Monad Transformers (based on notes by Stephanie Weirich)

1

Schedule

	presentations			
Tue Nov 26	Property Testing and QuickCheck		~~Quiz 5 on Tue~~ Lecture_Week14 Lecture14.hs	PS8 due on Tuesday
Tue Dec 3 Fri Dec 6	Monad Transformers		Lecture_Week15 Lecture15.hs Lecture15'.hs Jumbo Quiz 6 on Fri	**Checkpoint #2: attend office hours this week (or earlier)**
Tue Dec 10	Project presentations			**Project due** **5-8 min presentation in class**

2

Monads are Useful!

- We programmed many monadic computations
- `Maybe`, `[]` and `(Either a)` monads handle errors
- `(State s)` emulates stateful computations in a pure language
- `Parser` enables convenient encoding of recursive-descent parsers
- `(Gen a)` delivers powerful random test generation methodology
- `(IO a)` "hides" computation with inherent side effects

Outline

- Exception and State monads
- Monad transformers
  - Step 1: Bottling up monad features into type classes
  - Step 2: A Jumbo monad
  - Step 3: Instances of monad transformers
  - Step 4: Lifting
- The big picture

Download Lecture15.hs and code along

Exception Monad

- An expression language with only a division operation and an evaluation function:

```
data Expr = Val Int
          | Div Expr Expr
          deriving (Show)

eval (Val i) = i
eval (Div e1 e2) = eval e1 `div` eval e2
```

- What can go wrong and what can we do to fix?
- One way is a default value, another is the `(Either a)` monad we used in `zipTree` and in Parsec

- Two terms:

```
ok :: Expr                    1800/2/21
ok = Div (Div (Val 1800) (Val 2)) (Val 21)

err :: Expr                    2/(1/(2/3))
err = Div (Val 2) (Div (Val 1) (Div (Val 2) (Val 3)))

> eval ok
42
> eval err
*** Exception ...
```

- With an (Either a) monad a Left value means an error happened somewhere along the evaluation of the tree, and a Right n means evaluation succeeded and result is n

```
errorS :: Show a => a -> a -> String
errorS x y = "Error dividing " ++ show x ++ " by " ++ show y

-- evalEx is the exception-throwing eval
evalEx :: Expr -> Either String Int -- Int is encaps. value
evalEx (Val n) = return n
evalEx (Div x y) = … rx ← evalEx x
                     ry ← evalEx y
                     if ry == 0 then Left (errorS rx ry)
                        else return (rx `div` ry)
```


- For convenience, let's create a unified view of Expr: showEx for printing error message and goEx for evaluation

```
showEx :: (Int -> String) -> Either String Int -> String
showEx _ (Left m) = "Raise: " ++ m
showEx s (Right n) = "Result: " ++ s n

goEx :: Expr -> String           Fancy Haskell for
goEx e = evalEx e & showEx show    showEx show (evalEx e)
-- `&` is reverse application: partial function (showEx show)
-- is applied on (evalEx e) of Either String a type

> goEx ok
"Result: 42"
> goEx err
"Raise: Error dividing 1 by 0"
```

State Monad

- We now want to create a profiler that counts how many division ops an expression triggers, and the state monad `(State s)` comes in useful

```
-- evalSt is a profiling eval, independent of previous one
evalSt :: Expr -> State Int Int --- encapsulated value is Int
evalSt (Val n) = return n
evalSt (Div x y) = do rx <- evalSt x
                      ry <- evalSt y
                      s <- S.get
                      S.put (s + 1)
                      return (rx `div` ry)
```


- Analogously to exception monad we follow unified view: `showSt` for printing result of `runState` and `goSt` for evaluation

```
-- shows result of runState parameterized by f to show value
showSt :: (a -> String) -> (a, Int) -> String
showSt f (v, s) = f v ++ ", count: " ++ show s

-- profiling eval
goSt :: Expr -> String
goSt e = evalSt e & flip S.runState 0 & showSt show

> goSt ok
"42, count: 2"
> goSt err
"Exception: ..."
```

Monad Transformers

- So far, each monad does one thing. Exception monad does exception handling and state monad does stateful computation
- Our goal is to combine functionality, i.e., have an expression evaluation that does both exception handling stateful traversal
- Solution: add functionality as type-level functions from monad to monad with monad transformers. Identity monad passed to a StateT monad transformer produces a monad with state functionality, which in turn is passed to an ExceptT monad transformer that adds exception functionality. Etc.
- Decorator design patter is a useful analogy!

Step 1: A Type Class Describing Features

- First step is to define a type class that describes a monad with specific features
- `MonadError e` is a monad with the exception handling feature:

  `e` is the type of the error message, e.g., `String`; `m a` is the monad encapsulating value of type `a`, e.g., the `Int` result of evaluation

```
class Monad m => MonadError e m where
  throwError :: e -> m a
```

- Let's make `(Either e)` instance of the `(MonadError e)` type class

```
instance MonadError e (Either e) where
   -- throwError :: e -> Either e a
   throwError msg = Left msg
```

Aside

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

- We've seen many type classes, and they were all instantiated with either kind * (read: Type) or kind * -> * (read: Type goes to Type)

```
> :k Show
* -> Constraint
> :k Eq

> :k Monad
(* -> *) -> Constraint
> :k Foldable
```

- `MonadError` is a multi-parameter type class. What is its kind?

```
> :k MonadError
* -> (* -> *) -> Constraint
```

e param m param

- Remove signature of `evalEx` and replace `Left (errorS x y)` with `throwError (errorS x y)`:

```
errorS :: Show a => a -> a -> String
errorS x y = "Error dividing " ++ show x ++ " by " ++ show y

evalEx :: Expr -> Either String Int
evalEx (Val n) = return n
evalEx (Div x y) = do rx <- evalEx x
                      ry <- evalEx y
                      if ry == 0 then throwError (errorS rx ry)
                      else return (rx `div` ry)

> :t evalEx
evalEx :: MonadError String m => Expr -> m Int
```

- Return value is now a generic `m Int` where `m` is an instance of `(MonadError String)` > evalEx or :: Either String Int

- Analogously to `MonadError e`, `MonadState` s is a monad defining the key features of state: `get` and `put`

  s is the type of state, e.g., `Int`, `m a` is the monad encapsulating value of type a, e.g., the `Int` result of evaluation

```
class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()
```

- Make `(State s)` an instance of the (`MonadState s`) type class:

```
instance MonadState s (State s) where
   -- get :: State s s
   get = S.get
   -- put :: s -> State s ()
   put = S.put
```


- Analogously to `MonadError`, let's remove type signature and replace `S.get` and `S.put` with `get` and `put` (the interface of `StateMonad`):

```
~~evalSt :: Expr -> State Int Int~~
evalSt (Val n) = return n
evalSt (Div x y) = do rx <- evalSt x
                      ry <- evalSt y
                      (s :: Int) <- get
                      put (s + 1)
                      return (rx `div` ry)

> :t evalSt
evalSt :: MonadState Int m => Expr → m Int
```

Aside

```
{-# LANGUAGE ScopedTypeVariables #-}
```

- What happens if we did this:

```
~~evalSt :: Expr -> State Int Int~~
evalSt (Val n) = return n
evalSt (Div x y) = do rx <- evalSt x
                      ry <- evalSt y
                      s <- get -- <--
                      put (s + 1)
                      return (rx `div` ry)
```

- A type inference error, algorithm is unable to deduce a type for term `evalSt`. `(s :: Int)` explicitly instantiates to `MonadState Int`.

Step 2: A New Jumbo Monad

- `evalJumbo` has the functionality for exception handling and functionality for stateful computation. Note that `m` is both a `(MonadError String)` and a `(MonadState Int)`

```
evalJumbo :: (MonadError String m, MonadState Int m) =>
                                              Expr -> m Int
evalJumbo (Val n) = return n
evalJumbo (Div x y) = do rx <- evalJumbo x
                         ry <- evalJumbo y
                         if ry == 0
                           then throwError $ stateS rx ry
                           else do (s :: Int) <- get
                                   put (s + 1)
                                   return (rx `div` ry)
```

- Question is, how do we construct a monad to instantiate and run `evalJumbo` with?

- First, we need a newtype

```
newtype Jumbo a =
   Jumbo { runJumbo :: Int -> Either String (a, Int) }
```

- Next, define `return` and >>= to complete the instance of Monad. (Don't forget that Haskell forces us to write Applicative and Functor as well.)
- Finally, make Jumbo an instance of `MonadError String` (`throwError`) and `MonadState Int` (`get` and `put`)


```
newtype Jumbo a = Jumbo { runJumbo :: Int -> Either String (a, Int) }
```

- Our `Jumbo` monad will be a jumble of `(Either e)` monad functionality, if you remember from awhile ago, and `(State s)` functionality. We did both in Lecture11 on the State monad

```
instance Monad Jumbo where
  -- return :: a -> Jumbo a
  return v = Jumbo $ \s -> Right (v, s)
  -- >>= :: Jumbo a -> (a -> Jumbo b) -> Jumbo b
  ja >>= f = Jumbo $ \s -> case runJumbo ja s of
                              Left err -> Left err
                              Right (v, s') -> runJumbo (f v) s'
```

- We already wrote this code. It is even more annoying now as we need to combine with `(Either e)` and use a case-of

```
newtype Jumbo a = Jumbo { runJumbo :: Int -> Either String (a, Int) }
```

- We shouldn't forget that for the monad to work in Haskell, it needs to be Applicative as well:

```
instance Applicative Jumbo where
  -- pure :: a -> Jumbo a
  pure = return
  -- (<*>) :: Jumbo (a -> b) -> Jumbo a -> Jumbo b
  (<*>) = ap
```

- and a Functor:

```
instance Functor Jumbo where
  -- fmap :: (a -> b) -> Jumbo a -> Jumbo b
  fmap = liftM
```



```
newtype Jumbo a = Jumbo { runJumbo :: Int -> Either String (a, Int) }
```

- Finally, let's instantiate `MonadError String`

```
instance MonadError String Jumbo where
  -- throwError :: String -> Jumbo a
  throwError msg = Jumbo $ \_ -> Left msg
```

- and `MonadState Int` type classes

```
instance MonadState Int Jumbo where
  -- get :: Jumbo Int
  get = Jumbo $ \s -> Right (s,s)
  -- put :: Int -> Jumbo ()
  put s = Jumbo $ \_ -> Right ((),s)
```

```
newtype Jumbo a = Jumbo { runJumbo :: Int -> Either String (a, Int) }
```

- Take some time to call `evalJumbo`

```
goJumbo :: Expr -> String
goJumbo exp = evalJumbo exp &
                  flip runJumbo 0 & showJumbo
  where showJumbo :: Either String (Int,Int) -> String
        showJumbo = showEx (showSt show)

> goJumbo ok

> goJumbo err -- the error term
```


Step 3: Instances of Monad Transformers

- A better way to get a "jumbo monad" is to add features to existing monads
- A monad transformer is a type operator `t` that maps an existing monad `m` to a new monad `t m`. New monad "inherits" features of `m`

We'll start with adding exception functionality and this newtype. What is its kind?

```
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }

> :k ExceptT
```

```
> :k ExceptT
ExceptT :: * -> (* -> *) -> * -> *
```

- ExceptT takes argument e (i.e., the Left part, typically String), a monad m and an a (i.e., in our example a is Int, the type of the "happy path" result of expression evaluation)

- ExceptT is a lot like Either before, except that here Either is enclosed into a monad

```
-- ExceptT :: Type -> (Type -> Type) -> Type -> Type
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }
```

- So, what comes next? Notice the structure of ExceptT: We can make (ExceptT e m) a monad, just as we made (Either e) a monad before

- Importantly, (ExceptT e m) is of the right kind : * -> *


```
-- ExceptT :: Type -> (Type -> Type) -> Type -> Type
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }
```

- We'll use runExceptT and MkExc similarly to the Jumbo monad

```
instance Monad m => Monad (ExceptT e m) where
  return v = MkExc $ return (Right v)

  -- >>= :: ExceptT e m a -> (a -> ExceptT e m b) -> ExceptT e m b
  eta >>= f = MkExc $ runExceptT eta >>=
                 \s -> case s of
                         Left e -> return (Left e)
                         Right v -> runExceptT (f v)
```

- A lot to unpack… return wraps value v, first in Right, then this monad

- >>= unwraps argument eta and extracts s. s can be either Left or Right. Left simply propagates. Right first triggers (f v) then runExceptT (f v)

```
-- ExceptT :: Type -> (Type -> Type) -> Type -> Type
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }
```

- To conclude, make `(ExceptT e m)` an instance of `(MonadError e)` implementing `throwError`

```
instance Monad m => MonadError e (ExceptT e m) where
  -- throwError :: e -> ExceptT e m a
  throwError msg = MkExc (return (Left msg))
```

- Lots to unpack again… `return` wraps the `Either` value, `Left` in this case, into parameter monad `m`

Now we'll define the State functionality starting with this newtype: What is its kind?

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

> :k StateT
```

```
> :k StateT
StateT :: * -> (* -> *) -> * -> *
```

Notice the similarity with the `ExceptT` newtype:

```
-- ExceptT :: Type -> (Type -> Type) -> Type -> Type
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }
```

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

- We'll use `runStateT` and `MkStateT` analogously to `ExceptT`

```
instance Monad m => Monad (StateT s m) where
  return v = MkStateT $ \s -> return (v,s)
  -- >>= :: StateT s m a -> (a -> StateT s m b) -> StateT s m b
  sta >>= f = MkStateT $ \s -> do
                     (r,s') <- runStateT sta s
                     runStateT (f r) s'
```

- To unpack this… encapsulated type is NOT `m (s -> (a,s))`
- `>>=` runs `sta` on `s` and extracts `(r,s')`. It then runs `(f r)` monad on `s'`, much like what the old state transformer did

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

- What remains is making `(StateT s m)` an instance of `(MonadState s)` by implementing `put` and `get`

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

Step 4: Lifting

So far, we have

- Provided `m` is a monad, `(ExceptT e m)` is a Monad
  - Thus, we can use `return` and >>= to construct (complex) exception-throwing computations
- Provided `m` is a monad, `(ExceptT e m)` is also a `(MonadError e)`
  - Thus, we can use `throwError`

What we don't have, but would be nice to have

- `(ExceptT e m)` also be a `(MonadState s)`
  - Then we'll be able to use `get` and `put` as well
  - And as you might expect, we'll instantiate `m` with some `MonadState` instance and reuse `m`'s implementation of `get` and `put`

Analogously, so far, we have

- Provided `m` is a monad, `(StateT s m)` is a monad
  - Thus, we can use `return` and >>= to construct (complex) stateful computations
- Provided `m` is a monad, `(StateT s m)` is also a `(MonadState s)`
  - Thus, we can use `get` and `put`

What we don't have, but would be nice to have

- `(StateT s m)` also be a `(MonadError e)`
  - Then we'll be able to use `throwError`
  - Same here, we'll instantiate `m` with a `MonadError` and reuse its `throwError`

- The whole point is to instantiate and run `evalJumbo`. It uses `MonadError` features <u>and</u> `MonadState` features

```
evalJumbo :: (MonadError String m, MonadState Int m) =>
                                            Expr -> m Int
evalJumbo (Val n) = return n
evalJumbo (Div x y) = do rx <- evalJumbo x
                         ry <- evalJumbo y
                         if ry == 0
                           then throwError $ stateS rx ry
                           else do (s :: Int) <- get
                                   put (s + 1)
                                   return (rx `div` ry)
```

- To run `evalJumbo`, we need to instantiate `(m Int)` into either `StateT Int (…) Int` or `ExceptT String (…) Int`

`{-# LANGUAGE KindSignatures #-}`

- Generic solution: the `MonadTrans` type class that, informally speaking, defines an interface for lifting (i.e., transforming) one monad into another

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

Notes:

- We need to specify the kind for the monad transformer parameter
- Now, move to Lecture15'.hs

- Let's pick `evalJumbo :: Expr -> StateT Int (…) Int`
- First, make `(StateT s)` instance of `MonadTrans`. Why `(StateT s)`?

```
instance MonadTrans (StateT s) where
  -- lift :: m a -> StateT s m a
  -- Extracts value "a" from argument monad and
  -- encapsulates into a state:
  lift ma = MkStateT $ \s -> do r <- ma
                               return (r,s)
```

- And finally, using `lift`, we make `StateT s m` an error monad:

```
instance MonadError e m => MonadError e (StateT s m) where
  throwError :: e -> (StateT s m) a -- generic was e -> m a
  throwError err = lift $ throwError err
```



```
instance MonadError e m => MonadError e (StateT s m) where
  throwError :: e -> (StateT s m) a
  throwError err = lift $ throwError err
```

- So far so good!
- If `m` is an instance of `(MonadError e)`, then we have that `(StateT s m)` is an instance of `(MonadError e)` as well, and it supports `throwError`
- This is hard to wrap head around, but bottom line is callee `throwError` comes from inner monad `m`:

```
lift (throwError err)  is of type  State s m a
throwError err  is of type  m a
                     (as lift is m a -> State s m a)
```

- So how do we use the (StateT s m) monad transformer?
- Remember what our goal was: construct a computation that has both MonadError (throwError) and MonadState (get and put) features, then instantiate evalJumbo

```
evalJumbo :: (MonadError String m, MonadState Int m) =>
                                             Expr -> m Int
…
```

- One way is to instantiate m with (Either String), provided that (Either e) instantiates (MonadError e)

```
> runStateT (evalJumbo ok :: StateT Int (Either String) Int) 0

> runStateT (evalJumbo err :: StateT Int (Either String) Int) 0
```


- Another way is to instantiate m with (ExceptT String Identity), having in mind the definition of ExceptT:

```
-- ExceptT :: Type -> (Type -> Type) -> Type -> Type
newtype ExceptT e m a = MkExc { runExceptT :: m (Either e a) }
```

- What happens if we run this:

```
> runStateT (evalJumbo ok :: StateT Int (ExceptT String Identity) Int) 0
```

- Ah, value we want (either Raise error message or Int Result) is enclosed into a MkExc constructor, as well as an Identity monad

```
> runExceptT
     (runStateT
         (evalJumbo ok :: StateT Int (ExceptT String Identity) Int) 0)
Identity (Right (42,2))
```

```
> runExceptT
     (runStateT
         (evalJumbo err :: StateT Int (ExceptT String Identity) Int) 0)
Identity (Left "Error dividing 1 by 0")
```

- As an exercise, write a `goStEx` wrapper which will avoid the mess

Exercise

- Let's turn around: `evalJumbo :: Expr -> ExceptT String (…) Int`
- First, make `(ExceptT e)` instance of `MonadTrans`:

```
instance MonadTrans (ExceptT e) where
  -- lift :: m a -> ExceptT e m a
  -- Extracts value from ma and encloses into Right
  lift ma = …
```

- Next, make `ExceptT e m` a state monad (provide `m` is a state monad):

```
instance MonadState s m => MonadState s (ExceptT e m) where
  get :: ExceptT e m s
  get = …
  put :: s -> Except e m ()
  put v = …
```

- We have

```
evalJumbo ok :: ExceptT String (StateT Int Identity) Int
```

```
> …
```

```
> …
```

- Do you get the same result?

Big Picture

- What is the point of all this?
- We have many monads that do one thing: e.g., `ExceptT`, `StateT`, `SomeT`
- We'll make those monads into transformers by having each `ExceptT`, `StateT`, `SomeT` define corresponding instance of `MonadTrans`
- We can compose these in some sequence producing a Jumbo monad in a way that is very similar to the Decorator Design pattern