CS4012 Topics in Functional Programming

Glenn Strong

Monad Transformers

- We are going to investigate an advanced use of Monads
- We will do this because we want to talk about Concurrency, and life will (ultimately) be much easier if we have these monadic tools available

Let's launch straight in. This is derived from Koen Claessen's 1999 A poor man's concurrency monad

A Concurrency Monad

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

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

```
instance Functor CM where
  fmap = liftM

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

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

With operations

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

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'
```

All fine, but we would like more!

For example:

- More IO operations
- Inter-thread communication
- Integrated logging
- Error handling for when threads fail.

- We could add more primitives?
- We could embed monads for each thing
  - 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
    - (OK, let's embed *two* instances of `Writer`?)
  - Embed an error monad (`Maybe` for example)

This sounds like it will be a complex nightmare.

Monad Transformers

The problem is that monads do not naturally compose.

Imagine a calculator program. We might want a state monad to manage the calculator memory, letting us write things like:

```
do
  x <- get
  y <- divide 100 x
  put y
```

But what if we want to manage errors in the calculator as well? Using something like the `Maybe` monad we could allow `divide` to return `Nothing`.

Or (better!), we could use some other monad that let us embed an error message - some kind of "exception monad". For example,

```
data MonadOfExceptions e v = Error e | Success v
```

With all the usual monadic stuff, and a function like `handleError` that acts as an exception handler

But it's not at all obvious how to combine the two monads in one calculation.

```
do
  x <- get
  y <- ( divide 100 x ) `handleError` ( return 0 )
  put y
```

There is a problem with the types here...

Review

- Monads can be handy for capturing ideas like state, partiality, concurrency.
- What happens if we want to combine those ideas?
- Say, a monad that offers both state and partiality?
- We can write a new State-Maybe monad
- Do we want: `newtype SM s a = SM (s -> Maybe (a, s))`
- or maybe: `newtype SM s a = SM (s -> (Maybe a, s))`
- Not necessarily obvious which is "right".

There is going to be plenty of duplicated effort

- Patterns for specific effects will be repeated over and over
- Adding features from more than one monad sounds miserable!
- A different approach is in order!
- We will decorate the monads with extra features
- This will allow us to graft together features of monads

Monad transformers

- Key idea:
  - Define features (state, error handling, etc) not as monads themselves
  - Define them as functions from monads to monads
  - We'll have to do some work up front but it will pay off
- We want to be able to *combine* monads so that we can conveniently mix effects from different monads

Since there will be more than one concrete implementation we will make a *class* for the effects we plan to include. For example:

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

This states that for *any* monad `m` we can transform it into a monad that does all the stuff `m` does *and* it does error handling using `eFail` and `eHandle`

For this to be useful we need to have a way to access the operations of the original monad m.

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

The sole operation of this class, lift allows us to access the operations of the transformed monad.

We need to create at least one concrete instance of this:

```
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
```

Next we need to explain how actions in the error monad can access actions in the transformed monad

```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
```

OK, we are ready to go. Recall our division function?

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

The type here is:

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

That is, the divide function lives in our error monad, and does not care what the "other" (transformed) monad might be

We can combine these division actions:

```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 could run these in the IO monad. When we need to access the "wrapped" monad we use lift

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

The "normal" monadic computation is of our error monad type

What if we want just the error monad?

A dummy "identity" monad gives us a neutral semantics

```
newtype I a = I a
```

Make it a monad...

```
instance Functor I where
  fmap = liftM
instance Applicative I where
  pure = I

instance Monad I where
    (I m) >>= f = f m
```

And give it the usual "run" operation

```
runI :: I a -> a
runI (I x) = x
```

Let's work through a full example interpreter, inspired by the excellent Monad Transformers Step by Step by Martin Grabmüller.