Monad Transformers
Background: abstracting over effects

works with any effect

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

**IO**

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

**Failure**

```haskell
>>> add (Just 3) (Just 4)
Just 7
>>> add (Just 3) Nothing
Nothing
```

**Nondeterminism**

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

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

\[
m :: \ast \rightarrow \ast
\]
\[
t :: (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast
\]

Monad \((t \ m) \Rightarrow \)

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

\(\text{return for } m\)

\(\text{return for } t \ m\)

\[
\text{lift . return} \iff \text{return}
\]

\(\text{lift distributes over bind}\)

\[
\text{lift} (m >>= f) \iff \text{lift} m >>= (\text{lift} \ . \ f)
\]
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

\[
\text{MaybeT :: } (* \rightarrow *) \rightarrow * \rightarrow *
\]

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

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

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

```haskell
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

\[ \text{StateT :: } * \rightarrow (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast \]

```
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
        in d t
```
State monad transformer

\[
\text{StateT} :: \ast \to (\ast \to \ast) \to \ast \to \ast
\]

\[
data \text{StateT } s \ m \ a = \text{StateT } (s \to m (a,s))
\]

Recall original state monad:

\[
data \text{State } s \ a = \text{State } (s \to (a,s))
\]

\[
\begin{align*}
\text{instance Monad } m & \Rightarrow \text{Monad } (\text{StateT } s \ m) \quad \text{where} \\
\text{return } x & = \text{StateT } (\backslash s \to \text{return } (x,s)) \\
\text{StateT } c \gg= f & = \text{StateT } $ \backslash s \to \text{do} \\
& \quad (x,t) \leftarrow c \ s \\
& \quad \text{let } \text{StateT } d = f \ x \\
& \quad \text{return } (d \ t)
\end{align*}
\]

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

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

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

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

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

\textit{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: 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)