Monad transformers
AFP Summer School 2017
Alejandro Serrano
Combining functors

Functors and applicative are closed under composition: if $f$ and $g$ are applicative, so is $f \cdot g$.

```haskell
newtype Compose f g a = Compose { getCompose :: f (g a) }

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)

instance (Applicative f, Applicative g) => Applicative (Compose f g) where
  -- This is a nice exercise ;)
```
Monads with join

A monad can be defined via two sets of functions:

\[ \text{return} :: a \rightarrow m \ a \]

-- Choose one from the following:
\[ (\ggg) :: m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \]
\[ \text{join} :: m (m \ a) \rightarrow m \ a \]

Those descriptions are interchangeable:

\[ \text{join } m = m \ggg \text{id} \]
\[ xs \ggg f = \text{join} (\text{fmap } f \ xs) \]
Combining monads

Monads, however, are not closed under such compositions.

```haskell
instance (Monad f, Monad g) => Monad (Compose f g) where
  return x = Compose (return (return x))
  join (Compose (Compose x)) = -- ??
```

Intuitively, we want to build a function:

\[ f \circ (g \circ f \circ g) \rightarrow f \circ g \]

But we can only perform that join if we had a way to turn:

\[ f \circ (g \circ f \circ g) \rightarrow f \circ f \circ g \circ g \]
Can we define some other way to compose monads?
“List of successes” parsers

We have seen (applicative) parsers – but what about their monadic interface?

newtype Parser s a =
  Parser { runParser :: [s] -> [(a,[s])] } 

Question

How can we define a monad instance for such parsers?
instance Monad (Parser s) where
    return x = Parser (\xs -> [(x, xs)])
    p >>= f =
        Parser (\xs -> do (r, ys) <- runParser p xs
                   runParser (f r) ys)

This combines both the state and list monads that we saw previously.

Question

From which instance is the >>= which is used in the do-expression taken?
instance Monad (Parser s) where
  return x = Parser (\xs -> [(x,xs)])
  p >>= f =
        Parser (\xs -> do (r,ys) <- runParser p xs
                     runParser (f r) ys)

This combines both the state and list monads that we saw previously.

Question

From which instance is the >>= which is used in the do-expression taken?

Answer: instance Monad []
Monad transformers

We can actually assemble the parser monad from two blocks: a list monad, and a state monad transformer.

```haskell
newtype Parser s a = Parser { runParser :: [s] -> [(a, [s])] }

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

Modulo wrapper types StateT [s] [] a is the same as [s] -> [(a, [s])].

**Question**

What is the kind of StateT?
Monad transformers (contd.)

```haskell
instance (Monad m) => Monad (StateT s m) where
    return a = StateT (\s -> return (a, s))
    m >>= f = StateT (\s -> do (a, s') <- runStateT m s
                         runStateT (f a) s')
```

The instance definition is using the underlying monad \( m \) in the \texttt{do}-expression.
Monad transformers (contd.)

For (nearly) any monad, we can define a corresponding monad transformer, for instance:

```
newtype ListT m a = ListT { runListT :: m [a] }
```

Monad transformers (contd.)

For (nearly) any monad, we can define a corresponding monad transformer, for instance:

```haskell
newtype ListT m a = ListT { runListT :: m [a] }

instance (Monad m) => Monad (ListT m) where
    return a = ListT (return [a])
    m >>= f = ListT (do as <- runListT m
                       bss <- mapM (runListT . f) as
                       return (concat bbs))
```
Question:

Is ListT (State s) the same as StateT s []?
Order matters!

StateT s [] a
is
s -> [(a, s)]
whereas
ListT (State s) a
is
s -> ([a], s)

- Different orders of applying monads and monad transformers create subtly different monads!
- In the former monad, the new state depends on the result we select. In the latter, it doesn’t.
Building blocks

- In order to see how to assemble monads from special-purpose monads, let us first learn about more monads than Maybe, State, List and IO.
- The place in the standard libraries for monads is Control.Monad.*.
- The state monad is available in Control.Monad.State.
- The list monad is available in Control.Monad.List.
Except or Either

The Except monad is a variant of Maybe which is slightly more useful for actually handling exceptions:

```haskell
instance Monad (Either e) where
    return x = Right x
    (Left e) >>= _ = Left e
    (Right r) >>= k = k r

throwE :: e -> Either e a
throwE = Left

catchE :: Either e a -> (e -> Either e' a) -> Either e' a
catchE (Left e) h = h e
catchE (Right v) _ = Right v
```
Except versus Error

Previous versions of the monad transformers library defined a slightly different variation:

```haskell
class Error e where
  noMsg :: e -> m a
  strMsg :: String -> e

instance Error e => Monad (Either e) where
  -- return and (>>=) as before
  fail msg = Left (strMsg msg)
```

This version is now deprecated.
Deprecation of MonadFail

As of GHC 8.0, a new subclass of Monad was introduced.
   ▶ The plan is to remove fail from Monad in GHC 8.6.

```haskell
class Monad m => MonadFail m where
  fail :: String -> m a
```

Why was fail in Monad in the first place?
   ▶ Failure of pattern matching.

```haskell
do ...
  ...
  Just v <- foo bar
  ...
  foo bar >>= \e -> case e of Just v -> ...
  _ -> fail "..."
```
Like State, the Except monad has an interface, such that we can throw and catch exceptions without requiring a specific underlying datatype:

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

instance MonadError e (Either e)
```

The constraint `m -> e` in the class declaration is a *functional dependency*. It places certain restrictions on the instances that can be defined for that class.
Excursion: functional dependencies

- Type classes are *open relations* on types.
- Each single-parameter type class implicitly defines the set of types belonging to that type class.
- Instance definitions corresponds to membership.
- There is no need to restrict type classes to only one parameter.
- All parameters can also have different kinds.
Excursion: functional dependencies (contd.)

- Using a type class in a polymorphic context can lead to an *unresolved overloading* error:

  ```haskell
  show . read
  ```

  What instance of `show` and `read` should be used?
Excursion: functional dependencies

- Multiple parameters lead to more unresolved overloading:

```haskell
someComputation :: Either String Int
fallback :: Int
catchError someComputation (const (return fallback))
:: (MonadError e (Either String)) => Either String Int```

The ‘handler’ doesn’t give any information about what the type of the errors is.
Excursion: functional dependencies (contd.)

- A functional dependency (inspired by relational databases) prevents such unresolved overloading.
- The dependency $m \rightarrow e$ indicates that $e$ is uniquely determined by $m$. The compiler can then automatically reduce a constraint such as

$$(\text{MonadError} \ e \ (\text{Either} \ \text{String})) \Rightarrow ...$$

using

```haskell
instance MonadError e (Either e)
```

- Instance declarations that violate the functional dependency are rejected.
ExceptT monad transformer

Of course, there also is a monad transformer for errors:

\[
\text{newtype } \text{ExceptT} \ e \ m \ a = \\
\text{ExceptT} \ \{ \ \text{runErrorT} :: m (\text{Either} \ e \ a) \ \}\n\]

\[
\text{instance } \text{Monad} \ m \Rightarrow \text{Monad} (\text{ExceptT} \ e \ m)
\]

New combinations are possible.

Even multiple transformers can be applied
Examples

ExceptT e (StateT s IO) a  -- is the same as
StateT s IO (Either e a)   -- is the same as
s -> IO (Either e a, s)

StateT s (ExceptT e IO) a  -- is the same as
s -> ExceptT e IO (a, s)   -- is the same as
s -> IO (Either e (a, s))

Question

Does an exception change the state or not? Can the resulting monad use get, put, throwError, catchError?
Defining interfaces

Many monads can have a state-like interface, hence we define:

```haskell
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
  get = state (\s -> (s, s))
  state :: (s -> (a, s)) -> m a
  put s = state (\_ -> ((), s))

  state f = do s <- get
             let ~(a, s') = f s
             put s'
             return a
```
Using interfaces

With MonadError, MonadState and so on you can write functions which do not depend on the concrete monad transformer you are using.

```haskell
f :: (MonadError String m, MonadState Int m) => m Int
f = do i <- get
  if i < 0
    then throwError "Invalid number"
  else return (i + 1)
```

The concrete stack is fixed when “running” the monad.

```haskell
runExcept (runStateT f)
runState 0 (runExceptT f)
```
Using interfaces

With `MonadError`, `MonadState` and so on you can write functions which do not depend on the concrete monad transformer you are using.

```haskell
f :: (MonadError String m, MonadState Int m) => m Int
f = do i <- get
  if i < 0
    then throwError "Invalid number"
  else return (i + 1)
```

The concrete stack is fixed when “running” the monad.

```haskell
runExcept (runStateT 0 f)
runState 0 (runExceptT f)
```
Lifting takes an operation from a smaller to a larger stack, in a generic fashion.

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

```haskell
instance MonadTrans (ExceptT e) where
  lift m = ExceptT (do a <- m return (Right a))
```

```haskell
instance MonadTrans (StateT s) where
  lift m = StateT (\ s -> do a <- m return (a, s))
```
Lifting takes an operation from a smaller to a larger stack, in a generic fashion.

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

instance MonadTrans (ExceptT e) where
    lift m = ExceptT (do a <- m
                        return (Right a))

instance MonadTrans (StateT s) where
    lift m = StateT (\ s -> do a <- m
                      return (a, s))
```
Lifting

lift lets you define MonadThingy instances more easily.

```haskell
instance MonadState s m => MonadState s (ExceptT e m) where
  get = lift get
  put = lift . put

instance MonadError e m => MonadError e (StateT s m) where
  throwError = lift . throwError
  catchError = ?? -- We return to this later
```
Monad, transformer, interface

For each monad Thingy,

- In package transformers:
  - The base monad Thingy a with its Monad instance.
  - The transformer version ThingyT m a with its MonadTrans instance.
  - Run functions runThingy and runThingyT to “escape” the monad.

- In package mtl:
  - The interface as a type class MonadThingy m with instances for all transformers.
  - Instances for ThingyT m of all other MonadX classes.
Question

How many instances are required?
A tour of Haskell’s monads
The reader monad propagates some information, but unlike a state monad does not thread it through subsequent actions.

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
  return x = Reader (\r -> a)
  m >>= f = Reader (\r -> runReader (f (runReader m r)) r)
```
We can also capture the interface of the operations that the reader monad supports:

```haskell
instance (Monad m) =>
    MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```
The writer monad collects some information, but it is not possible to access the information already collected in previous computations.

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }
```

To collect information, we have to know

- what an empty piece of information is, and
- how to combine two pieces of information.

A typical example is a list of things (\([\ ]\) and (\(++)\)), but the library generalizes this to any *monoid*.
Monoids

Monoids are algebraic structures (defined in Data.Monoid) with a neutral element and an associative binary operation:

class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty

instance Monoid [a] where
  mempty = []
  mappend = (++)

...and many more! Note the similarity to monads!
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    m >>= f = Writer $
        let (a, w) = runWriter m
        (b, w') = runWriter (f a)
        in (b, w `mappend` w')
Writer Interface

class (Monoid w, Monad m) =>
  MonadWriter w m | m -> w where
  tell :: w -> m ()
  listen :: m a -> m (a, w)
  pass :: m (a, w -> w) -> m a
The continuation monad allows to capture the current continuation and jump to it when desired.

```haskell
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

**Question**

How is this a monad?
The continuation monad allows to capture the current continuation and jump to it when desired.

```haskell
newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```

**Question**

How is this a monad?

```haskell
instance Monad (Cont r) where
  return a = Cont (\ k -> k a)
  m >>= f = Cont (\ k -> runCont m (\ a -> runCont (f a) k))
```
Identity

The identity monad has no effects.

newtype Identity a =
    Identity { runIdentity :: a }

Question:
How is this a monad?
The identity monad has no effects.

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

Question:
How is this a monad?

instance Monad Identity where
  return x = Identity x
  m >>= f = Identity (f (runIdentity m))
```
Identity as base monad

The identity monad allows us to recover the usual monads from the transformers.

```hs
type Except e = ExceptT e Identity
type State s = StateT s Identity
type Reader r = ReaderT r Identity
type Writer w = WriterT w Identity
...
type Thingy = ThingyT Identity
```

In fact, this is how they are defined in transformers.
There is no transformer version of `IO`, so it is commonly used as base monad along with `Identity`.

`MonadIO` defines how to lift `IO` actions for your monad.

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```
MonadPlus adds a notion of failure and choice.

- Less powerful than MonadError, which has catch.
- Usually with a “monoidal” structure.
  - Although some laws are controversial.

```haskell
class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

  msum :: MonadPlus m => [m a] -> m a
  guard :: MonadPlus m => Bool -> m ()
```
MonadPlus (contd.)

instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` ys = ys
  xs      `mplus` ys = xs

instance Monoid e => MonadPlus (Either e) where
  mzero = Left mempty
  (Right x) `mplus` _               = Right x
  (Left  x) `mplus` (Right y)  = Right y
  (Left  x) `mplus` (Left   y) = Left (x <> y)
A monad for your application

It’s common to have a newtype defined for the specific monadic stack in your application.

```haskell
newtype App a = { runApp :: ReaderT Conf (ExceptT Errs IO) a }

Alas, this means that you need to reimplement MonadReader, MonadExcept and MonadIO.
A monad for your application

Not really, you can derive it automatically!

{-# language GeneralizedNewtypeDeriving #-}

newtype App a
  = { runApp :: ReaderT Conf (ExceptT Errs IO) a }
  deriving ( Functor, Applicative, Monad
            , MonadReader Conf, MonadError Errs
            , MonadIO )
Advanced lifting
Lifting more complex functions

Remember that we were trying to write this instance:

```haskell
instance MonadError e m => MonadError e (StateT s m) where
    throwError = lift . throw
    catchError = ??
```

Could we do it using `lift` from `MonadTrans`?
No, MonadTrans is not enough

The problem is this is the type we want to get:

```
catchError :: MonadError e m
dl lift :: m a -> StateT s m a
catchError :: MonadError e m
dl => StateT s m a -> (e -> StateT s m a)
dl => StateT s m a
```

but we only have:

```

catchError :: MonadError e m

dl => m a -> (e -> m a) -> m a
```
The trick is to realize that if we have a `liftCatch`:

```haskell
liftCatch :: MonadError e m
           => m (a, s) -> (e -> m (a, s))
           -> m (a, s)
```

Then the state gets injected and can be retrieved at the end.

```haskell
catchError = liftCatch catchError
```

We are “wrapping” and “unwrapping” the monad.

- This is how it is implemented in `mtl`.
Transformers with control operations

`monad-control` includes `MonadBaseControl` to handle these cases generically.

The core of what we need is the following function:

```haskell
control :: MonadBaseControl b m => (RunInBase m b -> b (StM m a)) -> m a
```

Details are quite convoluted because of the use of type families.
State transformer with control operations

In the case of StateT, the control operation reads:

\[
\text{control} :: ( (\forall a. \text{StateT} \ s \ m \ a \rightarrow m \ (a, s)) \\
\quad \rightarrow m \ (a, s) ) \\
\quad \rightarrow \text{StateT} \ s \ m \ a
\]

The type is complicated, but after careful read:

- You need to provide a function which “executes”.
- It takes as argument a function which “unwraps”.
- The end result is “wrapped” at the end.

\[
\text{catchError} \ v \ h = \text{control} \ \$ \ \text{\run} \rightarrow \\
\quad \text{catchError} \ (\text{run} \ v) \ (\text{run} \ . \ h)
\]
More about monad-control

The library has built a small ecosystem around it:

- Base libraries which are exported as lifted.
- lifted-base, lifted-async

Warning! monad-control is tricky to use:

- Computations might be arbitrarily duplicated or forgotten, so you need extra care.
- This affects severely the "stateful" monads.
- There are some proposals for "stateless" monads, like monad-unlift.
More about `monad-control`

The library has built a small ecosystem around it:

- Base libraries which are exported as lifted.
  - `lifted-base`, `lifted-async`

Warning! `monad-control` is tricky to use:

- Computations might be arbitrarily duplicated or forgotten, so you need extra care.
- This affects severely the “stateful” monads.
- There are some proposals for “stateless” monads, like `monad-unlift`. 
Generalizing stacks

Suppose you have an action of type:

\[ s :: \text{State} \ \text{Int} \ () \]

But now you want to use it within a stack. Could we generalize its type automatically?

\[ \text{magic } s :: \text{StateT} \ \text{Int} \ m () \]
Generalizing stacks with \texttt{mmorph}

The \texttt{mmorph} library provides a way to lift a \textit{monad morphism} to arbitrary monad stacks:

\texttt{hoist :: Monad m => (forall a. m a \rightarrow n a) \rightarrow t m b \rightarrow t n b}

In our case, we need to instantiate as:

\begin{align*}
\texttt{t m b} &= \texttt{State Int ()} = \texttt{StateT Int Identity ()} \\
\texttt{t n b} &= \texttt{StateT Int m ()}
\end{align*}
Let’s find a function `Identity a -> m a`:

\[
generalize :: \text{Identity} \ a \rightarrow m \ a
\]

\[
generalize (\text{Identity} \ a) = \text{return} \ a
\]

As a result, we have our magic function!

\[
\text{magic} = \text{hoist} \ \text{generalize} \\
\text{-- Given } s :: \text{State} \ \text{Int} \ () \\
\text{magic} \ s :: \text{StateT} \ \text{Int} \ m \ ()
\]
Mixing arbitrary stacks

By combining the previous functions you can insert layers in a transformer stack:

- lift inserts a new layer.
- hoist “goes down one layer” to apply a transformation.
- generalize changes the base monad from Identity to an arbitrary stack.
Mixing arbitrary stacks

By combining the previous functions you can insert layers in a transformer stack:

- **lift** inserts a new layer.
- **hoist** “goes down one layer” to apply a transformation.
- **generalize** changes the base monad from Identity to an arbitrary stack.

**mtl-style type classes leave the stack open.**

- No need to manipulate layers with these functions.
Algebraic effects

An alternative which has gained some interest recently.

- Many implementations, I show extensible-effects.

-- The actions look almost the same as mtl
f :: (Member (Exc String) r, Member (State Int) r) => Eff r Int
f = do i <- get
  if i < 0
    then throwExc "Invalid number"
  else return (i + 1)

-- No distinction between 'run' and 'runT'
runExc (runState 0 f)
-- Compare with mtl
runExcept (runStateT 0 f)
Algebraic effects

In the inside, algebraic effects are quite different from monad transformers.

*Core idea:* separate syntax from semantics.
  - First assemble what needs to be done.
  - Then use handlers to perform the operations

*Big advantage:* more modularity.
  - No need to write $n^2$ instances of MonadThingys.
Recap: Monad transformers

Monad transformers allow you to assemble complex monads in a structured fashion.

The do **not** commute.

Lifting various operations through stacks of monad transformers can be cumbersome.

We use various monadic operations (such as `get` or `throw`) and only later decide on the order that we want to stack the corresponding monad transformers.
Summary

▶ Common interfaces are extremely powerful and give you a huge amount of predefined theory and functions.
▶ Look for common interfaces in your programs.
▶ Recognise monads and applicative functors.
▶ Define or assemble your own monads.
▶ Add new features to the monads you are using.
▶ Monads and applicative functors make Haskell particularly suited for Embedded Domain Specific Languages.