Monads as containers

Categories: Tutorials | Monad

There now exists a translation of this article into Russian!

A *monad* is a container type together with a few methods defined on it. Monads model different kinds of computations.

Like Haskell lists, all the elements which a monadic container holds at any one time must be the same type (it is homogeneous).

There are a few ways to choose the basic set of functions that one can perform on these containers to be able to define a monad. Haskell generally uses a pair of functions called return and bind `(>>=)`, but it is more natural sometimes to begin with map (`fmap`), return and join, as these are simpler to understand at first. We can later define bind in terms of these.

The first of these three, generally called *map*, (but called `fmap` in Haskell 98) actually comes from the definition of a *functor*. We can think of a functor as a type of container where we are permitted to apply a single function to every object in the container.

That is, if `f` is a functor, and we are given a function of type `(a -> b)`, and a container of type `(f a)`, we can get a new container of type `(f b)`.

This is expressed in the type of `fmap`:

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

If you will give me a blueberry for each apple I give you `(a -> b)`*, and I have a box of apples* `(f a)`*, then I can get a box of blueberries* `(f b)`*.*

Every monad is a functor.

The second method, *return*, is specific to monads. If `m` is a monad, then return takes an element of type `a`, and gives a container of type `(m a)` with that element in it. So, its type in Haskell is

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

If I have an apple `(a)` *then I can put it in a box* `(m a)`*.*

The third method, *join*, also specific to monads, takes a container of containers `m (m a)`, and combines them into one `m a` in some sensible fashion. Its Haskell type is

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

If I have a box of boxes of apples `(m (m a))` *then I can take the apples from each, and put them in a new box* `(m a)`*.*

From these, we can construct an important operation called *bind* or *extend*, which is commonly given the symbol `(>>=)`. When you define your own monad in Haskell, it will expect you to

define just return and bind. It turns out that mapping and joining come for free from these two. Although only return and bind are needed to define a monad, it is usually simpler to think about map, return, and join first, and then get bind from these, as map and join are in general simpler than bind.

What bind does is to take a container of type `(m a)` and a function of type `(a -> m b)`. It first maps the function over the container, (which would give an `m (m b)`) and then applies join to the result to get a container of type `(m b)`. Its type and definition in Haskell is then

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
xs >>= f = join (fmap f xs)
    -- how we would get bind (>>=) in Haskell if it were join and fmap
    -- that were chosen to be primitive
```

If I have a box of apples `(m a)` *and for each apple, you will give me a box of blueberries* `(a -> m b)` *then I can get a box with all the blueberries together* `(m b)`*.*

Note that for a given container type, there might be more than one way to define these basic operations (though for obvious reasons, Haskell will only permit one instance of the Monad class per actual type). [Technical side note: The functions return and bind need to satisfy a few laws in order to make a monad, but if you define them in a sensible way given what they are supposed to do, the laws will work out. The laws are only a formal way to give the informal description of the meanings of return and bind I have here.]

It would be good to have a concrete example of a monad at this point, as these functions are not very useful if we cannot find any examples of a type to apply them to.

Lists are most likely the simplest, most illustrative example. Here, `fmap` is just the usual `map`, `return` is just `(\x -> [x])` and `join` is `concat`.

```
instance Monad [] where
    --return :: a -> [a]
    return x = [x] -- make a list containing the one element given

    --(>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
        -- collect up all the results of f (which are lists)
        -- and combine them into a new list
```

The list monad, in some sense, models computations which could return any number of values. Bind pumps values in, and catches all the values output. Such computations are known in computer science as *nondeterministic*. That is, a list [x,y,z] represents a value which is all of the values x, y, and z at once.

A couple examples of using this definition of bind:

```
[10,20,30] >>= \x -> [x, x+1]
   -- a function which takes a number and gives both it and its
   -- successor at once
 = [10,11,20,21,30,31]

[10,20,30] >>= \x -> [x, x+1] >>= \y -> if y > 20 then [] else [y,y]
 = [10,10,11,11,20,20]
```

And a simple fractal, exploiting the fact that lists are ordered:

```
f x | x == '#'  = "# #"
    | otherwise = "   "

"#" >>= f >>= f >>= f >>= f
 = "# #   # #         # #   # #                                 # #   # #         # #   # #"
```

You might notice a similarity here between bind and function application or composition, and this is no coincidence. The reason that bind is so important is that it serves to chain computations on monadic containers together.

You might be interested in how, given just bind and return, we can get back to map and join.

Mapping is equivalent to binding to a function which only returns containers with a single value in them -- the value that we get from the function of type `(a -> b)` which we are handed.

The function that does this for any monad in Haskell is called `liftM` -- it can be written in terms of return and bind as follows:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f xs = xs >>= (return . f)
   -- take a container full of a's, to each, apply f,
   -- put the resulting value of type b in a new container,
   -- and then join all the containers together.
```

Joining is equivalent to binding a container with the identity map. This is indeed still called `join` in Haskell:

```
join :: (Monad m) => m (m a) -> m a
join xss = xss >>= id
```

It is common when constructing monadic computations that one ends up with a large chain of binds and lambdas. For this reason, some syntactic sugar called "do notation" was created to simplify this process, and at the same time, make the computations look somewhat like imperative programs.

Note that in what follows, the syntax emphasizes the fact that the list monad models nondeterminism: the code `y <- xs` can be thought of as `y` taking on all the values in the list `xs` at once.

The above (perhaps somewhat silly) list computations could be written:

```
do x <- [10,20,30]
   [x, x+1]
```

and,

```
do x <- [10,20,30]
   y <- [x, x+1]
   if y > 20 then [] else [y,y]
```

The code for liftM could be written:

```
liftM f xs = do a <- xs
                return (f a)
```

If you understood the above, then you have a good start on understanding monads in Haskell.

Check out Maybe (containers with at most one thing in them, modelling computations that might not return a value) and State (modelling computations that carry around a state parameter using an odd sort of container described below), and you'll start getting the picture as to how things work.

A good exercise is to figure out a definition of bind and return (or fmap, join and return) which make the following tree type a monad. Just keep in mind what they are supposed to do.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

For more good examples of monads, and lots of explanation see All About Monads which has a catalogue of the commonest ones, more explanation as to why you might be interested in monads, and information about how they work.

The question that many people will be asking at this point is "What does this all have to do with IO?".

Well, in Haskell, IO is a monad. How does this mesh with the notion of a container?

Consider `getChar :: IO Char` -- that is, an IO container with a character value in it. The exact character that the container holds is determined when the program runs by which keys the user presses. Trying to get the character out of the box will cause a side effect: the program stops and waits for the user to press a key. This is generally true of IO values - when you get a value out with bind, side effects can occur. Many IO containers don't actually contain interesting values. For example,

```
putStrLn "Hello, World!" :: IO ()
```

That is, the value returned by `putStrLn "Hello, World!"` is an IO container filled with a value of type `()`, a not so interesting type. However, when you pull this value out during a bind operation, the string `Hello, World!` is printed on the screen. So another way to think about values of type `IO t` is as computations which when executed, may have side effects before returning a value of type t.

One thing that you might notice as well, is that there is no ordinary Haskell function you can call (at least not in standard Haskell) to actually get a value out of an IO container/computation, other than bind, which puts it right back in. Such a function of type `IO a -> a` would be very unsafe in the pure Haskell world, because the value produced could be different each time it was called, and the IO computation could have side effects, and there would be no way to control when it was executed (Haskell is lazy after all). So how do IO actions ever get run? The IO action called `main` runs when the program is executed. It can make use of other IO actions in the process, and everything starts from there.

When doing IO, a handy special form of bind when you just want the side effects and don't care about the values returned by the container on the left is this:

```
(>>) :: Monad m => m a -> m b -> m b
m >> k  =  m >>= \_ -> k
```

An example of doing some IO in do notation:

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello " ++ name ++ "!")
```

or in terms of bind, making use of the special form:

```
main = putStrLn "Hello, what is your name?" >>
       getLine >>= \name ->
       putStrLn ("Hello " ++ name ++ "!")
```

or, very primitive, without the special form for bind:

```
main = putStrLn "Hello, what is your name?" >>= \x ->
       getLine >>= \name ->
       putStrLn ("Hello " ++ name ++ "!")
```

Another good example of a monad which perhaps isn't obviously a container at first, is the Reader monad. This monad basically consists of functions from a particular type: `((->) e)`, which might be written `(e ->)` if that were supported syntax. These can be viewed as containers indexed by values of type `e`, having one spot for each and every value of type `e`. The primitive operations on them follow naturally from thinking this way.

The Reader monad models computations which read from (depend on) a shared environment. To clear up the correspondence, the type of the environment is the index type on our indexed containers.

```
type Reader e = (->) e -- our monad
```

Return simply produces the container having a given value at every spot.

```
return :: a -> (Reader e a)
return x = (\k -> x)
```

Mapping a function over such a container turns out to be nothing more than what composition does.

```
fmap :: (a -> b) -> Reader e a -> Reader e b
     = (a -> b) -> (e -> a) -> (e -> b)
         -- by definition, (Reader a b) = (a -> b)

fmap f xs = f . xs
```

How about join? Well, let's have a look at the types.

```
join :: (Reader e) (Reader e a) -> (Reader e a)
     = (e -> e -> a) -> (e -> a) -- by definition of (Reader a)
```

There's only one thing the function of type `(e -> a)` constructed could really be doing:

```
join xss = (\k -> xss k k)
```

From the container perspective, we are taking an indexed container of indexed containers and producing a new one which at index `k`, has the value at index `k` in the container at index `k`.

So we can derive what we want bind to do based on this:

```
(>>=) :: (Reader e a) -> (a -> Reader e b) -> (Reader e b)
      = (e -> a) -> (a -> (e -> b)) -> (e -> b)  -- by definition

xs >>= f = join (fmap f xs)
```

```
        = join (f . xs)
        = (\k -> (f . xs) k k)
        = (\k -> f (xs k) k)
```

Which is exactly what you'll find in other definitions of the Reader monad. What is it doing? Well, it's taking a container `xs`, and a function `f` from the values in it to new containers, and producing a new container which at index `k`, holds the result of looking up the value at `k` in `xs`, and then applying `f` to it to get a new container, and finally looking up the value in that container at `k`.

The Monads as computation perspective makes the purpose of such a monad perhaps more obvious: bind is taking a computation which may read from the environment before producing a value of type `a`, and a function from values of type `a` to computations which may read from the environment before returning a value of type `b`, and composing these together, to get a computation which might read from the (shared) environment, before returning a value of type `b`.

How about the State monad? Although I'll admit that with State and IO in particular, it is generally more natural to take the view of Monads as computation, it is good to see that the container analogy doesn't break down. The state monad is a particular refinement of the reader monad discussed above.

I won't go into huge detail about the state monad here, so if you don't already know what it's for, what follows may seem a bit unnatural. It's perhaps better taken as a secondary way to look at the structure.

For reference to the analogy, a value of type `(State s a)` is like a container indexed by values of type `s`, and at each index, it has a value of type `a` and another, new value of type `s`. The function `runState` does this "lookup".

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

What does return do? It gives a `State` container with the given element at every index, and with the "address" (a.k.a. state parameter) unchanged.

```
return :: a -> State s a
return x = State (\s -> (x,s))
```

Mapping does the natural thing, applying a function to each of the values of type `a`, throughout the structure.

```
fmap :: (a -> b) -> (State s a) -> (State s b)
fmap f (State m) = State (onVal f . m)
    where onVal f (x, s) = (f x, s)
```

Joining needs a bit more thought. We want to take a value of type `(State s (State s a))` and turn it into a `(State s a)` in a natural way. This is essentially removal of indirection. We take the new address and new box that we get from looking up a given address in the box, and we do another lookup -- note that this is almost the same as what we did with the reader monad, only we use the new address that we get at the location, rather than the same address as for the first lookup.

So we get:

```
join :: (State s (State s a)) -> (State s a)
join xss = State (\s -> uncurry runState (runState xss s))
```

I hope that the above was a reasonably clear introduction to what monads are about. Feel free to make criticisms and ask questions.

- CaleGibbard
