Monad Transformers

Background: abstracting over effects

works with any effect

```
add :: Monad m => m Int -> m Int -> m Int
add mx my = do
  x <- mx
  y <- my
  return (x + y)
```

IO

```
>>> add readIO readIO
5
7
12
```

Nondeterminism

```
>>> add [10,20] [1,3,5]
[11,13,15,21,23,25]
```

Failure

```
>>> add (Just 3) (Just 4)
Just 7

>>> add (Just 3) Nothing
Nothing
```

tracing, state, exceptions, …

Monads and effects

Monads help us to **structure** effects:

- write effect logic once (in `Monad` instance)
- sequence effectful code (with bind/do-notation)
- abstract over a variety of effects

What if we need more than one effect?

Monad transformers help us to **combine** effects:

- write *interaction logic* once (in `MonadTrans` instance)
- use multiple effects by *layering* monad transformers

Monad transformer

```
m :: * -> *
t :: (* -> *) -> * -> *
```

```
Monad (t m) =>
```

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

return for `m` *return for* `t m`

```
lift . return  <==>  return
```

`lift` *distributes over bind*

```
lift (m >>= f)  <==>  lift m >>= (lift . f)
```

Maybe monad transformer

```
MaybeT :: (* -> *) -> * -> *
```

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

Equivalent to:

```
data MaybeT m a = MaybeT (m (Maybe a))

runMaybeT :: MaybeT m a -> m (Maybe a)
runMaybeT (MaybeT x) = x
```

Maybe monad transformer

```
MaybeT :: (* -> *) -> * -> *
```

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

```
instance Monad m => Monad (MaybeT m) where
  return  = MaybeT . return . Just
  x >>= f = MaybeT $ do may <- runMaybeT x
                        case may of
                          Nothing -> return Nothing
                          Just a -> runMaybeT (f a)
```

do-block in m*!*

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

Maybe monad transformer

```
MaybeT :: (* -> *) -> * -> *
```

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

```
instance Monad m => MonadPlus (MaybeT m) where
  mzero     = MaybeT (return Nothing)
  mplus x y = MaybeT $ do may <- runMaybeT x
                          case may of
                            Just _  -> return may
                            Nothing -> runMaybeT y
```

(Password.hs) 

State monad transformer

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

```
data StateT s m a = StateT (s -> m (a,s))
```

Recall original state monad:

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

```
instance Monad (State s) where
  return x = State (\s -> (x,s))
  State c >>= f = State $ \s ->
                    let (x,t) = c s
                        State d = f x
                    in d t
```

State monad transformer

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

```
data StateT s m a = StateT (s -> m (a,s))
```

Recall original state monad:

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

```
instance Monad m => Monad (StateT s m) where
  return x = StateT (\s -> return (x,s))
  StateT c >>= f = StateT $ \s -> do
                     (x,t) <- c s
                     let StateT d = f x
                     return (d t)
```

do-block in `m`*!*

Other monad transformers

Box-like monads:

```
MaybeT (m (Maybe a))

ListT (m [a])

ExceptT (m (Either e a))
```

Computation-like monads:

	Original	*Transformer*
Writer	`Writer (a, w)`	`WriterT (m (a, w))`
Reader	`Reader (r -> a)`	`ReaderT (r -> m a)`
State	`State (s -> (a, s))`	`StateT (s -> m (a, s))`

Identity monad

A trivial monad – useful base of a monad transformer stack

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

```
instance Monad Identity where
  return = Identity
  Identity x >>= f = f x
```

```
   Maybe a  <~>  MaybeT Identity a
Writer w a  <~>  WriterT w Identity a
Reader w a  <~>  ReaderT w Identity a
 State s a  <~>  StateT s Identity a
            ...
```

Ordering monad transformers

The order that you layer effects matters!

```
StateT s (MaybeT Identity) a
```

corresponds to: `s -> Maybe (a, s)`

```
MaybeT (StateT s Identity) a
```

corresponds to: `s -> (Maybe a, s)`

new state even if computation fails!

(Semi-) automatic lifting

Some type classes to ease or automate lifting in deep stacks

Lift an IO action through all monad transformers:

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

"Primitives" that automate lifting:

check out the "mtl" library!

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

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

(KitchenSink.hs)