# Дифференцирование и обобщённые зипперы в Haskell

А. Болотина

16 ноября 2016
Семинар «Языки программирования и компиляторы»

# Содержание

# Huet's "Zipper"

- **Задача:** представление древовидной структуры данных вместе с фокусом на текущем узле, который может перемещаться влево, вправо, вниз и вверх по этой структуре.
- **Мотивация:** многие эффективные алгоритмы используют деструктивные операции над элементами структур данных. Сложность при использовании чисто функциональных структур данных $\Theta(\log n)$ или $\Theta(n)$.
- **Решение** (G. Huet. "The Zipper", 1997)**:** фокус хранит текущий узел и путь, восходящий от него к корню дерева.
- **Эффективность:** изменение элемента структуры за время $\Theta(1)$.

# Пример: бинарное дерево

## Определение бинарного дерева

```
data Tree a = Leaf a | Bin a (Tree a) (Tree a)
```

## Контекст

$\texttt{data Context}_{\texttt{Tree}}\ \texttt{a = CLeft a (Tree a)}$
$\qquad\qquad\qquad\quad\texttt{| CRight a (Tree a)}$

## Путь — список контекстов

$\texttt{type Path}_{\texttt{Tree}}\ \texttt{a = [Context}_{\texttt{Tree}}\ \texttt{a]}$

## Зиппер

$\texttt{type Zipper}_{\texttt{Tree}}\ \texttt{a = (Tree a, Path}_{\texttt{Tree}}\ \texttt{a)}$

# Навигация по дереву

## Движение вправо

```
goRight :: Zipper_Tree a -> Maybe (Zipper_Tree a)
goRight (t, CLeft x r : p) = Just (r, CRight x t : p)
goRight _                  = Nothing
```

## Движение вниз

```
goDown :: Zipper_Tree a -> Maybe (Zipper_Tree a)
goDown (Bin x l r, p) = Just (l, CLeft x r : p)
goDown _              = Nothing
```

## Движение вверх

```
goUp :: Zipper_Tree a -> Maybe (Zipper_Tree a)
goUp (t, CLeft  x r : p) = Just (Bin x t r, p)
goUp (t, CRight x l : p) = Just (Bin x l t, p)
goUp (_, [])             = Nothing
```

# Использование

## Начало и конец навигации

```haskell
enter :: Tree a -> Zipper_Tree a
enter t = (t, [])
```

```haskell
leave :: Zipper_Tree a -> Tree a
leave (t, []) = t
leave loc
  = leave $ fromJust $ goUp loc
```

## Обновление узла

```haskell
update :: (Tree a -> Tree a) -> Zipper_Tree a -> Zipper_Tree a
update f (t, ctxt) = (f t, ctxt)
```

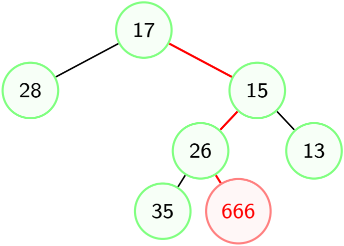
## Использование композиции

```haskell
(>>>) :: (a -> b) -> (b -> c) -> a -> c
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

# Пример использования

## Пример дерева

```haskell
tree :: Tree Int
tree = Bin 17 (Leaf 28)
              (Bin 15 (Bin 26 (Leaf 35)
                              (Leaf 666))
                      (Leaf 13))
```




## Замена одного узла

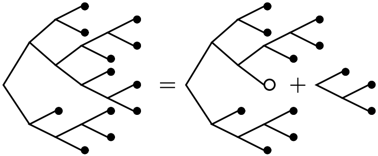
```haskell
change :: Tree Int -> Maybe (Tree Int)
change = enter >>> goDown >=> goRight >=> goDown
               >=> goDown >=> goRight >=> update (\_ -> Leaf 42)
     >>> leave >>> return

*Main> change tree
Just (Bin 17 (Leaf 28) (Bin 15 (Bin 26 (Leaf 35) (Leaf 42))
(Leaf 13)))
```

# Зипперы vs. контексты

Статья К. Макбрайда (2001)

*"The Derivative of a Regular Type is its Type of One-Hole Contexts."*

«Производная регулярного типа — это его тип одноместного контекста».



## План

- Дифференцирование.
- Регулярные типы.

# Правила дифференцирования

$$\partial_x x \mapsto 1 \quad (1) \qquad \partial_x(F + G) \mapsto \partial_x F + \partial_x G \quad (3)$$

$$\partial_x C \mapsto 0 \quad (2) \qquad \partial_x(F \times G) \mapsto F \times \partial_x G + \partial_x F \times G \quad (4)$$

$$\partial_x(F|_{y=G}) \mapsto \partial_x F|_{y=G} + \partial_y F|_{y=G} \times \partial_x G \quad (5)$$

$$\frac{\partial}{\partial x} f(a,\, b) = (\frac{\partial}{\partial u} f(u,\, v)\frac{\partial a}{\partial x} + \frac{\partial}{\partial v} f(u,\, v)\frac{\partial b}{\partial x})|_{(u,\, v)=(a,\, b)}, \quad a = x$$

$$\partial_x(\mu y.\, F) \mapsto \mu z.\, \underline{\partial_x F|_{y=\mu y.\, F}}_z + \underline{\partial_y F|_{y=\mu y.\, F}}_z \times z \quad (6)$$

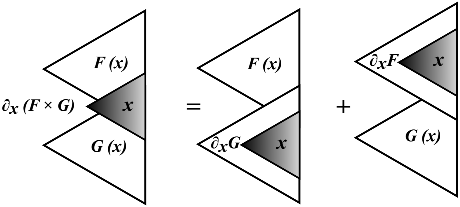
$$\begin{aligned} \partial_x(\mu y.\, F) &= \partial_x(F|_{y=\mu y.\, F}) \\ &= \partial_x F|_{y=\mu y.\, F} + \partial_y F|_{y=\mu y.\, F} \times \partial_x(\mu y.\, F) \end{aligned}$$

**Свойство оператора $\mu$**

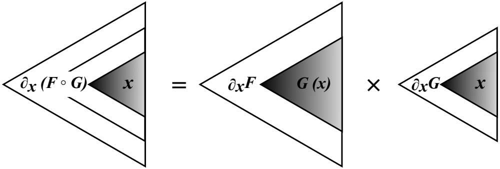
$$\mu y.\, F = F\,(\mu y.\, F)$$

# Контексты и дифференцирование

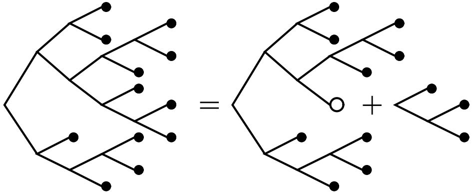
$$\partial_x(F \times G) = F \times \partial_x G + \partial_x F \times G$$

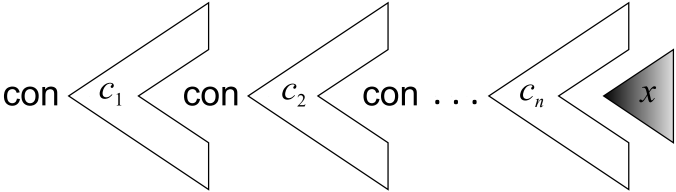



$$\partial_x(F \circ G) = (\partial_x F \circ G) \times \partial_x G$$



# Контекст как стек одноместных контекстов





# Пример: производная типа арифметического выражения

$$\begin{aligned}\mathbf{expr} &= C_{Int} + C_{Bool} + \mathbf{expr}^2 + \mathbf{expr}^3 \\ &\cong \mu x.\, C_{Int} + C_{Bool} + x^2 + x^3\end{aligned}$$

## Тип арифм. выражения

```
data Expr = LInt Int
          | LBool Bool
          | Add Expr Expr
          | If Expr Expr Expr
```

## Его производная (контекст)

```
data Context_Expr
          = Add₁ Expr
          | Add₂ Expr
          | If₁  Expr Expr
          | If₂  Expr Expr
          | If₃  Expr Expr
```

$$\begin{aligned}\partial\,\mathbf{expr} &\cong \partial_x(C_{Int} + C_{Bool} + x^2 + x^3) \\ &\cong \mu x.\, 2 \times x + 3 \times x^2 \\ &\cong 2 \times \mathbf{expr} + 3 \times \mathbf{expr}^2\end{aligned}$$

Ещё пример: производная списка — это пара списков (префикс и суффикс).

$$\begin{aligned}\partial_x \mathbf{list}\, x &\cong \partial_x(\mu y.\, 1 + x \times y) \\ &\cong \mu z.\, \underline{(\partial_x(1 + x \times y))|_{y=\mathbf{list}\, x}}_z \\ &\quad + \underline{(\partial_y(1 + x \times y))|_{y=\mathbf{list}\, x}}_z \times z \\ &\cong \mu z.\, \underline{y|_{y=\mathbf{list}\, x}}_z + \underline{x|_{y=\mathbf{list}\, x}}_z \times z \\ &\cong (\mathbf{list}\, x) \times (\mathbf{list}\, x)\end{aligned}$$

# Алгебраические типы данных (АТД), примеры

## Единичный тип

```
data UnitData = Unit
```

Пример значения:

```
unit = Unit
```

## Нулевой тип

```
data Zero
```

## Тип-сумма

```
data Either a b = Left  a
                | Right b
```

Перечисление:

```
data Bool = True | False
```

Пример значения:

```
left :: Either Bool Int
left = Left True
```

## Тип-константа

```
data ConstData a = Const a
```

Пример значения:

```
const :: ConstData Int
const = Const 13
```

## Тип-произведение

```
data (,,) a b c   = (,,) a b c
data PairData a b = Pair a b
```

Пример значения:

```
triple :: (String, Int, Char)
triple = ("hello", 15, 'b')
```

## Рекурсивный тип

```
data List a = Nil
            | Cons a (List a)
```

# Обобщённое представление АТД

## Типы для обобщённого представления АТД

```
data K a         x = K a                  -- константа
data I           x = I x                  -- x
data U           x = Unit                 -- 1
data (f :+: g) x = L (f x) | R (g x)   -- сумма
data (f :×: g) x = f x :×: g x         -- произведение
```

# Пример: логическое выражение

## Пример АТД

```
data Prop
     = T | F
     | And Prop Prop
     | Not Prop
```

## Пример значения

```
x :: Prop
x = And (Not F) T
```

$$x = \overline{0} \wedge 1$$

## Его обобщённый вид

```
type Prop' = (U :+: U :+: (I :×: I) :+: I) Prop
```

## Значение

```
x' :: Prop'
x' = R (R (L (I (Not F) :×: I T)))
```

# Понятие функтора, примеры

## Понятие функтора

*Функтор* — это отображение на типах (параметрический тип).

Пример — список:

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

## Примеры полиномиальных функторов (ПФ)

$$F(x) = x^2 + 2x; \qquad F(Int) = (Int,\ Int) + 2\,Int.$$
$$G(x) = x^3 + Char; \qquad G(Bool) = (Bool,\ Bool,\ Bool) + Char.$$

## Примеры знач. типа $F(Int)$

(3, 1729); (6, 24); 15; 18.

## Примеры знач. типа $G(Bool)$

$(True,\ True,\ False)$; 'a'; 's'.

# Представление ПФ с помощью обобщённого программирования

## Пример функтора

$$F(x) = x^2 + 3x + Bool + 2$$

## Его обобщённое представление

```haskell
type F = (I :×: I) :+: I :+: I :+: I :+: K Bool :+: U :+: U
```

## Примеры значений для типа $F(Int)$

$$y_1 = (13,\ 38)$$

```haskell
y1 :: F Int
y1 = L (I 13 :×: I 38)
```

$$y_2 = True$$

```haskell
y2 :: F Int
y2 = R (R (R (R (L (K True)))))
```

# Неподвижные точки

## Понятие регулярного типа

*Регулярный тип данных* — это тип, который может быть представлен в виде неподвижной точки некоторого полиномиального функтора.

## Неподвижная точка функтора $F$

$$\mu y.\, F \;=\; F\,(\mu y.\, F)$$

## Тип как неподвижная точка ПФ

Пусть для регулярного типа $a$ существует такой полиномиальный функтор $\mathrm{PF}_a$, что

$$a \;\cong\; \mu y.\, \mathrm{PF}_a.$$

Тогда

$$a \;\cong\; \mu y.\, \mathrm{PF}_a \;\cong\; \mathrm{PF}_a\,(\mu y.\, \mathrm{PF}_a) \;\cong\; \mathrm{PF}_a\,(a).$$

# Регулярные типы в Haskell

## Пример представления

```
data Prop       = T   | F   | And   Prop Prop | Not   Prop
data PF_Prop x  = T_F | F_F | And_F x    x    | Not_F x
type Prop'      = PF_Prop Prop
```

## Класс регулярного типа

```
class Regular a where
    type PF a :: * -> *
    from      :: a -> PF a a
    to        :: PF a a -> a
```

# Пример

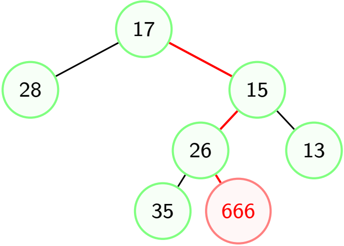
## Пример: определение для типа **Prop**

```
instance Regular Prop where
    type PF Prop = U :+: U :+: (I :×: I) :+: I
    from T          = L Unit
    from F          = R (L Unit)
    from (And x y) = R (R (L (I x :×: I y)))
    from (Not x)    = R (R (R (I x)))
    to (L Unit)                        = T
    to (R (L Unit))                    = F
    to (R (R (L (I x :×: I y)))) = And x y
    to (R (R (R (I x))))           = Not x
```

# Старый пример

## Пример дерева

```
tree :: Tree Int
tree = Bin 17 (Leaf 28)
              (Bin 15 (Bin 26 (Leaf 35)
                              (Leaf 666))
                      (Leaf 13))
```




## Замена одного узла

```
change :: Tree Int -> Maybe (Tree Int)
change = enter >>> goDown >=> goRight >=> goDown
               >=> goDown >=> goRight >=> update (\_ -> Leaf 42)
      >>> leave >>> return

*Main> change tree
Just (Bin 17 (Leaf 28) (Bin 15 (Bin 26 (Leaf 35) (Leaf 42))
(Leaf 13)))
```

# Старые типы

## Основные типы

```
data Tree          a = Leaf a | Bin a (Tree a) (Tree a)
data Context_Tree  a = CLeft  a (Tree a)       -- тип контекста
                     | CRight a (Tree a)
type Path_Tree     a = [Context_Tree a]
type Zipper_Tree   a = (Tree a, Path_Tree a)   -- тип зиппера
```

## Типы функций — зависят от основных типов

```
goRight :: Zipper_Tree a -> Maybe (Zipper_Tree a)
goDown  :: Zipper_Tree a -> Maybe (Zipper_Tree a)
goUp    :: Zipper_Tree a -> Maybe (Zipper_Tree a)

enter   :: Tree a -> Zipper_Tree a
leave   :: Zipper_Tree a -> Tree a
update  :: (Tree a -> Tree a) -> Zipper_Tree a -> Zipper_Tree a
```

# Контекст обобщённого зиппера

## Тип контекста: объявление

```
data family Context (f :: * -> *) a
```

## Определения для типа контекста

```
data instance Context (K b)     a
data instance Context U         a
data instance Context I         a = CI
data instance Context (f :+: g) a = CL (Context f a)
                                  | CR (Context g a)
data instance Context (f :×: g) a = C1 (Context f a) (g a)
                                  | C2 (f a) (Context g a)
```

# Класс обобщённого зиппера

## Класс зиппера

```
class Zipper (f :: * -> *) a where
    ...
```

## Фокус (*location*)

```
data Loc a where
    Loc :: (Regular a, Zipper (PF a) a)
        => a -> [Context (PF a) a] -> Loc a
```

# Интерфейс зиппера

## Класс обобщённого зиппера

```
class Zipper (f :: * -> *) a where
  -- Если возможно, сдвинуть фокус на крайнего
  -- слева потомка
  first :: (a -> Context f a -> b) -> f a -> Maybe b
  -- Если возможно, сдвинуть фокус на крайнего
  -- справа потомка
  last  :: (a -> Context f a -> b) -> f a -> Maybe b
  -- Поместить значение в контекст
  fill  :: a -> Context f a -> f a
  -- Попытаться сдвинуть фокус на один элемент влево
  prev  :: (a -> Context f a -> b) -> a -> Context f a
        -> Maybe b
  -- Попытаться сдвинуть фокус на один элемент вправо
  next  :: (a -> Context f a -> b) -> a -> Context f a
        -> Maybe b
```

# Реализация (1)

## Реализация функций **first**, **fill** и **next** (начало)

```
instance Zipper (K b) a where
  first _ _       = Nothing
  fill  _ _       = error "impossible"
  next  _ _ _     = error "impossible"

instance Zipper U a where
  first _ _       = Nothing
  fill  _ _       = error "impossible"
  next  _ _ _     = error "impossible"

instance (Regular a, Zipper (PF a) a)
      => Zipper I a where
  first f (I x) = return (f x CI)
  fill  x CI    = I x
  next  _ _ _   = Nothing
```

# Реализация (2)

## Реализация функций **first**, **fill** и **next** (продолжение)

```
instance (Zipper f a, Zipper g a) => Zipper (f :+: g) a where
  first f (L x)       = first (\z c  -> f z (CL c)) x
  first f (R y)       = first (\z c  -> f z (CR c)) y
  fill  x (CL c)      = L (fill x c)
  fill  y (CR c)      = R (fill y c)
  next  f x (CL c)  = next  (\z c' -> f z (CL c')) x c
  next  f y (CR c)  = next  (\z c' -> f z (CR c')) y c

instance (Zipper f a, Zipper g a) => Zipper (f :×: g) a where
  first f (x :×: y) = first (\z c  -> f z (C1 c y)) x
            `mplus` first (\z c  -> f z (C2 x c)) y
  fill   x (C1 c y) = fill x c :×: y
  fill   y (C2 x c) = x :×: fill y c
  next f x (C1 c y) = next  (\z c' -> f z (C1 c'           y)) x c
            `mplus` first (\z c' -> f z (C2 (fill x c) c')) y
  next f y (C2 x c) = next  (\z c' -> f z (C2 x          c')) y c
```

# Функции навигации

## Движение вниз

```
goDown :: Loc a -> Maybe (Loc a)
goDown (Loc hole cs)
    = first (\h c -> Loc h (c : cs)) (from hole)
```

## Движение вправо

```
goRight :: Loc a -> Maybe (Loc a)
goRight (Loc _     []) = Nothing
goRight (Loc hole (c : cs))
    = next (\h c' -> Loc h (c' : cs)) hole c
```

## Движение вверх

```
goUp :: Loc a -> Maybe (Loc a)
goUp (Loc _     []) = Nothing
goUp (Loc hole (c : cs))
    = Just (Loc (to $ fill hole c) cs)
```

# Ссылки


1. G. Huet. The Zipper. *JFP*, 1997.
2. C. McBride. The derivative of a regular type is its type of one-hole contexts, 2001.
3. A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic Programming with fixed points for mutually recursive datatypes. *ICFP*, 2009.
4. R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *SCP*, 2004.