

# Типы и пользовательские типы данных

Косарев Дмитрий а.к.а. Kakadu

матмех СПбГУ

3 октября 2019 г.

## В этих слайдах

1. Parametricity theorem
2. Упражнения: угадать поведение функции по её типу
3. Алгебра типов
4. Изоморфизм типов
5. Zipper

# Чистые функции

## Определение

Чистая функция – это

- Детерминированная
- В процессе работы не совершающая “побочных эффектов”

Т.е. запрещены: ввод-вывод, случайные значения, присваивания

Н.В. Это свойство *функции*, а не языка программирования

# Parametricity theorem [1]

## Теорема

*О параметричности. Чистые функции с параметрическим полиморфизмом работают одинаково для всех возможных типов.<sup>a</sup>*

---

<sup>a</sup>Для Haskell верна, для функциональных strict-языков (например, OCaml) с некоторыми оговорками.

```
{-# LANGUAGE ExplicitForAll #-}
```

```
id :: forall a . a -> a
id x = x
```

# Чем функции в программировании отличаются от математических?

# Чем функции в программировании отличаются от математических?

- Аварийное завершение
- Отсутствие завершения

# Чем функции в программировании отличаются от математических?

- Аварийное завершение
- Отсутствие завершения

Функции / Область	математика	программирование
всегда возвращают результат	функции	тотальные функции
могут не вернуть результат	частичные функции	функции

# Как может работать чистая тотальная функция со следующим типом?

? `[a] -> [a]`

? `[a] -> Bool`

? `(a -> b) -> [a] -> [b]`

? `(a -> a -> Bool) -> [a] -> [a]`

? `(a -> a -> Ordering) -> [a] -> [a]`, если

`Prelude> :i Ordering`

`data Ordering = LT | EQ | GT`

? `(a -> b -> a) -> a -> [b] -> a`

? `Maybe a -> Maybe b -> (a -> b -> c) -> Maybe c`



# Типы как множества

Тип  $T$  у значения с именем  $x$  – это множество совокупность значений, которые могут быть у  $x$ .

Если  $x :: T$ , то говорят, что тип  $T$  *населен* иксом.

Если  $\nexists x$ , таких что  $x :: T$ , то тип  $T$  *не населен*.

## Примеры "базовых" типов

Тип, который не населен (нет конструкторов, даже приватных)

```
Prelude> :i Data.Void.Void
data Data.Void.Void          -- Defined in 'Data.Void'
```

Тип, у которого только один житель

```
Prelude> :i ()
data () = ()                  -- Defined in 'GHC.Tuple'
Prelude> :t ()
() :: ()
```

# Мощность типа

## Определение

*Мощность типа* – количество различных значений (термов), которые населяют этот тип.

Что такое "различные" требует некоторых уточнений...

# Observational equivalence

## Определение

*Observational equivalence* – свойство двух сущностей быть неразличимыми, при наблюдении снаружи за их свойствами.

Другими словами: можем ли мы написать алгоритм, который принимает два значения и говорит различны ли они, и как этот алгоритм будет действовать?

# Как ведёт себя тотальная чистая функция с типом...

... и какова мощность этого типа?

?  $() \rightarrow \text{Int}$

?  $\text{Int} \rightarrow ()$

?  $\text{Void} \rightarrow \text{Int}$

?  $\text{Int} \rightarrow \text{Void}$

?  $a \rightarrow ()$

?  $\text{Void} \rightarrow a$

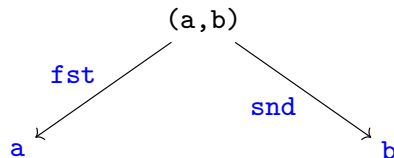
# Тип пары (Декартово произведение)

```
Prelude> :i (,)
data (,) a b = (,) a b
    -- Defined in 'GHC.Tuple'
```

Определим проекции:

```
Prelude> let fst (x,_) = x
Prelude> :t fst
fst :: (a, b) -> a
```

```
Prelude> let snd (_,y) = y
Prelude> :t snd
snd :: (a, b) -> b
```

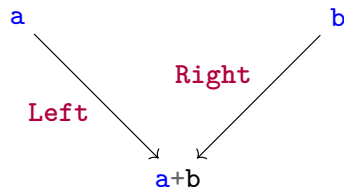


# Тип Either (сумма)

```
Prelude> :i Either
data Either a b = Left a | Right b
    -- Defined in 'Data.Either'
```

Определим проекции:

```
Prelude> :t Left
Left :: a -> Either a b
Prelude> :t Right
Right :: b -> Either a b
```



# Изоморфизм типов

Observational equivalence говорит только про значения одного типа, но можно пытаться рассуждать про различные типы...

## Определение

Типы **A** и **B** *изоморфны* ( $\sim$ ), если можно предъявить две тотальные функции

$l :: A \rightarrow B$  и  $r :: B \rightarrow A$ , такие что

$$l \circ r == id_B$$

$$r \circ l == id_A$$

где  $id$  – тождественная функция

$id :: \text{forall } a . a \rightarrow a$

$id\ x = x$



## Упражнения про произведение и сумму

- Кто населяет тип `(Void, a)` для произвольного типа `a`?
- Кто населяет тип `((), a)` для произвольного типа `a`?
- Кто населяет тип `Either Void a` для произвольного типа `a`?
- Кто населяет тип `Either () a` для произвольного типа `a`?
- В некоторых упражнениях выше можно строить изоморфизм между двумя типами. Сообразите между какими и постройте там, где возможно.

Haskell	Математика	Заметки
<code>data Void</code>	0	невозможно построить значение
<code>data Unit = Unit</code>	1	Ровно 1 житель
<code>data Bool = True   False</code>	2	
<code>data Maybe a = Just a   Nothing</code>	$a+1$	
<code>data Either a b = Left a   Right b</code>	$a+b$	Сумма или Either
<code>data (a, b) = (a, b)</code>	$a*b$	Произведение
<code>a -&gt; b</code>	$a^b$	

Типы – это коммутативное полукольцо (semiring или rig) с 1

- По сложению (**Either**) – коммутативный моноид
  - Ассоциативность сложения:  $a + (b + c) = (a + b) + c$
  - **Void** – нейтральный элемент:  $0 + a = a + 0 = a$
  - Коммутативность:  $a + b = b + a$
- По умножению  $((,))$  – коммутативный моноид
  - Ассоциативность усножения
  - **Unit** – нейтральный элемент
  - Коммутативность
- Умножение на ноль дает ноль:  $0 \cdot a = a \cdot 0 = 0$

Можно пытаться:

- раскладывать в ряды
- брать производные

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

```
-- or
```

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

$$\begin{aligned} L &= 1 + aL \\ &= 1 + a(1 + aL) \\ &= 1 + a + a^2(1 + aL) \\ &= \dots \end{aligned}$$

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

```
-- or
```

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

$$\begin{aligned}
 L &= 1 + aL \\
 &= 1 + a(1 + aL) \\
 &= 1 + a + a^2(1 + aL) \\
 &= \dots
 \end{aligned}$$

Или даже нечто более дикое:

$$\begin{aligned}
 L &= 1 + aL \\
 L(1 - a) &= 1 \\
 L &= \frac{1}{1 - a} \\
 L &= 1 + a + a^2 + \dots
 \end{aligned}$$

# Zipper для списков

Структура данных для "блуждания" туда-сюда

```
data Zipper a = Zip ![a] ![a]
```

```
fromList :: [a] -> Zipper a
```

```
start :: Zipper a -> Zipper a
```

```
end :: Zipper a -> Zipper a
```

```
left :: Zipper a -> Zipper a
```

```
right :: Zipper a -> Zipper a
```

```
emptyt :: Zipper a -> Bool
```

```
cursor :: Zipper a -> a
```

```
-- is not total
```

Пример использования

```
> ns = fromList [1..5]
```

```
Zip [] [1,2,3,4,5]
```

```
> right $ ns
```

```
Zip [1] [2,3,4,5]
```

```
> right $ right $ ns
```

```
Zip [2,1] [3,4,5]
```

```
> left $ right $ right $ ns
```

```
Zip [1] [2,3,4,5]
```

```
> :i ($)
```

```
($) :: (a -> b) -> a -> b
```

```
-- Defined in 'GHC.Base'
```

```
infixr 0 $
```

# Как вывести zipper?

$$L = 1 + aL$$

$$\begin{aligned}\frac{\partial L}{\partial a} &= \frac{\partial}{\partial a}(1 + aL) \\ &= L + a \frac{\partial L}{\partial a}\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial a} &= \frac{L}{1 - a} \\ &= L^2\end{aligned}$$

# Как вывести zipper?

$$L = 1 + aL$$

$$\begin{aligned}\frac{\partial L}{\partial a} &= \frac{\partial}{\partial a}(1 + aL) \\ &= L + a \frac{\partial L}{\partial a}\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial a} &= \frac{L}{1 - a} \\ &= L^2\end{aligned}$$

Вопрос к экзамену: вывести zipper для различных деревьев

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

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

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



# Конец

# Ссылки I



Theorems for free!

*Philip Wadler*

ссылка



The algebra (and calculus!) of algebraic data types

*Joel Burget*

web article



The Two Dualities of Computation: Negative and Fractional Types

*Roshan P. James & Amr Sabry*

PDF



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

*Conor McBride*

PDF

## Ссылки II



ListZipper haskell package  
ссылка