# Рекурсивные схемы

## Рекурсия в типах и коде

Денис Буздалов

18 декабря 2019

Раздел 1

# Вводный

# Нам потребуются

- Алгебраические типы данных

# Нам потребуются

- Алгебраические типы данных

```haskell
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

# Нам потребуются

- Алгебраические типы данных

```
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

# Нам потребуются

- Алгебраические типы данных

```haskell
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```haskell
type XT a =    (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)
```

# Нам потребуются

- Алгебраические типы данных

```
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```
type XT a =   (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)

class XC a where
  k :: a
  l :: Int → a
  m :: String → Int → a
```

# Нам потребуются

- Алгебраические типы данных

```haskell
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```haskell
type XT a =   (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)

class XC a where
  k :: a
  l :: Int → a
  m :: String → Int → a

data YT r a = (a, Int → a, r → a, r → Int → a)
```

# Нам потребуются

- Алгебраические типы данных

```
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```
type XT a =   (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)

class XC a where
  k :: a
  l :: Int → a
  m :: String → Int → a

data YT r a = (a, Int → a, r → a, r → Int → a)
```

- Изоморфизмы

# Нам потребуются

- Алгебраические типы данных

```
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```
type XT a =   (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)

class XC a where
  k :: a
  l :: Int → a
  m :: String → Int → a

data YT r a = (a, Int → a, r → a, r → Int → a)
```

- Изоморфизмы

$$X \rightleftharpoons (X \to a) \to a \rightleftharpoons XT\ a \to a \rightleftharpoons XI\ a \to a \rightleftharpoons XC\ a \Rightarrow a$$

# Нам потребуются

- Алгебраические типы данных

```haskell
data X = K | L Int | M String Int
data Y r = D | E Int | F r | G r Int
```

- Кодирование Чёрча

```haskell
type XT a =   (a, Int → a, String → Int → a)
data XI a = XI a (Int → a) (String → Int → a)

class XC a where
  k :: a
  l :: Int → a
  m :: String → Int → a

data YT r a = (a, Int → a, r → a, r → Int → a)
```

- Изоморфизмы

```haskell
X ⇌ (X → a) → a ⇌ XT a → a ⇌ XI a → a ⇌ XC a ⇒ a
```

```haskell
a → b → c        ⇌        (a, b) → c
```

# Нам также потребуются

- Списки

```
data [a] = [] | (:) a [a]
-- List a = Nil | Cons a (List a)
```

# Нам также потребуются

- Списки

```haskell
data [a] = [] | (:) a [a]
-- List a = Nil | Cons a (List a)
```

- Функторы

```haskell
class Functor (f :: * → *) where
  fmap :: (a → b) → f a → f b
```

# Нам также потребуются

- Списки

```
data [a] = [] | (:) a [a]
-- List a = Nil | Cons a (List a)
```

- Функторы

```
class Functor (f :: * → *) where
  fmap :: (a → b) → f a → f b
```

- Лукавство и колукавство ;-)

# О чём доклад

- Параллели и аналогии в программировании
- Рекурсия как таковая
  - “Обычная” рекурсия
  - Необычное представление “обычной” рекурсии
  - Рекурсия в данных
  - Аналогия необычного представления
- Списки и деревья
- Рекурсивные схемы
  - Вниз: катаморфизм, зигоморфизм, параморфизм
  - Вверх: анаморфизм, апоморфизм
  - Туда-сюда: хиломорфизм, метаморфизм

# О чём доклад

- **Всё**, что будет представлено, **можно** реализовать по-другому
- Что будет представлено — лишь инструмент, позволяющий
  - облегчать выполнение некоторых задач
  - (при должном владении) упрощать рассуждения о сложных структурах
  - писать код, в котором сложней допустить ошибку

Раздел 2

# Рекурсия

# Рекурсивные вычисления

```haskell
fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

# Рекурсивные вычисления

```haskell
fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

Внешняя псевдорекурсия

```haskell
factU :: (Integer → Integer) → Integer → Integer
factU _   0 = 1
factU rec n = n * rec (n - 1)
```

# Рекурсивные вычисления

```haskell
nmlist :: Integer → Integer → [Integer]
nmlist n m | n > m     = []
           | n == m    = [n]
           | otherwise = n : nmlist (n + 1) m
```

# Рекурсивные вычисления

```haskell
nmlist :: Integer → Integer → [Integer]
nmlist n m | n > m     = []
           | n == m    = [n]
           | otherwise = n : nmlist (n + 1) m
```

```haskell
nmlistU :: (Integer → Integer → [Integer])
        →  Integer → Integer → [Integer]
nmlistU rec n m | n > m     = []
                | n == m    = [n]
                | otherwise = n : rec (n + 1) m
```

# Рекурсивные вычисления

```haskell
nmlist :: Integer → Integer → [Integer]
nmlist n m | n > m     = []
           | n == m    = [n]
           | otherwise = n : nmlist (n + 1) m
```

```haskell
nmlistU :: (Integer → Integer → [Integer])
        →  Integer → Integer → [Integer]
nmlistU rec n m | n > m     = []
                | n == m    = [n]
                | otherwise = n : rec (n + 1) m
```

Отделили логику от рекурсивного вызова

*Кто бы вызывал?*

# Немного волшебства

```haskell
fix :: (a → a) → a
fix f = x
  where x = f x
```

# Немного волшебства

```haskell
fix :: (a → a) → a
fix f = x
  where x = f x
```

$$\texttt{fix f} \simeq f(f(f\ldots))$$

# Немного волшебства

```haskell
fix :: (a → a) → a
fix f = x
  where x = f x
```

$$\texttt{fix f} \quad \simeq \quad f(f(f\ldots))$$

А нам нужна была функция

# Немного волшебства

```haskell
fix :: (a → a) → a
fix f = x
  where x = f x
```

$$\texttt{fix f} \simeq f(f(f\ldots))$$

А нам нужна была функция

*примем* `a` *за* `(b → r)`

# Немного волшебства

```haskell
fix :: (a -> a) -> a
fix f = x
  where x = f x
```

$$\texttt{fix f} \quad \simeq \quad f(f(f\ldots))$$

А нам нужна была функция

*примем* `a` *за* `(b -> r)`

```haskell
fix :: (a -> a) -> a        -- ((b -> r) -> (b -> r)) -> (b -> r)
```

# Немного волшебства

```haskell
fix :: (a → a) → a
fix f = x
  where x = f x
```

$$\texttt{fix f} \quad \simeq \quad f(f(f\ldots))$$

А нам нужна была функция

*примем* `a` *за* `(b → r)`

```haskell
fix :: (a → a) → a        -- ((b → r) → (b → r)) → (b → r)
fix :: (a → a) → a        -- ((b → r) → b → r) → (b → r)
```

# fix your recursion

```
fix :: (a → a) → a
       ((b → r) → b → r) → (b → r)          -- a ~ (b → r)
```

# fix your recursion

```haskell
fix :: (a → a) → a
       ((b → r) → b → r) → (b → r)          -- a ~ (b → r)
```

```haskell
fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

# fix your recursion

```haskell
fix :: (a → a) → a
       ((b → r) → b → r) → (b → r)          -- a ~ (b → r)
```

```haskell
fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

```haskell
fact' :: Integer → Integer
fact' = fix $ \rec n → case n of
  0 → 1
  n → n * rec (n - 1)
```

# fix your recursion

```
fix :: (a → a) → a
       ((b→c→r) → b → c → r) → (b→c→r)     -- a ~ (b→c→r)
```

# fix your recursion

```haskell
fix :: (a → a) → a
       ((b→c→r) → b → c → r) → (b→c→r)     -- a ~ (b→c→r)
```

```haskell
nmlist :: Integer → Integer → [Integer]
nmlist n m | n > m     = []
           | n == m    = [n]
           | otherwise = n : nmlist (n + 1) m
```

# fix your recursion

```haskell
fix :: (a → a) → a
       ((b→c→r) → b → c → r) → (b→c→r)    -- a ~ (b→c→r)
```

```haskell
nmlist :: Integer → Integer → [Integer]
nmlist n m | n > m     = []
           | n == m    = [n]
           | otherwise = n : nmlist (n + 1) m
```

```haskell
nmlist' :: Integer → Integer → [Integer]
nmlist' = fix $ \rec n m → case () of
  () | n > m     → []
     | n == m    → [n]
     | otherwise → n : nmlist (n + 1) m
```

## Раздел 3

# Теперь о данных

# Данные могут быть рекурсивными

```haskell
data [a] = [] | (:) a [a]

data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)

data WideTree a = WLeaf a | WNode [WideTree a]

data JsonValue = JsonNull
               | JsonBool   Bool
               | JsonNumber Rational
               | JsonString String
               | JsonArray  [JsonValue]
               | JsonObject [(String, JsonValue)]
```

# Рекурсивные данные → рекурсивные функции

```haskell
data [a] = [] | (:) a [a]
```

```haskell
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x:tl else tl
  where tl = filter f xs
```

```haskell
foldr :: (a → b → b) → b → [a] → b
foldr _ z []     = z
foldr f z (x:xs) = f x $ foldr f z xs
```

# Рекурсивные данные → рекурсивные функции

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
```

```haskell
showBT :: Show a ⇒ BinTree a → String
showBT (BLeaf a) = show a
showBT (BNode l r) = "{ left: " ++ showBT l ++
                     ", right: " ++ showBT r ++ "}"
```

```haskell
foldBT :: (b → b → b) → (a → b) → BinTree a → b
foldBT _  lf (BLeaf a)   = lf a
foldBT nf lf (BNode l r) = nf (foldBT nf lf l) (foldBT nf lf r)
```

# Рекурсивные данные → рекурсивные функции

```haskell
data JsonValue = JsonNull | JsonBool Bool | JsonNumber Rational
               | JsonString String | JsonArray [JsonValue]
               | JsonObject [(String, JsonValue)]
```

```haskell
jsToStr :: JsonValue → String
jsToStr JsonNull          = "null"
jsToStr (JsonBool b)      = show b
jsToStr (JsonNumber n)    = show n
jsToStr (JsonString s)    = s
jsToStr (JsonArray vs)    =
  "[" ++ (intercalate ", " . map jsToStr $ vs) ++ "]"
jsToStr (JsonObject svs) =
  "{" ++ (intercalate "\n" . fmap showPair $ svs) ++ "}"
  where showPair (s, o) = s ++ " : " ++ jsToStr o
```

# Рекурсивные данные → рекурсивные функции

```haskell
data JsonValue = JsonNull | JsonBool Bool | JsonNumber Rational
               | JsonString String | JsonArray [JsonValue]
               | JsonObject [(String, JsonValue)]
```

```haskell
enrat :: JsonValue → JsonValue
enrat x@JsonNull       = x
enrat x@(JsonBool _)   = x
enrat x@(JsonNumber _) = x
enrat x@(JsonString s) = maybe x JsonNumber $ readMaybe s
enrat (JsonArray vs)   = JsonArray $ fmap enrat vs
enrat (JsonObject svs) = JsonObject $ fmap (fmap enrat) svs
```

# Повторяющиеся паттерны

```haskell
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x:tl else tl
  where tl = filter f xs

prod :: Num a ⇒ [a] → a
prod []     = 1
prod (x:xs) = x * prod xs
```

# Повторяющиеся паттерны

```
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x:tl else tl
  where tl = filter f xs
```

```
prod :: Num a ⇒ [a] → a
prod []     = 1
prod (x:xs) = x * prod xs
```

```
foldr :: (a → b → b) → b → [a] → b
```

# Повторяющиеся паттерны

```haskell
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x:tl else tl
  where tl = filter f xs
```

```haskell
prod :: Num a ⇒ [a] → a
prod []     = 1
prod (x:xs) = x * prod xs
```

```haskell
foldr :: (a → b → b) → b → [a] → b
```

```haskell
prod = foldr (*) 1
filter f = foldr (\x tl → if f x then x:tl else tl) []
```

# Повторяющиеся паттерны

```haskell
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x:tl else tl
  where tl = filter f xs

prod :: Num a ⇒ [a] → a
prod []     = 1
prod (x:xs) = x * prod xs
```

```haskell
foldr :: (a → b → b) → b → [a] → b

prod = foldr (*) 1
filter f = foldr (\x tl → if f x then x:tl else tl) []
```

Все функции вида `[a] → ...` представимы в виде `f . foldr ...`

Позволяет разделить логику обхода и обработки содержимого рекурсивных данных

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
```

$$b^{[a]} = x \qquad\qquad (x^b)^{(b^b)^a} \to x^{(b^b)^a b}$$

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
```

$$b^{[a]} = x \qquad\qquad (x^b)^{(b^b)^a} \to x^{(b^b)^a b}$$

```
foldr :: ((a → b → b), b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
```

# Истоки универсальности `foldr`

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
```

$$b^{[a]} = x \qquad (x^b)^{(b^b)^a} \to x^{(b^b)^a b} \to x^{(b^b)^a b^1}$$

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
```

```
foldr :: ((a → b → b), b) → [a] → b
```

$$b^{[a]} = x \qquad\qquad (x^b)^{(b^b)^a} \to x^{(b^b)^a b} \to x^{(b^b)^a b^1}$$

```
foldr :: ((a → b → b), () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
```

$$b^{[a]} = x \qquad (x^b)^{(b^b)^a} \to \cdots \to x^{(b^b)^a b^1} \to x^{b^{a\,b} b^1}$$

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
```

$$b^{[a]} = x \qquad (x^b)^{(b^b)^a} \to \dots \to x^{(b^b)^a b^1} \to x^{b^{ab} b^1}$$

```
foldr :: (((a, b) → b), () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
```

$$b^{[a]} = x \qquad (x^b)^{(b^b)^a} \to \cdots \to x^{b^{a\,b}b^1} \to x^{b^{a\,b+1}}$$

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
```

$$b^{[a]} = x \qquad (x^b)^{(b^b)^a} \to \cdots \to x^{b^{ab}b^1} \to x^{b^{ab+1}}$$

```
foldr :: (Either (a, b) () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
foldr :: (Either (a, b) () → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
foldr :: (Either (a, b) () → b) → [a] → b
foldr :: (Maybe (a, b) → b) → [a] → b
```

# Истоки универсальности foldr

```haskell
foldr :: (a → b → b) → b → [a] → b
foldr :: ((a → b → b), b) → [a] → b
foldr :: ((a → b → b), () → b) → [a] → b
foldr :: (((a, b) → b), () → b) → [a] → b
foldr :: (Either (a, b) () → b) → [a] → b
foldr :: (Maybe (a, b) → b) → [a] → b
```

```haskell
newtype ListF a b = Maybe (a, b)
foldr :: (ListF a b → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b

foldr :: ((a → b → b), b) → [a] → b

foldr :: ((a → b → b), () → b) → [a] → b

foldr :: (((a, b) → b), () → b) → [a] → b

foldr :: (Either (a, b) () → b) → [a] → b

foldr :: (Maybe (a, b) → b) → [a] → b

newtype ListF a b = Maybe (a, b)
foldr :: (ListF a b → b) → [a] → b

data ListF a b = Nil | Cons a b
foldr :: (ListF a b → b) → [a] → b
```

# Истоки универсальности foldr

```
foldr :: (a → b → b) → b → [a] → b

foldr :: ((a → b → b), b) → [a] → b

foldr :: ((a → b → b), () → b) → [a] → b

foldr :: (((a, b) → b), () → b) → [a] → b

foldr :: (Either (a, b) () → b) → [a] → b

foldr :: (Maybe (a, b) → b) → [a] → b

newtype ListF a b = Maybe (a, b)
foldr :: (ListF a b → b) → [a] → b

data ListF a b = Nil | Cons a b
foldr :: (ListF a b → b) → [a] → b

    foldr :: [a] → (ListF a b → b) → b
```

# Истоки универсальности foldr

```haskell
foldr :: (a → b → b) → b → [a] → b

foldr :: ((a → b → b), b) → [a] → b

foldr :: ((a → b → b), () → b) → [a] → b

foldr :: (((a, b) → b), () → b) → [a] → b

foldr :: (Either (a, b) () → b) → [a] → b

foldr :: (Maybe (a, b) → b) → [a] → b

newtype ListF a b = Maybe (a, b)
foldr :: (ListF a b → b) → [a] → b

data ListF a b = Nil | Cons a b
foldr :: (ListF a b → b) → [a] → b
```

```haskell
    foldr :: [a] → (ListF a b → b) → b
          X   ⇌    (X → a) → a
```

*с поправкой на рекурсию*

# foldr: fix для списков

```
foldr :: (ListF a b → b) → [a] → b
data ListF a b = Nil | Cons a b
```

# foldr: fix для списков

```
foldr :: (ListF a b → b) → [a] → b
data ListF a b = Nil | Cons a b
data [a]       = []  | (:)  a [a]
```

# foldr: fix для списков

```
foldr :: (ListF a b → b) → [a] → b
data ListF a b = Nil | Cons a b
data [a]       = []  | (:)  a [a]
```

*Вспомним*

```
factU :: (Integer → Integer) → Integer → Integer
factU _   0 = 1
factU rec n = n * rec (n - 1)
```

```
fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)
```

# foldr: fix для списков

```haskell
foldr :: (ListF a b → b) → [a] → b

data ListF a b = Nil | Cons a b

data [a]       = []  | (:)  a [a]
```

*Вспомним*

```haskell
factU :: (Integer → Integer) → Integer → Integer
factU _   0 = 1
factU rec n = n * rec (n - 1)

fact :: Integer → Integer
fact 0 = 1
fact n = n * fact (n - 1)

fix :: (a → a) → a             -- ((a → b) → a → b) → a → b

fact' :: Integer → Integer
fact' = fix factU
```

# Fix your data

```haskell
fix :: (a → a) → a                                    -- recap
```

$$\texttt{fix f} \simeq \texttt{f (f f (...))}$$

# Fix your data

```
fix :: (a → a) → a                    -- recap

            fix f  ≃  f (f f ( … ))

newtype Fix (f :: * → *) = Fix (f (Fix f))
```

# Fix your data

```haskell
    fix :: (a → a) → a                                    -- recap
```

```
                    fix f  ≃  f (f f ( ... ))
```

```haskell
newtype Fix (f :: * → *) = Fix (f (Fix f))
```

```
            Fix f  ≃  Fix (f (Fix (f (Fix (f  ... )))))
```

# Fix your data

```
fix :: (a → a) → a                            -- recap
```

```
              fix f  ≃  f (f f ( ... ))
```

```
newtype Fix (f :: * → *) = Fix (f (Fix f))
         Fix f  ≃  Fix (f (Fix (f (Fix (f ... )))))
```

```
data [a]        = []  | (:)  a [a]
data ListF a b = Nil | Cons a b
```

$$\boxed{\texttt{[a]} \rightleftharpoons \texttt{Fix (ListF a)}}$$

# Fix your data

```
fix :: (a → a) → a                                 -- recap
```

```
fix f  ≃  f (f f ( … ))
```

```
newtype Fix (f :: * → *) = Fix (f (Fix f))
```

```
Fix f  ≃  Fix (f (Fix (f (Fix (f  … )))))
```

```
data [a]        = []  | (:)  a [a]
data ListF a b = Nil | Cons a b
```

$$\boxed{\texttt{[a]} \rightleftharpoons \texttt{Fix (ListF a)}}$$

```
foldr :: (ListF a b → b) → [a] → b
```

# Fix your data

```
fix :: (a → a) → a                                   -- recap
```

```
fix f  ≃  f (f f ( … ))
```

```
newtype Fix (f :: * → *) = Fix (f (Fix f))
```

```
Fix f  ≃  Fix (f (Fix (f (Fix (f  … )))))
```

```
data [a]       = []  | (:)  a [a]
data ListF a b = Nil | Cons a b
```

```
[a]  ⇌  Fix (ListF a)
```

```
foldr :: (ListF a b → b) → [a] → b
foldr :: (ListF a b → b) → Fix (ListF a) → b
```

# Замечание про кодирование Чёрча

```haskell
data ListF a b = Nil | Cons a b
foldr :: (ListF a b → b) → [a] → b
```

# Замечание про кодирование Чёрча

```
data ListF a b = Nil | Cons a b

foldr :: (ListF a b → b) → [a] → b


class ListC a b where
  nil  :: b
  cons :: a → b → b
```

# Замечание про кодирование Чёрча

```haskell
data ListF a b = Nil | Cons a b

foldr :: (ListF a b → b) → [a] → b


class ListC a b where
  nil  :: b
  cons :: a → b → b

foldr :: ListC a b ⇒ [a] → b
```

# Замечание про кодирование Чёрча

```haskell
data ListF a b = Nil | Cons a b

foldr :: (ListF a b → b) → [a] → b
```

```haskell
class ListC a b where
  nil  :: b
  cons :: a → b → b

foldr :: ListC a b ⇒ [a] → b
```

```haskell
class Monoid a where
  mempty :: a
  (<>)   :: a → a → a
fold :: Monoid a ⇒ [a] → a
```

# Замечание про кодирование Чёрча

```
data ListF a b = Nil | Cons a b
foldr :: (ListF a b → b) → [a] → b
```

```
class ListC a b where
  nil  :: b
  cons :: a → b → b
foldr :: ListC a b ⇒ [a] → b
```

```
class Monoid a where
  mempty :: a
  (<>)   :: a → a → a
fold :: Monoid a ⇒ [a] → a
```

*правда, не забывайте про лукавство*

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
data BinTreeF a r = BLeaf a | BNode r r
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
data BinTreeF a r = BLeaf a | BNode r r
```

```haskell
data BinTree' a = BLeaf a | BNode (BinTree a) a (BinTree a)
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
data BinTreeF a r = BLeaf a | BNode r r
```

```haskell
data BinTree' a = BLeaf a | BNode (BinTree a) a (BinTree a)
data BinTreeF' a r = BLeaf a | BNode r a r
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)

data BinTreeF a r = BLeaf a | BNode r r
```

```haskell
data BinTree' a = BLeaf a | BNode (BinTree a) a (BinTree a)
data BinTreeF' a r = BLeaf a | BNode r a r
```

```haskell
foldBT :: (b → b → b) → (a → b) → BinTree a → b
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)

data BinTreeF a r = BLeaf a | BNode r r
```

```haskell
data BinTree' a = BLeaf a | BNode (BinTree a) a (BinTree a)
data BinTreeF' a r = BLeaf a | BNode r a r
```

```haskell
foldBT :: (b → b → b) → (a → b) → BinTree a → b

foldBT :: (BinTreeF a b → b) → BinTree a → b
```

# Работает ли тот же приём с другими?

```haskell
data BinTree a = BLeaf a | BNode (BinTree a) (BinTree a)
data BinTreeF a r = BLeaf a | BNode r r
```

```haskell
data BinTree' a = BLeaf a | BNode (BinTree a) a (BinTree a)
data BinTreeF' a r = BLeaf a | BNode r a r
```

```haskell
foldBT :: (b → b → b) → (a → b) → BinTree a → b
foldBT :: (BinTreeF a b → b) → BinTree a → b
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
```

# Работает ли тот же приём с другими?

```haskell
data BinTreeF a r = BLeaf a | BNode r r

foldBT :: (BinTreeF a b → b) → BinTree a → b
                                                        -- или
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
```

# Работает ли тот же приём с другими?

```haskell
data BinTreeF a r = BLeaf a | BNode r r

foldBT :: (BinTreeF a b → b) → BinTree a → b
                                                        -- или
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b


showAlg :: Show a ⇒ BinTreeF a String → String
showAlg (BLeafF a)   = show a
showAlg (BNodeF l r) = "{ left: " ++ l ++
                       ", right: " ++ r ++ "}"

depthAlg :: BinTreeF a Integer → Integer
depthAlg (BLeafF _)   = 0
depthAlg (BNodeF l r) = 1 + max l r
```

# Даже числа!

```haskell
data Nat = Z | S Nat
```

# Даже числа!

```haskell
data Nat = Z | S Nat

natRec :: (a → a) → a → Nat → a
```

# Даже числа!

```haskell
data Nat = Z | S Nat
natRec :: (a → a) → a → Nat → a
```

```haskell
data NatF r = Z | S r
```

# Даже числа!

```haskell
data Nat = Z | S Nat

natRec :: (a → a) → a → Nat → a
```

```haskell
data NatF r = Z | S r
```

$$\texttt{Nat} \rightleftharpoons \texttt{Fix NatF}$$

# Даже числа!

```
data Nat = Z | S Nat
natRec :: (a → a) → a → Nat → a
```

```
data NatF r = Z | S r
```

$$\texttt{Nat} \rightleftharpoons \texttt{Fix NatF}$$

```
natRec :: (NatF a → a) → Nat → a
```

# Даже числа!

```haskell
data Nat = Z | S Nat
natRec :: (a → a) → a → Nat → a
```

```haskell
data NatF r = Z | S r
```

$$\texttt{Nat} \rightleftharpoons \texttt{Fix NatF}$$

```haskell
natRec :: (NatF a → a) → Nat → a
natRec :: (NatF a → a) → Fix NatF → a
```

Раздел 4

## Собственно, рекурсивные схемы

# Свёртки

```
foldr  :: (ListF a b    → b) → Fix (ListF a)    → b
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
natRec :: (NatF b       → b) → Fix NatF         → b
```

# Катаморфизм

```
foldr  :: (ListF a b    → b) → Fix (ListF a)    → b
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
natRec :: (NatF b       → b) → Fix NatF         → b
```

```
cata :: Functor dataF ⇒ (dataF b → b) → Fix dataF → b
```

# Катаморфизм

```
foldr   :: (ListF a b    → b) → Fix (ListF a)    → b
foldBT  :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
natRec  :: (NatF b       → b) → Fix NatF         → b
```

```
cata :: Functor dataF ⇒ (dataF b → b) → Fix dataF → b
```

> *Катаморфизм — изменение горных пород в верхней зоне земной коры под влиянием воздействия атмосферы и циркуляции подземных вод*

# Катаморфизм

```
foldr  :: (ListF a b    → b) → Fix (ListF a)    → b
foldBT :: (BinTreeF a b → b) → Fix (BinTreeF a) → b
natRec :: (NatF b       → b) → Fix NatF         → b
```

```
cata :: Functor dataF ⇒ (dataF b → b) → Fix dataF → b
```

*Катаморфизм — обобщение операции свёртки на произвольные алгебраические типы данных, описанные при помощи начальных алгебр*

# ~~Катаморфизм~~ (вспомним пример)

```haskell
data JsonValue = JsonNull | JsonBool Bool | JsonNumber Rational
               | JsonString String | JsonArray [JsonValue]
               | JsonObject [(String, JsonValue)]     -- recap
```

```haskell
enrat :: JsonValue -> JsonValue
enrat x@JsonNull       = x
enrat x@(JsonBool _)   = x
enrat x@(JsonNumber _) = x
enrat x@(JsonString s) = maybe x JsonNumber $ readMaybe s
enrat (JsonArray vs)   = JsonArray $ fmap enrat vs
enrat (JsonObject svs) = JsonObject $ fmap (fmap enrat) svs
```

# Катаморфизм

```haskell
data JsonValueF r = JsonNullF| JsonBoolF Bool | JsonNumberF Rational
                  | JsonStringF String | JsonArrayF [r]
                  | JsonObjectF [(String, r)]
```

```haskell
enrat :: Fix JsonValueF → Fix JsonValueF
enrat = cata $ Fix . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x
```

Раздел 5

# Базовые функторы: автоматизация и упрощения

# Базовый функтор

```haskell
data XF r = X1F Int | X2F String (Either Int r)
```

# Базовый функтор

```haskell
data XF r = X1F Int | X2F String (Either Int r)
instance Functor XF where
  fmap _ (X1F i)    = X1F i
  fmap f (X2F s ei) = X2F s $ fmap f ei
```

# Базовый функтор

```haskell
data XF r = X1F Int | X2F String (Either Int r)
instance Functor XF where
  fmap _ (X1F i)    = X1F i
  fmap f (X2F s ei) = X2F s $ fmap f ei
```

```haskell
data XF r = X1F Int | X2F String (Either Int r)
  deriving (Functor)
```

# Базовый функтор

```haskell
data XF r = X1F Int | X2F String (Either Int r)
instance Functor XF where
  fmap _ (X1F i)     = X1F i
  fmap f (X2F s ei) = X2F s $ fmap f ei
```

```haskell
data XF r = X1F Int | X2F String (Either Int r)
  deriving (Functor)
```

```haskell
data YF r = Y1F Int | Y2F String (Either r Int)
```

# Базовый функтор

```haskell
data XF r = X1F Int | X2F String (Either Int r)
instance Functor XF where
  fmap _ (X1F i)      = X1F i
  fmap f (X2F s ei) = X2F s $ fmap f ei
```

```haskell
data XF r = X1F Int | X2F String (Either Int r)
  deriving (Functor)
```

```haskell
data YF r = Y1F Int | Y2F String (Either r Int)
```

```haskell
instance Functor YF where
  fmap _ (Y1F i)      = Y1F i
  fmap f (Y2F s ei) = Y2F s $ mapLeft f ei
```

# Базовый функтор

```haskell
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
```

# Базовый функтор

```haskell
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
    deriving (Functor)
```

# Базовый функтор

```haskell
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor)
```

```haskell
data ZF r = Z1F Int | Z2F String (Either r (Int, r))
```

# Базовый функтор

```haskell
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
    deriving (Functor)
```

```haskell
data ZF r = Z1F Int | Z2F String (Either r (Int, r))

instance Functor ZF where
  fmap _ (Z1F i)    = Z1F i
  fmap f (Z2F s ei) = Z2F s $ mapLeft f . fmap (fmap f) $ ei
```

# Автоматическая генерация базового функтора

# Автоматическая генерация базового функтора

```
{-# LANGUAGE TemplateHaskell, DeriveFunctor -#}
{-# LANGUAGE DeriveFoldable, DeriveTraversable #-}
import Data.Functor.Foldable.TH (makeBaseFunctor)
```

# Автоматическая генерация базового функтора

```haskell
    {-# LANGUAGE TemplateHaskell, DeriveFunctor -#}
    {-# LANGUAGE DeriveFoldable, DeriveTraversable #-}
    import Data.Functor.Foldable.TH (makeBaseFunctor)


data Z = Z1 Int | Z2 String (Either Char (Int, Z))
```

# Автоматическая генерация базового функтора

```haskell
{-# LANGUAGE TemplateHaskell, DeriveFunctor -#}
{-# LANGUAGE DeriveFoldable, DeriveTraversable #-}
import Data.Functor.Foldable.TH (makeBaseFunctor)
```

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z
```

# Автоматическая генерация базового функтора

```haskell
    {-# LANGUAGE TemplateHaskell, DeriveFunctor -#}
    {-# LANGUAGE DeriveFoldable, DeriveTraversable #-}
    import Data.Functor.Foldable.TH (makeBaseFunctor)


data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)
```

# Автоматическая генерация базового функтора

```haskell
    {-# LANGUAGE TemplateHaskell, DeriveFunctor -#}
    {-# LANGUAGE DeriveFoldable, DeriveTraversable #-}
    import Data.Functor.Foldable.TH (makeBaseFunctor)


data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)


traverse :: (Traversable t, Applicative f)
         ⇒ (a → f b) → t a → f (t b)
foldr :: Foldable f ⇒ (a → b → b) → b → t a → b
```

# Так ли нужен Fix, если DataF сгенерирован из Data?

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

# Так ли нужен Fix, если DataF сгенерирован из Data?

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

$$\texttt{Data} \rightleftharpoons \texttt{DataF Data} \rightleftharpoons \texttt{Fix DataF}$$

# Так ли нужен Fix, если DataF сгенерирован из Data?

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

$$\texttt{Data} \rightleftharpoons \texttt{DataF Data} \rightleftharpoons \texttt{Fix DataF}$$

```
cata :: Functor dataF ⇒ (dataF a → a) → Data → a        -- ???
```

# Так ли нужен `Fix`, если `DataF` сгенерирован из `Data`?

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

$$\texttt{Data} \rightleftharpoons \texttt{DataF Data} \rightleftharpoons \texttt{Fix DataF}$$

```
cata :: Functor dataF ⇒ (dataF a → a) → Data → a       -- ???
```

Как мы можем выразить зависимость между `Data` и `DataF`?

# Так ли нужен Fix, если DataF сгенерирован из Data?

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

$$\texttt{Data} \rightleftharpoons \texttt{DataF Data} \rightleftharpoons \texttt{Fix DataF}$$

```
cata :: Functor dataF ⇒ (dataF a → a) → Data → a        -- ???
```

Как мы можем выразить зависимость между Data и DataF?

```
class Recursive' (t :: *) (base :: * → *) where
  project' :: t → base t

cata :: Recursive' t base ⇒ (base a → a) → t → a
```

# Так ли нужен Fix, если DataF сгенерирован из Data?

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
```

$$\texttt{Data} \rightleftharpoons \texttt{DataF Data} \rightleftharpoons \texttt{Fix DataF}$$

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Data → a        -- ???
```

Как мы можем выразить зависимость между Data и DataF?

```haskell
class Recursive' (t :: *) (base :: * → *) where
  project' :: t → base t

cata :: Recursive' t base ⇒ (base a → a) → t → a
```

```haskell
type family Base (t :: *) :: * → *
class Recursive (t :: *) where
  project :: t → (Base t) t

cata :: Recursive t ⇒ (Base t a → a) → t → a
```

# Базовые функторы

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a
```

# Базовые функторы

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a
```

# Базовые функторы

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

type instance Base JsonValue = JsonValueF
```

# Базовые функторы

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

type instance Base JsonValue = JsonValueF

type instance Base (Fix f) = -- ???
```

# Базовые функторы

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

type instance Base JsonValue = JsonValueF

type instance Base (Fix f) = -- ???
   f
```

# Базовые функторы

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

type instance Base JsonValue = JsonValueF

type instance Base (Fix f) = -- ???
    f

data Nat = Z | S Nat
type instance Base Nat = -- ???
```

# Базовые функторы

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

type family Base (t :: *) :: * → *

class Functor (Base t) ⇒ Recursive (t :: *) where
  project :: t → Base t t

cata :: Recursive t ⇒ (Base t a → a) → t → a

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

type instance Base JsonValue = JsonValueF

type instance Base (Fix f) = -- ???
   f

data Nat = Z | S Nat
type instance Base Nat = -- ???
   Maybe
```

# Автоматическая генерация изморфизма

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z
```

# Автоматическая генерация изморфизма

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)
```

# Автоматическая генерация изморфизма

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)

type instance Base Z = ZF
```

# Автоматическая генерация изморфизма

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)

type instance Base Z = ZF

instance Recursive Z where
  project (Z1 x)    = Z1F x
  project (Z2 s ei) = Z2F s ei
```

# Автоматическая генерация изморфизма

```haskell
data Z = Z1 Int | Z2 String (Either Char (Int, Z))

makeBaseFunctor ''Z


-- automatically generated:
data ZF r = Z1F Int | Z2F String (Either Char (Int, r))
  deriving (Functor, Foldable, Traversable)

type instance Base Z = ZF

instance Recursive Z where
  project (Z1 x)      = Z1F x
  project (Z2 s ei) = Z2F s ei

instance Corecursive Z where
  embed (Z1F x)      = Z1 x
  embed (Z2F s ei) = Z2 s ei
```

# Старый пример в новой одежде

```
enrat :: Fix JsonValueF → Fix JsonValueF                    -- recap
enrat = cata $ Fix . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x
```

# Старый пример в новой одежде

```haskell
enrat :: Fix JsonValueF → Fix JsonValueF                    -- recap
enrat = cata $ Fix . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x


enrat' :: JsonValue → JsonValue
enrat' = cata $ embed . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x
```

# Старый пример в новой одежде

```haskell
enrat :: Fix JsonValueF → Fix JsonValueF                    -- recap
enrat = cata $ Fix . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x


enrat' :: JsonValue → JsonValue
enrat' = cata $ embed . eAlg where
  eAlg x@(JsonStringF s) = maybe x JsonNumberF $ readMaybe s
  eAlg x = x


enrat :: JsonValue → JsonValue
enrat = cata $ enratAlg . embed where
  eAlg x@(JsonString s) = maybe x JsonNumber $ readMaybe s
  eAlg x = x
```

Раздел 6

# Есть ли жизнь после катаморфизма?

# За катаморфизмом

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
cata :: Recursive t ⇒ (Base t a → a) → t → a
```

# За катаморфизмом

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
cata :: Recursive t ⇒ (Base t a → a) → t → a
```

```haskell
foldr :: (a → b → b) → b → [a] → b
```

# За катаморфизмом

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
cata :: Recursive t ⇒ (Base t a → a) → t → a
```

```haskell
foldr :: (a → b → b) → b → [a] → b
```

Находили ли вы себя за написанием функций следующего вида?

```haskell
f :: [X] → Res
f = fst . foldr f (initRes, initSt)
  where f x (res, st) = (combined x res st, updated st)
  combined x st res = ...
  updated st = ...
```

# Зигоморфизм

```
cata :: Functor dataF
     ⇒                  (dataF a      → a) → Fix dataF → a
```

```
cata :: Recursive t
     ⇒                   (Base t a      → a) → t → a
```

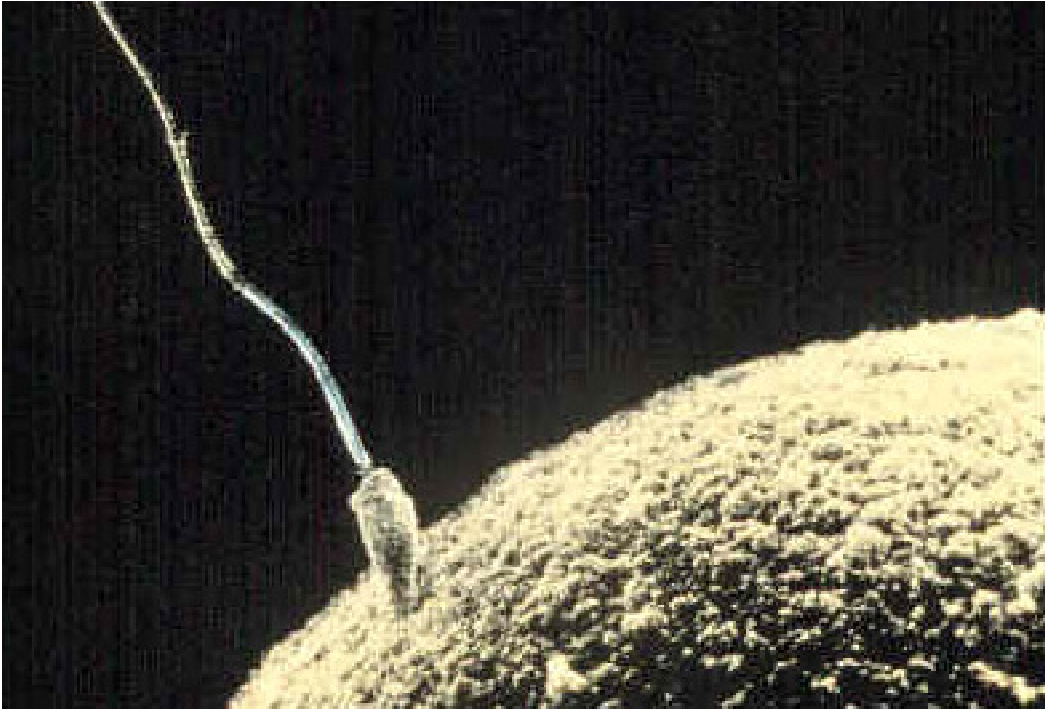
# Зигоморфизм

```
cata :: Functor dataF
     ⇒                (dataF a      → a) → Fix dataF → a

zygo :: Functor dataF
     ⇒ (dataF b → b) → (dataF (b, a) → a) → Fix dataF → a
```

```
cata :: Recursive t
     ⇒                 (Base t a      → a) → t → a

zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```



# Пример зигоморфизма

```haskell
showAlg :: Show a => BinTreeF a String -> String          --recap
showAlg (BLeafF a)    = show a
showAlg (BNodeF l r) = "{left: " ++ l ++ ", right: " ++ r ++ "}"
```

# Пример зигоморфизма

```
    showAlg :: Show a ⇒ BinTreeF a String → String        --recap
    showAlg (BLeafF a)   = show a
    showAlg (BNodeF l r) = "{left: " ++ l ++ ", right: " ++ r ++ "}"

data Color = Red | Green | Blue
  deriving (Show, Eq, Ord, Enum, Bounded)
nextColor :: Color → Color
```

# Пример зигоморфизма

```haskell
    showAlg :: Show a ⇒ BinTreeF a String → String          --recap
    showAlg (BLeafF a)   = show a
    showAlg (BNodeF l r) = "{left: " ++ l ++ ", right: " ++ r ++ "}"

data Color = Red | Green | Blue
  deriving (Show, Eq, Ord, Enum, Bounded)
nextColor :: Color → Color

colorsAlg :: BinTreeF a Color → Color
colorsAlg (BLeafF _)   = Red
colorsAlg (BNodeF l r) = nextColor $ max l r
```

# Пример зигоморфизма

```haskell
    showAlg :: Show a ⇒ BinTreeF a String → String       --recap
    showAlg (BLeafF a)    = show a
    showAlg (BNodeF l r) = "{left: " ++ l ++ ", right: " ++ r ++ "}"

data Color = Red | Green | Blue
  deriving (Show, Eq, Ord, Enum, Bounded)
nextColor :: Color → Color

colorsAlg :: BinTreeF a Color → Color
colorsAlg (BLeafF _)    = Red
colorsAlg (BNodeF l r) = nextColor $ max l r

colorify :: (Color, String) → String
colorify (c,s) = "<font color=" ++ show c ++ ">" ++ s ++ "</font>"
```

# Пример зигоморфизма

```haskell
    showAlg :: Show a ⇒ BinTreeF a String → String         --recap
    showAlg (BLeafF a)    = show a
    showAlg (BNodeF l r) = "{left: " ++ l ++ ", right: " ++ r ++ "}"

data Color = Red | Green | Blue
  deriving (Show, Eq, Ord, Enum, Bounded)
nextColor :: Color → Color

colorsAlg :: BinTreeF a Color → Color
colorsAlg (BLeafF _)    = Red
colorsAlg (BNodeF l r) = nextColor $ max l r

colorify :: (Color, String) → String
colorify (c,s) = "<font color=" ++ show c ++ ">" ++ s ++ "</font>"

showColored :: Show a ⇒ BinTree a → String
showColored = zygo colorsAlg (showAlg . fmap colorify)
```

# Обобщения (частные случаи) катаморфизма

```haskell
cata :: Functor dataF
    ⇒                    (dataF a        → a) → Fix dataF → a
```

```haskell
cata :: Recursive t
    ⇒                    (Base t a       → a) → t → a
```

# Обобщения (частные случаи) катаморфизма

```
cata :: Functor dataF
     ⇒                    (dataF a      → a) → Fix dataF → a
zygo :: Functor dataF
     ⇒ (dataF b → b) → (dataF (b, a) → a) → Fix dataF → a
```

```
cata :: Recursive t
     ⇒                    (Base t a      → a) → t → a
zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```

# Параморфизм

```
cata :: Functor dataF
     ⇒                    (dataF a      → a) → Fix dataF → a
zygo :: Functor dataF
     ⇒ (dataF b → b) → (dataF (b, a) → a) → Fix dataF → a
para :: Functor dataF
     ⇒         (dataF (Fix dataF, a) → a) → Fix dataF → a
```

```
cata :: Recursive t
     ⇒                    (Base t a      → a) → t → a
zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
para :: Recursive t
     ⇒                    (Base t (t, a) → a) → t → a
```

# Раздел 7

## А наоборот?

# Обратная свёртке

```haskell
nmlist :: Integer → Integer → [Integer]        -- recap
nmlist n m | n > m     = []
           | otherwise = n : nmlist (n + 1) m
```

# Обратная свёртке

```haskell
nmlist :: Integer → Integer → [Integer]        -- recap
nmlist n m | n > m     = []
           | otherwise = n : nmlist (n + 1) m
```

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]
```

# Обратная свёртке

```haskell
  nmlist :: Integer → Integer → [Integer]       -- recap
  nmlist n m | n > m     = []
             | otherwise = n : nmlist (n + 1) m
```

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]

nmlist n m = unfoldr f n where
  f x | x > m     = Nothing
      | otherwise = Just (x, x + 1)
```

# Обратная свёртке

```haskell
  nmlist :: Integer → Integer → [Integer]        -- recap
  nmlist n m | n > m     = []
             | otherwise = n : nmlist (n + 1) m
```

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]
```

```haskell
nmlist n m = unfoldr f n where
  f x | x > m     = Nothing
      | otherwise = Just (x, x + 1)
```

*Любые функции вида* `… → [a]` *могут быть выражены через* `unfoldr`

# Обратная свёртке

```haskell
nmlist :: Integer → Integer → [Integer]          -- recap
nmlist n m | n > m      = []
           | otherwise = n : nmlist (n + 1) m
```

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]
```

```haskell
nmlist n m = unfoldr f n where
  f x | x > m      = Nothing
      | otherwise = Just (x, x + 1)
```

*Любые функции вида* `... → [a]` *могут быть выражены через* `unfoldr`

```haskell
insertSort :: Ord a ⇒ [a] → [a]
insertSort = unfoldr f where
  f [] = Nothing
  f xs = Just (m, delete m xs)
    where m = minimum xs
```

# Обратная свёртке

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
```

# Обратная свёртке

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
```

$$\texttt{Maybe (a, b)} \rightleftharpoons \texttt{ListF a b}$$

# Обратная свёртке

```haskell
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
```

$$\texttt{Maybe (a, b)} \;\rightleftharpoons\; \texttt{ListF a b}$$

```haskell
unfoldr :: (b → ListF a b) → b → [a]
```

# Обратная свёртке

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
```

$$\texttt{Maybe (a, b)} \rightleftharpoons \texttt{ListF a b}$$

```
unfoldr :: (b → ListF a b) → b → [a]
```

$$\texttt{[a]} \rightleftharpoons \texttt{Fix (ListF a)}$$

# Обратная свёртке

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
                   Maybe (a, b)  ⇌  ListF a b
unfoldr :: (b → ListF a b) → b → [a]
                      [a]  ⇌  Fix (ListF a)
unfoldr :: (b → ListF a b) → b → Fix ListF
```

# Обратная свёртке

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
data ListF a b = Nil | Cons a b
```

$$\texttt{Maybe (a, b)} \rightleftharpoons \texttt{ListF a b}$$

```
unfoldr :: (b → ListF a b) → b → [a]
```

$$\texttt{[a]} \rightleftharpoons \texttt{Fix (ListF a)}$$

```
unfoldr :: (b → ListF a b) → b → Fix ListF
```

Напрашивается обобщение

```
ana :: Functor dataF ⇒ (b → dataF b) → b → Fix dataF
```

# Анаморфизм

```haskell
foldr    :: (a → b → b) → b  → [a] → b

cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a

cata :: Recursive t   ⇒ (Base t a → a) → t → a
```

# Анаморфизм

```
foldr   :: (a → b → b) → b  → [a] → b
unfoldr :: (b → Maybe (a, b)) → b   → [a]
```

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

```
cata :: Recursive t   ⇒ (Base t a → a) → t → a
ana  :: Corecursive t ⇒ (a → Base t a) → a → t
```

# Обобщения и аналогии

```haskell
cata :: Functor dataF ⇒ (dataF a      → a) → Fix dataF → a

ana ::    . . .    ⇒ (a → dataF a)          → a → Fix dataF



cata :: Recursive t ⇒ (Base t a      → a)          → t → a

ana :: Corecursive t ⇒ (a → Base t a)          → a → t
```

# Обобщения и аналогии

```
cata :: Functor dataF ⇒ (dataF a      → a) → Fix dataF → a
para :: Functor dataF ⇒ (dataF (t, a) → a) → Fix dataF → a
ana  ::    . . .      ⇒ (a → dataF a)      → a → Fix dataF
```

```
cata :: Recursive t ⇒ (Base t a      → a)      → t → a
para :: Recursive t ⇒ (Base t (t, a) → a)      → t → a
ana  :: Corecursive t ⇒ (a → Base t a)         → a → t
```

# Апоморфизм

```
cata :: Functor dataF ⇒ (dataF a      → a) → Fix dataF → a
para :: Functor dataF ⇒ (dataF (t, a) → a) → Fix dataF → a
ana  ::    . . .     ⇒ (a → dataF a)          → a → Fix dataF
apo  ::    . . .     ⇒ (a → dataF (Either t a)) → a → Fix dataF
```

```
cata :: Recursive t ⇒ (Base t a      → a)       → t → a
para :: Recursive t ⇒ (Base t (t, a) → a)       → t → a
ana  :: Corecursive t ⇒ (a → Base t a)          → a → t
apo  :: Corecursive t ⇒ (a → Base t (Either t a)) → a → t
```

# Refold

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

# Refold

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

Построить, затем разобрать

# Refold

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

Построить, затем разобрать

```
hylo :: Functor dataF ⇒ (dataF b → b) → (a → dataF a) → a → b
```

# Refold

```
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

Построить, затем разобрать

```
hylo :: Functor dataF ⇒ (dataF b → b) → (a → dataF a) → a → b
```

Разобрать и пересобрать

```
meta :: (Functor Dat1F, Functor Dat2F)
     ⇒ (Dat1F a → a) → (a → Dat2F a) → Fix Dat1F → Fix Dat2F
```

```
meta :: (Recursive t, Corecursive f)
     ⇒ (Base t a → a) → (a → Base f a) → t → f
```

# Refold

```haskell
cata :: Functor dataF ⇒ (dataF a → a) → Fix dataF → a
ana  :: Functor dataF ⇒ (a → dataF a) → a → Fix dataF
```

Построить, затем разобрать

```haskell
hylo :: Functor dataF ⇒ (dataF b → b) → (a → dataF a) → a → b
```

Разобрать и пересобрать

```haskell
meta :: (Functor Dat1F, Functor Dat2F)
     ⇒ (Dat1F a → a) → (a → Dat2F a) → Fix Dat1F → Fix Dat2F

meta :: (Recursive t, Corecursive f)
     ⇒ (Base t a → a) → (a → Base f a) → t → f

meta :: (Recursive t, Corecursive f)
     ⇒ (Base t a → a) → (a → b) → (b → Base f b) → t → f
```

# К применению

Возможно, несколько неожиданная область...

# К применению

Возможно, несколько неожиданная область...

- Катаморфизм
  - Encoding Recurrent Neural Networks

# К применению

Возможно, несколько неожиданная область...

- Катаморфизм
  - Encoding Recurrent Neural Networks
  - Recursive Neural Networks (TreeNets)

# К применению

Возможно, несколько неожиданная область...

- Катаморфизм
  - Encoding Recurrent Neural Networks
  - Recursive Neural Networks (TreeNets)
- Анаморфизм
  - Generating Recurrent Neural Networks
  - Inverse TreeNets

# К применению

Возможно, несколько неожиданная область...

- Катаморфизм
  - Encoding Recurrent Neural Networks
  - Recursive Neural Networks (TreeNets)
- Анаморфизм
  - Generating Recurrent Neural Networks
  - Inverse TreeNets
- Метаморфизм Гиббонса
  - General Recurrent Neural Networks

# К применению

Возможно, несколько неожиданная область...

- Катаморфизм
  - Encoding Recurrent Neural Networks
  - Recursive Neural Networks (TreeNets)
- Анаморфизм
  - Generating Recurrent Neural Networks
  - Inverse TreeNets
- Метаморфизм Гиббонса
  - General Recurrent Neural Networks
- Комбинация мета- и хиломорфизмов
  - Bidirectional Recursive Neural Networks

# Частный случай

```
hylo :: (Recursive t, Corecursive f)
     ⇒ (Base t b → b) → (a → Base f a) → a → b

zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```

# Частный случай

```haskell
hylo :: (Recursive t, Corecursive f)
     ⇒ (Base t b → b) → (a → Base f a) → a → b

zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```

Если заниматься перепаковкой

```haskell
hylo' :: (Recursive t, Corecursive f)
      ⇒ (Base f (b, f) → f) → (t → Base t (b, f)) → t → f

zygo' :: Recursive t
      ⇒ (Base t b → b)      → (Base t (b, f) → f) → t → f
```

# Частный случай

```haskell
hylo :: (Recursive t, Corecursive f)
     ⇒ (Base t b → b) → (a → Base f a) → a → b

zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```

Если заниматься перепаковкой

```haskell
hylo' :: (Recursive t, Corecursive f)
      ⇒ (Base f (b, f) → f) → (t → Base t (b, f)) → t → f

zygo' :: Recursive t
      ⇒ (Base t b → b)      → (Base t (b, f) → f) → t → f
```

- `hylo` и `zygo` имеют параллельно работающие алгебры

# Частный случай

```
hylo :: (Recursive t, Corecursive f)
     ⇒ (Base t b → b) → (a → Base f a) → a → b

zygo :: Recursive t
     ⇒ (Base t b → b) → (Base t (b, a) → a) → t → a
```

Если заниматься перепаковкой

```
hylo' :: (Recursive t, Corecursive f)
      ⇒ (Base f (b, f) → f) → (t → Base t (b, f)) → t → f

zygo' :: Recursive t
      ⇒ (Base t b → b)      → (Base t (b, f) → f) → t → f
```

- hylo и zygo имеют параллельно работающие алгебры
- но в zygo – сонаправленные, а hylo – противонаправленные

Раздел 8

# Заключительный

# Винни Пух и все-все-все

- folds
  - *катаморфизм*
  - *параморфизм*
  - *зигоморфизм*
  - хистоморфизм
  - препроморфизм
  - зигохистоморфный препроморфизм
- unfolds
  - *анаморфизм*
  - *апоморфизм*
  - футуморфизм
  - постпроморфизм
- refolds
  - *хиломорфизм*
  - хрономорфизм
  - синхроморфизм
  - экзоморфизм
  - *метаморфизм Гиббонса*
  - динаморфизм
  - алгебры и коалгебры Элгота

# Винни Пух и ~~все-все-все~~ не все

- folds
  - *катаморфизм*
  - *параморфизм*
  - *зигоморфизм*
  - хистоморфизм
  - препроморфизм
  - зигохистоморфный препроморфизм
- unfolds
  - *анаморфизм*
  - *апоморфизм*
  - футуморфизм
  - постпроморфизм
- refolds
  - *хиломорфизм*
  - хрономорфизм
  - синхроморфизм
  - экзоморфизм
  - *метаморфизм Гиббонса*
  - динаморфизм
  - алгебры и коалгебры Элгота

# Без стеснения вдохновлялся

- Harold Carr, Refactoring Recursion (2019)
- Jeremy Gibbons, Origami programming (2003)
- Alexander Konovalov, Recursion schemes, algebras, finall tagless, data types (2019)
- Tim Williams, Recursion Schemes (2013)
- Paweł Szulc, Going bananas with recursion schemes for fixed point data types (2017)
- Edward Kmett's library and blogposts
- Matryoshka library (examples, blogposts, external resources list)
- Олег Нижников, Современное ФП с Tagless Final (2018)
- Christopher Olah, Neural Networks, Types, and Functional Programming (2015)
- Rob Norris, Pure Functional Database Programming with Fixpoint Types (2016)
- и многими другими...

*элементы списка нажимабельны*

Спасибо, что дослушали

Вопросы?