The City College of New York

CSC 33500 Programming Language Paradigms

Fall 2017

Professor Douglas Troeger

Topic: Introduction to Monads

Group Members: Shirong Zheng & Tongzheng Zeng

December 18, 2017

Table of Contents

1. Introduction

The purpose of this project report is to provide a brief overview of functional programing and a detailed explanation of monad in scheme, and to showcase an interesting application which demonstrates the way that backtracking can be used to solve some similar problem. In this paper we will present some background information on how monad works in haskell. Second step we will show how it works in scheme. Then compare the difference and similar in each other. We will also give a brief explanation why monad is a useful solution. The goal of our project is to implement a scheme language system that performs pattern matching on monad.

2. Background

2.1 Overview Of Monads

Monads are very useful in Haskell, but the concept is often difficult to grasp at first. Since monads have so many applications, people often explain them from a particular point of view, which can derail your efforts towards understanding them in their full glory.

Historically, monads were introduced into Haskell to perform input and output – that is, I/O operations of the sort we dealt with in the Simple input and output chapter and the prologue to this unit. A predetermined execution order is crucial for things like reading and writing files, and monadic operations lend themselves naturally to sequencing. However, monads are by no means limited to input and output. They can be used to provide a whole range of features, such as exceptions, state, non-determinism, continuations, coroutines, and more. In fact, thanks to the versatility of monads, none of these constructs needed to be built into Haskell as a language; rather, they are defined by the standard libraries.

2.2 What is Monad

In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together. In functional programming, a monad is a design pattern that defines how functions, actions, inputs, and outputs can be used together to build generic types,with the following organization:

1. Define a data type, and how values of that data type are combined.
2. Create functions that use the data type, and compose them together into actions, following the rules defined in the first step.

We know that a pure function is a good thing that always produces the same output for the same input, denoted as:

```
a -> c
```

Pure function has no side effects, that is, it does not modify the data outside its own code. So, when our program is doing IO operations, we can not write IO codes separately in everywhere of the program, because the data modified by these IO codes is not modeled by our code. Think input as a, output as b. You can make the code to modify the IO data into a function of the input and output, so that our internal procedures can be keep pure. Functional programming make many small functions into a big function to produce whole program.

Let's discuss monad in simple way:

For example, In case we wish to write a function as below:

```
a -> c
```

Right now we already have two small functions,

```
a -> b
b -> c
```

Then glue them together as one function

```
(a -> b) -> (b -> c) -> (a -> c)
```

This solution could best represent the **monad.** Monad is the strong glue function which glue two functions together. Not only we can glue small functions but we can glue them together and connect their heads into an input stream then connects their tails into an output stream to form the final IO function. Monad has different class like IO, Maybe.

The a is input, that we turn it into c, and then set it inside of M .

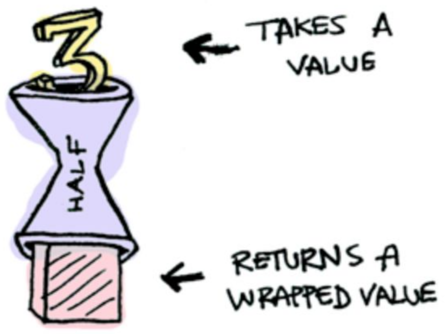
```
a -> M c
```

The below is the Monad's bind function in Haskell. It could miss "a" in the function.

```
(a -> M b) -> (b -> M c) -> (a -> M c)
```

```
M b -> (b -> M c) -> M c
```

As mentioned earlier, the return function is implemented exactly like the pure function, just as the "Just function". The function is to put a value x in the Maybe box and change to Just x. Similarly, for the implementation of (>> =) function, we need to separately deal with the two cases of the Maybe context, and return a new empty box directly when the box is empty; when the box is not empty, that is, Just x , Then x is removed and the func function is applied directly to x, and we know that the result of func x is a value in context. Let's look at a specific example below. Let's define a half function, which takes a number x as the argument, divides x by 2 if x is even and places the result in the Maybe box, or an empty box if x is not even:



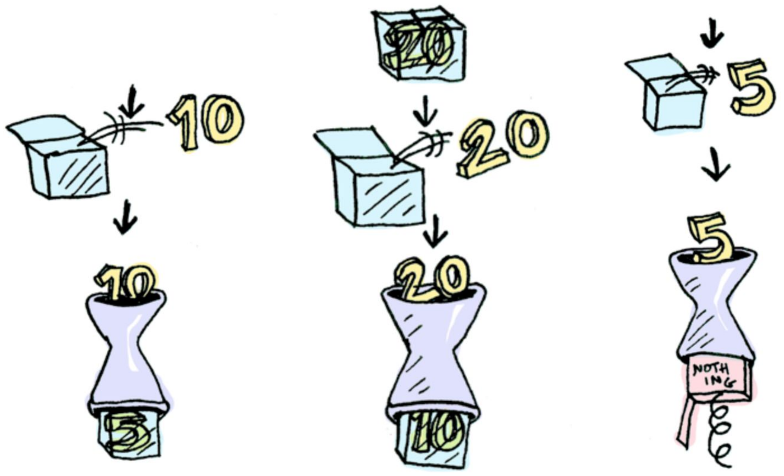
```
half x = if even x
    then Just (x `div` 2)
    else Nothing
```

Next, we apply the half function to Just 20 using the (>> =) function, assuming that the result y is obtained; then continue using the (>> =) function to apply the half function to the result y of the previous step, and so on What kind of result will be:

```
ghci> Just 20 >>= half
Just 10
ghci> Just 10 >>= half
Just 5
ghci> Just 5 >>= half
Nothing
```

The output from the previous step is used as the input to the next step, and the process goes on indefinitely if you so desire. I think you may have thought of it, yes, it is a **chain operation**.

All the operations are linked together like a production line. Each step of the operation is to process the input and produce the output. The whole operation can be regarded as the process of processing the original raw material Just 20 and finally producing the finished product Nothing:



```
ghci> Just 20 >>= half >>= half >>= half
Nothing
```

Note: Chained operations are just one of the major benefits that Monad brings to us; another major benefit not covered here is that Monad handles contexts automatically for us, and we only need to be concerned with real values.

A common construct in programming languages is the "functor." Maybe in Haskell is a good example:

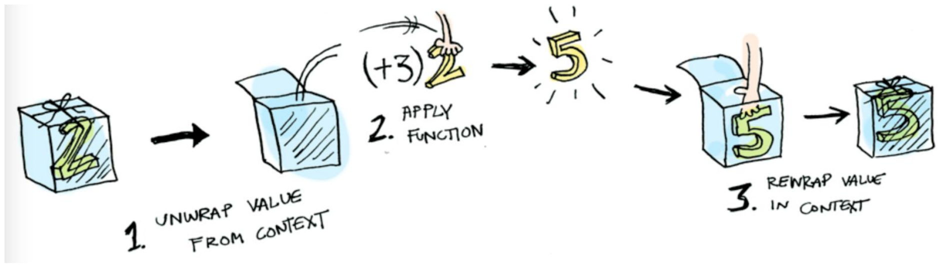
```
data Maybe a = Just a | Nothing
```

That declaration defines a type, Maybe a, which is parameterized by a type variable a, which just means that you can use it with any type in place of a. Maybe is a functor because it could through fmap (functor map) then put all function into "Maybe Space".

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

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

When we have a function of type a -> b and some data type f a, how do we map that function over the data type to end up with f b? Then we going to use fmap. The internal of fmap is like this: Open the encapsulated data type, remove the value, after the value of function processing, then encapsulated back to the data type.



The definition of fmap is:

```haskell
fmap :: (a -> b) -> (Maybe a -> Maybe b)
fmap f mbx = case mbx of
  Just x  -> Just (f x)
  Nothing -> Nothing
```

Maybe can easily be used for error handling. For functions that do not support Maybe as input, we can also be compatible with fmap. However, it is cumbersome to combine multiple functions that produce Maybe. For example, this division function:

```haskell
safeDiv :: Int -> Int -> Maybe Int
  safeDiv _ 0 = Nothing
  safeDiv x y = Just (x / y)
```

Consider the expression:

```haskell
case safeDiv a b of
  Nothing -> Nothing
  Just x  -> case safeDiv c x of
    Nothing -> Nothing
    Just y  -> case safeDiv d y of
      Nothing -> Nothing
      Just z  -> safeDiv e z
```

2.1.1 Definition

A monad is defined by three things:

i. a type constructor m;

ii. a function return;

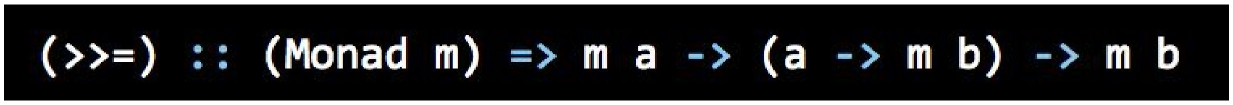
iii. an operator (>>=) which is pronounced "bind".

The function and operator are methods of the Monad type class and have types

```
return :: a -> m a
 (>>=)  :: m a -> (a -> m b) -> m b
```

and are required to obey three laws that will be explained later on. A monad represents some policy for chaining computations. Identity's policy is pure function composition, Maybe's policy is function composition with failure propagation, IO's policy is impure function composition and so on.

Monads are a natural extension of applicative functors and with them we're concerned with this: if you have a value with a context, m a, how do you apply to it a function that takes a normal a and returns a value with a context? That is, how do you apply a function of type a -> m b to a value of type m a? So essentially, we will want this function:



```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

2.2.2 Common monads

Most common applications of monads include:

- Representing failure using Maybe monad
- Nondeterminism using List monad to represent carrying multiple values
- State using State monad
- Read-only environment using Reader monad
- I/O using IO monad

2.2.3 Monad class

Monads can be viewed as a standard programming interface to various data or control structures, which is captured by the Monad class. All common monads are members of it:

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

In addition to implementing the class functions, all instances of Monad should obey the following equations, or Monad Laws:

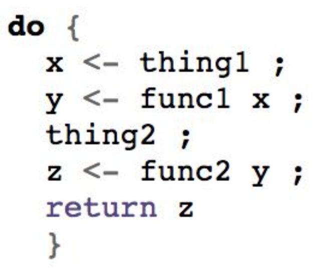
```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

2.2.4 Do-notation

In order to improve the look of code that uses monads Haskell provides a special syntactic sugar called do-notation. Code written using do-notation is transformed by the compiler to ordinary expressions that use the functions from the Monad class (i.e. the two varieties of bind, >>= and >>). When using do-notation and a monad like State or IO programs look very much like programs written in an imperative language as each line contains a statement that can change the simulated global state of the program and optionally binds a (local) variable that can be used by the statements later in the code block. It is possible to intermix the do-notation with regular notation. For example, the following expression:

```
thing1  >>= \x ->
func1 x >>= \y ->
thing2  >>= \_ ->
func2 y >>= \z ->
return z
```

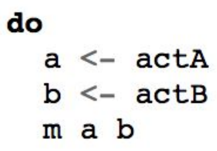
This can also be written using the do-notation as follows:



```
do {
  x <- thing1 ;
  y <- func1 x ;
  thing2 ;
  z <- func2 y ;
  return z
  }
```

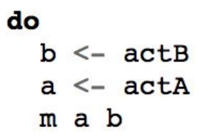
2.2.5 Commutative monads

Commutative monads are monads for which the order of actions makes no difference (they commute), that is when following code:



```
do
  a <- actA
  b <- actB
  m a b
```

is same with

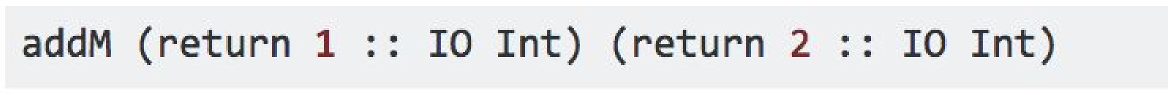


```
do
  b <- actB
  a <- actA
  m a b
```

2.3 Side-Effects

Why did Haskell introduce Monad? The biggest reason is to introduce side effects in pure function. Pure function has the advantage of safety and reliability. Function output depends entirely on the input, there is no implicit dependence. Its existence is perfect as a mathematical formula. Sometimes the more perfect has the more useless things, such as pure function, because isolated from the external environment, pure functions even cannot implement basic input and output. A simple Hello World can baffle pure functions. In order to introduce IO operation. Monad make IO operations with side effects into Haskell in a controlled way, then allowing purely functional and IO operations to get along peacefully into organize safe and useful programs.

First apply addM to two values in the IO monad



```
addM (return 1 :: IO Int) (return 2 :: IO Int)
```

The output just 3 in the IO monad. addM does not read or write any mutable state. Same goes for the State or ST monads. Let's use a different function:

```
fireTheMissiles :: IO Int  -- returns the number of casualties
```

Clearly the world will be different each time missiles are fired. Clearly. Now suppose, you're trying to write some totally innocuous, side effect free, non-missile-firing code. Perhaps you're trying once again to add two numbers, but this time without any monads flying around:

```
add :: Num a => a -> a -> a
add a b = a + b
```

and all of a sudden your hand slips, and you accidentally typo:

```
add a b = a + b + fireTheMissiles
```

An honest mistake, really. The keys were so close together. Fortunately, because fireTheMissiles was of type IO Int rather than simply Int, the compiler is able to avert disaster.

It is totally contrived example, but the point is that in the case of IO, ST and friends, the type system keeps effects isolated to some specific context. It doesn't magically eliminate side effects, making code referentially transparent that shouldn't be, but it does make it clear at compile time what scope the effects are limited to.

So getting back to the original point: what does this have to do with chaining or composition of functions? Well, in this case, it's just a handy way of expressing a sequence of effects:

```
fireTheMissilesTwice :: IO ()
fireTheMissilesTwice = do
    a <- fireTheMissiles
    print a
    b <- fireTheMissiles
    print b
```

There is one particular monad which called IO, that has something on it which we can use to get side effects. For example, we suppose there is a type called "World", which contains all the state of the external universe in it: it's got a copy of the contents of the hard drive, and a list of keypresses from the keyboard, and the current contents of the screen buffer, and the Internet connection, and everything else. Here's how to think about what IO does:

```
type IO t = World -> (t, World)
```

In other words, IO t is a function which takes a World and returns the t it's supposed to contain, along with a new, updated World formed by modifying the original one in the process of getting the t. This isn't actually how it's implemented, of course (where would you store the extra copies of the contents of your hard drive that each of the Worlds contains?), but it's a way of thinking about how this all works. Here's the monad implementation for our IO:

```
instance Monad IO where
  return x world = (x, world)
  (ioX >>= f) world0 =
    let
      (x, world1) = ioX world0
    in
      f x world1  -- Has type (t, World)
```

Because Haskell functions are curried, the return function takes an argument called x and gives back a function that takes a World and returns x along with the "new, updated" World formed by not modifying the World it was given. The implementation of bind is a bit trickier: the expression (ioX >>= f) has type World -> (t, World). This is to say, it's a function that takes a World, called world0, which it uses to extract x from its IO monad. This gets passed to f, resulting in another IO monad, which again is a function that takes a World and returns a t and a new, updated World. We give it the World we got back from getting x out of its monad, and the thing it gives back to us is the t with a final version of the World.

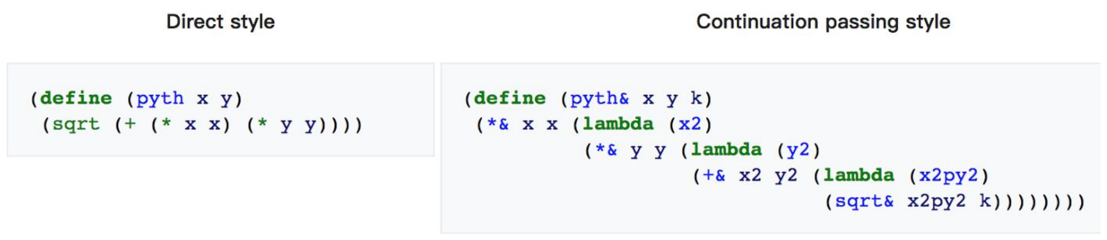
This IO appears to act like a monad, and we can see how it can modify the World it is given and give back an updated one. This main is given the initial World to start everything off, and it passes the updated ones from each IO to the next. If you have an IO that is not reachable from main, it will never be executed, and it doesn’t get a World passed to it. If, instead, you’re using GHCI to run your Haskell commands, everything is wrapped in an implicit IO, since the results get printed out to the screen. Every time you give it a new command, it passes in the current World, gets the result of your command back, calls print on it (which updates the World by modifying the contents of the screen or the list of defined variables or the list of loaded modules or whatever), and then saves the new World to give to the next command. The neat thing about this is that there’s only 1 World in existence at any given moment. Each IO takes that one and only World, consumes it, and gives back a single new World. Consequently, there’s no way to accidentally run out of Worlds, or have multiple ones running around.

3. Monads In Scheme

In this part, let’s introduce monads from a Scheme perspective. It will go through every step by the concept continuations, CPS, accumulators, and accumulator-passing style. For the main insight of monads is that all side effect, the common thing is order of evaluation matters from mutation to IO then to non-termination. In monads, terminating, pure lambda expressions, the order of evaluation is completely irrelevant. The final result is the same with no observable differences, no matter how you reduce it. But when you have side effects, they have to happen in the right order. (Monads aren't the only formalism for dealing with this -- CPS and A-normal form do, too. But they're all related.)

1. Continuation-Passing Style

Continuation-Passing Style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. This is contrasted with direct style, which is the usual style of programming. In CPS, each procedure takes an extra argument representing what should be done with the result the function is calculating. This, along with a restrictive style prohibiting a variety of constructs usually available, is used to expose the semantics of programs, making them easier to analyze. This style also makes it easy to express unusual control structures, like catch/throw or other non-local transfers of control. The key to CPS is to remember that (a) every function takes an extra argument, its continuation, and (b) every argument in a function call must be either a variable or a lambda expression (not a more complex expression). This has the effect of turning expressions "inside-out" because the innermost parts of the expression must be evaluated first, so CPS explicates the order of evaluation as well as the control flow. Some examples of code in direct style and the corresponding CPS appear below. These examples are written in the Scheme programming language; by convention the continuation function is represented as a parameter named "k":



Direct style

```
(define (pyth x y)
 (sqrt (+ (* x x) (* y y))))
```

Continuation passing style

```
(define (pyth& x y k)
 (*& x x (lambda (x2)
          (*& y y (lambda (y2)
                   (+& x2 y2 (lambda (x2py2)
                              (sqrt& x2py2 k))))))))
```

It will talking about some effects context in the pure semantics. We try to use pure scheme subset to implement the impure problem as below. We must define BEGIN to evaluate both its arguments , we'll just stick to a two-argument definition and evaluate to the value of the second.

```
            Impure Problem                    Pure Scheme Subset

(begin (turn-on-safety!)
       (pull-trigger!))         (define (begin v1 v2) v2)
```

The first is a pure scheme might evaluate the program in arbitrary order (but it is a bad way). In second, let's CPS our programs so we can enforce the evaluation order (I'm using the infix [[--]] to represent the CPS-translation of an expression):

```
   (begin (turn-on-safety!)
          (pull-trigger!))
-> (begin (turn-on-safety!)  ; effect: pull-trigger!
          #<void>)
-> (begin #<void>            ; effect: turn-on-safety!
          #<void>)
-> #<void>
```

```
 [[(begin (turn-on-safety!)
          (pull-trigger!))]]
= (lambda (k)
    ([[turn-on-safety!]] (lambda (res1)
                           ([[pull-trigger!]] (lambda (res2)
                                                (k res2))))))
```

So in a program written in CPS, we can define BEGIN in below function. That the result of the first expression is received in the argument res1, and subsequently ignored (no subexpression refers to res1).

```
(define (begin cps-exp1 cps-exp2)
  (lambda (k)
    (cps-exp1 (lambda (res1)
                (cps-exp2 (lambda (res2)
                            (k res2)))))))
```

Here is a more complex source example, the map function that takes a function and a list and produces a new list by applying the function to each element of the given list:

```
(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))
```

The CPS conversion proceeds as follows:

- Add a k argument to map2 :

```
(define (map2 f l k)
  (if (null? l)
      '()
      (cons (f (car l))(map2 f (cdr l)))))
```

- Pass results to k instead of returning them directly:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (k (cons (f (car l))(map2 f (cdr l))))))
```

- Lift the (f (car l)) nested call:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (f (car l)(lambda(v)(k (cons v (map2 f (cdr l))))))))
```

- Lift the (map2 f (cdr l)) nested call:

```
(define (map2 f l k)
  (if (null? l)
      (k '())
      (f (car l)(lambda (v) (map2 f (cdr l)(lambda (v2) (k (cons v v2))))))))
```

2. Accumulator-Passing Style

Accumulator passing style offers one approach to structuring a recursive algorithm so that it employs tail calls in situations where the easiest possible implementation would not do so. Let’s write programs in accumulator-passing style, where all procedures have to take an extra argument that represents a "register" that's being updated as a computation proceeds. Consider a little random-number generator:

```
(define seed (current-time))

(define (rand)
  (let ([ans (modulo (* seed 16807) 2147483647)])
    (begin (set! seed ans)
           ans)))
```

If we were to implement this in pure Scheme, we'd need to pass around the seed as an extra argument through any procedures that might update its value. We'd also have all our procedures return a pair of values: the actual result of the procedure, plus the new seed, in case it got updated during the computation of the procedure's result. Each one of these procedures has a couple features in common, namely that they take a "seed" parameter and return a pair consisting of their result and the new seed. Let's start abstracting that out by currying the seed parameter. At every application of RAND, it updates the seed to the newly generated random number.

```
;; rand : -> (number -> (number x number))
(define (rand)
  (lambda (seed)
    (let ([ans (modulo (* seed 16807) 2147483647)])
      (cons ans ans))))

;; rand-point : -> (number -> (point x number))
(define (rand-point)
  (lambda (seed)
    (let* ([r1 ((rand) seed)]
           [r2 ((rand) (cdr r1))]
           [r3 ((rand) (cdr r2))])
      (cons (make-point (car r1) (car r2))
            (cdr r2)))))

;; rand-segment : -> (number -> (segment x number))
(define (rand-segment)
  (lambda (seed)
    (let* ([r1 ((rand-point) seed)]
           [r2 ((rand-point) (cdr r1))])
      (cons (make-segment (car r1) (car r2))
            (cdr r2)))))
```

The whole program would start with an initial seed like so:

```
(run-my-program (current-time))
```

We call the function that can have side effects as "operations," however the function without side effect is "pure function". For example, the below is distance function which is pure because it doesn’t change seed. It doesn’t use extra parameter to represent current seed, also it doesn’t return pair. Then this function could use and recall by any side effects function. But this distance function cannot use these function. Because it will cause accumulated seed pass to next operation.

```
(define (distance pt1 pt2)
  (sqrt (+ (sqr (- (point-x pt1) (point-x pt2)))
           (sqr (- (point-y pt1) (point-y pt2)))
           (sqr (- (point-z pt1) (point-z pt2))))))
```

We can set a pair of operations to extract and assign seed values:

```
;; get-seed : -> (number -> (number x number))
(define (get-seed)
  (lambda (seed)
    (cons seed seed)))

;; set-seed : number -> (number -> (void x number))
(define (set-seed new)
  (lambda (old)
    (cons (void) new)))
```

That an operation that returns a value of type alpha has type T(alpha)

```
T(alpha) = number -> (alpha x number)
```

```
get-seed     : -> T(number)
set-seed     : number -> T(void)
rand         : -> T(number)
rand-point   : -> T(point)
rand-segment : -> T(segment)
```

Let’s Define the BEGIN, but it use accumulator to calculate order.

```
;; begin : T(alpha) T(beta) -> T(beta)
(define (begin comp1 comp2)
  (lambda (seed0)
    (let* ([res1 (comp1 seed0)]
           [val1 (car res1)]
           [seed1 (cdr res1)])
      (comp2 seed1))))
```

Let's write a new combinator to make the result of the first operation can be used by the second operation. This new combinator takes an operation and a function that receives the result of the first operation and constructs a second operation based on the result of the first. Finally, it runs the second operation.

```
;; pipe : T(alpha) (alpha -> T(beta)) -> T(beta)
(define (pipe comp1 build-comp2)
  (lambda (seed0)
    (let* ([res1 (comp1 seed0)]
           [val1 (car res1)]
           [seed1 (cdr res1)])
      ((build-comp2 val1) seed1))))
```

```
(define (rand)
  (pipe (get-seed)
        (lambda (seed)
          (let ([ans (modulo (* seed 16807) 2147483647)])
            (begin (set-seed ans)
                   (lambda (seed)
                     (cons ans ans)))))))
```

Then we could get find RAND function

```
(define (rand)
  (pipe (get-seed)
        (lambda (seed)
          (let ([ans (modulo (* seed 16807) 2147483647)])
            (begin (set-seed ans)
                   (lift ans))))))
```

4. Implement Scheme Code

```
;;--type Numbered a = (Int,a)

(define (make-numbered-value tag val) (cons tag val))

(define (nvalue-tag tv) (car tv))

(define (nvalue-val tv) (cdr tv))


;-- return:: a -> NumberedM a

(define (return val)

  (lambda (curr_counter)

    (make-numbered-value curr_counter val)))


(define incr

  (lambda (n)

    (make-numbered-value (+ 1 n) n)))


;for bind, -- (>>=):: NumberedM a -> (a -> NumberedM b) -> NumberedM b

(define (>>= m f)

  (lambda (curr_counter)

    (let* ((m_result (m curr_counter))

      (n1 (nvalue-tag m_result))        ; result of the delayed computation

      (v  (nvalue-val m_result))        ; represented by m

      (m1 (f v)))                       ; feed the result to f, get another m1

   (m1 n1))))                           ; The result of the bigger monad
```

```
(define (make-node val kids)
  (>>= incr
       (lambda (counter)
         (return (cons (make-numbered-value counter val) kids)))))


;try to produces a tagged binary tree
(define (build-btree-r depth)
  (if (zero? depth) (make-node depth '())
    (>>=
      (build-btree-r (- depth 1))
      (lambda (left-branch)
        (>>= (build-btree-r (- depth 1))
             (lambda (right-branch)
               (make-node depth (list left-branch right-branch))))))))


(define (runM m init-counter) (m init-counter))


; (letM ((name initializer)) expression) ==> >>= initializer (lambda (name) expression))
;regular let:
;  (let ((name initializer)) body) ==>(apply (lambda (name) body) (list initializer))
(define-macro letM
       (lambda (binding expr)
         (apply
          (lambda (name-val)
            (apply (lambda (name initializer)
                `(>>= ,initializer (lambda (,name) ,expr)))
               name-val))
          binding)))


(define-macro letM*
      (lambda (bindings expr)
        (if (and (pair? bindings) (pair? (cdr bindings)))
            `(letM ,(list (car bindings))
                   (letM* ,(cdr bindings) ,expr))
            `(letM ,bindings ,expr))))


;Each node of the tree is uniquely tagged
(define (build-btree depth)
      (if (zero? depth) (make-node depth '())
          (let*  ((left-branch (build-btree (- depth 1)))
                 (right-branch (build-btree (- depth 1))))
                (make-node depth (list left-branch right-branch)))))


 ;constructs a regular, non-tagged full binary tree:
 (define (build-btree-c depth)
      (if (zero? depth) (cons depth '())
          (let ((left-branch (build-btree-c (- depth 1)))
                (right-branch (build-btree-c (- depth 1))))
                (cons depth (list left-branch right-branch)))))
```

```
;=========
;for test:
; (runM (build-btree 3) 100)
; (build-btree-c 3))
```

5. Conclusion

As evidenced through the results, the goal of the project was successfully accomplished. Several functions were created to prove and explain the topic in Monad. It was not without its shortcomings. The whole paper started from the introduction of Monad in Haskell, then introduce the site effect. Finally it goes to the Scheme flow and implement code. In the end, we learned a lots knowledge about Monad in functional programming through this project.

6. Reference

“Monadic Programming in Scheme” n.p. N.d. Web. 16 December, 2017

http://okmij.org/ftp/Scheme/monad-in-Scheme.html

Herman, Dave “A Schemer's Introduction to Monads” n.p. 13 July, 2004 Web. 16 December, 2017

http://www.ccs.neu.edu/home/dherman/research/tutorials/monads-for-schemers.txt

“Continuation-Passing Style” n.p. Spring 2002 Web. 17 December,2017

https://www.cs.utah.edu/~mflatt/past-courses/cs6520/public_html/s02/cps.pdf

“A Fistful of Monads” n.p. N.d. Web. 17 December, 2017

http://learnyouahaskell.com/a-fistful-of-monads#getting-our-feet-wet-with-maybe

Davidson, Alan “Monads and Side Effects in Haskell” n.p. 2 October, 2013 Web. 17 December 2017

https://www.cs.hmc.edu/~adavidso/monads.pdf