Monads à la Mode

Cameron Swords Daniel P. Friedman

November 12, 2012

1 Introduction

The purpose of this article is to provide a concise introduction to monads for anyone who has an understanding of Scheme and simple types. It provides a motivation for using monads, a frank discussion of the theoretical and implementation aspects of monads, and a usage guide.

2 Motivation

Monads are an abstraction introduced by Eugenio Moggi [Moggi, 1989] as an attempt to reconcile the composition of unique extensions to the simply-typed λ-calculus (herein referred to as the λ-calculus) that would otherwise require mixing effectful operators. To provide a better understanding, we first consider two extensions to the λ-calculus and describe the problems in combining them.

2.1 On Termination

Before we begin, it is important to briefly talk about the λ-calculus in the context of termination. In short, the λ-calculus is strongly normalizing and thus non-termination is impossible as a general rule.

2.2 Continuation-Passing Style

Suppose that we would like to implement `call/cc` in the pure (side-effect free) λ-calculus. To achieve this, we first explicitly expose continuations in our programs by converting each term in our language into *continuation-passing style*.

Continuation-passing style is a set of transformations to the λ-calculus that provides access to continuations. Figure 1 presents rules for translating the λ-calculus into continuation-passing style. The transformation works by simply walking the grammar, converting each expression on the left into the corresponding expression on the right and then recurring.

After the transformation is performed on a λ-expression, continuations are explicitly exposed in the language. Adding features such as `call/cc`, `let/cc`, `throw`, and `abort` are now possible (`call/cc` is included in Figure 1 and the rest are left as an exercise for the reader); before this transformation, a language implementer would have to provide such functions without explicitly exposing continuations to the user—similar to how implementations of Scheme provide `call/cc` as an system-level feature.

2.3 Store-Passing Style

Suppose that instead of access to continuations, we would like to add a pair of operators, `ref` and `deref`, to our language. Here, `ref` places an expression into memory and returns a reference to the stored value and `deref` looks up a reference and returns the relevant value. To do this, we must expose some model of memory, introducing a *store* into the λ-calculus. This yields *store-passing style*.

Store-passing style is similar to continuation-passing style in that each expression needs access to a store. Figure 2 presents rules for this conversion—the store is passed through each expression, providing access when necessary. Using this style, we may safely encode both `ref` and `deref` without side effects. (This encoding assumes the operators `let`, `car`, `cdr`, `cons`, `reverse`, and `length`, but these are trivially implemented.)

This encoding of `ref` and `deref` works by first handing the store to the incoming expression (in case the expression needs to operate on it). It then retrieves both the (possibly) modified store and evaluated expression. Then, in the case of `ref`, a new reference is "allocated" and added to the store and the pair of this new store and the new reference is returned. In the case of `deref`, the reference is looked up in the store and the resulting value is returned.

Like `call/cc` presented above, it is possible to implement `ref` and `deref` as primitives of the language, compiled directly (which is the approach that most implementations of Scheme take). However, in languages that disallow side effects (such as Haskell), reproducing such operators *must* be done through some form of store-passing style.

2.4 Reconciling Styles

There are a great many extensions similar to the two presented here. (People seem to enjoy extending the λ-calculus—mathematicians enjoy it because the λ-calculus is well understood, formalized, and has strong ties to logic and developers enjoy it because it's easy to implement.) These extensions are varied and nuanced, but each adds a useful and unique feature to the original language.

Clearly, performing each of these transformations on the λ-calculus is rather mechanical—but what if we wish to perform both? Is it possible? Well, for these two, it might be, but the composition certainly violates the semantics of each; either each continuation mush deal with stores or each store must deal with continuations, and what should become of a store when a continuation is invoked?

This led to the sad state of affairs circa 1989: there were multiple extensions to the λ-calculus but none were composable. And yet, if we consider the case-by-case transformation of the *pure* lines (symbols, λs, and application), each looks incredibly similar in style.

In 1989, Moggi came to this realization—he demonstrated that each was analogous to the mathematic notion of a *strong monad* set forth by Anders Kock [Kock, 1972]. Moggi charac-

terized additions to the λ-calculus as either a pure translation (such as in the variable case), composition (function passing and composition), or one of these special cases that implements an added feature. He demonstrated that, if it were possible to decompose any feature's transformation into these three disjoint sets (translation, composition, and special features), it is possible to generalize these pure and compositional operations (called η and * respectively) and deal with the special features separately.

This generalization offered up, in many ways, modular language semantics. It allowed for using both continuations and stores in the language, so long as they were not mixed. Not long after, this was heralded at the programming level by Philip Wadler [Wadler, 1992], yielding monads.

3 Monads

Monads are the realized implementation of Moggi's abstraction. They offer, in Haskell, a way to use side effects, continuations, and other such *impure* operations in a purely functional, fully encapsulated fashion. Monads give us all of the dirty, effectful operations we'd like in a safe, functional world.

Monads rely on two main operators: η and *. However, before we jump into the syntax and usage of these operators, we will explore the *types* they use. Understanding these types will do more for your understanding of monads than any single other thing (including using them).

3.1 Category Theory

There is a long-running joke that to use ML, you must first fully understand type theory, and to use Haskell, you must first fully understand type theory *and* category theory. The latter comes from Haskell's eschewing effectful operations in favor of monads. This paper has neither the scope nor space to explain all of category theory. Instead, a few diagrams serve to explain enough to facilitate a discussion of the types of monads (in the study of programming languages, category theory is almost exclusively used for types).

`x`	`(λ (k) (k x))`

`(λ (x)` *body*`)`	`(λ (k)`
	`  (k (λ (x)`
	`  (λ (k) (`*body*` k)))))`

`(`*rator rand*`)`	`(λ (k)`
	`  (`*rator*` (λ (p)`
	`  (`*rand*` (λ (a)`
	`  ((p a) k))))))`

`(call/cc` *rand*`)`	`(λ (k)`
	`  (`*rand*` (λ (f)`
	`  ((f (λ (a)`
	`  (λ (k^)`
	`  (k a))))`
	`  k))))`

Figure 1: The translation of the pure λ-calculus extended with `call/cc` using continuation-passing style [Danvy and Filinski, 1992].

`x`	`(λ (s) `(,x.,s))
`(λ (x)` *body*`)`	`(λ (s)` `  `(,(λ (x) (λ (s) (`*body*` s))).,s))

`(`*rator rand*`)`	`(λ (s)`
	`  (let-pair ((p . s^) (`*rator*` s))`
	`  (let-pair ((a . s^^) (`*rand*` s^))`
	`  ((p a) s^^))))`

`(ref` *rand*`)`	`(λ (s)`
	`  (let-pair ((v . s^) (`*rand*` s))`
	`  (let ((loc (length s^)))`
	`  `(,loc.,(cons v s^))))))`

`(deref` *rand*`)`	`(λ (s)`
	`  (let-pair ((@ . s^) (`*rand*` s))`
	`  `(,(list-ref (reverse s^) @). ,s^))))`

Figure 2: The translation of the pure λ-calculus into store-passing style. A definition of `let-pair` is given in Figure 21.

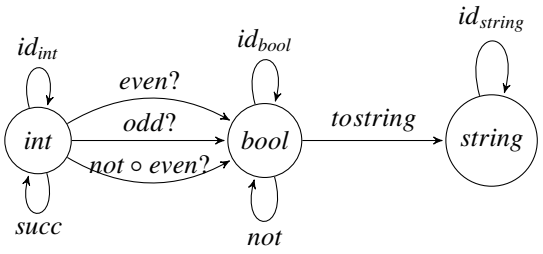

Figure 3: Types as a simple category theory diagram.

Consider, first, the diagram in Figure 3. It is a diagram of types and functions between them, allowing movement from one type to another. Here ∘ is the mathematical operator for function composition, where $(g \circ f)\ (x)$ may be written in Scheme as `(g (f x))`.

In programs, terms have types. Indeed, `even?` has type $int \rightarrow bool$. Thus the diagram does a lot toward specifying a language's semantics without providing code. And anything in this diagram *must* be composable—hence we see that *even*? may be composed with *not* to yield a *bool*. This is an example of *diagram chasing*, a method of mathematical proof where the proof is given by demonstrating a diagram showing that, given a specific starting point, one may follow a path to the desired end point. (This method is analogous to solving a system of equations, but is often easier to read.)

This leads us to our next diagram, an abstract diagram concerned with typing monads. Recall that we are using a pure language—a language with no infinite loops, no access to continuations, and no assignments. The language lacks `call/cc`, `ref`, `deref`, or any other impure operators.

Instead, the desirable operators, `ref` and `deref`, are encapsulated in the `State` monad while `call/cc` is encapsulated in the `Continuation` monad. The diagram in Figure 4 deals with the types and type transformations of a single, generic monad through η and *, two monadic operators. For now, simply remember that η provides a mechanism to move from the world of standard expressions into the world of monads and * pro-

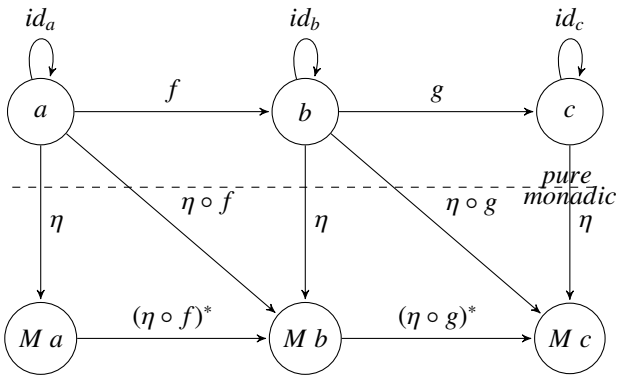

Figure 4: A category diagram demonstrating monadic composition.

vides a mechanism to use pure functions in the monadic world.

This diagram describes how η and * work. It should be clear that η takes some argument (like a) and returns an expression that inhabits the type $M\ a$, which is the monadic form of a. And, perhaps less clear, * provides a mechanism for performing composition—for taking functions from above the *monadic* line in the figure and using them below it.

3.2 Types

With Figure 4 in mind, we now qualify what it means to be of type $M\ a$. If an expression, e, has type a, we say that the expression e *evaluates* to an a—an *int* or a *bool* or some other base type. If, however, e has type $M\ a$ we say that e may do some work and then produce an a; indeed, both the store-passing and continuation-passing styles presented earlier took some data, potentially did some work (updating the store or modifying the continuation) and continued with the original expression's computation.

It should be clear, now, that η has type $a \to M\ a$; η simply takes an a and returns its monadic equivalent, yielding $M\ a$. The type of * is similarly straightforward; it takes a function of the form $a \to M\ b$ and some $M\ a$ and yields an $M\ b$. We write this as $(a \to M\ b) \to M\ a \to M\ b$.

It is worth describing how to build functions of type $a \to M\ b$. In Figure 4, f has type $a \to b$. If we compose this with η, taking the result of f and feeding it through η, then we can construct the function that expects an a and yields an $M\ b$.

It may seem alarming that * will take a function of type $a \to M\ b$ and an $M\ a$ and produce an $M\ b$, but keep in mind that the monad implementer knows exactly how to extract an a from an $M\ a$ and will write * as to perform this operation. Thus the user need never be concerned with how this process occurs.

3.3 Three Laws

Now that we have described how these monadic types interact, we can examine the properties of monads that *must* hold—they ensure that converting a, b, and c into $M\ a$, $M\ b$, and $M\ c$ maintains the original behavior. These laws are the responsibility of the monad implementer, and must be ensured when writing new monads.

1. $\eta^* = id$

   This provides the interesting property that, if we attempt to move η into the monad (via *), we simply produce the identity function.

2. $f^* \circ \eta = f$

   This provides the assurance that the two sides of any triangle in Figure 4 produce the same result as the hypotenuse. That is, $(\eta_b \circ f)^* \circ \eta_a = \eta_b \circ f$.

3. $(g^*_M \circ f_M)^* = g^*_M \circ f^*_M$

   This rule simply provides a mechanism for composing functions in the monad (f_M and g_M are functions that operate on and produce monadic values). It ensures that $(\eta \circ g)^* \circ (\eta \circ f)^*$ has, as expected, type $M\ a \to M\ c$ (by diagram chasing in Figure 4). Thus composition inside of M works the same as outside of M.

We have provided no mechanism for escaping from a monad. This helps reinforce the intended usage of monads: life in a pure world most of the time and, when it is necessary to escape into an effectful world, the entire computation must move into it. If any f_i in a series of functions, $f_1 \circ ... \circ f_n$, every single f must deal with the monad—we *must* stay below the *monadic* line.

4 Implementation

Given this rudimentary description of the formal definition of monads and their types, we now present several monads to help understand their usage and interactions. First, however, we introduce a new operator, `bind`, which serves a purpose akin to * with the single caveat that `bind` takes its arguments in reverse order (this is purely for the sake of readability).

That said, we need only implement η and `bind`. In Haskell, η is called `return` (which may be syntactically more pleasing when using the `do` syntax described below). Thus `return` has type $a \to M\ a$ and `bind` has type $M\ a \to (a \to M\ b) \to M\ b$. The rest of this section presents implementations of the identity, maybe, writer, and state monads.

4.1 Identity Monad

The identity monad provides a simple introduction to implementing monads without dealing with the complexity of further details. In Figure 5, we define the two functions required for monad usage: `return` and `bind`.

We define this as the *identity monad*—here, if something has type $M\ a$, we write it as $Id\ a$. And we define $Id\ a = a$ itself (because it is the identity). Thus, to write `return` (which also serves as η for the identity monad), we take something of type a and returns something of type $Id\ a$—thus `return` is exactly the identity function. Similarly, `bind` takes some $Id\ a$ and some f of type $a \to Id\ b$. Since we have defined $Id\ a = a$ and $Id\ b = b$, we can simply apply f to the argument `ma` (of

```
(define return-id (λ (a) a))

(define bind-id (λ (ma f) (f ma)))
```

Figure 5: Definitions of `return` and `bind` for the identity monad.

type *Id a* = *a*) and produce a *b* = *Id b*. (Here and below, `bind` will been written without currying.)

Now, for an example usage of the identity monad, we can write a simple function (such as addition) that takes two *int* arguments and return *Id int*. See Figure 6 for this implementation. It takes two arguments, both of type *int*, performs the addition, returns the result, and calls bind with a call to return to produce a value in *Id b*.

```
(define plus-id
  (λ (a b)
    (bind-id
      (return-id (+ a b))
      (λ (x) (return-id x)))))
```

Figure 6: Defining plus with the identity monad.

This provides the basic implementation of a simple monad and a single example of its usage. We now move on to defining more useful monads to demonstrate how this concept can represent possibly failing computations and stateful computations.

4.2 Maybe Monad

The *maybe monad* provides a way to return potential computations—to describe if a computation has returned nothing. For a motivating example, imagine writing `divide`—what happens when the second argument is a 0? The maybe monad provides a concise, safe solution by returning a tagged answer—either `(Nothing)`, meaning the computation failed to return anything, or `(Just a)`, where `a` is the answer. Consider, now, the definitions of `return` and `bind` in Figure 7.

To achieve further encapsulation, we also provide `fail`, a thunk that, when invoked, will fail with `(Nothing)`. It should be invoked instead of using `(Nothing)` to maintain representation independence. Given these definitions, we can define division (as in Figure 8) in such a way that, if the division is impossible (in the case of a 0), we can simply return `(Nothing)`. `divide-maybe` has a result type of *Maybe rat*. When attempting to divide by 0, the computation fails, but otherwise operates normally.

Now, what if we would like to sequence an algebraic expression such as `(/ (+ 7 8) 4)`? We must coerce this expression into the maybe monad. Figure 9 presents an encoding, using the division function in Figure 8.

```
(define return-maybe
  (λ (a)
    `(Just ,a)))

(define bind-maybe
  (λ (ma f)
    (cond
      [(eq? (car ma) 'Just) (f (cadr ma))]
      [(eq? (car ma) 'Nothing) '(Nothing)])))

(define fail
  (λ ()
    '(Nothing)))
```

Figure 7: Definitions of `return` and `bind` for the maybe monad.

```
(define divide-maybe
  (λ (a b)
    (if (zero? b)
      (fail)
      (return-maybe (/ a b)))))
```

Figure 8: Defining division with the maybe monad.

```
(bind-maybe
  (return-maybe (+ 7 8))
  (λ (x)
    (bind-maybe
      (divide-maybe x 4)
      (λ (x^)
        (return-maybe x^)))))
```

Figure 9: Encoding an algebraic expression in the maybe monad.

Here, `(return-maybe (+ 7 8))` is of type *Maybe int* and bind takes this and a function *int* → *Maybe rat*, returning a value of type *Maybe rat*. The body of the inner-most lambda simply hands this back, yielding the answer `(Just 15/4)`. Furthermore, this program provides an explicit sequencing of operations in the monad. Indeed, all monads require such explicit sequencing. This led Haskell to introduce the syntactic form `do`.

4.3 Do Syntax

Since everything in writing monads must be sequenced, it makes sense to introduce a piece of syntax that lets us specify the sequence easily—somewhat like `let*`. However, our syntax implicitly calls `bind` for us, meaning we only have to specify calls to `return`. We accomplish this with a small `syntax-rules` macro as presented in Figure 10.

This Scheme macro provides a method for building expressions such as in Figure 9. We can now rewrite that example using the macro presented here, yielding Figure 11, a concise implementation of the same program.

```
(define-syntax do
  (syntax-rules (<-)
    ((_ bind e) e)
    ((_ bind (v <- e0) e e* ...)
     (bind e0 (λ (v) (do bind e e* ...))))
    ((_ bind e0 e e* ...)
     (bind e0 (λ (_) (do bind e e* ...))))))
```

Figure 10: An implementation of the `do` macro.

```
(do bind-maybe
  (x <- (return-maybe (+ 7 8)))
  (x^ <- (divide-maybe x 4))
  (return-maybe x^))
```

Figure 11: An encoding of Figure 9 using `do` notation. The `bind` to use is the first argument to `do`.

This example provides the same functionality as in our first example of using `bind-maybe`. Here, `x` is bound to the result of placing `(+ 7 8)` inside of the maybe monad, then a call to `divide-maybe` with `x` produces `x^`. Finally, `x^` is returned (still inside the monad), completing the program and resulting in `(Just 15/4)`. The simplicity in syntax is far easier on the eyes (as opposed to the almost continuation-passing style used in Figure 9), providing a sequence of operations while relieving the explicit use of anonymous λs. Indeed, the expansion of Figure 11 is identical to the code presented in Figure 9 when the printing of `gensym` variables is suppressed.

The syntax of `do` is hopefully clearer now. In short, it works as follows: anything on the right of an arrow (`<-`) must evaluate to something of the type $M\ a$, where M matches the monad provided as the binding function in the first argument to `do`.

This syntax builds a function whose argument is bound (via a λ) to the variable that appears on the left side of the `<-`. The macro then recursively expands the rest of the `do` statement (which ultimately returns some $M\ b$) and embeds it as the body of the λ, yielding a function of type $a \to M\ b$. Thus we have an $M\ a$ and a function $a \to M\ b$, meaning that we can simply make a call to `bind`. This allows us to tersely express complex monadic terms.

4.4 Writer Monad

With this `do` syntax in mind, we now move on to provide a definition of the writer monad. The writer monad provides a system for carrying around a piece of data (in this case, a list) and from time to time writing data to it. However, the data already written may never be seen (only appended to). The implementation of the *writer monad* is presented in Figure 12. Included in this definition is `tell-writer`, a function that takes some message and writes it to the writer's buffer.

A nice application of this monad is for producing a log for an operation. Consider a program that walks a list of numbers,

```
(define return-writer
  (λ (a)
    `(,a.())))

(define bind-writer
  (λ (ma f)
    (let ((mb (f (car ma))))
      `(,(car mb) .
        ,(append (cdr ma) (cdr mb))))))

(define tell-writer
  (λ (to-write)
    `(_.(,to-write))))
```

Figure 12: A definition of the writer monad.

producing the reciprocal of each. If it encounters a 0, however, it discards the 0 and appends an error to the log (via the writer monad). See the definition of this function in Figure 13.

```
(define reciprocals
  (λ (l)
    (cond
      [(null? l) (return-writer '())]
      [(zero? (car l))
       (bind-writer
         (tell-writer "Saw␣a␣0")
         (λ (_)
           (reciprocals (cdr l))))]
      [else
       (bind-writer
         (reciprocals (cdr l))
         (λ (d)
           (return-writer
             (cons (/ 1 (car l)) d))))])))
```

Figure 13: An implementation of `reciprocals` using the writer monad.

Here we can see how the writer monad is used and how the list of writes is grown—`bind` in the writer monad takes a pair of a value to return and something to add (via `append`) to the list of writes. In the case of zero, we wish to return the natural recursion and call `tell-writer`, writing some information. Note that, because we do not care what `tell-writer` returns, we use a λ with an underscore argument and never use it.

We would like to call attention to the function call in the `zero?` case. Because `reciprocals` always returns something of type *Writer a*, wrapping a call to `return-writer` around it would double-layer the monad itself, leading to incorrect (and strange) results. Thus, we omit a call to `return` and leave the recursive call in its position. Contrast this with the call to `cons` wrapped in a call to `return-writer` in the `else` case—here, we need to return some value other than the direct recursive call, and so we must wrap our pure list in `return`.

Here we again demonstrate how our `do` operator may compress the code. Figure 14 presents this rewriting of `reciprocals`, discarding direct calls to `bind`. This allows us to remove explicit anonymous lambdas from our implementation, but it is worth noting that this syntax does little more than

move the placement (in text) of the bounded variable from the lambda argument to the left of the <-.

```
(define reciprocals
  (λ (l)
    (cond
      [(null? l) (return-writer '())]
      [(zero? (car l))
       (do bind-writer
         (tell-writer "Saw a 0")
         (reciprocals (cdr l)))]
      [else
       (do bind-writer
         (d <- (reciprocals (cdr l)))
         (return-writer
           (cons (/ 1 (car l)) d)))])))
```

Figure 14: An implementation of `reciprocals` using the writer monad.

The other piece of syntax introduced in Figure 14 is the arrow-free `do` clause. In some cases, we wish to call `bind` with an argument whose result is unimportant. One option in this case would be to write `(_ <- expression)`, using `_` to denote a value that is never used. Instead, we have a third line in the `do` macro, providing a mechanism to completely avoid this unnecessary binding.

4.5 State Monad

The last monad we consider is the *state monad*—a monad that allows us to carry a state (that directly reflects the store-passing style presented earlier). As with the two monads we have already seen, we need only provide implementations of `return` and `bind` to use it.

```
(define return-state
  (λ (a)
    (λ (s)
      `(,a.,s))))

(define bind-state
  (λ (ma f)
    (λ (s)
      (let ((vs (ma s)))
        (let ((v (car vs))
              (s^ (cdr vs)))
          ((f v) s^))))))

(define get-state
  (λ (s)
    `(,s.,s)))

(define put-state
  (λ (new-s)
    (λ (s)
      `(_.,new-s))))
```

Figure 15: A definition of the state monad.

Indeed, even the implementation of this monad looks like store-passing style—`return` resembles the variable case and `bind` is the generic abstraction of doing something with the store and returning the new one. Just like with store-passing style, the store, or *state*, gets passed into each expression, altered however necessary. When using this monad, it is important to remember that each expression is waiting to be passed the current state. Indeed, something of type *State s a* takes the state of type *s*, potentially changes the state, and then returns the expression's value paired with the new state.

Using this concept of state, we can reproduce store-passing style. Any *State s a* is a function expecting a state, and so any call to a function using the state monad must invoke the result of the function call with a starting state to retrieve the computation's result (as in `bind`).

Also notice the implementations of `get-state` and `put-state`. The former is an abstraction for retrieving the state being passed around as a pure value and the latter is a mechanism for passing a modified state back in. As with the writer monads `tell-writer`, these abstractions further encapsulate the internal representation of states.

To demonstrate these features, we implement a function to walk a list and determine if it has even length. We can accomplish this with an accumulator by passing an additional argument (initially `#t`) and inverting it once for every element we see. If, at the end, we have seen an even number of elements, we have inverted it an even number of times and produce a `#t` (or a `#f` if we have seen an odd number of elements and thus inverted it an odd number of times). Figure 16 provides an implementation.

```
(define even-length?
  (λ (l s)
    (cond
      [(null? l) s]
      [else
        (even-length? (cdr l) (not s))])))
```

Figure 16: A definition of `even-length?` using an accumulator.

However, if we have access to a state—as with the state monad—we need not pass this accumulator explicitly. Because we can change the value of the state passed around, we can implement the function in Figure 16 with minimal adjustment—as in Figure 17.

```
(define even-length?
  (λ (l)
    (cond
      [(null? l) (return-state '_)]
      [else
       (do bind-state
         (s <- get-state)
         (put-state (not s))
         (even-length? (cdr l)))])))
```

Figure 17: A definition of `even-length?` using the state monad.

This example has two interesting properties, each of which

helps better describe the usage of monads. The first interesting property deals more directly with the state monad itself. We can modify the state, as accomplished first retrieving the state with `get-state` and replacing it by the modified state with `put-state`. We provide a function expecting a state (and thus has type *State s a*) and, upon receiving the state, hands back the expected value/state pair with a modified state. In this case, we use the arrow-free syntax of `do` when calling `put-state` as we are not concerned with the result.

The second interesting property occurs in the base case of the recursion: we ensure that we return something of type *State s a* (where *s* is *bool* and *a* is *list*), but we do not care about the value of the computation itself, only the state. Thus we pass in a dummy variable to `return`. Consider the trace of a call to this program (in Figure 18) and keep in mind that anything of type *State s a* is actually a procedure waiting to be passed the state.

```
> ((even-length? '(1 2 3 4)) #t)
|(even-length? (1 2 3 4))
|#<procedure>
|(even-length? (2 3 4))
|#<procedure>
|(even-length? (3 4))
|#<procedure>
|(even-length? (4))
|#<procedure>
|(even-length? ())
|#<procedure>
(() . #t)
```

Figure 18: A call to monadic `even-length?`.

To demonstrate a larger program written using the state monad, consider writing a function that walks a (potentially deep) list of numbers, removing and counting any even number it encounters, returning a list of the form `(list . count)`. Implementing this as two separate passes over the list entails writing one function to count the even numbers and another function to remove the even numbers. This solution is presented in Figure 19.

This, however, leaves something to be desired. Using the state monad technology, it should be possible to implement it in a single pass, using the state to count the even numbers seen and the value to return the list with the even numbers removed. Using our `do` syntax, we provide the implementation concisely in Figure 20.

Here we see the principles and idioms presented throughout this section of the paper, all applied in a large, recursive function that yields the correct answer in a single pass over the data. (Also, note that a call to this program is of the form `((remberevensXcountevens list) 0)`—we must somehow seed the state, and so we start it with a 0.)

This example illustrates the real power monads provide. It would be possible to achieve equally terse results using `set!` in Scheme, but we have managed to abandon such effectful operations while producing the same results.

```
(define countevens
  (λ (l)
    (cond
      [(null? l) 0]
      [(pair? (car l))
       (+ (countevens (car l))
          (countevens (cdr l)))]
      [(or (null? (car l)) (odd? (car l)))
        (countevens (cdr l))]
      [else (add1 (countevens (cdr l)))])))

(define remberevens
  (λ (l)
    (cond
      [(null? l) '()]
      [(pair? (car l))
       (cons (remberevens (car l))
             (remberevens (cdr l)))]
      [(or (null? (car l)) (odd? (car l)))
       (cons (car l) (remberevens (cdr l)))]
      [else (remberevens (cdr l))])))

(define remberevensXcountevens
  (λ (l)
    `(,(remberevens l) . ,(countevens l))))
```

Figure 19: An implementation of `remberevensXcountevens` using two passes.

```
(define remberevensXcountevens
  (λ (l)
    (cond
      [(null? l) (return-state '())]
      [(pair? (car l))
        (do bind-state
          (a <- (remberevensXcountevens (car l)))
          (d <- (remberevensXcountevens (cdr l)))
          (return-state (cons a d)))]
      [(or (null? (car l)) (odd? (car l)))
        (do bind-state
          (d <- (remberevensXcountevens (cdr l)))
          (return-state (cons (car l) d)))]
      [else
        (do bind-state
          (s <- get-state)
          (put-state (add1 s))
          (remberevensXcountevens (cdr l)))])))
```

Figure 20: An implementation of `remberevensXcountevens` using the state monad.

5 Conclusion

In this paper, we have described the original motivation behind monads, provided a formal look at the types and type signatures of monads, and studied the implementation of four different monads, including `do` syntax and the `return` and `bind` primitives. While this has in no way covered the entirety of monads and understanding them, it has hopefully provided a working knowledge of their applications and usage.

While providing a look into monads, we have completely omitted some, including the exception monad (intended for exception and error handling), the reader monad (the mirror of the writer monad, sometimes known as the *environment* monad), the continuation monad (which exposes continua-

tions), and the list monad (which provides non-determinism).

Furthermore, we have entirely omitted the advanced topic of mixing monads. Mixing monads is subject to the same problems as mixing continuation-passing and store-passing styles—you must choose in which order they are composed. When composing the state monad with the continuation monad, you must decide which is "nested" in the other—either continuations get access to the state or the state gets access to continuations. Thus there are two ways to write the continuation-state monad that have distinctly different operational semantics.

5.1 Acknowledgements

We would like to thank Amr Sabry for providing a basis for much of the type discussion in this paper in *CSCI-B522: Programming Language Foundations* and Kyle Carter for his initial `do` macro that has been modified to appear above. We would also like to thank Zack Owens, Adam Foltzer, Kyle Carter, and Amr Sabry for all of their feedback on early drafts.

6 References and Further Reading

[Danvy and Filinski, 1992] Danvy, O. and Filinski, A. (1992). Representing control: a study of the cps transformation.

[Kock, 1972] Kock, A. (1972). Strong functors and monoidal monads. *Archiv der Math*, 23:113–120.

[Moggi, 1989] Moggi, E. (1989). Notions of computation and monads. *Information and Computation*, 93:55–92.

[Wadler, 1992] Wadler, P. (1992). The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA. ACM.

Appendix

```
(define-syntax let-pair
  (syntax-rules ()
    [(_ ((a . s) call) body)
      (let ((tmp-res call))
        (let ((a (car tmp-res))
              (s (cdr tmp-res)))
          body))]))
```

Figure 21: A macro definition for let-pair.