The State Mondad slowly dissected

Martin Drautzburg

February 15, 2014

Contents

1 The type

The type we are dealing with is the following:

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

1.1 Record syntax

To get a feeling for what this *State* type means, we will construct such a type ourselves. First we need to understand the record syntax used here.

Record syntax allows to define a tuple together with access functions to retrieve specific components of the tuple. To understand the motivation behind it, we will first try without record syntax.

1.1.1 Tuples

Consider a Pair of *Int* and *String*, where want to refer to the first component as `foo` and to the second component as `bar`. With such a type we can write access functions which retrieve either the first or second component and which have a name which reflects the name of the component.

```
type PairTuple = (Int, String)
fooTuple (foo,_) = foo
barTuple (_,bar) = bar
```

If we want to modify one of the components, we need additional functions like

```
modFooTuple::Int→PairTuple→PairTuple
modFooTuple foo (_, bar) = (foo, bar)

modBarTuple::String→PairTuple→PairTuple
modBarTuple bar (foo,_) = (foo, bar)
```

As an exercise let's create a Pair and then modify each of the components and retrieve the bar compnent

```
ex1 = barTuple $ (modBarTuple "changed") $ (modFooTuple 2) $ (1,"init")
```

```
*Main> ex1
"changed"
*Main>
```

1.1.2 Records

It seems unneccessary to define these four functions, because all the system needs to know is the types and names of the components. This is where record syntax comes in. We cannot get away with a simple **type** synonym anymore, but must define a new type, e.g. using the **data** keyword.

```
data PairRecord = PR {foo::Int, bar::String} deriving (Eq,Show)
```

This definition magically creates two functions `foo` and `bar` which correspond to the `fooTuple` and `barTuple` functions we created ourselves in the previous example.

```
*Main> :t foo
foo :: PairRecord -> Int
*Main> :t bar
bar :: PairRecord -> String
```

Using record syntax, we can create *PairRecords* ...

```
pr1 = PR {foo=1, bar="init"}
```

retrieve the components ...

```
*Main> foo pr1
1
*Main> bar pr1
"init"
```

and even update records

```
*Main> pr1 {bar = "changed"}
PR {foo = 1, bar = "changed"}
*Main>
```

where an *update* is of course not a real update, but the construction of a new PairRecord with one or more components changed.

1.2 A type with a function

The *State* type however, does not consist of simple types like *Int* and *String* but wraps around a function. To get a feeling of what this does, let's again create such a Type ourselves.

```
data FuncRecord = FR{run :: Int→Int}
```

To create such a record we must pass a function into the data constructor *FR*. This function could be an already existing function, or one which we create on-thy-fly, using a lambda expression. Let's try both:

```
inc x = x+1
frInc = FR inc
frDouble = FR  (λx → x*2)
```

So what we can do with such things? We know we can retrieve the `run` component, which will give us a function. This function can then be applied to an argument.

```
*Main> (run frInc) 5
6
*Main> (run frDouble) 5
10
```

2 Monads

A Monad can often be seen as a *something of something else*. If you have a *List* of *Int*s, then the *something* is *List* and the *something else* is *Int*.

In type signatures you often see a thing like *M* `a` (or `m a`). Here *M* is the *something* and `a` is the *something else*. The *M* is called a *type constructor* as it creates a new type from a base type `a`. If there was a type constructor *List*, then a List of Ints would be written as *List Int*.

As far as monadic operations are concerned, the *Int* is of little concern. The monadic operators like `return` and `>>= ("bind")`are spefic to the *type constructors* only.

This kind of abstraction is very common in Haskell. E.g. the `reverse` operation on Lists works on Lists of any types. It knows nothing about the type of elements in the List. This is the way it should be: "A function which inverses a list of bananas knows nothing about bananas"

However, Mondads are not about reversing, but about chaining. It is a good idea to know the type of the bind operator `>>=` by heart.

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

This just sais, that `(>>=)` creates a new monadic value from an old monadic value with the help of a function. It sais nothing about *how* this is done. There are in fact several options, but mostly one of them is overwhelmingly more useful than the others.

2.1 A first monad

Let's try to roll our own monad from our *FuncRecord* from above. We must change a number of things. First a monad needs a type variable (the *of something else*). So instead of functions from Int to Int, we use functions from Int to some type `a`. Then we must rename a number of things, to avoid name clashes with our *FuncRecord*

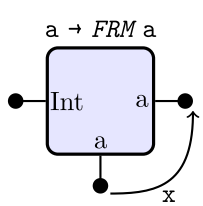
```
data FuncRecordMonad a = FRM{runm :: Int→a}

-- some examples:
frmInc = FRM (λx → x+1)
frmDouble = FRM (λx → x*2)
```

The experiments we did with the *FR* type will work with *FRM* just as well, namely `(runm frmInc) 5` and `(runm frmDouble) 5`

To make *FRM* a monad, we must define the two function `return` and `(>>=)`.

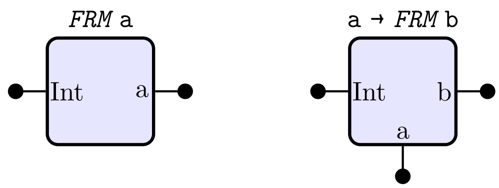
2.1.1 Return

The `return` function `return :: Monad m => a → m a` takes some value and constructs a Monad from it. In our case thus would be a function, which returns a value of type `a`. There aren't too many options, because the only variable whose type is definitly an `a` is the argument to `return`, whose actual type is not known. So our only option is to create a function, which returns this value regardless of its input, somthing like $return\, x = FRM(\lambda\_ \rightarrow x)$



2.1.2 Bind

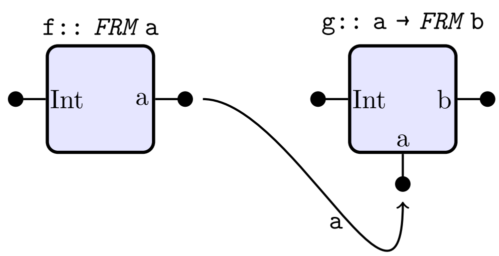
Now we ask ourselved the question: if we have such a function, wrapped in *FRM* and a function which creates another such such function (the `a → M b`), how can we construct a new *FRM* in a way which makes some sense?



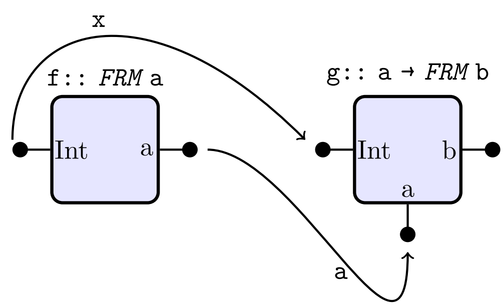
To implement `(>>=)` we must combine these into a single function. There aren't too many options. Remember that

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

The function `a → M b` expects an `a`. We might feel tempted to pass it some constant, but we do not really know its type. We only know it is an "`a`". So we cannot do this. The only way we can feed it an `a` is to take the return value from the first *FRM*.



We still have two *Int* inputs, but the resulting function should have only one. We could pass a constant *Int* to the second function. However the chosen value would be difficult to justify. Instead we will pass the agrument to the first function also to the second function.



The result indeed has the type *M* `b`, i.e. *FRM* `b`, a function from *Int* to `b`. To write this in proper Haskell, we first apply `f` to `x` by means of `(runm f) x)`. This gives us some value of type `a`. We pass this value to `g` which gives us another *FRM* `b`. Finally we apply this new function to the same `x` and get a value of type `b`. So we have constructed a new function `f2` from `f` and `g` and the only thing left to do, is to wrap in inside *FRM*.

```
instance Monad FuncRecordMonad where
        return x = FRM (\_ →  x)
        f >>= g = FRM f2
                where
                    f2 x = (runm (g ((runm f) x))) x
```

2.1.3 Doing

We designed our monad with no specific purpose in mind. But let's explore what it does anyways.

Return alone would create a function which always returns the same value To actually run this function we must unwrap it with `runm`.

```
f2 = runm frm
        where
            frm = return "42"
```

```
 *Main> :t f2
 f2 :: Int -> [Char]
 *Main> f2 1
 "42"
 *Main> f2 2
 "42"
```

Now let's try the chaining. We must invent that second function `g :: a →` *FRM* `b`, where the b-typed value inside *FRM* is a function from *Int* to some type `b`.

```
f3 = runm frm
        where
            frm = return "42" >>= λa → FRM (λx → (show x) ++ a)
```

So this function returns its agument suffixed by the String 42.

```
*Main> f3 1
"142"
*Main> f3 99
"9942"
```

Now, let's try to rewrite `f3` using do-notation. First, let's try to get rid of the value constructor *FRM* by using `return`. Furthermore let's get rid of the lambda by making it an argument to `f3`. Finally let's not unwrap right away, using `runm` but return a *FuncRecordMonad* and leave the unwapping to the caller.

```
f3a x = return "42" >>= λa → return ((show x) ++ a)

-- This translates to do-notation
f3b x = do
    a ← (return "42") :: FuncRecordMonad String
    return ((show x) ++ a)
```

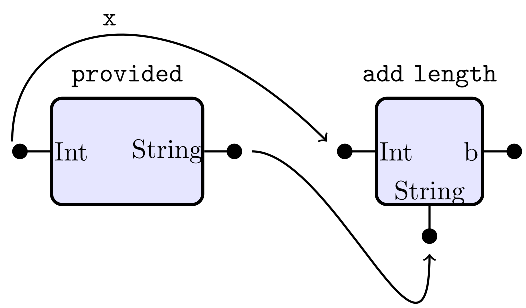
The type of `f3b` is now `f3b :: Show a => a → FuncRecordMonad [Char]`. If we pass it one parameter, we get `f3b 99 :: Num a => FuncRecordMonad [Char]` from which we can extract the function `runm (f3a 99) :: Num a => Int → [Char]` and when we finally call this function we get:

```
*Main> (runm (f3b 99)) 666
"9942"
```

Note that the final argument 666 is actually ignored. The result only depends on `x=99`, the parameter we passed first to f3b. Let's try to construct a more intelligent *FuncRecordMonad*, one which transforms an existing *FuncRecordMonad*.

```
f4 frm = do
    a ← frm :: FuncRecordMonad String
    b ← FRM $ λx → (x + length a)
    return b
```

`f4` takes some wrapped *Int → String* function and returns a function, which adds the length of the String to its *Int* parameter.




```
*Main>  (runm $ f4 (FRM $ \x -> show x)) 120
123
*Main>  (runm $ f4 (FRM $ \x -> take x "lkjlkjl")) 10
17
```