Failing gracefully with EitherT

Ben Barnes

Data61 | CSIRO

This talk is about failure.

- The why and how of failure
- How we model it in Haskell
- Introducing EitherT
- Field guide to EitherT

Why do we treat failure as special?

How we think:

```
procedure:
  decode HTTP request
  build DB query
  run DB query
  return HTTP response

on failure:
  return relevant HTTP status
```

Reality:

```
decode request:
  on failure, return 400
  on success, build DB query:
    on failure, return 500
    on success, run DB query:
      on failure, return 500
      on success, return 200
```

How do we model failure?

Exceptions (primitive)

```
getPosts :: User -> IO [Post]
getPosts user = do
  id <- getUserId user
  getPostsById id

tryGetPosts :: User -> IO [Post]
tryGetPosts user =
  catch
    (getPosts user)
    (\err -> return [])
```

Maybe, Either (values)

```
getPosts :: User -> Either Err [Post]
getPosts user = do
  id <- lookupEither user userMap
  lookupEither id postsMap

tryGetPosts :: User -> [Post]
tryGetPosts user =
  case getPosts user of
    Right posts -> posts
    Left err    -> []
```

These aren't interchangeable, but...

When doing both IO and logic together, you have a choice.

Exceptions

Exceptions are a good thing.

- They are the reality of computer systems.
  - Interrupts
  - Signals
- They allow control over threads.

Simon Marlow & Simon Peyton Jones, *Asynchronous Exceptions in Haskell*
https://simonmar.github.io/bib/papers/async.pdf

Exceptions can be tricky.

```
try $ throwIO MyException          :: IO (Either MyException ())
try $ throw MyException            :: IO (Either MyException ())
try $ evaluate $ throw MyException :: IO (Either MyException ())
try $ return $! throw MyException  :: IO (Either MyException ())
try $ return $ throw MyException   :: IO (Either MyException ())

throw    :: (Exception e) => e -> a
throwIO  :: (Exception e) => e -> IO a
evaluate :: a -> IO a
```

Michael Snoyman, *Async Execption Handling in Haskell*, April 2018
https://www.youtube.com/watch?v=T5y8sFmCFnA

Exceptions can be tricky.

```
try $ throwIO MyException          -- Left MyException
try $ throw MyException            -- Left MyException
try $ evaluate $ throw MyException -- Left MyException
try $ return $! throw MyException  -- Left MyException
try $ return $ throw MyException   -- Right (throw MyException)

throw    :: (Exception e) => e -> a
throwIO  :: (Exception e) => e -> IO a
evaluate :: a -> IO a
```

Michael Snoyman, *Async Execption Handling in Haskell*, April 2018
https://www.youtube.com/watch?v=T5y8sFmCFnA

Exceptions can be tricky.

There are further subtleties when catching them.

More still when threads are involved.

safe-exceptions: Safe, consistent, and easy exception handling
https://hackage.haskell.org/package/safe-exceptions

They don't show up in type signatures.

```
-- This function uses exceptions to report failure. Which ones?
openFile :: FilePath -> IOMode -> IO Handle
```

Types and values

Types and values

- Model your errors using types.
- Make functions total using Either.
- Compose using Applicative, Monad...
- Handle errors by pattern matching.

Running example: normalising numbers

Input file:

```
1.0
2.0
3.0
```

Output file:

```
0.0
0.5
1.0
```

Model your errors using types.

```
data ParseError
  = InputRemaining Text -- fails on "3.0foo"
  | InvalidNumber Text  -- fails on "bar"
```

Model your errors using types.

```
data ParseError
  = InputRemaining Text -- fails on "3.0foo"
  | InvalidNumber Text  -- fails on "bar"

data LineError
  = InvalidLine LineNumber ParseError
```

Make functions total using Either.

```
parseDouble :: Text -> Either ParseError Double
parseDouble t =
  case Text.rational t of
    Right (a, "") -> Right a
    Right (a, t') -> Left $ InputRemaining t'
    Left _        -> Left $ InvalidNumber t
```

Changing error types.

```
parseLines :: Text -> Either LineError [Double]
parseLines t =
  let lines = Text.lines t
      lineNumbers = LineNumber <$> [1..]
      parseLine (n, line) = first (InvalidLine n) (parseDouble line)
  in  traverse parseLine (lineNumbers `zip` lines)

newtype LineNumber = LineNumber Int
```

What happens when we add IO?

```
data FileErrorType
  = AlreadyInUse
  | DoesNotExist
  | Full
  | PermissionError

data FileError
  = ReadFileError FilePath FileErrorType
  | WriteFileError FilePath FileErrorType
```

What happens when we add IO?

```
readFile:: FilePath -> IO (Either FileError Text)
readFile filePath =
  let getFileContents = do
        contents <- withFile filePath ReadMode Text.hGetContents
        return $ Right contents

      handleError e = do
        error <- selectFileError e
        return $ Left (ReadFileError filePath error)

  in  catch getFileContents handleError

selectFileError :: IOException -> IO FileErrorType
selectFileError e | isAlreadyInUseError e = return AlreadyInUse -- repeat...
                  | otherwise             = throwIO e
```

What happens when we add IO?

```
writeFile :: FilePath -> Text -> IO (Either FileError ())
```

Putting it all together:

```
main = do
  contentsEither <- readFile "input"
  case first AppFileError contentsEither of
    Left e -> Text.hPutStrLn stderr $ renderAppError e
    Right cs -> do
      let normalised = do
            nums <- first AppLineError $ parseLines cs
            first AppNormaliseError $ normalise nums
      case normalised of
        Left e  -> Text.hPutStrLn stderr $ renderAppError e
        Right ns -> do
          status <- writeFile "output" $ renderDoubles ns
          case first AppFileError status of
            Left e -> Text.hPutStrLn stderr $ renderAppError e
            Right u -> return u
```

Cleaning up

```haskell
doIOEither :: IO (Either e a) -> (a -> IO (Either e b)) -> IO (Either e b)
doIOEither ioEitherA f = do
  eitherA <- ioEitherA
  case eitherA of
    Left e -> return (Left e)
    Right a -> f a
```

Cleaning up

```
doIOEither :: IO (Either e a) -> (a -> IO (Either e b)) -> IO (Either e b)
doIOEither ioEitherA f = do
  eitherA <- ioEitherA
  case eitherA of
    Left e -> return (Left e)
    Right a -> f a

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Cleaning up

```
newtype IOEither e a = IOEither (IO (Either e a))

doIOEither :: IOEither e a -> (a -> IOEither e b) -> IOEither e b
doIOEither (IOEither ioEitherA) f = do
  eitherA <- ioEitherA
  case eitherA of
    Left e -> IOEither (return (Left e))
    Right a -> f a

instance Monad (IOEither e) where
  return a = IOEither (return (Right a))
  (>>=) = doIOEither
```

Presenting EitherT

```
newtype EitherT e m a = EitherT (m (Either e a))

bind :: (Monad m) => EitherT e m a -> (a -> EitherT e m b) -> EitherT e m b
bind (EitherT mEitherA) f = do
  eitherA <- mEitherA
  case eitherA of
    Left e -> EitherT (return (Left e))
    Right a -> f a

instance (Monad m) => Monad (EitherT e m) where
  return a = EitherT (return (Right a))
  (>>=) = bind
```

Field guide to EitherT

Hackage: transformers-either
Written by Tim McGilchrist

```
hoistEither :: (Monad m) => Either x a -> EitherT x m a

firstEitherT :: (Functor m) => (x -> y) -> EitherT x m a -> EitherT y m a

orDie :: (e -> Text) -> EitherT e IO a -> IO a
```

All together, with EitherT:

```
main :: IO ()
main = orDie renderAppError $ do
  contents   <- firstEitherT AppFileError $ readFile "foo.txt"
  nums       <- hoistEither $ first AppLineError $ parseLines contents
  normalised <- hoistEither $ first AppNormaliseError $ normalise nums
  firstEitherT AppFileError $ writeFile "bar.txt" $ renderDoubles normalised

renderAppError :: AppError -> Text
-- Built out of smaller renderers.
```

Testing

- Thin EitherT wrappers over IO code.
- Property-based testing for Either code.
- Factor IO out of wrappers.

Part of a principled approach

- No partial functions.
- Wrapping library functions.
- Careful error type design.

Comparison with other approaches

- MTL style
- Free

Thanks for listening!

Particular thanks to Navin Keswani who helped write this talk.

Code at github.com/expression-oriented/failing-gracefully