Как достигать желаемых эффектов

Денис Буздалов

4 декабря 2019

Нам понадобится

- `{-# LANGUAGE MultiParamTypeClasses #-}`
- Типы высших порядков

```
class Monad m ⇒ MonadError e (m :: * → *) ...
```

Нам понадобится

- `{-# LANGUAGE MultiParamTypeClasses #-}`
- Типы высших порядков

  ```
  class Monad m ⇒ MonadError e (m :: * → *) ...
  ```

- `{-# LANGUAGE FunctionalDependencies #-}`

  ```
  class ... ⇒ MonadError e m | m → e where ...
  ```

Нам понадобится

- `{-# LANGUAGE MultiParamTypeClasses #-}`
- Типы высших порядков

  ```
  class Monad m ⇒ MonadError e (m :: * → *) ...
  ```

- `{-# LANGUAGE FunctionalDependencies #-}`

  ```
  class ... ⇒ MonadError e m | m → e where ...
  ```

  “Дописывание” в голове где это требуется

Нам понадобится

- `{-# LANGUAGE MultiParamTypeClasses #-}`
- Типы высших порядков

  ```
  class Monad m ⇒ MonadError e (m :: * → *) ...
  ```

- `{-# LANGUAGE FunctionalDependencies #-}`

  ```
  class ... ⇒ MonadError e m | m → e where ...
  ```

  “Дописывание” в голове где это требуется
- Код с алгебрами

  ```
  f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
  ```

- Функторы, аппликативы, монады

Нам понадобится

- `{-# LANGUAGE MultiParamTypeClasses #-}`
- Типы высших порядков

  ```
  class Monad m ⇒ MonadError e (m :: * → *) ...
  ```

- `{-# LANGUAGE FunctionalDependencies #-}`

  ```
  class ... ⇒ MonadError e m | m → e where ...
  ```

  “Дописывание” в голове где это требуется
- Код с алгебрами

  ```
  f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
  ```

- Функторы, аппликативы, монады
- Разное лукавство ;-)

О чём доклад

- Код на typeclass’ах
- Частный случай: монолит
- Отдельные эффекты
- Композиция: успехи и проблемы
- Всё не так плохо: другие системы эффектов

Пример

```haskell
import Prelude hiding (putStrLn)
import qualified Prelude (putStrLn)

class Monad m ⇒ MonadError e m where
  throwError :: e → m a
class Monad m ⇒ PrintConsole m where
  putStrLn :: String → m ()
```

Пример

```haskell
import Prelude hiding (putStrLn)
import qualified Prelude (putStrLn)

class Monad m ⇒ MonadError e m where
  throwError :: e → m a
class Monad m ⇒ PrintConsole m where
  putStrLn :: String → m ()

data Err = DivByZero | ...
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

Пример

```haskell
import Prelude hiding (putStrLn)
import qualified Prelude (putStrLn)

class Monad m ⇒ MonadError e m where
  throwError :: e → m a
class Monad m ⇒ PrintConsole m where
  putStrLn :: String → m ()


data Err = DivByZero | ...
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()

f x = do
  when (x == 0) $ throwError DivByZero
  putStrLn $ "100/x is " ++ (show $ 100 `div` x)
```

Пример

```haskell
import Prelude hiding (putStrLn)
import qualified Prelude (putStrLn)

class Monad m ⇒ MonadError e m where
  throwError :: e → m a
class Monad m ⇒ PrintConsole m where
  putStrLn :: String → m ()


data Err = DivByZero | ...
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()

f x = do
  when (x == 0) $ throwError DivByZero
  putStrLn $ "100/x is " ++ (show $ 100 `div` x)


when :: Applicative f ⇒ Bool → f () → f ()
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
instance Show e ⇒ MonadError e IO where
  throwError = error . show

instance PrintConsole IO where
  putStrLn = Prelude.putStrLn
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
instance Show e ⇒ MonadError e IO where
  throwError = error . show

instance PrintConsole IO where
  putStrLn = Prelude.putStrLn
```

```
ioFun :: IO ()
ioFun = f 5
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
M :: * → *
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
M :: * → *
M  ~~  Either Err [String]
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
M :: * → *
M  ~~  Either Err [String]
M a ~~ (Either Err [String], a)
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
M :: * → *
M  ~~  Either Err [String]
M a ~~ (Either Err [String], a)
M a ~~ Either Err ([String], a)
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
```

```
rsFun :: Either Err [String]
rsFun = f 5
```

Сработает?

```
Either Err [String] :: *
M :: * → *
M  ~~  Either Err [String]
M a ~~ (Either Err [String], a)
M a ~~ Either Err ([String], a)
Res e a ~~ Either e ([String], a)
```

Как бы запустить?

```
rsFun :: Either Err [String]
```

```
newtype Res e a = Res (Either e ([String], a))
```

Как бы запустить?

```
rsFun :: Either Err [String]

newtype Res e a = Res (Either e ([String], a))

instance Functor (Res e) where
  fmap f (Res ei) = Res $ fmap (fmap f) ei
instance Applicative (Res e) where
  pure x = Res $ Right ([], x)
  Res (Left e)  <*> _             = Res $ Left e
  _             <*> Res (Left e)  = Res $ Left e
  Res (Right p) <*> Res (Right q) = Res . Right $ p <*> q
instance Monad (Res e) where
  Res (Left e)        >>= _ = Res $ Left e
  Res (Right (sl, a)) >>= f = case f a of
    Res (Right (sr, b)) -> Res $ Right (sl ++ sr, b)
    Res er@(Left _)     -> Res er
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
newtype Res e a = Res (Either e ([String], a))
```

Как бы запустить?

```haskell
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
newtype Res e a = Res (Either e ([String], a))


instance MonadError e (Res e) where
  throwError = Res . Left
instance PrintConsole (Res e) where
  putStrLn s = Res $ Right ([s], ())
```

Как бы запустить?

```haskell
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
newtype Res e a = Res (Either e ([String], a))


instance MonadError e (Res e) where
  throwError = Res . Left
instance PrintConsole (Res e) where
  putStrLn s = Res $ Right ([s], ())


unRes :: Res e a → Either e [String]
unRes (Res (Left e))         = Left e
unRes (Res (Right (ss, _))) = Right ss
```

Как бы запустить?

```
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
newtype Res e a = Res (Either e ([String], a))


instance MonadError e (Res e) where
  throwError = Res . Left
instance PrintConsole (Res e) where
  putStrLn s = Res $ Right ([s], ())


unRes :: Res e a → Either e [String]
unRes (Res (Left e))         = Left e
unRes (Res (Right (ss, _))) = Right ss

rsFun :: Either Err [String]
rsFun = unRes $ f 5
```

А посложней?

```haskell
data Err = DivByZero                    -- recall ...
  deriving (Show)
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
f x = do
  when (x == 0) $ throwError DivByZero
  putStrLn $ "100/x is " ++ (show $ 100 `div` x)
```

А посложней?

```haskell
data Err = DivByZero                    -- recall ...
  deriving (Show)
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
f x = do
  when (x == 0) $ throwError DivByZero
  putStrLn $ "100/x is " ++ (show $ 100 `div` x)
```

```haskell
class Monad m ⇒ MonadReader r m where
  ask :: m r
```

А посложней?

```haskell
data Err = DivByZero                    -- recall ...
  deriving (Show)
f :: (MonadError Err m, PrintConsole m) ⇒ Integer → m ()
f x = do
  when (x == 0) $ throwError DivByZero
  putStrLn $ "100/x is " ++ (show $ 100 `div` x)
```

```haskell
class Monad m ⇒ MonadReader r m where
  ask :: m r

g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
g = ask >>= f
```

А посложней?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

ioGunc :: IO ()
ioGunc = g
```

Возможно?

А посложней?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

ioGunc :: IO ()
ioGunc = g
```

Возможно?

Например, можно спросить извне:

```
instance MonadReader Integer IO where
  ask = Prelude.putStrLn "Enter number" >> readLn
```

А посложней?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

ioGunc :: IO ()
ioGunc = g
```

Возможно?

Например, можно спросить извне:

```
instance MonadReader Integer IO where
  ask = Prelude.putStrLn "Enter number" >> readLn
```

Или считать из файла. Или сгенерировать случайно.
Всё, что позволяет нам IO!

А посложней?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

ioGunc :: IO ()
ioGunc = g
```

Возможно?

Например, можно спросить извне:

```
instance MonadReader Integer IO where
  ask = Prelude.putStrLn "Enter number" >> readLn
```

Или считать из файла. Или сгенерировать случайно.
Всё, что позволяет нам `IO`!

замечание о `newtype`

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

rsGunc :: Integer → Either Err [String]
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

rsGunc :: Integer → Either Err [String]
```

```
    Either Err [String] :: *        -- recall ...
    M :: * → *
    newtype M a = M (Either Err ([String], a))
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

rsGunc :: Integer → Either Err [String]
```

```
    Either Err [String] :: *        -- recall ...
    M :: * → *
    newtype M a = M (Either Err ([String], a))
    newtype Res e a = Res (Either e ([String], a))
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

rsGunc :: Integer → Either Err [String]

    Either Err [String] :: *        -- recall ...
    M :: * → *
    newtype M a = M (Either Err ([String], a))
    newtype Res e a = Res (Either e ([String], a))


Integer → Either Err [String] :: *
N :: * → *
newtype N a = N (Integer → Either Err ([String], a))
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()

rsGunc :: Integer → Either Err [String]

    Either Err [String] :: *        -- recall ...
    M :: * → *
    newtype M a = M (Either Err ([String], a))
    newtype Res e a = Res (Either e ([String], a))


Integer → Either Err [String] :: *
N :: * → *
newtype N a = N (Integer → Either Err ([String], a))

newtype Fer e r a = Fer (r → Either e ([String], a))
```

А в статике?

```haskell
newtype Fer e r a = Fer (r → Either e ([String], a))

instance Functor (Fer e r) where
  fmap f (Fer rei) = Fer $ fmap (fmap (fmap f)) rei
instance Applicative (Fer e r) where
  pure x = Fer . const $ Right ([], x)
  Fer f <*> Fer g = Fer $ \r → case (f r, g r) of
    (Left e , _)       → Left e
    (_      , Left e)  → Left e
    (Right p, Right q) → Right $ p <*> q
instance Monad (Fer e r) where
  Fer rei >>= f = Fer $ \r → case rei r of
    Left e         → Left e
    Right (sl, a) → let Fer fs = f a in case fs r of
      Right (sr, b) → Right (sl ++ sr, b)
      er@(Left _)   → er
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
newtype Fer e r a = Fer (r → Either e ([String], a))
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
newtype Fer e r a = Fer (r → Either e ([String], a))


instance MonadError e (Fer e r) where
  throwError = Fer . const . Left

instance PrintConsole (Fer e r) where
  putStrLn s = Fer . const $ Right ([s], ())

instance MonadReader r (Fer e r) where
  ask = Fer $ \r → Right ([], r)
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
newtype Fer e r a = Fer (r → Either e ([String], a))

instance MonadError e (Fer e r) where ...
instance PrintConsole (Fer e r) where ...
instance MonadReader r (Fer e r) where ...
```

А в статике?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
newtype Fer e r a = Fer (r → Either e ([String], a))

instance MonadError e (Fer e r) where ...
instance PrintConsole (Fer e r) where ...
instance MonadReader r (Fer e r) where ...


unFer :: Fer e r a → r → Either e [String]
unFer (Fer rei) r = case rei r of
  Left e        → Left e
  Right (ss, _) → Right ss

rsGunc :: Integer → Either Err [String]
rsGunc = unFer g
```

Промежуточный вывод

Если

- монадический интерфейс, выраженный typeclass’ax

Промежуточный вывод

Если

- монадический интерфейс, выраженный typeclass’ax
- есть тип требуемого результата

Промежуточный вывод

Если

- монадический интерфейс, выраженный typeclass’ах
- есть тип требуемого результата
- задача разрешима

Промежуточный вывод

Если

- монадический интерфейс, выраженный typeclass’ах
- есть тип требуемого результата
- задача разрешима

то

- существует специализированная монада
  - с инстансами требуемых typeclass’ов
  - с функцией, дающей нам требуемый результат

Собирать из кусочков

Мы же функциональные программисты!

Нас хлебом не корми, дай разбить на куски, переиспользовать и композировать!

Собирать из кусочков

Мы же функциональные программисты!

Нас хлебом не корми, дай разбить на куски, переиспользовать и композировать!

```haskell
newtype Res e a = Res (Either e ([String], a))
newtype Fer e r a = Fer (r → Either e ([String], a))
```

Собирать из кусочков

Мы же функциональные программисты!

Нас хлебом не корми, дай разбить на куски, переиспользовать и композировать!

```
newtype Res e a = Res (Either e ([String], a))
newtype Fer e r a = Fer (r → Either e ([String], a))
```

Самурай без меча подобен самураю с мечом,
но только без меча

Собирать из кусочков

Мы же функциональные программисты!

Нас хлебом не корми, дай разбить на куски, переиспользовать и композировать!

```
newtype Res e a = Res (Either e ([String], a))
newtype Fer e r a = Fer (r → Either e ([String], a))
```

Самурай без меча подобен самураю с мечом,
но только без меча

Посмотрим на `Fer` как на (`Fer` без `Res`) с `Res`

Преобразуем

```haskell
newtype Res e a = Res (Either e ([String], a))
```

```haskell
newtype Fer e r a = Fer (r → Either e ([String], a))
```

Преобразуем

```haskell
newtype Res e a = Res (Either e ([String], a))
```

```haskell
newtype Fer e r a = Fer (r → Either e ([String], a))
```

```haskell
newtype Fer e r a = Fer (r → Res e a)
```

Преобразуем

```
newtype Res e a = Res (Either e ([String], a))
```

```
newtype Fer e r a = Fer (r → Either e ([String], a))
```

```
newtype Fer e r a = Fer (r → Res e a)
```

Так ли важен конкретный `Res e`?

Преобразуем

```
newtype Res e a = Res (Either e ([String], a))
```

```
newtype Fer e r a = Fer (r → Either e ([String], a))
newtype Fer e r a = Fer (r → Res e a)
```

Так ли важен конкретный `Res e`?

```
newtype Fer (m :: * → *) r a = Fer (r → m a)
```

Преобразуем

```
newtype Res e a = Res (Either e ([String], a))
```

```
newtype Fer e r a = Fer (r → Either e ([String], a))
newtype Fer e r a = Fer (r → Res e a)
```

Так ли важен конкретный `Res e`?

```
newtype Fer (m :: * → *) r a = Fer (r → m a)
newtype Fer r m a = Fer (r → m a)
```

Преобразуем

```haskell
newtype Res e a = Res (Either e ([String], a))
```

```haskell
newtype Fer e r a = Fer (r → Either e ([String], a))
newtype Fer e r a = Fer (r → Res e a)
```

Так ли важен конкретный `Res e`?

```haskell
newtype Fer (m :: * → *) r a = Fer (r → m a)
newtype Fer r m a = Fer (r → m a)
newtype ReaderT r m a = ReaderT (r → m a)
```

Преобразуем

```
newtype Res e a = Res (Either e ([String], a))
```

```
newtype Fer e r a = Fer (r → Either e ([String], a))
newtype Fer e r a = Fer (r → Res e a)
```

Так ли важен конкретный `Res e`?

```
newtype Fer (m :: * → *) r a = Fer (r → m a)
newtype Fer r m a = Fer (r → m a)
newtype ReaderT r m a = ReaderT (r → m a)
```

замечание о kind polymorphism в реальности

Воспользуемся

```
newtype ReaderT r m a = ReaderT (r → m a)
```

Воспользуемся

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

runReaderT :: ReaderT r m a → r → m a
```

Воспользуемся

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

runReaderT :: ReaderT r m a → r → m a

instance Monad m ⇒ MonadReader r (ReaderT r m) where
  ask = ReaderT $ \r → pure r
```

Воспользуемся

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

runReaderT :: ReaderT r m a → r → m a

instance Monad m ⇒ MonadReader r (ReaderT r m) where
  ask = ReaderT $ \r → pure r

instance MonadError e (Res e) where ...
instance PrintConsole (Res e) where ...
unRes :: Res e a → Either e [String]
```

Воспользуемся

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

runReaderT :: ReaderT r m a → r → m a

instance Monad m ⇒ MonadReader r (ReaderT r m) where
  ask = ReaderT $ \r → pure r

instance MonadError e (Res e) where ...
instance PrintConsole (Res e) where ...
unRes :: Res e a → Either e [String]

g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
rsGunc' :: Integer → Either Err [String]
rsGunc' = unRes . runReaderT g
```

Воспользуемся

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

runReaderT :: ReaderT r m a → r → m a

instance Monad m ⇒ MonadReader r (ReaderT r m) where
  ask = ReaderT $ \r → pure r

instance MonadError e (Res e) where ...
instance PrintConsole (Res e) where ...
unRes :: Res e a → Either e [String]

g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
rsGunc' :: Integer → Either Err [String]
rsGunc' = unRes . runReaderT g
```

Взлетит?

Проброс несущественных эффектов

```haskell
newtype ReaderT r m a = ReaderT (r → m a)
```

Проброс несущественных эффектов

```haskell
newtype ReaderT r m a = ReaderT (r → m a)


instance MonadError e m ⇒ MonadError e (ReaderT r m) where
  throwError = ReaderT . const . throwError

instance PrintConsole m ⇒ PrintConsole (ReaderT r m) where
  putStrLn = ReaderT . const . putStrLn
```

Проброс несущественных эффектов

```haskell
newtype ReaderT r m a = ReaderT (r → m a)


instance MonadError e m ⇒ MonadError e (ReaderT r m) where
  throwError = ReaderT . const . throwError

instance PrintConsole m ⇒ PrintConsole (ReaderT r m) where
  putStrLn = ReaderT . const . putStrLn
```

- Работает! Но...
- Комбинаторный взрыв instance'ов
- Добавление новых классов?

Проброс несущественных эффектов

```haskell
newtype ReaderT r m a = ReaderT (r → m a)

instance MonadError e m ⇒ MonadError e (ReaderT r m) where
  throwError = ReaderT . const . throwError

instance PrintConsole m ⇒ PrintConsole (ReaderT r m) where
  putStrLn = ReaderT . const . putStrLn
```

- Работает! Но...
- Комбинаторный взрыв instance'ов
- Добавление новых классов?
  `ReaderT . const :: m a → ReaderT r m a`

Проброс несущественных эффектов

```haskell
class MonadTrans (t :: (* → *) → * → *) where
  lift :: m a → t m a
```

Проброс несущественных эффектов

```haskell
class MonadTrans (t :: (* → *) → * → *) where
  lift :: m a → t m a

newtype ReaderT r m a = ReaderT (r → m a)
instance MonadTrans (ReaderT r) where
  lift = ReaderT . const
```

Проброс несущественных эффектов

```haskell
class MonadTrans (t :: (* → *) → * → *) where
  lift :: m a → t m a

newtype ReaderT r m a = ReaderT (r → m a)
instance MonadTrans (ReaderT r) where
  lift = ReaderT . const

instance {-# OVERLAPPABLE #-} (MonadError e m,
    MonadTrans t, Monad (t m)) ⇒ MonadError e (t m) where
  throwError = lift . throwError

instance {-# OVERLAPPABLE #-} (PrintConsole m,
    MonadTrans t, Monad (t m)) ⇒ PrintConsole (t m) where
  putStrLn = lift . putStrLn
```

Проброс несущественных эффектов

```haskell
class MonadTrans (t :: (* → *) → * → *) where
  lift :: m a → t m a

newtype ReaderT r m a = ReaderT (r → m a)
instance MonadTrans (ReaderT r) where
  lift = ReaderT . const

instance {-# OVERLAPPABLE #-} (MonadError e m,
    MonadTrans t, Monad (t m)) ⇒ MonadError e (t m) where
  throwError = lift . throwError

instance {-# OVERLAPPABLE #-} (PrintConsole m,
    MonadTrans t, Monad (t m)) ⇒ PrintConsole (t m) where
  putStrLn = lift . putStrLn
```

kinda kind problem

```haskell
{-# UndecidableInstances #-}
```

Раздеваем дальше

```
newtype Fer e r a = Fer (r → Either e ([String], a))
Fer e r :: * → *
```

Раздеваем дальше

```haskell
newtype Fer e r a = Fer (r → Either e ([String], a))
Fer e r :: * → *

newtype Res e a = Res (Either e ([String], a))
data ReaderT r m a = ReaderT (r → m a)
ReaderT r (Res e) :: * → *
```

Раздеваем дальше

```
newtype Fer e r a = Fer (r → Either e ([String], a))
Fer e r :: * → *

newtype Res e a = Res (Either e ([String], a))
data ReaderT r m a = ReaderT (r → m a)
ReaderT r (Res e) :: * → *

newtype Strs a = Strs ([String], a)
data ExceptT e m a = ExceptT (m (Either e a))
ExceptT e (ReaderT r (Strs a)) :: * → *
```

Раздеваем дальше

```
newtype Fer e r a = Fer (r → Either e ([String], a))
Fer e r :: * → *

newtype Res e a = Res (Either e ([String], a))
data ReaderT r m a = ReaderT (r → m a)
ReaderT r (Res e) :: * → *

newtype Strs a = Strs ([String], a)
data ExceptT e m a = ExceptT (m (Either e a))
ExceptT e (ReaderT r (Strs a)) :: * → *

newtype Identity a = Identity a
data StateT s m a = StateT (s → m (a, s))
StateT [String] (ExceptT e (ReaderT r Identity)) :: * → *
```

Раздеваем дальше

```
newtype Fer e r a = Fer (r → Either e ([String], a))
Fer e r :: * → *

newtype Res e a = Res (Either e ([String], a))
data ReaderT r m a = ReaderT (r → m a)
ReaderT r (Res e) :: * → *

newtype Strs a = Strs ([String], a)
data ExceptT e m a = ExceptT (m (Either e a))
ExceptT e (ReaderT r (Strs a)) :: * → *

newtype Identity a = Identity a
data StateT s m a = StateT (s → m (a, s))
StateT [String] (ExceptT e (ReaderT r Identity)) :: * → *

StateT [String] (ExceptT e (Reader r)) :: * → *
```

Гюльчатай, открой личико

```haskell
data ReaderT r m a = ReaderT (r → m a)
data Reader r = ReaderT r Identity

runReaderT :: ReaderT r m a → r → m a
runReader  :: Reader r a    → r → a
```

Гюльчатай, открой личико

```haskell
data ReaderT r m a = ReaderT (r → m a)
data Reader r = ReaderT r Identity

runReaderT :: ReaderT r m a → r → m a
runReader  :: Reader r a    → r → a

data ExceptT e m a = ExceptT (m (Either e a))

runExceptT :: ExceptT e m a → m (Either e a)
runExcept  :: Except e a    → Either e a
```

Гюльчатай, открой личико

```haskell
data ReaderT r m a = ReaderT (r → m a)
data Reader r = ReaderT r Identity

runReaderT :: ReaderT r m a → r → m a
runReader  :: Reader r a    → r → a

data ExceptT e m a = ExceptT (m (Either e a))

runExceptT :: ExceptT e m a → m (Either e a)
runExcept  :: Except e a    → Either e a

data StateT s m a = StateT (s → m (a, s))

runStateT  :: StateT s m a → s → m (a, s)
evalStateT :: StateT s m a → s → m a
execStateT :: StateT s m a → s → m s

runState   :: State s a    → s → (a, s)
```

Так как же запускать?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
```

Так как же запускать?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
```

```
runReaderT :: ReaderT r m a → (r → m a)
runReader  :: Reader r a    → (r → a)
runExceptT :: ExceptT e m a → m (Either e a)
execStateT :: StateT s m a  → s → m s
```

Так как же запускать?

```
g :: (MonadError Err m, PrintConsole m,
      MonadReader Integer m) ⇒ m ()
```

```
runReaderT :: ReaderT r m a → (r → m a)
runReader  :: Reader r a    → (r → a)
runExceptT :: ExceptT e m a → m (Either e a)
execStateT :: StateT s m a  → s → m s
```

```
instance Monad m ⇒ PrintConsole (StateT [String] m) where
  putStrLn s = modify (++ [s])

rsGunc'' :: Integer → Either Err [String]
rsGunc'' = runReader . runExceptT . flip execStateT [] $ g
```

Итого: эволюция

```haskell
-- монолитная монада `Fer`
rsGunc = unFer g

-- композитная монада `Res` в `ReaderT Integer`
rsGunc' = unRes . runReaderT g

-- полностью раздетый стек из `StateT [String]`,
--   `ExceptT Error`, `ReaderT Integer` и `Identity`
rsGunc'' = runReader . runExceptT . flip execStateT [] $ g
```

The Good, the Bad and the Ugly

- Плюсы
  - Переиспользование
  - Композируемость
- Минусы MTL-style
  - Невозможность наслаивания однотипных ограничений

    ```
    f :: (MonadReader GlobalConf m, MonadReader LocalConf m) ⇒ ...
    ```
- Минусы транформеров
  - Скорость работы (постоянные перепаковки)
  - Ограниченность обработчиков
    - Можно снять только верхние трансформеры
    - Привязка к конкретным трансформерам в обработчиках
    - Ограниченность переупорядочения
  - Свои монады на базе трансформеров часто неудобны

    ```
    ... >> lift . lift . lift . op >>= ...
    ```
- Подводные камни
  - Много что держится на `OverlappableInstances`
  - Иногда взрывается вывод типов
  - Нечитаемые ошибки типизации

Всё не так плохо

Freer monad

```
g :: (Member (Error Err) eff, Member Console eff,
      Member (Reader Integer) eff) ⇒ Eff eff ()
g = do
  x ← ask
  when (x == 0) $ throwError DivByZero
  putStrLn' $ "100/x is " ++ (show $ 100 `div` x)
```

Всё не так плохо

Capabilities

```
g :: (HasThrow "arith" Err m, HasConsole "cons" m,
      HasReader "arg" Integer m) ⇒ m ()
g = do
  x ← ask @"arg"
  when (x == 0) $ throw @"arith" DivByZero
  putStrLn @"cons" $ "100/x is " ++ (show $ 100 `div` x)
```

Всё не так плохо

Capabilities

```
g :: (HasThrow "arith" Err m, HasConsole "cons" m,
      HasReader "arg" Integer m) ⇒ m ()
g = do
  x ← ask @"arg"
  when (x == 0) $ throw @"arith" DivByZero
  putStrLn @"cons" $ "100/x is " ++ (show $ 100 `div` x)
```

И многое другое...

Всё