Programación Declarativa: mónadas, parsers

2016-11-1

Mónadas

Motivación

A monad is handy whenever a programmer wants to sequence actions. The details of the monad says exactly how the actions should be sequenced. A monad may also store some information that can be read from and written to while performing actions.

We've already learned about the IO monad, which sequences its actions quite naturally, performing them in order, and gives actions access to read and write anything, anywhere. We'll also see the Maybe and `[]` (pronounced "list") monads, which don't give any access to reading and writing, but do interesting things with sequencing. And, for homework, you'll use the Rand monad, which doesn't much care about sequencing, but it does allow actions to read from and update a random generator.

One of the beauties of programming with monads is that monads allow programmers to work with mutable state from a pure language. Haskell doesn't lose its purity when monads come in (although monadic code is often called "impure"). Instead, the degree to which code can be impure is denoted by the choice of monad. For example, the Rand monad means that an action can generate random numbers, but can't for example, write strings to the user. And the Maybe monad doesn't give you any extra capabilities at all, but makes writing possibly-erroring computations much easier to write.

In the end, the best way to really understand monads is to work with them for a while. After programming using several different monads, you'll be able to abstract away the essence of what a monad really is. To demonstrate this, consider the following example. We would like to write a function that zips two binary trees together by applying a function to the values at each node. However, the function should fail if the structure of the two trees are different. Note that by fail, we mean return Nothing. Here is a first try at writing this function:

```
data Tree a = Node (Tree a) a (Tree a)
            | Empty
              deriving (Show)
```

```haskell
zipTree1 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree1 _ (Node _ _ _) Empty = Nothing
zipTree1 _ Empty (Node _ _ _) = Nothing
zipTree1 _ Empty Empty        = Just Empty
zipTree1 f (Node l1 x r1) (Node l2 y r2) =
    case zipTree1 f l1 l2 of
      Nothing -> Nothing
      Just l  -> case zipTree1 f r1 r2 of
                   Nothing -> Nothing
                   Just r  -> Just $ Node l (f x y) r
```

This code works, but it is not very elegant. Notice how we have nested case matches with very similar structures; if scrutinee of the case match, evaluates to Nothing, then it returns Nothing, otherwise it binds the value in the Just constructor to a variable and uses it in a computation. Ideally, we would want a helper function like:

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

Using this helper, we can refactor the code to be much more elegant.

```haskell
zipTree2 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree2 _ (Node _ _ _) Empty = Nothing
zipTree2 _ Empty (Node _ _ _) = Nothing
zipTree2 _ Empty Empty        = Just Empty
zipTree2 f (Node l1 x r1) (Node l2 y r2) =
    bindMaybe (zipTree2 f l1 l2) $ \l ->
        bindMaybe (zipTree2 f r1 r2) $ \r ->
            Just (Node l (f x y) r)
```

Monad

Believe it or not, the zipTree2 function uses Monads! The Monad type class is defined as follows:

```haskell
class Monad m where
  return :: a -> m a

   -- pronounced "bind"
  (>>=) :: m a -> (a -> m b) -> m b

  (>>)  :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2
```

We've, in fact, already seen `return`, specialized to the IO monad. Here, we see that it's available in every monad.

`(>>)` is just a specialized version of (>>=) (it is included in the Monad class in case some instance wants to provide a more efficient implementation, but usually the default implementation is just fine). So to understand it we first need to understand (>>=).

`(>>=)` (pronounced "bind") is where all the action is! Let's think carefully about its type:

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

`(>>=)` takes two arguments. The first one is a value of type `m a`. (Incidentally, such values are sometimes called monadic values, or computations, or actions. The one thing you must not call them is "monads", since that is a kind error: the type constructor m is a monad.) In any case, the idea is that an action of type `m a` represents a computation which results in a value (or several values, or no values) of type a, and may also have some sort of "effect":

- `c1 :: Maybe a` is a computation which might fail but results in an a if it succeeds.
- `c2 :: [a]` is a computation which results in (multiple) as.
- `c3 :: Rand StdGen a` is a computation which may use pseudo-randomness and produces an a.
- `c4 :: IO a` is a computation which potentially has some I/O effects and then produces an a.

And so on. Now, what about the second argument to `(>>=)`? It is a function of type `(a -> m b)`. That is, it is a function which will choose the next computation to run based on the result(s) of the first computation. This is precisely what embodies the promised power of Monad to encapsulate computations which can be sequenced.

So all `(>>=)` really does is put together two actions to produce a larger one, which first runs one and then the other, returning the result of the second one. The all-important twist is that we get to decide which action to run second based on the output from the first.

The default implementation of `(>>)` should make sense now:

```
(>>)  :: m a -> m b -> m b
m1 >> m2 = m1 >>= \_ -> m2
```

`m1 >> m2` simply does m1 and then m2, ignoring the result of m1.

Examples

Let's start by writing a Monad instance for Maybe:

```
instance Monad Maybe where
  return  = Just
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

`return`, of course, is Just. The implementation of `(>>=)` is exactly the same as bindMaybe above, but the pattern match of the first argument is inlined in to the function definition instead of in a separate case. If the first argument of `(>>=)` is Nothing, then the whole computation fails; otherwise, if it is `Just x`, we apply the second argument to x to decide what to do next.

Incidentally, it is common to use the letter k for the second argument of `(>>=)` because k stands for "continuation".

Now that we know about Monads, we can write zipTree in a more canonical way:

```
zipTree3 :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree3 _ (Node _ _ _) Empty = Nothing
zipTree3 _ Empty (Node _ _ _) = Nothing
zipTree3 _ Empty Empty        = Just Empty
zipTree3 f (Node l1 x r1) (Node l2 y r2) =
    zipTree3 f l1 l2 >>= \l ->
        zipTree3 f r1 r2 >>= \r ->
            return (Node l (f x y) r)
```

The do notation we've learned for working with IO can work with any monad. The backwards arrows that we use in a do block are just syntactic sugar for binds. For example, consider the following do block:

```
addM :: Monad m => m Int -> m Int -> m Int
addM mx my = do
  x <- mx
  y <- my
  return $ x + y
```

GHC will desugar this directly to a version that explicitly uses (>>=):

```
addM' :: Monad m => m Int -> m Int -> m Int
addM' mx my = mx >>= \x -> my >>= \y -> return (x + y)
```

Using do notation, we can refactor zipTree one last time:

```
zipTree :: (a -> b -> c) -> Tree a -> Tree b -> Maybe (Tree c)
zipTree _ (Node _ _ _) Empty = Nothing
zipTree _ Empty (Node _ _ _) = Nothing
zipTree _ Empty Empty        = Just Empty
```

```
zipTree f (Node l1 x r1) (Node l2 y r2) = do
    l <- zipTree f l1 l2
    r <- zipTree f r1 r2
    return $ Node l (f x y) r
```

Here are some more examples:

```
check :: Int -> Maybe Int
check n | n < 10    = Just n
        | otherwise = Nothing

halve :: Int -> Maybe Int
halve n | even n    = Just $ n `div` 2
        | otherwise = Nothing

ex01 = return 7 >>= check >>= halve
ex02 = return 12 >>= check >>= halve
ex03 = return 12 >>= halve >>= check
```

Or maybe you prefer doing it this way:

```
ex04 = do
  checked <- check 7
  halve checked
ex05 = do
  checked <- check 12
  halve checked
ex06 = do
  halved <- halve 12
  check halved
```

List Monad

How about a Monad instance for the list constructor `[]`?

```
instance Monad [] where
  return x = [x]
  xs >>= k = concatMap k xs
```

A simple example:

```
addOneOrTwo :: Int -> [Int]
addOneOrTwo x = [x+1, x+2]

ex07 = [10,20,30] >>= addOneOrTwo
ex08 = do
  num <- [10, 20, 30]
  addOneOrTwo num
```

The Haskell Prelude even defines a backwards bind `(=<<)` with the arguments reversed:

```
ex09 = addOneOrTwo =<< [10,20,30]
```

We can think of the list monad as encoding non-determinism, and then producing all possible values of a computation. Above, num is non-deterministically selected from [10, 20, 30] and then is non-deterministically added to 1 or 2. The result is a list of 6 elements with all possible results.

This non-determinism can be made even more apparent through the use of the function guard, which aborts a computation if its argument isn't True:

```
ex10 = do
  num <- [1..20]
  guard (even num)
  guard (num `mod` 3 == 0)
  return num
```

Here, we can think of choosing num from the range 1 through 20, and then checking if it is even and divisible by 3.

The full type of `guard` is `MonadPlus m => Bool -> m ()`. MonadPlus is another class (from Control.Monad) that characterizes monads that have a possibility of failure. These include Maybe and []. guard then takes a Boolean value, but produces no useful result. That's why its return type is m () – no new information comes out from it. But, `guard` clearly does affect sequencing, so it is still useful.

Ejercicios (parte 1)

Hacer ejercicios 1 y 2 (`stringFitsFormat` y `specialNumbers`).

Monad combinators

One nice thing about the Monad class is that using only return and `(>>=)` we can build up a lot of nice general combinators for programming with monads. Let's look at a couple.

First, sequence takes a list of monadic values and produces a single monadic value which collects the results. What this means depends on the particular monad. For example, in the case of Maybe it means that the entire computation succeeds only if all the individual ones do; in the case of IO it means to run all the computations in sequence.

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma:mas) = do
```

```
  a  <- ma
  as <- sequence mas
  return (a:as)
```

Using sequence we can also write other combinators, such as:

```
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n m = sequence (replicate n m)

void :: Monad m => m a -> m ()
void ma = ma >> return ()

join :: Monad m => m (m a) -> m a
join mma = do
  ma <- mma
  ma

when :: Monad m => Bool -> m () -> m ()
when b action =
  if b
  then action
  else return ()
```

Applicative Functors

Motivation

Consider the following Employee type:

```
type Name = String

data Employee = Employee { name    :: Name
                         , phone   :: String }
                deriving Show
```

Of course, the Employee constructor has type

```
Employee :: Name -> String -> Employee
```

That is, if we have a Name and a String, we can apply the Employee constructor to build an Employee object.

Suppose, however, that we don't have a Name and a String; what we actually have is a Maybe Name and a Maybe String. Perhaps they came from parsing some file full of errors, or from a form where some of the fields might have been left blank, or something of that sort. We can't necessarily make an Employee. But surely we can make a Maybe Employee. That is, we'd like to take our `(Name`

`-> String -> Employee)` function and turn it into a `(Maybe Name -> Maybe String -> Maybe Employee)` function. Can we write something with this type?

```
(Name -> String -> Employee) ->
(Maybe Name -> Maybe String -> Maybe Employee)
```

Sure we can, and I am fully confident that you could write it in your sleep by now. We can imagine how it would work: if either the name or string is Nothing, we get Nothing out; if both are Just, we get out an Employee built using the Employee constructor (wrapped in Just). But let's keep going...

Consider this: now instead of a Name and a String we have a [Name] and a [String]. Maybe we can get an [Employee] out of this? Now we want

```
(Name -> String -> Employee) ->
([Name] -> [String] -> [Employee])
```

We can imagine two different ways for this to work: we could match up corresponding Names and Strings to form Employees; or we could pair up the Names and Strings in all possible ways.

Or how about this: we have an (e -> Name) and (e -> String) for some type e. For example, perhaps e is some huge data structure, and we have functions telling us how to extract a Name and a String from it. Can we make it into an (e -> Employee), that is, a recipe for extracting an Employee from the same structure?

```
(Name -> String -> Employee) ->
((e -> Name) -> (e -> String) -> (e -> Employee))
```

No problem, and this time there's really only one way to write this function.

Generalizing

Now that we've seen the usefulness of this sort of pattern, let's generalize a bit. The type of the function we want really looks something like this:

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

Hmm, this looks familiar... it's quite similar to the type of fmap!

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

The only difference is an extra argument; we might call our desired function fmap2, since it takes a function of two arguments. Perhaps we can write fmap2 in terms of fmap, so we just need a Functor constraint on f:

```
fmap2 :: Functor f => (a -> b -> c) -> (f a -> f b -> f c)
fmap2 h fa fb = undefined
```

Try hard as we might, however, Functor does not quite give us enough to implement fmap2. What goes wrong? We have

```
h  :: a -> b -> c
fa :: f a
fb :: f b
```

Note that we can also write the type of h as a -> (b -> c). So, we have a function that takes an a, and we have a value of type f a... the only thing we can do is use fmap to lift the function over the f, giving us a result of type:

```
h          :: a -> (b -> c)
fmap h     :: f a -> f (b -> c)
fmap h fa  :: f (b -> c)
```

OK, so now we have something of type `f (b -> c)` and something of type f b... and here's where we are stuck! fmap does not help any more. It gives us a way to apply functions to values inside a Functor context, but what we need now is to apply a functions which are themselves in a Functor context to values in a Functor context.

Applicative

Functors for which this sort of "contextual application" is possible are called applicative, and the Applicative class (defined in `Control.Applicative`) captures this pattern.

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The `(<*>)` operator (often pronounced "ap", short for "apply") encapsulates exactly this principle of "contextual application". Note also that the Applicative class requires its instances to be instances of Functor as well, so we can always use fmap with instances of Applicative. Finally, note that Applicative also has another method, pure, which lets us inject a value of type a into a container. For now, it is interesting to note that fmap0 would be another reasonable name for pure:

```
pure  :: a             -> f a
fmap  :: (a -> b)      -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
```

Now that we have (<*>), we can implement fmap2, which in the standard library is actually called liftA2:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
```

In fact, this pattern is so common that Control.Applicative defines (<$>) as a synonym for fmap,

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

so that we can write

```
liftA2 h fa fb = h <$> fa <*> fb
```

What about liftA3?

```
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 h fa fb fc = ((h <$> fa) <*> fb) <*> fc
```

(Note that the precedence and associativity of `(<$>)` and `(<*>)` are actually efined in such a way that all the parentheses above are unnecessary.)

Nifty! Unlike the jump from fmap to liftA2 (which required generalizing from Functor to Applicative), going from liftA2 to liftA3 (and from there to liftA4, ...) requires no extra power—Applicative is enough.

Actually, when we have all the arguments like this we usually don't bother calling liftA2, liftA3, and so on, but just use the f <$> x <> y <> z <*> ... pattern directly. (liftA2 and friends do come in handly for partial application, however.)

But what about `pure`? `pure` is for situations where we want to apply some function to arguments in the context of some functor f, but one or more of the arguments is not in f—those arguments are "pure", so to speak. We can use pure to lift them up into f first before applying. Like so:

```
liftX :: Applicative f => (a -> b -> c -> d) -> f a -> b -> f c -> f d
liftX h fa b fc = h <$> fa <*> pure b <*> fc
```

Applicative examples

Maybe

Let's try writing some instances of Applicative, starting with Maybe. `pure` works by injecting a value into a Just wrapper; `(<*>)` is function application with possible failure. The result is Nothing if either the function or its argument are.

```
instance Applicative Maybe where
  pure              = Just
  Nothing <*> _     = Nothing
  _ <*> Nothing     = Nothing
  Just f <*> Just x = Just (f x)
```

Let's see an example:

```
m_name1, m_name2 :: Maybe Name
m_name1 = Nothing
m_name2 = Just "Brent"
```

```
m_phone1, m_phone2 :: Maybe String
m_phone1 = Nothing
m_phone2 = Just "555-1234"

ex01 = Employee <$> m_name1 <*> m_phone1
ex02 = Employee <$> m_name1 <*> m_phone2
ex03 = Employee <$> m_name2 <*> m_phone1
ex04 = Employee <$> m_name2 <*> m_phone2
```

Parsing

A parser is an algorithm which takes unstructured data as input (often a `ByteString`) and produces structured data as output. For example, when you load a Haskell file into ghci, the first thing it does is parse your file in order to turn it from a long ByteString into an abstract syntax tree representing your code in a more structured form.

For the rest of the assignment, we will be using the parsing package Attoparsec. This is the same library that is used by Aeson which we used in Homework 5 to parse JSON data. Attoparsec has many simple parsers already defined. For example, we can write the following parser that parses a word (sequence of letters):

```
word :: Parser ByteString
word = takeWhile $ inClass "a-zA-Z"
```

Now, let's write a parser for names. For our purposes, a name is just a capitalized word. In order to do this we will have to combine two different parsers. Namely, a parser for a single capital letter, and a parser for a sequence of lowercase ones:

```
upper :: Parser Word8
upper = satisfy $ inClass "A-Z"

lword :: Parser ByteString
lword = takeWhile (inClass "a-z")
```

Ideally, we would like to cons the Word8 that is obtained by running the upper parser, on to the ByteString obtained by running lword. To do this we need to use a parser combinator. Luckily for us, Parser has an Applicative instance! Using Applicative Functors, we can apply the cons function inside the Parser:

```
name :: Parser ByteString
name = BS.cons <$> upper <*> lword
```

Or, alternatively, we can lift the cons function into the Parser using liftA2:

```haskell
name' :: Parser ByteString
name' = liftA2 BS.cons upper lword
```

Now, suppose we want to parse full names (ie, first and last) instead of just single names. Instead of returning a ByteString, we might want to structure the output data in some way. In this example, we will return a tuple containing the first and last names as separate ByteStrings. Before we do this, we will need some way of skipping over the whitespace between words. We can use the following parser for this:

```haskell
skipSpace :: Parser ()
skipSpace = skipWhile isSpace_w8
```

Note that the type of skipSpace is `Parser ()`. This is because we don't care about the exact value of the whitespace that we are skipping over, we just want to discard it. We will also need some way of running a parser, but not including the result in the output. The `(*>)` operator does exactly that! Now, let's write the full name parser:

```haskell
firstLast :: Parser (ByteString, ByteString)
firstLast = (,) <$> name <*> (skipSpace *> name)
```

This parser uses the Applicative instance for Parser on the (,) data constructor to construct a parser for a tuple of ByteStrings.

This parser for names works, but it does not accomodate people who have middle names. Ideally, we would like to have a parser that can inspect the input and decide whether the name includes a middle name or not. In particular, we want to target the following data type:

```haskell
data Name = TwoName ByteString ByteString
          | ThreeName ByteString ByteString ByteString
```

Unfortunately, there is no way to do this using applicative functors as our combinator. The reason is that the Applicative interface only allows us to handle computations that take place over a fixed structure. For example, we could write a parser for the TwoName data constructor since it gets applied to a fixed number of arguments, but we cannot write a general parser for the Name data type since it can be constructed in two different ways. In order to support this sort of parsing pattern, we will have to use something stronger than applicative.

Monads to the Rescue!

Surprise! The stronger abstraction that we are going to use is the parser Monad. Recall that the Monad type class exposes two function:

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

In general, monads can be used to sequence actions. We can think of each Parser as a single action which can be combined with other Parsers using monadic sequencing. Let's write a parser for first and last name using monads instead of applicative functors like we did above.

```
firstLast' :: Parser (ByteString, ByteString)
firstLast' = do
  fname <- name
  lname <- skipSpace *> name
  return (fname, lname)
```

This was a very simple parser, so using monads is sort of overkill. It is generally preffered to use applicative functors whenever possible. However, as we just saw, sometimes it is not possible to use applicative functors. In order to write the full name parser we need monads. The idea is simple, first we will parse two names, then we will attempt to parse a third name and decide what to do based on whether or not the third parse succeeds.

```
fullName :: Parser Name
fullName = do
  n1 <- name
  n2 <- skipSpace *> name
  mn <- skipSpace *> optional name
  case mn of
    Just n3 -> return $ ThreeName n1 n2 n3
    Nothing -> return $ TwoName n1 n2
```

Note that we used the optional :: Parser a -> Parser (Maybe a) function above. This is a function defined by Attoparsec that allows a parser to fail without terminating the entire computation.

By allowing sequencing, monads greatly increase the power of the Parser type. In particular, they allow decisions about the parsing computation to be made based on previous data that has been parsed.

Let's consider another example of a parser that requires the power of monads. Instead of just parsing a single name, we may need to parse a list of names. However, we may not know where the boundaries of the names are. For example, we might have the string "Haskell Brooks Curry Simon Peyton Jones". Should this string be parsed as the list

```
[TwoName "Haskell" "Brooks", TwoName "Curry" "Simon", TwoName "Peyton" "Jones"]
```

or the list

```
[ThreeName "Haskell" "Brooks" "Curry", ThreeName "Simon" "Peyton" "Jones"]
```

Obviously, the second one is correct! Haskell Brooks Curry and Simon Peyton Jones are both very important people in the world of functional programming! But how would a computer know that? There is no way to disambiguate the parser based on the input string alone. To fix this, we will include a list of

booleans stating whether or not each person in the list has a middle name. So, instead of attempting to parse the (ambiguous) string:

```
"Haskell Brooks Curry Simon Peyton Jones"
```

we will parse:

```
"[true, true] Haskell Brooks Curry Simon Peyton Jones"
```

This signifies that there are two names in the sequence of words and both of them have middle names. First, let's write some code to parse the boolean list:

```
bool :: Parser Bool
bool = do
  s <- word
  case s of
    "true"  -> return True
    "false" -> return False
    _       -> fail $ show s ++ " is not a bool"

list :: Parser a -> Parser [a]
list p = char '(' *> sepBy p comma <* char ')'
    where comma = skipSpace *> char ',' <* skipSpace

boolList :: Parser [Bool]
boolList = list bool
```

Note that the sepBy function creates a parser for a list of values that are separated by some other parser. In this case, the parser that separates the list elements is a comma surrounded by arbitrary spacing. We can now use this list of Bools to figure out how to parse the names. However, unlike the version of the fullName parser that itself decides whether to construct a Name using TwoName or ThreeName, we need to choose which parser to run based on the list of Bools.

```
names :: Parser [Name]
names = boolList >>= mapM bToP
  where bToP True  = ThreeName <$> sn <*> sn <*> sn
        bToP False = TwoName   <$> sn <*> sn
        sn = skipSpace *> name
```

Again, we see here that the output does not have a fixed structure. Given some input string, the output is a list of arbitrary length and each element of the list has one of two shapes that we must choose between based on some metadata. This sort of behavior is impossible to capture using Applicative Functors; we really need to use the power of monadic sequencing.

If you would like to see more examples of monadic parsers in real-world code, check out the Haskell Thrift Protocols.

Ejercicios (parte 2)

Leer, hacer los ejemplos y ejercicios del cápitulo siguiente:

- Real World Haskell: 16.using parsec

En particular probar parsers de CSV con algun archivo de datos abiertos de la Municipalidad de Cordoba o de OpenDataCordoba.

Más lectura

- Parsec
- Relación entre Functor/Applicative/Monad y los parsers
- Funtores, Aplicativos y Mónadas en imágenes
- Aprende Haskell: 11.Functores, 12.mónadas, 13.más mónadas
- El estado actual de las librerías de parsers en Haskell (Feb. 2016)