The State Monad

The state monad is a built in monad in Haskell that allows for chaining of a state variable (which may be arbitrarily complex) through a series of function calls, to simulate stateful code. It is defined as:

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

which, honestly, can be pretty bewildering at first. There are a few things going on here for starters. First, if you haven't seen it before, *newtype* is a lot like *data*, except for some details that don't matter right now; just think of it as *data* for the moment. So what's "in" our data type? It's a function! And we're implicitly defining a function to extract our inner function from the data type: that function is called runState.

Now let's think about that inner function. It's type is *s -> (a,s)*. Essentially, it's a type for any function that takes some initial state and then returns a tuple of *(regular return value, new state)*. That makes sense. And because of partial applications, currying, etc. we can actually write something like the following:

```
data ProgramState = PS [Int] -- this is what our state "is"
foo :: Int -> String -> State (Int, ProgramState)
foo = ...
```

So, since our function ends with State (Int, ProgramState), we can infer that the type of the state we would be passing around, if using this function, would be ProgramState. The really cool bit though is that it **looks** like we're using State as a regular data type, but since it's actually a function type, it's probably better to think of this as a shorthand way of adding " -> s -> (a,s) " to the end of our type declaration for foo. In other words, we could rewrite foo as this:

```
--foo :: Int -> String -> State (Int, ProgramState)
  foo :: Int -> String -> ProgramState -> (Int, ProgramState)
```

In other words, saying we're returning a State, is actually saying that we're returning a function. But, since the partial function will get automatically applied if we call foo with a third parameter (the initial state), we're really just saying that foo is a function that takes an int, a string, a previous state, and then returns a tuple of *(return value, new state)*. Pretty cool. So, how do we use foo? Well, we have to use *runState* to extract the inner function so that we get it's return value! Observe:

```
initialState = PS []
doSomething = runState (foo 2 "blah") initialState
```

This is kind of confusing because Haskell collapses the partial applications for you, but it'll grow on you :) Essentially though, *doState* will return a tuple of type *(Int, ProgramState)*.

The state monad defines a function *get* that retrieves the state, and *put* which lets you overwrite the current state that's being passing along. Thus, a (slightly) more realistic example might be the following:

```
doSomethingCooler = runState (do (PS currentStack) <- get
                                 let top = head a
                                 returnFromFoo <- foo top
"myString"
                                 put (returnedFromFoo :
currentStack)
                                 return top
                              ) initialState
```

The successive functions get chained together by the state monads definition of >>= to pass along the current state. Each line where we call a function "in the state monad" behaves as follows: the arguments we give it on that line are applied, yielding a **partially** applied function. This is because each of the state monad functions yields a State function (which still needs an argument of type *s* before it can yields it's result tuple. The definition of >>= makes it so that all the functions in the do loop get passed the current state as that final parameter, execute, and then, once they return, the next function in the chain gets called using the previously return state.

Lastly, we could write a state monad function as follows:

```
bar = do a <- get
         c <- someOtherFunctionWeCouldWrite
         put c
         return ()
```

Bar's type is:

```
bar :: State ((), s)
```

where s might depend on the definition of the function we're not writing, but otherwise could just be free.

The really cool thing though is what this **is**. Basically, this do loop gets collapsed into one function, of type *State* (e.g. of type *s -> (a,s)*. So, it takes an initial state, and returns a tuple. To run it, we extract the "inner function" from it's State wrapping using runState, and call it on some initial state.

```
runState bar initialState
```