Полиморфизм в Haskell и typeclasses

Краткий ликбез и пример

Денис Buздалов

1 ноября 2019

Предположения докладчика

Предположения докладчика

Слушатели

- Воспринимают *густые* слайды

Предположения докладчика

Слушатели

- Воспринимают *густые* слайды

  с последовательно вываливающимися элементами

Предположения докладчика

Слушатели

- Воспринимают *густые* слайды

  с последовательно вываливающимися элементами
- Легко читают код со слайдов

Предположения докладчика

Слушатели

- Воспринимают *густые* слайды

  с последовательно вываливающимися элементами
- Легко читают код со слайдов
- Стремятся задать вопрос, когда непонятно

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x            g 5 6            f . g 5
```

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x             g 5 6              f . g 5
g 5 (f x)                 g 5 $ f x
```

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x             g 5 6             f . g 5
g 5 (f x)                   g 5 $ f x
x `elem` xs
```

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x              g 5 6                f . g 5
g 5 (f x)                   g 5 $ f x
x `elem` xs
```

- Базовые типы

```
Int      Int -> String      [Int]      Maybe Int
```

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x            g 5 6            f . g 5
g 5 (f x)                 g 5 $ f x
x `elem` xs
```

- Базовые типы

```
Int     Int -> String     [Int]     Maybe Int
```

- Pattern matching

Предположения докладчика

- Algebraic data types

```
data X = A | B Int | C Int String
```

- Применение функций

```
f x          g 5 6          f . g 5
g 5 (f x)              g 5 $ f x
x `elem` xs
```

- Базовые типы

```
Int     Int -> String     [Int]     Maybe Int
```

- Pattern matching
- Синтаксис списков

```
x:xs                [a, b, c]
```

О чём доклад

- Параметрический и ad-hoc полиморфизм
- Тайпклассы и полиморфизм
- Возможности, которые даёт использование полиморфизма
  - Чужой код работает иначе
  - Наш код работает по-разному

О чём доклад

- Параметрический и ad-hoc полиморфизм
- Тайпклассы и полиморфизм
- Возможности, которые даёт использование полиморфизма
  - Чужой код работает иначе
  - Наш код работает по-разному

Доклад не подразумевает полноту изложения

О чём доклад

- Параметрический и ad-hoc полиморфизм
- Тайпклассы и полиморфизм
- Возможности, которые даёт использование полиморфизма
  - Чужой код работает иначе
  - Наш код работает по-разному

Доклад не подразумевает полноту изложения

...и немного лукавит

Мономорфные функции

```
prepend :: Int -> [Int] -> [Int]
prepend a xs = a:xs
```

Мономорфные функции

```
prepend :: Int -> [Int] -> [Int]
prepend a xs = a:xs

append :: Int -> [Int] -> [Int]
append a []     = [a]
append a (x:xs) = x:(append a xs)
```

Мономорфные функции

```haskell
prepend :: Int -> [Int] -> [Int]
prepend a xs = a:xs

append :: Int -> [Int] -> [Int]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [Int] -> [Double] -> [(Int, Double)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Мономорфные функции

```haskell
prepend :: Int -> [Int] -> [Int]
prepend a xs = a:xs

append :: Int -> [Int] -> [Int]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [Int] -> [Double] -> [(Int, Double)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Важны ли конкретные типы `Int` и `Double`?

Мономорфные функции

```haskell
prepend :: String -> [String] -> [String]
prepend a xs = a:xs

append :: String -> [String] -> [String]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [String] -> [Bool] -> [(String, Bool)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Что именно они делают, не зависит от этих типов

Параметрический полиморфизм

```haskell
prepend :: a -> [a] -> [a]
prepend a xs = a:xs

append :: a -> [a] -> [a]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [a] -> [b] -> [(a, b)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Параметрический полиморфизм

```
prepend :: a -> [a] -> [a]
prepend a xs = a:xs

append :: a -> [a] -> [a]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [a] -> [b] -> [(a, b)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Type-level и value-level namespace’ы различны

Параметрический полиморфизм

```
prepend :: a -> [a] -> [a]
prepend a xs = a:xs

append :: a -> [a] -> [a]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: [a] -> [b] -> [(a, b)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Type-level и value-level namespace’ы различны

Только структура аргументов

Параметрический полиморфизм

```haskell
prepend :: forall a. a -> [a] -> [a]
prepend a xs = a:xs

append :: forall a. a -> [a] -> [a]
append a []     = [a]
append a (x:xs) = x:(append a xs)

zip' :: forall a b. [a] -> [b] -> [(a, b)]
zip' []     _      = []
zip' _      []     = []
zip' (a:as) (b:bs) = (a, b):(zip' as bs)
```

Параметрический полиморфизм

```haskell
head :: [a] -> Maybe a
head []    = Nothing
head (x:_) = Just x
```

Параметрический полиморфизм

```haskell
head :: [a] -> Maybe a
head []    = Nothing
head (x:_) = Just x
```

Даже `map`!

```haskell
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = (f x):(map xs)
```

Параметрический полиморфизм?

```haskell
maximum :: [a] -> Maybe a
```

Параметрический полиморфизм?

```haskell
maximum :: [a] -> Maybe a

maximumInt :: [Int] -> Maybe Int
maximumInt []     = Nothing
maximumInt (x:xs) = Just $ findMax xs x where
  findMax []     c = c
  findMax (x:xs) c = findMax xs $ if x < c then c else x

maximumDouble :: [Double] -> Maybe Double
maximumDouble []     = Nothing
maximumDouble (x:xs) = Just $ findMax xs x where
  findMax []     c = c
  findMax (x:xs) c = findMax xs $ if x < c then c else x
```

Параметрический полиморфизм?

```haskell
maximum :: [a] -> Maybe a

maximumInt :: [Int] -> Maybe Int
maximumInt []     = Nothing
maximumInt (x:xs) = Just $ findMax xs x where
  findMax []     c = c
  findMax (x:xs) c = findMax xs $ if x < c then c else x

maximumDouble :: [Double] -> Maybe Double
maximumDouble []     = Nothing
maximumDouble (x:xs) = Just $ findMax xs x where
  findMax []     c = c
  findMax (x:xs) c = findMax xs $ if x < c then c else x
```

Тела функций одинаковые

Но *ведут* функции себя немного по-разному

Параметрический полиморфизм?

```
fold :: [a] -> Maybe a
```

Параметрический полиморфизм?

```haskell
fold :: [a] -> Maybe a

foldInt :: [Int] -> Maybe Int
foldInt []     = Nothing
foldInt (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x +) $ fold xs

foldString :: [String] -> String
foldString []     = Nothing
foldString (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x ++) $ fold xs
```

Параметрический полиморфизм?

```haskell
fold :: [a] -> Maybe a

foldInt :: [Int] -> Maybe Int
foldInt []     = Nothing
foldInt (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x +) $ fold xs

foldString :: [String] -> String
foldString []     = Nothing
foldString (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x ++) $ fold xs
```

Функции принципиально делают одно и то же

Но *ведут* себя по-разному в зависимости от типа

Параметрический полиморфизм?

… от типа?

```
foldIntM :: [Int] -> Maybe Int
foldIntM []     = Nothing
foldIntM (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x +) $ fold xs

foldIntS :: [Int] -> Maybe Int
foldIntS []     = Nothing
foldIntS (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x *) $ fold xs
```

Два пути

- Оставаться в параметрическом полиморфизме

```
maximum :: forall a. (a->a->Bool) -> [a] -> Maybe a
fold    :: forall a. (a->a->a)    -> [a] -> Maybe a
```

Два пути

- Оставаться в параметрическом полиморфизме

```
maximum :: forall a. (a->a->Bool) -> [a] -> Maybe a
fold    :: forall a. (a->a->a)    -> [a] -> Maybe a
```

или даже

```
data Ord a = MkOrd (a -> a -> Bool)
maximum :: forall a. Ord a -> [a] -> Maybe a
```

Два пути

- Оставаться в параметрическом полиморфизме

```
maximum :: forall a. (a->a->Bool) -> [a] -> Maybe a
fold    :: forall a. (a->a->a)    -> [a] -> Maybe a
```

  или даже

```
data Ord a = MkOrd (a -> a -> Bool)
maximum :: forall a. Ord a -> [a] -> Maybe a
```

- Ad-hoc полиморфизм

```
maximum :: forall a ∈ ordered.    [a] -> Maybe a
fold    :: forall a ∈ squashable. [a] -> Maybe a
```

  ⇑ это **не** синтаксис Haskell ⇑

Налево пойдёшь — коня потеряешь

```haskell
data Ord a = MkOrd (a -> a -> Bool)
maximum :: Ord a -> [a] -> Maybe a

data Semi a = MkSemi (a -> a -> a)
fold :: Semi a -> [a] -> Maybe a
```

Налево пойдёшь — коня потеряешь

```haskell
data Ord a = MkOrd (a -> a -> Bool)
maximum :: Ord a -> [a] -> Maybe a

data Semi a = MkSemi (a -> a -> a)
fold :: Semi a -> [a] -> Maybe a


maxes :: Ord a -> [[a]] -> [Maybe a]
maxes ord = map $ maximum ord
```

Налево пойдёшь — коня потеряешь

```haskell
data Ord a = MkOrd (a -> a -> Bool)
maximum :: Ord a -> [a] -> Maybe a

data Semi a = MkSemi (a -> a -> a)
fold :: Semi a -> [a] -> Maybe a


maxes :: Ord a -> [[a]] -> [Maybe a]
maxes ord = map $ maximum ord

combMaxes :: Ord a -> Semi a -> [[a]] -> Maybe a
combMaxes o (MkSemi fs) = fold maybeSemi . maxes o
  where
    maybeSemi Nothing  y        = y
    maybeSemi x        Nothing  = x
    maybeSemi (Just x) (Just y) = Just $ fs x y
```

Налево пойдёшь — коня потеряешь

- Постоянно передавать
- Порядок важен
- Иерархия структур
- Переиспользование преобразований
- Полиморфизм на преобразованиях
- …

Тайпклассы

```
maximum :: forall a ∈ ordered. [a] -> Maybe a
```

⇑ это **не** синтаксис Haskell ⇑

Тайпклассы

```
maximum :: forall a ∈ ordered. [a] -> Maybe a
```

⇑ это **не** синтаксис Haskell ⇑

```
class Ord a where
  (<) :: a -> a -> Bool

maximum :: forall a. Ord a => [a] -> Maybe a
```

Тайпклассы

```
maximum :: forall a ∈ ordered. [a] -> Maybe a
```

⇑ это **не** синтаксис Haskell ⇑

```
class Ord a where
  (<) :: a -> a -> Bool

maximum :: forall a. Ord a => [a] -> Maybe a


maximum []     = Nothing
maximum (x:xs) = Just $ findMax xs x where
  findMax []     c = c
  findMax (x:xs) c = findMax xs $ if x < c then c else x
```

Тайпклассы

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

fold :: Semigroup a => [a] -> Maybe a
fold []     = Nothing
fold (x:xs) = Just $ foldNE x xs where
  foldNE x xs  = maybe x (x <>) $ fold xs
```

Тайпклассы: инстансы

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

fold :: Semigroup a => [a] -> Maybe a
```

Тайпклассы: инстансы

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

fold :: Semigroup a => [a] -> Maybe a


instance Semigroup Int where
  (<>) = (+)

instance Semigroup String where
  (<>) = (++)
```

Тайпклассы: инстансы

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

fold :: Semigroup a => [a] -> Maybe a


instance Semigroup Int where
  (<>) = (+)

instance Semigroup String where
  (<>) = (++)


fold [1, 2, 3]       -- gives Just 6
fold ["1", "2", "3"] -- gives Just "123"
```

Тайпклассы: условные инстансы

```haskell
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> y       = y
  x       <> Nothing = x
  Just x  <> Just y  = Just $ x <> y
```

Тайпклассы

Жизнь налаживается

```
maxes :: Ord a => [[a]] -> [Maybe a]
maxes = map maximum
```

Тайпклассы

Жизнь налаживается

```
maxes :: Ord a => [[a]] -> [Maybe a]
maxes = map maximum

combMaxes :: (Ord a, Semigroup a) => [[a]] -> Maybe a
combMaxes = fold . maxes
```

Тайпклассы

Жизнь налаживается

```haskell
maxes :: Ord a => [[a]] -> [Maybe a]
maxes = map maximum

combMaxes :: (Ord a, Semigroup a) => [[a]] -> Maybe a
combMaxes = fold . maxes
```

Но есть нюансы

- не так легко подставлять свои функции
- выбор тайпклассов должен быть удачным

Тайпклассы: прагматика и trade-off

Как управлять ad-hoc полиморфизмом?

Тайпклассы: прагматика и trade-off

Как управлять ad-hoc полиморфизмом?

```
newtype Last a = Last a
getLast :: Last a -> a
getLast (Last a) = a

newtype First a = First { getFirst :: a }
```

Тайпклассы: прагматика и trade-off

Как управлять ad-hoc полиморфизмом?

```
newtype Last a = Last a
getLast :: Last a -> a
getLast (Last a) = a

newtype First a = First { getFirst :: a }

instance Semigroup (Last a) where
  _ <> x = x
instance Semigroup (First a) where
  x <> _ = x
```

Тайпклассы: прагматика и trade-off

Как управлять ad-hoc полиморфизмом?

```haskell
newtype Last a = Last a
getLast :: Last a -> a
getLast (Last a) = a

newtype First a = First { getFirst :: a }

instance Semigroup (Last a) where
  _ <> x = x
instance Semigroup (First a) where
  x <> _ = x

head :: [a] -> Maybe a
head = fmap getFirst . fold . map First

last :: [a] -> Maybe a
last = fmap getLast . fold . map Last
```

Тайпклассы: функции по умолчанию

```haskell
class Ord a where
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool

  a > b = b < a
```

Тайпклассы: функции по умолчанию

```
class Ord a where
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool

  a > b = b < a

  (<=) :: a -> a -> Bool
```

Тайпклассы: функции по умолчанию

```haskell
class Ord a where
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool

  a > b = b < a

  (<=) :: a -> a -> Bool

  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  a <= b = a < b || a == b
```

Иерархия тайпклассов

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  a == b = not $ a /= b
  a /= b = not $ a == b

class Eq a => Ord a where
  (<)  :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool

  a <= b = a < b || a == b
```

Когерентность

```haskell
class Semigroup a => Monoid a where
  mempty :: a

class Semigroup a => ReversibleSemigroup a where
  rev :: a -> a
```

Когерентность

```
class Semigroup a => Monoid a where
  mempty :: a

class Semigroup a => ReversibleSemigroup a where
  rev :: a -> a
```

```
f :: (Monoid a, ReversibleSemigroup a) =
  rev mempty <> mempty
```

⇑ гарантируется, что <> *одна и та же* ⇑

Когерентность

Из-за этого иерархия инстансов повторяет иерархию классов

```haskell
instance Semigroup String where
  (<>) = (++)

instance Monoid String where
  mempty = ""
```

Поиск инстансов

Не *“если есть это, то инстанс вот”*, а *“вот инстанс, для него мне требуется”*

При поиске инстансов рассматривается только часть правее “=>”

Поиск инстансов

Не *“если есть это, то инстанс вот”*, а *“вот инстанс, для него мне требуется”*

При поиске инстансов рассматривается только часть правее “=>”

```
class Impossible a where
  magic :: forall b. a -> b

instance Impossible a => Semigroup a where
  a <> _ = magic a
```

Поиск инстансов

Не *“если есть это, то инстанс вот”*, а *“вот инстанс, для него мне требуется”*

При поиске инстансов рассматривается только часть правее “=>”

```
class Impossible a where
  magic :: forall b. a -> b

instance Impossible a => Semigroup a where
  a <> _ = magic a
```

Такой инстанс полностью выключает любой поиск для `Semigroup`

Типы высших порядков

kind *

`Int`, `[Int]`, `Maybe Int`, `Either String Int`, `Int -> String`

Типы высших порядков

kind *

Int, [Int], Maybe Int, Either String Int, Int -> String

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Типы высших порядков

kind *

Int, [Int], Maybe Int, Either String Int, Int -> String

```
class Semigroup a where
  (<>) :: a -> a -> a
```

kind * -> *

Maybe, Either String, (->) Int

Типы высших порядков

kind *

```
Int, [Int], Maybe Int, Either String Int, Int -> String

class Semigroup a where
  (<>) :: a -> a -> a
```

kind * -> *

```
Maybe, Either String, (->) Int

class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

О чём ещё можно продолжить

- instance resolution
  - overlapping instances
- много типов
  - multiparameter typeclasses
  - functional dependencies
  - type families
- вывод тайпклассов
  - derivable typeclasses
  - newtype deriving
  - deriving via
- ...

Пример: исходное состояние

```haskell
putStrLn :: String -> IO ()
getLine :: IO String

program :: IO ()
program = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hi, " ++ name)
```

Пример: исходное состояние

```
putStrLn :: String -> IO ()
getLine :: IO String

program :: IO ()
program = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hi, " ++ name)
```

Как протестировать, что функция делает ровно то, что там надо?

Пример: выделили абстракцию

```
class ConsoleIO m where
  putStrLn :: String -> m ()
  getLine :: m String
```

Пример: выделили абстракцию

```haskell
class ConsoleIO m where
  putStrLn :: String -> m ()
  getLine :: m String

program :: (Monad m, ConsoleIO m) => m ()
program = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hi, " ++ name)
```

Пример: выделили абстракцию

```haskell
class Monad m => ConsoleIO m where
  putStrLn :: String -> m ()
  getLine  :: m String

program :: ConsoleIO m => m ()
program = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hi, " ++ name)
```

Пример: инстансы для обычного запуска

```haskell
instance ConsoleIO IO where
  putStrLn = Prelude.putStrLn
  getLine  = Prelude.getLine
```

Пример: инстансы для обычного запуска

```
class MonadIO m where
  liftIO :: IO a -> m a
```

Пример: инстансы для обычного запуска

```haskell
class MonadIO m where
  liftIO :: IO a -> m a
```

```haskell
instance MonadIO m => ConsoleIO m where
  putStrLn = liftIO . Prelude.putStrLn
  getLine  = liftIO $ Prelude.getLine
```

Пример: инстансы для обычного запуска

```
class MonadIO m where
  liftIO :: IO a -> m a
```

```
instance MonadIO m => ConsoleIO m where
  putStrLn = liftIO . Prelude.putStrLn
  getLine  = liftIO $ Prelude.getLine
```

Скомпилируется?

Пример: инстансы для обычного запуска

```
class MonadIO m where
  liftIO :: IO a -> m a
```

```
instance MonadIO m => ConsoleIO m where
  putStrLn = liftIO . Prelude.putStrLn
  getLine  = liftIO $ Prelude.getLine
```

Скомпилируется?

Взлетит?

Пример: инстансы для обычного запуска

```haskell
class MonadIO m where
  liftIO :: IO a -> m a


newtype RealConsoleT m a = RealConsoleT
                           { runRealConsole :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)

instance MonadIO m => ConsoleIO (RealConsoleT m) where
  putStrLn = liftIO . putStrLn
  getLine  = liftIO $ getLine
```

Пример: инстансы для обычного запуска

```
class MonadIO m where
  liftIO :: IO a -> m a


newtype RealConsoleT m a = RealConsoleT
                            { runRealConsole :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)

instance MonadIO m => ConsoleIO (RealConsoleT m) where
  putStrLn = liftIO . putStrLn
  getLine  = liftIO $ getLine
```

Скомпилируется?

Пример: инстансы для обычного запуска

```haskell
class MonadIO m where
  liftIO :: IO a -> m a


newtype RealConsoleT m a = RealConsoleT
                             { runRealConsole :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)

instance MonadIO m => ConsoleIO (RealConsoleT m) where
  putStrLn = liftIO . putStrLn
  getLine  = liftIO $ getLine
```

Скомпилируется?

Взлетит?

Пример: инстансы для необычного запуска

```haskell
newtype NoConsoleT m a = NoConsoleT
                         { runNoConsole :: m a }
  deriving (Functor, Applicative, Monad, MonadIO)

instance Applicative m => ConsoleIO (NoConsoleT m) where
  putStrLn = const $ pure ()
  getLine  = pure ""
```

Пример: запуск программы

```
program :: ConsoleIO m => m ()

instance ConsoleIO IO where ...
instance MonadIO m => ConsoleIO (ReadConsoleT m) where ..
instance Applicative m => ConsoleIO (NoConsoleT m) where
```

Пример: запуск программы

```
program :: ConsoleIO m => m ()

instance ConsoleIO IO where ...
instance MonadIO m => ConsoleIO (ReadConsoleT m) where ..
instance Applicative m => ConsoleIO (NoConsoleT m) where



main = program
```

Пример: запуск программы

```
program :: ConsoleIO m => m ()

instance ConsoleIO IO where ...
instance MonadIO m => ConsoleIO (ReadConsoleT m) where ..
instance Applicative m => ConsoleIO (NoConsoleT m) where
```

```
main = program
```

```
main = runRealConsole program
```

Пример: запуск программы

```
program :: ConsoleIO m => m ()

instance ConsoleIO IO where ...
instance MonadIO m => ConsoleIO (ReadConsoleT m) where ..
instance Applicative m => ConsoleIO (NoConsoleT m) where
```

```
main = program
```

```
main = runRealConsole program
```

```
main = runNoConsole program
```

Пример: инстанс для тестирования

```haskell
data ScenarioAction = ExpectPrinting String
                    | ExpectReading String

newtype TestingConsoleT m a = TestingConsoleT
  (ExceptT String (StateT [ScenarioAction] m) a)
  deriving (Functor, Applicative, Monad,
    MonadState [ScenarioAction], MonadError String)

runTestingConsole :: [ScenarioAction]
                  -> TestingConsoleT m a
                  -> Either String a

instance Monad m => ConsoleIO (TestingConsoleT m) where
  ...
```

Пример: инстанс для тестирования

```haskell
instance Monad m => ConsoleIO (TestingConsoleT m) where
  putStrLn s = do
    sc <- get
    (curr, sc') <- case sc of
      [] -> throwError "Scenario ended, but putStrLn"
      (x:xs) -> pure (x, xs)
    put sc'
    case curr of
      ExpectPrinting exp ->
        when (s /= exp) $ throwError "Wrong is printed"
      ExpectReader _ -> throwError "Reading is expected"

  getLine = ...
```

Пример: запуск для тестирования

```
scenario =
  [ ExpectPrintlng "What is your name?"
  , ExpectReading "Denis"
  , ExpectPrinting "Hi, Denis" ]

runTestingConsole scenario program -- gives Right ()
```

Пример: запуск для тестирования

```
scenario =
  [ ExpectPrintlng "What is your name?"
  , ExpectReading "Denis"
  , ExpectPrinting "Hi, Denis" ]

runTestingConsole scenario program -- gives Right ()
```

—

В hpec можно это использовать так:

```
spec = describe "Hello program" do
         it "asks and responds" do
           testRun `shouldBe` Right ()
  where
    testRun = runTestingConsole scenario program
```

Спасибо