Monads: Implementation towards understanding

Ryan Domigan

> "Formally, a monad consists of a type constructor M and two operations, bind and return." - wikipedia

Introduction

Monads have entered the programming zeitgeist; carried form academia to the diaspora by undergraduates who have limited time and partial mastery. Monads are not amenable to quick explanations, and the slightly confused explanations appearing on stackoverflow and r/programming make them seem magical; a key to secret insights which the less adept are left commenting speculatively on - in the textual equivalent of hushed tones.

The trouble isn't that monads are terribly complex, but rather they must be understood as a whole. Analogies are all more or less inaccurate, and no other quick explanation presents itself. If you have some time though, monads are fairly simple, and fairly useful.

Monad basics

Why

Dealing with context in a purely functional language can be inconvenient. Monads make it easier.

Taking a simple interpreter as an example; evaluating an AST requires things like local bindings (which can change over time), an alphabet of free variables and so on. I'll call this the auxiliary state. When evaluating an multi-argument function, the auxiliary state may change with the evaluation of each argument, so it is not only necessary to pass along *a* state, but to update and pass along the *right* state.

Without monads each function dealing with the AST must do this explicitly.

```
apply map ids (Apply left right)
  = Apply left' right'
    where left' = apply map ids left
          right' = if' (left' == left)
                       (apply map ids
                           right)
                       right
```

Fig0: from λ-calculus interpreter without monads

```
apply (Apply left right)
  = apply left >>=
    (\lhs -> apply right >>=
             return . (Apply lhs))
```

Fig1: same function using a State monad.

While neither figure represents an ideal implementation, it is at least clear that Fig0 must accept and pass on two additional parameters, *map* and *ids*, while Fig1 does not. It is also the case that in both $left$ will be evaluated before *right*. In Fig0 this is a side effect of the $left' == left$ test which $right'$ depends on. In Fig1 it is a side effect of the bind. Due to haskell's laziness, the order of evaluation is difficult to infer. Although not a concern in this case, ordering is important when building up variable bindings over something like a $let*$ or while performing IO.

OK, so what are they?

Applicative Functors

The basic Monad concept is that there is some container type, and a set of functions to manipulate data in that container without needing to unpack it. This is not a novel concept; the function *map* does the same thing in Haskell and Scheme. When writing a function for map you don't need to worry about the *container*, just the *data*, and in fact any function you could use with map will work with Set.map or set-map (from Haskell and Racket respectively).

Haskell takes the *map* idea and generalizes it with $fmap$. Over a list, fmap does exactly what map would do. Over a $Maybe$, it will apply the function f to d such that $Justd \Rightarrow Just(fd)$, or if given $Nothing$ it returns $Nothing$ (just like map returns the empty list when it gets an empty list). Because we are dealing with mathematicians and not normal people, $fmap$ gets it's own infix operator $< \$ >$.

This is enough to carry into Monads: there is stuff in a container, and by using something like *map* you can interact the stuff without unpacking the container.

Monads

Like applicatives, monads deal with containers functions and data. Unlike applicatives, monads *can* interact with the container. Monads get this functionality by implementing bind and return, all the other forms (*join*, >>, *do* and so on) are implemented in terms of these two primitives. Bind (>>=) instead of $fmap$ is the fundamental data-inna-box manipulator of monads.

Bind

Conceptually, Bind takes the data part of a monad, puts it into a function, then combines the function's monadic result with the context from which the data came. Call it combining the old context and the new context. The combining can be done in whatever way is most useful to the programmer, and so long as it complies to the Monad Laws (http://www.haskell.org/haskellwiki/Monad_laws), all the derived forms will work as expected.

The simplest monad to work with is Maybe; maybe models 'fail' and 'success' (*Nothing* and *Just*). Any failure fails the whole expression; so combining old and new is straight forward. If either old or new is Nothing the result is Nothing, if both are Just, the result is Just result.

```
foo :: Int -> Maybe Int
foo n = Just (n + 1)
```

```
Just 1 >>= foo
```

$\Rightarrow 2$

```
Just 1 >>= foo >>= foo
```

$\Rightarrow 3$

```
Nothing >>= foo >>= foo
```

$\Rightarrow Nothing$

Lists are used to model non-determinism in haskell, so it makes sense to combine the old and new contexts by appending the new to old, giving the set of all possible results:

```
bar :: Int -> [Int]
bar n = [n + 1,n - 1]
```

```
[1,2] >>= bar
```

$\Rightarrow [2, 0, 3, 1]$

```
[1,2] >>= bar >>= bar
```

$\Rightarrow [3, 1, 1, -1, 4, 2, 2, 0]$

Rad.

return

Return puts a datum into its minimal context. What this means depends on the monad involved, in the case of maybe, *return*2 is the same as *Just*2, for a list, *return*2 is [2]. The 'minimal context' is similar to an identity function with respect to bind: combine 'minimal context' with context M, and you get M.

return 2 >>= bar >>= bar

⇒ [4, 2, 2, 0]

return 3 >> **return** 2 >>= bar >>= bar

⇒ [4, 2, 2, 0]

(>> ignores the data part and updates context only)
Notice that the minimal context of '3' has no effect on the context of '2'.

return 2 >>= foo >> **return** 1 >>= foo

⇒ *Just*2

Nothing >>= foo >> **return** 2 >>= foo

⇒ *Nothing*
Notice here how *return* ends up replacing the value in the monad (staying in Just), but doesn't replace the context.

Implementation

Haskell provides a State monad for generic contexts, it will carry whatever context you want to put in.
A slightly simplified and specialized version, the R monad, is provided at http://www.haskell.org/tutorial/monads.html. This state monad carries an int as it's 'Resource', thus the name R. Lets look at an implementation of R in scheme (because, reasons.)

First, R is a simple struct. It's (only) data field is a function which takes a resource, and returns a tuple of some data and a context. Don't worry too much about it just yet.

```
(define (return v)
  (R (lambda (resource)
       (cons resource (Left v)))))
```

Onward to bind, this is the crux of the implementation, so we'll break it down a little:

```
1 (define (>>= state fn)
2   (R (lambda (resource)
3        (match ((R-vv state) resource)
4          [(cons rr (Left vv)) ((R-vv (fn vv)) rr)]
5          [(cons rr (Right vv)) (cons rr (Right (>>= vv fn)))]
6          ))))
```

Bind generates a new R struct, puts a function fnR in its data field; ok that's line 1-2. The arguments *state* and fn are captured, and we're waiting for a resource. When fnR is applied to a number, the number is captured as *resource* and *state* is applied to it. This produces a pair in the form ($resource'.Either\, value$) (line 3), which is necessarily the same as the return type of fnR.

On Left (line 4), we apply our captured fn to this new *value*. The idea is to create a two step process, the first step is to feed a *resource* to the captured state (evaluating the left-hand-side of bind), this gives you a two part result. Captured fn is applied to the *value* part, and the $resource'$ part is fed into the result of that (evaluating the right-hand-side).

For a sense of how this should work (in lispy psuedo-code, and ignoring the Right value).

```
;; lhs >>= rhs
(>>= lhs rhs)

;; lhs >>= cntr >>= rhs
(>>= (>>= lhs cntr)
     rhs)
```

```
;; lhs >>= rhs
(lambda (resource)
  (let-values
      ([(resource' vv)
        ((lhs) resource)])
    ((rhs vv) resource')))

;; lhs >>= cntr >>= rhs
(lambda (resource)
  (let-values
      ([resource' vv]
       ((lambda (resource'')
          (let-values ([(resource''' vv)
                        ((lhs)
                         resource'')])
            ((cntr vv) resource''')))
        resource))
    ((rhs vv) resource')))
```

Well, that makes my head hurt, but the idea is there. Right is similar, but stops before applying fn. It returns instead a value which captures the $resource'$ and evaluated *state* bound to fn, bringing evaluation to an end.

Towards readability

With bind and return we have a everything we need for monadic constructs, and can start working thought the rest of the example. Function *step* is the monadic primitive, other functions use step to count down resources and jump to monad *Right* when they are exhausted; from here we have the monad building blocks, we can pass the 'resource' implicitly, and get ordering, but the syntax is not very good. An expression like:

```
(define (incR iR) (>>= iR (lambda (i) (step (+ i 1)))))
```

Is not much more convenient than passing a counter explicitly and checking it. But we can do better.
First off is the 'do' notation. To capture the >>= and λ pattern we've seen in a few places, the do form includes the $<-$ pattern; $(>>= \; iM(lambda(i)...))$ becoming $((<- \, i \, iM)...)$. Where the non-context part of a previous result isn't needed, $(>> \, (>> \, lhs \, cntr) \, rhs)$ becomes simple $(do \, lhs \, cntr \, rhs)$. Racket has a binding for do already, but an $m-do$ with these features can readily be implemented as a macro.

Here is a second working implementation of inc:

```
(define (incR_1 iR)
  (m-do (<- i iR)
        (step (+ i 1))))
```

This is more readable, but no more compact. Examining this and other functions reveals a pattern. To use a non-monadic function fn in a monadic context, first extract variables with >>= and lambda (or the m-do equivelent), apply fn to those extracted variables, and finally put the result into the monad using step. This pattern is captured with lift1 and lift2:

```
(define (lift1 f)
  (lambda (ra1)
   (m-do (<- a1 ra1)
         (step (f a1)))))
```

```
(define (lift2 f)
  (lambda (ra1 ra2)
    (m-do (<- a1 ra1)
          (<- a2 ra2)
          (step (f a1 a2)))))
```

To use it in inc;

```
(define (incR_2 iR) ((lift2 +) iR (return 1)))
```

or, remember to lift inc at it's point of use.

Lifting is fine for simple functions that consume one 'step', but the 'R' tutorial also introduces a factorial function. Simply lifting the function isn't very useful; we want to count each recursive step it takes (not to mention the internal arithmetic and testing steps), and (lift1 (fact (return 1))) will only ever consume one step. Manually lifting each sub-expression in the manner of inc above works, but is a bother.

The solution in Haskell was to make the R monad an instance the Num type-class, then calling a *fact* function defined in terms of Num in a context expecting Num (R a). Racket doesn't have an obvious way to do that, but looking at incR_2, we can see that there is a very regular pattern of where to 'lift' sub-expressions. With Racket's macros, we can be equally expressive, if not as type safe:

```
(def-lifted (fact x)
  (if (= x 0) 1 (* x (fact (- x 1)))))
```

Conclusion

Monads are by no means a panacea. If you look at the full source from which Fig0 and Fig1 come, both files are exactly the same length; all else being equal monads are not necessarily a win for expressiveness. However, the Monadic version allows an additional way to break down the problem. Some functions just worry about auxiliary state, some can just worry about the AST. Only a few have to worry about both, which is an improvement.

Monads have an obvious imperative counterpart in objects. Both have some advantage; monads preserve the purely functional nature of the context. If a chain of functions are evaluating in a Maybe context, nothing outside that chain can flip the state to Nothing; an OO method checking some 'weAreOk' class member has no such guarantee. On the flip side, objects are somewhat more intuitively accessible, and provide several well understood mechanisms for composition beyond simple nesting.

For an impure functional language like racket, monads are more a curiosity. They offer the same type of compactness and ordering as they do in Haskell, but the promise of predictable side effects depends on the user playing by the rules. For a disciplined programmer, this could be the best of both worlds; guarantees that can be counted on but the ability to break them temporarily for things like printf debugging.

An interesting further project would be an imperative/OO style implementation of min-steps and run for comparison. The monadic expression may prove cleaner in the long run as managing a global or per-def resource counter could easily be quite painful.

Notes:

examples are included in project.tgz
Fig0 and Fig1 are taken from project/Y/LambdaNoMonad.hs and project/Y/Lambda.hs respectively
Working scheme code is in project/Monads.rkt
If you're curious, I'm leaving the rest of the detritus for your perusal.