Haskell/Monad transformers

< Haskell

We have seen how monads can help handling `IO` actions, `Maybe`, lists, and state. With monads providing a common way to use such useful general-purpose tools, a natural thing we might want to do is using the capabilities of *several* monads at once. For instance, a function could use both I/O and `Maybe` exception handling. While a type like `IO (Maybe a)` would work just fine, it would force us to do pattern matching within `IO` do-blocks to extract values, something that the `Maybe` monad was meant to spare us from.

Enter **monad transformers**: special types that allow us to roll two monads into a single one that shares the behavior of both.

Passphrase validation

Consider a real-life problem for IT staff worldwide: getting users to create strong passphrases. One approach: force the user to enter a minimum length with various irritating requirements (such as at least one capital letter, one number, one non-alphanumeric character, etc.)

Here's a Haskell function to acquire a passphrase from a user:

```
getPassphrase :: IO (Maybe String)
getPassphrase = do s <- getLine
                   if isValid s then return $ Just s
                                else return Nothing

-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8
            && any isAlpha s
            && any isNumber s
            && any isPunctuation s
```

First and foremost, `getPassphrase` is an `IO` action, as it needs to get input from the user. We also use `Maybe`, as we intend to return `Nothing` in case the password does not pass the `isValid`. Note, however, that we aren't actually using `Maybe` as a monad here: the `do` block is in the `IO` monad, and we just happen to `return` a `Maybe` value inside it.

Monad transformers not only make it easier to write `getPassphrase` but also simplify all the code instances. Our passphrase acquisition program could continue like this:

```
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrase
                   case maybe_value of
                       Just value -> do putStrLn "Storing in
database..."  -- do stuff
                       Nothing -> putStrLn "Passphrase invalid."
```

The code uses one line to generate the `maybe_value` variable followed by further validation of the passphrase.

With monad transformers, we will be able to extract the passphrase in one go — without any pattern matching (or equivalent bureaucracy like `isJust`). The gains for our simple example might seem small but will scale up for more complex situations.

A simple monad transformer: `MaybeT`

To simplify `getPassphrase` and the code that uses it, we will define a *monad transformer* that gives the `IO` monad some characteristics of the `Maybe` monad; we will call it `MaybeT`. That follows a convention where monad transformers have a "`T`" appended to the name of the monad whose characteristics they provide.

`MaybeT` is a wrapper around `m (Maybe a)`, where `m` can be any monad (`IO` in our example):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This data type definition specifies a `MaybeT` type constructor, parameterized over `m`, with a data constructor, also called `MaybeT`, and a convenient accessor function `runMaybeT`, with which we can access the underlying representation.

The whole point of monad transformers is that *they transform monads into monads*; and so we need to make `MaybeT m` an instance of the `Monad` class:

```
instance Monad m => Monad (MaybeT m) where
  return  = MaybeT . return . Just
```

```
-- The signature of (>>=), specialized to MaybeT m:
-- (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                      case maybe_value of
                         Nothing    -> return Nothing
                         Just value -> runMaybeT $ f value
```

It would also have been possible (though arguably less readable) to write the return function as: `return  = MaybeT . return . return`.

Starting from the first line of the `do` block:

- First, the `runMaybeT` accessor unwraps `x` into an `m (Maybe a)` computation. That shows us that the whole `do` block is in `m`.
- Still in the first line, `<-` extracts a `Maybe a` value from the unwrapped computation.
- The `case` statement tests `maybe_value`:
  - With `Nothing`, we return `Nothing` into `m`;
  - With `Just`, we apply `f` to the `value` from the `Just`. Since `f` has `MaybeT m b` as result type, we need an extra `runMaybeT` to put the result back into the `m` monad.
- Finally, the `do` block as a whole has `m (Maybe b)` type; so it is wrapped with the `MaybeT` constructor.

It may look a bit complicated, but aside from the copious amounts of wrapping and unwrapping, the implementation of `MaybeT` 's bind works the same way as the implementation of `Maybe` 's familiar bind operator:

```
-- (>>=) for the Maybe monad
maybe_value >>= f = case maybe_value of
                        Nothing -> Nothing
                        Just value -> f value
```

Why use the `MaybeT` constructor before the `do` block while we have the accessor `runMaybeT` within `do`? Well, the `do` block must be in the `m` monad, not in `MaybeT m` (which lacks a defined bind operator at this point).

As usual, we also have to provide instances for the superclasses of `Monad`, `Applicative` and `Functor`:

```
instance Monad m => Applicative (MaybeT m) where
    pure = return
    (<*>) = ap

instance Monad m => Functor (MaybeT m) where
    fmap = liftM
```

In addition, it is convenient to make `MaybeT m` an instance of a few other classes:

```
instance Monad m => Alternative (MaybeT m) where
    empty   = MaybeT $ return Nothing
    x <|> y = MaybeT $ do maybe_value <- runMaybeT x
                          case maybe_value of
                               Nothing    -> runMaybeT y
                               Just _     -> return maybe_value

instance Monad m => MonadPlus (MaybeT m) where
    mzero = empty
    mplus = (<|>)

instance MonadTrans MaybeT where
    lift = MaybeT . (liftM Just)
```

`MonadTrans` implements the `lift` function, so we can take functions from the `m` monad and bring them into the `MaybeT m` monad in order to use them in `do` blocks. As for `Alternative` and `MonadPlus`, since `Maybe` is an instance of those classes it makes sense to make the `MaybeT m` an instance too.

Passphrase validation, simplified

The above passphrase validation example can now be simplified using the `MaybeT` monad transformer as follows:

```
getPassphrase :: MaybeT IO String
getPassphrase = do s <- lift getLine
                   guard (isValid s) -- Alternative provides
```

```
guard.
                   return s

askPassphrase :: MaybeT IO ()
askPassphrase = do lift $ putStrLn "Insert your new passphrase:"
                   value <- getPassphrase
                   lift $ putStrLn "Storing in database..."
```

The code is now simpler, especially in the user function `askPassphrase`. Most importantly, we do not have to manually check whether the result is `Nothing` or `Just`: the bind operator takes care of that for us.

Note how we use `lift` to bring the functions `getLine` and `putStrLn` into the `MaybeT IO` monad. Also, since `MaybeT IO` is an instance of `Alternative`, checking for passphrase validity can be taken care of by a `guard` statement, which will return `empty` (i.e. `IO Nothing`) in case of a bad passphrase.

Incidentally, with the help of `MonadPlus` it also becomes very easy to ask the user *ad infinitum* for a valid passphrase:

```
askPassphrase :: MaybeT IO ()
askPassphrase = do lift $ putStrLn "Insert your new passphrase:"
                 value <- msum $ repeat getPassphrase
                 lift $ putStrLn "Storing in database..."
```

Running your new version of askPassphrase on ghci is easy:

```
runMaybeT askPassphrase
```

A plethora of transformers

The transformers package provides modules with transformers for many common monads (`MaybeT`, for instance, can be found in Control.Monad.Trans.Maybe (http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-Trans-Maybe.html)). These are defined consistently with their non-transformer versions; that is, the implementation is basically the same except with the extra wrapping and unwrapping needed to thread the other monad. From this point on, we will use **precursor monad** to refer to the non-transformer monad (e.g. Maybe in MaybeT) on which a transformer is based and **base monad** to refer to the other monad (e.g. IO in MaybeT IO) on which the transformer is applied.

To pick an arbitrary example, `ReaderT Env IO String` is a computation which involves reading values from some environment of type `Env` (the semantics of `Reader`, the precursor monad) and performing some `IO` in order to give a value of type `String`. Since the bind operator and `return` for the transformer mirror the semantics of the precursor monad, a `do` block of type `ReaderT Env IO String` will, from the outside, look a lot like a `do` block of the `Reader` monad, except that `IO` actions become trivial to embed by using `lift`.

Type juggling

We have seen that the type constructor for `MaybeT` is a wrapper for a `Maybe` value in the base monad. So, the corresponding accessor `runMaybeT` gives us a value of type `m (Maybe a)` - i.e. a value of the precursor monad returned in the base monad. Similarly, for the `ListT` and `ExceptT` transformers, which are built around lists and `Either` respectively:

```
runListT :: ListT m a -> m [a]
```

and

```
runExceptT :: ExceptT e m a -> m (Either e a)
```

Not all transformers are related to their precursor monads in this way, however. Unlike the precursor monads in the two examples above, the `Writer`, `Reader`, `State`, and `Cont` monads have neither multiple constructors nor constructors with multiple arguments. For that reason, they have run... functions which act as simple unwrappers, analogous to the run...T of the transformer versions. The table below shows the result types of the run... and run...T functions in each case, which may be thought of as the types wrapped by the base and transformed monads respectively.[1]

Precursor	Transformer	Original Type ("wrapped" by precursor)	Combined Type ("wrapped" by transformer)
Writer	WriterT	`(a, w)`	`m (a, w)`
Reader	ReaderT	`r -> a`	`r -> m a`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`
Cont	ContT	`(a -> r) -> r`	`(a -> m r) -> m r`

Notice that the precursor monad type constructor is absent in the combined types. Without interesting data constructors (of the sort that `Maybe` and lists have), there is no reason to retain the precursor monad type after unwrapping the transformed monad. It is also worth noting that in the latter three cases we have function types being wrapped. `StateT`, for instance, turns state-transforming functions of the form `s -> (a, s)` into state-transforming functions of the form `s -> m (a, s)`; only the result type of the wrapped function goes into the base monad. `ReaderT` is analogous. `ContT` is different because of the semantics of `Cont` (the *continuation* monad): the result types of both the wrapped function and its function argument must be the same, and so the transformer puts both into the base monad. In general, there is no magic formula to create a transformer version of a monad; the form of each transformer depends on what makes sense in the context of its non-transformer type.

Lifting

We will now have a more detailed look at the `lift` function, which is critical in day-to-day use of monad transformers. The first thing to clarify is the name "lift". One function with a similar name that we already know is `liftM`. As we have seen in Understanding monads, it is a monad-specific version of `fmap`:

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

`liftM` applies a function `(a -> b)` to a value within a monad `m`. We can also look at it as a function of just one argument:

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

`liftM` converts a plain function into one that acts within `m`. By "lifting", we refer to bringing something into something else — in this case, a function into a monad.

`liftM` allows us to apply a plain function to a monadic value without needing do-blocks or other such tricks:

bind notation	do notation	liftM
`monadicValue >>= \x -> return (f x)`	`do x <- monadicValue` `return (f x)`	`liftM f monadicValue`

The `lift` function plays an analogous role when working with monad transformers. It brings (or, to use another common word for that, *promotes*) base monad computations to the combined monad. By doing so, it allows us to easily insert base monad computations as part of a larger computation in the combined monad.

`lift` is the single method of the `MonadTrans` class, found in Control.Monad.Trans.Class (http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-Trans-Class.html) . All monad transformers are instances of `MonadTrans`, and so `lift` is available for them all.

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

There is a variant of `lift` specific to `IO` operations, called `liftIO`, which is the single method of the `MonadIO` class in Control.Monad.IO.Class (http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad-IO-Class.html) .

```
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a
```

`liftIO` can be convenient when multiple transformers are stacked into a single combined monad. In such cases, `IO` is always the innermost monad, and so we typically need more than one lift to bring `IO` values to the top of the stack. `liftIO` is defined for the instances in a way that allows us to bring an `IO` value from any depth while writing the function a single time.

Implementing `lift`

Implementing `lift` is usually pretty straightforward. Consider the `MaybeT` transformer:

```
instance MonadTrans MaybeT where
    lift m = MaybeT (liftM Just m)
```

We begin with a monadic value of the base monad. With `liftM` (`fmap` would have worked just as fine), we slip the precursor monad (through the `Just` constructor) underneath, so that we go from `m a` to `m (Maybe a)`). Finally, we wrap things up with the `MaybeT` constructor. Note that the `liftM` here works in the base monad, just like the do-block wrapped by `MaybeT` in the implementation of `(>>=)` we saw early on was in the base monad.

Exercises

1. Why is it that the `lift` function has to be defined separately for each monad, where as `liftM` can be defined in a universal way?
2. `Identity` is a trivial functor, defined in `Data.Functor.Identity` as:

   ```
   newtype Identity a = Identity { runIdentity :: a }
   ```

   It has the following `Monad` instance:

   ```
   instance Monad Identity where
       return a = Identity a
       m >>= k  = k (runIdentity m)
   ```

   Implement a monad transformer `IdentityT`, analogous to `Identity` but wrapping values of type `m a` rather than `a`. Write at least its `Monad` and `MonadTrans` instances.

Implementing transformers

The State transformer

As an additional example, we will now have a detailed look at the implementation of `StateT`. You might want to review the section on the State monad before continuing.

Just as the State monad might have been built upon the definition `newtype State s a = State { runState :: (s -> (a,s)) }`, the StateT transformer is built upon the definition:

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

`StateT s m` will have the following `Monad` instance, here shown alongside the one for the precursor state monad:

State	StateT

<pre>newtype State s a =
  State { runState :: (s -> (a,s)) }
 	
instance Monad (State s) where
  return a   = State $ \s -> (a,s)
  (State x) >>= f = State $ \s ->
    let (v,s') = x s
    in runState (f v) s'</pre>
	<pre>newtype StateT s m a =
	  StateT { runStateT :: (s -> m (a,s)) }
	
	instance (Monad m) => Monad (StateT s m) where
	  return a = StateT $ \s -> return (a,s)
	  (StateT x) >>= f = StateT $ \s -> do
	    (v,s') <- x s -- get new value and state
	    runStateT (f v) s' -- pass them to f</pre>

Our definition of `return` makes use of the `return` function of the base monad. `(>>=)` uses a do-block to perform a computation in the base monad.

> *Note*
>
> Incidentally, we can now finally explain why, back in the chapter about `State`, there was a `state` function instead of a `State` constructor. In the transformers and mtl packages, `State s` is implemented as a type synonym for `StateT s Identity`, with `Identity` being the dummy monad introduced in an exercise of the previous section. The resulting monad is equivalent to the one defined using `newtype` that we have used up to now.

If the combined monads `StateT s m` are to be used as state monads, we will certainly want the all-important `get` and `put` operations. Here, we will show definitions in the style of the mtl package. In addition to the monad transformers themselves, mtl provides type classes for the essential operations of common monads. For instance, the `MonadState` class, found in Control.Monad.State (http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-State.html) , has `get` and `put` as methods:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

> *Note*
>
> `instance (Monad m) => MonadState s (StateT s m)` should be read as: "For any type `s` and any instance of `Monad` `m`, `s` and `StateT s m` together form an instance of `MonadState`". `s` and `m` correspond to the state and the base monad, respectively. `s` is an independent part of the instance specification so that the methods can refer to it – for instance, the type of `put` is `s -> StateT s m ()`.

There are `MonadState` instances for state monads wrapped by other transformers, such as `MonadState s m => MonadState s (MaybeT m)`. They bring us extra convenience by making it unnecessary to lift uses of `get` and `put` explicitly, as the `MonadState` instance for the combined monads handles the lifting for us.

It can also be useful to lift instances that might be available for the base monad to the combined monad. For instance, all combined monads in which `StateT` is used with an instance of `MonadPlus` can be made instances of `MonadPlus`:

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero = StateT $ \_ -> mzero
  (StateT x1) `mplus` (StateT x2) = StateT $ \s -> (x1 s) `mplus`
(x2 s)
```

The implementations of `mzero` and `mplus` do the obvious thing; that is, delegating the actual work to the instance of the base monad.

Lest we forget, the monad transformer must have a `MonadTrans`, so that we can use `lift`:

```
instance MonadTrans (StateT s) where
  lift c = StateT $ \s -> c >>= (\x -> return (x,s))
```

The `lift` function creates a `StateT` state transformation function that binds the computation in the base monad to a function that packages the result with the input state. If, for instance, we apply StateT to the List monad, a function that returns a list (i.e., a computation in the List monad) can be lifted into `StateT s []` where it becomes a function that returns a `StateT (s -> [(a,s)])`. I.e. the lifted computation produces *multiple* (value,state) pairs from its input state. This "forks" the computation in StateT, creating a different branch of the computation for each value in the list returned by the lifted function. Of course, applying `StateT` to a different monad will produce different semantics for the `lift` function.

Exercises

1. Implement `state :: MonadState s m => (s -> (a, s)) -> m a` in terms of `get` and `put`.
2. Are `MaybeT (State s)` and `StateT s Maybe` equivalent? (Hint: one approach is comparing what the `run...T` unwrappers produce in each case.)

Acknowledgements

This module uses a number of excerpts from All About Monads (http://www.haskell.org/haskellwiki/All_About_Monads) , with permission from its author Jeff Newbern.

Notes

1. The wrapping interpretation is only literally true for versions of the mtl package older than 2.0.0.0 .