Type classes over parametrized types: Foldables, Functors, Applicatives, Monads

In this section we discuss a number of important type classes that pertain to container types (that's just another name for a *parametrized type*: A parametrized type contains some kinds of values). Each of these type classes expresses a key idea:

- Foldables express the idea that we can fold over the container's values, conceptually just like foldr works for lists.
- 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.

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:

`Maybe a` 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]` 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.

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

`ST s a` The State "monad" is another example. It is meant to somehow maintain and update some "state" `s`, and also return a value of type `a`. We already saw this as the `ProgStateT a` type, which is basically `ST s a` with `s` being `Memory`. We will come back to it in the next lecture.

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, if provided with an appropriate function.

Practice: Without looking at the remaining notes, implement fmap for Maybe, IO and State:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: (a -> b) -> IO a -> IO b
fmap :: (a -> b) -> State s a -> State s b
```

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`, in other words mapping over the identy function does not change anything.
- 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 very much with this aspect 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 :: (a -> b) -> [a] -> [b]
    fmap f xs = [f x | x <- xs]   -- could also have written fmap = map
```

```
instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)

instance Functor IO where
    -- fmap :: (a -> b) -> IO a -> IO b
    fmap f action = do
        x <- action
        return $ f x

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

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

```
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]
```

The middle two functions effectively preserve the "container form", but using the provided value instead of the values in the container. Some examples:

```
"hi" <$ Just 3      -- Results in Just "hi"
"hi" <$ Nothing     -- Results in Nothing
4 <$ [1,3,4]        -- Results in [4, 4, 4]
5 <$ putStrLn "hi" -- A IO Integer action which prints "hi" and returns 5
```

Practice: Implement (<$) and void using fmap.

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:

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

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

Something very profound is going on here, so make sure you follow the above type considerations. The idea is that we combine the ability to only partially apply a function with the ability that <*> provides us, to apply a function within the applicative to a value within the applicative.

Practice: Think of possible definitions for <*> in some of the types we have seen:

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
(<*>) :: [a -> b] -> [a] -> [b]   -- List of functions and list of a-values
(<*>) :: IO (a -> b) -> IO a -> IO b
```

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. The difference with applicative is that *the second effect may depend on the output value from the first effect.*

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
```

We've already seen the (>>=) operator in a few places. In effect, the definition of (>>=) for the IO monad is:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
ioa >>= f = do
  x <- ioa
  f x
```

So for IO it simply says: Carry out the IO a action, then use its value to obtain an IO b action. That's your result (i.e. do that next). We could almost define the operation (>>=) this way, except for the fact that it is actually used behind the scenes to implement the do notation:

The do notation is behind the scenes nothing more than successive applications of (>>=) (and (>>)).

For example, when we write something like:

```
echo = do
  c <- getChar
  putChar c
```

What we are really doing is:

```
echo = getChar >>= putChar
-- getChar :: IO Char
-- putChar :: Char -> IO ()
-- (>>=) :: IO a -> (a -> IO ()) -> IO ()
```

In other words, each subsequent statement in a "do" block is behind the scenes a function of the variables stored during previous statements, chained via >>=. You will agree surely that the "do" notation simplifies things quite a lot. We will rewrite our expression-evaluating program using this notation in the next set of notes. For now, here's another example of this, using the fact that lists are a Monad:

```
addAll lst = do
    x <- lst
    y <- lst
    return $ x + y

-- Equivalent to: [x + y | x <- lst, y <- lst]

addAll [1,2,3]    --    Produces [2,3,4,3,4,5,4,5,6]
```

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]
    -- Here f :: a -> [b]
    xs >= f  = [y | x <- xs, y <- f x]

instance Monad ProgStateT where
    return x = PST (\mem -> (x, mem))  -- Called yield before
    -- (>>=) :: ProgStateT a -> (a -> ProgStateT b) -> ProgStateT b
    pst1 >>= f = PST (\mem -> let (v, mem') = run pst1 mem
                              in run (f v) mem')
```

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.

As another example, here's how we could write the list concat function using monads:

```
concat :: [[a]] -> [a]
concat lsts = lsts >>= (\lst -> lst)
-- Or simpler, using the identity function `id :: a -> a`
concat lsts = lsts >>= id
-- Using operator section syntax, now it's getting hard to understand
concat = (>>= id)
```

Make sure you understand the above, work out the types of each piece!

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 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
```

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