Monads

CIS 194 Week 07
4 March 2015

Suggested reading:

- **The Typeclassopedia**
- **LYAH Chapter 12: A Fistful of Monads**
- **LYAH Chapter 9: Input and Output**
- **RWH Chapter 7: I/O**
- **RWH Chapter 14: Monads**
- **RWH Chapter 15: Programming with monads**

```
import Control.Monad
```

Motivation

Despite their scary reputation, there's nothing all that frightening about monads. The concept of a monad started its life as an abstract bit of mathematics from the field of Category Theory (Monads are just monoids on the category of endofunctors!). It so happened that functional programmers stumbled upon it as a useful programming construct!

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)

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:

```
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.

```
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:

```
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 `(>>=)`.

There is actually a fourth method called `fail`, but putting it in the `Monad` class was a mistake, and you should never use it, so I won't tell you about it (you can **read about it in the Typeclassopedia** if you are interested). There are active plans afoot to change the Haskell standard libraries to remove `fail` from `Monad`.

`(>>=)` (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) `a`s.
- `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
```

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.

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 ()
```

List comprehensions

The monad for lists gives us a new notation for list building that turns out to be quite convenient. Building lists using monad-like operations is so useful that Haskell has a special syntax for it, called *list comprehensions*. It is best shown by examples:

```
evensUpTo100 :: [Int]
evensUpTo100 = [ n | n <- [1..100], even n ]

-- this next one is a bit inefficient, but it works
oddPerfectSquares :: [Int]
oddPerfectSquares = [ n | n <- [1..100]
                        , odd n
                        , root <- [1..10]
                        , root * root == n ]

cartesianProduct :: [a] -> [b] -> [(a,b)]
cartesianProduct as bs = [ (a,b) | a <- as, b <- bs ]

combine :: (a -> b -> c) -> [a] -> [b] -> [c]
combine f as bs = [ f a b | a <- as, b <- bs ]

-- inefficient again
primes :: [Int]
primes = [ p | p <- [2..]
             , all ((/= 0) . (p `mod`)) [2..p-1] ]
```

List comprehensions work just like set-builder notation you may have learned in a high-school math class. In a list comprehension, the statements to the right of the | are carried out, in order. A statement with a `<-` selects an element from a list. Statements without `<-` are Boolean expressions; if the expression is `False`, then the current choice of elements is thrown out. (`let` statements, just like in `do` notation – without an `in` – are also allowed.)

In turns out that there is a straightforward translation from list comprehensions to `do` notation:

```
[ a | b <- c, d, e, f <- g, h ]
```

is exactly equivalent to

```
do b <- c
   guard d
   guard e
   f <- g
   guard h
   return a
```

Note that, in the translation, lists aren't mentioned anywhere! With the GHC language extension `MonadComprehensions`, you can use list comprehension notation for *any* monad.

Generated 2015-05-17 10:25:31.874664