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.

– let e = Seq (Assign "x" (Add (Numb 2) (Numb 4))) $ Seq (Print $ Var "x") $ PrintMem

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

```
evalStmt :: Stmt -> State Memory (IO ())
```

We will look at the details in a moment. But first, our evaluate becomes:

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

Now let's consider the details of evalStmt. One of the cases we considered was:

```
evalStmt PrintMem mem = (printMemory mem, mem)
```

This would now become:

```
evalStmt PrintMem = ST (\mem -> (printMemory mem, mem))
```

This weird-looking case is one important example of state-handling. In this case we do not need to *change* the state, only to *use* it. We could think of the above as a helper function:

```
gets :: (s -> a) -> ST s a
gets f = ST (\mem -> (f mem, mem))
```

Then we can rewrite that part of evalStmt very simply as:

```
evalStmt PrintMem = gets printMemory
```

TODO: Rest needs rewrite! We can then model the program state via the ST type, using the Memory type as the state:

```
type ProgState a = ST Memory a
```

We can now write functions that turn expressions and statements into ProgState values: They all will carry the Memory with them and update as needed, while the type a will differ: statements will contain IO () to indicate that they interact with the user (printing values when asked). Expressions will need to contain a Value type.

The advantage we get from this approach is that the task of updating and maintaining the state through every step is more or less hidden from our code, implemented in a single place in the (>>=) function.

```
evalStmt :: Stmt  -> ProgState (IO ())
evalStmt (Assign symbol expr) = do
    v <- evalExpr expr   -- Evaluate expr, possibly state update
    modify (store symbol v)
    return $ return ()   -- Second return is the IO
evalStmt (Seq stmt1 stmt2) = do
    io1 <- evalStmt stmt1
    io2 <- evalStmt stmt2
    return $ io1 >> io2
```

```
evalStmt (Print expr) = do
    v <- evalExpr expr
    return $ putStrLn $ show v
evalStmt PrintMem = do
    mem <- getState
    return $ printMemory mem

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

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

The State Monad

The idea of the State type is similar to our view of IO as a function that changed the "world" in some way and also produced a value of type a. The State type makes that more precise. It can work with very generic "states", represented here with the type s. We can then make the following definition:

```
newtype ST s a = S (s -> (a, s))
```

So a value of type ST s a is a *transition* function that takes the current state, and produces a value of type a along with a new (updated) state. For technical reasons we place that function inside an S tag. We can easily write a function that removes the tag:

```
runState :: ST s a -> s -> (a, s)
runState (S trans) x = trans x
-- Could also have done: runState (S trans) = trans
```

We can also write a function that evaluates a given stateful computation for a state, discards the resulting state and simply returns the final value:

```
evalState :: ST s a -> s -> a
evalState (S trans) st = x'
    where (x', _) = runState (S trans) st
-- Alternative definition: evalState state = fst . runState state
```

In order to meaningfully work with this new state structure though, we will need a couple of things:

- A way to set the state to a new value. This is akin to putStr printing something to the screen and hence changing the state of IO.
- A way to read the state, while going through a stateful computation.

- A way to turn a normal value into a stateful computation. We did this for IO with a function called return, and we will do the same here.
- A way to apply a normal function to the value in the computation, while keeping the state the same.
- A way to chain two computations together, so that the state transfers from the first to the second. In Haskell that operation has a name, (>>=), typically called a "bind" operation.

Let us take a look at each of these. Getting and setting the state is easy:

```
getState :: ST s s
getState = S (\st -> (st, st))

putState :: s -> ST s ()
putState st = S (\_ -> ((), st))
```

Now we need functions for returning a normal value as the result of a stateful computation that does not change the state, and also mapping the result values through a normal function:

```
return :: a -> ST s a
return x = S (\st -> (x, st))

fmap :: (a -> b) -> ST s a -> ST s b
fmap f (S trans) = S trans'
    where trans' st = (f x, st')
                      where (x, st') = trans st
```

Now comes the tricky bit: We want to combine two stateful computations into a new stateful computation. Actually what we will do is slightly different: We will combine one stateful computation with a function that takes as input a result of the first computation, and uses it to produce a second stateful computation that is then carried out. It is probably the hardest part of the whole story:

```
(>>=) :: ST s a -> (a -> ST s b) -> ST s b
act1 >>= f = S trans
    where trans st = let (x, st') = runState act1 st
                         act2     = f x
                     in runState act2 st'
```

This all looks a bit messy, but we only had to do it and understand it once! Now we can use this chaining instead of having to effectively manually do it every time. The even cooler thing is that this effectively allows us to use the IO-type notation with do, and Haskell will unravel that for us. For example, we can build a function that modifies the current state, as follows:

```
modify :: (s -> s) -> ST s ()
modify f = do
    st <- getState
    putState (f st)
```

And Haskell turns that into:

```
modify f = getState >>= (\st -> putState (f st))
-- Can also write as:   getState >>= (putState . f)
```

which is perhaps elegant but somewhat harder to read, especially if it involved more steps. In order for Haskell to be able to carry this out, it must know that our ST structure is a "monad". We will discuss what that means later in the chapter, but effectively it just means having the "bind" operation we just defined.

Practice:

1. Using "do" notation, write a function `incr :: ST Int ()` that increments the integer state by 1. Also do it by instead using `modify`.
2. Using "do" notation, write a function `account :: a -> ST Int a` which "accounts" for the computation that produces a value of type `a`, by incrementing the integer state. Your function should simply increment the state and return the provided value.