An introduction to category theory, category theory monads, and their relationship to functional programming

Jonathan M.D. Hill and Keith Clarke*
Department of Computer Science
Queen Mary & Westfield College
University of London

Abstract

Incorporating imperative features into a purely functional language has become an active area of research within the functional programming community [10, 7, 12]. One of the techniques gaining widespread acceptance as a model for imperative functional programming is monads [13, 9]. The purpose of this technical report is to give a category theoretic introduction to monads, and to explore the relationship to what functional programmers term a monad.

Keywords: Monads; Category theory; Kleisli triple; Imperative functional programming.

1 Motivation

This paper stems from the desire for an understanding of Moggi's work on the computational λ-calculus and Monads [9, 8]. The presentation here owes much to the papers of Wadler [13, 14], and the basic texts on category theory [11, 1, 4, 6, 2].

2 Basic category theory and monads

Category theory is concerned with the observation that many of the properties from algebra can be simplified by a presentation in terms of diagrams containing arrows. A category $\mathcal{C}$ comprises of a collection of objects and a collection of morphisms (also called arrows). If the morphism f has a domain A and a co-domain B, then the morphism is written as $f : A \rightarrow B$ or $A \xrightarrow{f} B$. The collection of all morphisms in the category $\mathcal{C}$ with a domain A and co-domain B is called a *hom-set* written $\mathcal{C}(A, B)$. The composition of morphisms must obey the associative law such that for any morphism $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$ then $h \circ (g \circ f) \equiv (h \circ g) \circ f$ must hold. Another requirement of a category is that for each object in the category $\mathcal{C}$ (written $Obj(\mathcal{C})$), there exists an identity morphism id_A in the hom-set $\mathcal{C}(A, A)$ such that for any morphism $f : A \rightarrow B$, $f \circ id_A \equiv f$ and $id_B \circ f \equiv f$. These two requirements of a category provide an introduction to our first pair of commuting diagrams, which provide a pictorial representation of the relationships between objects and morphisms *within a single category.* The following diagrams express the identity and associative laws of the morphisms f, g, and h in the category $\mathcal{C}$:

[*] Email: {Jon.Hill,keithc}@dcs.qmw.ac.uk

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g \circ f"'] & B \arrow[d, "h \circ g"] \arrow[dl, dashed, "g"] \\
C \arrow[r, "h"] & D
\end{tikzcd}$$

in category $\mathcal{C}$

$$\begin{tikzcd}
A \arrow[r, "id_A"] \arrow[d, "f"'] \arrow[dr, "f"] & A \arrow[d, "f"] \\
B \arrow[r, "id_B"] & B
\end{tikzcd}$$

in category $\mathcal{C}$

Category theory has a potentially important application to data-parallel programming. As associativity of morphism composition is one of the building blocks of a category, morphisms can be used as the higher order arguments to either parallel fold or scan.

2.1 Functors: mapping between categories

Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* $F : \mathcal{C} \rightarrow \mathcal{D}$ is a mapping between the two categories and consists of an object mapping and a morphism mapping. The object mapping F_{obj} defines a relationship between each object in the category $\mathcal{C}$, and objects in the category $\mathcal{D}$, written as $F_{obj} : Obj(\mathcal{C}) \rightarrow Obj(\mathcal{D})$. By convention, if A is an object in $\mathcal{C}$, then $F_{obj}(A)$ is the corresponding object in category $\mathcal{D}$. The morphism mapping is an analogous relationship between the morphisms of two categories. Given each morphism in the hom-set $\mathcal{C}(A, B)$, the morphism mapping F_{mor}, defines a relationship with the morphisms in the hom-set $\mathcal{D}(F_{obj}(A), F_{obj}(B))$, written as $F_{mor} : F_{obj}(A) \rightarrow F_{obj}(B)$. A functor is required to preserve the structure of the compositions of morphisms such that given the identity morphism id_A, and two morphisms f and g in the category $\mathcal{C}$, the following equations hold:

$$\begin{aligned}
F_{mor}(id_A) &= id_{F_{obj}(A)} \\
F_{mor}(f \circ g) &= F_{mor}(f) \circ F_{mor}(g)
\end{aligned}$$

Category theorists enjoy omitting unnecessary syntactic baggage from their notation. By convention, the subscripts *obj* and *mor* can be dropped from a functor as it is always clear from the context whether an object or morphism mapping is being described. The commuting diagram on the left below defines the associativity of composition for the morphisms f and g in the category $\mathcal{C}$; the commuting diagram on the right defines the corresponding relationship between the morphisms $F(f)$ and $F(g)$ in category $\mathcal{D}$:

$$\begin{tikzcd}
& B \arrow[dr, "g"] & \\
A \arrow[ur, dashed, "f"] \arrow[rr, "g \circ f"] & & C
\end{tikzcd}$$

in category $\mathcal{C}$

$$\begin{tikzcd}
& F(B) \arrow[dr, "F(g)"] & \\
F(A) \arrow[ur, dashed, "F(f)"] \arrow[rr, "F(g \circ f)"] & & F(C)
\end{tikzcd}$$

in category $\mathcal{D}$

Like morphisms, functors can be composed, such that for any two functors $F : \mathcal{A} \rightarrow \mathcal{B}$ and $G : \mathcal{B} \rightarrow \mathcal{C}$, then the composition $G \circ F$ is a functor that maps objects and morphisms in the category $\mathcal{A}$ to corresponding elements in $\mathcal{C}$. By convention, if A is an object in the category $\mathcal{A}$, then the corresponding element in category $\mathcal{C}$ is written as $GF(A)$ in favour of the more obvious $G(F(A))$.

2.2 Natural transformations: sliding between categories

Given two functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$ which form two mappings between the same categories, a *natural transformation* between F and G is a structure preserving translation from one functor to another. The natural transformation τ from F to G can be thought of as a way of "sliding" the picture defining the functor F onto that of G (the left commuting diagram is in the category $\mathcal{C}$, and the sliding commuting diagram on the right is in category $\mathcal{D}$):

$$\begin{tikzcd}
A \arrow[dr, "f"] \arrow[dd, "g \circ f"'] & \\
& B \arrow[dl, dotted, "g"] \\
C &
\end{tikzcd}$$

in category $\mathcal{C}$

$$\begin{tikzcd}
F(A) \arrow[rr, dashed, "\tau_A"] \arrow[dr, "F(f)"] \arrow[dd, "F(g \circ f)"'] & & G(A) \arrow[dr, "G(f)"] \arrow[dd, "G(g \circ f)"'] & \\
& F(B) \arrow[rr, dashed, "\tau_B"] \arrow[dl, dotted, "F(g)"] & & G(B) \arrow[dl, dotted, "G(g)"] \\
F(C) \arrow[rr, dashed, "\tau_C"] & & G(C) &
\end{tikzcd}$$

in category $\mathcal{D}$

Concentrating on a single morphism $f : A \longrightarrow B$, then a natural transformation τ from F to G, written $\tau : F \longrightarrow G$ can be defined by assigning to each object A in $Obj(\mathcal{C})$ a *morphism* $\tau_A : F(A) \longrightarrow G(A)$, such that for every morphism f in the hom-set $\mathcal{C}(A, B)$, the following diagram commutes:

$$\begin{tikzcd}
A \arrow[d, "f"'] \\
B
\end{tikzcd}
\qquad
\begin{tikzcd}
F(A) \arrow[r, "\tau_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\
F(B) \arrow[r, "\tau_B"] & G(B)
\end{tikzcd}$$

in category $\mathcal{C}$ in category $\mathcal{D}$

The morphisms τ_A and τ_B in the diagram above are called the *components* of the natural transformation τ. A common pictorial representation of a natural transformation is a diagram of the form:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \tau"] & \mathcal{D}
\end{tikzcd}$$

which can be read as the two functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$, with the natural transformation τ that provides a "translation" between the two.

2.3 Composing natural transformations

Natural transformations can be composed either "vertically" or "horizontally". Given two natural transformations τ and σ, the diagram below shows a vertical composition:

$$\begin{tikzcd}[row sep=large, column sep=huge]
\mathcal{A} \arrow[r, bend left=60, "F"{name=F}] \arrow[r, "G"{name=G, description}] \arrow[r, bend right=60, "H"'{name=H}] & \mathcal{B} \arrow[from=F, to=G, Rightarrow, "\sigma"] \arrow[from=G, to=H, Rightarrow, "\tau"]
\end{tikzcd}$$

For every object A in the category $\mathcal{A}$, then by the definition of a natural transformation of the previous section, the following commuting diagram in the category $\mathcal{B}$ can be drawn (remember that σ_B and τ_B are the *morphisms* that are the components of the natural transformation):

$$\begin{tikzcd}
A \arrow[d, "f"'] \\
B
\end{tikzcd}$$

in category $\mathcal{A}$

$$\begin{tikzcd}[column sep=huge, row sep=huge]
F(A) \arrow[r, "\sigma_A"] \arrow[d, "F(f)"'] \arrow[rr, bend left=30, "(\tau \circ \sigma)_A"] & G(A) \arrow[r, "\tau_A"] \arrow[d, "G(f)"'] & H(A) \arrow[d, "H(f)"'] \\
F(B) \arrow[r, "\sigma_B"] \arrow[rr, bend right=30, "(\tau \circ \sigma)_B"'] & G(B) \arrow[r, "\tau_B"] & H(B)
\end{tikzcd}$$

in category $\mathcal{B}$

The commuting diagram on the right above can be used as the basis for a new category $\mathcal{F}$ that has functors as its objects, and natural transformations as its morphisms—*the functor category*. The prior commuting diagram can now be written as a commuting diagram in $\mathcal{F}$:

$$\begin{tikzcd}[column sep=huge, row sep=huge]
F \arrow[r, "\sigma"] \arrow[dr, "\sigma \circ \tau"'] & G \arrow[d, "\tau"] \\
& H
\end{tikzcd}$$

in category $\mathcal{F}$

Therefore given two natural transformations, τ and σ, the composition of the two $\tau \circ \sigma$ also forms a natural transformation between the functor F and H, written as $\sigma \circ \tau : F \longrightarrow H$.

Next we consider the horizontal composition of τ and σ, expressed by the following natural transformation diagram:

$$\begin{tikzcd}[column sep=huge]
\mathcal{A} \arrow[r, bend left=50, "F"{name=F}] \arrow[r, bend right=50, "G"'{name=G}] & \mathcal{B} \arrow[r, bend left=50, "H"{name=H}] \arrow[r, bend right=50, "K"'{name=K}] & \mathcal{C} \arrow[from=F, to=G, Rightarrow, "\tau"] \arrow[from=H, to=K, Rightarrow, "\sigma"]
\end{tikzcd}$$

For each object in the category $\mathcal{A}$ we consider the corresponding objects in the category $\mathcal{C}$. By the definition of a functor and a natural transformation, given an object A in category $\mathcal{A}$, then *one of* *the* corresponding objects in category $\mathcal{C}$ is $HF(A)$ (by composition of the functors on the top line of the natural transformation diagram); another object in category $\mathcal{C}$ is $HG(A)$. The morphism that describes the relationship between these two objects in $\mathcal{C}$ is $H(\tau_A)$, i.e., the morphism τ_A which is the component of the natural transformation τ, is used to select the morphism mapping from the functor H. Another interesting morphism in the diagram is between the objects $HG(A)$ and $KG(A)$ in the category $\mathcal{C}$. Applying the functor G to the object A in category $\mathcal{A}$ gives the object $G(A)$ in category $\mathcal{B}$. This object is then used to select the component of the natural transformation σ that describes the morphism $\sigma_{G(A)}$ between $HG(A)$ and $KG(A)$. Applying similar techniques to each of the remaining paths in the natural transformation diagram, gives the following commuting diagram in category $\mathcal{C}$:

$$\begin{tikzcd}
HF(A) \arrow[r, "H(\tau_A)"] \arrow[d, "\sigma_{F(A)}"'] & HG(A) \arrow[d, "\sigma_{G(A)}"] \\
FK(A) \arrow[r, "K(\tau_A)"] & KG(A)
\end{tikzcd}$$

in category $\mathcal{C}$

In a similar vein to the treatment of horizontal compositions, a new category $\mathcal{F}$ can be defined with functors as its objects and natural transformations as its morphisms. If $id_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ is the identity functor for the category $\mathcal{C}$ and $1_{\mathcal{C}} : id_{\mathcal{C}} \rightarrow id_{\mathcal{C}}$ the identity natural transformation of that functor to itself, then by the composition of natural transformations $1_{\mathcal{D}} \circ \tau \equiv \tau \equiv \tau \circ 1_{\mathcal{C}}$. Now for the utterly revolting part—if F is a functor, then it also convenient to use the symbol F to represent the identity natural transformation from F to F, i.e., $F : F \rightarrow F$. If G, H, and K are also identity natural transformations from the appropriate functors, then the following commuting diagram in $\mathcal{F}$ can be given which induces four new natural transformations $H\tau : HF \rightarrow HG$, $\sigma G : HG \rightarrow KG$, $K\tau : FK \rightarrow HK$, and $\sigma F : HF \rightarrow FK$.

$$\begin{tikzcd}
HF \arrow[r, "H\tau"] \arrow[d, "\sigma F"'] & HG \arrow[d, "\sigma G"] \\
FK \arrow[r, "K\tau"] & KG
\end{tikzcd}$$

in category $\mathcal{F}$

By the commutativity of the diagram, the following equivalence of the compositions of the morphisms of $\mathcal{F}$ (i.e., natural transformations) hold, $(\sigma \circ G) \circ (H \circ \tau) \equiv (K \circ \tau) \circ (\sigma \circ F)$. ♠ **ToDo:** *Should this be* $\bullet$*?* ♠

2.4 Monads in category theory

The principle underlying Moggi's work on monads and the computational lambda calculus is the distinction between simple data valued functions and functions that perform computations. A data-valued function is one in which the value returned by the function is determined solely by the values of its arguments. In contrast, a function that performs a *computation* can encompass ideas such as side-effects or non-determinism, which implicitly produce more results as a consequence of an application of the function than the result explicitly returned.

In category theory, a monad over the category $\mathcal{C}$ is the triple (T, η, μ), where T is the *endofunctor*[1] $T : \mathcal{C} \to \mathcal{C}$, and η and μ are two natural transformations. As already mentioned, the convention for writing the composition of functors $T \circ T$ is TT, which is simplified further here to the exponentiation notation T^2. Using this abbreviation, the natural transformations of the monad are defined as $\eta : id_c \to T$ and $\mu : T^2 \to T$:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "id_{\mathcal{C}}", ""{name=U, below}] \arrow[r, bend right=50, "T"', ""{name=D}] \arrow[from=U, to=D, "\eta"] & \mathcal{C}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "T^2", ""{name=U, below}] \arrow[r, bend right=50, "T"', ""{name=D}] \arrow[from=U, to=D, "\mu"] & \mathcal{C}
\end{tikzcd}$$

Moggi's work on monads views the endofunctor T as a mapping between all the objects $Obj(\mathcal{C})$ of the category $\mathcal{C}$ which are to be viewed as the set of all values of type τ, to a corresponding set of objects $T(Obj(\mathcal{C}))$ which are to be interpreted as the set of computations of type τ. The natural transformation η can be thought of as an operator that includes values into a computation; the natural transformation μ "flattens" a computation of computations into a single computation.

For T, η, μ to be termed a monad, three laws called the *associative law of a monad*, and the *left and right identity laws* must hold. Rather than giving the laws and attempting to justify them, the laws are presented in a step-by-step manner by considering various compositions of the monads components T, η, and μ.

We first consider the associative law by investigating the characteristics of the flattening natural transformation μ. For each object A in the category $\mathcal{C}$ we consider the corresponding elements in the category $\mathcal{C}$ after applying a combination of the morphism mappings of the endofunctors T and T^2, and the components of the natural transformation μ. The diagram on the left below defines a left combination of the endofunctor T with the natural transformation μ. By the definition of a functor and a natural transformation, given an object A in category $\mathcal{C}$, then *one of the* corresponding objects in category $\mathcal{C}$ is $T^2T(A)$ (by composition of the functors on the top line of the natural transformation diagram); another object in category $\mathcal{C}$ is $TT(A)$. The morphism that describes the relationship between these two objects in $\mathcal{C}$ is $\mu_{T(A)}$, i.e., the object $T(A)$ from the object mapping of the functor T, is used to select the component of the natural transformation μ. The diagram on the right shows this relationship pictorially:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "T"] & \mathcal{C} \arrow[r, bend left=50, "T^2", ""{name=U, below}] \arrow[r, bend right=50, "T"', ""{name=D}] \arrow[from=U, to=D, "\mu"] & \mathcal{C}
\end{tikzcd}
\qquad
\begin{tikzcd}
T^2T(A) \arrow[r, "\mu_{T(A)}", "\text{in category } \mathcal{C}"'] & TT(A)
\end{tikzcd}$$

A similar technique can be used in the right-wards combination of the endofunctor T with the natural transformation μ:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "T^2", ""{name=U, below}] \arrow[r, bend right=50, "T"', ""{name=D}] \arrow[from=U, to=D, "\mu"] & \mathcal{C} \arrow[r, "T"] & \mathcal{C}
\end{tikzcd}
\qquad
\begin{tikzcd}
TT^2(A) \arrow[r, "T(\mu_A)", "\text{in category } \mathcal{C}"'] & TT(A)
\end{tikzcd}$$

If we now define a new category $\mathcal{F}$ with functors as its objects, and natural transformations as its morphisms, then the commuting diagram on the left below which combines the morphisms of the prior diagrams can be used as the basis for the diagram on the right—the morphism T on the right is the identity natural transformation from the endofunctor T to T:

[1] i.e., inside functor—a functor with a mapping to and from the same category.

$$\begin{tikzcd}
T^3(A) \arrow[r, "T(\mu_A)"] \arrow[d, "\mu_{T(A)}"'] & T^2(A) \arrow[d, "\mu_A"] \\
T^2(A) \arrow[r, "\mu_A"] & T(A)
\end{tikzcd}$$

in category $\mathcal{C}$

$$\begin{tikzcd}
T^3 \arrow[r, "T \circ \mu"] \arrow[d, "\mu \circ T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

in category $\mathcal{F}$

By the commutativity of the diagram, $\mu \circ (T \circ \mu) \equiv \mu \circ (\mu \circ T)$, which is termed the *associative law of the monad.*

Next we consider the *left and right identity laws* by investigating the characteristics of the injective operator η, in a similar manner to the treatment of μ. For each object A in the category $\mathcal{C}$ we consider the corresponding elements in the category $\mathcal{C}$ after applying η and the endofunctor T in a leftwards manner:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "T"] & \mathcal{C} \arrow[r, bend left, "id_{\mathcal{C}}"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{C}
\end{tikzcd}$$

$$\begin{tikzcd}
id_{\mathcal{C}}(T(A)) \arrow[r, "\eta_{T(A)}"] & TT(A)
\end{tikzcd}$$

in category $\mathcal{C}$

, a similar technique can be applied to a right-wards combination of the endofunctor T with the natural transformation η:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "id_{\mathcal{C}}"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{C} \arrow[r, "T"] & \mathcal{C}
\end{tikzcd}$$

$$\begin{tikzcd}
T(id_{\mathcal{C}}(A)) \arrow[r, "T(\eta_A)"] & TT(A)
\end{tikzcd}$$

in category $\mathcal{C}$

If we now define a new category $\mathcal{F}$ with functors as its objects, and natural transformations as its morphisms, then the commuting diagram on the left below which combines the morphisms of the prior diagrams can be used as the basis of the diagram on the right—the morphism T on the right is the identity natural transformation from the endofunctor T to T:

$$\begin{tikzcd}
T(id_{\mathcal{C}}(A)) \arrow[r, "T(\eta_A)"] \arrow[dr, "="'] & T^2(A) \arrow[r, "\eta_{T(A)}"] \arrow[d, "\mu_A"] & id_{\mathcal{C}}(T(A)) \arrow[dl, dashed, "="] \\
& T(A) &
\end{tikzcd}$$

in category C

$$\begin{tikzcd}
Tid_{\mathcal{F}} \arrow[r, "T\eta"] \arrow[dr, "1_{\mathcal{F}}"'] & T^2 \arrow[d, "\mu"] & id_{\mathcal{F}}T \arrow[l, "\eta T"'] \arrow[dl, dashed, "1_{\mathcal{F}}"] \\
& T &
\end{tikzcd}$$

in category F

By the commutativity of the diagram, $\mu \circ (\eta \circ T) \equiv 1_{\mathcal{F}} \equiv \mu \circ (T \circ \eta)$, which is termed the *left and right unit laws of a monad.*

2.5 The Kleisli Star

Given the monad (T, η, μ) in category $\mathcal{C}$, consider for each morphism $f : A \rightarrow T(B)$ a new morphism $f^* : T(A) \rightarrow T(B)$, where $\_^*$ is the "extension" operator that lifts the domain A of the morphism $f : A \rightarrow T(B)$ to a computation $T(A)$. In the context of Moggi's work on computations,

f is a function from values to computations, whereas f^* is a function from computations to computations. The expression $g^* \circ f$, where $f : A \rightarrow T(B)$ and $g : B \rightarrow T(C)$ is interpreted as applying f to some value a to produce some computation $f\ a$; this computation is evaluated to produce some value b, and g is applied to b to produce a computation as a result. The monad (T, η, μ) and the extension operator can be used to create the following commuting diagrams in $\mathcal{C}$:

$$\begin{tikzcd}
 & A \arrow[dl, dotted, "\eta_A"'] \arrow[dr, "f"] & \\
T(A) \arrow[rr, "f^*"] \arrow[d, "(\mu_C \circ T(g) \circ f)^*"'] & & T(B) \arrow[dll, dotted, "g^*"] \arrow[d, "T(g)"] \\
T(C) & & T^2(C) \arrow[ll, "\mu_C"']
\end{tikzcd}$$

in category $\mathcal{C}$

$$\begin{tikzcd}
 & T(A) \arrow[d, "\eta_A^*"] \\
A \arrow[ur, "\eta_A"] \arrow[r, "\eta_A"] \arrow[d, "\eta_A"'] & T(A) \arrow[dl, "id_{T(A)}"] \\
T(A) &
\end{tikzcd}$$

in category $\mathcal{C}$

The Kleisli triple $(T_{obj}, \eta, \_^*)$ can be constructed from the monad (T, η, μ), where T_{obj} is the restriction of the endofunctor T to objects; $(g : B \rightarrow T(C))^* \equiv \mu_C \circ T(g)$ from the commutativity of the diagram above left. Kleisli triples can be thought of as a different syntactic presentation of a monad, as their is a one-to-one correspondence between a Kleisli triple and a monad (see [6, page 143] for a proof). In a similar vein to the monad laws, the following laws can be constructed from the commuting diagrams above:

$$\begin{array}{lll}
\textit{Left Unit}: & f^* \circ \eta_A & \equiv f \\
\textit{Right Unit}: & \eta_A^* \circ h & \equiv id_{T(A)} \circ h \\
 & \eta_A^* & \equiv id_{T(A)} \\
\textit{Associativity}: & (g^* \circ (f^* \circ h)) & \equiv (g^* \circ f)^* \circ h \\
 & g^* \circ f^* & \equiv (g^* \circ f)^* \\
 & \text{by assoc. of } \circ \text{, and eliding } h. &
\end{array}$$

2.6 Monads in functional programming

Wadler [13] adapted Moggi's ideas of using monads to structure the semantics of computations into a tool for structuring functional programs. Given a monad (T, η, μ), the functor T and the two natural transformations are modelled in a functional programming language by a type constructor[2], a function *map* parameterised on the type constructor M (see [3] for parametrising *map* in this way), and two polymorphic functions.

Given a functor $T : \mathcal{C} \rightarrow \mathcal{D}$, a morphism $f : A \rightarrow B$, and the set of objects A in category $\mathcal{C}$, the corresponding elements in the category $\mathcal{D}$ will be the objects $T(A)$ and the morphisms $T(f) : T(A) \rightarrow T(B)$. The object mapping part of a functor is represented in a functional language by a type-constructor. For example, if an object A has the type α, then the object $T(A)$ have the type $M\ \alpha$; where M is a type constructor that represents a "computation". The morphism mapping of the functor is modelled by a function analogous to *map*:

$$map :: (\alpha \rightarrow \beta) \rightarrow (M\ \alpha \rightarrow M\ \beta)$$

[2] The type constructor M on lists is normally written as $M\ \alpha \equiv [\alpha]$.

The corresponding morphism of f in the category $\mathcal{D}$ is $T(f)$ which is modelled in a functional language by the curried function application *map* f.

Finally, a natural transformation can be thought of as a family of arrows from each object in a category (the components of a natural transformation), to objects in another category. A natural transformation is therefore *similar* to a polymorphic function, and as a consequence η and μ are written as the polymorphic functions *unit* and *join* of type:

$$\begin{array}{lll} \mathit{unit} & :: & \alpha \rightarrow M\ \alpha \\ \mathit{join} & :: & M\ (M\ \alpha) \rightarrow M\ \alpha \end{array}$$

In summary, given a category theory monad (T, η, μ), a functional programming monad is represented by the quadruple $(M, \mathit{map}, \mathit{unit}, \mathit{join})$, which must obey the monad laws of section 2.4 which are re-expressed as:

$$\begin{array}{lll} \textit{Left Unit}: & \mathit{join} \circ \mathit{unit} & \equiv \mathit{id} \\ \textit{Right Unit}: & \mathit{join} \circ \mathit{map}\ \mathit{unit} & \equiv \mathit{id} \\ \textit{Associativity}: & \mathit{join} \circ \mathit{map}\ \mathit{join} & \equiv \mathit{join} \circ \mathit{join} \end{array}$$

2.7 How people really use monads in functional programming

In Wadler's more recent papers [14, 10], and with the onset of the application of monads as a method of incorporating imperative features into a purely functional language, the current use of monads bears a closer resemblance to Kleisli triples. The triple $(M, \mathit{unit}, \star)$ forms a monad when the following laws hold:

$$\begin{array}{lll} \textit{Left Unit}: & \mathit{unit}\ a \star (\lambda b \rightarrow n) & \equiv n[a/b] \\ \textit{Right Unit}: & m \star (\lambda b \rightarrow \mathit{unit}\ b) & \equiv m \\ \textit{Associativity}: & m \star ((\lambda a \rightarrow n) \star (\lambda b \rightarrow m)) & \equiv (m \star (\lambda a \rightarrow n)) \star (\lambda b \rightarrow m) \end{array}$$

As with the monads in the previous section, M is a type-constructor such that an expression of type $M\ \alpha$ can be thought of as a computation delivering an object of type α as a result. *unit* is the inclusion function of the monad (same as η) of type $\alpha \rightarrow M\ \alpha$ that converts a value v of type α into a computation that does nothing but deliver the value v as a result. The expression $f \star g$ is the same as $g^* \circ f$ of the Kleisli triple, and has the type $\star :: M\alpha \rightarrow (\alpha \rightarrow M\beta) \rightarrow M\beta$.

3 An historical aside

Monads are typically equated with single-threadedness, and are therefore used as a technique for incorporating imperative features into a purely functional language. Category theory monads have little to do with single-threadedness; it is the sequencing imposed by composition that ensures single-threadedness. In a Wadler-ised monad this is a consequence of bundling the Kleisli star and flipped compose into the `bind` operator. There is nothing new in this connection. Peter Landin in his Algol 60 paper [5] used functional composition to model semi-colon. Semi-colon can be thought of as a state transforming operator that threads the state of the machine throughout a program. The work of Peyton-Jones and Wadler [10] has turned full circle back to Landin's earlier work as their use of Moggi's sequencing monad enables real side-effects to be incorporated into monad operations such as `print`. This is similar to Landin's implementation of his sharing machine where the *assignandhold* function can side-effect the store of the sharing machine because of the sequencing imposed by functional composition. Landin defined that "*Imperatives are treated as null-list producing functions*"[3]. The *assignandhold* imperative is subtly different in that it enables Algol's compound statements to be handled. The function takes a store location and a value as its argument, and performs the assignment to the store of the sharing machine, returning the value assigned as a result of the function. Because Landin assumed applicative order reduction, the

[3] In Landin's paper, () is the syntactic representation of the empty list and not the unit.

K−combinator[4] was used to return `()`, and the imperative was evaluated as a side effect by the unused argument of the **K**−combinator. Statements are formed by wrapping such an imperative in a lambda expression that takes `()` as an argument. Two consecutive Algol-60 assignments would be encoded in the lambda calculus as:

Algol 60	Lambda Calculus
`x:= 2;`	$((\lambda() \rightarrow \mathbf{K}\ ()\ (\mathit{assignandhold}\ x\ 2))\ \tilde{\circ}$
`x:= -3;`	$(\lambda() \rightarrow \mathbf{K}\ ()\ (\mathit{assignandhold}\ x\ (-3))))\ ()$

By using a lambda with `()` as its parameter, `()` can be thought of as the "state of the world" that is threaded throughout a program by functional composition.

4 Conclusions

At first sight the objects functional programmers term a monad, and those which catageory theorists term a monad bear little resemblance. Hopefully this technical report will help those unfamilliar with "monads" to bridge the gap between the two.

References

[1] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures.* The MIT Press, 1991.

[2] M. Barr and C. Wells. *Toposes, Triples and Theories.* Number 278 in Comprehensive studies in mathematics. Springer-Verlag, 1985.

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Lisp and functional programming*, 1992. Enables things like map to be overloaded on vectors, lists, etc..

[4] P. M. Cohn. *Universal Algebra.* Harper & Row, 1965.

[5] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8(2):89–101, February 1965. Part 2 in CACM Vol 8(2) 1965, pages 158–165.

[6] Saunders Mac Lane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[7] John Launchbury. Lazy imperative programming. In *Proc ACM Sigplan workshop on State in Programming Languages*, pages 46–56, June 1993.

[8] E. Moggi. Computational lambda-calculus and monads. In *IEEE symposium on Logic in computer science*, 1989. University of Edinburgh LFCS technical report ECS-LFCS-88-66 is an extended version of the paper.

[9] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1990.

[10] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, 1993.

[11] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[12] Jean-Pierre Talpin. *Aspects théoriques et praticques de l'inférence de type et d'effets.* PhD thesis, L'Ecole nationale supérieure des mines de Paris, May 1993. (Thesis in English).

[4] $\mathbf{K} = \lambda x\ y \rightarrow x$

[13] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and functional programming*, ACM Conferences, pages 61–78. ACM, June 1990.

[14] Philip Wadler. The essence of functional programming. In *ACM Symposium on Principles of Programming Languages*, January 1992.