Monads

CIS 194 Week 12
8 April 2013

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

Motivation

Over the last couple of weeks, we have seen how the `Applicative` class allows us to idiomatically handle computations which take place in some sort of "special context"—for example, taking into account possible failure with `Maybe`, multiple possible outputs with `[]`, consulting some sort of environment using `((->) e)`, or construct parsers using a "combinator" approach, as in the homework.

However, so far we have only seen computations with a fixed structure, such as applying a data constructor to a fixed set of arguments. What if we don't know the structure of the computation in advance – that is, we want to be able to decide what to do based on some intermediate results?

As an example, recall the `Parser` type from the homework, and assume that we have implemented `Functor` and `Applicative` instances for it:

```
newtype Parser a = Parser { runParser :: String -> Maybe (a, String) }

instance Functor Parser where
  ...

instance Applicative Parser where
  ...
```

Recall that a value of type `Parser a` represents a *parser* which can take a `String` as input and possibly produce a value of type `a`, along with the remaining unparsed portion of the `String`. For example, a parser for integers, given as input the string

```
"143xkkj"
```

might produce as output

```
Just (143, "xkkj")
```

As you saw in the homework, we can now write things like

```
data Foo = Bar Int Int Char

parseFoo :: Parser Foo
parseFoo = Bar <$> parseInt <*> parseInt <*> parseChar
```

assuming we have functions `parseInt :: Parser Int` and `parseChar :: Parser Char`. The `Applicative` instance automatically handles the possible failure (if parsing any of the components fail, parsing the entire `Foo` will fail) and threading through the unconsumed portion of the `String` input to each component in turn.

However, suppose we are trying to parse a file containing a sequence of numbers, like this:

```
4 78 19 3 44 3 1 7 5 2 3 2
```

The catch is that the first number in the file tells us the length of a following "group" of numbers; the next number after the group is the length of the next group, and so on. So the example above could be broken up into groups like this:

```
78 19 3 44   -- first group
1 7 5        -- second group
3 2          -- third group
```

This is a somewhat contrived example, but in fact there are many "real-world" file formats that follow a similar principle—you read some sort of header which then tells you the lengths of some following blocks, or where to find things in the file, and so on.

We would like to write a parser for this file format of type

```
parseFile :: Parser [[Int]]
```

Unfortunately, this is not possible using only the `Applicative` interface. The problem is that `Applicative` gives us no way to decide what to do next based on previous results: we must decide in advance what parsing operations we are going to run, before we see the results.

It turns out, however, that the `Parser` type *can* support this sort of pattern, which is abstracted into the `Monad` type class.

Monad

The `Monad` type class is defined as follows:

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

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

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

This should look familiar! We have seen these methods before in the context of `IO`, but in fact they are not specific to `IO` at all. It's just that a monadic interface to `IO` has proved useful.

`return` also looks familiar because it has the same type as `pure`. In fact, every `Monad` should also be an `Applicative`, with `pure = return`. The reason we have both is that `Applicative` was invented *after* `Monad` had already been around for a while.

`(>>)` 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).

`(>>=)` (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*. It has also been proposed to call them *mobits*. 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 a mobit 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 :: Parser a` is a computation which implicitly consumes part of a `String` and (possibly) 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 choose what to do next based on the results of previous computations.

So all `(>>=)` really does is put together two mobits 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 mobit 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`. 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". I wish I was joking.

Some 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

exM1 = return 7 >>= check >>= halve
exM2 = return 12 >>= check >>= halve
exM3 = return 12 >>= halve >>= check
```

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

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

A simple example:

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

```
exL1 = [10,20,30] >>= addOneOrTwo
```

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; in the case of `Parser` it means to run all the parsers on sequential parts of the input (and succeed only if they all do).

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

And now we are finally in a position to write the parser we wanted to write: it is simply

```
parseFile :: Parser [[Int]]
parseFile = many parseLine

parseLine :: Parser [Int]
parseLine = parseInt >>= \i -> replicateM i parseInt
```

(`many` was also known as `zeroOrMore` on the homework).