Expressing State in Haskell

A challenge for those new to Haskell and its lack of mutation is how Haskell handle state. In this section we discuss how this can be accomplished. The main idea is "weave your state through the computation".

A State Example: Statement Interpretation

As an illustration of this idea, let us imagine a small programming language. It has expressions that perform basic arithmetic (addition and multiplication), but also allows us to store values in variables as well as to print values. This is done via statements. Here is a listing of the basic types.

```
type Symbol = String  -- alias for strings when used as language symbols/variables.
data Expr = Numb Double      -- number literal
          | Var Symbol       -- variable lookup
          | Add Expr Expr    -- expression for addition
          | Prod Expr Expr   -- expression for multiplication
data Stmt = Assign Symbol Expr  -- variable assignment
          | Seq Stmt Stmt       -- statement followed by another statement
          | Print Expr          -- print the result of an expression evaluation
          | PrintMem            -- print all stored values
```

A program is simply a `Stmt` value, which can in turn be a sequence of `Stmt`s using the `Seq` constructor. For example here is one such program:

```
Seq (Assign "x" (Add (Numb 2) (Numb 4))) $    -- x <- 2 + 4
Seq (Print $ Var "x") $                        -- print x
PrintMem                                       -- print all memory
```

In order to execute such a program, we need to maintain a "memory" of stored values for the variables:

```
type Value = Double     -- Doubles are the only possible values in this language
type Memory = [(Symbol, Value)]

store   :: Symbol -> Value -> Memory -> Memory
store s v []              = [(s, v)]
store s v ((s',v'):rest) = case compare s s' of
   LT -> (s, v):(s', v'):rest
   EQ -> (s, v):rest
   GT -> (s', v'):store s v rest

lookup :: Symbol -> Memory -> Maybe Value
lookup s []               = Nothing
lookup s ((s', v'):rest) = case compare s s' of
   LT -> Nothing
   EQ -> Just v'
   GT -> lookup s rest
```

Now we need to write the main functions, one to evaluate expressions and one to evaluate statements. The challenge is this: In order for them to do their work, these functions must have the current state of the Memory available to them, and in the case of the statement must also be able to *change* the value of Memory by returning an updated Memory. Therefore the “types” of these functions might be as follows:

```
evalExpr :: Expr -> Memory -> Value
evalStmt :: Stmt -> Memory -> (IO (), Memory)
```

Note the distinction: expressions return values, while statements interact with the user (e.g. print something).

Let’s consider how evalExpr may be implemented. It should be a simple set of cases for each type of expression:

```
evalExpr :: Expr -> Memory -> Value
evalExpr (Numb x) _ = x
evalExpr (Var s) mem =
    case lookup s mem of
        Nothing -> error ("Cannot␣find␣symbol:␣" ++ s)
        Just v  -> v
evalExpr (Add e1 e2) mem = v1 + v2
    where v1 = evalExpr e1 mem
          v2 = evalExpr e2 mem
evalExpr (Prod e1 e2) = v1 * v2
    where v1 = evalExpr e1 mem
          v2 = evalExpr e2 mem
```

Next we would have evalStmt, which is trickier as it often has to *update* the memory. It must therefore return the updated memory:

```
evalStmt :: Stmt  -> Memory -> (IO (), Memory)
evalStmt (Assign symbol expr) mem =  (return (), mem')
    where v    = evalExpr expr mem   -- Evaluate the expression
          mem' = store symbol v mem  -- Update the memory with the new value
                                     -- Return the updated memory!
evalStmt (Seq stmt1 stmt2) mem = (io' >> io'', mem'')
    where (io', mem')    = evalStmt stmt1 mem
          (io'', mem'') = evalStmt stmt2 mem'    -- Use updated memory
evalStmt (Print expr) mem = (putStrLn $ show v, mem)
    where v = evalExpr expr mem
evalStmt PrintMem mem = (printMemory mem, mem)

printMemory :: Memory -> IO ()
printMemory []            = return ()
printMemory ((s, v):rest) = do
    putStrLn $ s ++ "␣=␣" ++ show v
    printMemory rest

evaluate :: Stmt -> IO ()
evaluate stmt = io
    where (io, _) = evalStmt stmt []
```

This all works reasonably well. But note for example the kind of work that evalStmt had to do to handle a Seq case: It has to update the memory with the result of the first statement, then make sure to execute the second statement with the updated memory. We often describe that as “weaving in the memory through the steps”.

It would be good if we had a better way to express this. This is where the *State Monad* comes in.

The State Monad

Effectively the state monad is this, if we were allowed to write it:

```
data State a = mem -> (a, mem)
```

So a "state" is a function that takes a memory and returns a pair of the memory as well as some kind of value of a certain parametric type. In fact, since mem is just what the "state" is in our case, we should probably parametrize the "state" by another parametric type, s:

```
data State s a = s -> (a, s)
```

This is the essence of the state monad. However, this is not valid, so we need more something like this:

```
data State s a = ST (s -> (a, s))
```

As we won't need this in that generality, let's stick to the original version with Memory as the state that is maintained. We will call a value of this type a ProgStateT for "program state transformer":

```
data ProgStateT a = PST (Memory -> (a, Memory))
```

So a ProgStateT value is a transformation that takes a memory, possibly transforms it in some way, and also produces a value of type a.

With that in mind, our evalStmt function will gain the signature:

```
evalStmt :: Stmt -> ProgStateT (IO ())
```

And our evaluate becomes:

```
evaluate :: Stmt -> IO ()
evaluate stmt = getResult (evalStmt stmt) []

getResult :: ProgStateT a -> Memory -> a
getResult (PST f) mem = fst $ f mem
```

Now we would like to look at the parts of evalStmt. Here is a direct translation of the previous version:

```
evalStmt :: Stmt -> ProgStateT (IO ())
evalStmt (Assign symbol expr) =
    PST (\mem -> let v    = evalExpr expr mem
                     mem' = store symbol v mem
                 in (return (), mem'))
evalStmt (Seq stmt1 stmt2) =
  PST (\mem -> let PST pst1     = evalStmt stmt1
                   PST pst2     = evalStmt stmt2
                   (io', mem')  = pst1 mem
                   (io'', mem'') = pst2 mem'
               in (io' >> io'', mem''))
evalStmt PrintMem =
  PST (\mem -> (printMemory mem, mem))
evalStmt (Print expr) =
  PST (\mem -> let v = evalExpr expr mem
               in (putStrLn $ show v, mem))
```

We moved the mem parameter to the other side, creating a lambda expression, then wrapped this in PST. Having to wrap and unwrap the PSTs is a bit awkward, but we'll be able to write things nicer later on.

Let's take a look at the PrintMem clause: It effectively simply applies a function to the memory, then returns the resulting value along with the unaltered memory. We can write functions that perform these two steps separately: One function turns our memory into a value, done as a Program State Transformer:

```
getMemory :: ProgStateT Memory
getMemory = PST (\mem -> (mem, mem))
```

The other function takes any kind of function a->b and turns it into a function ProgStateT a -> ProgStateT b, by applying the function to the value without affecting the memory. This function is appropriately called fmap:

```
fmap :: (a -> b) -> ProgStateT a -> ProgStateT b
fmap f (PST pst) =
  PST (\mem -> let (x, mem') = pst mem
               in (f x, mem'))
```

fmap basically says "Do the thing that the function says while staying within the Memory-state-managing context".

Now we can say:

```
evalStmt PrintMem = fmap printMemory getMemory
-- was: evalStmt PrintMem = PST (\mem -> (printMemory mem, mem))
```

This is a lot nicer to read once we get used to the pieces that brought it to life.

Now let's consider the Print case. It's a bit different, as it uses evalExpr:

```
evalStmt (Print expr) =
  PST (\mem -> let v = evalExpr expr mem
               in (putStrLn $ show v, mem))
```

We can still use fmap however (the comments explain the types):

```
evalStmt (Print expr) = fmap (print . evalExpr expr) getMemory
  -- print is putStrLn and show combined
  -- evalExpr expr :: Memory -> Value
  -- print :: Value -> IO ()
  -- print . evalExpr expr :: Memory -> IO ()
```

We tackle Assign next:

```
evalStmt (Assign symbol expr) =
    PST (\mem -> let v    = evalExpr expr mem
                     mem' = store symbol v mem
                 in (return (), mem'))
```

Let's think of the pieces of this function. The first is computing the value, the other is updating the memory. For this we might find the following function helpful. It simply applies the function to the memory to obtain a new memory.

```
updateMemory :: (Memory -> Memory) -> ProgStateT ()
updateMemory f = PST (\mem -> ((), f mem))
```

In our case store symbol v :: Memory -> Memory is such a function. Therefore we can consider updateMemory (store symbol v) as a ProgStateT () value. Our main problem is that we need to compute the v, which depends on the memory as well. So perhaps we should better think of the storing problem as a function:

```
store symbol :: Value -> ProgStateT ()
```

We also have a function that produces a value, given a memory, and hence is a memory transformer that produces a value:

```
evalExpr expr :: ProgStateT Value
```

So our goal is to now combine these two steps:

```
evalExpr expr :: ProgStateT Value
store symbol :: Value -> ProgStateT ()
```

This is a special case of a more generic problem. A function that combines these two by first performing the evalExpr expr, then performing the store symbol with the resulting value. In general we would like a function as follows:

```
(>>=) :: ProgStateT a -> (a -> ProgStateT b) -> ProgStateT b
```

We can define this operation as follows:

```
(>>=) :: ProgStateT a -> (a -> ProgStateT b) -> ProgStateT b
(PST pst1) >>= f = PST (\mem -> let (v, mem') = pst1 mem
                                    PST pst2  = f v
                                in pst2 mem')
```

This looks a bit awkward, but it will look better if we introduce a simply run function:

```
run :: ProgStateT a -> Memory -> (a, Memory)
run (PST pst) = pst
```

Then we can say:

```
(>>=) :: ProgStateT a -> (a -> ProgStateT b) -> ProgStateT b
pst1 >>= f = PST (\mem -> let (v, mem') = run pst1 mem
                          in run (f v) mem')
```

In other words, we run the first action pst1, on the provided memory, then run the second action, f v on the updated memory, mem'.

Using this, function, we can write our Assign part thus:

```
evalStmt (Assign symbol expr) = fmap return (eval' >>= store')
    where eval'  = fmap (evalExpr expr) getMemory
          store' = \v -> updateMemory (store symbol v)
```

We had to add an extra fmap return step, which uses return :: () -> IO () to take us from a ProgStateT () value to a ProgStateT (IO ()) value, because that's the return value expected of evalStmt.

We will do one final optimization: It seems we have often had a need for the following function, so let's give it a name:

```
useMemory :: (Memory -> a) -> ProgStateT a
useMemory f = fmap f getMemory
```

Then we can write the eval' part a bit easier:

```
evalStmt (Assign symbol expr) = fmap return (eval' >>= store')
    where eval'  = useMemory (evalExpr expr)
          store' = \v -> updateMemory (store symbol v)
```

This is perhaps still hard to read, but an important aspect is that the specific memory weaving is all tucked away in the behavior of the (>>=) operator.

Lastly, let's look at the Sequence step:

```
evalStmt (Seq stmt1 stmt2) =
  PST (\mem -> let PST pst1      = evalStmt stmt1
                   PST pst2      = evalStmt stmt2
                   (io', mem')   = pst1 mem
                   (io'', mem'') = pst2 mem'
               in (io' >> io'', mem''))
```

Boy, what a mess! using our run function, we can improve on it a bit:

```
evalStmt (Seq stmt1 stmt2) =
  PST (\mem -> let (io', mem')   = run (evalStmt stmt1) mem
                   (io'', mem'') = run (evalStmt stmt2) mem'
               in (io' >> io'', mem''))
```

This is certainly easier to read! But it still has the weaving of memory a bit too detailed. The main building blocks are the two pieces:

```
evalStmt stmt1 :: ProgStateT (IO ())
evalStmt stmt2 :: ProgStateT (IO ())
```

Let's see if we can use our (>>=) operator, as *it* should effectively do the run bits:

```
evalStmt (Seq stmt1 stmt2) =
    evalStmt stmt1 >>= \io1 -> (evalStmt stmt2
                   >>= \io2 -> yield (io1 >> io2))

yield :: a -> ProgStateT a
yield v = PST (\mem -> (v, mem))
```

And now, here is our final version for evalStmt:

```
evalStmt :: Stmt -> ProgStateT (IO ())
evalStmt (Assign symbol expr) =
    fmap return (eval' >>= store')
    where eval'  = useMemory $ evalExpr expr
          store' = \v -> updateMemory (store symbol v)
evalStmt (Seq stmt1 stmt2) =
    evalStmt stmt1 >>= \io1 -> (evalStmt stmt2
                   >>= \io2 -> yield (io1 >> io2))
evalStmt PrintMem = useMemory printMemory
evalStmt (Print expr) = useMemory (print . evalExpr expr)
```

Let's recap some of the helper functions we used along the way:

```
fmap :: (a -> b) -> ProgStateT a -> ProgStateT b

yield :: a -> ProgStateT a

(>>=) :: ProgStateT a -> (a -> ProgStateT b) -> ProgStateT b
```

In fact these are all common properties of many “container classes”, like `[a]`, `IO a`, as well as `Maybe a`. We will discuss the details more in the next section, before returning to this example.