Getting the Most Out of Monad Transformers

George Wilson

Ephox

george.wilson@ephox.com

June 27, 2015

Motivation

We need to write programs that

- pass configuration
- handle errors
- maintain some kind of state
- perform IO
- write logs
- ...

Motivation

We need to write programs that

- **pass configuration**
- **handle errors**
- maintain some kind of state
- **perform IO**
- write logs
- ...

Goals

- We want compositionality

Goals

- We want compositionality
- We want types to help us

Warning: Lies

TRANSFORMERS

Transformer usage in apps

Let’s focus on two things for a moment:
Configuration and error handling

Config

```
data DbConfig =
  DbConfig {
    dbConn  :: DbConnection
  , schema  :: Schema
  }
```

Config

```haskell
data DbConfig =
  DbConfig {
    dbConn :: DbConnection
  , schema :: Schema
  }

data NetworkConfig =
  NetConfig {
    port :: Port
  , ssl  :: Ssl
  }
```

Config

```
data DbConfig =
  DbConfig {
    dbConn  :: DbConnection
  , schema  :: Schema
  }

data NetworkConfig =
  NetConfig {
    port  :: Port
  , ssl   :: Ssl
  }

data AppConfig =
  AppConfig {
    appDbConfig  :: DbConfig
  , appNetConfig  :: NetworkConfig
  }
```

Errors

```
data DbError =
    QueryError Text
  | InvalidConnection
```

Errors

```haskell
data DbError =
    QueryError Text
  | InvalidConnection

data NetworkError =
    Timeout Int
  | ServerOnFire
```

Errors

```
data DbError =
    QueryError Text
  | InvalidConnection

data NetworkError =
    Timeout Int
  | ServerOnFire

data AppError =
    AppDbError DbError
  | AppNetError NetworkError
```

A monad just for my application

```
newtype App a =
  App {
    unApp :: ReaderT AppConfig (ExceptT AppError IO) a
  } deriving (
    Functor,
    Applicative,
    Monad,
    MonadReader AppConfig,
    MonadError AppError,
    MonadIO
  )
```

A monad just for my application

```
newtype App a =
  App {
    unApp :: ReaderT AppConfig (ExceptT AppError IO) a
  } deriving (
    Functor,
    Applicative,
    Monad,
    MonadReader AppConfig,
    MonadError AppError,
    MonadIO
  )
```

…How do we use this thing?

MTL REFRESHER

mtl

Idea: Associate with each transformer a typeclass specifying its operations

Reader

```
class Monad m => MonadReader r m | m -> r where
```

Reader

```
class Monad m => MonadReader r m | m -> r where

    -- Retrieves the monad environment.
    ask   :: m r
    ask = reader id
```

Reader

```
class Monad m => MonadReader r m | m -> r where

    -- Retrieves the monad environment.
    ask   :: m r
    ask = reader id

    -- Retrieves a function of the current environment.
    reader :: (r -> a)
           -> m a
    reader f = do
      r <- ask
      return (f r)
```

MonadReader Example

```
getPort :: MonadReader NetworkConfig m
        => m Port
getPort = reader port
```

MonadReader Example

```
getPort :: MonadReader NetConfig m
        => m Port
getPort =
  do cfg <- ask
     return (port cfg)
```

MonadIO

```
class (Monad m) => MonadIO m where
    -- Lift a computation from the 'IO' monad.
    liftIO :: IO a -> m a
```

MonadIO Example

```
printM :: MonadIO m
       => String -> m ()
printM s = liftIO (putStrLn s)
```

Errors

```
class (Monad m) => MonadError e m | m -> e where
```

Errors

```
class (Monad m) => MonadError e m | m -> e where

    -- Is used within a monadic computation to
    -- begin exception processing.
    throwError :: e -> m a
```

Errors

```
class (Monad m) => MonadError e m | m -> e where

    -- Is used within a monadic computation to
    -- begin exception processing.
    throwError :: e -> m a

    -- A handler function to handle previous errors
    -- and return to normal execution.
    catchError :: m a -> (e -> m a) -> m a
```

MonadError Example

```haskell
mightFail :: MonadError Err m
          => m Int

couldFail :: MonadError Err m
          => m String
```

MonadError Example

```
mightFail :: MonadError Err m
          => m Int

couldFail :: MonadError Err m
          => m String

maybeFail :: MonadError Err m
          => m (Maybe (Int,String))
maybeFail =
  ( do a <- mightFail
       b <- couldFail
       pure (Just (a,b))
  ) `catchError` (\err -> pure Nothing)
```

Vocabulary

```
-- instance MonadReader AppConfig App
ask :: App AppConfig
```

Vocabulary

```
-- instance MonadReader AppConfig App
ask :: App AppConfig

-- instance MonadError AppError App
throwError :: AppError -> App a
catchError :: App a -> (AppError -> App a) -> App a
```

Vocabulary

```
-- instance MonadReader AppConfig App
ask :: App AppConfig

-- instance MonadError AppError App
throwError :: AppError -> App a
catchError :: App a -> (AppError -> App a) -> App a

-- instance MonadIO App
liftIO :: IO a -> App a
```

Yet Another Example

```
loadFromDb :: App MyData

sendOverNet :: MyData -> App ()

loadAndSend :: App ()
loadAndSend = loadFromDb >>= sendOverNet
```

Not Good Enough!

```
loadFromDb :: App MyData

sendOverNet :: MyData -> App ()

loadAndSend :: App ()
loadAndSend = loadFromDb >>= sendOverNet
```

Generalise Everything

```
loadFromDb :: (MonadReader DbConfig m,
               MonadError  DbError  m,
               MonadIO m)
           => m MyData
```

Generalise Everything

```
loadFromDb :: (MonadReader DbConfig m,
               MonadError  DbError  m,
               MonadIO m)
           => m MyData

sendOverNet :: (MonadReader NetworkConfig m,
                MonadError  NetworkError  m,
                MonadIO m)
            => MyData -> m ()
```

Generalise Everything

```haskell
loadFromDb :: (MonadReader DbConfig m,
               MonadError  DbError  m,
               MonadIO m)
           => m MyData

sendOverNet :: (MonadReader NetworkConfig m,
                MonadError  NetworkError  m,
                MonadIO m)
            => MyData -> m ()

loadAndSend = loadFromDb >>= sendOverNet
```

Oops

```
Couldn’t match type NetworkConfig with DbConfig
arising from a functional dependency between constraints:
  MonadReader DbConfig m
    arising from a use of loadFromDb at P.hs:447:15-24
  MonadReader NetworkConfig m
    arising from a use of sendOverNet at P.hs:447:30-40
In the first argument of (>>=), namely loadFromDb
In the expression: loadFromDb >>= sendOverNet
In an equation for loadAndSend:
    loadAndSend = loadFromDb >>= sendOverNet
```

Optics

Optics?

Optics come from `lens` on Hackage.
We're going to talk about **lenses** and **prisms** today.
There are many more optics in `lens`!

What is a lens?

A lens is a getter-setter

It lets us get at one part of a whole

```
-- basic lens usage
view :: Lens source target
     -> (source -> target) -- getter

set  :: Lens source target
     -> target -> source -> source -- setter
```

What is a lens?

A lens is a getter-setter

It lets us get at one part of a whole

```
-- basic lens usage
view :: Lens source target
    -> (source -> target) -- getter

set  :: Lens source target
    -> target -> source -> source -- setter

-- Construct a lens
lens :: (source -> target) -- getter
    -> (source -> target -> source) --setter
    -> Lens source target
```

Lenses compose!

```
(.)  :: Lens s t -> Lens t u -> Lens s u
id   :: Lens a a
```

Lens examples

```
_1  ::  Lens  (a,b)  a
_2  ::  Lens  (a,b)  b
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d

 > view _1 (("hello", Nothing), 3)

("hello", Nothing)
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d

 > view _1 (("hello", Nothing), 3)

("hello", Nothing)

 > view (_1 . _1) (("hello", Nothing), 3)

"hello"
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d

 > set _2 1000 (("hello", Nothing), 3)

(("hello", Nothing), 1000)
```

Lens examples

```
_1 :: Lens (a,b) a
_2 :: Lens (a,b) b

(_1 . _2) :: Lens ((c,d),e) d

 > set _2 1000 (("hello", Nothing), 3)

(("hello", Nothing), 1000)

 > set (_1 . _2) (Just "lens") (("hello", Nothing), 3)

(("hello", Just "lens"), 3)
```

What is a prism?

A prism is like a first-class pattern match
It lets us get at one branch of an ADT

```
-- basic prism usage
preview :: Prism a b
        -> (a -> Maybe b) -- partial getter

review  :: Prism a b
        -> (b -> a) -- constructor
```

What is a prism?

A prism is like a first-class pattern match
It lets us get at one branch of an ADT

```
-- basic prism usage
preview :: Prism a b
        -> (a -> Maybe b) -- partial getter

review  :: Prism a b
        -> (b -> a) -- constructor

-- construct a prism
prism :: (target -> source)
      -> (source -> Maybe target)
      -> Prism source target
```

Prisms compose!

```
(.)  ::  Prism  s  t -> Prism  t  u -> Prism  s  u
id   ::  Prism  a  a
```

Prism examples

```
_Left  :: Prism (Either a b) a
_Right :: Prism (Either a b) b
```

Prism examples

```
_Left   :: Prism (Either a b) a
_Right  :: Prism (Either a b) b

_Just     :: Prism (Maybe a) a
_Nothing  :: Prism (Maybe a) ()
```

Prism examples

```
_Left  :: Prism (Either a b) a
_Right :: Prism (Either a b) b

_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()

> preview _Left (Left (Just 4))

(Just (Just 4))
```

Prism examples

```
_Left  :: Prism (Either a b) a
_Right :: Prism (Either a b) b

_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()

> preview _Left (Left (Just 4))

(Just (Just 4))

> preview (_Left . _Just) (Left (Just 4))

Just 4
```

```
_Left  :: Prism (Either a b) a
_Right :: Prism (Either a b) b

_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()

 > preview _Right (Left (Just 4))

Nothing
```

```
_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()
```

```
> review (_Right . _Just) ”hello”
```

```
Right (Just ”hello”)
```

```
_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()
```

```
> review (_Right . _Just) "hello"

Right (Just "hello")
```

```
> review (_Just . _Left) 42

Just (Left 42)
```

Classy Optics

Classy optics?

Idea: Associate with each type a typeclass full of optics for that type

Classy Lenses

```
data DbConfig =
  DbConfig {
    _dbConn :: DbConnection
  , _schema :: Schema
  }
```

Classy Lenses

```
data DbConfig =
  DbConfig {
    _dbConn :: DbConnection
  , _schema :: Schema
  }

class HasDbConfig t where
  dbConfig :: Lens t DbConfig
  dbConn   :: Lens t DbConnection
  dbSchema :: Lens t Schema
```

Classy Lenses

```haskell
data DbConfig =
  DbConfig {
    _dbConn :: DbConnection
  , _schema :: Schema
  }

class HasDbConfig t where
  dbConfig :: Lens t DbConfig
  dbConn   :: Lens t DbConnection
  dbSchema :: Lens t Schema

instance HasDbConfig DbConfig where
  dbConfig = id
  dbConn   =
    lens _dbConn (\d c -> d { _dbConn = c })
  dbSchema =
    lens _dbSchema (\d s -> d { _dbSchema = s})
```

Classy Lenses

```
data DbConfig =
  DbConfig {
    _dbConn :: DbConnection
  , _dbSchema :: Schema
  }

class HasDbConfig t where
  dbConfig :: Lens t DbConfig
  dbConn   :: Lens t DbConnection
  dbSchema :: Lens t Schema

  dbConn = dbConfig . dbConn
  dbSchema = dbConfig . dbSchema

instance HasDbConfig DbConfig where
  dbConfig = id
  dbConn   =
    lens _dbConn (\d c -> d { _dbConn = c })
  dbSchema =
    lens _dbSchema (\d s -> d { _dbSchema = s})
```

Classy Lenses

```
data NetworkConfig =
  NetConfig {
    _port :: Port
  , _ssl  :: Ssl
  }
```

Classy Lenses

```
data NetworkConfig =
  NetConfig {
    _port :: Port
  , _ssl  :: Ssl
  }

class HasNetworkConfig t where
  netConfig :: Lens t NetworkConfig
  netPort   :: Lens t Port
  netSsl    :: Lens t Ssl
```

Classy Lenses

```haskell
data NetworkConfig =
  NetConfig {
    _port :: Port
  , _ssl  :: Ssl
  }

class HasNetworkConfig t where
  netConfig :: Lens t NetworkConfig
  netPort   :: Lens t Port
  netSsl    :: Lens t Ssl

instance HasNetworkConfig NetworkConfig where
  netConfig = id
  netPort   =
    lens _port (\n p -> n { _port = p })
  netSsl    =
    lens _ssl (\n s -> n { _ssl = s})
```

Classy Lenses

```haskell
data NetworkConfig =
  NetConfig {
    _port :: Port
  , _ssl  :: Ssl
  }

class HasNetworkConfig t where
  netConfig :: Lens t NetworkConfig
  netPort   :: Lens t Port
  netSsl    :: Lens t Ssl

  netPort = netConfig . netPort
  netSsl  = netConfig . netSsl

instance HasNetworkConfig NetworkConfig where
  netConfig = id
  netPort   =
    lens _port (\n p -> n { _port = p })
  netSsl    =
    lens _ssl (\n s -> n { _ssl = s})
```

Classy Lenses

```
data AppConfig =
  AppConfig {
    appDbConfig :: DbConfig
  , appNetConfig :: NetworkConfig
  }
```

Classy Lenses

```haskell
data AppConfig =
  AppConfig {
    appDbConfig :: DbConfig
  , appNetConfig :: NetworkConfig
  }

instance HasDbConfig AppConfig where
  dbConfig =
    lens appDbConfig
      (\app db -> app { appDbConfig = db })
```

Classy Lenses

```
data AppConfig =
  AppConfig {
    appDbConfig :: DbConfig
  , appNetConfig :: NetworkConfig
  }

instance HasDbConfig AppConfig where
  dbConfig =
    lens appDbConfig
      (\app db -> app { appDbConfig = db })

instance HasNetworkConfig AppConfig where
  netConfig =
    lens appNetConfig
      (\app net -> app { appNetConfig = net })
```

Classy Prisms

```
data DbError =
    QueryError Text
  | InvalidConnection
```

Classy Prisms

```
data DbError =
    QueryError Text
  | InvalidConnection

class AsDbError t where
  _DbError       :: Prism t DbError
  _QueryError    :: Prism t Text
  _InvalidConn   :: Prism t ()
```

Classy Prisms

```haskell
data DbError =
    QueryError Text
  | InvalidConnection

class AsDbError t where
  _DbError      :: Prism t DbError
  _QueryError   :: Prism t Text
  _InvalidConn  :: Prism t ()

instance AsDbError DbError where
  _DbError = id
  _QueryError =
    prism QueryError $ \case QueryError t -> Just t
                             _            -> Nothing
  _InvalidConn =
    prism InvalidConnection $
      \case InvalidConnection -> Just ()
            _                 -> Nothing
```

Classy Prisms

```
data DbError =
    QueryError Text
  | InvalidConnection

class AsDbError t where
  _DbError     :: Prism t DbError
  _QueryError  :: Prism t Text
  _InvalidConn :: Prism t ()

  _QueryError = _DbError . _QueryError
  _InvalidConn = _DbError . _InvalidConn

instance AsDbError DbError where
  _DbError = id
  _QueryError =
    prism QueryError
      $ \case
          QueryError t -> Just t
          _            -> Nothing
  _InvalidConn =
    prism (const InvalidConnection)
```

Classy Prisms

```
data NetworkError =
    Timeout Int
  | ServerOnFire
```

Classy Prisms

```haskell
data NetworkError =
    Timeout Int
  | ServerOnFire

class AsNetworkError t where
  _NetworkError :: Prism t NetworkError
  _Timeout      :: Prism t Int
  _ServerOnFire :: Prism t ()
```

Classy Prisms

```haskell
data NetworkError =
    Timeout Int
  | ServerOnFire

class AsNetworkError t where
  _NetworkError :: Prism t NetworkError
  _Timeout      :: Prism t Int
  _ServerOnFire :: Prism t ()

instance AsNetworkError NetworkError where
  _NetworkError = id
  _Timeout = prism Timeout $ \case Timeout t -> Just t
                                   _         -> Nothing
  _ServerOnFire =
    prism (const ServerOnFire)
      $ \case ServerOnFire -> Just ()
              _            -> Nothing
```

Classy Prisms

```haskell
data NetworkError =
    Timeout Int
  | ServerOnFire

class AsNetworkError t where
  _NetworkError :: Prism t NetworkError
  _Timeout      :: Prism t Int
  _ServerOnFire :: Prism t ()

  _Timeout = _NetworkError . _Timeout
  _ServerOnFire = _NetworkError . _ServerOnFire

instance AsNetworkError NetworkError where
  _NetworkError = id
  _Timeout = prism Timeout $ \case Timeout t -> Just t
                                   _         -> Nothing
  _ServerOnFire =
    prism (const ServerOnFire)
      $ \case ServerOnFire -> Just ()
              _            -> Nothing
```

Classy Prisms

```
data AppError =
    AppDbError { dbError :: DbError }
  | AppNetError { netError :: NetworkError }
```

Classy Prisms

```haskell
data AppError =
    AppDbError { dbError :: DbError }
  | AppNetError { netError :: NetworkError }

instance AsDbError AppError where
  _DbError =
    prism AppDbError
      $ \case AppDbError dbe -> Just dbe
              _              -> Nothing
```

Classy Prisms

```
data AppError =
    AppDbError { dbError :: DbError }
  | AppNetError { netError :: NetworkError }

instance AsDbError AppError where
  _DbError =
    prism AppDbError
      $ \case AppDbError dbe -> Just dbe
              _              -> Nothing

instance AsNetworkError AppError where
  _NetworkError =
    prism AppNetError
      $ \case AppNetError ne -> Just ne
              _              -> Nothing
```

If that's too much typing…

```
class AsNetworkError t where
  _NetworkError :: Prism t NetworkError
  _Timeout      :: Prism t Int
  _ServerOnFire :: Prism t ()
  _Timeout = _NetworkError . _Timeout
  _ServerOnFire = _NetworkError . _ServerOnFire

instance AsNetworkError NetworkError where
  _NetworkError = id
  _Timeout = prism Timeout $ \case Timeout t -> Just t
                                   _         -> Nothing
  _ServerOnFire =
    prism (const ServerOnFire)
      $ \case ServerOnFire -> Just ()
              _            -> Nothing
```

Template Haskell!

```
makeClassyPrisms ''NetworkError
```

If this is too much typing…

```haskell
class HasDbConfig t where
  dbConfig :: Lens t DbConfig
  dbConn   :: Lens t DbConnection
  dbSchema :: Lens t Schema
  dbConn = dbConfig . dbConn
  dbSchema = dbConfig . dbSchema

instance HasDbConfig DbConfig where
  dbConfig = id
  dbConn   =
    lens _dbConn (\d c -> d { _dbConn = c })
  dbSchema =
    lens _dbSchema (\d s -> d { _dbSchema = s})
```

Template Haskell again!

```
makeClassy ''DbConfig
```

PUTTING IT ALL TOGETHER

The Pay-off

```
loadFromDb :: (MonadError e m, MonadReader r m,
               AsDbError e,    HasDbConfig r,
               MonadIO m)
           => m MyData
```

The Pay-off

```
loadFromDb :: (MonadError e m, MonadReader r m,
               AsDbError e,    HasDbConfig r,
               MonadIO m)
           => m MyData

sendOverNet :: (MonadError e m, MonadReader r m,
                AsNetworkError e, HasNetworkConfig r,
                MonadIO m)
            => MyData -> m ()
```

The Pay-off

```haskell
loadFromDb :: (MonadError e m, MonadReader r m,
               AsDbError e,     HasDbConfig r,
               MonadIO m)
           => m MyData

sendOverNet :: (MonadError e m, MonadReader r m,
                AsNetworkError e, HasNetworkConfig r,
                MonadIO m)
            => MyData -> m ()

-- Finally!
loadAndSend = loadFromDb >>= sendOverNet
```

We’ve done it

```haskell
loadAndSend :: (MonadError e m, MonadReader r m,
                AsNetworkError e, HasDbConfig r,
                AsDbError e, HasNetworkConfig r,
                MonadIO m)
            => m ()
loadAndSend = loadFromDb >>= sendOverNet
```

We've done it

```haskell
loadAndSend :: (MonadError e m, MonadReader r m,
                AsNetworkError e, HasDbConfig r,
                AsDbError e, HasNetworkConfig r,
                MonadIO m)
            => m ()
loadAndSend = loadFromDb >>= sendOverNet

mainApp :: App ()
mainApp = loadAndSend
```

Wrapping up. . .

- Abstractions $>$ Concretions
- Typeclass constraints stack up better than monolithic transformers
- Lens gives us a compositional vocabulary for talking about data

THE END

THE END

. . . or is it?

References

- I talked about these things:
  hackage.haskell.org/package/mtl
  lens.github.io
- I encourage you to also look at these things:
  github.com/benkolera/talk-stacking-your-monads/
  hackage.haskell.org/package/hoist-error