Notions of Category Theory in Functional Programming

by

Shauna C. A. Gammon

B.Sc., Memorial University of Newfoundland, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Mathematics)

The University Of British Columbia

August, 2007

Abstract

We present a detailed examination of applications of category theory to functional programming languages, with a primary focus on monads in Haskell. First, we explore E. Moggi's work in categorical semantics, which provides the theoretical foundation for employing monads in functional languages. In particular, we examine his use of Kleisli triples to model notions of computation. We then study P. Wadler's implementation of Moggi's ideas as a means to mimic side-effects in the purely functional language Haskell. We explicitly demonstrate the connections between Kleisli triples, category-theoretic monads, and Haskell monads. It is our principal aim to provide a coherent translation between the abstracted concept of monads that exists in category theory, and the formulation of monads as type-constructors that is implemented in Haskell.

Table of Contents

Acknowledgements

I would like to acknowledge my supervisor, John MacDonald, for his guidance, encouragement, and kindness. In addition, I am grateful to my second thesis examiner, Kee Lam, for his helpful suggestions. Finally, I would like to acknowledge Steve Wolfman and the ELMers for their enthusiastic and enlightening dinnertime discussions.

Dedication

To Leonard, Art & Adam.

Chapter 1

Introduction

We begin with a brief overview of the key concepts from category theory that will be needed to investigate monads. Notably, we will not provide an independent section pertaining to background Haskell material, choosing instead to offer relevant introductory concepts and syntax within the contexts they are needed. We refer the Haskell-interested reader to [4] for an excellent introduction.

1.1 Background

A *monoid* $(M, \cdot, e)$ is a set M together with an associative binary operation $\cdot : M \times M \to M$ and identity element $e \in M$, such that $\forall m \in M, e \cdot m = m = m \cdot e$. The properties of this familiar algebraic construct can be expressed via commutative diagrams—which abound in category theory—as follows. Let $p_1 : M \times M \to M$ and $p_2 : M \times M$ be projections defined by: $\forall m, m' \in M$, $p_1(m, m') = m = p_2(m', m)$. We define a monoid M to be a set M together with functions $\mu : M \times M \to M$ and $\eta : 1 \to M$, where $1 = \{e\}$ with $e \in M$, such that the following diagrams commute:

$$\begin{tikzcd}
M \times M \times M \arrow[r, "1\times\mu"] \arrow[d, "\mu\times 1"'] & M \times M \arrow[d, "\mu"] \\
M \times M \arrow[r, "\mu"] & M
\end{tikzcd}
\qquad
\begin{tikzcd}
1 \times M \arrow[r, "\eta\times 1"] \arrow[d, "p_2"'] & M \times M \arrow[d, "\mu"] & M \times 1 \arrow[l, "1\times\eta"'] \arrow[d, "p_1"] \\
M \arrow[r, equal] & M & M \arrow[l, equal]
\end{tikzcd}.$$

The theoretical settings in which we work are categories: the structures central to the study of category theory. A *category* $\mathcal{C}$ is a 6-tuple $(O_\mathcal{C}, A_\mathcal{C}, \mathrm{dom}, \mathrm{cod}, \circ, \mathrm{id})$, where:

- $O_\mathcal{C}$ and $A_\mathcal{C}$ are sets of objects and morphisms (or arrows), respectively;
- dom and cod are functions that assign to each morphism $f \in A$ objects $\mathrm{dom}f$ and $\mathrm{cod}f$, the domain and codomain of f, respectively. For an arrow f with $\mathrm{dom}f = a$, $\mathrm{cod}f = b$, we write $f : a \to b$;

- id is a function that assigns to each object $c \in O_{\mathcal{C}}$ an identity morphism $\mathrm{id}_c \in A_{\mathcal{C}}$, with $\mathrm{id}_c : c \rightarrow c$ satisfying the (standard algebraic) identity law;
- $\circ$ is an associative binary operation that assigns to a pair of morphisms $(f : a \rightarrow b, g : b \rightarrow c) \in A_{\mathcal{C}} \times A_{\mathcal{C}}$ a morphism $g \circ f : a \rightarrow c$.

The associativity of morphism composition can be used to prove the following useful (albeit, simple) property of commutative digrams.

Theorem 1.1.1. *Let $A, B, C, D, E, F \in O_{\mathcal{C}}$ and $f, g, h, q, r, s, t \in A_{\mathcal{C}}$, with domains and codomains as shown below. If the squares $ABED$ and $BCFE$ are commutative, then the outer rectangle $ACFD$, is also commutative.*

$$\begin{tikzcd}
A \arrow[r, "q"] \arrow[d, "f"'] & B \arrow[r, "r"] \arrow[d, "g"'] & C \arrow[d, "h"'] \\
D \arrow[r, "s"] & E \arrow[r, "t"] & F
\end{tikzcd}$$

Proof. The commutativity of the left-hand square implies $g \circ q = s \circ f$, and the commutativity of the right-hand square implies $h \circ r = t \circ g$. Thus, by the associativity of $\circ$, we have $h \circ (r \circ q) = (h \circ r) \circ q = (t \circ g) \circ q = t \circ (g \circ q) = t \circ (s \circ f) = (t \circ s) \circ f$. ■

Many constructions in category theory capture set-theoretic ideas with the language of objects and arrows. An example that we will encounter in the context of computational semantics is that of a terminal object, which is the category-theoretic generalization of a singleton set. An object $\mathbf{1}$ in a category $\mathcal{C}$ is a *terminal object* if for every $\mathcal{C}$-object c there is exactly one arrow from c to $\mathbf{1}$ in $\mathcal{C}$. Any two terminal objects in a category $\mathcal{C}$ must be isomorphic because if $\mathbf{1}$ and $\mathbf{1}'$ are both terminal objects then there must exist $\mathcal{C}$-arrows $f : \mathbf{1} \rightarrow \mathbf{1}'$ and $g : \mathbf{1}' \rightarrow \mathbf{1}$. Since $\mathbf{1}$ and $\mathbf{1}'$ are terminal, $\mathrm{id}_{\mathbf{1}}$ and $\mathrm{id}_{\mathbf{1}'}$ are the only arrows from $\mathbf{1} \rightarrow \mathbf{1}$ and $\mathbf{1}' \rightarrow \mathbf{1}'$, respectively. Thus, $f \circ g = \mathrm{id}_{\mathbf{1}}$, $g \circ f = \mathrm{id}_{\mathbf{1}'}$ and so $f : \mathbf{1} \cong \mathbf{1}'$.

A functor is a structure-preserving map between categories. Explicitly, a (covariant) *functor* $F : \mathcal{C} \rightarrow \mathcal{D}$ is a function that satsfies:

- $\forall c \in O_{\mathcal{C}}, \exists d \in O_{\mathcal{D}}$, with $F(c) = d$;
- $\forall f : c_1 \rightarrow c_2 \in A_{\mathcal{C}}, \exists F(f) : F(c_1) \rightarrow F(c_2) \in A_{\mathcal{D}}$, such that:
  i. $\forall c \in O_{\mathcal{C}}, F(\mathrm{id}_c) = \mathrm{id}_{F(c)}$;
  ii. $F(g \circ f) = F(g) \circ F(f)$, whenever $g \circ f \in A_{\mathcal{C}}$ is defined.

There are several functors that will provide nice examples in our study of monads.

1. First, let $F : \mathbf{Set} \to \mathbf{Grp}$ map a set $X \in \mathbf{Set}$ to the free group Free(X) generated by the elements of X. Note that any function between sets X and Y can be viewed as a map between the generators of Free(X) and those of Free(Y), which extends to words in X, preserves composition and equivalences, and is the uniquely determined homomorphism between these groups with these extension properties. Thus, we can define F to be the *free group functor* by setting $F(f) = h_f$, where f is an arbitrary function between objects in **Set** and h_f is the corresponding group homomorphism as described.
2. We define another functor $U : \mathbf{Grp} \to \mathbf{Set}$ to be the *forgetful functor*, which maps a group $(G, \cdot)$ to its underlying set of elements and a group homomorphism to the corresponding function between the underlying sets.
3. The *power set functor* $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ is defined by mapping a set to its power set and a function $f : X \to Y$ between sets to the function $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y)$ induced by taking f-images of sets in $\mathcal{P}(X)$.

Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\tau : F \dot{\to} G$ is a mapping with the property that $\forall a \in O_{\mathcal{C}}, \exists \tau_a \in A_{\mathcal{D}}$ such that $\forall f : a \to b \in A_{\mathcal{C}}$, we have $\tau_b \circ F(f) = G(f) \circ \tau_a$. Intuitively, τ can be thought of as a map that uses the structure of $\mathcal{D}$ to slide F-diagrams onto G-diagrams. The situation is described by the following diagram, in which the square is commutative:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F"{name=U}] \arrow[r, bend right, "G"'{name=D}] & \mathcal{D} \arrow[from=U, to=D, "\tau"]
\end{tikzcd}$$

$$\begin{tikzcd}
a \arrow[d, "f"'] & Fa \arrow[r, "\tau_a"] \arrow[d, "Ff"'] & Ga \arrow[d, "Gf"'] \\
b & Fb \arrow[r, "\tau_b"] & Gb.
\end{tikzcd}$$

1.2 Composition of Natural Transformations

The definition of a monad includes compositions of natural transformations. There are two ways that natural transformations can be combined (depending on the domain and range functors of the natural transformations involved), which we now introduce.

1.2.1 Vertical Composition

Given any three functors $F, G, H : \mathcal{C} \to \mathcal{D}$ and natural transformations $\sigma : F \dot{\to} G$, $\tau : G \dot{\to} H$, viewed vertically as

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"{name=F}] \arrow[r, "G"{name=G, description}] \arrow[r, bend right=50, "H"'{name=H}] & \mathcal{D}, \arrow[from=F, to=G, Rightarrow, "\sigma"] \arrow[from=G, to=H, Rightarrow, "\tau"]
\end{tikzcd}$$

it is easy to see how to define a composite natural transformation $\tau \cdot \sigma : F \dot{\to} H$. For any object $c \in \mathcal{C}$, there exist $\mathcal{D}$-arrows $\sigma_c : Fc \to Gc$ and $\tau_c : Gc \to Hc$ (the c components of the given natural transformations). We define $(\tau \cdot \sigma)_c = \tau_c \circ \sigma_c$ to be the components of the vertical composition of natural transformations σ and τ. To show that such proposed components for $\tau \cdot \sigma$ indeed comprise a natural transformation, we must show commutivity of the outer rectangle in following diagram, for $f : b \to c$ in $\mathcal{C}$:

$$\begin{tikzcd}
Fc \arrow[r, "\sigma_c"] \arrow[d, "Ff"'] & Gc \arrow[r, "\tau_c"] \arrow[d, "Gf"'] & Hc \arrow[d, "Hf"'] \\
Fd \arrow[r, "\sigma_d"] & Gd \arrow[r, "\tau_d"] & Hd.
\end{tikzcd}$$

The respective naturalities of σ and τ imply the commutativity of the left and right squares, and the commutativity of the outer rectangle immediately follows from Theorem 1.1.1.

1.2.2 Horizontal Composition

We now define another type of composition for natural transformations. Consider functors $F, G : \mathcal{B} \to \mathcal{C}$, $H, J : \mathcal{C} \to \mathcal{D}$, and natural transformations $\sigma : F \dot{\to} G$, $\tau : H \dot{\to} J$, viewed horizontally as

$$\begin{tikzcd}
\mathcal{B} \arrow[r, bend left, "F"{name=F}] \arrow[r, bend right, "G"'{name=G}] & \mathcal{C} \arrow[r, bend left, "H"{name=H}] \arrow[r, bend right, "J"'{name=J}] & \mathcal{D}. \arrow[from=F, to=G, Rightarrow, "\sigma"] \arrow[from=H, to=J, Rightarrow, "\tau"]
\end{tikzcd}$$

Details are as follows: for any $\mathcal{B}$-object b, there exist $\mathcal{C}$-objects Fb and Gb, and $\mathcal{C}$-arrow $\sigma_b : Fb \to Gb$. We have two ways by which we can map these items to $\mathcal{D}$; namely, via H or J. Thus, we have the following diagram in $\mathcal{D}$, which is commutative by the naturality of τ :

$$\begin{tikzcd}
HFb \arrow[r, "\tau_{Fb}"] \arrow[d, "H\sigma_b"'] & JFb \arrow[d, "J\sigma_b"'] \\
HGb \arrow[r, "\tau_{Gb}"] & JGb
\end{tikzcd}$$

That is,

$$\begin{tikzcd}
\mathcal{B} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\downarrow\sigma"] & \mathcal{C} \arrow[r, "H"] & \mathcal{D}
\end{tikzcd} \quad \text{followed by} \quad \begin{tikzcd}
\mathcal{B} \arrow[r, "G"] & \mathcal{C} \arrow[r, bend left, "H"] \arrow[r, bend right, "J"'] \arrow[r, phantom, "\downarrow\tau"] & \mathcal{D}
\end{tikzcd}$$

equals

$$\begin{tikzcd}
\mathcal{B} \arrow[r, "F"] & \mathcal{C} \arrow[r, bend left, "H"] \arrow[r, bend right, "J"'] \arrow[r, phantom, "\downarrow\tau"] & \mathcal{D}
\end{tikzcd} \quad \text{followed by} \quad \begin{tikzcd}
\mathcal{B} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\downarrow\sigma"] & \mathcal{C} \arrow[r, "J"] & \mathcal{D}.
\end{tikzcd}$$

We define the components of the horizontal composition of σ and τ to be the composition of arrows from HFb to JGb as shown in the above diagram; explicitly, $(\tau \circ \sigma)_b = \tau_{Gb} \circ H\sigma_b = J\sigma_b \circ \tau_{Fb}$. To verify that these components do comprise a natural transformation, we must show that the following diagram commutes for $f : b \to c$ in $\mathcal{B}$:

$$\begin{tikzcd}
HFb \arrow[r, "(\tau\circ\sigma)_b"] \arrow[d, "HFf"'] & JGb \arrow[d, "JGf"'] \\
HFc \arrow[r, "(\tau\circ\sigma)_c"] & JGc.
\end{tikzcd}$$

But given the definition of the components for $(\tau \circ \sigma)_c$, this diagram can be broken into two squares:

$$\begin{tikzcd}
HFb \arrow[r, "H\sigma_b"] \arrow[d, "HFf"'] & HGb \arrow[r, "\tau_{Gb}"] \arrow[d, "HGf"'] & JGb \arrow[d, "JGf"'] \\
HFc \arrow[r, "H\sigma_c"] & HGc \arrow[r, "\tau_{Gc}"] & JGc.
\end{tikzcd}$$

Since σ is natural, we have $Gf \circ \sigma_b = \sigma_c \circ Ff$. By mapping these objects and arrows to $\mathcal{D}$ with the functor H, we establish the commutativity of the left-hand square. Since $Gf : Gb \to Gc$ is a $\mathcal{C}$-arrow, the naturality of τ implies the commutativity of the right-hand square. Thus, the above rectangle commutes.

A Note on Notation

We take this opportunity to emphasize a notational usage of key import in the study of monads. Consider the following arrangement:

$$\begin{tikzcd} \mathcal{B} \arrow[r, bend left, "F"{name=U}] \arrow[r, bend right, "F'"'{name=D}] & \mathcal{C} \arrow[r, "T"] & \mathcal{D}. \arrow[from=U, to=D, Rightarrow, "\mu"] \end{tikzcd}$$

Let $b \in O_{\mathcal{B}}$. Then, $\mu_b : Fb \to F'b$ is an arrow in $\mathcal{C}$ and $T(\mu_b) : TFb \to TF'b$ is an arrow in $\mathcal{D}$. We thus let $T\mu : TF \dot{\to} TF'$ denote the natural transformation with such $\mathcal{D}$-arrows as its components: $(T\mu)_b = T(\mu_b)$. We thus recognize that $T\mu$ is an example of horizontal composition. Specifically, $T\mu = 1_T \circ \mu$, where 1_T is the identity natural transformation on the functor T :

$$\begin{tikzcd} \mathcal{B} \arrow[r, bend left, "F"{name=U1}] \arrow[r, bend right, "F'"'{name=D1}] & \mathcal{C} \arrow[r, bend left, "T"{name=U2}] \arrow[r, bend right, "T"'{name=D2}] & \mathcal{D}. \arrow[from=U1, to=D1, Rightarrow, "\mu"] \arrow[from=U2, to=D2, Rightarrow, "1"] \end{tikzcd}$$

Next, consider the arrangement:

$$\begin{tikzcd} \mathcal{B} \arrow[r, "F"] & \mathcal{C} \arrow[r, bend left, "T"{name=U}] \arrow[r, bend right, "T'"'{name=D}] & \mathcal{D}. \arrow[from=U, to=D, Rightarrow, "\nu"] \end{tikzcd}$$

Again, let $b \in O_{\mathcal{B}}$. Then, $Fb \in \mathcal{C}$ and $\nu_{Fb} : TFb \to T'Fb$ is an arrow in $\mathcal{D}$. We let $\nu F : TF \dot{\to} T'F$ denote the natural transformation with components given precisely by the F-image components of ν : $(\nu F)_b = \nu_{Fb}$. This is another example of horizontal composition, where $\nu F = \nu \circ 1_F$:

$$\begin{tikzcd} \mathcal{B} \arrow[r, bend left, "F"{name=U1}] \arrow[r, bend right, "F"'{name=D1}] & \mathcal{C} \arrow[r, bend left, "T"{name=U2}] \arrow[r, bend right, "T'"'{name=D2}] & \mathcal{D}. \arrow[from=U1, to=D1, Rightarrow, "1"] \arrow[from=U2, to=D2, Rightarrow, "\nu"] \end{tikzcd}$$

Thus, while the symmetry of such notations as 'μT' and '$T\mu$' reflects the conceptual symmetry of the horizontal compositions discussed, care must be taken when applying these maps due to the relatively asymmetric nature of the component-wise formulations of the natural transformations.

1.3 Monads in Category Theory

1.3.1 Definition

We now give the first formulation of the concept that is of primary interest to us. In category theory, a monad in a category $\mathcal{C}$ is a monoid in the category of endofunctors of $\mathcal{C}$, with the binary operation being functor composition and the unit element being the identity endofunctor. We can present the definition of a monad diagrammatically using commutative diagrams like those given previously in the diagrammatic definition of a monoid. That is, given functors $I_{\mathcal{C}}, T, T^2, T^3 : \mathcal{C} \to \mathcal{C}$ and natural transformations $\eta : I_{\mathcal{C}} \to T, \mu : T^2 \to T$, the 3-tuple (T, η, μ) is a *monad* in $\mathcal{C}$ if the following diagrams commute:

$$\begin{tikzcd}
T^3 \arrow[r, "\mu T"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
IT \arrow[r, "\eta T"] \arrow[d, equal] & T^2 \arrow[d, "\mu"] & TI \arrow[l, "T\eta"'] \arrow[d, equal] \\
T \arrow[r, equal] & T \arrow[r, equal] & T.
\end{tikzcd}$$

For notational brevity, we will often denote a monad (T, η, μ) by its underlying endofunctor T. In addition, we will refer to the equations corresponding to the left and right diagrams above as monad laws 1 and 2, respectively.

1.3.2 Examples

1. We begin by working through the details of a simple example. Let $(M, \cdot, e)$ be a monoid and $T : \mathbf{Set} \to \mathbf{Set}$ a functor defined by $TX = M \times X$ for any $X \in \mathbf{Set}$. Define the natural transformations $\mu_X : M \times M \times X \to M \times X$ by $\mu_X((m, n, x)) = (m \cdot n, x)$ and $\eta_X : X \to M \times X$ by $\eta_X(x) = (e, x)$. This gives the following situation:

$$\begin{tikzcd}
\mathbf{Set} \arrow[r, bend left, "T^2", ""{name=U, below}] \arrow[r, bend right, "T"', ""{name=D, above}] & \mathbf{Set}
\arrow[from=U, to=D, "\mu"]
\end{tikzcd}$$

$$\begin{array}{ll}
X & \begin{array}{ccc} T(T^2X) & \xrightarrow{T(\mu_X)} & T(TX) \\ (l, m, n, x) & \mapsto & (l, m \cdot n, x) \end{array} \\[2em]
TX & \begin{array}{ccc} T^2(TX) & \xrightarrow{\mu_{TX}} & T(TX) \\ (l, m, n, x) & \mapsto & (l \cdot m, n, x). \end{array}
\end{array}$$

We can thus demonstrate the commutativity of the first defining diagram for a monad by showing that elements are mapped via pertinent paths in the same way.:

$$\begin{tikzcd}
M \times M \times M \times X \arrow[r, "(\mu T)_X"] \arrow[d, "(T\mu)_X"'] & M \times M \times X \arrow[d, "\mu_X"'] \\
M \times M \times X \arrow[r, "\mu_X"] & M \times X
\end{tikzcd}$$

$$\begin{tikzcd}
(l, m, n, x) \arrow[r, "\mu_{TX}"] \arrow[d, "T(\mu_X)"'] & (l \cdot m, n, x) \arrow[d, "\mu_X"'] \\
(l, m \cdot n, x) \arrow[r, "\mu_X"] & (l \cdot m \cdot n, x).
\end{tikzcd}$$

Similarly, for monad law 2 we have the situation:

$$\begin{tikzcd}
\textbf{Set} \arrow[r, shift left=3, "I"] \arrow[r, shift right=3, "T"'] \arrow[r, phantom, "\downarrow \eta"] & \textbf{Set}
\end{tikzcd}$$

$$\begin{array}{lccc}
X & T(IX) & \xrightarrow{T(\eta_X)} & T(TX) \\
 & (m, x) & \mapsto & (m, e, x) \\
 & & & \\
TX & I(TX) & \xrightarrow{\eta_{TX}} & T(TX) \\
 & (m, x) & \mapsto & (e, m, x).
\end{array}$$

These mappings give rise to the following verification of commutativity:

$$\begin{tikzcd}
TX \arrow[r, "(\eta T)_X"] \arrow[d, equal] & T^2X \arrow[d, "\mu_X"'] & TX \arrow[l, "(T\eta)_X"'] \arrow[d, equal] \\
TX \arrow[r, equal] & TX \arrow[r, equal] & TX
\end{tikzcd}$$

$$\begin{tikzcd}
(m, x) \arrow[r, "\eta_{TX}"] \arrow[rd, phantom, "\diagdown\diagdown"] & (e, m, x) \arrow[d, "\mu_X"] & (m, e, x) \arrow[d, "\mu_X"'] & (m, x) \arrow[l, "T(\eta_X)"'] \arrow[ld, phantom, "//"] \\
 & (e \cdot m, x) \arrow[r, phantom, "="] & (m \cdot e, x). &
\end{tikzcd}$$

Therefore, (T, η, μ) is a monad.

2. Let $(P, \leq)$ be a partially ordered set. If we consider the elements of P to be objects, and the relationship $x \leq y$ where $x, y \in P$ to be equivalent to an arrow from x to y, then P can be viewed as a category. By the definition of a poset and that of arrows in the category P, there can be at most one arrow between any two objects. Thus, any endofunctor defined on P is a monotonic function. Let T be such a functor and observe that the conditions describing the natural transformations η and μ for a monad are equivalent to the conditions $x \leq Tx$ and $T(T(x)) \leq T(x), \forall x \in P$. By applying T to the first of these inequalities, we have $T(x) \leq T(T(x))$. Combined with the second inequality and the definition of a poset, we have $T(x) = T(T(x))$; that is, a monad on a poset P is a closure operation.

3. Recall the definitions of the free group functor F : **Set** $\to$ **Grp** and the forgetful functor U : **Grp** $\to$ **Set**. By composing these functors, we have an endofunctor $T = U \circ F$: **Set** $\to$ **Set**, which maps a set X to the underlying set of Free(X). We define a *word* in X to be a string of finite length of elements from $X \cup X^{-1}$, where $X^{-1} = \{x^{-1} | x \in X\}$ is a set of formal inverses of elements in X. Then TX is the set of equivalence classes of words made up of symbols x and x^{-1} for all $x \in X$—equivalent under deletion of string segments 'xx^{-1}' and '$x^{-1}x$,' for any $x \in X$. If w is a word in X, we denote its equivalence class by $[w]$ and typically refer to the entire "equivalence class of the word w" as simply the "word w."

   If we define η_X by sending $x \mapsto [x]$ and μ_X by the concatenation of words in TX, then T is a monad. Note that a word in T^2X is composed of words from TX, and concatenating words amounts to removing brackets, with our notation. For example, if $[z^{-1}yx], [x^{-1}y] \in TX$, then $[[z^{-1}yx][x^{-1}y]] \in T^2X$ and $\mu_X([[z^{-1}yx][x^{-1}y]]) = [z^{-1}yxx^{-1}y] = [z^{-1}yy]$.

   We provide illustrations of the commutativity of the monad diagrams. For monad law 1:

$$\begin{tikzcd}
{[[[xy][zxx][y]][[x^{-1}][y]]]} \arrow[r, "\mu_{TX}"] \arrow[d, "T(\mu_X)"'] & {[[xy][zxx][y][x^{-1}][y]]} \arrow[d, "\mu_X"'] \\
{[[xyzxxy][x^{-1}y]]} \arrow[r, "\mu_X"] & {[xyzxxyx^{-1}y].}
\end{tikzcd}$$

Notice that the commutativity of this diagram can be summarized by stating that the removal of inner brackets can be done in either order. For monad law 2:

$$\begin{tikzcd}
TX \arrow[r, "(\eta T)_X"] \arrow[d, equal] & T^2X \arrow[d, "\mu_X"'] & TX \arrow[l, "(T\eta)_X"'] \arrow[d, equal] \\
TX \arrow[r, equal] & TX \arrow[r, equal] & TX
\end{tikzcd}$$

$$\begin{tikzcd}
{[xyz^{-1}]} \arrow[r, "\eta_{TX}"] \arrow[dr, equal] & {[[xyz^{-1}]]} \arrow[d, "\mu_X"] & {[[xyz^{-1}]]} \arrow[d, "\mu_X"'] & {[xyz^{-1}]} \arrow[l, "T(\eta_X)"'] \arrow[dl, equal] \\
& {[xyz^{-1}]} \arrow[r, phantom, "="] & {[xyz^{-1}]}.
\end{tikzcd}$$

While the above diagram appears rather unenlightening, it is interesting to note that the mapping along the upper left arrow adds the outer set of brackets, while mapping via the upper right arrow adds the inner set.

4. The power set monad is described by the power set functor along with natural transformations η and μ, defined as follows. For any $x \in X, \eta_X(x) = \{x\}$. Let L be a set whose elements are subsets of X. Define μ_X to be the map which takes L to the union of the elements of L. Note that $\mu_X(L) \subseteq X$, as is necessary. This example looks similar to that preceding, but there are key differences which we illustrate in the example below. Specifically, μ's definition involves taking the union of subsets; therefore, elements that are members of more than one subset will show up as only a single element in the union. In the case of string concatenation, any 'repeated' element is not condensed into a single copy of that element, but cancellations can occur in accord with the given equivalence relation of word reduction. Further, under the operation of union, the order of elements in the resultant set is unimportant; by contrast, the order is important in string concatenation. Compare the diagrams below to those of the preceding example.

$$\begin{tikzcd}
\{\{\{x,y\}\{z,x\}\{y\}\}\{\{x^{-1}\}\{y\}\}\} \arrow[r, "\mu_{TX}"] \arrow[d, "T(\mu_X)"'] & \{\{x,y\}\{z,x\}\{y\}\{x^{-1}\}\} \arrow[d, "\mu_X"'] \\
\{\{x,y,z\}\{x^{-1},y\}\} \arrow[r, "\mu_X"] & \{x,x^{-1},y,z\}
\end{tikzcd}$$

$$\begin{tikzcd}
\{x,y,z^{-1}\} \arrow[r, "\eta_{TX}"] \arrow[dr, equal] & \{\{x,y,z^{-1}\}\} \arrow[d, "\mu_X"] & \{\{x,y,z^{-1}\}\} \arrow[d, "\mu_X"'] & \{x,y,z^{-1}\} \arrow[l, "T(\eta_X)"'] \arrow[dl, equal] \\
 & \{x,y,z^{-1}\} \arrow[r, equal] & \{x,y,z^{-1}\}. &
\end{tikzcd}$$

1.4 Category Theory in Haskell

The main focus of our study is the application of category theory to the programming language Haskell. Haskell is a polymorphically typed, purely functional language. In the appropriate contexts, we will define each of these terms. We begin with a brief discussion of the fundamental connection between category theory and the functional programing paradigm.

1.4.1 Functional Languages and the λ-Calculus

Traditional approaches to the design and implementation of programming languages have been centred around the idea of a *variable*: a modifiable association between a name and values [9]. Such languages are called *imperative* because programs based on this concept are comprised of sequences of commands. Each command usually consists of the evaluating of an expression and assigning the return value to a variable's name. Moreover, a command found at any step of the computation sequence can be dependent on variables used—and potentially modified—in preceding commands. In this way, values can be passed down through a command sequence, and the value assigned to a name in one command can be replaced by a new value in a subsequent command. These ideas imply that the order of command execution typically affects a computation result.

In contrast, the *functional* programming paradigm is founded on structured function calls, a function call being a function name followed by a list of arguments in parentheses. Programs in functional languages consist of nested function calls or, in other words, are compositions of functions.

As such, function names behave as they do in the context of mathematical functions: they are formal parameters of other functions and are associated with values through calls to functions that have been given actual values. Once an association between a function name and a value has been made, it cannot be modified. In summary, a functional program is a single expression and executing such a program corresponds to evaluating the expression.

This mathematical approach offers the following advantages to programming with functional languages [14]:

- Declarativeness — The order of function execution need not be specified by the programmer but rather can be inferred by the system based on relations that must be satisfied by evaluated functions.
- Referential Transparency — If two functions `f` and `g` produce the same values when given the same arguments, then any occurence of `f` can be replaced by `g` without changing the meaning of the program.
- Higher-Order Functions — Functions can accept functions as arguments and can output new functions.
- Polymorphism — Individual algorithms can be written in such a way that they apply to several types of input (in certain cases, this is also referred to as function overloading). This is somewhat akin to saying that we can define the "same function" on different domains: an idea which makes mathematical sense in the context of a function being defined on a set and certain supersets. The most obvious candidate for a polymorphic function is the identity function. We can define the identity function for *any* input `a` by returning `a` unchanged. It is often more convenient to think of a single, "umbrella" identity function than to define a new identity function for each type of input being considered. Thus, in Haskell for example, the identity function `id` is a polymorphic function that accepts an input of any type and returns a copy of this input.

Many approaches to the semantics of polymorphic type systems are founded on the *typed λ-calculi*; in particular, Haskell is based on λ-calculus. The two core concepts of the λ-calculus are those of function abstraction and application. The former refers to the generalization of expressions via the introduction of names, and the latter refers to the assignment of particular values to names as a means to evaluate generalized expressions [14]. In this work, we will not provide a comprehensive introduction to or detailed explanations of the λ-calculus. We will, however, include some references to the λ-calculus on occassion, with aims to benefit those already familiar and entice unfamiliar readers to learn more independently.

Formally, a typed λ-calculus is an axiomatic system consisting of terms, types, variables and equations. For every term a in a typed λ-calculus, there is a corresponding primitive type A called the type of a, with the correspondence denoted by $a \in A$. Since Haskell's type system will be of recurring importance throughout our study, we cite the λ-calculus axioms pertaining to term- and type-formation here as general background information [2] (for the 11 other rules necessary to define a typed λ-calculus, also see [2]):

- There is a unit type **1**.
- There is a term $*$ of type **1**.
- If A and B are types, then there exists a product type $A \times B$ and a functional type $[A \rightarrow B]$, where $[A \rightarrow B]$ denotes the set of functions from A to B.
- If a and b are terms of type A and B, respectively, then there is a term (a, b) of type $A \times B$.
- If c is a term of type $A \times B$, there are terms $\mathrm{proj}_1(c)$ and $\mathrm{proj}_2(c)$ of type A and B, respectively.
- If a is a term of type A and f is a term of type $[A \rightarrow B]$, then there is a term $f\,a$ of type B.
- For each type A, there is a countable set of terms x_i^A of type A called variables of type A.
- If x is a variable of type A and $\phi(x)$ is a term of type B, then $\lambda_{x \in A}\phi(x) \in [A \rightarrow B]$. Here, $\phi(x)$ denotes a term ϕ that might contain the variable x.

In addition, we define a *free variable* to have the following properties: If x is a variable, then x is free in the term x. If an occurrence of x is free in either of the terms a or b, then that occurrence of x is free in (a, b). If x occurs freely in either f or a, then that occurrence of x is free in $f\,a$. Finally, there exists at least one occurrence of x that is not free—called bound—in $\lambda_x\phi(x)$. Note that the last item in the above list offers a notation for defining functions in the λ-calculus. Another notation that is commonly found in the literature is $\lambda x : \phi(x)$, which would be equivalent to $\lambda_x\phi(x)$ in the above notation.

It can be shown that it is possible to define every computable function using function abstraction and application of the λ-calculus. This means that the λ-calculus is a theoretical, universal machine code for programming languages. It is further suited for describing programming languages in that its purely mathematical nature facilitates constructing proofs about the language being described. Lastly, the simplicity of the λ-calculus makes its implementation in languages relatively easy [9].

1.4.2 Cartesian Closed Categories

The λ-calculus provides the first large link between functional programming languages and category theory. The connection is in the correspondence between typed λ-calculi and cartesian closed categories.

A category $\mathcal{C}$ is a *cartesian closed category* if it satisfies the following axioms [2]:

- There is a terminal object $\mathbf{1}$.
- For every pair of $\mathcal{C}$-objects A and B, there exists a $\mathcal{C}$-object $A \times B$, the product of A and B, with projections $p_1 : A \times B \to A$ and $p_2 : A \times B \to B$.
- For every pair of objects A and B, there exists an object $[A \to B]$ and an arrow $\text{eval} : [A \to B] \times A \to B$ with the property that for any arrow $f : C \times A \to B$, there is a unique arrow $\lambda f : C \to [A \to B]$ such that the composite arrow

  $$C \times A \xrightarrow{\lambda f \times A} [A \to B] \times A \xrightarrow{\text{eval}} B$$

  is equal to f.

With respect to the third property, we note that the exponential $[A \to -]$ is a right adjoint to $- \times A : \mathcal{C} \to \mathcal{C}$ and that eval is the counit of this adjunction. Note also that the composition of a functor F with its right adjoint U determines a monad (T, η, μ) with $T = UF$.

A λ-theory is a typed λ-calculus that is extended with some additional terms, types, or equivalences between expressions [13]. It can be shown that the concepts of typed λ-calculus and cartesian closed category are equivalent in the following sense: (i) Given a λ-theory L, we can construct a cartesian closed category in which each object corresponds to a type in L and in which an arrow between objects A and B is an equivalence class of terms of type B with one free variable of type A; (ii) given a cartesian closed category $\mathcal{C}$ that is assumed to have finite products, we can define a typed λ-calculus which has as types the objects of $\mathcal{C}$. In particular, the types required by the first and second typed λ-calculus axioms given above are the objects $\mathbf{1}$, $A \times B$ and $[A \to B]$.

Note that the equivalence proof shows the connection between the notation used for functions in the λ-calculus and the choice of the function name

λf in the last listed item. In fact, f and λf are simply corresponding maps under the adjunction $(- \times A, -) \cong (-, [A \to -]) : \mathcal{C} \times \mathcal{C} \to \mathbf{Set}$. We refer the reader to, for example, [2] for a version of the equivalence proof.

1.4.3 Types in Haskell

Any programming language has a means by which to group similar kinds of values and expressions. A syntactic method for organizing and implementing levels of abstraction in programs is called a *type system*, with the constrained data values referred to as data types or *types*. Typical examples of types in programming languages include primitive types (such as integers, floating point numbers and characters), tuples, lists, function types and abstract data types.

Haskell is a statically typed programming language. This means that every expression in Haskell is assigned a type, and type checking is completed as part of the compile-time, rather than run-time, process. As a result, a compile-time error will be generated if any function in the program is given an argument of the wrong type. This is helpful in locating and reducing the number of programming errors (bugs) that can occur when programming.

The type signature of a function is a specification of its argument and return types. For example, the function `toUpper` is a Haskell function that takes a character as input and returns the corresponding upper case character as output. We denote the type signature of this function with the (Prelude) notation `toUpper :: Char -> Char`. The type signature of a function `f` of n variables is denoted by `f ::` $\mathtt{t}_1$ `->` $\mathtt{t}_2$ `->` $\cdots$ `->` $\mathtt{t}_n$ `-> r`, where $\mathtt{t}_1, \ldots \mathtt{t}_n$ are the (possibly distinct) input types and `r` is the return type. Equivalently, we can think of `f` as a function that accepts k arguments $\mathtt{t}_1, \ldots \mathtt{t}_k$ and returns a function of $n-k$ arguments $\mathtt{t}_{k+1}, \ldots \mathtt{t}_n$, with the output type of the return function of course being `r`.

A functional programming language can be construed as a category if we view the types of the language to be the objects and the functions defined between types to be the morphisms. As a simple example, consider a functional language with primitive types

`Bool`	having one of the values true or false
`Char`	characters
`Float`	floating-point numbers: real numbers with no fixed number of digits before or after their decimal points
`Int`	(fixed-precision) integers,

and primitive functions

`floor` :: `Float` ⟶ `Int`	greatest integer less than or equal to input
`id` :: ∗ ⟶ ∗	identity (returns an identical copy of input)
$\texttt{inc}_{\texttt{Int}}$ :: `Int` ⟶ `Int`	increment an integer by 1
$\texttt{inc}_{\texttt{Float}}$ :: `Float` ⟶ `Float`	increment a floating-point number by 1
`iszero` :: `Int` ⟶ `Bool`	test for zero
`sqr` :: $\substack{\texttt{Float}\\\texttt{Int}} \longrightarrow \substack{\texttt{Float}\\\texttt{Int}}$	square a number
`toFloat` :: `Int` ⟶ `Float`	convert an integer to a floating-point number
`toUpper` :: `Char` ⟶ `Char`	convert a lower-case character to upper-case.

Clearly, the polymorphic function `id` will correspond to the identity morphism for any object in the category. Excluding identity and composition arrows, the category corresponding to the above data would look like:

$$\begin{tikzcd}
 & \texttt{Int} \arrow[loop left, "\texttt{inc}_{\texttt{Int}}\texttt{, sqr}"] \arrow[dd, shift left, "\texttt{toFloat}"] \arrow[dr, "\texttt{iszero}"] & \\
\texttt{Char} \arrow[loop above, "\texttt{toUpper}"'] & & \texttt{Bool} \\
 & \texttt{Float} \arrow[loop left, "\texttt{inc}_{\texttt{Float}}\texttt{, sqr}"] \arrow[uu, shift left, "\texttt{floor}"] &
\end{tikzcd}$$

We can neatly display the commutativity of certain operations in a language using commutative diagrams. For example,

$$\begin{tikzcd}
\texttt{Int} \arrow[r, "\texttt{inc}_{\texttt{Int}}"] \arrow[d, "\texttt{toFloat}"'] & \texttt{Int} \arrow[d, "\texttt{toFloat}"] \\
\texttt{Float} \arrow[r, "\texttt{inc}_{\texttt{Float}}"'] & \texttt{Float}
\end{tikzcd}$$

summarizes the equivalence between incrementing an integer and then converting it to a floating-point number with converting an integer to a floating-point number and incrementing the result.

Overloaded Functions and Implicit Conversions

There is an interesting connection between the transformations whose commutativity is considered semantically valid in a given language and the use of overloading in that language. *Overloading* is a case of type polymorphism in which an operator has a different implementation depending on the type of its arguments. The main purpose of overloading is to increase the intuitiveness or convenience of a language's syntax. For example, it would be disheartening if programmers had to distinguish between two operations ($+_{\mathtt{Int}}$) and ($+_{\mathtt{Float}}$) when trying to add two "numbers." The same idea is true for the `sqr` function shown in the category above (note the overloading implemented there). Yet, the action that a compiler should take when faced with an expression such as `let x in Float = i + j`, where `i` and `j` are integers is not obvious, the two possibilities being $\mathtt{x} = \mathtt{toFloat(i)}+_{\mathtt{Float}}\mathtt{toFloat(j)}$ and $\mathtt{x} = \mathtt{toFloat(i}+_{\mathtt{Int}}\mathtt{j)}$. Of course, from a mathematical standpoint these options are equivalent, which suggests that perhaps a compiler should be indifferent to the interpretation of these expressions as well. This has motivated approaches to formalizing proper designs of overloaded functions and implicit conversions, so that their commutivity upon insertion into a compiler is semantically irrelevant.

In the case of `inc`, `sqr` or `(+)` discussed above, the 'semantic commutivity' that arises does so because $\mathbb{Z} \subset \mathbb{R}$. In our programming context, this means that data of type `Int` can be implicitly converted to that of type `Float`. In many cases, an implicit conversion function corresponds to the inclusion function between appropriate types. This mathematical analogy does not nicely extend to all cases of implicit conversions, however. For example, it may be useful to consider integer expressions as a type; that is, a mathematical expression in which all terms are integers (for example, `x + y`, where `x` and `y` are of type **Int**). Here, the integer variables (for example, `x`, where `x` is of type **Int**) can be viewed as a subtype of the integer expressions, when viewing an integer variable as a subtype of integer expressions. The difficulty in defining a "set of integer expressions" prohibits a natural extension of the inclusion analogy for these types.

To generalize the notion, we briefly introduce Reynold's approach of considering an alternate construction of a category representative of a functional language [15]. We again let the objects of the category be the types of the language, but we use them to form a partial order in which the arrows between types correspond to valid implicit conversions. As an example, we use the same objects as in the type category figure above, along with a 'universal' object Value. This object is a contrived maximum for the type partial order, it being defined as a supertype of all other types in the category. With this, the partial order Ω would look like:

$$\begin{tikzcd}
& \text{Value} & \\
\texttt{Char} \arrow[ur] & \texttt{Float} \arrow[u] & \texttt{Bool} \arrow[ul] \\
& \texttt{Int} \arrow[u] &
\end{tikzcd}$$

We define a *carrier* $C : \Omega \to \mathbf{Set}$ to be a functor that maps a type $t \in \Omega$ to the set of values of type t. The morphisms $t \leq t'$ in Ω are mapped to implicit conversion functions between sets of types, whose behaviour is restricted by the defining properties of the functor C. Explicitly, the definition of a functor $C : \Omega \to \mathbf{Set}$ on a partial order Ω is a map that satisfies:

(a) $C(t \leq t') \in \mathrm{Hom}(C(t), C(t'))$
(b) $C(t \leq t) = I_{C(t)}$
(c) If $t \leq t'$ and $t' \leq t''$, then $C(t \leq t') = C(t' \leq t'') \circ C(t \leq t')$.

Let $C^2 : \Omega^2 \to \mathbf{Set}^2$ denote the functor defined on types t_1, t_2 by

$$C^2(t_1, t_2) = (C(t_1), C(t_2))$$

and on arrows $t_1 \leq t_1', t_2 \leq t_2'$ by

$$C^2(t_1 \leq t_1', t_2 \leq t_2') = (C(t_1 \leq t_1'), C(t_2 \leq t_2')).$$

Let $x^2 : \mathbf{Set}^2 \to \mathbf{Set}$ denote the functor defined on sets S_1, S_2 by

$$x^2(S_1, S_2) = S_1 \times S_2$$

and on set functions f_1, f_2 by

$$x^2(f_1, f_2) = f_1 \times f_2.$$

In the context of implicit conversions, we are interested in converting a type t_1 into another type t_2, where $t_1 \leq t_2$. For example, we might consider the conversion from integers to real numbers or from an integer to an integer expression (for the latter, consider converting the integer 1 to the expression '1' that evaluates to the integer 1). Therefore, from a categorical standpoint, we always assume that the partial order Ω has a largest (catchall) element as described above–call it Value–that semantically encompasses all other types. This is the terminal object of the partial order.

We thus let $\Gamma_\delta : \Omega^2 \to \Omega$ denote the functor defined on any two types t_1, t_2 by

$$\Gamma_\delta(t_1, t_2) = \max\{t_1, t_2\}$$

and on arrows $t_1 \leq t_1', t_2 \leq t_2'$ by

$$\Gamma_\delta(t_1 \leq t_1', t_2 \leq t_2') = (\Gamma_\delta(t_1, t_2) \leq \Gamma_\delta(t_1', t_2')).$$

The compiler-recognized commutivity of overloaded functions and implicit conversions can then be expressed by stating that for any types t_1, t_2 with $t_1 \leq t_2$ an overloaded function γ_δ is a natural transformation between the composite functors $\Omega^2 \xrightarrow{C^2} \mathbf{Set}^2 \xrightarrow{x^2} \mathbf{Set}$ and $\Omega^2 \xrightarrow{\Gamma_\delta} \Omega \xrightarrow{C} \mathbf{Set}$:

$$\begin{tikzcd}
x^2 \circ C^2(t_1, t_2) \arrow[r, "\gamma_\delta(t_1,t_2)"] \arrow[d, "x^2 \circ C^2(t_1 \leq t_1', t_2 \leq t_2')"'] & C \circ \Gamma_\delta(t_1, t_2) \arrow[d, "C \circ \Gamma_\delta(t_1 \leq t_1', t_2 \leq t_2')"] \\
x^2 \circ C^2(t_1', t_2') \arrow[r, "\gamma_\delta(t_1',t_2')"'] & C \circ \Gamma_\delta(t_1', t_2')
\end{tikzcd}$$

The preceding discussion regarding the overloaded function (+)—or, in our new notation, γ_+—can now be understood as an instance of the above diagram. If we let $t_1 = t_2 =$ Int, then $C^2(t_1, t_2) = (C(t_1), C(t_2)) = (C(\texttt{Int}), C(\texttt{Int}))$ and $x^2 \circ (C(\texttt{Int}), C(\texttt{Int})) = C(\texttt{Int}) \times C(\texttt{Int})$. Further, in the partial order category there is an identity arrow Int $\leq$ Int, which corresponds intuitively to the idea that any value of type Int can be implicitly converted (trivially) to a value of type Int. Thus, $\Gamma_\delta(\texttt{Int}, \texttt{Int}) = \texttt{Int}$, and $C \circ \Gamma_\delta(\texttt{Int}, \texttt{Int}) = C(\texttt{Int})$. Similar derivations are achieved by setting $t_1' = t_2' =$ Float, and defining the conversion function $C(\texttt{Int} \leq \texttt{Float})$ to be toFloat gives rise to the following diagram:

$$\begin{tikzcd}
C(\texttt{Int}) \times C(\texttt{Int}) \arrow[r, "\gamma_+(\texttt{Int},\texttt{Int})"] \arrow[d, "\texttt{toFloat} \times \texttt{toFloat}"'] & C(\texttt{Int}) \arrow[d, "\texttt{toFloat}"] \\
C(\texttt{Float}) \times C(\texttt{Float}) \arrow[r, "\gamma_+(\texttt{Float},\texttt{Float})"'] & C(\texttt{Float}).
\end{tikzcd}$$

1.5 Haskell

1.5.1 Haskell's Type System

Type Classes

As previously mentioned, an overloaded function is a function that can accept parameters of more than one type. This is somewhat problematic for a static type checker, since there can be ambiguity as to which types a given operation is defined for. Consider first the identity function. This function can accept arguments of all types. In Haskell, its type signature is `id :: a -> a`, where `a` is a *type variable* that can stand for any defined type. As such, ambiguity is avoided and compiler complications need not arise. However, consider the addition function. While it is desirable for the function `(+)` to be defined for types such as `Int` or `Float`, it should not be defined for, say, `Bool` or `Char`. Thus, a type signature such as `(+) :: a -> a -> a` is too general for the overloaded operation we have in mind.

The system used in Haskell to address this issue is that of type classes. A *type class* is a set of function (or constant) names, together with their respective types, which must be given definitions for every type that is to belong to the class. In other words, it is a set of constraints for a type variable. Specific definitions of the function names of a type class are called *methods*. Once each of the methods corresponding to the function names of a type class has been given, an *instance* of the class has been defined; that is, the type of an instance `instance` of type class `Class` is `instance :: Class`.

For example, the class **Eq** is intended to be a class for types for which it makes sense to test for equality or inequality. The class declaration is given by

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

The types `Char`, `Double`, `Float`, `Int` and `Integer` are instances of **Eq**, where each of them provides definitions for the methods `(=)` and `(/=)`. Now, the domain or range types of a function can be constrained to a type class that suits the desired boundedness of the polymorphism. For example, the type of `(+)` in Haskell is correctly written as `(+) :: Num a => a -> a -> a`, where **Num** is a class with instances `Double`, `Float`, `Int` and `Integer`, and `=>` denotes the restriction of the type variable `a` to the **Num** class.

The Functor Type Class

Let [x] denote a list of elements of type x. In the Prelude, a function map is defined with signature `map :: (x -> y) -> [x] -> [y]`. Its action is to accept a function and a list, and output the list of image elements under the given function; that is `map f [x1, x2,...] = [f(x1), f(x2),...]`. For example, `map negate [1,-2,3] = [-1,2,-3]`. The map function, then, is a means to transform a function that normally acts on elements of type a into a function that acts on lists of elements of type a without changing the essence of the function's behaviour. Of course, this idea alludes to an implementation of the category-theoretic concept of a functor.

As anticipated, we define **Hask** to be a category with Haskell types as objects and Haskell functions as morphisms. Note that the (associative) operation of function composition in Haskell is denoted . and that Haskell has a parametric polymorphic identity function. We can conceive an endofunctor on **Hask** by considering `([]) :: a -> [a]` to be the "object part" of the functor and map to be the "arrow part" of the functor.

The idea is generalized to fmap in Haskell's built-in **Functor** class. It has the following declaration:

```
class Functor (f ::  * -> *) where
  fmap ::  (a -> b) -> (f a -> f b).
```

That is, to define an instance of **Functor**, a method fmap must be defined that accepts any function `g :: a -> b` and returns a function fg with type signature `fg :: f a -> f b`. Equivalently, we can view fmap as a function that accepts a function `g :: a ->b` and a functor element `f a`, and returns another functor element `f b`. The name of the **Functor** class is motivated by the analogy between the definition of the class and the definition of a functor in category theory. Any instance of **Functor** is comprised of a function f that is defined on Haskell types (objects in **Hask**), and a means to map an arrow `g :: a -> b` in **Hask** to an arrow `fg :: f a -> f b` that is also in **Hask**. Typically, category theorists would label the method fmap defined on arrows as f, the same name as the part of the functor that acts on objects. In Haskell, however, f and fmap taken together are what correspond to a category-theoretic functor, where f is the part of the functor that acts on objects and fmap is the part that acts on morphisms.

Of course, the definition of any functor f in category theory includes two equations that f must satsify. For an instance of the **Functor** class to behave predictably and consistently in a Haskell program, analogous laws must hold. We demonstrate the consistency of ideas by giving the following functor laws, which must hold for any instance of **Functor**. Recall that `id` is a polymorphic function and so can act on objects both of type `a` and of type `f a` (where, analogous to lists, `f a` denotes a datatype in which the type of the contained data is `a`). Then, for `f` and `fmap` as given above, as well as functions `g,h :: * -> *`, the Haskell functor laws are given as:

(F1) `fmap id f a` = `f a`
(F2) `fmap g . h f a` = `(fmap g . fmap h) f a`.

As a demonstration of these laws, recall the list example. This is really just an instance of the **Functor** class with `[]` being the '`f :: * -> *`' from the class declaration, and `map` fulfilling the definition for the method `fmap`. In this case,

- `map id [a];` = `[id a]` = `[a]` = `id [a]`
- `map g . h [a]` = `[(g . h) a]` = `map g [h a]` = `map g (map h [a])` = `(map g . map h) [a]`.

Intuitively, we can think of `fmap` as a function that applies a given operation to objects inside a "container" and then returns a container of the same shape. The shape of the container is left unchanged by `fmap` (F1) and the contents are not shuffled relative to one another under the mapping operation (F2). That is, `fmap` preserves the structure of the category **Hask**.

Chapter 2

Moggi's Monads

2.1 Motivation for Monads in Haskell

Haskell is a *purely functional* language. This means that whenever a function is executed, no effects other than the computation of the function output will occur. This is a product of the fact that Haskell abides by the principle of referential transparency (see Section 1.4.1), and that all data flow is made explicit. Since a program written in a functional language consists of a set of equations, the latter guarantees that the value of a function is only dependent on the function's free variables (Section 1.4.1). It also ensures that the order in which computations are made does not affect a program's meaning [14].

This is not the general case for impure functional or imperative languages, and any state changes unrelated to the output of a function are called *side-effects.* Printing to the screen, reading a file, or updating a datum assignment to a variable are examples of side-effects. Of course, it is hard to imagine doing much useful programming without such capabilities! Haskell employs monads to achieve the usefulness and convenience of impure effects without compromising the purity of the language.

While these facts motivate our study, the introduction of monads to computer science actually began with Eugenio Moggi's work in formalizing programming language semantics using category-theoretic concepts, which he called a "categorical semantics of computations" [12]. Moggi's work provides the foundational theory for monads in Haskell, making a detailed examination of it both interesting and important in our context. His approach began with the adoption of the view taken by the denotational theory of semantics, in which programs are thought of as morphisms in a category with the types of a given language as objects.

2.1.1 Heuristics

A classical approach to computational semantics is to view a program as a function that maps an input value to an output value. Heuristically, a program (or a program in an environment) f could be described as a composition of functions $f = f_{n-1} \circ \cdots \circ f_1$, where $f_1 : A_1 \to A_2, \ldots, f_{n-1} : A_{n-1} \to A_n$ and $\forall i \in \{1 \ldots n\}$ the A_i are types of a given language. Thus, the program would accept an input value $a_1 \in A_1$ and output a value $a_n \in A_n$.

Moggi's twist on the classic model was to consider programs not as maps from values to values, but rather as maps from values to *computations.* Here, a computation is to be thought of as the denotation or meaning of a program and the term "notion of computation" will refer to a qualitative description of the denotation of a program. In accord with denotational semantics, we can rephrase this idea as: a computation is to be thought of as the meaning of an arrow in a category of types. If T is a given notion of computation, then TA will denote a computation with return type A, and will be referred to as a computation of type A.

Bearing in mind that the goal is to construct a category whose morphisms are programs, we now have the choice to model programs as maps of the form $f : A_1 \to TA_n$ or $g : TA_1 \to TA_n$. We choose the former because it subsumes the latter: any computation of type A_1 is a value of type TA_1. In summary, the categorical construction used by Moggi is as follows [11]:

- We consider a category $\mathcal{C}$ with an operation T that carries a $\mathcal{C}$-object A, representative of the set of values of type A, to TA, representative of the set of computations of type A. T can be thought of as a unary type-constructor that is some notion of computation, with the elements of TA being viewed as the denotations of programs of type A.
- A program that takes a value of type A as input, performs a computation, and returns a value of type B, can then be identified with a $\mathcal{C}$-arrow from A to TB.
- A minimum set of requirements for values and computations is found so that programs indeed correspond to morphisms of a suitable category.

But what sorts of programming concepts can notions of computations capture? We can think of them as additional effects that can occur during the evaluation of a function. Computations with exceptions and computations with state modifications are both examples of notions of computations. In the former, the denotation of a program (that is, its semantic interpretation) is either a value or an exception, depending on whether the execution of the program involves the ordinary calculation of a value or the consideration of a user-defined special case (for example, a division by zero), respectively; in the latter, a program denotes a map from a store to a pair comprised of a value and a modified store. These notions of computations are precisely the kinds of effects that were previously particular to imperative languages and thus lay the conceptual foundations for implementation of such features in specific pure languages, such as Haskell.

Now recall that, in general, we consider a program to be a function that is the composition of several functions, and consider the task of modifying such a program so that at some stage of calculation (that is, at some level of composition), a notion of computation is introduced. For example, let us initially assume the classic model, with a program given by

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D.$$

Say that we want to introduce a side-effecting computation T to the function g, but otherwise leave the program unchanged. We thus want to alter the output type of g from C to TC and the output type of h from D to TD (the latter so that the side-effect is carried through to the end of the program's evaluation). This alone cannot be done, however, because while we can easily envision replacing g by a function $\tilde{g} : B \rightarrow TC$, the domain of h is C and not TC.

$$A \xrightarrow{f} B \xrightarrow{\tilde{g}} TC \xrightarrow{?} TD.$$

We thus need a means to (i) compute the desired side-effect; (ii) alter the function h so that it can be composed with $\tilde{g}$; (iii) leave the behaviour of the program otherwise unchanged.

These guidelines lead to the category-theoretic concepts of Kleisli triples and Kleisli categories for use in programming semantics.

2.2 Kleisli constructions

A *Kleisli triple* over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where $T : \mathcal{O}_\mathcal{C} \rightarrow \mathcal{O}_\mathcal{C}$, $\eta_A : A \rightarrow TA$ and $f^* : TA \rightarrow TB$, for $A, B, C \in \mathcal{O}_\mathcal{C}$, and $f : A \rightarrow TB$, and the following equations hold:

(KT1) $\eta_A^* = \mathrm{id}_{TA}$

(KT2) $f^* \circ \eta_A = f$

(KT3) $g^* \circ f^* = (g^* \circ f)^*$ with $g : B \rightarrow TC \in \mathcal{A}_\mathcal{C}$.

A Kleisli triple models a notion of computation and the above properties can be intuitively understood in the following manner. The mapping η_A is the inclusion mapping of values into computations (any value can be thought of as a "constant" computation) and f^* is the extension of a function f from values to computations to one from computations to computations. Intuitively, this extension can be thought of as first evaluating a computation and then applying f to the return value. The formal function definition for f^* can be given in terms of the **let**-constructor, which is introduced in Section 2.4.

Any Kleisli triple gives rise to a corresponding Kleisli category: given a Kleisli triple $(T, \eta, \_^*)$ over $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_T$ is defined in terms of objects and arrows in $\mathcal{C}$. Since we are constructing $\mathcal{C}_T$ from $\mathcal{C}$, we may use the same label to refer to an object or arrow in $\mathcal{C}_T$ as one in $\mathcal{C}$. In the case of possible ambiguity, the category of context will be made explicit. The construction of the Kleisli category is defined in terms of $\mathcal{C}$-objects A, B and $\mathcal{C}$-arrows $f : A \to TB, g : B \to TC$ is as follows:

- $\mathcal{O}_{\mathcal{C}_T} = \mathcal{O}_{\mathcal{C}}$
- $\text{Hom}_{\mathcal{C}_T}(A, B) = \text{Hom}_{\mathcal{C}}(A, TB)$, where $\forall A, B \in \mathcal{C}$ an arrow $A \to B \in \mathcal{C}_T$ is an arrow $A \to TB$.
- $\forall A \in \mathcal{O}_{\mathcal{C}_T}, \exists\ \text{id}_A \in \mathcal{A}_{\mathcal{C}_T}$ such that $\text{id}_A = \eta_A$, where $\eta_A : A \to TA \in \mathcal{A}_{\mathcal{C}}$.
- The composition $g \circ_T f$ of $\mathcal{C}_T$ arrows is given by the composition $g^* \circ f : A \to TC$ in $\mathcal{C}$.

With regard to these axioms, note that the defined composition $g^* \circ f$ can be thought of as: (i) taking a value $a \in A$; (ii) applying f to obtain a computation fa; (iii) executing or evaluating fa to obtain a value $b \in B$; (iv) applying g to b to produce the resultant computation.

We can use the construction axioms for $\mathcal{C}_T$ to express the left unit, right unit, and associativity axioms for $\mathcal{C}_T$ solely in terms of objects and arrows from $\mathcal{C}$ as follows. Let A, B, C be objects in $\mathcal{C}_T$. For the left unit axiom corresponding to id_B, let $f : A \to B$ be an arrow in $\mathcal{C}_T$, so that $\text{id}_B \circ_T f = f \in \mathcal{C}_T$. We can express the left-hand side of the preceding equation in terms of $\mathcal{C}$-objects and $\mathcal{C}$-arrows using the third and fourth construction axioms listed above: $\text{id}_B \circ_T f = \eta_A^* \circ f$. Similarly, for the right unit axiom corresponding to id_B, let $g : B \to C \in \mathcal{C}_T$, so that $g \circ_T \text{id}_B = g \in \mathcal{C}_T$, and moving from $\mathcal{C}_T$ to $\mathcal{C}$, we have: $g \circ_T \text{id}_B = g^* \circ \eta_B$.

Finally, the associativity axiom for $\mathcal{C}_T$ is easily expressed in terms of $\mathcal{C}$-objects and $\mathcal{C}$-arrows using the fourth construction axiom. Let $f : A \to B, g : B \to C, h : C \to D \in \mathcal{C}_T$. Then, $h \circ_T (g \circ_T f) = h^* \circ (g \circ_T f) = h^* \circ (g^* \circ f)$ and $(h \circ_T g) \circ_T f = (h \circ_T g)^* \circ f = (h^* \circ g)^* \circ f$. Thus, $\mathcal{C}_T$ is a category.

We summarize entirely in terms of objects and arrows in $\mathcal{C}$, via the correspondence between $\text{Hom}_{\mathcal{C}_T}(A, B)$ and $\text{Hom}_{\mathcal{C}}(A, TB)$ given in the second construction axiom. That is, given $\mathcal{C}$-arrows $f : A \to TB$, $g : B \to TC$ and $h : C \to TD$, we interpret the $\mathcal{C}_T$ axioms in $\mathcal{C}$ as:

(KC1)	$\eta_B^* \circ f = f$	left unit
(KC2)	$f^* \circ \eta_A = f$	right unit
(KC3)	$h^* \circ (g^* \circ f) = (h^* \circ g)^* \circ f$	associativity.

Now, we can directly prove the equivalence between the Kleisli triple axioms (KT1)–(KT3) and the Kleisli category axioms (KC1)–(KC3). First, assume (KT1)–(KT3). By (KT1),

$$\begin{aligned} & \eta_B^* = \mathrm{id}_{TB} \\ \Rightarrow\ & \eta_B^* \circ f = \mathrm{id}_{TB} \circ f \\ \Rightarrow\ & \eta_B^* \circ f = f. \end{aligned}$$

Thus, (KC1) holds.

Trivially, we have (KC2) = (KT2).

(KT3) states that $h^* \circ g^* = (h^* \circ g)^*$. Thus,

$$\begin{aligned} h^* \circ (g^* \circ f) &= (h^* \circ g^*) \circ f \\ &= (h^* \circ g)^* \circ f, \end{aligned}$$

which implies (KC3).

Next, assume (KC1)–(KC3). By (KC1), $\eta_A^* \circ f = f$, for any f that has codomain TA. In particular, take $f = \mathrm{id}_{TA}$. Then,

$$\begin{aligned} & \eta_A^* \circ \mathrm{id}_{TA} = \mathrm{id}_{TA} \\ \Rightarrow\ & \eta_A^* = \mathrm{id}_{TA}, \end{aligned}$$

and so (KT1) holds.

Again, we immediately have (KT2) = (KC2).

Finally, (KC3) states that $h^* \circ (g^* \circ f) = (h^* \circ g)^* \circ f$. Recall that $g : B \to TC$, so that $g^* : TB \to TC$ and $(h^* \circ g)^* : TB \to TD$. Take f in (KC3) to be the identity arrow on TB, $f = \mathrm{id}_{TB}$. Then,

$$\begin{aligned} h^* \circ g^* &= (h^* \circ g)^* \circ \mathrm{id}_{TB} \\ &= (h^* \circ g)^*, \end{aligned}$$

and (KC3) implies (KT3). Thus, (KT1)–(KT3) are equivalent to (KC1)–(KC3), as desired.

2.2.1 Example: State Updates

We begin to concretize our work with triples by examining a side-effecting computation; namely, modifications to (variable) state. To show that state updates can be modelled using a suitable Kleisli triple, we must propose definitions for the three maps $(T, \eta, \_^*)$, and show that they satisfy the Kleisli triple axioms.

As briefly mentioned above, a computation with a state update can be thought of as a computation that takes an initial store or state variable, and returns a value (the output of the computation's underlying function) along with the modified store. For example, say that we have a simple program that adds several numbers together, and that we want to include a side-effecting computation in which the number of additions that the function performs is kept track of as the function is evaluated. This computation could accept `s = 0` as the initial store, and return the pair $\langle \texttt{sum}, \texttt{s}' \rangle$, where `sum` is the return value of the function, and $\texttt{s}'$ is the number of addition operations that were performed to obtain `sum`.

With this motivation, we define the first component of the state update triple to be the functor $T(\_) = (\_ \times S)^S$, where S is a nonempty set of stores [10]. We offer the corresponding maps η and $\_^*$ to form a Kleisli triple $(T, \eta, \_^*)$ that models state updates:

- $$\begin{array}{l} \eta_A : A \to (A \times S)^S \\ \quad a \mapsto \eta_A(a) : \begin{array}[t]{l} S \to A \times S \\ s \mapsto \langle a, s \rangle \end{array} \end{array}$$
- for $f : A \to (B \times S)^S$, $$f^* : \begin{array}[t]{lcl} (A \times S)^S & \to & (B \times S)^S, \\ \alpha & \mapsto & f^*(\alpha) : \begin{array}[t]{l} S \to B \times S \\ s \mapsto f(b)(s) \end{array} \end{array}$$

where $\langle b, s \rangle = \alpha(s)$.

At a glance, these maps appear a bit ghastly to deal with, so we introduce a small amount of notation that will make calculations easier. Any function $f : D \to R^S$ maps an element d of its domain to a map $f(d) : S \to R$. We can, of course, evaluate this function on an element $s \in S$ to obtain a value $r \in R$; that is, $f(d)(s) = r$. Thus, f can be interpreted as a function that accepts a pair of values and returns a single image value. It will be convenient to emphasize this using the following notation: given a map f as above, we define a (redundant) map $f_1 : D \to R^S$ by $f_1 \equiv f$, and a corresponding map $f_2 : D \times S \to R$, such that $\forall d \in D, \forall s \in S, f_2(\langle d, s \rangle) \overset{\text{def}}{=} f_1(d)(s)$.

As a word of caution, while this notation will prove to be expedient in the calculations below, there is one case in which it is somewhat counterintuitive; namely, with regard to the identity map. For example, if D is taken to be a function space, say $D = (A \times S)^S$, with $\alpha \in (A \times S)^S$ and $s \in S$, then we could write $\mathrm{id}_1(\alpha)(s) = \mathrm{id}_2(\langle \alpha, s\rangle)$. Notice that while the right-hand side of this equality is suggestive of an evaluation to $\langle \alpha, s\rangle$, the actual evaluation should be $\mathrm{id}_2(\langle \alpha, s\rangle) = (\mathrm{id}_1(\alpha))(s) = \alpha(s) \neq \langle \alpha, s\rangle$.

With this setup, the proposed Kleisli triple can be presented in a new form. We claim that the 3-tuple $(T, \eta, \_^*)$ defined by:

- $TA = (A \times S)^S \quad (S \neq \emptyset)$
- $\forall a \in A, \forall s \in S, \quad \eta_{A2}(\langle a, s\rangle) = \langle a, s\rangle$
- $\forall A, B \in O_{\mathcal{C}}$ with $f : A \to (B \times S)^S$ and $\alpha \in (A \times S)^S$, $f_2^*(\langle \alpha, s\rangle) = f_2(\alpha(s))$, where $f_2^* : (A \times S)^S \times S \to B \times S$ and $f_1^*(\alpha)(s) = f^*(\alpha)(s) = f_2^*(\langle \alpha, s\rangle)$

is a Kleisli triple.

We note here that a common way to define this triple is to use λ-calculus notation. We have avoided introducing that notation here, but give the formulation for reference [12]:

- $\eta_A(a) = \lambda s : S.\langle a, s\rangle$
- for $f : A \to (B \times S)^S$ and $\alpha \in (A \times S)^S$, $f^*(\alpha) = \lambda s : S.(\text{let } \langle a, s'\rangle = \alpha(s) \text{ in } f(a)(s'))$.

The notation λs : denotes that the function following the ':' is given in terms of the independent variable s, and the notation $S.$ denotes that the independent variable in the function following the '.' is restricted to S (in this case, we have $s \in S$).

We now show that the given 3-tuple satisfies (KT1)–(KT3) as follows. For (KT1), we have $\mathrm{id}_{TA} : (A \times S)^S \to (A \times S)^S$, where $\forall \alpha \in (A \times S)^S$

$$\mathrm{id}_{TA}(\alpha) = \alpha \iff \forall s \in S, \mathrm{id}_{TA}(\alpha)(s) = \alpha(s).$$

Thus, $\eta_A^* = \mathrm{id}_{TA}$ trivially by the second formulation for η given above.

For (KT2), let $f : A \to (B \times S)^S, a \in A, s \in S$. Then,

$$\begin{aligned} (f^* \circ \eta_A)_1(a)(s) &= f_1^*(\eta_{A1}(a))(s) \\ &= f_2(\eta_{A1}(a)(s)) \\ &= f_2(\langle a, s\rangle) \\ &= f_1(a)(s). \end{aligned}$$

Thus, $f^* \circ \eta_A = f$.

To prove that (KT3) holds, we first prove the following proposition:

Proposition 2.2.1. *Given maps $f : A \to (B \times S)^S$ and $g : B \to (C \times S)^S$, we have:*

$$g_1^* \circ f_1 = g_2 \circ f_2.$$

Proof. Let f and g be as given, and let $a \in A, s \in S$. Then,

$$\begin{aligned}(g_1^* \circ f_1)_1(a)(s) &= g_1^*(f_1(a))(s) \\ &= g_2(f_1(a)(s)) \\ &= g_2(f_2(\langle a, s\rangle)) \\ &= (g_2 \circ f_2)_2(\langle a, s\rangle) \\ &= (g_2 \circ f_2)_1(a)(s),\end{aligned}$$

as desired. ∎

Now, to show (KT3), let $\alpha \in (A \times S)^S$ and $s \in S$. We have:

$$\begin{aligned}(g^* \circ f)_1^*(\alpha)(s) &= g_1^*(f_1^*(\alpha))(s) \\ &= g_2(f_1^*(\alpha)(s)) \\ &= g_2(f_2(\alpha(s))) \\ &= (g_2 \circ f_2)_2(\alpha(s)) \\ &= (g_1^* \circ f_1)_2(\alpha(s)) && \text{proposition} \\ &= (g_1^* \circ f_1)_1^*(\alpha)(s).\end{aligned}$$

Thus, $g^* \circ f^* = (g^* \circ f)^*$ and $(T, \eta, \_^*)$ is a Kleisli triple.

2.3 Triples

While Kleisli triples are intuitively justifiable as a means to theoretically model computations, they are equivalent to another very interesting category-theoretic construction; namely... monads! Incidently, the equivalence explains the other name "triples" for monads that appears in much of the literature.

Theorem 2.3.1. *There is a one-to-one correspondence between Kleisli triples and monads.*

Proof. Given a Kleisli triple $(T, \eta, \_^*)$, we claim that the corresponding monad is (T, η, μ), where: (i) T is the extension of the function $T : O_\mathcal{C} \to O_\mathcal{C}$ to the endofunctor $T : \mathcal{C} \to \mathcal{C}$, defined on any $\mathcal{C}$-arrow $f : A \to B$ by $T(f) = (\eta_B \circ f)^*$ and (ii) $\mu_A = \mathrm{id}_{TA}^*$. To prove this, we first show that T is indeed a functor. Let $f : A \to B, g : B \to C$. Then,

$$\begin{array}{rcll}
T(g \circ f) & = & (\eta_C \circ (g \circ f))^* & \\
& = & ((\eta_C \circ g) \circ f)^* & \text{associativity in } \mathcal{C} \\
& = & (((\eta_C \circ g)^* \circ \eta_B) \circ f)^* & \text{(KT2)} \\
& = & ((\eta_C \circ g)^* \circ (\eta_B \circ f))^* & \\
& = & (\eta_C \circ g)^* \circ (\eta_B \circ f)^* & \text{(KT3)} \\
& = & Tg \circ Tf. &
\end{array}$$

Further,

$$T(\mathrm{id}_A) = (\eta_A \circ \mathrm{id}_A)^* = \eta_A^* = \mathrm{id}_{TA}.$$

Thus, T is an endofunctor on $\mathcal{C}$.

Next, we show that η and μ are natural transformations. Let $f : A \to B$ be an arrow in $\mathcal{C}$. For η, we have

$$\begin{array}{rcll}
Tf \circ \eta_A & = & (\eta_B \circ f)^* \circ \eta_A & \text{defn of extension of } T \\
& = & \eta_B \circ f. & \text{(KT2)}
\end{array}$$

Thus, η is a natural transformation.

To show μ is natural, we have

$$\begin{array}{rcll}
Tf \circ \mu_A & = & (\eta_B \circ f)^* \circ \mu_A & \text{defn of } T \\
& = & (\eta_B \circ f)^* \circ \mathrm{id}_{TA}^* & \text{defn of } \mu_A \\
& = & ((\eta_B \circ f)^* \circ \mathrm{id}_{TA})^* & \text{(KT3)} \\
& = & ((\eta_B \circ f)^*)^* & \\
& = & (\mathrm{id}_{TB} \circ (\eta_B \circ f)^*)^* & \\
& = & ((\mathrm{id}_{TB}^* \circ \eta_{TB}) \circ (\eta_B \circ f)^*)^* & \text{(KT2)} \\
& = & (\mathrm{id}_{TB}^* \circ (\eta_{TB} \circ (\eta_B \circ f)^*))^* & \\
& = & \mathrm{id}_{TB}^* \circ (\eta_{TB} \circ (\eta_B \circ f)^*)^*) & \text{(KT3)} \\
& = & \mu_B \circ (\eta_{TB} \circ (\eta_B \circ f)^*)^* & \text{defn of } \mu_B \\
& = & \mu_B \circ T(\eta_B \circ f)^* & \text{defn of } T \\
& = & \mu_B \circ T(Tf). & \text{defn of } T
\end{array}$$

Thus, μ is a natural transformation.

We now show that the monad laws are satisfied. Note that $\mathrm{id}_{TA}^* : T^2A \to TA$ and $\mathrm{id}_{T^2A}^* : T^3A \to T^2A$. Thus, we have

$$\begin{array}{rcll}
\mu_A \circ \mu_{TA} & = & \mathrm{id}_{TA}^* \circ \mathrm{id}_{T^2A}^* & \\
& = & (\mathrm{id}_{TA}^* \circ \mathrm{id}_{T^2A})^* & \text{(KT3)} \\
& = & (\mathrm{id}_{TA}^*)^* & \\
& = & (\mathrm{id}_{TA} \circ \mathrm{id}_{TA}^*)^* & \\
& = & ((\mathrm{id}_{TA}^* \circ \eta_{TA}) \circ \mathrm{id}_{TA}^*)^* & \text{(KT2)} \\
& = & (\mathrm{id}_{TA}^* \circ (\eta_{TA} \circ \mathrm{id}_{TA}^*))^* & \\
& = & \mathrm{id}_{TA}^* \circ (\eta_{TA} \circ \mathrm{id}_{TA}^*)^* & \text{(KT3)}
\end{array}$$

$$\begin{aligned} &= \mathrm{id}^*_{TA} \circ T(\mathrm{id}^*_{TA}) \\ &= \mu_A \circ T\mu_A. \end{aligned}$$

To show that the proposed 3-tuple satisfies the second monad law, we compute:

$$\begin{aligned} \mu_A \circ \eta_{TA} &= \mathrm{id}^*_{TA} \circ \eta_{TA} & \\ &= \mathrm{id}_{TA} & \text{(KT2)} \\ &= \eta^*_A & \text{(KT1)} \\ &= (\mathrm{id}_{TA} \circ \eta_A)^* & \\ &= ((\mathrm{id}^*_{TA} \circ \eta_{TA}) \circ \eta_A)^* & \text{(KT2)} \\ &= (\mathrm{id}^*_{TA} \circ (\eta_{TA} \circ \eta_A))^* & \\ &= \mathrm{id}^*_{TA} \circ (\eta_{TA} \circ \eta_A)^*. & \text{(KT3)} \end{aligned}$$

Since η_A is an arrow from A to TA, by the definition of the extension of T from a function to a functor given in Theorem 2.3.1, we have $T(\eta_A) = (\eta_{TA} \circ \eta_A)^*$. Thus, .

$$\begin{aligned} &= \mathrm{id}^*_{TA} \circ T\eta_A & \text{defn of extension of } T \\ &= \mu_A \circ T\eta_A. & \end{aligned}$$

Thus, the Kleisli triple gives rise to a monad.

To show the converse, if we are given a monad (T, η, μ), then we claim the corresponding Kleisli triple is $(T, \eta, \_^*)$, where: (i) T is the restriction of the endofunctor T to the objects of $\mathcal{C}$ and (ii) for $f : A \rightarrow TB \in A_{\mathcal{C}}$, $f^* = \mu_B \circ (Tf)$. We show that this 3-tuple is indeed a Kleisli triple as follows. First,

$$\eta^*_A = \mu_A \circ T\eta_A = \mathrm{id}_{TA},$$

by the second monad law. Thus, (KT1) holds. Next, we compute

$$\begin{aligned} f^* \circ \eta_A &= (\mu_B \circ Tf) \circ \eta_A & \\ &= \mu_B \circ \eta_{TB} \circ f & \\ &= \mathrm{id}_{TB} \circ f & \text{monad law 2} \\ &= f. & \end{aligned}$$

Thus, we have (KT2). Finally, we show that (KT3) holds by

$$\begin{aligned} g^* \circ f^* &= (\mu_C \circ Tg) \circ (\mu_B \circ Tf) & \\ &= \mu_C \circ \mu_{TC} \circ T^2 g \circ Tf & \\ &= \mu_C \circ T\mu_C \circ T^2 g \circ Tf & \text{monad law 1} \\ &= \mu_C \circ T(\mu_C \circ Tg \circ f) & \\ &= (\mu_C \circ Tg \circ f)^* & \\ &= (g^* \circ f)^*, & \end{aligned}$$

which implies that $(T, \eta, \_^*)$ is a Kleisli triple. Therefore, there is a one-to-one correspondence between Kleisli triples and monads. ■

This correspondence immediately gives us an alternative description for Kleisli categories, which is stated thus [8]: given a monad (T, η, μ) in a category $\mathcal{C}$, assign to each object $A \in \mathcal{C}$ a new object A_T and to each $\mathcal{C}$-arrow $f : A \to TB$ a new arrow $f^K : A_T \to B_T$. These new objects and arrows constitute the Kleisli category of the monad $\mathcal{C}_T$ when the composition of $\mathcal{C}_T$-arrows $f^K : A_T \to B_T$, $g^K : B_T \to C_T$ is defined by $g^K \circ f^K = (\mu_C \circ Tg \circ f)^K$. Proofs of the right and left unit laws and the associative law can be summarized with the following commutative diagrams, where $f : A \to TB$, $g : B \to TC$, $h : C \to TD \in A_{\mathcal{C}}$. Notice that the identity arrow for $\mathcal{C}_T$ is $(\eta_A)^K : A_T \to A_T$.

$$f^K \circ (\eta_A)^K = f^K :$$

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

$$(\eta_C)^K \circ g^K = g^K :$$

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, "g"'] & TC \arrow[d, "T\eta_C"'] \\
TC & T^2C \arrow[l, "\mu_C"']
\end{tikzcd}$$

$$h^K \circ (g^K \circ f^K) = (h^K \circ g^K) \circ f^K :$$

$$\begin{tikzcd}
T^2C \arrow[r, "T^2h"] \arrow[d, "\mu_C"'] & T^3D \arrow[r, "T\mu_D"] \arrow[d, "\mu_{TD}"'] & T^2D \arrow[d, "\mu_D"'] \\
TC \arrow[r, "Th"] & T^2D \arrow[r, "\mu_D"] & TD
\end{tikzcd}$$

2.3.1 Example Revisited: State Updates

Using the correspondence given in Theorem 2.3.1, we can now reformulate our earlier example in terms of monadic, rather than Kleisli, maps. Recall that for state updates, we define the endofunctor T to act on objects by $TA = (A \times S)^S$. Thus, from the theorem we have

$$\mu_A = \mathrm{id}_{TA}^* : ((A \times S)^S \times S)^S \to (A \times S)^S.$$

We can semantically interpret this map in terms of state modifications by observing how this function behaves on input values. Let $f \in ((A \times S)^S \times S)^S, f' \in (A \times S)^S, a \in A$ and $s, s', s'' \in S$. Further, let $f(s) = \langle f', s' \rangle$ and $f'(s') = \langle a, s'' \rangle$. Then,

$$\begin{aligned} \mu_A(f)(s) &= \mathrm{id}^*_{TA1}(f)(s) \\ &= \mathrm{id}_{TA2}(f(s)) \\ &= \mathrm{id}_{TA2}(\langle f', s' \rangle) \\ &= \mathrm{id}_{TA1}(f')(s') \\ &= f'(s') \\ &= \langle a, s'' \rangle . \end{aligned}$$

That is, $\mu_A(f)$ is a computation which, on input s (a store), first computes a pair comprised of a new computation and a modified store, $f(s) = \langle f', s' \rangle$ and then outputs the value-store pair $f'(s') = \langle a, s'' \rangle$. We note that in the λ-calculus, this would typically be denoted by $\mathrm{id}^*_{TA}(f) = \lambda s : S.\mathrm{eval}(fs)$.

The theorem likewise supplies definitions for η and for the action of T on a given function. Using these, we can verify that the monad laws hold. First, we show equality of the outputs returned by each of the functions $\mu_A \circ \mu_{TA}$ and $\mu_A \circ T\mu_A$ on input $\langle f, s \rangle$, for $f \in T^3 A = (((A \times S)^S \times S)^S \times S)^S$ and $s \in S$. To facilitate this, let $f' \in T^2 A, f'' \in TA, a \in A$ and $s', s'' \in S$, with

$$\begin{aligned} f(s) &= \langle f', s' \rangle \\ f'(s') &= \langle f'', s'' \rangle \\ f''(s'') &= \langle a, s''' \rangle . \end{aligned}$$

We have:

$$\begin{aligned} (\mu_A \circ \mu_{TA})_1(f)(s) &= (\mathrm{id}^*_{TA} \circ \mathrm{id}^*_{T^2A})_1(f)(s) \\ &= \mathrm{id}^*_{TA1}(\mathrm{id}^*_{T^2A1}(f))(s) \\ &= \mathrm{id}_{TA2}(\mathrm{id}^*_{T^2A1}(f)(s)) \\ &= \mathrm{id}_{TA2}(\mathrm{id}_{T^2A2}(f(s))) \\ &= \mathrm{id}_{TA2}(\mathrm{id}_{T^2A2}(\langle f', s' \rangle)) \\ &= \mathrm{id}_{TA2}(\mathrm{id}_{T^2A1}(f')(s')) \\ &= \mathrm{id}_{TA2}(f'(s')) \\ &= \mathrm{id}_{TA1}(f'')(s'') \\ &= \langle a, s''' \rangle \end{aligned}$$

and

$$\begin{aligned}
(\mu_A \circ T\mu_A)(f)(s) &= \mathrm{id}^*_{TA} \circ (\eta_{TA} \circ \mathrm{id}^*_{TA})^*)(f)(s) \\
&= \mathrm{id}^*_{TA1}((\eta_{TA} \circ \mathrm{id}^*_{TA})_1)^*(f))(s) \\
&= \mathrm{id}_{TA2}((\eta_{TA} \circ \mathrm{id}^*_{TA})^*_1(f)(s)) \\
&= \mathrm{id}_{TA2}((\eta_{TA} \circ \mathrm{id}^*_{TA})_1(f')(s')) \\
&= \mathrm{id}_{TA2}(\eta_{TA1}(\mathrm{id}^*_{TA1}(f'))(s')) \\
&= \mathrm{id}_{TA2}(\langle \mathrm{id}^*_{TA1}(f'), s'\rangle) \\
&= \mathrm{id}^*_{TA1}(f')(s') \\
&= \mathrm{id}_{TA2}(f'(s')) \\
&= \mathrm{id}_{TA2}(\langle f'', s''\rangle) \\
&= f''(s'') \\
&= \langle a, s'''\rangle.
\end{aligned}$$

Thus, the first monad diagram commutes.

Similarly, we compute the return values for $\mu_A \circ \eta_{TA}$ and $\mu_A \circ T\eta_A$ on $\langle f'', s''\rangle \in (A \times S)^S \times S$ as follows:

$$\begin{aligned}
(\mu_A \circ \eta_{TA})_1(f'')(s'') &= (\mathrm{id}^*_{TA} \circ \eta_{TA})_1(f'')(s'') \\
&= \mathrm{id}^*_{TA1}(\eta_{TA1}(f''))(s'') \\
&= \mathrm{id}_{TA2}(\eta_{TA1}(f'')(s'')) \\
&= \mathrm{id}_{TA2}(\langle f'', s''\rangle) \\
&= f''(s'') \\
&= \mathrm{id}_{TA}(f'')(s'')
\end{aligned}$$

and

$$\begin{aligned}
(\mu_A \circ T\eta_A)(f'')(s'') &= (\mathrm{id}^*_{TA} \circ T\eta_A)_1(f'')(s'') \\
&= (\mathrm{id}^*_{TA} \circ (\eta_{TA} \circ \eta_A)^*)_1(f'')(s'') \\
&= \mathrm{id}^*_{TA1}((\eta_{TA} \circ \eta_A)^*_1(f''))(s'') \\
&= \mathrm{id}_{TA2}((\eta_{TA} \circ A)^*_1(f'')(s'')) \\
&= \mathrm{id}_{TA2}((\eta_{TA} \circ \eta_A)_2(f''(s''))) \\
&= \mathrm{id}_{TA2}((\eta_{TA} \circ \eta_A)_1(a)(s''')) \\
&= \mathrm{id}_{TA2}(\eta_{TA1}(\eta_{A1}(a))(s''')) \\
&= \mathrm{id}_{TA2}(\langle \eta_{A1}(a), s'''\rangle) \\
&= \eta_{A1}(a)(s''')
\end{aligned}$$

$$\begin{aligned} &= \langle a, s''' \rangle \\ &= f''(s''). \end{aligned}$$

Thus, we have verified that the 3-tuple (T, η, μ) for state updates is a monad.

2.4 Strong Monads

We are about to investigate the connections between our work thus far with the so-called "monads" of Haskell. The implementation of monads in Haskell was led by Philip Wadler, whose work on the subject uses the theory of Moggi's notions of computations. We conclude this chapter by examining another category-theoretic concept used in Moggi's work, which is technically required for Wadler's implementation.

In Section 1.4.1, we commented that the λ-calculus served as a foundation for many functional languages, such as Haskell. Since Moggi's aim was to provide a categorical semantics of computations for studying languages based on typed λ-calculi, his theory had to be rich enough to be able to interpret λ-terms. We have already noted the equivalence between the λ-calculus and cartesian closed categories, but a monad over a cartesian closed category does not have sufficient structure to interpret λ-terms in a computational model [10].

One problem that arises involves a form of local definition that is used in functional languages. A local definition is the introduction of an association between a name and value for use within an expression, and the form we are interested in here is called the *let-constructor*. The **let**-constructor is equivalent to function application, and we briefly motivate its definition before explaining its relevance in Moggi's computational λ-calculus.

A typical λ-function applied to an argument can be represented as

$$\lambda\langle\text{name}\rangle.\langle\text{body}\rangle\ \langle\text{argument}\rangle,$$

where $\langle$name$\rangle$ is a bound variable, and $\langle$body$\rangle$ and $\langle$argument$\rangle$ are λ-expressions. The function evaluation requires that we replace all free occurrences of $\langle$name$\rangle$ in $\langle$body$\rangle$ with $\langle$argument$\rangle$, prior to evaluating $\langle$body$\rangle$. Equivalently, we can think of associating $\langle$name$\rangle$ and $\langle$argument$\rangle$ throughout the evaluation of $\langle$body$\rangle$. We notationally capture this equivalent conception of function evaluation using the **let**-constructor [9]:

$$\begin{aligned} &\textbf{let}\ \langle\text{name}\rangle = \langle\text{argument}\rangle \\ &\textbf{in}\ \langle\text{body}\rangle. \end{aligned}$$

As referred to in Section 2.2, the **let**-constructor is used to define the extension of a function f to f^*; specifically, given a notion of computation T, types A, B, a program $f : A \to TB$, and a denotation of a program $c \in TA$, we define the extension of f by $f^*(c) = (\mathbf{let}\ x = c\ \mathbf{in}\ f(x))$. Despite its importance, the interpretation of an expression such as $(\mathbf{let}\ x = e\ \mathbf{in}\ e')$ is problematic in a category with finite products and a monad, when e' has free variables aside from x, as is shown in [10]. Essentially, the problem is that even in a category with cartesian products and a monad, there is not enough structure to transform a value-computation or computation-computation pair into a computation of a pair. A similar problem arises in the interpretation of functional types, in which it is necessary to obtain a computation that computes a pair from a given pair of computations. We can resolve these problems by requiring our monad to be a strong monad—a monad along with a natural transformation that is defined as follows.

A *strong monad* over a category $\mathcal{C}$ with finite products is a monad (T, η, μ) together with a natural transformation t with value $t_{A,B}$ from $A \times TB$ to $T(A \times B)$, such that the following diagrams are commutative [11]:

$$\begin{tikzcd}
1 \times TA \arrow[r, "t_{1,A}"] \arrow[d, "r_{TA}"'] & T(1 \times A) \arrow[d, "Tr_A"] \\
TA \arrow[r, equal] & TA
\end{tikzcd}$$

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

$$\begin{tikzcd}
A \times B \arrow[rr, "\mathrm{id}_{A \times B}"] \arrow[d, "\mathrm{id}_A \times \eta_B"'] & & A \times B \arrow[d, "\eta_{A \times B}"] \\
A \times TB \arrow[rr, "t_{A,B}"] & & T(A \times B) \\
A \times T^2B \arrow[r, "t_{A,TB}"] \arrow[u, "\mathrm{id}_A \times \mu_B"] & T(A \times TB) \arrow[r, "Tt_{A,B}"] & T^2(A \times B) \arrow[u, "\mu_{A \times B}"']
\end{tikzcd}$$

where $r_A : 1 \times A \dot{\rightarrow} A$ and $\alpha_{A,B,C} : (A \times B) \times C \dot{\rightarrow} A \times (B \times C)$ are natural transformations.

The given natural transformation $t_{A,B}$ acts to transform a value-computation pair into a computation of a pair of values in the following manner:

$$\begin{array}{rlcl} t_{A,B}: & A \times TB & \rightarrow & T(A \times B) \\ & \langle a, c \rangle & \mapsto & (\mathbf{let}\ y = c\ \mathbf{in}\ \langle a, y \rangle). \end{array}$$

For example, in the case of state updates, the natural transformation $t_{A,B}(a, c)$ is a function which, given a store s returns the computation ($\mathbf{let}\ \langle b, s' \rangle = c(s)\ \mathbf{in}\ \langle \langle a, b \rangle, s' \rangle$) [12].

Since Haskell is based on the λ-calculus, the applied category-theoretic concepts we explore must be compatible with the interpretation of λ-terms. Hence, the "monads" that we will examine in Haskell are actually strong monads.

Chapter 3

Wadler's Monads

By now, it should not be a surprise that in Haskell, *monads* are data types that are used to represent computations. If `m` is a monad, then an object of type `m a` represents a computation with return type `a`. The definition of a particular monad will reflect the sort of side effect that the monad is trying to computationally mimic. Its use will ensure that type checker can verify that the side-effects are managed consistently throughout the program. In this way, monads are a means to encode the behaviour of computations, as opposed to values, in Haskell's type system. Before we examine the `Monad` class declaration, let us informally consider a motivating example.

3.0.1 The Maybe Monad

Consider a function or operation that is undefined at one or more values: for example, division. In a composition of functions in which at least one of the functions involves division, an attempt to divide by zero means that the computation can not be completed. For instance, consider the simple function

```
f ::  Int -> Int -> Int -> Float
f x y z = (1 / x) + 1 / (y + z)
```

On initial inspection, it appears that before this function can be evaluated we must check to see if both of the terms are actually integers. In general terms, to add error handling to a program written in a purely functional language, the result type of the program would have to be modified to include error values, and failures would have to be checked for and handled appropriately at each recursive call in the program. We can avoid such failure checking, however, by carrying out such an operation in the `Maybe` monad. In Haskell, `Maybe` is the name of both a data type and a monad, with the declaration of the former given as follows:

```
data Maybe a = Just a | Nothing
```

As a technical aside, we note the difference between 'data' declarations and 'type' declarations. The former introduces a new data type (for example, `Maybe`) and new constructors for values of that type (`Just` and `Nothing`), while the latter defines a synonym for a pre-existing type. Either kind of declaration can be used in the definition of a monad.

In the `Maybe` monad, the validity of an expression such as that given above can be implicitly checked and used to determine the value that is returned: if there are no undefined terms in the expression, the function will return a `Maybe` value of type `Just`, whereas an erroneous evaluation will return `Nothing`. For example, if our function above is evaluated using an implementation of the `Maybe` monad, then `f 1 0 1` would return `Just 2` and `f 1 -1 1` would return `Nothing`.

The benefit is that this function can now be composed with other functions and, in the case of failure, the "error" will cascade through the program in a safe and predictable way. Note that in a series of computations that included the latter evaluation of `f` above, any computations that did not involve `f` would evaluate as normal, and computations that did involve `f` would now return `Nothing`. In general, the `Maybe` monad provides a means to combine a chain of computations, each of which may return `Nothing`, by terminating the chain early if any step does in fact return `Nothing`.

3.0.2 Definition of a Monad

In the context of Haskell, we define a *monad* to be a 3-tuple `(M, return, >>=)`, where `M` is a type constructor and `return` and `>>=` (read "bind") are polymorphic functions with type signatures as shown in the (minimal) class declaration below:

```
class Monad m where
  return ::  a -> m a
  (>>=) ::  m a -> (a -> m b) -> m b
```

Let us examine these methods in detail. For use as a model monad by which to exemplify the concepts discussed, we provide the instance declaration of the `Maybe` monad:

```
instance Monad Maybe where
   return a       = Just a
   Nothing >>= f  = Nothing
   Just x >>= f   = f x
```

The `return` Operator

Any compositions within a monad must involve monadic-typed arguments. Informally, the purpose of the `return` operator is to get us into the monad. More specifically, it converts a value into a computation that returns a value of the same type—that is, into a corresponding monadic type. The technical means by which this is done will depend on the particular monad under consideration, but in a general sense, `return` carries a value to the "simplest" corresponding monadic-typed value, while leaving the original value otherwise unaltered.

As shown in the instance declaration, `return` is defined to be `Just` in the `Maybe` monad.

The `(>>=)` Operator

For a monad `M`, The infix operator `(>>=)` provides a way to apply a function of type `a -> M b` to a computation of type `M a`. Intuitively, we can think of the operation `(M a) >>= f` in the following way. First, the monadic structure of the argument `M a` is removed to unveil any number of values of type `a` inside. The actual means of doing this is specific to the definition of `(>>=)` for the particular monad under consideration. These non-monadic values can then be passed to `f` one at a time, with each application of `f` to a value of type `a` producing a corresponding monadic element of type `M b`. Next, the monadic structure of each of these `f`-output elements is removed, yielding a collection of elements of type `b`. Finally, the monadic structure is reassembled over this collection (in a way that must be defined specifically for the particular monad `M` under consideration), producing a single monadic element of the correct type, `M b`. We may refer to the process in its entirety as "binding `M a` to `f`."

As an example of this process, consider the `Maybe` monad, in which we have `(Just x) >>= f = f x`, for `x :: t` and `f :: a -> Maybe a`. That is, the monadic layer of `Just x` is "removed" and just the value `x` is unveiled. The value `x` is then passed to `f :: a -> Maybe a`, which returns a `Maybe`-typed value (either `Just y` for some `y`, or `Nothing`). As is also shown in the instance declaration, `Nothing >>= k = Nothing`. This is consistent with the intuitive idea that no value is extracted from `Nothing`, so nothing is passed to `k` and no output value is produced.

Each instance of the monad class must satisfy 3 laws in order to behave properly in the program. Note that for user-defined monads, it is up to the user to ensure that the laws are satisfied, since it is impossible for the type checker to do so [6]. The statement of the laws makes use of Haskell syntax that is based on shorthand λ-calculus notation; namely, an expression of the form $\backslash x \rightarrow e(x)$ explicitly defines the function that maps a variable x to an expression $e(x)$, with $e(x)$ dependent on x. That is, we code the λ-calculus expression $\lambda x : e(x)$ in Haskell as `\x -> e(x)`. For example, the expression `\x -> sin x` is similar to defining the function $e(x) = \sin x$ in standard mathematical notation, except that we do not need to explicitly give the function defined by the former expression a name (such as 'e' in the latter). The Haskell notation is helpful to use in conjunction with expressions involving more than one occurrences of the (infix) operator `(>>=)`, which accepts a monadic element on the left and a function on the right.

With this, the Haskell monad laws are:

```
(M1)        return x >>= f = f x
(M2)        n >>= return = n
(M3)        (n >>= f) >>= g = n >>= (\x -> (f x >>= g)),
```

where `x :: a`, `n :: m a`, `f :: a -> m b` and `g :: b -> m c`. The first two monad laws ensure that `return` preserves all information about its argument. That is, for (M1), if we define a trivial computation `return` that accepts a value `a` and returns `a`, and bind this computation with another computation `f`, then the result is the same as if we had applied `f` to `a` directly. For (M2), if we perform the computation `n` and then pass the result to the trivial computation `return`, the result is the same as if we had just performed the computation `n`. The third law (M3) is essentially an associativity law for `(>>=)`, which accounts for the type signature of the operator.

Given this definition of a monad and the associated laws, we raise our primary question of interest:

What is the precise connection between monads as defined and used in Haskell, and the category theoretic definition of monads discussed in Section 1.3?

3.1 Examples

We present further examples of monads in Haskell, so as to build an intuition of how they work.

1. The simplest monad is the identity monad, with the triple defined as:

   ```
   type Id a   =   a
   return      ::  a -> Id a
   return a    =   a
   (>>=)       ::  Id a -> (a -> Id b) -> Id b
   x >>= f     =   f x.
   ```

   Here, `Id` is the identity function restricted to types (implicitly), `return` is the identity function, and `(>>=)` simply evaluates `f x`. Were this monad implemented, no non-trivial computation would take place along with the regular evaluation of the function.

2. We have already investigated in detail the theory behind state updates as monads, and we continue this example with a Haskell implementation. To begin, we give a simple Haskell evaluator whose task is to compute a quotient of two integers [17]. We define a data type `Term` by:

   ```
   data Term = Con Int | Div Term Term,
   ```

   where a term is to be either a constant `Con a` for an integer `a`, or a quotient `Div t u` of terms `t` and `u`. The evaluator is then simply:

   ```
   eval            ::  Term -> Int
   eval (Con a)    =   a
   eval (Div t u)  =   eval t `quot` eval u.
   ```

   As indicated by the type signature, the function `eval` accepts a term and returns an integer. If the term is a constant (that is, an integer rather than an expression involving a division), then that constant is returned. If the term is a quotient, then the expressions comprising the numerator and denominator are: (i) evaluated as subterms; (ii) the division is computed once the subterms are reduced to constants; (iii) the quotient is returned (the function `` `quot` `` is integer division). For example, we have

   ```
   eval(Div(Div(Con 11904)(Con 2))(Con 3)) = 1984,
   ```

   With this as an evaluator template, we can consider making modifications that mimic side-effects. Say we want to keep track of the number of divisions performed during a calculation made by `eval` using state updates. We begin with Moggi's notion of computation and declare two data types:

```
type M a = State -> (a, State)
type State = Int.
```

Certainly, the evaluator can be adapted to the task of counting operations as it calculates, without the use of monads. However, the alteration is tedious because for each call of the evaluator, the current state must be passed in, and the updated state must be extracted from the result and passed on appropriately.

The modified evaluator can be coded to require an additional input integer that is to be thought of as a counter variable, with `s0` being the initial state. Divisions are carried out using recursive substitutions (until a constant is reached), where each recursive step also involves substituting an incremented state variable value for the previous state variable value. This can be done using the **let**-constructor syntax in Haskell. Haskell code involving `let` statements is to be interpreted as the obvious substitution operation: **let** $x = x'$ **in** $E(x)$ means substitute x' for x in expression E. Such a modified evaluator could look like:

```
eval                ::  Term -> Int -> M Int
eval (Con a) s0     =   (a,s0)
eval (Div t u) s0   =   let (a,s1) = eval t s0 in
                        let (b,s2) = eval u s1 in
                        (a 'quot' b, s2+1)
```

To demonstrate its operation, an input of

```
eval(Div(Div(Con 11904)(Con 2))(Con 3)) 0
```

corresponds to `t = Div(Con 11904)(Con 2)`, `u = Con 3` and `s0 = 0`. Bearing in mind Haskell's lazy evaluation, this computation proceeds as:

- **let** $(a, s1)$ = eval Div(Con 11904)(Con 2) 0 **in**
  let $(b, s2)$ = eval (Con 3) $s1$ **in**
  $(a$ 'quot' $b, s2 + 1)$

- **let** $(a, s1)$ = eval Div(Con 11904)(Con 2) 0 **in**
  let $(b, s2) = (3, s1)$ **in**
  $(a$ 'quot' $b, s2 + 1)$

- **let** $(a, s1)$ = eval Div(Con 11904)(Con 2) 0 **in**
  $(a$ 'quot' $3, s1 + 1)$

- $\textbf{let } (a, s1) = (\textbf{let } (a', s1') = \text{ eval (Con 11904) } 0 \textbf{ in}$
  $\textbf{let } (b', s2') = \text{ eval (Con 2) } s1' \textbf{ in}$
  $(a' \text{ `quot` } b', s2' + 1)) \textbf{ in}$
  $(a \text{ `quot` } 3, s1 + 1)$
- $\textbf{let } (a, s1) = (\textbf{let } (a', s1') = \text{ eval (Con 11904) } 0 \textbf{ in}$
  $\textbf{let } (b', s2') = (2, s1') \textbf{ in}$
  $(a' \text{ `quot` } b', s2' + 1)) \textbf{ in}$
  $(a \text{ `quot` } 3, s1 + 1)$
- $\textbf{let } (a, s1) = (\textbf{let } (a', s1') = \text{ eval (Con 11904) } 0 \textbf{ in}$
  $(a' \text{ `quot` } 2, s1' + 1)) \textbf{ in}$
  $(a \text{ `quot` } 3, s1 + 1)$
- $\textbf{let } (a, s1) = (\textbf{let } (a', s1') = (11904, 0) \textbf{ in}$
  $(a' \text{ `quot` } 2, s1' + 1)) \textbf{ in}$
  $(a \text{ `quot` } 3, s1 + 1)$
- $\textbf{let } (a, s1) = (11904 \text{ `quot` } 2, 0 + 1) \textbf{ in}$
  $(a \text{ `quot` } 3, s1 + 1)$
- $(5952 \text{ `quot` } 3, 1 + 1)$
- $(1984, 2)$

As desired, the return of the computation is the ordered pair that has the result of the division calculation as its first component and the number of divisions performed as its second.

To implement monadic abstractions in the evaluator, we re-define the evaluation function in terms of the `return` and `>>=` operations. As offered in [17], we have:

```
eval            :: Term -> M Int
eval (Con a)    =  return a
eval (Div t u)  =  eval t >>= \a ->
                     eval u >>= \b -> return a 'quot' b,
```

noting that the compiler implicitly inserts brackets into the above expression as follows (that is, the compiler automatically interprets the expression with the following precedence):

$$\text{eval (Div } t\ u) \;=\; ((\text{eval } t) >>= (\backslash a \rightarrow$$
$$((\text{eval } u) >>= (\backslash b \rightarrow (\text{return } (a \text{ `quot` } b)))))).$$

This monadic evaluator can be verbally described as follows: the type signature indicates that the evaluator accepts a term and performs a computation with return type `Int`. To evaluate a constant term, simply return it. A computation of the form (Div t u) is performed by first computing t (which may be a constant or an expression involving the `Div` operator) and binding a to the result, then computing u and binding b to the result, and lastly returning a 'quot' b.

Informally, the expression `t` is simplified to a constant, this result being a monadic type `M Int`. The integral value contained in the monadic-typed term is extracted from the monad and passed as an argument to the function `\a -> eval u >>= \b -> return a 'quot' b`. In standard mathematical notation, if we call the extracted integral value c and label the preceding function as $f(a)$, then the last step amounts to removing the monadic layer from the output of eval t to obtain c, and then computing $f(c)$. Similarly, the expression `eval u >>= \b -> return a 'quot' b` simplifies `u` to a constant element in `M Int` and passes it as an argument to a function that maps input `b` to the (monadic-typed) constant `a 'quot' b`.

Essentially, then, all that is happening is that the given expressions `t` and `u` are simplified to constants and the quotient of these constants is returned. The `>>=` operation allows us to easily pass along the simplified results, accounting for the monadic types as required.

Having offered a general monadic (division) evaluator, we now define the triple specific to the case of state updates. In the following code, `return a` returns the computation that accepts an initial state `x` and returns the pair comprised of the value `a` and the unchanged state variable `x`. The call `m >>= f` first performs the computation `m` in the initial state `s0`. This computation yields the value `a` and updated state `s1`. Then, the computation `f a` is performed in state `s1`, which returns the pair comprised of the value `b` and the final state `s2`. The triple is given as:

```
type M a     =   State -> (a, State)
type State   =   Int
return       ::  a -> M a
return a     =   \s0 -> (a, s0)
(>>=)        ::  M a -> (a -> M b) -> M b
m >>= f      =   \s0 -> let (a,s1) = m s0 in
                        let (b,s2) = f a s1 in
                        (b,s2)
```

To complete the task of counting the number of divisions that are performed during a calculation, we define a method `count` by:

```
count  ::  M()
count  =   \s -> ((),s+1)
```

This function increments the state by 1 and returns the empty value `()`. To count the number of divisions as they occur in a calculation, we simply replace the call `return a 'quot' b` by `count >>= \_ -> return a 'quot' b` in the original monadic evaluator [17].

3. The monadic evaluator from the preceding example can also be employed to demonstrate use of the monad laws (M1)–(M3). We modify it slightly so that it performs a multiplication, rather than division operation:

   ```
   data Term        =  Con Int | Mult Term Term
   eval             :: Term -> M Int
   eval (Con a)     =  return a
   eval (Mult t u)  =  eval t >>= \a ->
                         eval u >>= \b -> return a * b
   ```

   and use the monad laws to show that multiplication as performed by the evaluator is associative; that is,

   $$\text{eval Mult } t \text{ (Mult } u\ v) = \text{ eval Mult (Mult } t\ u)\ v.$$

   We begin by expanding the left-hand side using the definition of `eval`:

   $$\begin{aligned} & \text{eval (Mult } t \text{ (Mult } u\ v)) \\ = \quad & \text{eval } t >>= (\backslash a \to \text{eval (Mult } u\ v) >>= (\backslash x \to \text{return } a * x)) \\ = \quad & \text{eval } t >>= (\backslash a \to (\text{eval } u >>= (\backslash b \to \\ & \quad \text{eval } v >>= (\backslash c \to \text{return } b * c))) >>= (\backslash x \to \text{return } a * x)). \end{aligned}$$

   Law (M3) implies that the order of parentheses does not affect the output of the computation:

   $$\begin{aligned} = \quad & \text{eval } t >>= (\backslash a \to \text{eval } u >>= (\backslash b \to \\ & \quad \text{eval } v >>= (\backslash c \to \text{return } b * c >>= (\backslash x \to \text{return } a * x)))), \end{aligned}$$

   and by (M1), we have the return of the evaluator on input `Add t (Add u v)`:

   $$\begin{aligned} = \quad & \text{eval } t >>= (\backslash a \to \text{eval } u >>= (\backslash b \to \\ & \quad \text{eval } v >>= (\backslash c \to \text{return } a * (b * c)))). \end{aligned}$$

Similarly, for the right-hand term we have:

$$\begin{aligned}
& \text{eval (Mult (Mult } t\ u)\ v) \\
= \ & \text{eval Mult } t\ u >>= (\backslash x \rightarrow \text{eval } v >>= (\backslash c \rightarrow \text{return } x * c)) \\
= \ & (\text{eval } t >>= (\backslash a \rightarrow \text{eval } u >>= (\backslash b \rightarrow \\
& \quad \text{return } a * b))) >>= (\backslash x \rightarrow \text{eval } v >>= (\backslash c \rightarrow \text{return } x * c)) \\
= \ & \text{eval } t >>= (\backslash a \rightarrow \text{eval } u >>= (\backslash b \rightarrow \\
& \quad \text{return } a * b >>= (\backslash x \rightarrow \text{eval } v >>= (\backslash c \rightarrow \text{return } x * c)))) \\
= \ & \text{eval } t >>= (\backslash a \rightarrow \text{eval } u >>= (\backslash b \rightarrow \\
& \quad \text{eval } v >>= (\backslash c \rightarrow \text{return } (a * b) * c))).
\end{aligned}$$

Thus, associativity of integer multiplication implies associativity of the evaluator multiplication.

4. Lists were previously introduced in the discussion on the `Functor` type class, where we mentioned that `[a]` denotes a list of elements of type `a`. Some additional pertinent syntax includes the expression of a list in the form `x:[xs]`, where `x` is the first element of the list (called the 'head'), and `xs` is the remainder of the list (called the 'tail'). For example, `[1,2,3] = 1:[2,3] = 1:2:3:[]`. This notation is tailored to facilitate recursive list definitions and computations. As well, the operation `(++) :: [a] -> [a] -> [a]` is defined as the concatenation of the two input lists.

   With this, we claim that the list type constructor, which we denote as L, forms a monad when combined with the following maps:

```
return      ::  a -> [a]
return a    =   [a]
(>>=)       ::  [a] -> (a -> [b]) -> [b]
[] >>= f    =   []
x:xs >>= f  =   f x ++ (xs >>= f)
```

   Here, **return** maps an element `x` of type `a` to the list that contains the (single) element `x`.

   Note that the instance definition of `(>>=)` for this proposed monad in terms of concatenation leads us to assume that `f x` is finite. With this definition, the computational interpretation of the expression `x:xs >>= f` begins by passing the first element of the list `x:xs` (namely, `x`) to the function `f:: a -> [b]`, which outputs a list of elements of type `b`. This list `f x` is to be concatenated with the list `xs >>= f`, which must be recursively simplified according to the previous description until the final concatenation of some list with the empty list occurs (that is, once `[] >>= f` = `[]` is computed in the recursion).

Let us prove that lists do in fact satisfy (M1)–(M3). For (M1), let `x :: a` and `f :: a -> [b]`. Then,

```
return x >>= f  =  [x] >>= f              defn of return for lists
                =  x:[] >>= f             defn of (:)
                =  f x ++ ([] >>= f)      defn of (>>=) for lists
                =  f x ++ []              defn of (>>=) for lists
                =  f x,                   defn of (++)
```

as required. For (M2), we let `m` = `x:xs` be a list with $n+1$ elements and induct over the length of `m`. Taking `m` to be the empty list as the base case, we have

```
[] >>= return = [].
```

Now, assume that (M2) is true for a list `xs` of length n; that is, assume `xs >>= return = xs`. Then,

```
x:xs >>= return  =  (return x) ++ (xs >>= return)
                 =  [x] ++ xs
                 =  x:xs.
```

Thus, by induction (M2) holds for lists.

Before proving (M3), we first prove the following small proposition, which states that `(>>=)` is right distributive over list concatenation `(++)`.

Proposition 3.1.1. *Given a finite list* `l :: [a]`, *a list* `j :: [a]` *and a function* `f :: a -> [b]`, *we have*

```
(l ++ j) >>= f = (l >>= f) ++ (j >>= f).
```

Proof. To show that the proposition holds for `l = x:xs` of arbitrary, finite length, we induct over the length of `l` and let `j` be arbitrary. In the base case, we have

```
([] ++ j) >>= f  =  j >>= f
                 =  [] ++ (j >>= f)
                 =  ([] >>= f) ++ (j >>= f),
```

as desired. Next, assume as the induction hypothesis that

```
(xs ++ j) >>= f = (xs >>= f) ++ (j >>= f),
```

where `xs` is a list of length n. Then for `l` = `x:xs`, we have

```
(x:xs ++ j) >>= f  =  f x ++ ((xs ++ j) >>= f)
                   =  f x ++ (xs >>= f) ++ (j >>= f)
                   =  (x:xs >>= f) ++ (j >>= f),
```

by the associativity of `(++)`. Thus, the proposition holds by mathematical induction. ■

Finally, to prove (M3), let `g :: b -> [c]`, and `m` be as in (M2) above. We have

```
     (x:xs >>= f) >>= g
  =  (f x ++ (xs >>= f)) >>= g
  =  (f x >>= g) ++ ((xs >>= f) >>= g)            proposition
  =  (f x >>= g) ++ (xs >>= (\a -> (f a >>= g)))
  =  x:xs >>= (\a -> (f a >>= g)),        defn of (>>=) for lists
```

as desired. We have thus shown that (`L`,**`return`**,`(>>=)`) is a monad. This proof is also a nice demonstration of the ease of synatactic manipulation and equational reasoning offered by purely functional languages!

Thus far, each monad we have considered has corresponded to a computation type, so it is natural to consider if the same is true for lists. Indeed, we can think of lists as a means to model relations, where a computation of type `[a]` offers one choice for each element in the list. Thus, modifying a function `f :: a -> b` to its monadic equivalent `f :: a -> [b]`, entails offering a choice of results for each argument [17].

Chapter 4

Equivalence

Thus far, we have examined: i) Moggi's incorporation of category theory into programming semantics via his category-theoretic semantics of computations, and ii) Wadler's implementation of Moggi's theoretical work in structuring the particular functional language Haskell. While the theory of i) was motivated using Kleisli triples, Theorem 2.3.1 gave the explicit connection between Kleisli triples and the "purer" category-theoretic concept of monads. We are now ready to give a similarly explicit connection between ii), which is centred around the maps `return` and `>>=`, and the standard formulation of monads in category theory.

4.1 Preliminary Connections

The first connection is easy to identify. Recall the type signature of `return` for any (Haskell) monad:

```
return ::  Monad m => a -> m a,
```

where `a` is any object in **Hask**. This is exactly analogous to the domain and codomain of the map η that must be defined for any (category-theoretic) monad M in a category $\mathcal{C}$:

$$\eta_A : A \to MA,$$

where A is an object in $\mathcal{C}$. We denote this analogy by `return`$\sim \eta$.

Unfortunately, no such obvious comparison between `>>=` and μ exists. There is, however, a Haskell function called `join` that is defined in the `Monad` module. It has the following type declaration:

```
join ::  Monad m => m (m a) -> m a,
```

which is indeed analogous to the domain and codomain of μ :

$$\mu_A : M^2A \to MA.$$

That is, `join`$\sim \mu$.

The action of `join` is to remove one layer of nesting of a monadic value. As is the case for `return` and `>>=`, the definition of `join` is particular to the monad under consideration. In the case of `Maybe`, for example, the definition of `join` is given by:

```
join                    ::   Maybe (Maybe a) -> Maybe a
join Nothing            =    Nothing
join (Just Nothing)     =    Nothing
join (Just (Just x))    =    Just x
```

Since `join` (along with `return`) is another direct link between Haskell functions and category-theoretic monad maps, if we can display an explicit connection between `join` and `>>=`, then we will have the desired translation between the Haskell and category-theoretic formulations of monads!

Continuing with the `Maybe` example, recall that one equation in the original instance declaration of the `Maybe` monad is:

```
Just x0 >>= f = f x0
```

Consider the substitutions $x_0 = \text{Just } x$ and $f = \text{id}$ in the above. We have:

```
Just (Just x) >>= id  =  id (Just x)
                      =  Just x
                      =  join (Just (Just x)).
```

This gives a first relationship between `>>=` and `join` in the case of the `Maybe` monad. In fact, this relationship is the `Maybe`-specific case of the following general relation in Haskell. Given a monad `m` and a monadic element `m (m x)`, we have:

```
join m (m x) = (m (m x)) >>= id.
```

That is, removing one monadic layer using the `join` operation is exactly the same as extracting the element `m x` from the (outer level of the)monad and leaving it unchanged. Therefore, for any Haskell monad `m` defined using the `>>=` function, we can obtain the definition for `join` corresponding to `m` via the above equality.

To establish an equivalence between these functions, it remains to provide a definition for `>>=` in terms of `join`. There is a small and obvious catch, however. The function `>>=` accepts two arguments, one of which is a *function*, whereas `join` is a function from a double-layered monadic *object* to a monadic *object*. Also note that every (category-theoretic) monadic map

μ is a natural transformation, which means it is defined on objects and returns a function. Clearly, if `join` is to be used to define a monad, we must integrate functions into its defined behaviour.

To do this, we recall the method `fmap` that was introduced in Section 1.5.1 in conjunction with the **Functor** type class; specifically `fmap` is a function that must be defined for any instance of **Functor**, and it provides a means for functions defined outside a set of functor-type elements to act on functor-type elements. Let us assume a (presently unjustified) consistency between Haskell's class names **Functor** and **Monad**, and the category-theoretic definitions of those two terms: namely, that any instance of **Monad** must also be an instance of **Functor**. Then,

```
fmap ::  a -> b -> f a -> f b
```

can be defined for both functors and monads `f`.

Using `fmap` with this additional context, we have the following Haskell relation between `>>=` and `join`:

```
x >>= f = join (fmap f x).
```

That is, for a particular monad `m`, if we are given a definition for `join` and `fmap`, then we can obtain the definition for `>>=`.

For example, assume the three equations constituting the `Maybe` definition of `join` given above. In addition, assume the following `Maybe`-definition for `fmap`:

```
fmap                ::   a -> b -> Maybe a -> Maybe b
fmap _ Nothing      =    Nothing
fmap f (Just x)     =    Just f x.
```

We show that these five equations imply those two involving `>>=` given in the original instance declaration of the `Maybe` monad as follows. Let `f :: a -> Maybe b`. To show the first equation:

```
Nothing >>= f  =  join (fmap f Nothing)
               =  join Nothing
               =  Nothing.
```

For the second equation, since

```
Just x >>= f = join (fmap f (Just x)),
```

we consider the following two cases:

Case 1: `f x = Nothing`

We have,

$$\begin{aligned}\texttt{join (fmap f (Just x))} &= \texttt{join (Just (Nothing))}\\ &= \texttt{Nothing}\\ &= \texttt{f x}.\end{aligned}$$

Case 2: `f x = Just y`

In this case,

$$\begin{aligned}\texttt{join (fmap f (Just x))} &= \texttt{join (Just (Just y))}\\ &= \texttt{Just y}\\ &= \texttt{f x}.\end{aligned}$$

Therefore, `join (fmap f (Just x))` = `f x`, and the definitions for `fmap` and `join` are sufficient to provide an instance declaration for `Maybe`.

4.1.1 Comparative Motivations for >>=, join, and _*

We have now given the equivalence between `>>=` and `join`. Using this equivalence along with Theorem 2.3.1, we can easily see the relationship between `>>=` and $\_^*$. Let `m` be a monad, `x` be a monadic element of type `x :: m a`, and `f` be a function of type `f :: a -> m b`. Then, as we have seen, `x >>= f` = `join (fmap f x)`. Recall that a category-theoretic functor is defined on both objects and morphisms, and that Haskell's `fmap` can be thought of as corresponding to the part of a functor that is defined on morphisms. Therefore, if we think of `m` as a monad M in the category-theoretic sense, we have an analogy between `fmap f x` and $(M(f))(x)$. Furthermore, given the type signatures of `x` and `f`, we see that `fmap f x` has return type `m (m b)`. Thus, the map `join` is analogous to μ_B and we have:

$$\begin{aligned}\texttt{join (fmap f x)} &\sim \mu_B(Mf(x))\\ &= (\mu_B \circ Mf)(x)\\ &= f^*(x),\end{aligned}$$

where the last equality follows from Theorem 2.3.1. We thus have the following relationship between `>>=` and $\_^*$:

$$\texttt{x >>= f} \sim f^*(x).$$

Now that we have explicitly established equivalancy relationships between `>>=`, `join` (or μ), and $\_^*$, the question remains as to why these different operations were chosen in their respective contexts. We have already seen that Kleisli triples, with $\_^*$ as a means of composing operations, were nicely justifiable from a computational standpoint. In fact, the aforementioned **let**-constructor (Section 2.4) corresponds directly to composition in a given Kleisli category [10].

The **let**-constructor similarly provides a motivation for the characterization of `>>=`. An expression of the form $x >>= \backslash a \to y$, where x and y are expressions, is computationally described by the sequence of actions: perform computation `x`, bind `a` to the resulting value, and then perform computation `y`. This is analogous to the computational interpretation of the expression **let** $a = x$ **in** y [17]. The `>>=` function thus provides a means to implement impure/imperative sequential-style reasoning in a pure functional language.

Finally, understanding the connections between these functions with μ is useful, because the category-theoretic formulation of monads is given solely in terms of functors and natural transformations. This formulation is easier to reason about, and integrate into the established abstract setting of category theory.

4.2 Monad Laws

To complete our exploration of the equivalence between monads in Haskell and category theory, it remains to justify the use of `fmap` as a component in the characterization of `>>=` in terms of `join`. To do this, we propose a new set of monad laws as well as a definition for `fmap` in terms of `>>=`. We then prove the equivalence between the new laws and the original laws (M1)–(M3).

4.2.1 Alternate Laws

We offer and discuss the following new set of monad laws involving the polymorphic Haskell functions `fmap`, `id`, `return`, and `join`, and arbitrary functions `f :: a -> b`, `g :: b -> c` [17]:

```
(L1)        fmap id = id
(L2)        fmap (f . g) = fmap f . fmap g
(L3)        fmap f . return = return . f
(L4)        fmap f . join = join . fmap (fmap f)
(L5)        join . fmap join = join . join
```

(L6) `join . fmap return = id`
(L7) `join . return = id`

The astute reader may immediately notice similarities between these laws and the mathematical relationships pertinent to category-theoretic monads. We have already introduced the functor laws (L1) and (L2) as being analogous to the defining equations for a category-theoretic functor in Section 1.5.1. To make the analogy explicit, let $F : \mathcal{C} \to \mathcal{C}$ be a functor, $M : \mathcal{C} \to \mathcal{C}$ be a monad, $x \in O_{\mathcal{C}}$, and $f : a \to b, g : b \to c \in A_{\mathcal{C}}$. We have:

$$\texttt{fmap id} = \texttt{id} \quad \sim \quad F(\mathrm{id}_x) = \mathrm{id}_{Fx}$$
$$\texttt{fmap (f . g)} = \texttt{fmap f . fmap g} \quad \sim \quad F(f \circ g) = F(f) \circ F(g).$$

It is similarly easy to see that (L3) and (L4) correspond to the naturality conditions for $\eta : I \to M$ and $\mu : M^2 \to M$, respectively; that is, for any objects $a, b \in O_{\mathcal{C}}$, and any map between those objects $f : a \to b \in A_{\mathcal{C}}$, we have:

$$\texttt{fmap f . return} = \texttt{return . f} \quad \sim \quad Mf \circ \eta_a = \eta_b$$
$$\texttt{fmap f . join} = \texttt{join . fmap (fmap f)} \quad \sim \quad (Mf) \circ \mu_a = \mu_b \circ M^2 f$$

We expect the last three laws to correspond to the category-theoretic monad laws given by the standard commutative diagrams for monads. However, on examining (L5), it looks as though we should have:

$$\texttt{join . fmap join} = \texttt{join . join} \quad \overset{?}{\sim} \quad \mu \circ M\mu = \mu \circ \mu.$$

Not only does the right-hand side of the similarity fail to correspond to the first category-theoretic monad law, but the composition $\mu \circ \mu$ is not even defined!

The apparent inconsistency is actually caused by the polymorphism of `join`. The function `join` can accept any object of the form `m (m a)`, including the case when `a` itself is a monadic type (for example, if `a` = `m b`). Thus, while it is important to mathematically distinguish between the functions μ and μM, Haskell's polymorphic type system allows the name `join` to stand for more than one function, each being technically distingusihed by the domain it is defined on.

An exemplary demonstration of (L5) using the list monad `L` helps to clarify the issue. In Section 1.5.1 we noted that, in the case of lists, `fmap` is equivalent to `map`. In Section 3.1 we gave the definition of `>>=` for lists, which we can now use (informally) to recover `join` as follows. Let `l0`, `l1`, `l2`,...,`ln` be lists of the same type, and form the list of lists `[l0, l1,..., ln]`$\in$`L`^{2}. Then,

```
join [l0, l1,..., ln]  =  [l0, l1,..., ln] >>= id
                       =  id l0 ++ [l1,l2,..., ln] >>= id
                       ⋮
                       =  l0 ++ (l1 ++ (l2 ++ (··· ++ ln))···).
```

A Haskell expression involving the concatenation of more than two lists is typically written in terms of the operation `concat :: [[a]] -> [a]`. The last equation in the above derivation is equivalent to `l0 ++ (l1 ++ (··· ++ ln))···) = concat(l0,l2,...,ln)`. Thus, the instance of `join` defined for lists is equivalent to `concat`.

Giving the instance (L5) in terms of `fmap` and `join` for lists, we have:

```
concat . map concat = concat . concat.
```

Recall that the type signature of `map` is `map :: (a -> b) -> [a] -> [b]`. Taken with the type signature of `concat`, this means that the type signature of the inner function on the left-hand side of the preceding equation is `map concat :: [[[a]]] -> [[a]]`. The return type of this function matches the input type of `concat`, so that the type signature of the composed function is `concat . map concat :: [[[a]]] -> [a]`.

For example, the action of the left-hand side function can be demonstrated on the following "list of lists of lists":

```
   concat . map concat [[[1,2],[1,2],[3,4,5]],[[1,3],[]]]
=  concat [concat [[1,2],[1,2],[3,4,5]],concat [[1,3],[]]]
=  concat [[1,2,1,2,3,4,5],[1,3]]
=  [1,2,1,2,3,4,5,1,3]
```

The apparent problem with the similarity for (L5) arose with respect to the composition `join . join` $\not\sim \mu \circ \mu M$, but we can see in the list example that the "M"—or L in the case of lists—is accounted for in the polymorphism of `concat`. Explicitly, a list of lists `[[a]]`$\in$ L^2 is equivalently a list `[b]`$\in$ L, where `b` has type `[a]`. As such, the compiler would recognize an element of `[[[a]]]` as simply a list and allow the polymorphic function `concat` to accept such an element as an input with the correct type. Thus, the map `concat . concat` also has type `concat . concat :: [[[a]]] -> [a]` and acts on the previously given member of L^3 as follows:

```
   concat . concat [[[1,2],[1,2],[3,4,5]],[[1,3],[]]]
=  concat [[1,2],[1,2],[3,4,5],[1,3],[]]
=  [1,2,1,2,3,4,5,1,3],
```

which matches the evaluation of `concat . map concat` on the same list.

This reasoning regarding polymorphism for `concat` generalizes to `join` and similar reasoning applies to `return`. Thus, the analogies between (L5)–(L7) and the category-theoretic monad laws do, in fact, hold:

$$\begin{array}{lcl} \texttt{join . fmap join} = \texttt{join . join} & \sim & \mu \circ M\mu = \mu \circ \mu M \\ \texttt{join . fmap return} = \texttt{id} & \sim & \mu \circ M\eta = \mathrm{id} \\ \texttt{join . return} = \texttt{id} & \sim & \mu \circ \eta M = \mathrm{id}. \end{array}$$

4.2.2 Law Equivalence

We now show the equivalence of laws (L1)–(L7) with laws (M1)–(M3), given in Section 3.0.2. First, we assume that laws (L1)–(L7) hold, as well as the previously given definition of `>>=` in terms of `join`. To prove the laws imply (M1), we have:

```
return x >>= f  =  join (fmap f (return x))
                =  join (return (f x))          (L3)
                =  (join . return)(f x)
                =  id (f x)                     (L7)
                =  f x.
```

For (M2), we have:

```
n >>= return  =  join (fmap return n)
              =  (join . fmap return) n
              =  id n                      (L6)
              =  n.
```

Finally, we prove (M3), recalling that any single-variable function h can equivalently be expressed with Haskell syntax as either `h` or `\x-> h x`. We have:

```
   (n >>= f) >>= g
=  (join (fmap f n)) >>= g
=  join (fmap g (join (fmap f n))
=  (join . fmap g . join)(fmap f n)
=  (join . join . fmap (fmap g))(fmap f n)           (L4)
=  (join . join . fmap (fmap g) . fmap f) n
=  (join . join . fmap ((fmap g) . f)) n             (L2)
=  (join . join . fmap (\x -> fmap g (f x))) n
=  (join . fmap join . fmap (\x -> fmap g (f x))) n  (L5)
=  (join . fmap (join . (\x -> fmap g (f x)))) n     (L2)
=  (join . fmap (\x -> join (fmap g (f x)))) n
= (join . fmap (\x -> f x >>= g)) n (def of >>= in terms of join)
= (join (fmap (\x -> f x >>= g) n)
= n >>= (\x -> f x >>= g).
```

Thus, monad laws (L1)–(L7) imply monad laws (M1)–(M3).

For the converse, we assume laws (M1)–(M3), along with the definition for `join` in terms of `>>=`. In addition, we assume the following definition for `fmap`, given a function `f :: a -> b` and monadic element `n :: m a`:

```
fmap f n = n >>= return . f
```

The proofs of the laws are then as follows:

(L1):

```
fmap id n = n >>= return . id
          = n >>= return
          = n                       (M2)
```

(L2):

```
    fmap (f . g) n
  = n >>= return . f . g
  = n >>= (\x -> return f (g x))
  = n >>= (\x -> return . f g x)
  = n >>= (\x -> (return g x >>= return . f))    (M1)
  = n >>= (\x -> (return . g x >>= return . f))
  = (n >>= return . g) >>= return . f            (M3)
  = (fmap g n) >>= return . f
  = fmap (fmap g n)
  = (fmap . fmap g) n
```

(L3):

```
fmap f . return n = fmap f return n
                  = return n >>= return . f
                  = return . f n              (M1)
```

(L4):

```
     fmap . join n
  =  fmap f (join n)
  =  join n >>= return . f
  =  (n >>= id) >>= return . f
  =  n >>= (\x -> (id x >>= return . f))           (M3)
  =  n >>= (\x -> (x >>= return . f))
  =  n >>= (\x -> fmap f x)
  =  n >>= (\x -> id (fmap f x))
  =  n >>= (\x -> (return (fmap f x) >>= id))      (M1)
  =  (n >>= return . fmap f) >>= id
  =  join (n >>= return . fmap f)
  =  join (fmap (fmap f) n)
  =  join . fmap (fmap f) n
```

(L5):

```
     (join . fmap join) n
  =  join (fmap join n)
  =  join (n >>= return . join)
  =  (n >>= return . join) >>= id
  =  n >>= (\x -> (return . join x >>= id))       (M3)
  =  n >>= (\x -> (return (join x) >>= id))
  =  n >>= (\x -> id (join x))                    (M1)
  =  n >>= (\x -> join x)
  =  n >>= (\x -> (x >>= id))
  =  n >>= (\x -> (id x >>= id))
  =  (n >>= id) >>= id
  =  join (n >>= id)
  =  join (join n)
  =  join . join n
```

(L6):

```
     (join . fmap return) n
  =  join (fmap return n)
  =  join (n >>= return . return)
  =  (n >>= return . return)
  =  (n >>= return . return) >>= id
  =  n >>= (\x -> (return . return x >>= id))     (M3)
  =  n >>= (\x -> (return (return x) >>= id))
  =  n >>= (\x -> id (return x))                  (M1)
  =  n >>= return
  =  id n                                         (M2)
```

(L7):

```
(join . return) n  =  join (return n)
                   =  return n >>= id
                   =  id n              (M1)
```

Therefore, laws (M1)–(M3) imply laws (L1)–(L7) and the two sets of monad laws are equivalent.

Previously, we assumed a consistency between the Haskell class names **Functor** and Monad with part of the category-theoretic definition of a monad; namely, that every monad is a functor. Since we have shown the monad laws (M1)–(M3) to be equivalent to laws (L1)–(L7), with the latter set including the functor laws (L1) and (L2), we have also shown that our assumption is justified. In fact, we can use laws (M1)–(M3) to prove the definition of >>= in terms of join as follows:

```
n >>= f  =  n >>= (\x -> id (f x))
         =  n >>= (\x -> (return (f x) >>= id))
         =  n >>= (\x -> (return . f x >>= id))
         =  (n >>= return . f) >>= id
         =  join (n >>= return . f)
         =  join (fmap f n)
```

Therefore—at last!—we can provide an alternate class declaration for **Monad** that looks very similar to the category-theoretic formulation of a monad, with return $\sim \eta$ and join $\sim \mu$:

```
class Functor m => Monad m where
  return ::  a -> m a
  join ::  m (m a) -> m a.
```

4.3 Concluding Connections

4.3.1 List in Translation

We conclude by using our work with equivalences to tie together our knowledge of monads in Haskell with those in category theory. In Section

3.1, Example 4, we showed that (M1)–(M3) hold for the list type constructor. In examining the alternate law formulations in Section 4.2.1, we saw that for the list monad `L`, `fmap` is given by `map` and `join` by `concat`. Notably, the maps of the latter formulation seemed easier to understand and manipulate than the recursive definition of `>>=` for lists.

If we begin to translate to category-theoretic terms, we see that `L` is a functor which, given a Haskell type `a`, constructs the set of all lists of type `a` (denoted `[a]`). In other words, given a set $X \in$ **Hask**, LX is the set of all possible lists that can be formed using elements in X, $\mathtt{L}^2X$ is the set of all lists of lists, and so on. Further, we recall that `return` for the List monad is given by `return x = [x]`, where $x \in X$.

Thus, given a set $X \in \mathtt{L}$, we define η_X by sending $x \mapsto [x]$ and μ_X by the concatenation of lists in $\mathtt{L}X$. Note that the concatenation of lists does not entail 'discarding' repeated elements (as with, say, a union operation) or 'cancelling' elements in accord with some given relations (as for, say, invertible elements). Comparing this to Example 3, Section 1.3.2, we see that the List monad is none other than the free-forgetful semigroup monad!

4.3.2 Distributive Laws

In many applications of monads in Haskell to structuring functional languages, the desired monad is most easily conceived as a combination of simpler monads. For example, we considered using monads to facilitate error handling (the `Maybe` monad) and to count the number of operations performed during a calculation (state updates). It is easy to see the desirability of being able to combine these monads so that if an error occurred during a calculation in which a counter was running, then the error would be handled properly.

In general, it is not the case that when endofunctors comprising monads are composed, the result is another monad. However, there is a category-theoretic concept that formalizes the cases when this does happen; the concept being that of the distributive laws for monads. A distributive law for monads is a means by which the functorial composition of two monads can be made into a more complex, combined monad. The idea is a similar to viewing the ordinary distributive law of multiplication over addition as a means by which abelian groups and monoids are combined into the more complex structure of rings.

Let $M_1 = (M_1, \eta_1, \mu_1)$ and $M_2 = (M_2, \eta_2, \mu_2)$ be monads on a category $\mathcal{C}$. A *distributive law* of M_1 over M_2 is a natural transformation $\lambda : M_1 \circ M_2 \longrightarrow M_2 \circ M_1$ for which the following diagrams commute [1]:

$$\begin{tikzcd}
M_1 \arrow[r, equal] \arrow[d, "M_1\eta_2"'] & M_1 \arrow[d, "\eta_2 M_1"] \\
M_1M_2 \arrow[r, "\lambda"'] & M_2M_1
\end{tikzcd}
\qquad
\begin{tikzcd}
M_2 \arrow[r, equal] \arrow[d, "\eta_1 M_2"'] & M_2 \arrow[d, "M_2\eta_1"] \\
M_1M_2 \arrow[r, "\lambda"'] & M_2M_1
\end{tikzcd}$$

$$\begin{tikzcd}
M_1M_2^2 \arrow[r, "\lambda M_2"] \arrow[d, "M_1\mu_2"'] & M_2M_1M_2 \arrow[r, "M_2\lambda"] & M_2^2M_1 \arrow[d, "\mu_2 M_1"] \\
M_1M_2 \arrow[rr, "\lambda"'] & & M_2M_1
\end{tikzcd}$$

$$\begin{tikzcd}
M_1^2M_2 \arrow[r, "M_1\lambda"] \arrow[d, "\mu_1 M_2"'] & M_1M_2M_1 \arrow[r, "\lambda M_1"] & M_2M_1^2 \arrow[d, "M_2\mu_1"] \\
M_1M_2 \arrow[rr, "\lambda"'] & & M_2M_1
\end{tikzcd}$$

A distributive law λ of M_1 over M_2 gives rise to the composite or *combined monad defined by* λ, given by

$$M_2M_1 = (M_2M_1, \eta^{M_2M_1}, \mu^{M_2M_1}),$$

where the composite maps $\eta^{M_2M_1}$ and $\mu^{M_2M_1}$ are defined by:

$$\begin{tikzcd}
M_1 \arrow[r, "\eta^{M_2M_1}"] & M_2M_1 \arrow[d, equal] \\
I_C \arrow[u, "\eta^{M_1}"] \arrow[r, "\eta^{M_2M_1}"] \arrow[d, "\eta^{M_2}"'] & M_2M_1 \arrow[d, equal] \\
M_2 \arrow[r, "M_2\eta^{M_1}"] & M_2M_1
\end{tikzcd}
\qquad
\begin{tikzcd}
 & M_2^2M_1 \arrow[r, "\mu^{M_2M_1}"] & M_2M_1 \arrow[d, equal] \\
M_2M_1M_2M_1 \arrow[r, "M_2\lambda M_1"] & M_2^2M_1^2 \arrow[u, "M_2^2\mu^{M_1}"] \arrow[r, "\mu^{M_2M_1}"] \arrow[d, "\mu^{M_2}M_1^2"'] & M_2M_1 \arrow[d, equal] \\
 & M_2M_1^2 \arrow[r, "M_2\mu^{M_1}"] & M_2M_1
\end{tikzcd}$$

The natural question for us to ask is: Can we implement this theoretical definition to combine certain monads in Haskell? The answer is "Yes!" and we show how the distributive laws can be used to define a monad `ML`, where `M` is an arbitrary monad and `L` is the list monad.

We begin by using the same essential idea that we explored in Section 4.2.1; namely, we draw analogies between the category-theoretic distributive laws and Haskell code. The difference here, is that while in Section 4.2.1 we had sets of laws from category theory and Haskell, and we justified an analogy between them, in this case we use a set of category-theoretic laws to motivate, via analogy, a set of relations between Haskell functions.

At this stage, we feel confident enough to blend notations for convenience. Thus, we let λ be a Haskell function with type signature λ `:: L (M a) -> M (L a)`, as well as $\eta^{\mathtt{M}}$ `:: a -> M a`, $\mu^{\mathtt{M}}$ `:: M (M a) -> M a` and $\mathrm{m}^{\mathtt{M}}$ `:: (a -> b) -> M a -> M b` be the monad maps for `M`, and $\eta^{\mathtt{L}}$, $\mu^{\mathtt{L}}$ and $\mathrm{m}^{\mathtt{L}}$ denote the monad maps for `L`. Then the first analogous Haskell relation corresponds to the naturality of λ :

$$\text{(D0)} \quad \lambda\ .\ \mathrm{m}^{\mathtt{L}}\ (\ \mathrm{m}^{\mathtt{M}}\ \mathtt{f}\ ) \quad = \quad \mathrm{m}^{\mathtt{M}}\ (\ \mathrm{m}^{\mathtt{L}}\ \mathtt{f}\ )\ .\ \lambda\ ,$$

where `f` is an arbitrary function with type signature `f :: a -> b`. By letting M_1 and M_2 correspond to `L` and `M`, respectively, in the diagrams above, we can infer an analogous set of Haskell distributive laws. For example, the equation corresponding to the second diagram given in the definition of a distributive law is $\lambda \circ \eta_1 M_2 = M_2\eta_1$. To come up with a corresponding Haskell law, we: (i) let the `fmap` methods defined for the instances `L` and `M`—namely, $\mathrm{m}^{\mathtt{L}}$ and $\mathrm{m}^{\mathtt{M}}$, respectively—replace the functor maps M_1 and M_2, respectively; (ii) let the `return` methods for `L` and `M`—namely, $\eta^{\mathtt{L}}$ and $\eta^{\mathtt{M}}$, respectively—replace the monad units η_1 and η_2, respectively; (iii) account for Haskell's polymorphic type system (see Section 4.2.1 for a detailed introduction of the manifestation of polymorphism in this sort of context).

The Haskell function $\mathrm{m}^{\mathtt{M}}$ can be thought of as accepting two arguments: a function of type `a -> b` and a monadic element of type `M a`. Thus, we convert the expression $M_2\eta_1$ to the corresponding Haskell function $\mathrm{m}^{\mathtt{M}}\ \eta^{\mathtt{L}}$, where the type signatures of $\mathrm{m}^{\mathtt{M}}$ `:: (a -> b) -> M a -> M b` and $\eta^{\mathtt{L}}$ `:: a -> L a` imply that $\mathrm{m}^{\mathtt{M}}\ \eta^{\mathtt{L}}$ accepts an argument of type `M a` and has return type `M (L a)`. For the left-hand side of the distributive law equation under consideration, we convert $\lambda \circ \eta_1 M_2$ to $\lambda\ .\ \eta^{\mathtt{L}}$. In the category-theoretic case, we have $\lambda \circ \eta_1 M_2$ as a map from M_2 to $M_2 M_1$. In the Haskell version, the polymorphism of $\eta^{\mathtt{L}}$ means that we do not need to explicitly include the inner monad map M_2 in the law's formulation; rather, we just note that $\eta^{\mathtt{L}}$ can accept monadic-typed arguments. Moreover, the type signature of λ `:: L (M a) -> M (L a)` guarantees that we must give $\eta^{\mathtt{L}}$ an argument of type `M a`, so that its return type will be `L (M a)` (which corresponds to the input type of λ). Thus, the Haskell version of this distributive law equation is $\lambda\ .\ \eta^{\mathtt{L}} = \mathrm{m}^{\mathtt{M}}\ \eta^{\mathtt{L}}$.

We can similarly translate the remaining distributive law equations. In summary, the category-theoretic distributive laws give rise to the following relations in Haskell:

$$\begin{array}{lllll}
\text{(D1)} & \lambda \;.\; \eta^{\mathtt{L}} & = & \mathrm{m}^{\mathtt{M}}\;\; \eta^{\mathtt{L}} \\
\text{(D2)} & \lambda \;.\; \mathrm{m}^{\mathtt{L}}\;\; \eta^{\mathtt{M}} & = & \eta^{\mathtt{M}} \\
\text{(D3)} & \lambda \;.\; \mathrm{m}^{\mathtt{L}}\;\; \mu^{\mathtt{M}} & = & \mu^{\mathtt{M}} \;.\; \mathrm{m}^{\mathtt{M}}\;\; \lambda \;.\; \lambda \\
\text{(D4)} & \lambda \;.\; \mu^{\mathtt{L}} & = & \mathrm{m}^{\mathtt{M}}\;\; \mu^{\mathtt{L}} \;.\; \lambda \;.\; \mathrm{m}^{\mathtt{L}}\;\; \lambda
\end{array}$$

It is important to note that to assume that these analogous Haskell laws hold, we must first assume that the Haskell function λ exists. What makes the case of `L` over an arbitrary `M` special is that such a distributive law function λ does indeed exist. Showing this involves working with a so-called Haskell "fold" function, and defining a sort of Cartesian product for lists. Interested readers are referred to [7] for details.

By continuing with our stated assumptions, we propose that the following 3-tuple defines a combined monad defined by λ.

Proposition 4.3.1. *The triple defined by:*

$$\begin{array}{lll}
unit^{\mathtt{ML}} & = & \eta^{\mathtt{M}} \;.\; \eta^{\mathtt{L}} \\
fmap^{\mathtt{ML}}\; \mathtt{f} & = & m^{\mathtt{M}}\; (\; m^{\mathtt{L}}\; \mathtt{f}\;) \\
join^{\mathtt{ML}} & = & m^{\mathtt{M}}\;\; \mu^{\mathtt{L}} \;.\; \mu^{\mathtt{M}}\;\; m^{\mathtt{M}}\;\; \lambda
\end{array}$$

is a combined monad, `ML`*, in Haskell.*

Proof. To prove the 3-tuple (`ML`, $\mathrm{eta}^{\mathtt{ML}}$, $\mathrm{join}^{\mathtt{ML}}$) is a monad, we must show that the maps $\mathrm{unit}^{\mathtt{ML}}$, $\mathrm{fmap}^{\mathtt{ML}}$, and $\mathrm{join}^{\mathtt{ML}}$ satisfy (L1)–(L7), given that this same set of laws holds for each of `M` and `L`, and that (D0)–(D4) also hold. We have:

(L1):

$$\begin{array}{rcl}
\mathrm{fmap}^{\mathtt{ML}}\; (\; \mathrm{id}\;) & = & \mathrm{m}^{\mathtt{M}}\; (\; \mathrm{m}^{\mathtt{L}}\; \mathrm{id}\;) \\
& = & \mathrm{m}^{\mathtt{M}}\; (\; \mathrm{id}\;) \\
& = & \mathrm{id}
\end{array}$$

(L2):

$$\begin{array}{rcl}
\mathrm{fmap}^{\mathtt{ML}}\; (\; \mathtt{f}\;\; .\, \mathrm{g}\;) & = & \mathrm{m}^{\mathtt{M}}\; (\; \mathrm{m}^{\mathtt{L}}\; \mathtt{f}\; .\; \mathrm{g}) \\
& = & \mathrm{m}^{\mathtt{M}}\; (\; \mathrm{m}^{\mathtt{L}}\; \mathtt{f}\; .\; \mathrm{m}^{\mathtt{L}}\; \mathrm{g}\;) \\
& = & \mathrm{m}^{\mathtt{M}}\;\; \mathrm{m}^{\mathtt{L}}\; \mathtt{f}\; .\; \mathrm{m}^{\mathtt{M}}\;\; \mathrm{m}^{\mathtt{L}}\; \mathrm{g} \\
& = & \mathrm{fmap}^{\mathtt{ML}}\; \mathtt{f}\; .\, \mathrm{fmap}^{\mathtt{ML}}\; \mathrm{g}
\end{array}$$

(L3):

$$
\begin{aligned}
\text{fmap}^{\texttt{ML}}\ (\ \texttt{f}\ .\ \text{unit}^{\texttt{ML}}\ ) &= \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ (\ \texttt{f}\ .\ \eta^{\texttt{M}}\ .\ \eta^{\texttt{L}}\ )\ )\\
&= \text{m}^{\texttt{M}}\ (\ \eta^{\texttt{L}}\ .\texttt{f}\ .\ \eta^{\texttt{M}}\ )\\
&= \eta^{\texttt{M}}\ .\ \eta^{\texttt{L}}\ .\texttt{f}\\
&= \text{unit}^{\texttt{ML}}\ .\texttt{f}
\end{aligned}
$$

Note that our particular interest in combining the list monad with an arbitrary monad is highlighted in the remaining laws, where we assume the existence of λ for `L` over `M`.

(L4):

$$
\begin{aligned}
& \text{fmap}^{\texttt{ML}}\ .\ \text{join}^{\texttt{ML}}\\
= \;& \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ .\ \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ \text{m}^{\texttt{M}}\ \lambda\\
= \;& \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ .\ \mu^{\texttt{L}}\ )\ .\ \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ \lambda\\
= \;& \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ .\ \mu^{\texttt{L}}\ )\ )\ .\ \text{m}^{\texttt{M}}\ \lambda\\
= \;& \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ .\ \mu^{\texttt{L}}\ )\ .\ \lambda\ )\\
= \;& \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ (\ \mu^{\texttt{L}}\ .\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ ).\ \lambda\ )\\
= \;& \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ )\ .\ \lambda\ )\\
= \;& \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ )\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ )\ .\ \lambda\ )\\
= \;& \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ )\ .\ \lambda\ )\\
\overset{(D0)}{=} \;& \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ (\ \lambda\ .\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ )\ )\\
= \;& \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ \lambda\ .\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ (\ \text{m}^{\texttt{M}}\ (\ \text{m}^{\texttt{L}}\ \texttt{f}\ )\ )\ )\\
= \;& \text{join}^{\texttt{ML}}\ .\ \text{map}^{\texttt{ML}}\ (\ \text{map}^{\texttt{ML}}\ \texttt{f}\ )
\end{aligned}
$$

(L6):

$$
\begin{aligned}
\text{join}^{\texttt{ML}}\ .\ \text{unit}^{\texttt{ML}} &= \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ .\ \text{m}^{\texttt{M}}\ \lambda\ .\ \eta^{\texttt{M}}\ \eta^{\texttt{L}}\\
&= \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \mu^{\texttt{M}}\ .\ \eta^{\texttt{M}}\ .\ \lambda\ \eta^{\texttt{L}}\\
&= \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \text{id}\ .\ \lambda\ \eta^{\texttt{L}}\\
&\overset{(D1)}{=} \text{m}^{\texttt{M}}\ \mu^{\texttt{L}}\ .\ \text{m}^{\texttt{M}}\ \eta^{\texttt{L}}\\
&= \text{m}^{\texttt{M}}\ (\ \mu^{\texttt{L}}\ .\ \eta^{\texttt{L}}\ )\\
&= \text{m}^{\texttt{M}}\ (\ \text{id}\ )\\
&= \text{id}
\end{aligned}
$$

(L7):

$$\begin{aligned}
& \mathrm{join}^{\mathrm{ML}} \;.\; \mathrm{map}^{\mathrm{ML}}\ \mathrm{unit}^{\mathrm{ML}} \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{L}} (\ \eta^{\mathrm{M}} \;.\; \eta^{\mathrm{L}}\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \eta^{\mathrm{M}} \;.\; \eta^{\mathrm{L}}\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{L}}\ ) \\
\overset{(D2)}{=} \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \eta^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{L}}\ ) \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}}\ \eta^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{L}}\ ) \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{L}}\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{L}}\ \eta^{\mathrm{L}}\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mathrm{id}\ ) \\
= \;& \mathrm{id}
\end{aligned}$$

(L5):

$$\begin{aligned}
& \mathrm{join}^{\mathrm{ML}} \;.\; \mathrm{join}^{\mathrm{ML}} \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}}\ \lambda \\
= \;& \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}}\ ) \;.\; \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mu^{\mathrm{L}}\ ).\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \\
= \;& \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \\
= \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda \;.\; \mu^{\mathrm{L}}\ )\ ) \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \\
= \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}}\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda \;.\; \mu^{\mathrm{L}}\ ) \;.\; \lambda\ ) \\
\overset{(D4)}{=} \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}}\ \lambda\ ) \;.\; \lambda\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda\ ) \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{L}}\ \lambda\ ) \;.\; \lambda\ ) \\
\overset{(D0)}{=} \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \lambda\ )\ \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}}\ ) \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;. \\
& \quad \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}}\ )\ ) \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ )\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
\overset{(D3)}{=} \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mathrm{m}^{\mathrm{L}}\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{L}} (\ \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ )\ ) \;.\; \mathrm{m}^{\mathrm{M}} (\ \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mu^{\mathrm{L}}\ ) \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ ) \\
= \;& \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}} (\ \mathrm{m}^{\mathrm{M}} (\ \mu^{\mathrm{L}} \;.\; \mathrm{m}^{\mathrm{L}}\ \mu^{\mathrm{L}}\ ) \;.\; \lambda \;.\; \mathrm{m}^{\mathrm{L}} (\ \mu^{\mathrm{M}} \;.\; \mathrm{m}^{\mathrm{M}}\ \lambda\ )\ )
\end{aligned}$$

$$
\begin{aligned}
&= \mu^{M} \,.\, m^{M} \,(\, m^{M} \;\mu^{L} \,.\, m^{M} \,(\, m^{L} \;\mu^{L} \,)\,.\, \lambda \,.\, m^{L} \,(\, \mu^{M} \,.\, m^{M} \;\lambda \,)\,) \\
&\overset{(D0)}{=} \mu^{M} \,.\, m^{M} \,(\, m^{M} \;\mu^{L} \,.\, \lambda \,.\, m^{L} \,(\, m^{M} \;\mu^{L} \,)\,.\, m^{L} \,(\, \mu^{M} \,.\, m^{M} \;\lambda \,)\,) \\
&= \mu^{M} \,.\, m^{M} \,(\, m^{M} \;\mu^{L} \,.\, \lambda \,.\, m^{L} \,(\, m^{M} \;\mu^{L} \,.\, \mu^{M} \,.\, m^{M} \;\lambda \,)\,) \\
&= \mu^{M} \,.\, m^{M} \,(\, m^{M} \;\mu^{L} \,.\, \lambda \,)\,.\, m^{M} \,(\, m^{L} \,(\, m^{M} \;\mu^{L} \,.\, \mu^{M} \,.\, m^{M} \;\lambda \,)\,) \\
&= \mu^{M} \,.\, m^{M} \,(\, m^{M} \;\mu^{L} \,)\,.\, m^{M} \;\lambda \,.\, m^{M} \,(\, m^{L} \,(\, \mathrm{join}^{ML} \,)\,) \\
&= m^{M} \;\mu^{L} \,.\, \mu^{M} \;m^{M} \;\lambda \,.\, \mathrm{fmap}^{ML} \,(\, \mathrm{join}^{ML} \,) \\
&= \mathrm{join}^{ML} \,.\, \mathrm{fmap}^{ML} \,(\, \mathrm{join}^{ML} \,)
\end{aligned}
$$

Thus, the 3-tuple (ML, eta^{ML}, join^{ML}) is a monad. ■

Bibliography

[1] M. Barr and C. Wells, *Toposes, Triples, and Theories.* Springer-Verlag, 1985.

[2] M. Barr and C. Wells, *Category Theory for Computing Science.* Prentice Hall, 1990.

[3] J. Beck, Distributive laws. *Lecture Notes in Mathematics*, 80:119-140, Springer, 1969.

[4] H. Daumé III, *Yet Another Haskell Tutorial.* Hal Daume, 2002–2006.

[5] R. Goldblatt, Topoi: The Categorical Analysis of Logic, Studies in Logic and the Foundations of Mathematics, Volume 98. North-Holland Publishing Company, 1979.

[6] P. Hudak, J. Peterson, and J. H. Fasel, *A Gentle Introduction to Haskell 98.* Paul Hudak, John Peterson and Joseph Fasel, 1999.

[7] D. King and P. Wadler, Combining Monads. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992.

[8] S. Mac Lane, *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[9] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus.* Addison-Wesley Publishers Ltd., 1989.

[10] E. Moggi, Computational lambda-calculus and monads. In *Sympasium on Logic in Computer Science*, Asilomar, California; IEEE, June 1989.

[11] E. Moggi, An abstract view of programming languages. Course notes, University of Edinburgh, 1989.

[12] E. Moggi, Notions of computations and monads. *Information and Computation*, 93:55–92, 1989.

[13] M. Pierce, *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[14] F. Rabhi and G. Lapalme, *Algorithms: A Functional Programming Approach.* Pearson Education Limited, 1999.

[15] J. C. Reynolds, Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 97 in Lecture Notes in Computer Science. Springer-Verlag, January 1980.

[16] P. Wadler, Comprehending Monads. In *Conference on Lisp and Functional Programming*, Nice, France; ACM, June 1990.

[17] P. Wadler, Monads for functional programming. In *M. Broy, editor, Marktoberdorf Summer School on Program Design Calculi*, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Volume 118, August 1992.

[18] P. Wadler, The essence of functional programing (invited talk). In *19'th Symposium on Principles of Programming Languages*, Albuquerque, New Mexico; ACM, January 1992.