The State Monad

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Describe the `newtype` keyword and the record type we use for representing state.
- Implement the `pure` operation for the state monad.
- Implement the `bind` operation for the state monad and trace an execution.
- Define `get` and `put` to allow direct manipulation of the stateful part of the monad.

Defining the Types

- The incoming `Integer` is the state.
- The output tuple is a result and a state.

```
1 ex1 :: Integer -> (Integer, Integer)
2 ex1 s = (s * 2, s+1)
3
4 *Main> ex1 10
5 (20,11)
```

Encapsulation

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 ex2a :: State Integer Integer
4 ex2a = State { runState = ex1 }
5
6 ex2b :: State Integer Integer
7 ex2b = State ex1
8
9 *Main> runState ex2a 10
10 (20,11)
11 *Main> runState ex2b 10
12 (20,11)
```

Functor

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 ex2b :: State Integer Integer
4 ex2b = State ex1
5
6 inc x = x + 1
7
8 *Main> runState ex2a 10
9 (20,11)
10 *Main> runState (fmap inc ex2a) 10
11 (21,11)
```

Functor Definition, 1

- Remember, `Functor` takes a container type.
- Think of `(State s a)` as a container that has values of type `a` in it.
- We need to define `fmap`.

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 instance Functor (State s) where
4    fmap :: (a -> b) -> (State s a) -> (State s b)
5    fmap f g = ...
```

Functor Definition, 2

- We need to return a `State` ...

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 instance Functor (State s) where
4    fmap :: (a -> b) -> (State s a) -> (State s b)
5    fmap f g = State ...
```

Functor Definition, 3

- That contains a function ...

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 instance Functor (State s) where
4    fmap :: (a -> b) -> (State s a) -> (State s b)
5    fmap f g = State (\s1 -> ...
```

Functor Definition, 4

- That contains a function ...

```
1 newtype State s a = State { runState :: s -> (a,s) }
2
3 instance Functor (State s) where
4    fmap :: (a -> b) -> (State s a) -> (State s b)
5    fmap f g = State (\s1 -> let (x,s2) = runState g s1
6                             in (f x, s2))
```

Applicative

- Similar reasoning gives us the `Applicative` functor.

```
instance Applicative (State s) where
  pure x = State (\s -> (x,s))
  -- (<*>) :: State s (a->b) -> State s a -> State b
  f1 <*> x1 = State (\s -> let (f,s2) = runState f1 s
                               (x,s3) = runState x1 s2
                           in (f x,s3))
```

The Monad

```haskell
instance Monad (State s) w
 return = pure
 -- x :: State s a
 -- f :: a -> State s b
 -- output :: State s b
 x >>= f = State (\s -> let (y,s2) = runState x s
                            (z,s3) = runState (f y) s2
                        in (z,s3))
```