

Functional Programing

Functional Programing

Organization stuff

22-09-05

Function declaration

Data Types

Конструкторы в хаскелле

Comments

Lists

Functions: declaration and definition

Алгебраические типы

Pattern matching

Compilation

22-09-12

`:kind`, `:t`, `:info`

=>

Union types

`:` in lists

Аксиомы алгебраических типов данных (ADT)

Способы задизайнить выражение

`let` в коде

`case` и `where` в коде

Cons aka AtLeastOne, Nil aka Empty

Maybe

`map'`

`fold`

Анонимная лямбда-функция

22-09-19

HOF (higher order functions)

1. Функция как аргумент

2. Currying

3. Композиция

Associativity and priority. Function application

Lazy evaluation

Lazy lists

Strict lists

NF, WHNF: (weak head) normal form

`Const`

`|` (guards)

`:+`

Lazy patterns

Benchmarking

`:set`

`foldl`, `foldr`

`Filter`

Симбиоз pattern matching и `|`

List comprehension

22-09-26

Tail recursion

Пример с практики

Records

Практика 22-09-29

`curry` & `uncurry` \todo

Один аргумент или два?

`zip`

`zip3`

zipWith

Пример с композицией: squareEven

class Eq \todo

Практика 22-10-06

22-10-10

Linked list

Memoization in Haskell

Streams

Pure Functional Queues

Simplest implementation of a queue

Практика 22-10-13

Паттерн `node@(Node x)`

Strictness

`seq`

`!`

Laziness

`~`

Использование для аргументов

Использование для функций

Моноиды

22-10-17

Лямбда-исчисления

Обозначения

Свободные и связанные переменные

Пример

Правила \todo

def. (замкнутое множество)

α -conversion

Пример

β -reduction

Примеры

Capture-avoiding substitution \todo

Последовательность де Брёйна (?) \todo

Практика 22-10-20

функция `bind`

Монады

Правила монадов

Maybe и `>>=`

Lists

Пример использования монадов

do-нотации

IO()

read

Пример

Практика 22-10-24

Кодирование Чёрча: способ 1

Элементарные операции

isZero

22-10-24

Кодирование Чёрча: способ 2

Функции

Базисные

Композиция

Примитивно-рекурсивные (ПРФ)

Частично-рекурсивные (ЧРФ)

Th. (о реализуемости)

Th. (о неподвижной точке λ -терма)

Th. (про Y -комбинатор)

Th. (первая теорема рекурсии)

Практика 22-11-03

22-11-07 \todo

Теорема Чёрча — Россера

Правила вывода

Практика 22-11-10

class Functor

fmap

pure

`<*>`

Пример
Хотим создать что-то нестандартное...
newtype
Parser \todo
22-11-14
Типизация
Терм в контексте
Правила типизирования
Примеры
Леммы
Standard problems
Лемма инверсии
22-11-17
Property-based тестирование
Пример
Range
Size
Что, если нет явного свойства?
Пример: проверка min/max элемента в отсортированном списке
Генерируем арифметическое выражение

Organization stuff

Daniil Berezun: danya.berezun@gmail.com

Learn You Haskell for Great Good: <http://learnyouahaskell.com/chapters>

Tg Chat 'Haskell Start': https://t.me/haskell_learn

22-09-05

`info (:)` -- find out what `:` means.

`:type sqrt` или `:t sqrt` -- узнать тип и возвращаемое значение

Function declaration

Math	Haskell
<code>f(x)</code>	<code>f x</code>
<code>f(x, y)</code>	<code>f x y</code>
<code>f(g(x))</code>	<code>f (g x)</code>
<code>f(x, g(y))</code>	<code>f x (g y)</code>
<code>f(x)g(y)</code>	<code>f x * g y</code>

Data Types

- `Bool`
- `Int` -- фиксированного размера
- `Integer` -- длинная арифметика
- `Float, Double`
- `f :: a -> b -> c` -- функция `f` с аргументами `a`, `b` возвращает `c`
- `[a]` -- список типа `a`
- `Char`
- `String = [Char]` (список чаров)
- `Tuple`: `(42, "Hello!") :: (Int, String)`. Кол-во эл-тов ≥ 2 и ≤ 62 в GHC

Конструкторы в хаскелле

```
1 | Data Number a = Int
```

`Int` это конструктор

```
1 | Data Number = Int | Double
```

`Int` и `Double` это раз конструктор и два конструктор

Comments

```
1 | -- comment in a line
2 |
3 | {-
4 | big comment
5 | -}
```

Lists

```
1 | 1 : [] -- adding 1 into the beginning of an empty list
2 |
3 |
4 | let b = [1] -- declaring of a variable. 'let' can be omitted.
5 | b          -- show value of b
6 |
7 |
8 | c = [2, 2] ++ b -- [2, 2, 1]
9 | c = b ++ [3, 3] -- [1, 3, 3]
10 | c = b ++ [2..6] -- [1, 2, 3, 4, 5, 6]
11 | aboba = "ab" ++ " " ++ "oba"
12 |
13 |
14 | c = 5 : 4 : 3 : 2 : b -- [5, 4, 3, 2, 1]
15 |
16 | head c -- [5]
17 | tail c -- [1]
18 | length c -- 5
19 | take 3 c -- [5, 4, 3]
20 |
```

```

21 reverse c -- [1, 2, 3, 4, 5]
22 null c    -- False (checks whether c is empty)
23
24 maximum c -- [5]
25 minimum c -- [1]
26 -- minimum in a string is a letter which is the earliest in the alphabet

```

Functions: declaration and definition

```

1 f :: a -> c
2 -- gets arg of type 'a' and returns 'c'
3
4 g :: a -> b -> c
5 -- gets args of type 'a', 'b' and returns 'c'

```

```

1 SimpleFunc :: Int -> Bool
2
3 SimpleFunc x =
4   if (x == 1)
5   then True
6   else False

```

Объявляем функцию `SimpleFunc`, которая принимает `Int` и возвращает `Bool`. Дальше идёт определение функции.

Алгебраические типы

```

1 data PointT = PointD Double Double
2 {-      [1]      [2]  -----[3]-----
3   [1]: Type constructor
4   [2]: Data constructor
5   [3]: Types wrapped
6   `|` is a Pipe operator
7 -}

```

Pattern matching

```

1 data Cardinal = North | East | South | West
2
3 Checker :: Cardinal -> Bool
4 -- function Checker. It is better to declare a returning type clearly
5 Checker North = True;
6 Checker South = True;
7 Checker _     = False;

```

`_` -- wildcard -- подстановочный знак

Хотим функцию, которая будет возвращать `True` только на типы `North` и `South`.

Для остальных типов подходит только последняя строчка, и возвращаемое значение будет `False`.

Compilation

```
1 | ghc aboba.hs -o a
2 | ./a
```

22-09-12

`:kind`, `:t`, `:info`

`:kind` -- система типов над типами. Показывает параметры для **типа**.

```
1 | ghci> :k Int
2 | ghci> *
3 |
4 | ghci> data PPoint = PPoint a a
5 | ghci> :k PPoint
6 | ghci> PPoint :: a -> a -> a
```

`:t` показывает сигнатуру функции

`:info (,)` -- показывает, что значит `,`

=>

```
1 | dist (Point x1 y1) (Point x2 y2) = sqrt((x1 - x2)^2 + (y1 - y2)^2)
2 |
3 | ghci> :t dist
4 | ghci> dist :: Floating a => Point a -> Point a -> a
```

Штука до `=>` означает, какие типы у переменных. После этой штуки -- объявление функции

Union types

Union type встречается в конструкторах. Буквально означает "либо одно, либо другое"

Пример:

```
1 | data Point = Point2D Double Double | Point3D Double Double Double    -- union type
2 |
3 | pointToList :: Point -> [Double]
4 | pointToList (Point2D x y) = [x, y]
5 | pointToList (Point3D x y z) = [x, y, z]
6 |
7 | ghci> a = Point2D 3 4
8 | ghci> b = Point3D 3 4 5
```

`:` in lists

```
1 data List a = Nil | Cons a (List a)  -- `Cons` -- конструктор.
2
3 (: ) == Cons                        -- `Cons` эквивалентен `:`
4 [] == Nil                          -- `[]` в haskell определяется как `Nil`.
5 1 : [] == Cons 1 Nil               -- вызови конструктор на 1 и пустом списке
6 [1, 2] == Cons 1 (Cons 2 Nil)     -- рекурсивный вызов конструкторов
```

Аксиомы алгебраических типов данных (ADT)

1. *Distinctness*: разные конструкторы \Rightarrow получаются разные значения

$$\forall i \neq j : C_i(x) \neq C_j(y)$$

2. *Injectivity*: значения конструкторов равны \Rightarrow у конструкторов одинаковые параметры

$$C_i(x_1, \dots, x_n) = C_j(y_1, \dots, y_n) \Rightarrow x_k = y_k, \forall k$$

3. *Exhaustiveness*: x -- алгебраический тип \Rightarrow для него есть конструктор

$$x \text{ of some ADT} \Rightarrow \exists i, n : x = C_i(y_1, \dots, y_n)$$

4. *Selection*: с помощью паттерн-матчинга можно получить нужный элемент

$$\exists s_i^k : s_i^k(C_k(x_1^k, \dots, x_n^k)) = x_i^k$$

Способы задизайнить выражение

Variant 1

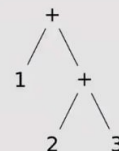
```
data Expr =
  | Const Int
  | VarCalledX
  | Plus Expr Expr
  | Asterisk Expr Expr
  | Dash Expr Expr
  | Slash Expr Expr
```

Variant 2

```
data Op = Plus | Asterisk
  | Dash | Slash
data Expr =
  | Const Int
  | VarCalledX
  | BinOp Op Expr Expr
```

Looks exactly like AST

```
Plus (Const 1,
  Plus (Const 2,
    Const 3))
```



(2): вводим определение операции `Op`

(3): представляем выражение в виде дерева

При выборе дизайна нужно учитывать, что создание элемента должно быть безопасным. Можно кидать исключения, но лучше в принципе не дать возможности создать "опасный" элемент.

let в коде

```
1 isSquare :: Shape -> Bool
2 isSquare (Rectangle (PointD x0 y0) (PointD x1 y1)) =
3     let x = abs(x0 - x1) in
4     let y = abs(y0 - y1) in
5     x == y
```

case и where в коде

```
1 slideShape :: Shape -> PointT -> Shape
2 slideShape shape point =
3     case shape of
4         Circle center radius -> Circle (slidePoint center point) radius
5         Rectangle left right -> Rectangle (slidePoint left point) (slidePoint right point)
6     where
7         slidePoint :: PointT -> PointT -> PointT
8         slidePoint (PointD x0 y0) (PointD x1 y1) = PointD (x0 + x1) (y0 + y1)
```

Тут написано

```
1 case shape of
2     ...
3 where
4     ...
```

Аналогично можно написать, используя let

```
1 let slidePoint :: ... in
2     slidePoint ... = ... in
3 case shape of
4     ....
```

```
1 a -> b -> c == a -> (b -> c)
2 -- функция принимает аргументы a, b и возвращает c
3
4 (a -> b) -> c
5 -- функция принимает в качестве аргумента функцию и возвращает c
```


Cons aka AtLeastOne, Nil aka Empty

```
1 module List where
2
3 data List a = Nil | Cons a (List a)
4 data List a = Empty | AtLeastOne a (List a)
```

`Nil` <=> `Empty` <=> пустой список

`Cons` <=> `AtLeastOne` <=> конструктор списка

Maybe

```
1 safeHead :: List a -> Maybe a
2 safeHead Empty = Nothing
3 safeHead (AtLeastOne x _) = Just x
4
5 safeTail :: List a -> Maybe (List a)
6 safeTail Empty = Nothing
7 safeTail (AtLeastOne _ xs) = Just xs
```

`Maybe` значит, что тип у переменной либо `Just`, либо `Nothing` (`= null`).

`_`, чтобы не было имен, которые не используются внутри функции.

Внимание: функция теперь возвращает не число, а возможно-число. Если в дальнейшем хочется его использовать, нужно аккуратно вынимать значение.

map'

Наивное решение

```
1 double :: List a -> List a
2 double (AtLeastOne x xs) = AtLeastOne (x * 2) xs
3 double Empty = Empty
4
5 triple :: List a -> List a
6 triple (AtLeastOne x xs) = AtLeastOne (x * 3) xs
7 triple Empty = Empty
```

Продвинутое решение

```
1 map' :: (Int -> Int) -> List a -> List a
2 map' f (AtLeastOne x xs) = AtLeastOne (f x) xs
3 map' f Empty = Empty
4
5
6 -- Пример для map'
7 double' xs = map' doubleInt xs
8   where
9     doubleInt x = x * 2
10
11
```

```

12  -- аналогичное определение double'
13  double' xs = map' (*2) xs
14          -- аргумент, переданный в f, умножается на 2

```

`map'` получает на вход функцию `f` и список. Функция `f` получает `Int` и возвращает `Int`, об этом говорит `(Int -> Int)` в первой строчке.

fold

```

1  sumListUp :: List Int -> Int
2  sumListUp Empty = 0
3  sumListUp (AtLeastOne x xs) = x + sumListUp xs
4
5  MultListUp :: List Int -> Int
6  MultListUp Empty = 1
7  MultListUp (AtLeastOne x xs) = x * MultListUp xs
8
9  fold :: (Int -> Int -> Int) -> Int -> List Int -> Int
10 fold f acc (AtLeastOne x xs) = fold f (x `f` acc) xs
11 fold f acc Empty = acc

```

`fold` принимает на вход функцию от двух аргументов, "аккумулятор", список интов

`x 'f' acc <=> x` взаимодействует с `acc`

```

1  ghci> fold (+) [1, 2, 3, 4, 5]
2  ghci> 15
3  ghci> fold (*) [1, 2, 3, 4, 5]
4  ghci> 120
5  ghci> fold (\x y -> x * 2 + y) [1, 2, 3, 4, 5]
6  ghci> 15

```

`\x y -> x * 2 + y` -- лямбда-функция

Анонимная лямбда-функция

Синтаксис: `\x -> 2 * x`. `\` означает, что дальше идет лямбда-функция

```

1  map' :: (a -> b) -> List a -> List b
2  map' f Empty = Empty
3  map' f (AtLeastOne x xs) = AtLeastOne (f x) (map f xs)
4
5  double' xs = map' (\x -> 2 * x) xs

```

Сборка проекта

`stack build` -- собрать проект

`stack test` -- собрать тесты

22-09-19

```
1 f a b = a + b
2 a `f` b = a + b
3 -- одно и то же
```

HOF (higher order functions)

Higher order function -- функция, которая получает другую функцию в качестве аргумента или возвращает функцию (currying).

1. Функция как аргумент

```
1 foo :: (Int -> Int) -> Int -> Int
2 foo bar x = x + bar x
3
4 ghci> foo (\x -> x * x) 3
5 ghci> 12
```

2. Currying

По сути, это подстановка функции вместо слова.

`sum x` \Leftrightarrow `helper 0 x`

```
1 sum = helper 0 where
2     helper acc x | x > 0 = helper (acc + x * x) (x - 1)
3                     | otherwise = acc
4
5 ghci> sum 4
6 ghci> 30    -- 1 + 4 + 9 + 16
```

3. Композиция

```
1 f . g . h . e $ x    --    <=>    (f . g . h . e) x    <=>    f(g(h(e(x))))
```

`.` == композиция функций

`$` == подставь `x` как аргумент в функцию `e` (`$` баксик 0_0)

Associativity and priority. Function application

Associativity and Priority

```
infixl 9 !!
infixr 9 .
infixr 8 ^, ^^, **
infixl 7 *, /, `quot`, `rem`,
`div`, `mod`
infixl 6 +, -
infixr 5 ++, :
infix 4 ==, /=, <, <=, >=, >,
`elem`, `notElem`
infixr 3 &&
infixr 2 ||
infixl 1 >>, >=>
infixr 1 <<=
infixr 0 $, $!, `seq`
```

Function application

> Function application
highest priority, a-la 10

> \$ application

```
infixr 0 $
f $ x = f x
f (g x (h y)) ≡ f $ g x $ h y
```

> Function composition

```
infixr 9 .
f . g = \ x -> f (g x)

f(g(h(e(x)))) ≡
f . g . h . e $ x
```

Lazy evaluation

Lazy lists

```
1 | let ones = 1 : ones      -- если захотим вывести, то зациклимся
2 | ghci> take 5 ones      -- не зациклимся
```

Strict lists

Нужно расширение `XBangPatterns`

`!!` -- конструктор для строгих списков. Отключает ленивое вычисление и заставляет досчитать до конца прежде, чем переходить к следующей строке.

```
1 | data List a = Nil | Cons a (List a)
2 | let onesS = 1 !! onesS
3 |
4 | ghci> take 5 onesS      -- Error: CInterrupted, потому что невозможно сразу
5 |                        -- вычислить список onesS
```

NF, WHNF: (weak head) normal form

NF -- нормальная форма. Выражение имеет нормальную форму, если нельзя сделать какие-то вычисления над ним или над его фрагментом.

```
1 | -- In normal form:
2 | 42
3 | (2, "hello")
4 | \x -> x + 1
5 |
6 | -- Not in normal form:
7 | 1 + 2
8 | "he" ++ "llo"
9 | (\x -> x + 1) 2      -- аналог `f 2`. Можно применить функцию к `2`
```

WHNF -- слабая нормальная форма. Выражение имеет слабую нормальную форму, если это "последняя стадия" вычисления выражения. NF является WHNF.

В чем по сути разница, не понятно. Да и не важно, в принципе.

Const

Встроенная функция, которая игнорирует второй аргумент и возвращает первый

```
1 | const 42 undefined
2 | ghci> 42
3 |
4 | f 42 $ undefined    -- f x == f $ x
5 | ghci> 42
6 |
7 | f 42 !$ undefined   -- `!` значит вычислить выражение по значению
8 |                    -- Нужно сначала посчитать правый аргумент, потом функцию `f`
```

| (guards)

```
1 | factorial = helper 1 where
2 |     helper acc k | k > 1 = helper (acc * k) (k - 1)
3 |                   | otherwise = acc
4 |
5 | ghci> factorial 5
6 | ghci> 120
```

| аналогично if

Замечание к оознанию кода: `factorial = helper 1 <=> factorial x = helper 1 x`. То есть когда мы вызываем `factorial 5`, вместо `factorial` подставляется `helper 1`. Получаем `helper 1 5` и работаем.

Эта техника называется Currying (см. 22-09-26. Currying).

:+

```
1 | data Complex a = !a :+ !a <=> data Complexgt a = Pair !a !a
```

```
1 | case (1, undefined) of (_,_) -> 42
2 | ghci> 42
3 |
4 | case (1, undefined) of (_ :+ _) -> 42
5 | ghci> Error    -- потому что в определении `:+` второй аргумент должен быть вычислен
```

Lazy patterns

Значит "что бы тебе ни передали, подставь в функцию"

```
1 | h (a, b) = g a b    -- strict: проверит, что действительно передали пару (a, b)
2 | h ~(a, b) = g a b   -- lazy: не проверит
```

Benchmarking

Всякие тесты, чтобы посмотреть, как и сколько работает программа. Строит графики

`:set`

`:set +s` можно посмотреть, сколько времени работает программа. Есть и другие флаги.

foldl, foldr

Левая и правая свёртки.

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr _ acc [] = acc
3 foldr f acc (x : xs) = x `f` foldr f acc xs
4
5 foldl :: (b -> a -> b) -> b -> [a] -> b
6 foldl _ acc [] = acc
7 foldl f acc (x : xs) = foldl f (acc `f` x) xs
```

Функция буквально подставляется к каждому элементу

```
1 foldr f acc [1, 2, 3, 4] = f 1 (f 2 (f 3 (f 4 acc)))
2 foldl f acc [5, 6, 7, 8] = (((acc `f` 5) `f` 6) `f` 7) `f` 8
```

Filter

Получаем на вход список, и оставляет элементы, удовлетворяющие какому-то условию

```
1 filter :: (a -> Bool) -> [a] -> [a]
2
3 ghci> filter odd [1, 2, 3, 4, 5]
4 ghci> [1, 3, 5]
5 ghci> filter (\x -> length x > 3) ["a", "bb", "ooo", "bbbb", "aaaaa"]
6 ghci> ["bbbb", "aaaaa"]
```

Симбиоз pattern matching и |

```

1 eqList :: [Int] -> [Int] -> Bool
2 eqList [] [] = True
3 eqList (x : xs) (y : ys) | x == y = True
4                           | otherwise = False
5 eqList _ _ = False
6
7 -- Если первые два случая не подошли, то всегда возвращается False, поэтому ветка `otherwise`
  является лишней.
8
9 -- Более короткий вариант:
10
11 eqList [] [] = True
12 eqList (x : xs) (y : ys) | x == y = True
13 eqList _ _ = False

```

List comprehension

Способ создавать список на основе другого списка

Бесконечный список от a: `[a..]`

Диапазон от a до b: `[a..b]`

Диапазон от a до c с шагом (b - a): `[a, b..c]`

```

1 b = [2 * i | i <- [0..5]]
2 ghci> [0,2,4,6,8,10]
3
4 c = [(i, j) | i <- [0..1], j <- ['a'..'c']]
5 ghci> [(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')]

```

Можно еще дописать условие и выбрать нужное

```

1 [(a, b, c) | a <- [1..10], b <- [1..a], c <- [1..b], a^2 == b^2 + c^2]
2 ghci> [(5,4,3), (10,8,6)]

```

22-09-26

Tail recursion

Отличается от обычной тем, что в момент возвращения ничего не нужно делать дополнительно. То есть вызов рекурсии стоит на последнем месте.

Например, в `usual` нужно сначала утопиться в рекурсии, а потом в начало списка добавить элемент `(x : make x (n-1))`. Элемент лежит на стеке, а стека может и не хватить.

В `tail` на стеке ничего дополнительно не хранится. Сначала добавляем элемент, потом топимся в рекурсии \Rightarrow `acc` обновляется во время вызова рекурсии.

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
          else x : make x (n-1)
```

Tail recursion

```
make2 :: a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x : acc) (n-1)
```

Пример с практики

```
1  -- плебейская рекурсия:
2  rev :: [a] -> [a]
3  rev [] = []
4  rev (h : t) = rev t ++ [h]
5
6  {-
7  rev [1, 2, 3]
8      rev [2, 3] then ++ [2]
9          rev [3] then ++ [2]
10             rev [] then ++ [3]
11
12  Рекурсия запускается, штуки для `++` лежат на стеке. Долго
13  -}
14
15  -- хвостовая рекурсия:
16  rev' :: [a] -> [a]
17  rev' xs =
18      go [] xs
19  where
20      go acc [] = acc
21      go acc (h : t) = go (h : acc) t
```

Хвостовая рекурсия быстрее, потому что она добавляет элемент за константу, а не за размер списка.

Records

Расширение для `unions`, позволяющее давать имена полям.

RecordWildCards == pattern patching в unions. Вместо `..` можно подставить любое имя.

Usual Union Data Definition

```
data SimplePerson = SimplePerson String String String Int String
firstPerson = SimplePerson "Alan" "Smith" "asmith@gmail.com" 42 "Lawyer"
incomplete = SimplePerson "Michael" "Smith" "msmith@gmail.com" 42
complete = incomplete "Dancer"
```

Records

> are an extension of union ADT that allow fields to be named:

```
data Person = Person
{ age :: Int
, name :: String
}
ghci> a = Person 3 "a"
ghci> a
Person {age = 3, name = "a"}
ghci> age a
3
ghci> b = a {name = "BB"} --"update"
ghci> b
Person {age = 3, name = "BB"}
```

ghci> growUp person =
person {age = age person + 1}
ghci> c = growUp a
ghci> c
Person {age = 4, name = "a"}

RecordWildCards

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) =
map toLower name
lowerCaseName (Person { .. }) =
map toLower name
f (Person {age = 3, ..}) = name++"b"
```

Практика 22-09-29

curry & uncurry \todo

`curry` -- функция, разворачивающая туппл от два аргументов в просто два аргумента

`uncurry` -- функция, которая к двум значениям применяет `f` и возвращает одно значение

```
1 | uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

Это функция `(a -> b -> c)`, которая возвращает функцию `((a, b) -> c)`.

Один аргумент или два?

В хаскелле функция от нескольких переменных -- это, на самом деле, функция от одного аргумента, возвращающая функцию

```
1 | f :: a -> a -> a
2 | f x y = x * y
3
4 | g :: a -> (a -> a)
5 | g x = \y -> x * y
6
7 | -- `:type g` и `:type g` одинаковые
```

zip

получает два списка и возвращает список из пар

```
1 | zip :: [a] -> [b] -> [(a, b)]
2
3 | ghci> zip [1..] ['a', 'b', 'c']
4 | ghci> [(1, 'a'), (2, 'b'), (3, 'c')]
```

zip3

получает три списка и возвращает тупл

zipWith

получает функцию и два списка

```
1 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
2
3 ghci> zipWith (+) [1, 2, 3] [4, 5, 6]
4 ghci> [5,7,9]
```

Пример с композицией: squareEven

```
1 squareEven :: [Int]
2 squareEven = map (^2) (map (*2) [0..]) -- обходим список два раза
3
4 squareEven = map (\x -> (x * 2)^2) [0..] -- обходим список один раз
5
6 powEven :: Int -> [Int]
7 powEven n = map ((^n) . (*2)) [0..] -- == f(g(x)) -- композиция функций, теперь умножение и
8 --          f          g
   степень независимые
```

```
1 map2funcs f g xs = map (f . g) xs
2
3 ghci> map2funcs (^2) (*2) [0..]
```

class Eq \todo

```
1 class MyEq a where
2     equals :: a -> a -> Bool
3     nequals :: a -> a -> Bool
4
5     nequals x y = not & equals x y -- выразили `equals` через `nequals`, но
6                                     -- не дали реализацию ни для одной из функций
7
8
9 instance MyEq Bool where -- реализуем инстанс `Bool`
10     equals :: Bool -> Bool -> Bool
11     equals True x = x -- принимаем True и какой-то аргумент. Вернем True, только если x == True
12     equals False x = not x -- принимаем False и аргумент. Вернем True, только если x == False
13
14
15
16 eqList :: MyEq a => List a -> List a -> Bool -- сравниваем списки на равенство
17 eqList Nil Nil = True
18 eqList (x : xs) (y : ys) = equals x y && eqList xs ys
19 eqList _ _ = False
```

Практика 22-10-06

```
1 | import qualified Data.Set as Set
```

Чтобы компилилось, в файлике `package.yaml` добавляем `containers` в `dependencies`

```
1 | ...
2 |
3 | library:
4 |   source-dirs: ...
5 |   dependencies:
6 |     - containers
7 |
8 | ...
```

22-10-10

Linked list

При конкатенации списков левый копируется. Поэтому слева должен быть список меньшей длины

```
1 | xs ++ (ys ++ zs)
```

`ys` копируется, потом `xs` копируется

```
1 | (xs ++ ys) ++ zs
```

`xs` копируется, потом `(xs + ys)` копируется, получается дольше

Memoization in Haskell

Мемоизация -- сохранение результатов для жорогих вызовов функций.

Так вот в хаскелле её нет.

Streams

aka Lazy lists. Список, в котором операции отложенные.

В данном случае будем использовать `$` (баксик!!), чтобы показать, что что-то ленивое.

`zip :: stream x stream -> stream` -- суммирует стримы поэлементно

```
1 | fibs = $Cons(1, $Cons(1, zip(fibs, tail(fibs))))
```

Pure Functional Queues

```
1 | empty :: queue -> bool
2 | enqueue :: queue -> int -> queue    -- adds elem into a queue
3 | head :: queue -> int
4 | tail :: queue -> queue
```

Simplest implementation of a queue

Разделяем список пополам. `f` -- первая часть списка в обычном порядке, `r` -- вторая часть списка в обратном порядке.

Инвариант: $|f| \geq |r|$ -- `f` станет пустым, только если `r` пустой.

Пример:

```
1 | queue = [1, 2, 3, 4, 5, 6]
2 | f = [1, 2, 3]
3 | r = [6, 5, 4]
```

Можно её улучшить, если вместо списков использовать стримы, сохраняя размер текущего стрима. Добавим мемоизацию для дорогих вызовов функций.

Практика 22-10-13

Паттерн `node@(Node x)`

```
1 foo :: Tree a -> Int -> Tree a
2 ...
3 foo aboba@(Leaf x) = aboba
4 -- <=> `foo (Leaf x) = Leaf x`
```

Просто дали имя переданному аргументу. Вместо того, чтобы возвращать конструктор от значения, возвращаем результат вычисления аргумента.

Strictness

Если все поля должны вычисляться до конца, а ставит восклицательные знаки лень, можно подключить расширение `StrictData`:

```
1 { -# LANGUAGE StrictData #- }
```

Тогда все поля становятся строгими.

`seq`

Берет два аргумента и заставляет первый аргумент высчитывать до конца. Используется, чтобы избежать ленивости.

```
1 sumBest :: Int -> Int
2 SumBest n =
3     go 0 n
4     where
5         go acc 0 = acc
6         go acc n = acc `seq` go (acc + n) (n - 1)
```

Вычисление `acc` не будет отложенным.



Форсирует вычисление аргумента. В идеале должен работать так же, как `seq`.

Нужно подключить расширение:

```
1 { -# LANGUAGE BangPatterns #- }
```

```

1 sumBest :: Int -> Int
2 sumBest n =
3     go 0 n
4     where
5         go acc 0 = acc
6         go acc n = go ($! (acc + n)) (n - 1)

```

Laziness

~

Использование для аргументов

```

1 { -# LANGUAGE StrictData #- }
2
3 data Tree a = Leaf a | Node a ~(Tree a) ~(Tree a)

```

Подключено расширение \Rightarrow все поля строгие. ~ делает вычисления поддеревьев ленивыми.

Использование для функций

```

1 foo ~(Just x, Just y) = Just (x + y)
2 foo _ = Nothing

```

Если без ~, то первая сигнатура foo будет использоваться, только если переданы в точности (Just x, Just y).

Если есть ~, то всегда будет использоваться первая сигнатура вне зависимости от типов переданных аргументов. Вторая сигнатура никогда не используется.

Моноиды

Data.Monoid

Моноиды -- типы с ассоциативными бинарными операциями. Есть функции mempty и mappend.

mempty -- как нейтральный элемент в моноиде.

Инстансы моноидов должны удовлетворять условиям:

- mappend mempty x = x
- mappend x mempty = x
- mappend x (mappend y z) = mappend (mappend x y) z
- mconcat = foldr mappend mempty

```

1 module Monoid where
2
3 class MyMonoid a where
4     mempty :: a
5     mappend :: a -> a -> a
6
7 instance MyMonoid [a] where
8     mempty = []
9     mappend = (++)
10
11 instance MyMonoid Int where
12     mempty = 0      -- начальное значение для суммы
13     mappend = (+)   -- будем складывать числа

```

А что, если хочется два инстанса для типа `Int`? Тогда создадим специальные типы:

```

1 data Sum = Sum Int deriving Show
2 data Prod = Prod Int deriving Show
3
4 instance MyMonoid Sum where
5     mempty = Sum 0
6     mappend (Sum x) (Sum y) = Sum (x + y)
7
8 instance MyMonoid Prod where
9     mempty = Prod 1
10    mappend (Prod x) (Prod y) = Prod (x * y)

```

В современных версиях ghc `Monoid` не содержит операцию `mappend`, вместо неё используется `<>` (живет в классе `Semigroup`):

```

1 class MyMonoid a where
2     mempty :: a
3     (< >) :: a -> a -> a
4
5 instance MyMonoid [a] where
6     mempty = []
7     (< >) = (++)

```

22-10-17

Лямбда-исчисления

$$\Lambda ::= \underbrace{v}_{\text{переменная}} \mid \underbrace{\Lambda \Lambda}_{\text{применение функции}} \mid \underbrace{\lambda v. \Lambda}_{\lambda\text{-абстракция}}$$

Обозначения

1. v -- переменная.
2. "Создание функции":
 $\lambda v. \Lambda$ -- лямбда абстракция от Λ по v .

v -- название аргумента функции.

Λ -- лямбда выражение -- тело функции.

Пример: $(\lambda x . * 2 x)$ -- функция, принимающая x и возвращающая $2x$.

3. "Применение функции":

$M N$ -- применение функции M к аргументу N .

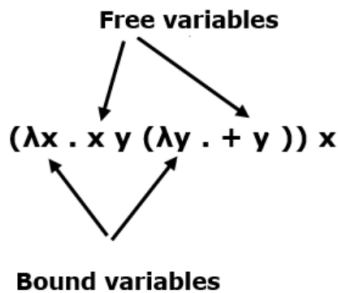
4. $[x/M]$ -- подставили значение M вместо переменной x .

Свободные и связанные переменные

$FV(T)$ -- free variables -- **свободные переменные** -- переменные внутри тела функции. Вместо них можно что-то подставить.

$BV(T)$ -- bound variables -- **связанные переменные** -- переменные, являющиеся аргументами в объявлении функции. Вместо них нельзя что-то подставить.

Пример



Правила \todo

$$FV(x) = \{x\}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\}$$

$$BV(x) = \emptyset$$

$$BV(e_1 e_2) = BV(e_1) \cup BV(e_2)$$

$$BV(\lambda x . e) = BV(e) \cup \{x\}$$

def. (замкнутое множество)

T называется *замкнутым*, если $FV(T) = \emptyset$

α -conversion

Договорились, что если у двух абстракций имена аргументов одинаковые, то это не значит, что сами переменные -- это одно и то же.

α — *conversion*: если есть вложенные абстракции, то у вложенной можно поменять название аргумента

$$\lambda x. \Lambda[x] \xrightarrow{\alpha} \lambda y. \Lambda[y]$$

Пример

$$\lambda x. (\lambda x. + (-x 1)) x 3 9 = \lambda x. (\lambda y. + (-y 1)) x 3 9 = \lambda x. (+ (-x 1) 3) 9 = + (-9 1) 3 = + 8 3 = 11$$

β -reduction

Вместо формального аргумента можно подставить значение.

$$(\lambda x. e_1) e_2 \xrightarrow{\beta} e_1[x/e_2]$$

Примеры

1. $(\lambda y. (\lambda x. x y)) x \xrightarrow{\beta} (\lambda x. x y)[y/x]$ -- вместо y подставляется x

2.

$$\lambda x. (\lambda x. + (-x 1)) x 3 9 \underset{\text{подставили } x \text{ вместо } x}{=} \lambda x. (+ (-x 1) 3) 9 \underset{\text{подставили } 9 \text{ вместо } x}{=} + (-9 1) 3 = + 8 3 = 11$$

Capture-avoiding substitution \todo

Определяет статическое связывание переменных.

$$x[x/M] = M$$

$$y[x/M] = y \text{ if } y \neq x$$

$$(e_1 e_2)[x/M] = (e_1[x/M])(e_2[x/M])$$

$$(\lambda x. e)[x/M] = \lambda x. e$$

$$(\lambda y. e)[x/M] = \lambda y. e[x/M], \text{ if } x \neq y \ \&\& \ y \notin FV(M)$$

$$(\lambda y. e)[x/M] =$$

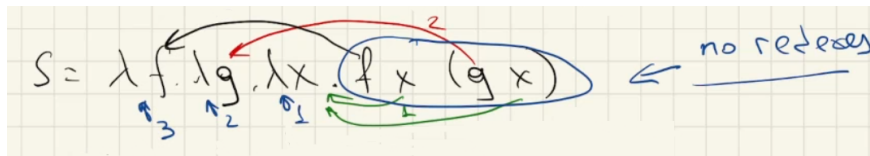
Последовательность де Брёйна (?) \todo

$$\Lambda^d ::= N[\Lambda^d \Lambda^d] \lambda. \Lambda^d$$

Теперь индексы отвечают за то, куда что подставлять

Пример

$$s = \lambda f. \lambda g. \lambda x. f x (g x) = \lambda. \lambda. \lambda. 3 1 (2 1)$$



Практика 22-10-20

функция `bind`

Чтобы не копипастить `case` внутри функций, создадим вспомогательную функцию `bind`:

```

1 Data Expr = Val Double | Div Expr Expr | Log Expr
2
3 bind :: Maybe a -> (a -> Maybe b) -> Maybe b
4 bind value f =
5     case value of
6         Just x -> f x
7         Nothing -> Nothing
8
9 eval :: Expr -> Maybe Double
10 eval (Val n) = Just n
11 eval (Div x y) =
12     eval x `bind` \x' ->
13     eval y `bind` \y' ->
14     totalDiv x' y'
15 eval (Log x) =
16     eval x `bind` \x' -> totalLog x'
17
18
19 totalLog :: Double -> Maybe Double
20 totalLog n | n <= 0 = Nothing
21             | otherwise = Just log(n)
22
23 totalDiv :: Double -> Double -> Maybe Double
24 totalDiv _ 0 = Nothing
25 totalDiv x y = Just x / y

```

Монады

`Monad` -- какой-то контейнер с переменной.

```

1 return :: a -> Monad a
2 return = pure

```

Заворачивает `a` в монаду

`pure` означает, что функция не меняет состояние переданной переменной

Правила монадов

- left identity: `return a >>= k = k a`
- right identity: `m >>= return = m`
- associativity: `m >>= (\x -> k x >>= h) = (m >>= k) >>= h`

Maybe и >>=

`Monad a` достаёт значение из коробки. Если получилось, применяется функция `a -> Monad b`.

```
1 instance Monad Maybe where
2   (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
3   Nothing >>= f = Nothing
4   (Just x) >>= f = f x
```

Lists

```
1 instance Monad [] where
2   (>>=) :: [a] -> (a -> [b]) -> [b]
3
4   -- способ 1
5   xs >>= f = concat (map f xs)
6
7   -- способ 2
8   xs >>= f = [y | x <- xs, y <- f x]
9
10  -- способ 3
11  xs >>= f =
12    case xs of
13      (h : t) -> f h ++ t >>= f
14      [] -> []
15
16
```

проблема `map` в том, что она вернет `[[b]]`. Функция `concat` склеит полученные списки.

Пример использования монадов

```
1 Data Expr = Val Double | Div Expr Expr | Log Expr
2
3 eval :: Expr -> Maybe Double
4 eval (Val n) = Just n
5 eval (Div x y) =
6   eval x >>= \x' ->
7   eval y >>= \y' ->
8   safeDiv x' y'
9 eval (Log x) =
10  eval x >>= (\x' -> saveLog x')
```

```

11
12
13 saveLog :: Double -> Maybe Double
14 saveLog n | n <= 0 = Nothing
15           | otherwise = Just log(n)
16
17 safeDiv :: Double -> Double -> Maybe Double
18 safeDiv _ 0 = Nothing
19 safeDiv x y = Just x / y

```

do-нотации

Синтаксический сахар, который позволяет записать штуки с `>>=` более читабельно.

```

1 thing1 >>= (\x ->
2   func1 x >>= (\y ->
3     thing2 >>= (\_ ->
4       func2 y >>= (\z ->
5         return z))))

```

Может быть переписано с использованием `do`:

```

1 do {
2   x <- thing1;
3   y <- func1 x;
4   thing2;
5   z <- func2 y;
6   return z
7 }

```

Если `thing1 = Nothing`, выходим из `do`-блока.

IO()

Специальная монада, чтобы общаться с внешним миром.

```

1 module Main (main) where
2
3 main :: IO()
4 main = do
5   putStrLn "Enter your name:"
6   name <- getLine  -- достаем строку из `getLine` и
7                   -- записываем её в переменную `name`
8   putStrLn $ "Hello, " ++ name ++ "!"
9

```

read

Позволяет прочитать значение переменной как инстанс определенного типа

```
1 | let aboba = "123"
2 | let aboba_int = read aboba :: Int -- `aboba_int` теперь чиселко
```

Пример

```
1 | module Main (main) where
2 |
3 | main :: IO()
4 | main = do
5 |     putStrLn "Enter your name:"
6 |     name <- getLine
7 |     putStrLn $ "Hello, " ++ name ++ "!"
8 |     putStrLn "Enter your age:"
9 |     arg <- getLine
10 |    let age = read arg :: Int
11 |    putStrLn $ "You will be " ++ (show $ age + 1) ++ " in the next year!"
```

Практика 22-10-24

Кодирование Чёрча: способ 1

Элементарные операции

True: $\lambda a . \lambda b . a$

False: $\lambda a . \lambda b . b$

Not: $\lambda p . p \text{ False True}$ (Вместо p подставляем абстракцию **True** или **False**, внутри которой меняем аргументы местами)

And: $\lambda p . \lambda q . p \ q \text{ False}$

Or: $\lambda p . \lambda q . p \text{ True } q$

IfThenElse: $\lambda p . \lambda t . \lambda e . p \ t \ e \ (p == \text{True} \Rightarrow \text{вернется } t, \text{ иначе вернется } e)$

isZero

$0 = \lambda s . \lambda z . z$

$1 = \lambda s . \lambda z . s \ z$

$2 = \lambda s . \lambda z . s \ (s \ z)$

$3 = \lambda s . \lambda z . s \ (s \ (s \ z))$

$n = \lambda s . \lambda z . s^n \ z$

isZero = $\lambda n . n \ (\lambda x . \text{False}) \text{ True}$

$$\text{isZero}(0) = (\lambda n . n (\lambda x . False) True) (\lambda s . \lambda z . z) = (\lambda s . \lambda z . z) (\lambda x . False) True = True$$

$$\text{isZero}(1) = (\lambda n . n (\lambda x . False) True) (\lambda s . \lambda z . s z) = (\lambda x . False) True = False$$

22-10-24

Кодирование Чёрча: способ 2

$d_0 = \lambda x . x$ -- выступает в роли нуля или пустого множества

$d_{n+1} = [False, d_n]$ -- пара переменных

$$True = \lambda x . \lambda y . x$$

$$False = \lambda x . \lambda y . y$$

$$[u, v] = \lambda x . (x u) v. \quad [u, v] True = u, \quad [u, v] False = v$$

$\text{if } B \text{ then } U \text{ else } V = B U V$ (определяем так, потому что подразумевается, что B возвращает либо $True$, либо $False$)

$$\text{isZero} = \lambda x . x True$$

$$\text{isZero}(0) = (\lambda y . y) True = True$$

$$\text{isZero}(2) = [False, [False, \lambda y . y]] True = False$$

$$\text{prev} = \lambda x . x False$$

$$\text{prev}(0) = False$$

$$\text{prev}(5) = [False, d_4] False = d_4$$

Функции

Базисные

Функции вида $f : N^k \rightarrow N$

$z : N \rightarrow 0 \iff \lambda x . d_0$ -- принимает натуральное число и возвращает ноль. $z(x) = d_0$

$s : N \rightarrow N \iff \lambda x . [False, x]$ -- возвращает следующее число. $s(d_n) = d_{n+1}$

$i_k^n : N^n \rightarrow N \iff \lambda x_1 \dots \lambda x_n . x_k$ -- получает ряд натуральных чисел и возвращает одно из них.
 $i_k^n(x_1, \dots, x_n) = x_k$

Композиция

$$g : N^m \rightarrow N$$

$$f_1, \dots, f_m : N^n \rightarrow N$$

$$h(x_1, \dots, x_n) = g(f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n))$$

$$G \rightsquigarrow g; \quad F_1, \dots, F_m \rightsquigarrow f_1, \dots, f_m; \quad H \rightsquigarrow h$$

$$H = \lambda x_1 \dots \lambda x_n. G (F_1 x_1 \dots x_n) \dots (F_m x_1 \dots x_n) \iff h(x_1, \dots, x_n)$$

Примитивно-рекурсивные (ПРФ)

$$\begin{cases} f(0, x) = g(x) \\ f(n+1, x) = h(f(n, x), n, x) \end{cases}$$

Ну типа просто рекурсия, которая всегда завершается. По построению определена во всех аргументах *натуральных чисел*. Каждой функции соответствует какой-то λ -терм.

$$F \rightsquigarrow f$$

$$F = \lambda y. \lambda x_1 \dots \lambda x_k. \text{ if } isZero(y) \text{ then } G x_1 \dots x_k \text{ else } H (F (y-1) x_1 \dots x_k) (y-1) x_1 \dots x_k$$

Тут используется рекурсия, но мы не знаем, что такое рекурсия. Но с помощью Y -комбинатора (см. ниже) и первой теоремы рекурсии (см. ниже) мы можем определить рекурсию.

Частично-рекурсивные (ЧРФ)

$$\text{Как ПРФ, но еще есть оператор минимизации } \mu : \mu z. g(t, x) = \min\{z \mid g(z, x) = 0\}$$

Оператор не всегда существует, потому что g может не принимать значение 0 или вообще заикнуться на каком-то вызове.

$$\text{ЧРФ} = \text{ПРФ} + \mu$$

Th. (о реализуемости)

$$f : N^k \rightarrow N \text{ -- частичная функция, представима в виде } \lambda\text{-терма } f(n_1, \dots, n_k) = m$$

$$f \text{ вычислима} \iff f = ((False\ d_{n_1})\ d_{n_2} \dots) \xrightarrow{\beta} d_m$$

Th. (о неподвижной точке λ -терма)

$$\forall F \exists V : V = F V$$

Для любого терма существует другой, являющийся неподвижной точкой. Равенство означает, что работает в обе стороны

Proof: $V = (\lambda x . F (x x)) (\lambda x . F (x x))$

$(\lambda x . F (x x)) (\lambda x . F (x x)) \Leftrightarrow F ((\lambda x . F (x x)) (\lambda x . F (x x))) \Leftrightarrow F V$

Th. (про Y-комбинатор)

$\exists Y : \forall F \quad Y F = F (Y F)$
 β

Proof: $Y = \lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$

Th. (первая теорема рекурсии)

$\forall M \exists F : F = M[f/F]$
 β

Proof: $F = Y (\lambda f . M)$, где Y -- Y-комбинатор

$Y (\lambda f . M) = (\lambda f . M) (Y (\lambda f . M))$ (по Th.) $= M[f / Y (\lambda f . M)] = M[f / F]$

Практика 22-11-03

Хотим, чтобы у каждого листа был уникальный ключ

```
1 relabel :: Tree a -> Int -> (Tree (Int, a), Int)
2 relabel (Leaf x) i = (Leaf (i, x), i + 1)
3 relabel (Node l r) i =
4     let (l', i') = relabel l i in
5     let (r', i'') = relabel r i' in
6     (Node (l', r'), i'')
7
8 ghci> relabel tree 0
```

Спрячем элемент, отвечающий за счётчик

```
1 type Counter a = Int -> (a, Int)
2
3 relabel :: Tree a -> Counter (Tree (Int, x))
4 relabel (Leaf x) = \i -> (Leaf (i, x), i + 1)
5 relabel (Node l r) =
6     relabel l `next` \l' ->
7     relabel r `next` \r' ->
8     ret $ Node l' r'
9
10 next :: Counter a -> (a -> Counter b) -> Counter b
11 next f g = \i ->
12     let (a, i') = f i in
13     g a i'
14
15 ret :: a -> Counter a -- "return"-функция
16 ret x = \i -> (x, i)
```


В монадах есть штука `State`, работающая как `Counter`.

```
1 import Control.Monad.State
2
3 type State s a = s -> (a, s)
4
5 relabel :: Tree a -> Int -> Tree (Int, a)
6 relabel tree i =
7     evalState (go tree) i
8     where
9         go :: Tree a -> State Int (Tree (Int, a))
10        go (Leaf x) = do
11            i <- get
12            pur (i + 1)
13            return (Leaf (i, x))
14        go (Node l r) = do
15            l' <- go l
16            r' <- go r
17            return (Node l' r')
```

`evalState` -- игнорирует состояние, возвращает только значение (наверное)

`execState` -- игнорирует значение, возвращает только состояние (наверное)

`runState` -- что-то делает

`get` -- достаёт значение из монады

`put` -- кладёт новое значение.

`modify(*2)` -- увеличит текущий счётчик состояния в 2 раза

`gets` -- используется, если в `State` поддерживают несколько состояний.

22-11-07 \todo

Теорема Чёрча — Россера

$$U \xrightarrow{\beta} M, U \xrightarrow{\beta} N$$

$$\exists K : M \xrightarrow{\beta} K, N \xrightarrow{\beta} K$$



Смысл: как бы мы ни редуцировались, мы всегда можем найти общего потомка.

Правила вывода

1. $x \rightarrow x$
2. $\lambda x . e \rightarrow \lambda x . e$
3.
$$\frac{e_1 \rightarrow \lambda x . e \quad e[x/e_2] \rightarrow e'}{e_1 e_2 \rightarrow e'}$$
4.
$$\frac{e_1 \rightarrow e'_1 : e'_1 \text{ не } \lambda\text{-абстракция}}{e_1 e_2 \rightarrow e'_1 e_2}$$

Практика 22-11-10

class Functor

```
1 class (Functor f) => Applicative f where
2   pure :: a -> f a
3   (<*>) :: f (a -> b) -> f a -> f b
```

fmap

применяет переданную функцию к элементу, если он \neq `Nothing`.

```
1 ghci> fmap (+1) (Right 23)
2 ghci> Right 24
3 ghci> fmap (+1) (Left 42)
4 ghci> Left 42
```

`<$>` -- аналог для `fmap`

```
1 ghci> (+1) <$> (Right 23)
2 ghci> Right 24
```

pure

превращает объект в инстанс данной структуры.

```
1 class (Functor f) => Applicative f where
2   pure :: a -> f a
3
4 instance Applicative Maybe where
5   pure :: a -> Maybe a
6   pure x = Just x
7
8 instance Applicative [] where
9   pure :: a -> [a]
10  pure x = [x]
```

<*>

описывает, как применяется функция к аргументам, в зависимости от их типов.

```
1 class (Functor f) => Applicative f where
2   (<*>) :: f (a -> b) -> f a -> f b
3
4
5 instance Applicative Maybe where
6   <*> :: Maybe (a -> b) -> Maybe a -> Maybe b
7   <*> Nothing _ = Nothing
8   <*> (Just f) x = fmap f x      -- <=>   <*> _ Nothing = Nothing
9                                   --         <*> (Just f) (Just x) = Just (f x)
```

(Just f) -- контейнер с функцией f.

```
1 instance Applicative [] where
2   (<*>) :: [a -> b] -> [a] -> [b]
3   <*> fs xs = [f x | f <- fs, x <- xs]
```

fs -- список функций. Берутся функции из списка, аргументы из списка, и всё применяется друг к другу.

Пример

```
1 ghci> (+) <*> [1, 2, 3] <*> [4, 5]
```

Не работает, ожидается список функций

Рабочие варианты:

```
1 ghci> [(+)] <*> [1, 2, 3] <*> [4, 5]
2 ghci> [5,6,6,7,7,8]
3 ghci> pure (+) <*> [1, 2, 3] <*> [4, 5]
4 ghci> [5,6,6,7,7,8]
```

Хотим создать что-то нестандартное...

Заведём свой тип, в котором <*> не будет возвращать декартово произведение.

```
1 data ZipList a = ZipList [a] deriving (Show)
2
3 instance Functor ZipList where
4   fmap :: (a -> b) -> ZipList a -> ZipList b
5   fmap f (ZipList xs) = ZipList $ fmap f xs
6
7 instance Applicative ZipList where
8   pure :: a -> ZipList a
9   pure x = ZipList $ repeat x -- бесконечный список повторяющихся `x`
10
11   (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
12   ZipList fs <*> ZipList xs = ZipList $ zipWith $ fs xs
```

fs[0] применяется к xs[0], fs[1] применяется к xs[1] и т.д.

```

1 | ghci> pure (,) <*> [1, 2, 3] <*> [4, 5]
2 | ghci> [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
3 | ghci> pure (,) <*> ZipList [1, 2, 3] <*> ZipList [4, 5]
4 | ghci> ZipList [(1, 4),(2, 5)]
5 | ghci> ZipList [(+), (*), (/)] <*> [1, 2, 3] <*> [4, 5, 6]
6 | ghci> ZipList [5, 10, 0.5]

```

newtype

Используется вместо `data` в объявлении нового типа, когда нужна только обёртка над новым типом данных.

Работает для типов данных, у которых один аргумент и один конструктор.

Parser \todo

Получает строку, пытается найти и вернуть какое-то значение

```

1 | module Parser where
2 |
3 | -- 1 + 2 - 3 ==> BinOp Plus (Number 1) (BinOp Minus (Number 2) (Number 3))
4 | data Expr = BinOp Op Expr Expr
5 |           | Number Int
6 |
7 | newtype Parser a = Parser { runParser :: String -> Maybe (String, a) }
8 | Parser :: String -> a
9 |
10 | charA :: Parser Char -- считывает одну заглавную букву A
11 | charA = Parser $ \input ->
12 |

```

22-11-14

Типизация

по Чёрчу -- явно прописывается тип: X^α

по Карри -- тип явно не прописывается, но его можно вывести

Терм в контексте

$$\underbrace{x_1 : \alpha_1, \dots, x_n : \alpha_n}_r \vdash u : \beta$$

Каждому x_i сопоставляется тип α_i . r — это контекст *aka* множество элементов вида $x_i : \alpha_i$.

Правила типизирования

картинка 1

$$\frac{\Gamma \vdash U : \alpha \rightarrow \beta \quad \Gamma, X : \alpha \vdash U : \beta}{\Gamma \vdash UV : \beta} \rightarrow E$$

Примеры

картинка 2

Леммы

1. Если по Чёрчу вывелся тип τ , то по Карри тоже выведется тип τ
2. Если по Карри у U вывелся тип τ , то по Чёрчу существует такой терм V у которого тип τ и V сводится к U

Standard problems

1. Type Checking: $\vdash U : \tau ? \Leftrightarrow$ правда ли у U тип τ ?
2. Type inference: $\vdash U : ? \Leftrightarrow$ какой тип у U ?
3. Type inhabitation: $\vdash ? : \tau \Leftrightarrow$ существует ли терм с типом τ ?

Лемма инверсии

1. $\Gamma \vdash X : \tau \Rightarrow X^\tau \in \Gamma$

Если смогли вывести тип τ , то в Γ есть терм типа τ .

2. $\Gamma \vdash UV : \tau \Rightarrow \exists \sigma : \Gamma \vdash U : \sigma \wedge \Gamma \vdash V : \sigma$

3. Для Карри: если λ -абстракция типизируется, то ее термы тоже типизируются
4. то же самое, но для Чёрча

22-11-17

- Когда можно считать, что программа работает правильно?
- Никогда.

— Это хорошо, что уже сейчас у вас такие пессимистичные взгляды на мир.

Property-based тестирование

Идея:

Есть какое-то свойство, которому удовлетворяет наша программа. Генерируя случайное значение, мы проверяем это свойство.

Используется библиотека `Hedgehog`.

Пример

`Property` -- монада, у которой есть методы, чтобы проверить условие на введенных данных (`assert`, `check`, etc).

```
1 module Test.List where
2
3 import Hedgehog
4 import qualified Hedgehog.Gen as Gen
5 import qualified Hedgehog.Range as Range
6 import Test.Tasty
7 import Test.Tasty.Hedgehog
8
9 import Data.List (sort)
10 import List
11
12 -- Условие: в списке все элементы не больше, чем максимальное
13 prop_maximum :: Property
14 prop_maximum = property $ do
15   list <- forAll $ genList 1 100
16   let maxValue = maximumValue list
17   assert (all (<= maxValue) list)
18
19 genInt :: Gen Int
20 genInt = Gen.Int (Range.constant (-10) 10)
21 -- генерируем чиселки от -10 до 10
22
23 genList :: Gen [Int] -- возвращает сгенерированный список интов
24 genList = Gen.List (Range.constant 1 10) genInt
25 --      тип      кол-во эл-тов      как генерируем
26 --      структуры (от 1 до 10)      элементы
27
28
29
30 -- сами тесты
31 props :: [TestTree]
32 props =
33   [ testProperty "Maximum value is not less than all elements of the list" prop_maximum ]
```

`Hedgehog.Gen.sample genList` -- покажет пример того, что делает функция `genList`.

Range

Минимизация -- уменьшение разницы между значениями `minBound` и `maxBound`.

У `Range` есть три характеристики:

- `minBound`
- `maxBound`
- `origin` -- к какому значению будут сводиться тесты при минимизации. По умолчанию, это левая граница.

Size

Параметр со значением $\in [0, 99]$. Показывает размер теста.

Правая граница растёт \Rightarrow `size` растёт. Эта зависимость может быть линейной (`Range.Linear`) или экспоненциальной (`Range.Exponential`).

Что, если нет явного свойства?

Можно написать примитивную реализацию и сравнивать её результаты с более эффективной.

Пример:

`reverse` глупый за $O(n^2)$

```
1 reverseList :: [a] -> [a]
2 reverseList [] = []
3 reverseList (x : xs) = reverseList xs ++ [x]
```

`reverse` умный за $O(n)$

```
1 fastReverseList :: [a] -> [a]
2 = go []
3 where
4   go acc [] = acc
5   go acc (h : t) = go (h : acc) t
```

```
1 module Test.List where
2
3 import Hedgehog
4 import qualified Hedgehog.Gen as Gen
5 import qualified Hedgehog.Range as Range
6 import Test.Tasty
7 import Test.Tasty.Hedgehog
8
9 import List
10
11
12 genInt :: Gen Int
13 genInt = Gen.Int (Range.constant (-10) 10)
14
15 genList :: Gen [Int]
16 genList = Gen.List (Range.constant 1 10) genInt
```

```

17
18
19 -- Проверяем, что дважды обернув список получаем исходный
20 prop_reverseList :: Property
21 prop_reverseList = property $ do
22     list <- forAll $ genList 1 100
23     reverseList (reverseList list) === list
24
25 -- Проверяем, что быстрая версия обращения списка работает так же, как эталонная
26 prop_fastReverseList :: Property
27 prop_fastReverseList = property $ do
28     list <- forAll genList
29     reverseList list === fastReverseList list
30
31 props :: [TestTree]
32 props =
33     [ testProperty "Reversing list twice gets the input list" prop_reverseList
34     , testProperty "Fast reverse gives the same " prop_fastReverseList
35     ]

```

Пример: проверка min/max элемента в отсортированном списке

```

1 module List where
2
3 -- простая реализация отсортированного списка за O(n)
4 class Ranged t where
5     maxVal :: Ord a => t a -> a
6     minVal :: Ord a => t a -> a
7
8 instance Ranged [] where
9     maxVal xs = go (head xs) (tail xs)
10    where
11        go curMax [] = curMax
12        go curMax (h : t) | h < curMax = go h t
13                           | otherwise = go curMax t
14
15    minVal xs = go (head xs) (tail xs)
16    where
17        go curMin [] = curMin
18        go curMin (h : t) | h < curMin = go h t
19                           | otherwise = go curMin t
20
21
22 -- умная реализация
23 newtype SortedList a = Sorted { getSorted :: [a] } deriving (Show, Eq)
24
25 instance Ranged SortedList where
26     maxVal = last . getSorted
27     minVal = head . getSorted

```

```

1 module Test.List where
2
3 import Hedgehog
4 import qualified Hedgehog.Gen as Gen
5 import qualified Hedgehog.Range as Range
6 import Test.Tasty
7 import Test.Tasty.Hedgehog
8
9 import Data.List (sort)
10 import List

```



```

11
12
13
14 genInt :: Gen Int
15 genInt = Gen.Int (Range.constant (-10) 10)
16
17 genList :: Gen [Int]
18 genList = Gen.List (Range.constant 1 10) genInt
19
20 genSortedList :: Gen (SortedList Int)
21 genSortedList = do
22     xs <- genList
23     return $ Sorted (sort xs)
24
25
26 prop_minSorted :: Property
27 prop_minSorted = property $ do
28     list <- forAll $ genSortedList
29     minVal list ==

```

Генерируем арифметическое выражение

```

1  data Expr = BinOp Op Expr Expr
2             | Number Int
3             deriving (Show, Eq)
4
5  data Op = Plus
6           | Minus
7           | Mult
8           | Div
9           | Pow
10          deriving (Show, Eq)
11
12
13 -- выбирает элемент из множества
14 genOp :: Gen Op
15 genOp = Gen.element [Plus, Minus, Mult, Div, Pow]
16
17 -- хотим создавать выражение рекурсивно (BinOp) или нерекурсивно (Number)
18 -- `Gen.choice` выбирает, что из этого генерировать
19 genExpr = Gen.recursive Gen.choice [numGen] [binOpGen]
20     where
21         numGen = Number <$> Gen.Int (Range.linearFrom 0 (-10) 10)
22         binOpGen = do
23             op <- genOp
24             Gen.subterm2 genExpr genExpr (BinOp op)
25 --             (BinOp op) <=> \l r -> BinOp op l r

```

`Gen.subterm2` рекурсивно генерирует 2 подтерма, потом оборачивает их в третий аргумент и возвращает результат.

Есть еще `Gen.subterm1` и `Gen.subterm3`.