CS 115
Functional Programming

Lecture 22:

Monad Transformers

```
(* -> *) -> (* -> *)
(* -> *) -> (* -> *)
(* -> *) -> (* -> *)
(* -> *) -> (* -> *)
(* -> *) -> (* -> *)
```

Previously

- State monads
- Error-handling monads
- Parser combinators

Today

- More on the `IO` monad
- Monad transformers
  - Combining state-passing and error-handling monads

More on the `IO` monad

- So far, we've been using the `IO` monad in a hand-wavy manner
  - more "practical" than theoretical
- We haven't worried too much about how it actually works
- Now that we've studied state monads, we can go "inside the `IO` monad"

Good news!

- If you understand state monads, you already understand the `IO` monad!
- The `IO` monad is essentially just a state monad that uses a special value (which we can call `realWorld`) as its state value
- (I'm glossing over some nonessential details here)
- The purpose of the `realWorld` value is to ensure that `IO` actions are performed in the correct sequence

Sequencing

- Consider the code:

```
putStr "hello" >> putStrLn "world"
```

- We know what we want this to do:
  - print out `hello`, then
  - print out `world`
- We need to enforce sequencing so that `world` doesn't get printed out before `hello`!
- Using a state monad with the `realWorld` value makes sure that this happens
  - the `putStrLn` can't proceed until it gets the `realWorld` value as the (state) result of the `putStr`

Uniqueness of `realWorld`

- OK, so the state monad will give us the sequencing we need
- Why should we need the **`realWorld`** value?
- Why not just use an arbitrary value of *e.g.* an integer?
- Answers:
  - there should only be *one* value of the **`realWorld`** type
  - you should not be able to manipulate this value directly (opaque value) or conjure it up out of nothing

Uniqueness of `realWorld`

- Therefore, the `realWorld` value is unique (the only instance of the type `RealWorld`) and not user-manipulable
- When the program as a whole (of type `IO ()`) starts, the runtime system of GHC passes it the `realWorld` value and the program runs to completion, passing `realWorld` to every `IO` computation that uses it
- This ensures that `IO` computations are sequenced properly (like a baton in a relay race)
- However, there is one way around this...

`unsafePerformIO`

- The `unsafePerformIO` function (in the `System.IO.Unsafe` module)
- `unsafePerformIO` takes the `realWorld` value and passes it to an `IO` computation to get the result, even when it is *not* part of a sequence of `IO` computations joined together with `>>=`
- In other words, it ignores the sequencing constraint of `realWorld` and just "does the computation"
- This can have a number of unpleasant effects if used unwisely

`unsafePerformIO`

- `unsafePerformIO` is only safe to use if
  - the order in which the `IO` computation is performed relative to the "main line" of `IO` computations doesn't matter
    - *e.g.* in printing outputs by the `trace` function in `Debug.Trace`
  - the compiler is not allowed to inline the code (compiler pragma `NOINLINE` must be in effect)
    - this is beyond the scope of this class
- Bottom line: if you're not an expert, don't use `unsafePerformIO`

New topic!

Monad transformers

- So far, we have seen a number of monads
  - `IO`, `Maybe`, list, error, `State`, parsing
- Each monad represents a particular kind of computation and makes it convenient to perform that kind of computation
  - `Maybe`: computations that may fail
  - list: computations that may return multiple results
  - error: computations that may fail with particular error conditions
  - `State`: computations that may manipulate state
  - `IO`: computations that do input/output
  - parsing: computations that involve parsing! ☺

Monad transformers

- Problem: each monad represents only *one* kind of computation
- If we have a "notion of computation" that combines the notions of computation of two or more monads, using the monads we've just described won't suffice
- We have two options:
  - build a new (complex) monad that combines the computational effects of more than one monad
  - use *monad transformers* to generate the new monad from two or more old ones

Example

- We will structure the rest of the discussion around computations that can read/write state (like the `State` monad) and throw/catch particular errors (like error-handling monads)
- We will first build our own custom monad to do this
- Then we will instead use a monad transformer to do the exact same thing with far less effort

Sample problem

- To make this more concrete, we'll use the resulting monad to solve an actual (trivial) problem: computing factorials using a stateful algorithm, with error reporting if the argument is negative
- However, the monads we generate can be used in any situation where you want to have a combination of state-passing and error-handling
- They will also be purely functional!

The `StateError` monad

- We will call our monad **`StateError`**, since it's conceptually a combination of a **`State`** monad and an error-handling monad
- The computations we are modeling can be represented schematically like this:

```
     [read/write state, throw/catch errors]
a -----------------------------------> b
```

The `StateError` monad

- We need to represent `StateError` as a datatype
- Recall that `State` was represented like this:

```
data State s a = State (s -> (a, s))
```

- And error-handling monads are usually represented as `Either e a` for some error type `e`
- We need to combine them, but how?
- We need to think about how we want to structure the computation, then represent that in the datatype

The `StateError` monad

- Our datatype will pass a state value of type `s` from computation to computation
- Each computation takes a state value and either
  - succeeds, computing a new state value of type `s` and returning a value of type `a`, OR
  - fails, returning an error condition of type `e`
- Therefore, the natural way to represent this in a Haskell datatype is:

```
data StateError e s a =
  StateError (s -> Either e (a, s))
```

The `StateError` monad

```
data StateError e s a =
  StateError (s -> Either e (a, s))
```

- *Note:* an alternative way to represent this is:

```
data StateError e s a =
  StateError (s -> (Either e a, s))
```

- This represents a computation that always returns the state, even if an error occurs
- Depending on what you want, you might prefer this
- (Interesting design decisions!)

The `StateError` monad

```
data StateError e s a =
  StateError (s -> Either e (a, s))
```

- The monad will therefore be what...?
- Recall that a monad is a unary type constructor (of kind `* -> *`)
- We will have:

```
instance Monad (StateError e s) where
  return = {- to be filled in -}
  mv >>= f = {- to be filled in -}
```

return

- Recall that `return` in the `State` monad had the definition

```
return x = State (\st -> (x, st))
```

- and in error-handling monads, it was:

```
return x = Right x
```

- What should it be in the `StateError` monad?

```
return x = StateError (\st -> Right (x, st))
```

>>=

- Recall that `>>=` in the `State` monad had the (fairly complex) definition:

```
mv >>= g
    = State (\st ->
            let (State ff) = mv
                (y, st') = ff st
                (State gg) = g y
            in gg st')
```

>>=

- For the error monad `(Either e)`, the definition is:

```
(Left x)  >>= _ = Left x
(Right y) >>= f = f y
```

- We need to somehow combine this with

```
mv >>= g
    = State (\st ->
          let (State ff) = mv
              (y, st') = ff st
              (State gg) = g y
          in gg st')
```

- to get the definition for `StateError`

>>=

- First, we convert the `State` constructors to `StateError` constructors:

```
mv >>= g
    = StateError (\st ->
            let (StateError ff) = mv
                (y, st') = ff st
                (StateError gg) = g y
            in gg st')
```

- This will not type check! Why?

>>=

```
mv >>= g
    = StateError (\st ->
          let (StateError ff) = mv
              (y, st') = ff st   -- wrong!
              (StateError gg) = g y
          in gg st')
```

- Note that **`StateError`** has the type:

```
data StateError e s a =
  StateError (s -> Either e (a, s))
```

- Need a **`case`** statement to unpack **`Either`** type

>>=

```
mv >>= g
  = StateError (\st ->
      let (StateError ff) = mv in
        case ff st of
          Left err -> Left err
          Right (y, st') ->
            let (StateError gg) = g y
            in gg st')
```

- This is the correct definition of `>>=` for the `(StateError e s)` monad

Monad laws

- We're not going to try to verify the monad laws for this monad
- It's straightforward and very tedious
- It would only get more tedious to derive the `return` method and `>>=` operator and verify the monad laws for an even more complex monad (say, that also incorporated `IO`)

So far

- The full **Monad** instance for **StateError** is:

```
instance Monad (StateError e s) where
  return x = StateError (\st -> Right (x, st))

  mv >>= g
    = StateError (\st ->
        let (StateError ff) = mv in
          case ff st of
            Left err -> Left err
            Right (y, st') ->
              let (StateError gg) = g y
              in gg st')
```

Simplifying

- We can simplify this by defining a `runStateError` function much like the `runState` function:

```
runStateError :: StateError e s a -> (s -> Either e (a, s))
runStateError (StateError f) = f
```

- We can even put this right into the definition of the `StateError` datatype:

```
data StateError e s a = StateError
  { runStateError :: s -> Either e (a, s) }
```

- We can use `runStateError` to simplify the `Monad` instance for `StateError`

Simplifying

- Using **`runStateError`** gives us:

```
instance Monad (StateError es) where
  return x = StateError (\st -> Right (x, st))

  mv >>= g
    = StateError (\st ->
        case runStateError mv st of
          Left err -> Left err
          Right (y, st') -> runStateError (g y) st')
```

MonadState

- We need to also define a `MonadState` instance for `StateError` so we can use `get` and `put` in the `StateError` monad
- This is easy because neither `get` nor `put` can fail:

```
instance MonadState s (StateError e s) where
  get = StateError (\st -> Right (st, st))
  put st = StateError (\_ -> Right ((), st))
```

MonadError

- Similarly, we need to define a **MonadError** instance for **StateError** so we can use **throwError** and **catchError** in the **StateError** monad
- **throwError** is easy: discard the state and return the error value

```
instance MonadError e (StateError e s) where
   throwError err = StateError (\_ -> Left err)
   catchError mv h = {- to be defined -}
```

MonadError

- **catchError** is trickier; let's handle the non-failure case first
- It just passes the monadic value through:

```
instance MonadError e (StateError e s) where
  throwError err = StateError (\_ -> Left err)
  catchError mv h =
    StateError (\st ->
      case runStateError mv st of
        Left err -> {- to be defined -}
        Right (x, st') -> Right (x, st'))
```

MonadError

- If no error was thrown, this would be equivalent to:

```
catchError mv h =
  StateError (\st ->
    runStateError mv st)
```

- Or:

```
catchError mv h =
    StateError (runStateError mv)
```

- Or:

```
catchError mv h = mv
```

MonadError

- If an error was thrown, we would have to apply the handler `h` to the error value `err` to get:

```
catchError mv h =
  StateError (\st ->
    case runStateError mv st of
      Left err -> ... (h err) ...
      Right (x, st') -> Right (x, st'))
```

- `(h err)` has the type `StateError e s a`
- We need a value of type `Either e (a, s)`
- Try applying `runStateError` to `(h err)`

MonadError

- This gives us:

```
catchError mv h =
  StateError (\st ->
    case runStateError mv st of
      Left err -> runStateError (h err) ...
      Right (x, st') -> Right (x, st'))
```

- `runStateError (h err)` has the type `(s -> Either e (a, s))`
- Need to apply this to the state to get a value of type `Either e (a, s)`

MonadError

```
catchError mv h =
  StateError (\st ->
    case runStateError mv st of
      Left err -> runStateError (h err) st
      Right (x, st') -> Right (x, st'))
```

- This is the final definition of `catchError`
- Now we have the instance of `MonadError` for the `(StateError e s)` monad

Recap

- We've defined a new datatype called `StateError` to represent computations that manipulate state and that can throw/catch errors
- We defined a `Monad` instance for the type to allow us to compose `StateError` monadic functions
- We defined `MonadState` and `MonadError` instances which allow us to use convenient functions for accessing/changing state and throwing/catching errors in the monad

Recap

- We've seen that defining a custom monad that combines features from two other monads is a real pain in the neck (or other body parts)!
- Let's see how monad transformers will simplify this

StateT

- We will be using the `StateT` monad transformer
  - monad (T)ransformer names typically end in a capital `T`
- Its job is to "layer" state-manipulation behavior "on top of" another monad
- A monad transformer thus takes one monad, adds some features, and outputs another monad, which is the one we want
  - [Aside: Kind of reminiscent of functors in OCaml!]

StateError revisited

- The StateError datatype can be defined using the StateT monad transformer as follows:

```
data StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
```

Contrast this with the previous version:

```
data StateError e s a =
  StateError { runStateError :: s -> Either e (a, s) }
```

- StateT is defined in Control.Monad.State as:

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

`StateError` revisited

- `StateT` is defined in `Control.Monad.State` as:

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

- Here, `m` can be any monad
- It's called the "inner monad" of the transformer
- The monad we're actually using is the `Either e` monad, so this is equivalent to:

```
StateT { runStateT :: s -> Either e (a,s) }
```

`StateError` revisited

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

- The cool thing about `StateT` is that as long as `m` is a monad, `(StateT s m)` will be a monad too!
- In fact, there is a definition for these monads:

```
instance (Monad m) => Monad (StateT s m) where
  return a = StateT (\st -> return (a, st))
  mv >>= g  = StateT (\st -> do
       (y, st') <- runStateT mv st
       runStateT (g y) st')
```

`StateError` revisited

- Notice that the definitions of `return` and `>>=` are monadic as well (in the *inner* monad `m`)
- If `m` is `Either e`, this reduces to:

```
instance (Monad m) => Monad (StateT s m) where
  return a = StateT (\st -> Right (a, st))
  mv >>= g  = StateT (\st ->
      case runStateT mv st of
        Left err -> Left err
        Right (y, st') -> runStateT (g y) st')
```

- which is the same definition we derived before

StateError revisited

- If we had defined **StateError** like this:

```
type StateError e s a = StateT s (Either e) a
```

- then the previous **Monad** instance would work for this
- Since we wrapped this in a **data** definition:

```
data StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
```

- we have to "unwrap" the datatype to define a **Monad** instance
- This is boilerplate code, and Haskell should be able to do it for you

`StateError` revisited

- Let's make a small change:

```
newtype StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
```

- Recall that `newtype` is like `data` from the standpoint of type checking but behaves like a `type` alias at runtime
- Since this basically is the same as the `type` definition, we should be able to use the previously-defined `Monad` instance on this datatype

`StateError` revisited

- And we can!

```
newtype StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
    deriving (Monad)
```

- Cool! ☺

`StateError` revisited

- However, we still don't have the **`MonadState`** and **`MonadError`** instance definitions we need
- In fact, the **`MonadState`** instance has already been defined for all monads derived from the **`StateT`** monad transformer, so we can write:

```
newtype StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
    deriving (Monad, MonadState s)
```

- One down, one to go

`StateError` revisited

- You might just want to see if we can automatically derive **`MonadError`** as well:

```
newtype StateError e s a =
  StateError { runStateError :: StateT s (Either e) a }
    deriving (Monad, MonadState s, MonadError e)
```

- Amazingly enough, this works too!
- The module **`Control.Monad.Except`** defines a **`MonadError`** instance for monads derived from the **`StateT`** monad transformer, so we're all set
- I want to show you one more cool trick…

The `Identity` monad

- Recall that monads extend the notion of function application and function composition
- What would be the simplest possible monad?
- Answer: one that doesn't extend that notion at all!
- There is a monad called `Identity` which is essentially the same as function application and composition
- It lives in the module `Control.Monad.Identity`

The `Identity` monad

- It has this definition:

```
newtype Identity a = Identity { runIdentity :: a }
instance Monad Identity where
  return x = Identity x
  mv >>= g = g (runIdentity mv)
```

- Notice that except for the `Identity` wrapper, this is just the same as normal function application e.g.

```
return x = x
mv >>= g = g mv
```

- What's the use of this?

The `Identity` monad

- The `Identity` monad can be used in conjunction with monad transformers to construct "simple" monads *i.e.* ones that don't wrap anything
- Here is the actual definition of the `State` monad:

```
type State s = StateT s Identity
```

The `Identity` monad

- Similarly, there is a monad transformer called **`ExceptT`** for error-handling monads using the **`Either`** type constructor, and we can define our **`StateError`** type as:

```
newtype StateError e s a =
  StateError { runStateError ::
                    StateT s (ExceptT e Identity) a }
    deriving (Monad, MonadState s, MonadError e)
```

- We are now layering two monad transformers on top of the **`Identity`** monad to get our final datatype

Identity -> IO?

- If we used the `IO` monad as the innermost monad instead of `Identity`, we'd have:

```
newtype StateErrorIO e s a =
  StateErrorIO { runStateErrorIO ::
                   StateT s (ExceptT e IO) a }
    deriving (Monad, MonadState s, MonadError e)
```

- We would now have a monad that combines `IO`, error-handling and state-passing (very powerful!)
- We won't need this, so we'll go back to `StateError` and use it to do some computations

Recap

- Although we've done a lot of work, the total amount of code we've generated is quite small:

```
newtype StateError e s a =
  StateError { runStateError ::
                    StateT s (ExceptT e Identity) a }
    deriving (Monad, MonadState s, MonadError e)
```

- This is the entire definition of our monad!
- Using monad transformers has taken what would have been a very complicated and messy task and made it trivially easy

Note

- Recall the definition of `StateT`:

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

- Let's look at the kind of `StateT`

```
ghci> :kind StateT
StateT :: * -> (* -> *) -> * -> *
```

- This is the most complicated kind we've seen yet!

Note

- It's more intuitive if we supply the state:

```
ghci> :kind StateT (Integer, Integer)
StateT (Integer, Integer) :: (* -> *) -> * -> *
```

- Note that `->` associates to the right, so this is the same as:

```
StateT (Integer, Integer) :: (* -> *) -> (* -> *)
```

- Monads have the kind `(* -> *)`
- So we see that `StateT (Integer, Integer)` is like a type-level function on monads, which is what a "monad transformer" really is

Using `StateError`

- Now *using* the monad is quite trivial
- We'll write a stateful factorial function that can handle errors (factorial of negative numbers)
- The state will be contained in two `Integer` variables
- The result will be an `Integer`
- Errors will be represented by `String`
- Our principal datatype will thus be:

```
StateError String (Integer, Integer) Integer
```

Using `StateError`

- Here's the guts of the factorial function, written as a state transformer that also handles errors:

```
factorialSE ::
  StateError String (Integer, Integer) Integer
factorialSE = do
  (n, r) <- get
  case compare n 0 of
    LT -> throwError "invalid argument"
    EQ -> return r
    GT -> put (n - 1, r * n) >> factorialSE
```

Using `StateError`

- To use this, we also need to write a function to run the state transformer on the initial state and extract the result:

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- Let's unpack this definition

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- `mv` has the type `StateError e s a`

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- **`runStateError mv`** unpacks the state transformer from the **`StateError`** constructor and has the type **`StateT s (ExceptT e Identity) a`**

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- `runStateT se` unpacks the state transformer from the `StateT` constructor and has the type `s -> ExceptT e Identity (a, s)`

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- `runStateT se st` applies the state `st` to the previous state transformer and has the type `ExceptT e Identity (a, s)`

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- `runExceptT $ runStateT se st` unpacks the contents of the `ExceptT` datatype and has the type `Identity (Either e (a, s))`

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- `runIdentity $ runExceptT $ runStateT se st` unpacks the contents of the `Identity` datatype and has the type `(Either e (a, s))`
- Each step in the chain just unwraps one more layer until we see the true contents of the datatype

Using `StateError`

```
runSE :: StateError e s a -> s -> Either e (a, s)
runSE mv st =
  let se = runStateError mv in
    runIdentity $ runExceptT $ runStateT se st
```

- Since we don't want the final state, we can also define **`evalSE`** by analogy with **`evalState`** to discard the state at the end of the computation:

```
evalSE :: StateError e s a -> s -> Either e a
evalSE mv st =
  case runSE mv st of
    Left err -> Left err
    Right (x, _) -> Right x  -- discard state
```

Using `StateError`

- With all this done, defining **`factorial`** is easy:

```
factorial :: Integer -> Either String Integer
factorial n = evalSE factorialSE (n, 1)
```

- More importantly, we now have defined all the "plumbing" necessary for writing arbitrary computations that can
  - access/modify state variables
  - throw/catch errors
- in only a few lines of code!

One more thing

- Our definition of **`StateError`** is:

```
newtype StateError e s a =
  StateError { runStateError ::
                 StateT s (ExceptT e Identity) a }
    deriving (Monad, MonadState s, MonadError e)
```

- It turns out that any instance of the **`Monad`** type class also has to be an instance of **`Functor`** and **`Applicative`**
  - We haven't talked much about **`Applicative`** so far
  - but all **`Monad`** instances are automatically instances of **`Applicative`** and **`Functor`** anyway, so GHC can derive those instances

One more thing

- We need to rewrite **StateError** to:

```
newtype StateError e s a =
  StateError { runStateError ::
                    StateT s (ExceptT e Identity) a }
    deriving (Functor, Applicative,
                 Monad, MonadState s, MonadError e)
```

- This will silence any warnings

Summary

- We often want to combine monads
  - to get the effect of multiple "notions of computation" in a single computation
  - *e.g.* state + error, state + error + I/O
- Combining monads by hand is tedious and difficult
- Monad transformers make it extremely easy

Summary

- Most monads in Haskell are defined as monad transformers, and the "simple" monad is defined as the transformer acting on the `Identity` monad
- Once you've defined a monad in terms of monad transformers (a few lines of code) and written some interface code to run the computation (a few more lines) you can write code in the monad very easily
- Monad transformers are a powerful tool!

Summary

- Many if not most large Haskell programs are defined in terms of one or more "principal monads" which combine multiple monads using monad transformers
- The principal monad contains all the effects needed for the computations carried out by that program
- Each program will typically need a different group of effects

Caveats

- There are some issues/problems with monad transformers
- Some monads (`IO`, `ST`) can only be the innermost monad of a transformer for complicated reasons
- The order in which you stack monads (which one is innermost, which one is outermost) can have performance implications
- Writing "one big monad" can be more efficient than layers of monads

Next time

- Last lecture!
- Controlling strictness in Haskell programs