The State Monad: A Tutorial for the Confused?

OCT 24TH, 2009 12:16 AM

I've written this ~~brief~~ tutorial on haskell's `State` monad to help bridge some of the elusive gaps that I encountered in other explanations I've read, and to try to cut through all the sticky abstraction. This is written for someone who has a good understanding of the `Maybe` and `List`monads, but has gotten stuck trying to understand State. I hope it's helpful!

The Data Declaration:

To understand a monad you look at it's datatype and then at the definition for bind (`>>=`). Most monad tutorials start by showing you the data declaration of a `State s a` in passing, as if it needed no explanation:

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

But this *does* need explanation! This is crazy stuff and nothing like what we've seen before in the list monad or the Maybe monad:

1. The constructor `State` *holds a function*, not just a simple value like Maybe's `Just`. This looks weird.
2. Furthermore there is an accessor function `runState` with a *weirdly imperative-sounding name*.
3. Finally, there are two free variables on the left side, not just one.

Yikes! Let's try to get our head on straight and figure this out:

First of all the State monad is just *an abstraction for a function that takes a state and returns an intermediate value and some new state value*. To formalize this abstraction in haskell, we wrap the function in the newtype `State` allowing us to define a `Monad` class instance.
Stepping back from the abstract and conceptual, what we have is *the* `State` *constructor acting as a container for a function* `:: s -> (a,s)`,

while the definition for bind just provides *a mechanism for "composing" a function* `state -> (val,state)` *within the* `State` *wrapper*.
Just as you can chain together functions using `(.)` as in `(+1) . (*3) . head :: (Num a) => [a] -> a`, the state monad gives you `(>>=)` to *chain together functions* that look essentially like `:: a -> s -> (a,s)`into a single function `:: s -> (a,s)`.
Let's bring the discussion back to actual code and try to make sure we understand those three points of weirdness outlined above. Here's a stupid example of a function that can be "contained" in our state type:

```
-- look at our counter and return "foo" or "bar"
-- along with the incremented counter:
fromStoAandS :: Int -> (String,Int)
fromStoAandS c | c `mod` 5 == 0 = ("foo",c+1)
               | otherwise = ("bar",c+1)
```

If we just wrap that in a `State` constructor, we're in the State monad:

```
stateIntString :: State Int String
stateIntString = State fromStoAandS
```

But what about `runState`? All that does of course is *give us the "contents" of our State constructor*: i.e. a single function `:: s -> (a,s)`. It could have been named `stateFunction` but someone thought it would be really clever to be able to write things like:

```
runState stateIntString 1
```

See, all we've done there is used `runState` to take our function (`fromStoAandS`) out of the State wrapper; it is then applied to its initial state (`1`). We would do this `runState` business **after building up our composed function with `(>>=)`, `mapM`, etc**.
That leaves point 3 unanswered. Let's start exploring the instance declaration for State.

The Instance Declaration

We'll start with the first line:

```
instance Monad (State s) where
```

We create a Monad instance for (State s) not State. You can think of this as a **partially-applied type**, which is equivalent to a partially-applied function:

```
(State)     <==> (+)
(State s)   <==> (1+)
(State s a) <==> (1+2)
```

So (`State s`) is the `m` in our `m a`. This means the *type* of our state will remain the same as we compose our function with `(>>=)`, whereas the intermediate values (the `a`s) may well change type as they move through the chain.

Before we move on to the meat of the instance declaration, I'd like to get your mind calibrated to look at the definitions for `return` and `(>>=)`:

Whenever you see `m a`, as in

```
return :: (Monad m) => a -> m a
```

…remember that `m a` is actually

```
State s a
```

…and when you remember `(State s a)`, *think*

```
(s -> (s,a))
```

So in your mind, `m a` becomes `function :: s -> (a,s)` everywhere you see it. Just forget about the silly `State` wrapper (the compiler does)!

The definition of return and Bind

Let's wet our feet with the definition for `return`:

```
return a = State $ \s -> (a, s)
```

All return does is take some value a and make a function that takes a state value and returns (value, state value). If we ignore the whole State wrapping business, then return is just `(,) :: a -> b -> (a, b)`

Now recall the definition of bind:

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

Which in our case is:

```
(>>=) :: State s a        ->
         (a -> State s b) ->
         State s b
```

And which is just a silly abstraction for the **super special function composition** that's going on, which looks like:

```
(>>=) :: (s -> (a,s))      ->
         (a -> s -> (b,s)) ->
         (s -> (b,s))
```

So on the left hand side of (`>>=`) is a function that takes some initial state and produces a `(value,new_state)`. On the right hand side is a function that takes that value and that new_state and generates it's own `(new_value, newer_state)`. The job of bind is simply *to combine those two functions into one bigger function from the initial state to `(new_value,newer_state)`*, just like the simple function composition operator `(.) :: (b -> c) -> (a -> b) -> a -> c`

At this point, we can show you bind's definition:

```
m >>= k = State $ \s -> let (a, s') = runState m s
                        in runState (k a) s'
```

You can work through that on your own, keeping in mind that we're doing function composition here. The main thing to remember is that the `s` at the top, right after `State`, **won't actually be bound to a value until we unwrap the function with `runState` and pass it the initial state value**, at which point we can evaluate the entire chain.

A Final Note About The State Monad with do Notation

`State` is often used like this:

```
stateFunction :: State [a] ()
stateFunction = do x <- pop
```

```
               pop
               push x
```

Remember that the functions above are desugaring to `m >>= \a-> f...`or if there is no left arrow on the previous line: `m >>= \_-> f...` That `a` in there is an intermediate value, the `fst`in the tuple. The push function might look like:

```
push :: State [a] ()
push a = State $ \as -> (() , a:as)
```

The function doesn't return any meaningful `a` value, so we don't bind it by using the `<-`. For more work with do notation and some fine pictures, see Bonus's post on something awful.

Getting more general: `StateT` and `MonadState`

added by request 2/16/2012

So now you're an expert on the State monad. Unfortunately (actually it's a good thing) the `State` type I describe above isn't in any of the standard libraries. Instead `State` is defined in terms of the `StateT` monad transformer here.

```
type State s = StateT s Identity
```

If you haven't seen Monad transformers before, see if you can figure out how `StateT s Identity` is equivalent to `State s` as I defined it above. Just follow the links on hackage.

You might also have noticed a typeclass called `MonadState`, also in the mtl package, and be wondering how that fits in. Here's what it looks like; I'll explain the odd-looking bits in a moment:

```
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()
```

Whereas above we were discussing `State`, a concrete data type, `MonadState` is a new *typeclass*for types that are monads and for which we can define `get` and `put` operations.
The class allows for a variety of Monad Transformer "stacks" that use state-passing to share a common interface for the basic state operations.

Small aside: since I neglected doing this above, this is how we would define `get` and `put` as regular functions (not methods of `MonadState`) on our `State` type:

```
-- return the state value being passed around:
get :: State s s
get = State $ \s -> (s,s)

-- replace the current state value with 's':
put :: s -> State s ()
put s = State $ \_ -> (s,())
```

Quite useful. See if you can understand how those work now that you've got a better grasp of `State`. Excercise: define `modify :: (s->s) -> State s ()`.
Back to `MonadState`: If this class and its instances look confusing to you, you need to know about two extensions to haskell-98 (both of which are very common and sticking around): Multi-Parameter Type Classes, and Functional Dependencies.
In GHC you can enable both these extensions in your source by putting this at the top:

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
```

Multi-parameter Type Classes

Ignoring the functional dependencies (a.k.a "fundeps") in `MonadState` you get:

```
class Monad m => MonadState s m where
```

In which…

- `MonadState` is the name of the class
- `s` is the first class type variable (in this case the type of our state)
- `m` is the second type variable (the type of our state-like monad, e.g. `State s`)

Multi-parameter type classes are tricky in that they define a *relationship between multiple types*with associated operations. You can read more about them in the GHC docs.

Functional dependencies

With multiple parameters in a single class, you can often end up with instances that are disallowed or difficult to use because they are ambiguous. Functional Dependencies help resolve ambiguity by allowing a way to specify that certain type parameters can be determined by knowledge of one or more of the other parameters.

In the case of `MonadState`, the part of the class declaration that looks like:

```
| m -> s
```

says that the type of `m` (say `StateT Int IO`) *uniquely determines* the type of `s` (Int). Again please see the linked GHC docs for details; the `Collects` example used is very similar to `MonadState`.
Putting it all together, the class declaration might read in english as:

> *The relation of types `m` and `s` where `m` uniquely determines `s` is in the `MonadState` class.*

A `MonadState` instance for State

Finally here's what the `MonadState` instance for the `State` type we've been discussing would look like (but again, it's not because `mtl` builds around the more general `StateT`):

```
instance MonadState s (State s) where
    get = State $ \s -> (s,s)
    put s = State $ \_ -> (s,())
```