Overview: Monads, in effect

Monads are a programming concept used in functional programming languages to manage the sequencing of computations. A monad is a design pattern that allows a programmer to chain together a series of operations in a way that is both concise and expressive.

In practical terms, a monad is a way of encapsulating side effects (such as input/output operations or exceptions) within a sequence of computations. This allows the programmer to perform operations that have side effects while maintaining referential transparency, meaning that the same input will always produce the same output.

Functor

Before we delve into the topic of monads, we need to establish the concept of functors. In simple terms, a functor is a type T that acts as a container for any arbitrary type a. It enables the mapping of functions from type a to type b within the container.

As an example, consider lists, which are functors. In this case, T refers to List, and a can be any type, such as numbers, strings, booleans, and so on. In Racket lists also already have a function called map which given a function a -> b transforms a List a into a List b:

```
(define (even? [n : Number]) : Boolean
  (eq? (modulo n 2) 0))

(map even? '(1 2 3))
> '(#f #t #f)
```

What happens is like this

```
[ 1        2        3  ]
  |        |        |
even?    even?    even?
  |        |        |
  v        v        v
[ #f       #t       #f ]
```

Here we converted a List Integer into a List Boolean.

Universal Unitarianism

So now we know what functors are: higher-order container types that permit arbitrary functions to be mapped inside of them.

There is one limitation, though. Say you’re using the list functor; if you map a function over a list of 5 elements, you’ll get back a list of 5 elements.

But what if you can’t or shouldn’t assume that your result list will have the same length as your argument list? That is the case for the motivating example at the start

of this essay. An even simpler example would be filtering a list of numbers: the result may be shorter than the argument list.

There are many ways to skin this cat but one is to return a list of lists. Filtering entails mapping each value of the list into an empty list (failure) or a list of one item (success). A function of type a -> T a, where T is a functor, is called unit.

```
(define (list-unit x) : (Listof 'x)
  (cons x '()))

(list-unit 3)
> '(3)
```

Using it, let’s write a function which checks if a number is even and instead of returning #t or #f returns a singleton list or an empty list:

```
(define (list-even? [n : Number]) : (Listof Number)
  (if (eq? (modulo n 2) 0)
      (list-unit n)
      '()))

(define (my-filter predicate lst)
  (map predicate last))

(list-even? 3)
> '()
(list-even? 2)
> '(2)
(my-filter list-even? '(1 2 3))
> '(() (2) ())
```

The computation proceeds like this:

```
[ 1           2           3  ]
  |           |           |
list-even?   list-even?   list-even?
  |           |           |
  v           v           v
[ []         [2]         []  ]
```

So the result list is the same length as the input list, but all the values are even.

Join

Getting back a list of lists is cool and all, but when I filter items out of a list I expect to get back a list of values, not a list of lists. The reason is I might want to process this list further, and I don’t want to have to write brittle, specialized procedures dealing with increasingly-nested layers of lists.

No, when I filter a List a I want to get back a List a. We already have a List (List a); can we write a routine to flatten out a list of lists?

Absolutely! If you have a function of type T (T a) -> T a for some functor T, it is often called join. Here is the list join:

```
(define (list-join xss)
  (foldr (λ ([ y  : (Listof 'a)]
             [ ys : (Listof 'a)]) (append y ys))
    empty xss))

(list-join '(() (2) ()))
> '(2)
(list-join (my-filter list-even? '(1 2 3 4)))
> '(2 4)
```

Bind

Actually, this pattern of joining the result of a map and unit is so common it has its own name: bind.

bind is a map which can alter structure as it goes from element to element. Regardless of what T is, bind always has the same definition:

```
(define (bind container f) (join (map f container)))
```

This won't work in Plait; we need to substitute specific join and map functions.

```
(define (list-bind lst f)
  (list-join (map f last)))

(list-bind '(1 2 3) list-even?)
> '(2 4 6)
```

Note that list-even? accepts a single scalar number as an argument, yet using list-bind it is run over the whole list, resulting in a list of a different length.

The whole computation looks like this:

```
[ 1           2           3 ]
  |           |           |
list-even?   list-even?   list-even? -- map / unit
  |           |           |
  v           v           v
[ []         [2]         [] ]      -- join
  |           |           |
  v           v           v
[             2             ]
```

This is, in essence, what a monad is: a structure that lets you map a function into the structure which may, as a side-effect, change the structure. Different monads allow for different side-effects.

I want to demonstrate another functor (more succinctly than I did for lists, to be certain) but first, let's tackle the example from the beginning of this post.

As in that snippet, assume here that get-referrals is defined meaningfully. So that this example will compile, I'll write a dummy version that spits out three referrals for each customer given:

```
(define (get-referrals customer)
  (map (λ ([n : String])
    (string-append (string-append customer " referral: ") n))
    '("1" "2" "3")))

(define (get-customer-referrals customers)
  (list-bind customers get-referrals))

(get-referrals "Dave")
>'("Dave referral: 1"
   "Dave referral: 2"
   "Dave referral: 3")
(get-customer-referrals '("Dave" "John"))
>'("Dave referral: 1"
   "Dave referral: 2"
   "Dave referral: 3"
   "John referral: 1"
   "John referral: 2"
   "John referral: 3")
```

Conclusion

Monads aren't hard: they are containers of other values which not only allow the values to be transformed, but the container itself. The structural change is called a side-effect and different monads allow for the controlled propagation of different side effects.

Where all this nonsense really becomes interesting is when you write generic "monadic" functions - not specific to any particular monad - and are able to get different behaviors depending on the monad you choose.

For instance, I could write a simple monadic function to multiply a number by two. Fed into a Box monad, this simply lets me use it in an imperative function. Fed into a List monad, this function is automatically applied to a list of values.

By: Love and Dharmik Patel