Monad Transformers
Background: abstracting over effects

works with any effect

\[
\begin{align*}
\text{add} :: \text{Monad } m \Rightarrow m \text{ Int} & \rightarrow m \text{ Int} & \rightarrow m \text{ Int} \\
\text{add } mx \text{ my} & = \text{ do} \\
& x \leftarrow mx \\
& y \leftarrow my \\
& \text{return } (x + y)
\end{align*}
\]

\[
\begin{align*}
\text{IO} \\
\text{>>> add readIO readIO} & \text{ 5} \\
& \text{ 7} \\
& \text{ 12}
\end{align*}
\]

\[
\begin{align*}
\text{Failure} \\
\text{>>> add (Just 3) (Just 4)} & \text{ Just 7} \\
& \text{ >>> add (Just 3) Nothing} \\
& \text{ Nothing}
\end{align*}
\]

\[
\begin{align*}
\text{Nondeterminism} \\
\text{>>> add } [10,20] & [1,3,5] \\
& [11,13,15,21,23,25]
\end{align*}
\]

\[
\text{tracing, state, exceptions, …}
\]
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

MonadTrans class

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

### Monad (t m) =>

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

**return for m** and **return for t m**

**lift distributes over bind**

- `lift (m >>= f)  <==>  lift m >>= (lift . f)`
Maybe monad transformer

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

Equivalent to:

```haskell
data MaybeT m a = MaybeT (m (Maybe a))
runMaybeT :: MaybeT m a -> m (Maybe a)
runMaybeT (MaybeT x) = x
```
Maybe monad transformer

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)

instance MonadTrans MaybeT where
    lift m = MaybeT (m >>= return . Just)
Maybe monad transformer

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
State monad transformer

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

Recall original state monad:

```haskell
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
        in  d t
        State d = f x
```

State monad transformer

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

Recall original state monad:
```
```haskell
data State s a = State (s -> (a, s))
```

```haskell
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:

<table>
<thead>
<tr>
<th>Original</th>
<th>Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Writer</strong></td>
<td><strong>WriterT (m (a, w))</strong></td>
</tr>
<tr>
<td><strong>Reader</strong></td>
<td><strong>ReaderT (r -&gt; m a)</strong></td>
</tr>
<tr>
<td><strong>State</strong></td>
<td><strong>StateT (s -&gt; m (a, s))</strong></td>
</tr>
</tbody>
</table>
Identity monad

A trivial monad – useful base of a monad transformer stack

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

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

```haskell
Maybe a  <->  MaybeT Identity a
Writer w a <->  WriterT w Identity a
State s a <->  StateT s Identity a
... 
```
Ordering monad transformers

The order that you layer effects matters!

\[
\text{StateT} \ s \ (\text{MaybeT} \ \text{Identity}) \ a
\]

corresponds to: \( s \rightarrow \text{Maybe} (a, s) \)

\[
\text{MaybeT} \ (\text{StateT} \ s \ \text{Identity}) \ a
\]

corresponds to: \( s \rightarrow (\text{Maybe} \ a, s) \)

\[
\text{new state even if computation fails!}
\]
(Semi-) automatic lifting

Some type classes to ease or automate lifting in deep stacks

Lift an IO action through all monad transformers:

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

“Primitives” that automate lifting:

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

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

check out the “mtl” library!

(KitchenSink.hs)