Monads in Functional Programming

George Sittas

Department of Informatics and Telecommunications (DiT)
University of Athens

May, 2022

What is a monad?

- An abstract mathematical concept, originally introduced in Category Theory by Roger Godement (1958).
- Use in Functional Programming due to Wadler, Moggi (1990).
- Software design pattern, which enables the **composition** of functions that take *normal* values and wrap their results in a type (*fancy value*) with additional **computation** (*context*).

How does it help us?

- Reduces code complexity by transforming complicated function sequences into succinct pipelines that abstract away control flow and side-effects (separation of concerns).

How does it help us?

- Reduces code complexity by transforming complicated function sequences into succinct pipelines that abstract away control flow and side-effects (separation of concerns).
- Bridges pure and impure code (eg. I/O, mutable state) in pure functional languages, by isolating the side-effects.

How does it help us?

- Reduces code complexity by transforming complicated function sequences into succinct pipelines that abstract away control flow and side-effects (separation of concerns).
- Bridges pure and impure code (eg. I/O, mutable state) in pure functional languages, by isolating the side-effects.
- Makes semantics (*effects*) explicit for a kind of computation, thus improving code clarity.

How does it help us?

- Reduces code complexity by transforming complicated function sequences into succinct pipelines that abstract away control flow and side-effects (separation of concerns).
- Bridges pure and impure code (eg. I/O, mutable state) in pure functional languages, by isolating the side-effects.
- Makes semantics (*effects*) explicit for a kind of computation, thus improving code clarity.
- Provides a toolkit for expressing and solving a wide range of problems, due to its abstract nature.

Example: computations that can fail

Let's suppose that we want to write an expression evaluator. For simplicity, our expressions will only consist of integers that can be combined through division. In Haskell, we could express this as:

```
data Expr = Val Int | Div Expr Expr
```

In this way, we could represent the expression $5 \div 4 \div 3$ as:

```
Div (Div (Val 5) (Val 4)) (Val 3)
```

Note: division is left-associative!

Example: computations that can fail

Given the above definition, we could then write a simple evaluator:

```
eval :: Expr -> Int
eval (Val n)   = n
eval (Div x y) = (eval x) / (eval y)
```

Of course, this definition is bound to fail, were we to try and divide by zero. So, we would like to write a division-safe version instead:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv n m = if m == 0 then Nothing
                        else Just (n / m)
```

Example: computations that can fail

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
  Nothing -> Nothing
  Just n  -> case eval y of
      Nothing -> Nothing
      Just m  -> n `safeDiv` m
```

- Program is now less readable: filled with boilerplate code.
- Pattern: when trying to chain together *computations that might fail*, we first inspect the value we get from the first computation, and if it's a Just value, we feed it forward to the next computation, otherwise the whole computation fails.

Example: computations that can fail

We can now abstract this pattern away, by defining a new operator which we will call "bind", that handles all the dirty work for us:

```
bind :: Maybe Int -> (Int -> Maybe Int)
                  -> Maybe Int
bind Nothing  f = Nothing
bind (Just x) f = Just (f x)
```

Its job is to *un-wrap* a Maybe value, and if it's a Just value, then apply whatever function it's passed as argument to it (here, it's f), and finally re-wrap the value inside a Maybe, in order to *preserve the context of the computation* (possible failure).

Example: computations that can fail

The above code can then be transformed equivalently into:

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x `bind` (\n ->
                 eval y `bind` (\m ->
                 n `safeDiv` m))
```

- We reduced a portion of the boilerplate code, but in terms of readability, this version isn't *a lot* better (yet).
- Here, the "bind" operator *binds* x to n, and y to m, if both eval computations succeed, and then safeDiv is called.
- Inspecting the type signatures can help to understand this!

Example: computations that can fail

We actually don't need to define the "bind" operator, because Haskell has already done it for us! Its type signature is (roughly):

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

- m is a type constructor
- a is its type argument
- Describes the application of a function to a value that's contained in a context m, producing a value in the same context (of possibly different type)

So we can rewrite the above program as:

Example: computations that can fail

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= (\n ->
                 eval y >>= (\m ->
                 n `safeDiv` m))
```

The nice thing is that Haskell also provides us with a shorthand notation, called "do notation", for code segments like this one.

Example: computations that can fail

This leads us to the final form of our initial program:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv n m = if m == 0 then Nothing
                        else Just (n / m)

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    n `safeDiv` m
```

Example: computations that can fail

- Similar level of complexity to the initial, unsafe program.
- The failure management is now handled for us automatically.

We have essentially rediscovered the *Maybe monad*! Right away, we can see how doing all this work helped us:

- Reduce the complexity of the program, by abstracting away all additional computation in the definition of bind.
- Express the *effect* of the computation in its type explicitly, namely: the evaluation of an expression *might fail*.

Overview

> *For a monad m, a value of type m a represents having access to a value of type a within the context of a monad.*
>
> *(C. A. McCann)*

- In the last example, we briefly mentioned the Maybe monad.
- A question now arises: *what structure does a monad have*?

Let's see how we can answer this question in Haskell.

Definition

A monad can be created by defining a **type constructor** m of one type parameter a, and two operations:

- **return** (or *unit*), which receives a value of type a and *wraps* it into a **monadic value** of type m, using the type constructor:

```
return :: a -> m a
```

- **bind** (»=), which receives a function f over a type a and can transform monadic values m a applying f to the unwrapped value a, returning a monadic value m b:

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

Definition

- The bind operator is a way to combine monadic values, and its precise definition differs, depending on the monad at hand.
- The return operator simply takes a value and puts it in some sort of default context – a minimal context that still yields that value.

Note: return has no relation to the return *statement* in imperative languages whatsoever. It's just a way to "promote" a *normal* value into a *fancy* (monadic) one.

Definition

The following **monadic laws** must also be satisfied:

- Left-identity:

```
return a >>= h = h a
```

- Right-identity:

```
m >>= return = m
```

- Associativity:

```
(m >>= g) >>= h =
= m >>= (\x -> g x >>= h)
```

Maybe monad, revisited

Having discussed about the general structure of a monad, we can finally see the complete definition of the Maybe monad in Haskell:

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

The List monad

Haskell defines various monads, each representing a kind of computation with additional context or effects.

The list type constructor is another classic example. Observe:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

The List monad

But . . . *how is a list a value with additional context*? One way to look at this, is that a list is a value that contains many values at the same time, something like *a non-deterministic computation*.

- return just takes a normal value and wraps it around a list, providing a minimal context.
- bind is more interesting, see the following example.

The List monad

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Note: remember, the bind operator takes a monadic value and a function, and returns a monadic value in the same context (list)!

- bind takes a non-deterministic computation and feeds every result to the provided function, which results in a new non-deterministic computation

The List monad

List comprehensions are actually syntactic sugar for using lists as monads! Observe:

```
[1,2] >>= \n -> ['a','b'] >>= \c -> return (n,c)
```

```
do { n <- [1,2]; c <- ['a','b']; return (n,c) }
```

```
[ (n,c) | n <- [1,2], c <- ['a','b'] ]
```

These three are equivalent. They all output the following:

```
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

The State monad

- Another cool monad is the *State* monad.
- It represents computations that need some kind of state (context) (examples: assignment, RNG, parsers, games).
- These computations can be thought of as functions that take a state and output a value and a new state with it:

```
s -> (a,s)
```

  Here, s is the state's type and a is the result's type.
- The point is to *lazily* construct chains of functions that simulate changing state, yet in an immutable, referentially transparent way.

The State monad

The State monad can be defined as follows:

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

instance Monad (State s) where
  return x = State (\s -> (x,s))
  (State h) >>= f =
    State (\s -> let (a,s')     = h s
                     (State g) = f a
                 in g s')
```

- return: take a value and make a *stateful computation* that always has that value as its result.
- bind: get the current state, feed it to a computation f and get a new *stateful computation* as output, with its state updated accordingly.

The State monad

Here, runState is a way of executing such a computation: we pass a state to it and it executes each consecutive stateful computation, threading each new state to the next function each time. It returns the *actual value* of the computation and the resulting state.

There are two useful functions operating on stateful computations:

```
get = State ( \s -> (s,s) )
put s = State ( \s -> ((),s) )
```

- get: take current state and present it as the result
- put: take some state and make a stateful function that replaces the current state with it.

The State monad

Let's see a simple string parsing game as an example:

- Produce a number from a string of a's, b's and c's.
- By default, the game is off, "c" toggles the game on and off.
- An "a" gives +1, and a "b" gives -1.

The State monad

```
type GameValue = Int
type GameState = (Bool, Int)

playGame :: String -> State GameState GameValue
playGame [] = do
    (_, score) <- get
    return score

playGame (x:xs) = do
    (on, score) <- get
    case x of
      'a' | on -> put (on, score + 1)
      'b' | on -> put (on, score - 1)
      'c' |    -> put (not on, score)
      _        -> put (on, score)
    playGame xs
```

The State monad

And we can then play the game like so:

```
ghci> fst (runState (playGame "cabca") (False, 0))
0
```

Which is the expected result. If it's not easy to see what's going on, maybe applying currying and lazy evaluation will help to grasp the execution flow.

Recap

Again, we can see the benefits that monads give us, as we discussed earlier:

- In lists, abstracting away the chaining of concat and map gave us a pretty elegant tool, which enhances the program's expressibility: list comprehensions
- In stateful computations, we have abstracted away the threading of the states, creating a sense of "mutable" state in our program, even though it's all pure under the hood!
- The types express the computational effects clearly: with lists, each computation can have 0 or more values, whilst with stateful computations, additional state is needed.

Monads in the wild

Monads can be found in various different pieces of software. For instance, here are a couple of them:

- Parsing: the Parsec library uses monads to combine simpler parsing rules into more complex ones.
- LINQ by Microsoft provides a query language for the .NET framework that is heavily influenced by functional prog. concepts, including operators for composing queries monadically.

Another point of view

Typically, programmers will use the bind operator to chain *monadic functions* into a sequence, which has led some to describe monads as **programmable semicolons**, which references the way other languages (imperative) use semicolons to separate statements.

References

- Learn You A Haskell For Greater Good
- Wikipedia - Monads
- Haskell wiki - All About Monads
- Computerphile - What is a Monad?