Functors, Applicatives, and Monads

In this section we discuss three important type classes that pertain to container types. They each express a key idea:

- Functors express the idea that we can `map` a function over a container type, by applying the function to its contents.
- Applicatives express the idea of mapping a function over possibly multiple arguments.
- Monads express the idea of sequencing effectful operations.

Reading

- Sections 12.1-12.3
- Practice exercises (12.5): 1, 4, 7
- Optional practice: 2, 3, 8
- https://wiki.haskell.org/State_Monad

Container Types and the State "Monad"

Before we start exploring functors, let us consider three important container types that share certain features. All three types express the idea of "producing a value of a given type", but they do so in different ways:

The Maybe type expresses the idea that the operation we performed may or may not have produced a value (i.e. it may have failed).

[a]Maybe a The list type expresses the idea that the operation we performed may have returned more than one possible values (but all of the same type). A good example is the list comprehensions we discussed earlier in the course. A list is a convenient way to represent these possibly multiple results.

The `IO` type expresses the idea that our operation interacts with the world in some way, and produces a value of type `a`.

The State "monad" (we will discuss what that means later) is another example. It is meant to somehow maintain and update some "state" `s`, and also return a value of type `a`. We will discuss this type now in some more detail.

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:

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

So a value of type ST 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:

```haskell
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:

```haskell
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:

```haskell
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.

A State Example: Statement Interpretation

As an illustration of the state type described above, let us imagine a small programming language. It has expressions that perform basic arithmetic, 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 Stmts 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
```

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 Double 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.

```
evalExpr :: Expr  -> ProgState Double
evalExpr (Numb x) = return x
evalExpr (Var s) = do
    mem <- getState
    case lookup s mem of
        Nothing -> error ("Cannot␣find␣symbol:␣" ++ s)
        Just v  -> return v
evalExpr (Add e1 e2) = do
    v1 <- evalExpr e1   -- Evaluate e1, possibly state update
    v2 <- evalExpr e2   -- Evaluate e2, possibly state update
    return $ v1 + v2
evalExpr (Prod e1 e2) = do
    v1 <- evalExpr e1   -- Evaluate e1, possibly state update
    v2 <- evalExpr e2   -- Evaluate e2, possibly state update
    return $ v1 * v2

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
```

Functors

We will now begin our investigation of key operations that all the aforementioned “container types” have in common. The first of these is the idea of a functor.

The container/polymorphic type `f a` is a **functor** if it comes equipped with a function:

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

In other words, functors come equipped with a natural way to transform the contained values, is provided with an appropriate function. There are certain “naturality” conditions that such a function must obey, often described via a set of "properties":

```
fmap id      = id
fmap (g . h) = fmap g . fmap h
```

These look complicated, but basically they say that functors behave in a reasonable way:

- If the function we use on the functor is the identity function `id :: a -> a` (defined by `id a = a`, then the resulting transformation `fmap id :: f a -> f a` is just the identity function `id :: f a -> f a`.
- If we have functions `h :: a -> b` and `g :: b -> c`, then we have two different ways to produce a function `f a -> f c`:
  - The first is to first compute the composition `g . h :: a -> c` and feed that into `fmap`.
  - The other is to compute `fmap h :: f a -> f b` and `fmap g :: f b -> f c`, then to compose them.

  And the result is the same whichever way we do it. It does not matter if you compose first, or if you apply `fmap` first.

These two properties allow Haskell to simplify some functor operations. But we will not concern ourselves with them very much right now.

The key insight is that very many container types are naturally functor types. This is captured via the `Functor` class:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

We can now see how each of the types we have defined earlier become instances of `Functor`, by transforming their contained value:

```
instance Functor [] where
    fmap f xs = [f x | x <- xs]    -- could also have written fmap = map

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)

instance Functor IO where
    fmap f action = do
        x <- action
        return $ f x

instance Functor (ST s) where
    fmap f (S trans) = S trans'
        where trans' st = let (x, st') = trans st
                          in (f x, st')
```

The `Functor` provides us for free with a number of standard functions[1]:

[1] http://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Functor.html#t:Functor

```
void   :: Functor f => f a -> f ()
(<$)   :: Functor f => a -> f b -> f a
($>)   :: Functor f => f a -> b -> f b
(<$>)  :: Functor f => (a -> b) -> f a -> f b
```

The last function is simply a synomym for fmap. Note that it is very similar to ($) :: (a -> b) -> a -> b. So whereas to apply a function to a value we would do f $ x, if it is a container value we would use <$>:

```
(+ 2) <$> Just 3       -- Results in Just 5
(+ 1) <$> [2, 3, 4]  -- Results in Just [3, 4, 5]
```

Applicatives

The Applicative class is an extension of the Functor class. It essentially allows us to map functions as Functor does, but works with functions of possibly more than one parameter. We can think of it as providing for us a "tower" of functions:

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

It does so by requiring two functions:

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

If we compare <*> to the Functor class' fmap, the main difference is that in the Applicative case we allow the function itself to be in the container type.

Before we see some examples, let us see what these type signatures have to do with the functions described above.

```
fmap0 g       = pure g
fmap1 g x     = pure g <*> x
fmap2 g x y   = pure g <*> x <*> y
fmap3 g x y z = pure g <*> x <*> y <*> z
```

Let us take a closer look at the types for fmap2:

```
g :: a -> b -> c
x :: f a
y :: f b
pure g :: f (a -> (b -> c))
(<*>) :: f (a -> (b -> c)) -> f a -> f (b -> c)
pure g <*> x :: f (b -> c)
(<*>) :: f (b -> c) -> f b -> f c
(pure g <*> x) <*> y :: f c
```

Now let us discuss how our earlier types are instances of Applicative. For the Maybe type, if we have a function and we have a value, then we apply the function to the value. Otherwise we return Nothing. For lists, we have a list of functions and a list of values, and we apply all functions to all values.

```
instance Applicative Maybe where
    pure x = Just x
    Just f  <*> Just x  = Just $ f x
    Nothing <*> _       = Nothing
    _       <*> Nothing = Nothing

instance Applicative [] where
    pure x = [x]
    gs <*> xs = [g x | g <- gs, x <- xs]

instance Applicative IO where
    pure x = return x
    actionf <*> actionx = do
        f <- actionf
        x <- actionx
        return $ f x

instance Applicative (ST s) where
   pure x = S (\st -> (x, st))
   actf <*> act1 = S trans
      where trans st = let (f, st')  = runState actf st
                           (x, st'') = runState act1 st'
                       in (f x, st'')
```

As a fun example, try this:

```
[(+), (*), (-)] <*> [1, 2, 3] <*> [4, 5]
```

There are rules for the Applicative class similar to the rules for Functor. we list them here for completeness, but will not discuss them further.

```
pure id <*> x      = x
pure g <*> pure x = pure (g x)
h <*> pure y      = pure (\g -> g y) <*> h
x <*> (y <*> z)   = (pure (.) <*> x <*> y) <*> z
```

Monads

The Monad class is the last step in this hierarchy of classes.

- Functor allowed us to apply "pure" functions (a->b) to our effectful values f a.
- Applicative allowed us to apply effectful function values f (a -> b) to our effectful values f a.
- Monad will allow us to combine functions that result in effectful values a -> f b with effectful values f a to produce a result f b.

Here is the definition of the Monad class:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

    return = pure  -- default implementation via Applicative
```

The most common way to use monads is via the "do" notation. For example, when we wrote earlier something like

```
evalStmt PrintMem = do
    mem <- getState
    return $ printMemory mem
```

Haskell would convert that program code into:

```
evalStmt PrintMem = getState >>= (\mem -> return $ printMemory mem)
```

In other words, each subsequent statement in a "do" block is behind the scenes a function of the variables stored during previous statements. You will agree surely that the "do" notation simplifies things quite a lot.

Let us see how the various classes we have seen earlier can be thought of as instances of Monad:

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x

instance Monad [] where
    return x = [x]
    xs >= f  = [y | x <- xs, y <- f x]

instance Monad ST where
    return x = S (\st -> (x, st))
    act1 >>= f = S trans
    where trans st = let (x, st') = runState act1 st
                         act2     = f x
                     in runState act2 st'
```

The definition for lists is particularly interesting: If we have a list of elements, and for each element we can produce a list of result elements via the function f, we can then iterate over each resulting list and concatenate all the results together. This is close to what list comprehensions are doing.

Similar to the other two classes, instances of Monad are expected to further satisfy certain rules, that we will not explore in detail:

```
return x >>= f     = f x
mx >>= return      = mx
(mx >>= f) >>= g   = mx >>= (\x -> (f x >> g))
```

Monadic operations

Having a monad structure provides a number of functions for us, for a host of common operations. Most of these functions can be found in the Control.Monad[2] module. Most use the Traversable class, which in turn extends the Foldable class. The former provides a traverse function and the latter foldr and foldMap functions. For simplicity

[2] http://hackage.haskell.org/package/base-4.10.0.0/docs/Control-Monad.html#t:Monad

you can assume that t a in these is just [a], and to illustrate this we write both type declarations:

```
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
sequence :: Monad m => [m a] -> m [a]
sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()
sequence_ :: Monad m => [m a] -> m ()
join :: Monad m => m (m a) -> m a
```