CS 421 --- The State Monad

Introduction and Objectives

At this point, you have seen monad definitions for the `Maybe` and list types. I'm not going to assume you are completely comfortable with them yet, but it will help if you are able to reproduce the monad definitions for those two types from memory.

As you recall, a monad is a container that has two operations defined for it. The `pure` function takes an item and places into a monad. The bind operation takes a container and a function, unpacks the container and applies the function to it, resulting in a new container. Sometimes the bind operation needs to do some processing first.

The thing that makes the State Monad difficult to understand at first is that the container isn't a simple data type --- the container is a function. But *that's all that changes*. The `return` and bind operations follow the same kind of logic as before.

Review of Record Syntax

The State Monad implementation uses records, so a quick refresher might be helpful. Consider the following code:

```
data Foo = Bar { x : Integer, y : String }
  deriving Show
```

This defines a new type `Foo` with constructor `Bar`. The content of `Bar` is a record containing two fields `x` and `y`.

To create instances of this type you have two choices, which you can see in the code snippet below. The first style has the programmer write out the record explicitly, the second style is more implicit, but requires you to give the arguments in the same order they were declared.

```
-- Explicit Style
t1 = Bar { x = 10, y = "hi" }
t2 = Bar { y = "hi", x = 10 }
-- Implicit Style
t3 = Bar 10 "hi"
```

Note that `t1`, `t2`, and `t3` produce the same value.

To access the data of a record, use the field names as functions. In this examples, we have `x :: Foo -> Integer` and `y :: Foo -> String`.

```
Prelude> x t1
10
Prelude> y t1
"hi"
Prelude> map x [t1,t2,t3]
[10,10,10]
```

Representing State

Here is our representation of state.

```
data State a b = State { runState :: a -> (b,a) }
```

Things are going to be notationally complicated, because there are four different things we will want to call ``state''. We have a type named `State`, a constructor named `State`, and we also have the `a` parameter to the `a -> (b,a)` function that represents a state.

To distinguish these things, we will refer to the ``State type'' or ``state instance'' when we mean an instance of `State a b`, and ``state parameter'' when we mean the input to the `runState` function.[1]

[1] If we use the word "state" without any qualifications, it means that we think the meaning is clear from context, or else we are talking about the concept of state in general.

Let's represent a stateful computation equivalent to the C code `2 * i++`. Assuming that `i` has value 10, this code will do two things: it will return the value 20, and increase the value of `i` to 11.

In Haskell, a rough equivalent would be this code:

```
ex1 = State (\i -> (2*i, i+1))

Prelude> runState ex1 10
(20,11)
```

Thus, the value is in the first part of the tuple (the ``value part''), and the resulting state is in the second part (the ``state part'').

The Type Classes

In order to build a monad out of this we have to provide the `Functor` and `Applicative` definitions.

Functor

Supposing we have a function `f` and a state instance `st`, we want a new state instance that has `f` applied to the value part of its tuple. So if our instance was `State (\a -> (b,a))`, then we would want to get back `State (\a -> (f b, a))`. Here is the implementation.

```haskell
instance Functor (State a) where
  fmap f st = State (\s -> let (b,s') = runState st s
                           in (f b, s'))
```

We declare that `(State a)` is a functor.[2] Since `a` is the type of the state part, this indicates that the functor will operate on the value part.

For `fmap`, we are given `f` and a state instance `st`, and want to return a new state instance where `f` has been applied to the value. So the new instance takes a state and uses it to call `runState` on `st`. This gives us a new value `b` and new state `s'`. We then return the updated tuple `(f b, s')`.

You should type this in and play with it until it makes sense to you. You should also check that you understand the types of all the variables in the code. Once you understand those, you will be able to reproduce this definition yourself if you ever need it.

[2] We need to say `instance Functor (State a)` because a `Functor` takes an argument of kind `* -> *`, but `State` has two type parameters, making it of kind `* -> * -> *`. Filling in the first type with a type variable `a` fixes that.

Applicative

An applicative is a step up from functor. What we will have now is two state instances. We will call them `sf` and `sx`. The instance `sf` has a function in its value part, and `sx` has an argument to that function. What we want is a new state instance in which the `f` in `sf` has been applied to the `x` in `sx`.

Like functor, once you know the types you will find there is only one way that make sense to define the applicative instance for `State`.

```haskell
instance Applicative (State a) where
  pure x = State (\s -> (x,s))
  sf <$> sx = State (\s -> let (f,s') = runState sf s
                               (x,s'') = runState sx s'
                           in (f x, s''))
```

The definition of `pure` is it just puts its argument into a `State`.

For the apply operation, we run an initial state through `sf` to get `f` and a new state `s'`, and then use `s'` with `sx` to get `x` and final state `s''`.

Again, run this code and play with it until you understand it. One thing you might want to try; what if you didn't use `s'` in the `runState sx` part, and used `s` instead?

The Monad

We are now ready to show you the monad.

Recall that the operation bind, written `s >>= f`, will take a state instance `s` and unpack it. It will apply `f` to the result, which will result in a new state instance. Assuming you know the types, maybe you can try deriving this yourself before reading the code! The discussion continues on the next page.

```haskell
instance Monad (State a) where
  return = pure
  sx >>= f = State (\s -> let (x,s') = runState sx s
                          in runState (f x) s')
```

To ``unpack'' `sx`, we need to run it as a state to get a value `x` and new state `s'`. If we apply `f` to `x`, that returns a new state instance. We don't want nested state instances, so we call `runState` on that too, using `s'` as the state argument.

Get and Put

Now that we have this monad, we want two helper functions that can communicate the state part to the value part and *vice versa*.

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

The `get` function copies the state component into the value component, and the `put` function replaces the state component with `x`. The value component is called a *unit*, and plays a role similar to `void*` in C++. It is the only value in the type, also pronounced *unit*, and the type itself is written as `()`.

Colophon

This document was written in Emacs with the Spacemacs extensions and the AucTex package. It was compiled using Lua LaTeX and the `tufte-book` package. The body text is set in the *Equity* font, and the headers are set in the *Concourse* font. Both these fonts are available from Matthew Butterick. The source code is set in the *Inconsolata* font.