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.

[5] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

[6] Christoph Lüth and Neil Ghani. Composing monads using coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 133–144, New York, NY, USA, 2002. ACM.

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

[8] Ernie Manes and Philip Mulry. Monad compositions i: general constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.

[9] Ernie Manes and Philip Mulry. Monad compositions ii: Kleisli strength. *Mathematical. Structures in Comp. Sci.*, 18(3):613–643, June 2008.

[10] Oleksandr Manzyu. Calculating monad transformers with category theory. `https://oleksandrmanzyuk.files.wordpress.com/2012/02/calc-mts-with-cat-th1.pdf`, February 2012.

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

[12] Eugenio Moggi. An abstract view of programming languages. ECS-LFCS-90-113 90, LFCS University of Edinburgh, 1989.

[13] 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.

[14] 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.

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