Monad Transformers

The critical feature of functional languages with parametric polymorphism

WTF are Monads?

- An interface to an abstract “context” for values
- ...containing a set of behaviors that adhere to defined laws

Some Contexts and Their Monads

Nonexistance	`Maybe`, `Option`
Error-generating	`Either`, `Error`, `Validation`
Asynchronicity	`Continuation`, `Promise`
Side-effectful	`IO`, `Effect`
Nondeterminism	`List`

The Interface

- `return` wraps a value in the context
- `bind` (aka `>>=`) allows for chaining successive computations

Monad Laws (a.k.a. Sausage)

```
return value >>= fn == fn value

wrapped >>= return == wrapped

wrapped >>= (\x -> fn1 x >>= fn2) ==
  (wrapped >>= fn1) >>= fn2
```

- Implementations differ per monad!

Haskell's Do-Notation

```haskell
myFunction = do
  x <- getValueX
  y <- getValueY
  return (x + y)
```

```haskell
myFunction =
  getValueX >>= \x ->
    getValueY >>= \y ->
      return (x + y)
```

Why should I bother?

- Successive contextual computations is **how we conceptually model applications**!

You Can Have This: 😍

```haskell
readKey :: JSON -> String -> Either err a
validateName :: String -> Either err Name
validateAge :: Int -> Either err Age
```

```haskell
getUser :: JSON -> Either err User
getUser json = do
  name  <- readKey "username" json
  age   <- readKey "age" json
  name' <- validateName name
  age'  <- validateAge age
  return $ User name' age'
```

Or This: 😨🔫

```
def userFromJson(json)
  if json.nil? || notActuallyJson?(json)
    raise ProgramIsScrewed "Wat r JSON?"
  end
  name = json.readKey "name"
  unless name && isValidName(name)
    raise ProgramIsScrewed "In death you have a name."
  end
  age = json.readKey "age"
  ...
```

Just Be a Better Programmer!

- …is an utterly crap argument
- Relying on developers to mentally track and handle every context is like removing lights and paint from roadways

However...

Did you spot the weakness?

A monad can only track a single context!

```haskell
getUser :: JSON -> Either err User
getUser json = do
  name  <- readKey "username" json
  age   <- readKey "age" json
  name' <- validateName name
  age'  <- validateAge age
  return $ User name' age'
```

We can try to handle it manually, but...

In the IO monad we've lost the ability to chain errors!

```
readUser :: IO (Either err User)
readUser = do
    userFile <- readFile "users.json"
    users <- case userFile of
      Left err -> return $ Left err
      Right file -> return $ parseUser
    case users of
      Left parseErr
      ...
```

Super sad face!

...but wait!

We can do better!

Enter the **Monad Transformer**!

Transformer class

```
class MonadTrans t where
```

The class of monad transformers. Instances should satisfy the following
`lift` is a transformer of monads:

```
lift . return = return
```

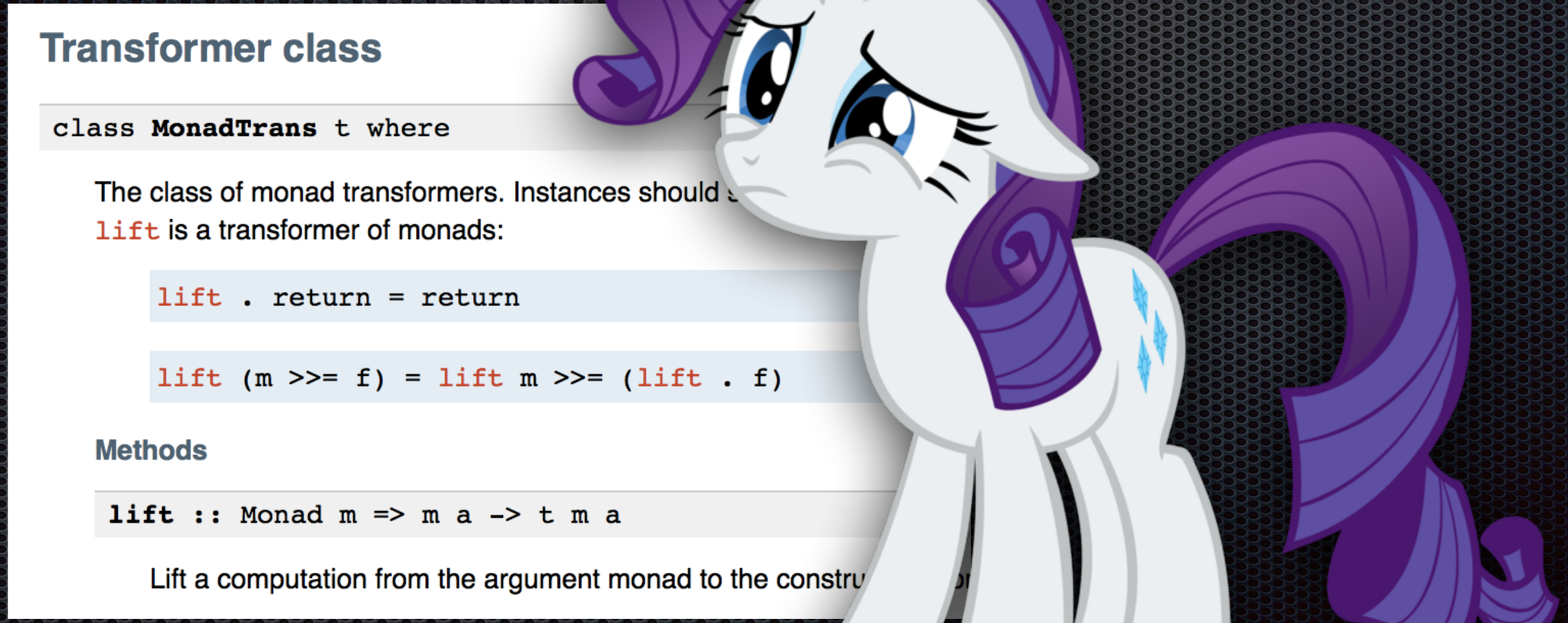
```
lift (m >>= f) = lift m >>= (lift . f)
```

Methods

```
lift :: Monad m => m a -> t m a
```

Lift a computation from the argument monad to the constructed mor

Enter the **Monad Transformer**!



Transformer class

```
class MonadTrans t where
```

The class of monad transformers. Instances should
`lift` is a transformer of monads:

```
lift . return = return
```

```
lift (m >>= f) = lift m >>= (lift . f)
```

Methods

```
lift :: Monad m => m a -> t m a
```

Lift a computation from the argument monad to the constru

Let’s start with a working example…

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

```
myWebGLProgram :: WebGL Unit
myWebGLProgram = do
  clearColor 1.0 0.0 0.0 1.0
  clear ColorBuffer
  debugger
```

…and how we run it…

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

```
main :: Eff (canvas :: Canvas) (Either WebGLError Unit)
main = do
    (Just el) <- getCanvasElementById "webgl"
    (Just webgl) <- getWebGLContext el
    runErrorT $ runReaderT myWebGLProgram webgl
```

…and why!

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

- All WebGL methods are called on a WebGL object
- Any WebGL method has the possibility of failure
- All WebGL methods cause side-effects

So WTF is a **Monad Transformer**?

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

- Allows interleaving of monadic contexts
- Each context is “run” in the wrapping monad
- Monad results are then handled during unwrapping

What about the sausage?

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

Make simple, composable functions…

```
isContextLost :: WebGL Boolean
isContextLost = do
  ctx <- ask
  liftReaderT $ liftEff $ Raw.isContextLost ctx

getError :: WebGL Number
getError = do
  ctx <- ask
  liftReaderT $ liftEff $ Raw.getError ctx
```

What about the sausage?

```
type WebGL a = ReaderT WebGLContext
  (ErrorT WebGLError (Eff (canvas :: Canvas))) a
```

...to create complex behavior!

```
debugger :: WebGL Unit
debugger = do
  hasCtx <- not <$> isContextLost
  errNum <- getError
  when (hasCtx && errNum == Enum.noError)
    (throwError WebGLError)
```

While retaining access to all monads!

“Lift” functions move computations to next monad

```
isContextLost :: WebGL Boolean
isContextLost = do
  ctx <- ask
  liftReaderT $ liftEff $ Raw.isContextLost ctx

getError :: WebGL Number
getError = do
  ctx <- ask
  liftReaderT $ liftEff $ Raw.getError ctx
```

Finally— the f&%$ing point:

- Monad transformers allow us to model arbitrarily complex behavior in a composable fashion
- The types ensure that all contexts are handled
- Nearly all applications are a nesting of transformers!

```
type ApiServerM s = ScottyT ApiException
  (ReaderT ServerState (Session Postgres s IO))
```

Case in point:

- Monad transformers are **so critical** to application modelling that all Haskell monads were rewritten **in terms of their transformers**

```haskell
type Reader r = ReaderT r Identity

reader :: Monad m => (r -> a) -> ReaderT r m a
reader f = ReaderT (return . f)

runReader :: Reader r a -> r -> a
runReader m = runIdentity . runReaderT m
```