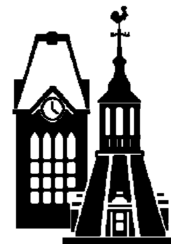
WPI-CS-TR-03-21 June 2003
emended, 18 September 2003

Monads for Programming Languages

by

John N. Shutt

Computer Science
Technical Report
Series



WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Monads for Programming Languages

John N. Shutt
jshutt@cs.wpi.edu
http://www.cs.wpi.edu/~jshutt/
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

June 2003
emended, 18 September 2003

Abstract

This paper assesses the relationship between the abstract mathematical concept of monads, and the applied area of programming languages.

Contents

0 Introduction **1**

1 The concept of monad **2**
1.1 Origins of the math 2
1.2 Notions of computation 10

2 Monadic programming **12**
2.1 Composing monads 13
2.2 Abandoning monads 14

3 The basis of monadic style **15**

4 Concluding note **17**

Acknowledgments **18**

Bibliography **18**

0 Introduction

Monads are a kind of mathematical structure that arises in category theory. Originally identified in that setting, since around 1990 they have been studied in the context of programming languages, on three successively narrower scales.

1. Eugenio Moggi applied monads to computation on a universal scale; e.g., [Mo89]. He was looking for a categorical tool to describe the shape of impure computation (state, imperative control, etc.), and settled on monads.

2. Others subsequently used monads on a merely large scale, to encapsulate the handling of impure language features by an interpreter; e.g., [Wa92c].

3. Monads have since been used on a finer-grained scale, to encapsulate individual impure phenomena occurring within arbitrary programs; e.g., [SeSa99].

The "universal" scale overarches one universe of discourse —which is to say, one programming language— at a time. Commensurate with the basic principle that *all software engineering artifacts are languages*[1], anything that applies to languages applies to arbitrarily small entities within a program. Thus the subsequent narrowing of scale was a natural development.

There is no universally applicable technique for "composing" monads, i.e., combining monads that represent arbitrary impure phenomena to produce a single monad that represents both phenomena at once. This lack doesn't seem particularly problematic at the universal scale, because there are only so many universes one expects to live in. However, as the scale of application narrows, lack of monadic composition becomes increasingly worrisome. At a sufficiently narrow scale, it becomes apparent that the problem of combining monads is a mathematical manifestation of the problem of combining software components.

Another lingering question concerning monads is whether they are really the right tool for the job. They seem to have suggested themselves to Moggi originally because he was looking for something with nice mathematical properties, and monads were a known form with nice mathematical properties. Borrowing structures from one field to another like this can be hazardous; just because monads have properties that are helpful in mathematics doesn't mean they're necessarily right for modeling general computation (just as the suitability of traditional OO inheritance for coding 1960s-style simulation software doesn't necessarily make it ideal for general-purpose software engineering).

The following sections relate the mathematical concept of monads to their application to programming languages. Although technical details are present when necessary (as in the initial explanation of the mathematical concept), the emphasis

[1] This principle is central to the RAG model [Sh98]. Here I've deliberately phrased it to compare and contrast with the principle from [Kr01] that programming languages are software engineering artifacts.

throughout is on the big picture: what monads are (§1.1); why and (to some extent) how they were first applied to computation (§1.2); and why and how their application has diverged from its mathematical roots (§2).

1 The concept of monad

In his original application of monads to computation, Moggi seems [Mo89] to have been motivated by an interest in handling impure forms of computation in the framework of the λ-calculus. Work in the late 1980s had directly constructed variants of λ-calculus to handle various particular impure facilities (e.g., [FeFr89]). Moggi, however, followed the principle that category theory is prior to λ-calculus (because category theory is a generalization of the set-based function theory on which λ-calculus is founded); therefore, he reasoned, natural generalizations of the foundations of λ-calculus ought to be cleanly expressible in categorical form. He chose monads as a categorical structure that would serve his purpose for a variety of computational impurities.

Below, §1.1 discusses the mathematical and conceptual underpinnings of monads; while §1.2 describes how Moggi's approach applies monads to some particular forms of computational impurity.

1.1 Origins of the math

This subsection is about the big picture of the mathematical concept of monad. The material is approached in parallel on three mutually supporting levels: the big picture itself, formal definitions, and simple examples.

Mathematics is (in one view of the beast) the systematic study of things that are, in some sense or other, well-behaved. If there isn't some kind of nice behavior there, it isn't possible to conduct a study that is (intrinsically) systematic. On the other hand, anything that *does* exhibit some kind of well-behavedness is subject to mathematical study; and it happens that *well-behavedness* is itself, unsurprisingly, rather well-behaved, so it should be possible to study it mathematically. In essence, the mathematical study of well-behavedness is *category theory*.

The major concepts of category theory build one on another. In order (and as they will be defined below), they are: *categories*, *functors*, *natural transformations*, and *adjunctions*. The first three were manifest in category theory from the start; but the last and highest-level, adjunctions, didn't emerge explicitly for about fifteen years after the others [Ma71, §IV endnotes]. Adjunctions are actually just one way of describing a high-level phenomenon that also manifests itself in a number of other related forms — one of which, in particular, is *monads*.

Categories

The starting point for category theory is the notion of a well-behaved family of morphisms —or *arrows*— each from an object of type X to another object of type X. This is a generalization of the family of all functions between sets. A category consists of

- a directed graph; the vertices are called *objects*, the directed edges are called *arrows* or *morphisms*, the source of an arrow is its *domain*, the destination is its *codomain*.
- an associative binary partial operation on arrows called *composition*, that is defined just when the codomain of one edge is the domain of another.
- an *identity arrow* for each object, whose domain and codomain are both that object, and that is an identity under composition both on the right and on the left.

Here is the definition expressed formally.

Definition 1.1 A graph G consists of the following collections and operations.

- A collection of objects, $\mathrm{Obj}\, G$. $a \in \mathrm{Obj}\, G$ may be written as "$a \in G$".
- A collection of arrows, $\mathrm{Arr}\, G$. $f \in \mathrm{Arr}\, G$ may be written as "f in G".
- Operations **dom** and **cod** mapping each arrow f to an object $\mathbf{dom}\, f$ called its domain and $\mathbf{cod}\, f$ its codomain. "$a = \mathbf{dom}\, f$ and $b = \mathbf{cod}\, f$" may be written as "$f\colon a \to b$", or diagramatically as

$$\overset{a}{\bullet} \xrightarrow{\quad f \quad} \overset{b}{\bullet}$$

A category C is a graph with the following additional operations and properties.

- A partial binary operation $\circ$ on arrows, mapping each of certain pairs of arrows f, g to an arrow $g \circ f$ called their composition.
  - For any f, g in C, $g \circ f$ is defined iff $\mathbf{cod}\, f = \mathbf{dom}\, g$, and if it is defined, $\mathbf{dom}\, g \circ f = \mathbf{dom}\, f$ and $\mathbf{cod}\, g \circ f = \mathbf{cod}\, g$. Diagramatically,

$$\begin{array}{ccc} & \bullet & \\ {\scriptstyle f}\nearrow & & \searrow{\scriptstyle g} \\ \bullet & \xrightarrow[g \circ f]{} & \bullet \end{array}$$

  - Composition is associative; that is, $h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$ whenever the relevant compositions are defined.
- An operation **id** mapping each object x to an arrow $\mathbf{id}_x$ such that
  - for every arrow $f\colon a \to b$, $f \circ \mathbf{id}_a = \mathbf{id}_b \circ f = f$.

□

The canonical example of a category is the category **Set**, whose objects are sets[2] and whose arrows are total functions from set to set. Very many of the interesting examples of categories consist of all sets with a certain kind of additional structure, together with all functions from set to set that preserve that structure; for example, category **Grp** has as objects, groups, and as morphisms, group homomorphisms; **Mon** has as objects monoids, and as arrows monoid homomorphisms; and so on.

Note, in passing, that the core of a category is its *composition operation*; all the other parts of the category are implicit in that. Although, given the objects of a category, there is often a single most obvious choice of arrows, it is not uncommon for two categories to have the same objects but different arrows; for example, the category **Pfn** has as objects all sets, but as arrows all *partial* functions from set to set. Usually, once the set of arrows has been defined there is one really obvious and natural way to define composition; but occasionally, two categories of interest will have the same objects *and* the same arrows, but different rules for composition[3].

Functors

A category may itself be viewed as "a set with a certain kind of additional structure"; so that, given a suitable definition of category-structure-preserving morphisms, one could form a category of categories. The natural definition of a (homo)morphism of categories, called a *functor*, is a mapping of objects to objects and arrows to arrows that preserves all the operations — domain, codomain, composition, and identity. That is,

---

[2] I'm ignoring some obfuscating complications that a very careful treatment would have to address, involving the foundations of mathematics and avoiding Russell's Paradox (whether the set of all sets that don't contain themselves contains itself). See [Ma71, §I]

[3] For example [MaAr86], category **Mfn** has sets for objects, and multivalued functions for arrows —that is, an arrow $f\colon A \to B$ maps each value $a \in A$ to a set of values $f(a) \in \mathcal{P}(B)$— with composition defined by

$$(g \circ f)(x) \quad = \quad \bigcup_{y \in f(x)} g(y)$$

while **ANMfn**, the category of "multivalued functions with all-or-nothing composition" has the same objects and arrows, but composition is defined by

$$(g \circ f)(x) \quad = \quad \begin{cases} \emptyset & \text{if } \exists y \in f(x) \text{ such that } g(y) = \emptyset \\ \bigcup_{y \in f(x)} g(y) & \text{otherwise} \end{cases}$$

Definition 1.2 Given categories C, D, a functor $T: C \to D$ consists of an object function $T: \mathrm{Obj}\, C \to \mathrm{Obj}\, D$ and an arrow function $T: \mathrm{Arr}\, C \to \mathrm{Arr}\, D$, such that

$$
\begin{aligned}
\mathbf{dom}\, Tf &= T\, \mathbf{dom}\, f \\
\mathbf{cod}\, Tf &= T\, \mathbf{cod}\, f \\
(Tf) \circ (Tg) &= T(f \circ g) \\
\mathbf{id}_{Ta} &= T\, \mathbf{id}_a
\end{aligned}
$$

□

The category whose objects are all categories and whose arrows are all functors is called **Cat**, the category of all categories.[4]

For example, for any set A, the monoid freely generated over A consists of the set A^* of all strings over alphabet A, together with concatenation as the binary operation of the monoid, and the empty string Λ as the identity element (since $\forall$ strings $w \in A^*$, $\Lambda w = w \Lambda = w$). Let's call this monoid MA; so M maps each object of category **Set** to an object of category **Mon**. Further, for any function on sets $f: A \to B$, there is an obvious monoid homomorphsim $Mf: MA \to MB$ that takes any string over A, and uses f to map each element $a \in A$ of the string to the corresponding element $fa \in B$. (This is the operation that is provided in Scheme by procedure ***map***.) So M is a functor from **Set** to **Mon**; in symbolic notation, $M: \mathbf{Set} \to \mathbf{Mon}$.

On the other hand, we can also define a functor $U: \mathbf{Mon} \to \mathbf{Set}$ that maps each monoid N to its underlying set of elements UN, and maps each monoid homomorphism $h: N_1 \to N_2$ to its underlying function Uh from elements of N_1 to elements of N_2. This functor U is called the *forgetful functor from* **Mon** *to* **Set**.[5]

Note that the composed functor $U \circ M: \mathbf{Set} \to \mathbf{Set}$ maps each set A to the set A^* of strings over alphabet A.

Natural transformations

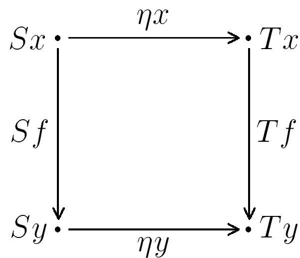
Given two functors $S, T: B \to C$ (i.e., *parallel* functors), a natural transformation η from S to T is a family of morphisms in C that relate the image of S to the image of T (in, of course, a very well-behaved way — this being category theory, after all). Formally,

Definition 1.3 Given any two functors $S, T: B \to C$, a natural transformation η from S to T, denoted $\eta: S \dot{\to} T$, is a mapping $\eta: \mathrm{Obj}\, B \to \mathrm{Arr}\, C$ such that for all arrows $f: x \to y$ in B, $(\eta y) \circ (Sf) = (Tf) \circ (\eta x)$. This equation may also be expressed by saying that the following diagram *commutes*, meaning that for any

[4]Actually **Cat** only contains all *small* categories, which means among other things that it doesn't contain itself.

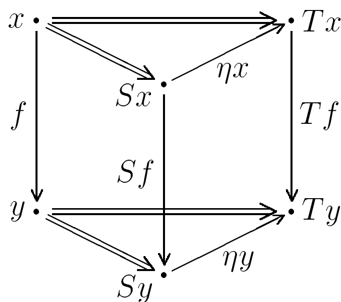
[5]Any functor such as this, whose action is essentially to "forget about some of the structure" of an object (typically, a set with some additional structure), is called a *forgetful functor*. Forgetful functors are usually called U.

two points shown on the diagram, the composition of arrows on every path from one point to the other are equal[6].



□

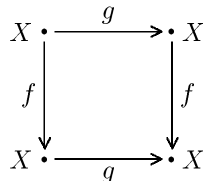
While the above commutative diagram shows how the interaction of f and η manifests itself in category C, the overall interaction spans categories B and C and is only implicit in the diagram. To provide a more complete visualization of the situation in a single picture (of a 3D figure, specifically a triangular prism), we'll adopt the further notational convention of depicting the action of a functor on an object by a double-shafted arrow.



To avoid cluttering the picture, only the objects and arrows are labeled. The front left face of the prism is swept out by S acting on f. The back face of the prism is swept out by T acting on f. The front right face is the commutative diagram in category C. η sweeps out the angle from the plane of S to the plane of T.

Now, here is a simple example of a natural transformation. Consider the *identity functor* on category **Set**, $\mathbf{Id_{Set}}\colon \mathbf{Set} \to \mathbf{Set}$, which maps each object and arrow of **Set** to itself. There is an obvious natural transformation η from this functor $\mathbf{Id_{Set}}$ to

[6]This use of the word *commute* may seem less idiosyncratic if one considers that, given any two functions $f, g\colon X \to X$ on a set X, the following diagram *commutes* iff f and g commute in the usual sense; that is, iff $f \circ g = g \circ f$.



the functor $U \circ M\colon \mathbf{Set} \to \mathbf{Set}$ described earlier, which maps each set X to the set of strings over alphabet X. That is to say, $\eta\colon \mathbf{Id}_{\mathbf{Set}} \dot{\to} U \circ M$. For each set X, arrow ηX maps each letter in the alphabet X to the string of length one consisting of that letter; that is, $(\eta X)x = x$. For any function on sets $f\colon X \to Y$, function $(U \circ M)f = UMf$ maps strings over alphabet X to strings over alphabet Y by applying f to each letter of the string. Then η is a natural transformation because the following diagram commutes.

$$\begin{array}{ccc} X & \xrightarrow{\eta X} & UMX \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle UMf} \\ Y & \xrightarrow[\eta Y]{} & UMY \end{array}$$

It should be clear that this diagram does commute: if you apply f to a letter $x \in X$ and then make a string of length one out of the result, you get the same thing as if you'd first converted x to a string of length one and then applied f to each letter of the string.

Adjunctions

Recall that we first characterized a category as a well-behaved family of morphisms each from an object of type X to another object of type X. Category theory also provides a more general notion of a well-behaved family of morphisms each from an object of type X to an object of type Y; a family of morphisms (or perhaps a less loaded term would be *directed relationships*) of this more general kind is called an *adjunction*.

Rather than attempt to motivate each component of an adjunction in terms of this high-level view of the construction, with the specter of the unknown formal definition hanging over the discussion, we'll present the full-blown formal definition first, secure it with a concrete example, and only then address the roles of its various parts in the high-level view.

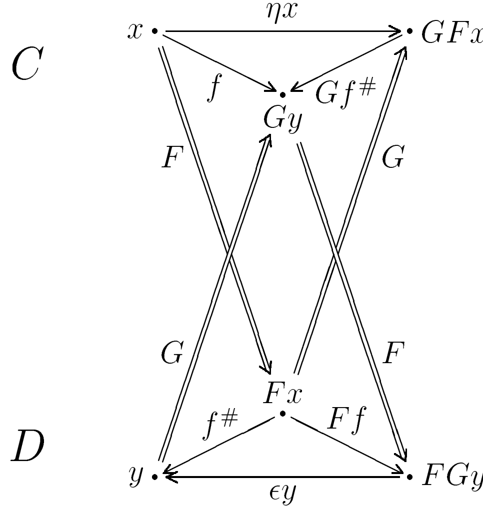
Definition 1.4 Given two categories C and D, an adjunction from C to D is a four-tuple $\langle F, G, \eta, \epsilon \rangle$, denoted $\langle F, G, \eta, \epsilon \rangle\colon C \rightharpoonup D$, where

- F is a functor $F\colon C \to D$, called the *left adjoint*.
- G is a functor $G\colon D \to C$, called the *right adjoint*.
- η is a natural transformation $\eta\colon \mathbf{Id}_C \dot{\to} G \circ F$, called the *unit*.
- ϵ is a natural transformation $\epsilon\colon F \circ G \dot{\to} \mathbf{Id}_D$, called the *counit*.

- For every arrow $f\colon x \to Gy$, there is a unique arrow $f^{\#}\colon Fx \to y$ such that $(Gf^{\#}) \circ (\eta x) = f$.
- For every arrow $f^{\#}\colon Fx \to y$, there is a unique arrow $f\colon x \to Gy$ such that $(\epsilon y) \circ (Ff) = f^{\#}$.

□

The situation is summed up by the following picture (again of a solid figure).



The top surface of the figure is a commutative diagram in category C, expressing the constraint that $(Gf^{\#}) \circ (\eta x) = f$; the bottom surface is a commutative diagram in category D, expressing $(\epsilon y) \circ (Ff) = f^{\#}$. The two are connected by two parallelograms: one swept out by F acting on f (from f to Ff), and the other swept out by G acting on $f^{\#}$ (from $f^{\#}$ to $Gf^{\#}$).

In assembling a specific example of an adjunction, we already have most of the pieces from our running example. Let $C = \mathbf{Set}$, $D = \mathbf{Mon}$, $F = M$ (that freely generates monoids over sets), and $G = U$ (forgetful from **Mon** to **Set**).

To fill out the rest of the labels in the diagram, we can rely heavily on common sense; with adjunctions, the obvious guess is usually right (as one might expect in a systematic study of well-behaved well-behavedness). The unit is a natural transformation from $\mathbf{Id}_{\mathbf{Set}}$ to $G \circ F = U \circ M$; we've already seen it, mapping each letter in alphabet x to itself as a string of length one in x^*. To reason out the counit, consider the particular case that monoid $y = \langle \mathbb{N}, +, 0 \rangle$, the additive monoid of nonnegative integers. $Gy = U\langle \mathbb{N}, +, 0 \rangle = \mathbb{N}$ is the set of nonnegative integers, so an arrow $f\colon x \to Gy$ maps each letter of alphabet x to a nonnegative integer. $FGy = M\mathbb{N} = \langle \mathbb{N}^*, \cdot, \Lambda \rangle$ is the monoid of strings of nonnegative integers, and $Ff = Mf$ is the monoid homomorphism that applies f to each letter of a string over x, producing a string of nonnegative integers. The counit has to map a string of integers to a single integer;

it's not much of a stretch to think of *adding up* the elements of the string. Returning to the general case of arbitrary monoid y, MUy is the monoid of strings over the underlying set of y, and $\epsilon y\colon MUy \to y$ combines the elements of each such string using the associative binary operation provided by monoid y (using the identity of y for the empty string)[7].

Monoid homomorphism $f^{\#}\colon Mx \to y$ takes a string over alphabet x, applies f to each letter of the string, and combines the results using the associative binary operation of monoid y.

Recall the initial characterization of an adjunction as a well-behaved family of directed relationships from objects of one type to objects of another. Since the adjunction establishes a bijection between arrows $f\colon x \to Gy$ and $f^{\#}\colon Fx \to y$, we can view both arrows as manifestations, in C and D, of a directed relationship from set x to monoid y that does not belong strictly to either category. The adjoint functors F and G allow each of objects x, y to manifest in the other category so that the relation from x to y can appear in both places. The unit and counit are simply the form of the family's well-behavedness.

Monads

A monad is, intuitively, the shadow cast by an adjunction in its domain category (which we've been calling C).

Since the codomain category D will never occur explicitly in the monad, we only need one functor, the composite of the right and left adjoints. Call this composite $T = G \circ F\colon C \to C$. The unit η of the adjunction can now be described as a natural transformation $\eta\colon \mathbf{Id}_C \dot{\to} T$. The counit casts its shadow in C by means of the adjoint functors: Starting with an object $x \in C$, the left adjoint F maps it to an object $Fx \in D$, which ϵ maps to an arrow $\epsilon Fx\colon FGFx \to Fx$ in D, which the right adjoint G maps to an arrow $G\epsilon Fx\colon GFGFx \to GFx$ in C. Call this natural transformation[8] $\mu = G\epsilon F\colon T \circ T \dot{\to} T$.

Here is the formal definition of a monad, using the convenient notation $T^0 = \mathbf{Id}_C$ and $T^{n+1} = T \circ T^n$.

Definition 1.5 Given a category C, a monad in C is a triple $\langle T, \eta, \mu \rangle$ where

- T is a functor $T\colon C \to C$.
- η is a natural transformation $\eta\colon T^0 \dot{\to} T$, called the *unit*.

[7] Scheme programmers familiar with the Wizard Book [AbSuSu96] may recognize this homomorphism as procedure `accumulate`.

[8] Of course the result of all this manipulation is still a natural transformation, because in category theory everything has an uncanny way of coming out right — a corollary of the fact that category theory is positively *dripping* with well-behavedness.

- μ is a natural transformation $\mu: T^2 \dot{\rightarrow} T$, sometimes called the *multiplication*[9].
- For every object $x \in C$,

$$\begin{array}{lllll} \mu x \circ T\mu x & = & \mu x \circ \mu T x & & : T^3 x \rightarrow T x \\ \mu x \circ T\eta x & = & \mu x \circ \eta T x & = \mathbf{id}_{Tx} & : T x \rightarrow T x \end{array}$$

□

The monad equations follow from the properties of an adjunction[10]. On the other hand, since a monad evidently forgets some of the details of the adjunction, it shouldn't come as a surprise that many adjunctions may define the same monad. In particular, an adjunction considers all objects $y \in D$, but the corresponding monad in C ignores all objects in D that aren't of the form Fx for some $x \in C$.

Even though category D is no longer explicitly present in monad $\langle T, \eta, \mu \rangle$, its identity and composition operations are still visible. Given any two arrows $f: a \rightarrow Tb$ and $g: b \rightarrow Tc$ in C, they can be "composed" through the monad to produce an arrow $(f; g): a \rightarrow Tc$, as follows. (Here, notation $f; g$ follows the common programming idiom for sequencing. Note carefully that this notation uses the opposite ordering from categorical $g \circ f$.)

The adjunction matches arrow $f: a \rightarrow GFb$ in C with $f^{\#}: Fa \rightarrow Fb$ in D, and $g: b \rightarrow GFc$ in C with $g^{\#}: Fb \rightarrow Fc$ in D; obviously these two arrows compose in D, giving $g^{\#} \circ f^{\#}: Fa \rightarrow Fc$, which the adjunction matches with an arrow $a \rightarrow GFc$ in C. A moment's thought (and perhaps sketching a commutative diagram or two) will affirm that the appropriate "composed" arrow is $(f; g) = \mu c \circ Tg \circ f$.

The identity operation of D is also effectively visible, in the form of natural transformation η. For every arrow $f: a \rightarrow b$ in C, $(\eta a; f) = (f; \eta b) = f$. (This happens because the adjunction matches each arrow $\eta x: x \rightarrow GFx$ in C with arrow $\mathbf{id}_{Fx}: Fx \rightarrow Fx$ in D.)

1.2 Notions of computation

Moggi uses a monad $\langle T, \eta, \mu \rangle$ to describe what he calls a *notion of computation*[11]. Objects in his category of discourse C are *types*; a type t may be thought of as a pair $t = \langle d, S \rangle$ of a type designation d and a set of values S. C is thus somewhat distinct from **Set** in that, depending on the type system chosen, types may have all the same

[9]This terminology alludes to the formal analogy between monads and monoids, in which η and μ correspond respectively to the identity element and binary operation of a monoid. Observe that the term *unit* makes far more sense in the context of this analogy, than it did when it was applied to the same natural transformation in the underlying adjunction of the monad.

[10]The first equation is derived ultimately from the fact that ϵ is a natural transformation in D, while the other two equations come from setting $f = \eta x$ in the adjunction's commutative diagram in D, and $f^{\#} = \epsilon y$ in the adjunction's commutative diagram in C.

[11]In [Mo89], he first defines a *computational monad* to be a monad $\langle T, \eta, \mu \rangle$ such that for all objects $x \in C$, ηx is mono (the categorical generalization of a one-to-one function), but then immediately admits that the mono requirement may not hold for all interesting cases.

possible values but still be distinct because they have different designations. Arrows in C are arbitrary functions between the sets of values. The functor T of the monad is a *type constructor*, since it maps any given type a to a type Ta; he characterizes T as mapping a type a of values to the type Ta of "computations of type a". A program is a function mapping values to computations, thus an arrow in C of the form $a \to Tb$.

Moggi's use of the word "computation" to describe elements of Ta must be interpreted carefully. In the realm of automata (and therefore operational semantics), computation is typically a function on some kind of configuration space. For example, a configuration of a stateful computation might be a pair $\langle v, s\rangle \in V \times S$ of a value $v \in V$ with a machine state $s \in S$; computation would then be a function $V \times S \to V \times S$ mapping one configuration to another. However, in the scenerio Moggi envisions, a program $p: a \to Tb$ maps each input value $v \in a$ to an element $c \in Ta$; so, in our stateful computation, element c already knows what the input value is. Thus, instead of a general function $V \times S \to V \times S$, c is a function $S \to V \times S$. So $Tb = (S \to b \times S)$, and $p: a \to (S \to b \times S)$.

In general, an element of type Ta is a computation whose output value is of type a and whose input value is fixed.

For another typical example, consider nondeterminism. In the usual sense, a nondeterministic computation with input type a and output type b would be simply a function $a \to \mathcal{P}b$; but once the input value is known, the remaining "computation" is simply a subset of b; so $Tb = \mathcal{P}b$, and program $p: a \to \mathcal{P}b$.

Where there is a monad, there is an underlying adjunction. Its domain category C is the category of data types, while the implicit codomain category D is the category of computation types. Programs are arrows from computation type to computation type; so, assuming that every object in D belongs to the image of the left adjoint functor F, the adjunction matches each program $Fa \to Fb$ in D with an arrow $a \to GFb$ in C.

For any data type a, program $\eta a: a \to Ta$ is the identity under program composition (since it corresponds to an identity arrow in the implicit category D). Conceptually, program ηa takes an input value $v \in a$ and returns a computation that "does nothing" and produces output value v. For nondeterministic computation, $(\eta a)v = \{v\}$, the singleton set containing the input value; while for stateful computation, $(\eta a)v = \lambda s.\langle v, s\rangle$, the function mapping a machine state s to the pair of the input value and the same machine state s.

The program composition operation $\cdot\,;\cdot$ supported by μ defines the semantics of directing the output of one program to the input of another. (This is the composition operation of category D.) In the nondeterminism example,

$$(p_1; p_2)x \quad = \quad \bigcup_{y \in p_1 x} p_2 y$$

meaning that the output of $p_1; p_2$ on x could be anything output by p_2 on any output

of p_1 on x; while for stateful computation,

$$(p_1; p_2)x \quad = \quad \lambda s.p_2(p_1\langle x, s\rangle)$$

which is to say that, to run computation $(p_1; p_2)x$ with initial state s, first run computation $p_1 x$ with initial state s to produce a configuration $\langle x', s'\rangle$, then run computation $p_2 x'$ with initial state s'.

Building on his use of monads as notions of computation, Moggi defines semantic rules for a generic programming language using λ-calculus-style syntax [Mo89], which he calls *computational lambda calculus*, or $\lambda_{\mathbf{c}}$. His semantic rules are parameterized by the monad $\langle T, \eta, \mu\rangle$, so that results deduced from his rules will automatically apply to all variant calculi that fit his monadic framework; a particular variant calculus is constructed by fixing the monad, and adding appropriate language primitives for accessing whatever form of impurity the monad supports.

The two core assumptions underlying Moggi's monadic strategy (embodied by $\lambda_{\mathbf{c}}$) are that

1. every pure function can be understood as an impure function (via η), and

2. all impure function composition can be done through the monad (via $\cdot;\cdot$).

In applied "monadic" programming style, the mathematical structure used will almost never be a monad; but these two assumptions will linger (cf. §2.2).

2 Monadic programming

In effect, Moggi used monads in his mathematics as an *encapsulation* device, to isolate different mathematical concerns from each other, making them more independently tractable[12]. Advocates of "pure functional" programming languages had been looking for a way to correct the obvious deficit between their desire to use pure functions, and the need for impure behavior —such as input and output— in real-world programs; they quickly latched onto monads as a way of incorporating impurities into a "pure" functional language in a controlled manner, isolable from the pure portions of the language. (See [PJWa93].)

Wadler also took the natural step[13] of scaling down Moggi's parameterization of programming language semantics by a monad, to parameterization of an *interpreter* for a language ([Wa92c]). In this context, the monad is more explicitly an encapsulation device: the interpreter is constructed with a "monad-shaped hole" in it, and the semantics of the interpreted language can be varied by plugging in an appropriate

[12] The division of orthogonal concerns isn't usually called *encapsulation* when it occurs in mathematics per se; instead it is referred to, if at all, by various other names such as "independence", "orthogonality", or even (in a specialized but prominent case) "separation of variables".

[13] Of course the natural step isn't necessarily obvious, and when it *is* obvious it generally takes even longer for someone to think of it.

monad. The encapsulation of impurities isn't complete because, as [St94] points out, one must also tweak the code for the interpreter to add appropriate syntax to exploit whatever semantic features have been provided by the monad; so the variations in the interpreter are not quite entirely restricted to the monad itself[14].

2.1 Composing monads

Because Moggi had originally been looking at the problem of defining the semantics of impure forms of computation, he'd had what seemed to be a quite finite number of targets; so hand-fashioning a monad (and, as already noted, a specialized syntax) for each targeted combination of impurities would not have seemed unreasonable. Wadler, however, had promoted monads to the status of software components, and thus engaged the programmer's reflex to play around with them, build lots of different ones, and find ways to parameterize and combine them to build new ones with ever-greater facility.

Unfortunately, at this point the monad's mathematical underpinnings come back to haunt it. The monad is a manifestation of an *adjunction*, which is a directed relation from the explicit category C to the implicit category D. There is in fact a very natural way to compose an adjunction $\phi\colon C \rightharpoonup D$ with another adjunction $\chi\colon D \rightharpoonup E$ to produce an adjunction $(\chi \circ \phi)\colon C \rightharpoonup E$; and this composition[15] has the appropriate properties for a category — it's associative, and every category has an identity adjunction. The category whose objects are categories and whose arrows are adjunctions is called **Adj**.

For a monad to be a "notion of computation", though, it has to be in the category C of data types; and that means that the underlying adjunction must have domain C. The codomain category D almost certainly isn't C, since that would mean that the category of computations (D) has no more structure than the category of pure typed functions (C). So if ϕ and χ are the underlying adjunctions of two (nontrivial) notions of computation, they almost certainly aren't composable —at least, not via ordinary composition of adjunctions— because they both have domain C and neither has codomain C.

One straightforward way of working around the non-composability problem is to define, not monads directly, but higher-level functions that take a monad in C as a parameter and return another monad in C as a result — a "monad parameterized by another monad". Steele advocated this approach in [St94], under the name

[14]Customizing syntax is really a problem only at the large-but-not-universal scale that Wadler was working at. Moggi had had no difficulty when working at a universal scale because he was perfectly willing to customize the syntax for each variant; and the smaller-scale phenomena to which monads have since been applied already had syntactic strategies associated with them before monads were ever brought into the picture.

[15]The reader can readily work out this composition operation by playing around with the functors and natural transformations of ϕ and χ — because, once again, category theory is so steeped in well-behavedness that everything tends to work out right.

"pseudomonads".

Another approach was suggested by Jones and Duponcheel in [JoDu93]. Whereas Steele had added facility to each individual monad, in order to parameterize it, Jones and Duponcheel added facility to a particular *pair* of monads. In particular, given monads $\langle M, \eta_M, \mu_M \rangle$ and $\langle N, \eta_N, \mu_N \rangle$, they assumed that a "composition" monad would have functor $T = M \circ N$ and unit $\eta = \eta_M \circ \eta_N$, and identified several different sufficient conditions for the construction of a natural transformation $\mu : T^2 \dot{\rightarrow} T$ such that $\langle T, \eta, \mu \rangle$ is a monad[16]. They also admitted mathematical structures that do not have all the properties of a monad, a fact that they were quite open and pragmatic about, noting that a monad-like entity may still usefully serve as an encapsulating structure even though it lacks some of the mathematical well-behavedness of a monad.

2.2 Abandoning monads

Papers on monadic programming tend to use the notation of one or another extant functional language (typically Haskell or ML, except for Moggi's early work which was about the expression of language semantics through means other than actual program code). Functional languages, however, cannot express any of the well-behavedness properties[17] that are essential to the applicability of the underlying concepts of monads — essential because without those properties there is no underlying adjunction, just a type constructor and a couple of polymorphic functions. Consequently, as work on monads in programming has become increasingly applied, the well-behavedness properties have tended to fade from view, leaving only a template for mechanical structure of program modules. For example, [Pr97] describes a practical programming device in which "features" are defined using mechanically monadic structure, and then pairwise compositions of features are defined using "lifting" structures mechanically similar to the monad transformers of [JoDu93]. His features are essentially OO abstraction classes, and lifting is a generalization of OO inheritance — neatly exemplifying the nature of the practical interaction between composition and encapsulation.

The properties of monads have also been gradually weakened in *theoretical* work on programming languages, as theory is devised to describe applications that themselves favor practical encapsulation over theoretical well-behavedness. A typical (also topical) example is [Wa99], which recasts an *effect* system in monadic style.

Effects are a device for encapsulating computational impurities that has emerged from the (comparatively) applied tradition of *type systems* (whereas monadic style emerged for the same purpose out of the theoretical tradition of category theory). As

[16]They expressed their conditions entirely as equations in polymorphic functions on category C. Whether the conditions can be recast elegantly in terms of underlying adjunctions is beyond the scope of this paper. I do suspect —based mostly on general principles— that such a recasting once identified would be extremely simple and, consequently, *explaining* it would not be at all difficult.

[17]The notion of a programming language that incorporates correctness proofs in the program code, while not within current technology, is discernible in current research trends.

a conventional type constrains the range of permissible values of a datum, an effect constrains the range of permissible side-effects of an impure function. Function type notation is amended by writing the effect above the arrow; thus function $f: \tau \xrightarrow{\sigma} \tau'$ takes input of type τ, has effect σ, and produces output of type τ'. Effects have an associative binary operation $\cup$ (union) and an identity $\emptyset$ (the null effect); pure functions have effect $\emptyset$, and the effect of a composition of functions is the union of their effects, so that $f: \tau \xrightarrow{\sigma} \tau'$ and $g: \tau' \xrightarrow{\sigma'} \tau''$ imply $g \circ f: \tau \xrightarrow{\sigma \cup \sigma'} \tau''$.

Wadler defines a "monad" structure in which the functor is parameterized by an effect, thus T^σ. Object $T^\sigma a$ is the type of computations with effect σ and output type a. For each effect σ there is a naturally associated monad $\phi^\sigma = \langle T^\sigma, \eta^\sigma, \mu^\sigma \rangle$, but Wadler has no interest in these monads individually. Instead he views the entire family of them as a single monad-like entity. Binary operation $\cdot ; \cdot$ (representing the third element of a monadic form) maps $f: a \to T^\sigma b$ and $g: b \to T^{\sigma'} c$ to $(f; g): a \to T^{\sigma \cup \sigma'} c$. Note that this cannot correspond to categorical composition in the implicit codomain of an adjunction, because in general the codomain of g isn't the codomain of $(f; g)$; but in case $\sigma' = \sigma$ it collapses to program composition in monad ϕ^σ, while in general it connects monads ϕ^σ and $\phi^{\sigma'}$ to a monad $\phi^{\sigma \cup \sigma'}$ that is therefore, in a sense, their composite (though in not at all the sense of Jones and Duponcheel). The "unit" of Wadler's structure is $\eta = \eta^\emptyset$, which is the natural choice for the entire family of monads because $\eta^{\sigma'} a$ is a left and right identity under $\cdot ; \cdot$ with programs in monad ϕ^σ iff $\sigma' \subseteq \sigma$, hence $\eta^\emptyset a$ is a left and right identity under $\cdot ; \cdot$ with programs in *all* monads ϕ^σ.

3 The basis of monadic style

The essence of the mathematical concept of monad is the existence of an implicit underlying adjunction; in effect (whatever his intent), Moggi's $\lambda_\mathbf{c}$ presumes that the category of pure functions is adjunctively related to the category of programs. However, the subsequent departure of "monadic style" from the mathematical concept demonstrates that, if monadic style *has* a conceptual basis, that basis isn't monads. So it's worthwhile to ask what such a basis might be[18].

A particularly rich source of insight into what monadic style is, and is not, is Wadler's paper [Wa93] on composable continuations. Wadler characterizes monadic style as a generalization of continuation-passing style. The idea behind continuations is that a configuration (i.e., intermediate state of computation) can be partitioned into a *value*, representing the culmination of past computation, and a *continuation*, representing all future computation. The usual monadic treatment of continuations uses functor $Ta = ((a \to O) \to O)$; since Moggi's "computations of type a" already know what their input value is, what remains to be specified is the continuation $a \to$

[18]This question has an inherent subjectivity, in consequence of which this section will necessarily have a distinct thread of editorial content — subdued as feasible, but nonetheless present.

O. The fully explicated type of a program $p: a \rightarrow Tb$ is thus $p: a \rightarrow ((b \rightarrow O) \rightarrow O)$.

The generalization for "monadic style" is that a program p curries computation —which is a mapping from initial configurations to final configurations— so that p inputs only a data value of type a, leaving the rest of the initial configuration (if any) unspecified, so that a "computation" (in Moggi's sense) of type Tb maps *all of an initial configuration except the input data value* to a complete final configuration. However, stating the principle thus baldly, it becomes evident that two assumptions have to have been stipulated before monadic style can be applied:

1. Computation is a mapping from initial configurations to final configurations.

2. Each configuration has a distinguishable part that may be regarded as an "input value".

At first glance, both of these assumptions sound reasonable[19]; however, there *is* a conceptual problem here, originating in assumption (1) and visible in the continuation-monad functor $Ta = ((a \rightarrow O) \rightarrow O)$. Type O is the data type of the final result of computation. But as observed earlier, a continuation represents all future computation; as Scheme first-class continuations (for example) are actually experienced by a programmer, a continuation *doesn't return*: it has an input type (which would be only implicit in Scheme, of course), but it shouldn't *have* an output type. O is an artifact of the pure functional programmer's (or mathematician's) determination to express computation entirely in terms of pure functions.

(I don't claim to have an alternative approach ready to hand; I merely suggest that our approach to describing computation should be driven by the nature of computation, but at present seems to be driving our perception of computation instead. As to whether the solution is a drastic change of strategy or a subtle modulation of tactics, I proffer no opinion.)

Wadler's treatment of composable continuations further suggests the existence of some kind of implicit conceptual structure that the mathematics is failing to exploit.

An ordinary continuation is captured by (in the syntax adopted by Wadler) an expression (escape $f.e$), which evaluates e with variable f bound to the continuation surrounding the escape expression. Continuation f is a "function that never returns". A *composable* continuation is captured by an expression (shift $f.e$), which snips off a prefix of the continuation surrounding the shift expression and evaluates e with f bound to that prefix. The prefix stops at the nearest dynamically enclosing reset expression, (reset e). Because the prefix has a stopping point as well as a starting point, it's a function, hence composable. Here's a very simple example (from [Wa93]):

$$1 + (\text{reset } (10 + (\text{shift } f.(f(f\ 100)))))$$

The construct $(\text{reset}(10 + (\text{shift } f.\text{—})))$ binds variable f in the body of the shift expression to $(\lambda x.(10 + x))$. The expression in the body, $(f(f\ 100))$, therefore evaluates

[19] Choice examples of mixed metaphors are quoted under "mixed metaphor" in most (printed) dictionaries of the English language.

to 120, which is returned directly to the context enclosing the reset expression because the intervening $(10 + \text{—})$ was removed when it was bound to f. The result of evaluating the entire expression[20] is 121.

A general type system for computations involving composable continuations has in general to keep track of *three* constituent data types: a type for the current expression, a type for the nearest enclosed shift, and a type for the nearest enclosing reset. The functor in Wadler's monad-like structure is therefore parametric in two types. Program composition $\cdot;\cdot$ maps $f\colon a \to (Txy)b$ and $g\colon b \to (Tyz)c$ to $(f;g)\colon a \to (Txz)c$.

As with his (chronologically much more recent) treatment of effects, this composition operation isn't categorical in general because the codomain of g is not the codomain of $f;g$. The only time they *are* the same is when $x = y$, and consequently his general mathematical structure only reduces to a monad when the two parameters of T are both fixed at some particular type x. This means that objects $(Txy)a$ for $x \neq y$ are not in the codomain of any monad, and Wadler is moved to observe that his treatment is "quite satisfactory. . . [but] *not* a monad."

There is also something suspiciously categorical in the typing of this 'composition' operation that usually doesn't reduce to a monad. The parameters of T in the general type $((a \to (Txy)b) \times (b \to (Tyz)c)) \to (a \to (Txz)c)$ follow the pattern $xy \times yz \to xz$; and *that* is the pattern of domains and codomains in the composition of arrows in a category.

(Once again, I have no suggestions to offer as to just what *is* actually going on; only an unsettled feeling that the treatment is missing something dreadfully important because its conceptual foundations are insufficiently solid.)

4 Concluding note

The original objective of this work was to relate the abstract mathematical concept of monads to the applied area of programming languages. My overall assessment is that the mechanical form of monads has inspired extensive (more-or-less ad hoc) work in programming languages, while thus far no strong relation has been demonstrated between the mathematical concept itself and the applied area.

[20] In case this example isn't confusing enough, Wadler also presents the following expression that reverses the list [1,2,3].

```
letrec perverse = (λl.if (null l)
                       then []
                       else (shift f.((head l) : (f (perverse (tail l)))))
in (reset (perverse [1, 2, 3]))
```

Acknowledgments

I wish to thank my dissertation committee for keeping after me to complete this work — without which, I'd have been missing an important piece of the Big Picture.

References

[AbSuSu96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, New York: The MIT Press, 1996. Available (as of June 2003) at URL:
`http://mitpress.mit.edu/sicp/sicp.html`

[FeFr89] Matthias Felleisen and Daniel P. Friedman, "A Syntactic Theory of Sequential State", *Theoretical Computer Science* 69 no. 3 (18 December 1989), pp. 243–287.

[JoDu93] Mark P. Jones and Luc Duponcheel, "Composing monads", Research Report YALEU/DCS/RR-1004, Yale University, December 1993. Available (as of June 2003) at URL:
`http://www.cse.ogi.edu/~mpj/pubs/composing.html`

[Kr01] Shriram Krishnamurthi, "Linguistic Reuse", Ph.D. Dissertation, Rice University, May 2001. Available (as of June 2003) at URL:
`http://www.ccs.neu.edu/scheme/pubs/`

[Ma71] Saunders Mac Lane, *Categories for the Working Mathematician*, New York: Springer-Verlag, 1971.

[MaAr86] Ernest G. Manes and Michael A. Arbib, *Algebraic Approaches to Program Semantics*, Springer-Verlag, 1986.

[Mo89] Eugenio Moggi, "Computational Lambda-Calculus and Monads", *Proceedings, Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989, pp. 14–23.

[PJWa93] S.L. Peyton Jones and P. Wadler, "Imperative functional programming", in *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993. Available (as of June 2003) at URL:
`http://www.research.avayalabs.com/user/wadler//topics/monads.html`

[Pr97] Christian Prehofer, "From Inheritance to Feature Interaction or Composing Monads", TUM-I9715, Technische Universitaet Muenchen, Institut fuer Informatik, April 1997. Available (as of June 2003) at URL:
`http://wwwbib.informatik.tu-muenchen.de/infberichte/1997/`

[SeSa99] Miley Semmelroth and Amr Sabry, "Monadic Encapsulation in ML", *SIGPLAN Notices* 34 no. 9 (September 1999) [*Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, Paris, France, 27–29 September 1999], pp. 8–17.

[Sh98] John N. Shutt, *Recursive Adaptable Grammars*, M.S. Thesis, WPI CS Department, 10 August 1993, emended 4 June 1998. Available (as of June 2003) at URL:
`http://www.cs.wpi.edu/~jshutt/thesis/top.html`

[St94] Guy L. Steele Jr., "Building Interpreters by Composing Monads", *Proceedings of the ACM Conference on Principles of Programming Languages*, 1994, pp. 472–492.

[Wa92c] Philip Wadler, "Monads for Functional Programming", in M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi* [NATO ASI Series F: Computer and system sciences, volume 118 (August 1992)], Springer-Verlag, 1992. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer-Verlag, 1995. Available (as of June 2003, with some errata fixed August 2001) at URL:
`http://www.research.avayalabs.com/user/wadler//topics/monads.html`

[Wa93] Philip Wadler, "Monads and composable continuations", *Lisp and Symbolic Computation* 7 no. 1 (January 1994) [Special issue on continuations], pp. 39–56. Available (as of June 2003) at URL:
`http://www.research.avayalabs.com/user/wadler//topics/monads.html`

[Wa99] Philip Wadler, "The marriage of effects and monads", *SIGPLAN Notices* 34 no. 1 (January 1999) [*Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, Baltimore, Maryland, 27–29 September 1998], pp. 63–74. Available (as of June 2003; including a longer version, submitted to *ACM Transactions on Computational Logic*, with coauthor Peter Thiemann) at URL:
`http://www.research.avayalabs.com/user/wadler//topics/monads.html`