Error Handling in Haskell

Savanni D'Gerinel

2014-07-17

The Kinds of Error Reports

- Exceptions
- Either
- ErrorT or EitherT

Parsing a File

- parseImage :: ByteString -> Either ParseError (Image PixelRGB8)
- readFile :: FilePath -> IO ByteString
- readImageFile fname = readFile fname >>= return . parseImage :: FilePath -> IO (Either ParseError (Image PixelRGB8))

Exceptions

- throw :: Exception e => e -> a
- throwIO :: Exception e => e -> IO a
- catch :: Exception e => IO a -> (e -> IO a) -> IO a
- handle :: Exception e => (e -> IO a) -> IO a -> IO a
- try :: Exception e => IO a -> IO (Either e a)
- throwTo :: Exception e => ThreadId -> e -> IO ()
- error :: String -> a

Exceptions

```haskell
readFileExc :: FilePath -> IO ByteString
readFileExc = handle silenceENoEnt . readFile
  where
    silenceENoEnt :: IOException -> IO ByteString
    silenceENoEnt exc | isDoesNotExistError exc = return empty
                      | otherwise = throw exc

readImageFileExc :: FilePath -> IO Image
readImageFileExc fn = do
    bs <- readFileExc fn
    either throw return (parseImage bs)
```

Either

```haskell
data ReadImageError = ParseError ParseError | ReadError IOException

readFileEither :: FilePath -> IO (Either IOException ByteString)
readFileEither fn = try (readFile fn) >>= return . either silenceENoEnt Rig
  where
    silenceENoEnt :: IOException -> Either IOException ByteString
    silenceENoEnt exc | isDoesNotExistError exc = Right empty
                      | otherwise = Left exc

readImageFileEither :: FilePath -> IO (Either ReadImageError Image)
readImageFileEither fn = do
    mBs <- readFileEither fn
    return $ case mBs of
        Left err -> Left (ReadError err)
        Right bs -> either (Left . ParseError) Right (parseImage bs)
```

ErrorT and EitherT

- newtype ErrorT e m a = ErrorT { runErrorT :: m (Either e a) }
- newtype EitherT e m a = EitherT { runEitherT :: m (Either e a) }

ErrorT and EitherT

- newtype ErrorT e m a = ErrorT { runErrorT :: m (Either e a) }
- newtype EitherT e m a = EitherT { runEitherT :: m (Either e a) }
- EitherT :: m (Either e a) -> EitherT e m a
- runEitherT :: EitherT e m a -> m (Either e a)

EitherT

- catch :: Exception e => IO a -> (e -> IO a) -> IO a
- catch :: Exception e => IO (Either e a) -> (e -> IO (Either e' a)) -> IO (Either e' a)
- catchT :: Monad m => EitherT e m a -> (e -> EitherT e' m a) -> (EitherT e' m a)

EitherT

- handle :: Exception e => (e -> IO a) -> IO a -> IO a
- handleT :: Monad m => (e -> EitherT e' m a) -> EitherT e m a -> (EitherT e' m a)

EitherT

- try :: Exception e => IO a -> IO (Either e a)
- tryIO :: MonadIO m => IO a -> EitherT IOException m a
- EitherT . try :: Exception e => IO a -> EitherT e IO a

EitherT

```
catchT :: forall e e' m a. Monad m
          => EitherT e m a
          -> (e -> EitherT e' m a)
          -> EitherT e' m a
catchT action handler = EitherT $ do
    res <- runEitherT action :: m (Either e a)
    case res of
        Right val -> return (Right val)
        Left err -> runEitherT (handler err)
```

MonadError

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

Revisiting readImageFileEither

```haskell
data ReadImageError = ParseError ParseError | ReadError IOException

readImageFileEither :: FilePath -> IO (Either ReadImageError Image)
readImageFileEither fn = do
    mBs <- readFileEither fn
    return $ case mBs of
        Left err -> Left (ReadError err)
        Right bs -> either (Left . ParseError) Right (parseImage bs)
```

Revisiting readImageFileEither

```haskell
data ReadImageError = ParseError ParseError | ReadError IOException

readImageFileEither :: FilePath -> EitherT ReadImageError IO Image
readImageFileEither fn =
```

Revisiting readImageFileEither

```
readImageFileEither' :: FilePath -> EitherT ReadImageError IO Image
readImageFileEither' fn = do
    bs <- readFileEither' fn :: EitherT IOException IO ByteString
```

Revisiting readImageFileEither

```
readImageFileEither' :: FilePath -> EitherT ReadImageError IO Image
readImageFileEither' fn = do
    bs <- handleT (throwError . ReadError) (readFileEither' fn)
                :: EitherT ReadImageError IO ByteString
```

Revisiting readImageFileEither

```haskell
readImageFileEither' :: FilePath -> EitherT ReadImageError IO Image
readImageFileEither' fn = do
    bs <- handleT (throwError . ReadError) (readFileEither' fn)
    case parseImage bs of
        Right val -> return val
        Left exc -> throwError (ParseError exc)
```

Revisiting readImageFileEither

```haskell
readImageFileEither' :: FilePath -> EitherT ReadImageError IO Image
readImageFileEither' fn = do
    bs <- handleT (throwError . ReadError) (readFileEither' fn)
    either (throwError . ParseError) return (parseImage bs)
```

Revisiting readImageFileEither

```
readFileExc :: FilePath -> IO ByteString
readFileExc = handle silenceENoEnt . readFile
  where
    silenceENoEnt :: IOException -> IO ByteString
    silenceENoEnt exc | isDoesNotExistError exc = return empty
                      | otherwise = throw exc

readFileEither' :: FilePath -> EitherT IOException IO ByteString
readFileEither' = handleT silenceENoEnt . tryIO . readFile
  where
    silenceENoEnt :: IOException -> EitherT IOException IO ByteString
    silenceENoEnt exc | isDoesNotExistError exc = return empty
                      | otherwise = throwError exc
```

Revisiting readImageFileEither

```
silenceENoEnt' :: (MonadError IOException m, MonadIO m)
               => IOException
               -> m ByteString
silenceENoEnt' exc
    | isDoesNotExistError exc = return empty
    | otherwise = throwError exc

readFileExc :: FilePath -> IO ByteString
readFileExc = handle silenceENoEnt' . readFile

readFileEither' :: FilePath -> EitherT IOException IO ByteString
readFileEither' = handleT silenceENoEnt' . tryIO . readFile
```

Building an Application Stack

```
data DiskStore = DiskStore { root :: FilePath }

newtype DiskStoreM a =
        DSM {
            uDSM :: ReaderT DiskStore
                            (EitherT DataStoreError IO)
                            a }
    deriving ( Functor, Applicative, Monad, MonadIO,
             , MonadReader DiskStore, MonadError DataStoreError )
```

Put an Object In

```
putObject :: DataObject obj
          => Path
          -> obj
          -> Maybe ObjVersion
          -> DiskStoreM ObjVersion
putObject path obj mVer = do
    DiskStore{..} <- ask
    let fsPath = root </> "objects" </> T.unpack (T.intercalate "/" elems)
    createDirectoryIfMissing True (dropFileName fsPath)
    BS.writeFile fsPath (content obj)
    return (ObjVersion T.empty)
```

- createDirectoryIfMissing :: Bool -> FilePath -> IO ()
- writeFile :: FilePath -> ByteString -> IO ()

hoistIO

```
(IOException -> EitherT DataStoreError IO a) -> IO a -> DiskStoreM a
```

hoistIO

```
hoistIO :: (IOException -> EitherT DataStoreError IO a)
        -> IO a
        -> EitherT IOException IO a
hoistIO handler action =
    tryIO action
```

hoistIO

```
hoistIO :: (IOException -> EitherT DataStoreError IO a)
        -> IO a
        -> EitherT DataStoreError IO a
hoistIO handler action =
    handleT handler (tryIO action)
```

hoistIO

```
hoistIO :: (IOException -> EitherT DataStoreError IO a)
        -> IO a
        -> ReaderT DiskStore (EitherT DataStoreError IO) a
hoistIO handler action =
    ReaderT (\_ -> handleT handler
                           (tryIO action))

hoistIO handler action =
    lift (handleT (throwError . handler)
                  (tryIO action))
```

hoistIO

```
hoistIO :: (IOException -> EitherT DataStoreError IO a)
        -> IO a
        -> DiskStoreM a
hoistIO handler action =
    DSM (lift (handleT handler (tryIO action)))
```

Back to adding the object

```
putObject :: DataObject obj => Path
                            -> obj
                            -> Maybe ObjVersion
                            -> DiskStoreM ObjVersion
putObject (Path elems) obj mVer = do
    DiskStore{..} <- ask
    let fsPath = root </> "objects" </> T.unpack (T.intercalate "/" elems)
    hoistIO (throwError . trIOExc) $ do
        createDirectoryIfMissing True (dropFileName fsPath)
        BS.writeFile fsPath (content obj)
    return (ObjVersion T.empty)
```

Catching multiple types of Exceptions

This page left blank

Catching and handling inside the monad

```
putObject :: DataObject obj => Path
                            -> obj
                            -> Maybe ObjVersion
                            -> DiskStoreM ObjVersion
putObject (Path elems) obj mVer = do
    DiskStore{..} <- ask
    let fsPath = root </> "objects" </> T.unpack (T.intercalate "/" elems)
    res <- liftIO $ do
        createDirectoryIfMissing True (dropFileName fsPath)
        BS.writeFile fsPath (content obj)
    return (ObjVersion T.empty)
```

Links

- My website – `http://www.savannidgerinel.com/`
- errors –