Topic: Monads in Scheme

By: Mohamed Sondo and Abu Butt

Class: CSC 335 programing Language Paradigms Fall 2017/CCNY

Professor: Douglas Troeger

Date: December 17, 2017

Abstract

The purpose of this paper is to discuss how to apply monads to structure functional programs in scheme. Monads maintain a convenient framework for simulating effects found in the other languages, such as global state, exception handling, output or non determinism. In this paper, we will focus in details on introducing the effects of exceptions handling when a runtime error occurs, with the help of monads in scheme program.

1. Introduction

From our research, we found that the functional programming languages can be classified into two categories. Pure and Impure functional programming languages. Language such as Haskell is a pure lambda calculus language and language such as scheme and standard ML are impure programming languages. Impure programming languages contain augment lambda calculus with a number of possible effects, such as assignment, exceptions, or continuations. Recent researches in the field of mathematics and theoretical computer science, mainly in the areas of type theory and category theory, have proposed new solutions that may be accommodate to the benefits of the impure and pure functional programming languages. The monad concept arises from category theory in mathematics, where monads are one type of a functor, which is used to map between categories. Category theory represents the mathematical structures and its concepts in terms of labeled direct graphs called category, whose nodes are called objects and whose labelled direct edges are called arrows. Whereas, a functor is a type of mapping between categories arises in the field of category theory. We call functor F as a transformation between two categories A and B. A monad is a structure that perform computation defined as sequences of steps: a type with a monad structures defines what it means to chain operators or function of the same type together. In addition, the concept of monads has been applied to structure the denotational semantics of a programming languages. Furthermore, monads are suitable for many programming operations, such as:

- Operations that perform nullable/optional values
- Operations that return a value or an error
- Operations that perform a Promise
- Operations that perform a List/Array

In each case discussed above, the code can make a decision on how to execute the rest of the code, for instance:

- Execute the rest of the code only if the value is null
- Execute the rest of the code if there was no error
- Execute the rest of the code once the promise fires
- Execute the rest of the code once for each value in the list

2. Monads in Functional Programming

In functional programming, a monad is like a design pattern that illustrates how functions, actions, inputs and outputs can be used to build generic types with following rules:

- Define a data type, and how those values are combined with the data types
- Construct functions that use the data type and compose them together into actions, by following the rules defined in the first step.

We can then visualize monad as a pair of functions, $unit_M$ and $star_M$, that contribute to do important things. A $unit_M$ and $star_M$ pair is a monad if the following monadic laws hold:

- $(unit_M\ star_M) = Identity_M$
- $((\ \mathrm{o}\ (\mathrm{star}_\mathrm{M}\ \mathrm{f}\ )\ \mathrm{unit}_\mathrm{M}) = \mathrm{f}$
- $(\mathrm{star}_\mathrm{M}\ (\mathrm{o}\ (\mathrm{star}_\mathrm{M}\ \mathrm{f}\ )\ \mathrm{g})) = (\mathrm{o}\ (\mathrm{star}_\mathrm{M}\ \mathrm{f}\ )\ (\mathrm{star}_\mathrm{M}\ \mathrm{g}))$

where o is the composition function, defined as (lambda (f g) (lambda (x) (f (g x)))), that takes two functions and compose them together. Just like how a function composition works in math, monad behave closely. This will requires us to prove that the monadic laws hold for our proposed unit_M and star_M. In order to prove these laws, we need to write our code so that given two expressions, we can quickly observe which of the two expressions occurs first in order to support monadic code style. Understanding how to write in monadic style become easy if we can recognize when a function call is (and is not) a tail call with simple arguments. Knowing that monad also behaves like a constructor, when given an underlying a set of rules assigned to a type, we then embeds a corresponding monadic type system. This monadic type system preserves all significant aspect of the underlying type system, while adding features to the monad. The usual monad has the following components:

- A *Type constructor*, that defines, for underlying type, how to obtain a corresponding a monadic type.
- A *unit function*, that injects a value in an underlying type to a value in the corresponding monadic type. The unit function has the polymorphic type t -> Mt. The result is normally the “simplest” value in the corresponding type that completely preserve the original value.
- A *binding operation* of polymorphic type (M t) -> (t->M u)-> (M u), where it is represented by the infix operator >>=. Its first value argument is a value of monadic type, whereas, the second argument is a function that maps from the underlying type of the first argument to another monadic type, and finally it result is in other monadic type.

A monad encapsulates values of a particular data type, creating a new data type associated with a specific computation, typically to handle special cases of that type. Monads provide a uniform framework, which allows new ways of thinking about programing with effects. The monads represents computations with a sequential structure. A monad is defined by a return operator that creates values, and the bind operator, which is used to link the actions in the pipelines. This process allows a programmer to design and build pipelines that could process the data in steps, in which each action is associated with additional process rules provided by the monads called monadic law or axioms. From these elements, a sequence of functions calls is composed with several bind operators chained together in an expression. When a function is called, it transform its input plain type value and then bind operator handles the returned monadic value, which is inserted into the next level in the sequences. With these elements, the program composes a sequence of function calls in pipeline with several bind operators chained together in an expression. Each function call transforms its input plain type value, and the bind operator handles the returned monadic value, which is fed into the next step in the sequence. Between each pair of composed function calls, the bind operator can inject into the monadic value some additional information that is not accessible within the function, and pass it along. It can also exert finer control of the flow of execution, for example by calling the function only under some conditions, or executing the function calls in a particular order. In general monads have also been explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time.

3. Monads Evaluation in Scheme

In pure functional language data flow are explicit and this present disadvantages: sometimes it is painfully explicit. A program in a pure functional language is written as a set of equations. Explicit data flow ensures that the value of an expression depends only on its free variables. Explicit data flow also ensures that the order of computation is irrelevant, making such programs susceptible to lazy evaluation. In pure Functional language to add error handling to it we will need to modify each recursive call to check for and handle errors appropriately. But with Scheme and others impure language with exceptions, no such restructuring would be needed. In scheme we want to interpret a monadic bind (>>=) as a ; (let ((x1 v1) (x2 v2) ...) BODY) in expression, there are a few expectations that we might have. With scheme, if we start with a definition of a recursive function f call that looks like this:

```
(define f
    (lambda  (. . . ) body ))
```

then the same function will look like this:

```
(define f
    (lambda  (unit star )
        (if  (monad? unit star )
            (letrec  ((f  (lambda  (. . . ) body* )))
        f ))))
```

Unfortunately, ensuring that unit and star from a monad requires more effort than writing a simple predicate. For now, we will guess that the user can trust unit and star to be a monad, which will simplify the definition.

```
(define f
    (lambda (unit star  )
        (letrec  ((f  (lambda  (. . . ) body* )))
    f  )))
```

We use body* to denote that the body is in monadic style. But, if instead of passing in a specific unit and star, we define them globally with unique names like unitstate and starstate , then body* looks exactly like it was before except that now specific unit s and star s are used. Making this decision allows us to use define instead of letrec to support recursion.

```
(define f  (lambda (. . . ) body* ))
```

Using global definitions is only one of several ways to package monads. We could have packaged these two functions in a cons pair. Generally, to support the illusion of an effect, there will be one or more auxiliary functions that also work well with unit and star, and as we encounter them, we will point them out. These auxiliary functions return the same kind of values as the invocation of unit .

4. Maybe Monad Design in Scheme

The *Maybe* monad represents computations where expressions can contain null values. Consider the option type (polymorphic type that represents encapsulation of an optional value) *Maybe t,* represents a value that is either a single value of type *t* or no value at all. To differentiate between these, there are two algebraic data type constructor *Just t,* containing the value *t*, or *Nothing*, containing no value. We used this type of monad as a simple sort for exception handling. For instances, a failure at any point when a program is performing a computation. If the computation fails, it causes the rest of the computations to be skipped and final result will be *Nothing*, which will be returned. On the other hand, if all the process succeed, the final result would be *Just x* for some value x.

Below are the Structure we want in scheme for writing Maybe Monad for Error handling:

Function	**Usage**
>>=	Maps to bind
return	Maps to unit
fail	Maps to fail

just *x* : creates a "just" instance of the Maybe monad

Nothing :creates a "nothing" instance of the Maybe monad

just? *X*: checks whether an object is a "just" instance of the maybe monad

nothing? *X*: checks whether an object is a "nothing" instance of the maybe monad

mapM *monad-name func lst*: The monadic equivalent of map.

Example: (**mapM** 'maybe (**lambda** (x) (**perform** 'maybe (**return** (* 2 x)))) '(1 2 3 4));Value 17:

(just (2 4 6 8))

Defines a new monad by specifying the name, the return and the bind (>>=) operations. For example, here's how the Maybe monad might be defined:

```
(define (just x)
          (list 'just x))
 (define (nothing)
           (list 'nothing))

 (define just?
          (t? 'just))
 (define nothing?
           (t? 'nothing))

 (define monad 'maybe
          just
          (lambda (m1 f)
                    (if (just? m1)
                              (f (cadr m1))
          m1)))
```

```
>>= (just 50) (lambda (x) (return (* 2 x))))  ;Value 33: (just 100)

 (>> (nothing) (>>= (just 50) (lambda (x) (return (* 2 x))))) ;Value 34: (nothing)
```

Since our paper's focus is on error handling, *Maybe* monad comes handy when a dividing a number by 0 in the example below:

```
(monad? 'maybe) => #t

;Value: safe-division
                    (define (safe-division a b)
                              (if (eq? b 0)
                                        (nothing)
                                        (just (/ a b))))
```

```
 (perform 'maybe
         (safe-division 10 5)
          (letrec ((x (safe-division 20 5))
          (y (safe-division 30 5)))
         (return (* x y)))) ;Value 15: (just 24)

(perform 'maybe
         (safe-division 10 5)
         (letrec ((x (safe-division 20 0))  ;division by zero
         (y (safe-division 30 5)))
         (return (* x y)))) ;Value 16: (nothing)
```

Another Example: Associative list

An association list, or alist, is a data structure used very frequently in Scheme. An alist is a list of pairs, each of which is called an association. The car of an association is called the key. An advantage of the alist representation is that an alist can be incrementally augmented simply by adding new entries to the front. If an alist is viewed as a mapping from keys to data, then the mapping can be not only augmented but also altered in a non-destructive manner by adding new entries to the front of the alist. Knowing that, these procedures find the first pair in alist whose car field is obj, and returns that pair. If no pair in alist has obj as its car, then #f (not the empty list) is returned. `Assq' uses `eq?' to compare obj with the car fields of the pairs in alist, this situation could be handled by maybe monad as discussed below. Maybe monad will make sure to compare exactly two objects and return the correct result.

```
(define unit_maybe
(lambda (a)
     ‘(,a . ))) ; <=  This maybe monad gets its type from the type of a.

(define star_maybe
     (lambda (sequel)
     (lambda (ma)
          (cond ; <= This is a maybe monad.
               ((eq? (cdr ma) ’_)
               (let ((a (car ma)))
                    (sequel a)))
```

```
                                (else (let ((mb ma))
                                          mb))))))
```

The tag in the cdr indicates that the pure value is in the car just as in the state monad. We immediately see that there are what appear to be extraneous aspects to this monad. If we recall, in the state monad everything was self contained; here however, things are not so clean, but since we are only concerned with $\text{unit}_{\text{maybe}}$ in the first two certification equations, and since there is an symbol that is used in $\text{unit}_{\text{maybe}}$ and a dispatch for the symbol in $\text{star}_{\text{maybe}}$, it is at least possible that the first two certification equations Hold. If you have ever used Scheme's assq, then you know what an ill-structured mess it is to always have to check for failure. The maybe monad allows the programmer to think at a higher level when handling of failure is not relevant. We Considered new-assq, which is like assq. Its job is to return a maybe monad (a pair) whose car is the cdr of the first pair in p^* whose car matches v.

```
(define new-assq
        (lambda (v p*)
                (cond
                        ((null? p*) '(_ . fail)); <= (_ . fail) is a maybe monad
                        ((eq? (caar p*) v) (unit_maybe (cdar p*)))
                        (else ((star_maybe (lambda (a) (unit_maybe a)))
                                  (new-assq v (cdr p*)))))))
```

The above code (new-assq v (cdr p^*)) is a tail recursive call, we can rewrite new-assq relying on η reduction and the first monad certification equation, leading to

```
(define new-assq
        (lambda (v p*)
                (cond
                        ((null? p*) '( . fail))
                        ((eq? (caar p*) v) (unit_maybe (cdar p*)))
                        (else (new-assq v (cdr p*))))))
```

All right-hand sides of each cond-clause must be maybe monads, of course, and they are since the only way to terminate is in the first two cond-clauses, and each is a MA.

```
((star_maybe (lambda (a) (new-assq a '((1 . 10) (2 . 20)))))
        ((lambda (ma1 ma2)
                (cond
                        ((eq? (cdr ma1) ' ) ma1)
                (else ma2)))
(new-assq 8 '((7 . 1) (9 . 3)))
(new-assq 8 '((9 . 4) (6 . 5) (8 . 2) (7 . 3)))))
```

We have to verify that the second (curried) argument to $\text{star}_{\text{maybe}}$ is a maybe monad. In either clause of the cond expression above, the result is a maybe monad. Here we are looking up 8 in two different association lists. We are then taking the pure value 2 and looking it up in a third association list. This returns the pair (20 . _). In the cond-clause when we fail, we try the other MA, but in the case where we succeed, we use the one that succeeded. The pure variable a will get bound to the pure value 2. The downside of this definition is that the first two calls to new-assq will get evaluated.

```
 ((star_maybe (lambda (a) (new-assq a '((1 . 10) (2 . 20)))))
        ((lambda (Ma1 Ma2 )
                (cond
                        ((eq? (cdr Ma1) ' ) Ma1)
                (else (Ma2))))
        (new-assq 8 '((7 . 1) (9 . 3)))
        (lambda () (new-assq 8 '((9 . 4) (6 . 5) (8 . 2) (7 . 3))))))
```

Structurally, we could have chosen #f instead of (. fail) and appropriately revised the cond-clauses, however, because we show the exception monad next, we wanted to stay with the representations.For exception handling below is a monad, where again the pure value is in the car, but this time an exception, a string, is in the cdr, though any value other than the symbol would suffice.

```
(define unitexception
        (lambda (a)
                `(,a . ))) ; ⇚ This maybe monad gets its type from the type of a.
(define starexception
        (lambda (sequel)
                (lambda (ma)
        (cond; ⇚= This is a maybe monad.
                ((eq? (cdr ma) ' )
                (let ((a (car ma)))
                (sequel a)))
        (else (let ((mb ma))
                mb)))
```

5. Conclusion

We have seen from our research that *monads* provide a useful framework that describes many different types of computation that intimate to be very different and difficult at the beginning (errors, lists and promises, state etc). Monads are a way to compose type lifting functions: *f: a=> M(b)*, *g: b => M(c)*. Monads must flatten *M(b)* to *b* before applying the function *f()*. In other words, functors are useful to map over. We also discussed in the paper how monad can useful of error checking in expression to see if failure does occur somewhere early on in a complex computation, the effect of this will cascade in a safe and predictable way.This is the true power of monad. Programing with effects.

References

Philip Walder, The essence of functional programing, University of Glasgow, 2017

[http://www.cse.chalmers.se/edu/year/2017/course/pfp/Papers/monadsWadler.pdf].

Philip Walder, Monads for functional programing, University of Glasgow, 2017

[http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf].

Cameron Swords and Daniel P. Friedman, Mondas a la Mode, 2012

[https://www.cs.indiana.edu/~cswords/monads.pdf]

Taesoo Kim, In search of a Monad for system call abstractions, MIT CSAIL

[https://ocw.mit.edu/courses/mathematics/18-s996-category-theory-for-scientists-spring-2013/projects/MIT18_S996S13_Monad.pdf].