Monad Transformers

- Advanced application of monads

A Concurrency Monad

```haskell
data Thread = Action (IO ()) Thread
            | Fork   Thread  Thread
            | End
```

```haskell
newtype CM a = CM {
  continueWith :: (a -> Thread) -> Thread
}
```

```haskell
instance Functor CM where
  fmap = liftM

instance Applicative CM where
  pure a = CM $ \k ->
    k a
```

```haskell
instance Monad CM where
  m >>= f = CM $ \k ->
    m `continueWith` \x ->
                       f x `continueWith` k
```

With operations

```haskell
print :: Char -> CM ()
print c = CM $ \k ->
  Action (putChar c) (k ())
```

-

```haskell

fork :: CM a -> CM ()
fork m = CM $ \k ->
  Fork (thread m) (k ())

end :: CM a
end = CM $ \_ -> End
```

You can run a computation with an interpretation function.

```haskell
data Actions = [ IO () ]

runContinuation :: CM a -> Actions
runContinuation m = schedule [thread m]
```

A simple round-robin scheduler

```haskell
schedule :: [ Thread ] -> Actions
schedule []                     = []
schedule ( Action c t : ts ) = c : (schedule $ ts ++ [t])
schedule ( Fork t0 t1 : ts ) = schedule $ ts ++ [t0,t1]
schedule ( End : ts)            = schedule ts
```

```haskell
p1 :: CM ()
p1 = do print 'a'; print 'b'; ... ; print 'j'

p2 :: CM ()
p2 = do print '1'; print '2'; ... ; print '0'

p3 :: CM ()
p3 = do fork p1; print 'A';fFork p2 ; print 'B'
```

```
aAbc1Bd2e3f4g5h6i7j890
```

```haskell
loop :: Char -> CM ()
loop c = do
  print c ; loop c

p4 = do fork (loop 'a') ; loop 'b'
```

- This is fine, but would like more IO communication, inter-thread communication, integrated logging, error handling for thread failures, more primitives
- Embedded monads
    - Embed the IO monad - done
    - Embed a state monad to hold communication variables and locks

- Embed a writer monad to track logs or gather actions (or embed two, one for each)
- Embed an error monad (like Maybe)

- **Monad Transformers**
    - Monad composition isn't natural
    - Imagine a calculator - state monad could be used to manage the memory
        - do x <- get ; y <- divide 100 x ; put y
        - But division by zero…?
        - Add an Exception monad! Give it some function `handleError`
        - do x <- get ; y <- (divide 100 x) `handleError` (return 0)
    - **This seems useful - maybe a new monad to build**
        - **newtype SM s a = SM (s-> Maybe (a , s))**
        - Why not? **newtype SM s a = SM (s -> (Maybe a, s))**
        - Which is right? Both? Neither?
        - This will introduce a lot of extra leg work -> **Transformers**
    - **Key ideas:**
        - Define features, not monads
        - Define them as functions from monads to monads
        - **Monad combination as a way of mixing effects from different monads**
    - Since there will be more than one concrete implementation we will make a class for the desired effects

      ```
      class Monad m => Err m where
          eFail :: m a
          eHandle :: m a -> m a -> m a
      ```

        - This states that: for a monad m we can transform it into a monad which does everything that m can do, but can also error handle with eFail and eHandle
        - **To be useful we need a way to access the operations of monad m**

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

            - lift's sole purpose is to access the operations of the transformed monad
        - This requires at least one concrete instance

          ```
          newtype ErrTM m a = ErrTM (m (Maybe a))
          ```

          Which has to be a monad:

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

          instance Monad m => Applicative (ErrTM m) where
            pure a = ErrTM (return (Just a))
          ```

          ```
          instance Monad m => Monad (ErrTM m) where
             (ErrTM m) >>= f = ErrTM $ m >>= r
                 where unwrapErrTM (ErrTM v) = v
                       r (Just x) = unwrapErrTM $ f x
                       r Nothing = return Nothing
          ```

```haskell
instance Monad m => MonadTransformer ErrTM m where
  lift m = ErrTM $ do
    a <- m
    return (Just a)
```

Finally, we need to provide the actions of the error monad:

```haskell
instance Monad m => Err (ErrTM m) where
   eFail = ErrTM (return Nothing)

   eHandle (ErrTM m1)(ErrTM m2) = ErrTM $ do
      ma <- m1
      case ma of
        Nothing -> m2
        Just _ -> return ma
```

```haskell
runErrTM :: Monad m => ErrTM m a -> m a
runErrTM (ErrTM etm) = do
  ma <- etm
  case ma of
    Just a -> return a
```

- We can now divide in our monad

```haskell
divide _ 0 = eFail
divide x y = return (x `div` y)
```

The type here is:

```haskell
divide :: Monad m => Int -> Int -> ErrTM m Int
```

```haskell
divisions :: Monad m => ErrTM m [Int]
divisions = do
  a <- divide 10 20
  b <- divide 30 40

  c <- divide 10 02
  return [a,b,c]
```

- We can even run it in the IO monad and use lift to access the wrapped IO monad

```haskell
ex1c = runErrTM $ do
  eHandle (do x <- divisions
              lift $ print x )
          (do lift $ putStrLn "Error")
```

Further Reading

- http://taylor.fausak.me/2015/05/14/monad-transformers/
    - Intro to how monad transformers work
        - Monads used in the example:
            - Identity - does nothing, expansion of identity function, can use to use the do notation without introducing restrictions
            - Reader - provides read only data, good for configuration handling

-
```haskell
type Input = Integer
type Output = String

aReader :: Reader Input Output
aReader = do
    x <- ask
    let s = "The input was " ++ show x
    return s

>>> runReader aReader 3
"The input was 3"
```

- Writer - opposite of the reader monad, giving write-only data - useful for logging - similar usage to Reader

-
```haskell
type Output = [String]
type Result = Integer

aWriter :: Writer Output Result
aWriter = do
    let x = 3
    tell ["The number was " ++ show x]
    return x

>>> runWriter aWriter
(3,["The number was 3"])
```

- lift used to interact with Monads in the stack
- Identity monad will be the base - does nothing but can lets us stack monads. IO is a similar alternative
- Next layer is a Reader, this would probably be some config.
- Top layer is our Writer, it will accumulate a list of strings

-
```haskell
import Control.Monad.Trans.Class (lift)
import Data.Functor.Identity (Identity, runIdentity)
import Control.Monad.Trans.Reader (ReaderT, ask, runReader)
import Control.Monad.Trans.Writer (WriterT, tell, runWriter)

type Input = Integer
type Output = [String]
type Result = Integer

stack :: WriterT Output (ReaderT Input Identity) Result
stack = do
    x <- lift ask
    tell ["The input was " ++ show x]
    return x
```

  - That's the stack in action! Writer is the outermost monad so we dont need to lift tell
  - Reader is within the Writer monad, so we **lift our ask operation <u>into</u> the Reader monad**
  - Only one lift is needed no matter how many monads are in our stack

- The only downside is that you need to run all of these monads. Doing so can be a little tedious.

  ```
  >>> let newReader = runWriterT stack
  >>> let newIdentity = runReaderT newReader 3
  >>> runIdentity newIdentity
  (3,["The number was 3"])
  ```

- https://page.mi.fu-berlin.de/scravy/realworldhaskell/materialien/monad-transformers-step-by-step.pdf