Functional Programing

Author: Daria Shutina

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
`f(x)`	`f x`
`f(x, y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x, g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

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

```
Data Number a = Int
```

`Int` это конструктор

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

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

Comments

```
-- comment in a line

{-
big comment
-}
```

Lists

```
1 : [] -- adding 1 into the begining of an empty list


let b = [1] -- declaring of a variable. 'let' can be omited.
b           -- show value of b


c = [2, 2] ++ b   -- [2, 2, 1]
c = b ++ [3, 3]   -- [1, 3, 3]
c = b ++ [2..6]   -- [1, 2, 3, 4, 5, 6]
aboba = "ab" ++ " " ++ "oba"


c = 5 : 4 : 3 : 2 : b  -- [5, 4, 3, 2, 1]

head c     -- [5]
tail c     -- [1]
length c   -- 5
take 3 c   -- [5, 4, 3]

reverse c  -- [1, 2, 3, 4, 5]
null c     -- False (checks whether c is empty)

maximum c  -- [5]
minimum c  -- [1]
-- minimum in a string is a letter which is the earliest in the alphabet
```

Functions: declaration and definition

```haskell
f :: a -> c
-- gets arg of type 'a' and returns 'c'

g :: a -> b -> c
-- gets args of type 'a', 'b' and returns 'c'
```

```haskell
SimpleFunc :: Int -> Bool

SimpleFunc x =
  if (x == 1)
  then True
  else False
```

Объявляем функцию `SimpFunc`, которая принимает `Int` и возвращает `Bool`. Дальше идёт опредление функции.

Чтобы игнорировать вывод функции `aboba`, можно написать `aboba_`

Алгебраические типы

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

Pattern matching

```haskell
data Cardinal = North | East | South | West

Checker :: Cardinal -> Bool
-- function Checker. It is better to declear a returning type clearly
Checker North = True;
Checker South = True;
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

```
ghc aboba.hs -o a
./a
```

22-09-12

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

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

```
ghci> :k Int
ghci> *

ghci> data PPoint = PPoint a a
ghci> :k PPoint
ghci> PPoint :: a -> a -> a
```

`:t` показывает сигнатуру функции

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

=>

```
dist (Point x1 y1) (Point x2 y2) = sqrt((x1 - x2)^2 + (y1 - y2)^2)

ghci> :t dist
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:

```
data Point = Point2D Double Double | Point3D Double Double Double     -- union type

pointToList :: Point -> [Double]
pointToList (Point2D x y) = [x, y]
pointToList (Point3D x y z) = [x, y, z]

ghci> a = Point2D 3 4
ghci> b = Point3D 3 4 5
```

`:` in lists

```haskell
data List a = Nil | Cons a (List a)   -- `Cons` -- конструктор.

(:) === Cons                       -- `Cons` эквивалентен `:`
[] === Nil                         -- `[]` в haskell определяется как `Nil`.
1 : [] === Cons 1 Nil              -- вызови конструктор на 1 и пустом списке
[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

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

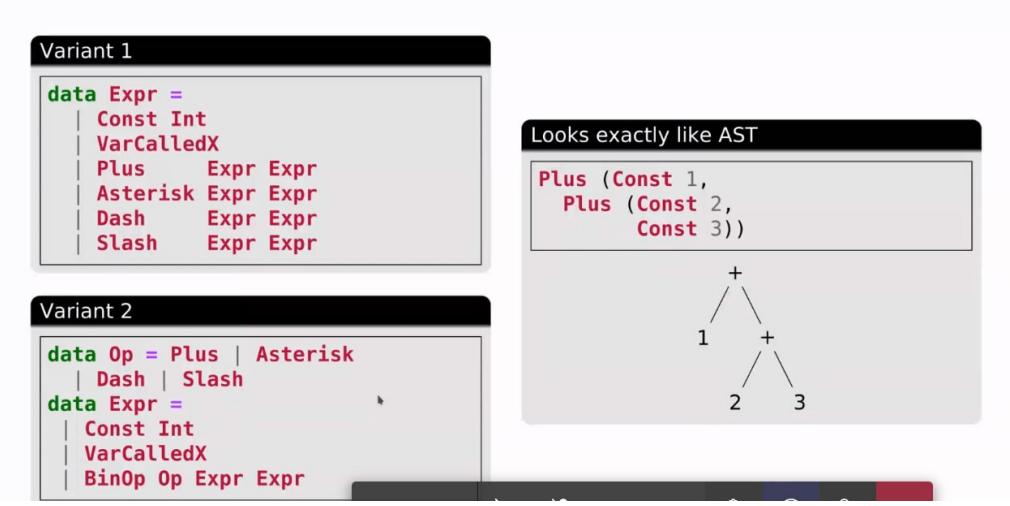
3. *Exhaustiveness*: x -- алгебраический тип $\Rightarrow$ для него есть конструктор

   x of some ADT $\;\Rightarrow\; \exists i, n \;:\; x = C_i(y_1, \ldots, y_n)$

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

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

Способы задизайнить выражение



(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 прицнипе не дать возможности создать "опасный" элемент.

`let` в коде

```haskell
isSquare :: Shape -> Bool
isSquare (Rectangle (PointD x0 y0) (PointD x1 y1)) =
    let x = abs(x0 - x1) in
    let y = abs(y0 - y1) in
    x == y
```

`case` и `where` в коде

```haskell
slideShape :: Shape -> PointT -> Shape
slideShape shape point =
    case shape of
        Circle center radius -> Circle (slidePoint center point) radius
        Rectangle lecft right -> Rectngle (slidePoint left point) (slidePoint right point)
    where
        slidePoint :: PointT -> PointT -> PointT
        slidePoint (PointD x0 y0) (PointD x1 y1) = PointD (x0 + x1) (y0 + y1)
```

Тут написано

```haskell
case shape of
    ...
where
    ...
```

Аналогично можно написать, используя `let`

```haskell
let slidePoint :: ... in
    slidePoint ... = ... in
case shape of
    ....
```

```haskell
a -> b -> c === a -> (b -> c)
-- функция принимает аргументы a, b и возвращает c

(a -> b) -> c
-- функция принимает в качестве аргумента функцию и возвращает c
```

Cons aka AtLeastOne, Nil aka Empty

```haskell
module List where

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

`Nil` <=> `Emplty` <=> пустой список

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

Maybe

```haskell
safeHead :: List a -> Maybe a
safeHead Empty = Nothing
safeHead (AtLeastOne x _) = Just x

safeTail :: List a -> Maybe (List a)
safeTail Empty = Nothing
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

```haskell
double :: List a -> List a
double (AtLeastOne x xs) = AtLeastOne (x * 2) xs
double Empty = Empty

triple :: List a -> List a
triple (AtLeastOne x xs) = AtLeastOne (x * 3) xs
triple Empty = Empty
```

Продвинутое решение

```haskell
map' :: (Int -> Int) -> List a -> List a
map' f (AtLeastOne x xs) = AtLeastOne (f x) xs
map' f Empty = Empty


-- Пример для map'
double' xs = map' doubleInt xs
    where
        doubleInt x = x * 2


```

```haskell
-- аналогичное определение double'
double' xs = map' (*2) xs
             -- 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

```haskell
sumListUp :: List Int -> Int
sumListUp Empty = 0
sumListUp (AtLeastOne x xs) = x + sumListUp xs

MultListUp :: List Int -> Int
MultListUp Empty = 1
MultListUp (AtLeastOne x xs) = x * MultListUp xs

fold :: (Int -> Int -> Int) -> Int -> List Int -> Int
fold f acc (AtLeastOne x xs) = fold f (x `f` acc) xs
fold f acc Empty = acc
```

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

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

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

`\x y -> x * 2 + y` -- лямбда-функция

Анонимная лямбда-функция

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

```haskell
map' :: (a -> b) -> List a -> List b
map' f Empty = Empty
map' f (AtLeastOne x xs) = AtLeastOne (f x) (map f xs)

double' xs = map' (\x -> 2 * x) xs
```

Сборка проекта

`stack build` -- собрать проект

`stack test` -- собрать тесты

22-09-19

```
f a b = a + b
a `f` b = a + b
-- 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

```
foo :: (Int -> Int) -> Int -> Int
foo bar x = x + bar x

ghci> foo (\x -> x * x) 3
ghci> 12
```

2. Currying

По сути, это подстановка функции вместо слова.

`sum x` ⇔ `helper 0 x`

```
sum = helper 0 where
    helper acc x | x > 0 = helper (acc + x * x) (x - 1)
                 | otherwise = acc

ghci> sum 4
ghci> 30     -- 1 + 4 + 9 + 16
```

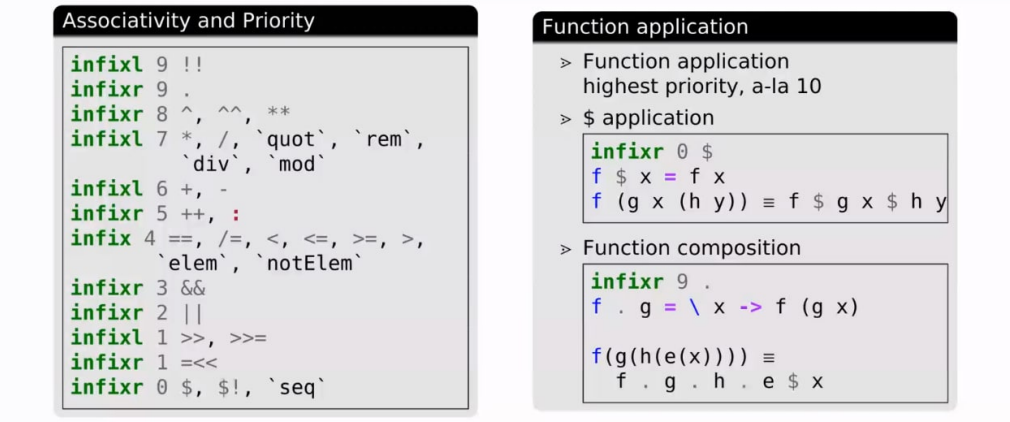
3. Композиция

```
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



Lazy evaluation

Lazy lists

```
let ones = 1 : ones    -- если захотим вывести, то зациклимся
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.

```
data List a = Nil | Cons a (List a)
let onesS = 1 :! onesS

ghci> take 5 onesS    -- Error: CInterrupted, потому что невозможно сразу
                      -- 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.

```
-- In normal form:
42
(2, "hello")
\x -> x + 1

-- Not in normal form:
1 + 2
"he" ++ "llo"
(\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

```
const 42 undefined
ghci> 42

f 42 $ undefined    -- f x == f $ x
ghci> 42

f 42 $! undefined    -- `!` значит вычислить выражение по значению
                     -- Нужно сначала посчитать правый аргумент, потом функцию `f`
```

| (guards)

```
factorial = helper 1 where
    helper acc k | k > 1  = helper (acc * k) (k - 1)
                 | otherwise = acc

ghci> factorial 5
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. Curring`).

:+

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

```
case (1, undefined) of (_,_) -> 42
ghci> 42

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

Lazy patterns

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

```
h (a, b) = g a b    -- strict: проверит, что действительно передали пару (a, b)
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.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc [] = acc
foldr f acc (x : xs) = x `f` foldr f acc xs

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc [] = acc
foldl f acc (x : xs) = foldl f (acc `f` x) xs
```

Функция буквально подставляется к каждому элементу

```
foldr f acc [1, 2, 3, 4] = f 1 (f 2 (f 3 (f 4 acc)))
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

```
filter :: (a -> Bool) -> [a] -> [a]

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

Симбиоз pattern matching и `|`

```haskell
eqList :: [Int] -> [Int] -> Bool
eqList [] [] = True
eqList (x : xs) (y : ys) | x == y = True
                         | otherwise = False
eqList _ _ = False

-- Если первые два случая не подошли, то всегда возвращается False, поэтому ветка `otherwise` является лишней.

-- Более короткий вариант:

eqList [] [] = True
eqList (x : xs) (y : ys) | x == y = True
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]`

```haskell
b = [2 * i | i <- [0..5]]
ghci> [0,2,4,6,8,10]

c = [(i, j) | i <- [0..1], j <- ['a'..'c']]
ghci> [(0,'a'),(0,'b'),(0,'c'),(1,'a'),(1,'b'),(1,'c')]
```

Можно еще дописать условие и выбрать нужное

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

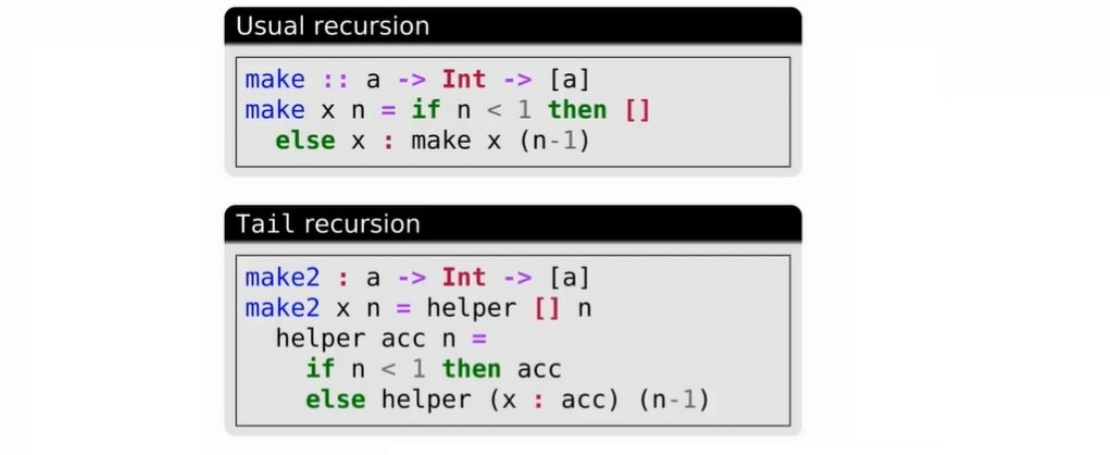
22-09-26

Tail recursion

Отличаеся 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 ⇒ `acc` обновляется во время вызова рекурсии.



Пример с практики

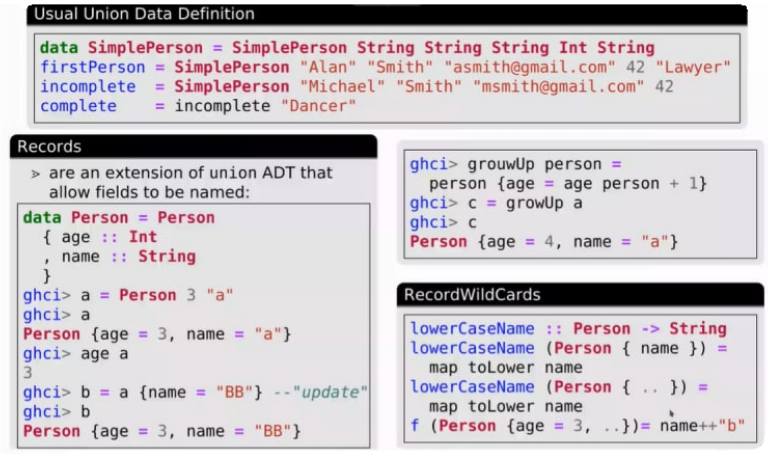
```haskell
-- плебейская рекурсия:
rev :: [a] -> [a]
rev [] = []
rev (h : t) = rev t ++ [h]

{-
rev [1, 2, 3]
    rev [2, 3] then ++ [2]
        rev [3] then ++ [2]
            rev [] then ++ [3]

Рекурсия запускается, штуки для `++` лежат на стеке. Долго
-}

-- хвостовая рекурсия:
rev' :: [a] -> [a]
rev' xs =
        go [] xs
    where
        go acc [] = acc
        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.



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

```
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

```
f :: a -> a -> a
f x y = x * y

g :: a -> (a -> a)
g x = \y -> x * y

-- `:type g` и `:type g` одинаковые
```

zip

получает два списка и возвращает список из пар

```
zip :: [a] -> [b] -> [(a, b)]

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

zip3

получает три списка и возвращает тупл

zipWith

получает функцию и два списка

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

ghci> zipWith (+) [1, 2, 3] [4, 5, 6]
ghci> [5,7,9]
```

Пример с композицией: squareEven

```haskell
squareEven :: [Int]
squareEven = map (^2) (map (*2) [0..])    -- обходим список два раза

squareEven = map (\x -> (x * 2)^2) [0..]  -- обходим список один раз

powEven :: Int -> [Int]
powEven n = map ((^n) . (*2)) [0..]    -- == f(g(x)) -- композиция функций, теперь умножение и степень независимые
--                f       g
```

```haskell
map2funcs f g xs = map (f . g) xs

ghci> map2funcs (^2) (*2) [0..]
```

class Eq \todo

```haskell
class MyEq a where
    equals :: a -> a -> Bool
    nequals :: a -> a -> Bool

    nequals x y = not & equals x y  -- выразили `equals` через `nequals`, но
                                    -- не дали реализацию ни для одной из функций


instance MyEq Bool where  -- реализуем инстанс `Bool`
    equals :: Bool -> Bool -> Bool
    equals True x = x    -- принимаем True и какой-то аргумент. Вернем True, только если x == True
    equals False x = not x -- принимаем False и аргумент. Вернем True, только если x == False



eqList :: MyEq a => List a -> List a -> Bool  -- сравниваем списки на равенство
eqList Nil Nil = True
eqList (x : xs) (y : ys) = equals x y && eqList xs ys
eqList _ _ = False
```

Практика 22-10-06

```haskell
import qualified Data.Set as Set
```

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

```yaml
...

library:
  source-dirs: ...
  dependencies:
  - containers

...
```

22-10-10

Linked list

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

```haskell
xs ++ (ys ++ zs)
```

`ys` копируется, потом `xs` копируется

```haskell
(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

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

Pure Functional Queues

```
empty :: queue -> bool
enqueue :: queue -> int -> queue   -- adds elem into a queue
head :: queue -> int
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: $|f| \geqslant |r|$ -- `f` станет пустым, только если `r` пустой.

Пример:

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

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

Практика 22-10-13

Паттерн `node@(Node x)`

```
foo :: Tree a -> Int -> Tree a
...
foo aboba@(Leaf x) = aboba
-- <=>  `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 ьвосклицателльные знаки лень, можно подключить расширение `StrictData`:

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

Тогда все поля становятся строгими.

`seq`

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

```
sumBest :: Int -> Int
SumBest n =
        go 0 n
    where
        go acc 0 = acc
        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:

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

```haskell
sumBest :: Int -> Int
SumBest n =
        go 0 n
    where
        go acc 0 = acc
        go acc n = go ($! (acc + n)) (n - 1)
```

Laziness

`~`

Использование для аргументов

```haskell
{ -# LANGUAGE StrictData #- }

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

Подключено расширение ⇒ все поля строгие. `~` делает вычисления поддеревьев ленивыми.

Использование для функций

```haskell
foo ~(Just x, Just y) = Just (x + y)
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`

```haskell
module Monoid where

class MyMonoid a where
    mempty :: a
    mappend :: a -> a -> a

instance MyMonoid [a] where
    mempty = []
    mappend = (++)

instance MyMonoid Int where
    mempty = 0      -- начальное значение для суммы
    mappend = (+)   -- будем складывать чиселки
```

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

```haskell
data Sum = Sum Int deriving Show
data Prod = Prod Int deriving Show

instance MyMonoid Sum where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)

instance MyMonoid Prod where
    mempty = Prod 1
    mappend (Prod x) (Prod y) = Prod (x * y)
```

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

```haskell
class MyMonoid a where
    mempty :: a
    (<>) :: a -> a -> a

instance MyMonoid [a] where
    mempty = []
    (<>) = (++)
```

22-10-17

Лямбда-исчисления

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

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: `(λx . * 2 x)` -- функция, принимающая x и возвращающая $2x$.

3. ***"Применение функции":***

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

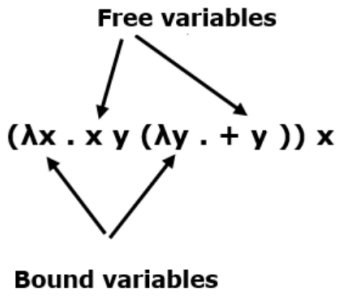
4. $[x/M]$ -- подставили значение M вместо переменной x.

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 фукнции. Вместо них нельзя что-то подставить.

Пример



Правила \todo

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

$FV(e_1e_2) = FV(e_1) \cup FV(e_2)$

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

$BV(x) = \varnothing$

$BV(e_1e_2) = BV(e_1) \cup BV(e_2)$

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

def. (замкнутое множество)

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

α-conversion

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

$\alpha - conversion$: если есть вложенные абстракции, то у вложенной можно поменять название аргумента

$\lambda x.\, \Lambda[x] \underset{\alpha}{\to} \lambda y.\, \Lambda[y]$

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.

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

Примеры

1. $(\lambda y.\, (\lambda x.\, x\ y))x \underset{\beta}{\to} (\lambda x.\, x\ y)[y/x]$ -- 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.

$x[x/M] = M$

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

$(e_1e_2)[x/M] = (e_1[x/M])(e_2[x/M])$

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

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

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

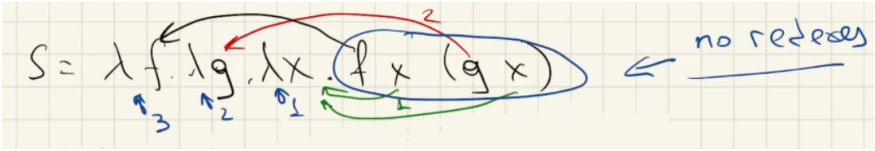
Последовательность де Брёйна (?) \todo

$\Lambda^d ::= N | \Lambda^d\Lambda^d | \lambda\ .\ \Lambda^d$

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`:

```haskell
Data Expr = Val Double | Div Expr Expr | Log Expr

bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind value f =
    case value of
        Just x -> f x
        Nothing -> Nothing

eval :: Expr -> Maybe Double
eval (Val n) = Just n
eval (Div x y) =
    eval x `bind` \x' ->
    eval y `bind` \y' ->
    totalDiv x' y'
eval (Log x) =
    eval x `bind` \x' -> totalLog x'


totalLog :: Double -> Maybe Double
totalLog n | n <= 0 = Nothing
           | otherwise = Just log(n)

totalDiv :: Double -> Double -> Maybe Double
totalDiv _ 0 = Nothing
totalDiv x y = Just x / y
```

Монады

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

```haskell
return :: a -> Monad a
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 коробочки. Если получилось, применяется функция `a -> Monad b`.

```haskell
instance Monad Maybe where
    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```

Lists

```haskell
instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]

    -- способ 1
    xs >>= f = concat (map f xs)

    -- способ 2
    xs >>= f = [y | x <- xs, y <- f x]

    -- способ 3
    xs >>= f =
        case xs of
            (h : t) -> f h ++ t >>= f
            [] -> []


```

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

Пример использования монадов

```haskell
Data Expr = Val Double | Div Expr Expr | Log Expr

eval :: Expr -> Maybe Double
eval (Val n) = Just n
eval (Div x y) =
    eval x >>= \x' ->
    eval y >>= \y' ->
    safeDiv x' y'
eval (Log x) =
    eval x >>= (\x' -> saveLog x')
```

```haskell

saveLog :: Double -> Maybe Double
saveLog n | n <= 0 = Nothing
          | otherwise = Just log(n)

safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv x y = Just x / y
```

do-нотации

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

```haskell
thing1 >>= (\x ->
    func1 x >>= (\y ->
        thing2 >>= (\_ ->
            func2 y >>= (\z ->
                return z))))
```

Может быть переписано с использованием `do`:

```haskell
do {
  x <- thing1;
  y <- func1 x;
  thing2;
  z <- func2 y;
  return z
}
```

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

IO()

Специальная монада, чтобы общаться с внешним миром.

```haskell
module Main (main) where

main :: IO()
main = do
    putStrLn "Enter your name:"
    name <- getLine   -- достаем строку из  `getLine` и
                      -- записываем её в переменную `name`
    putStrLn $ "Hello, " ++ name ++ "!"

```

read

Позволяет прочитать значение переменной как инстанс определенного типа

```haskell
let aboba = "123"
let aboba_int = read aboba :: Int  -- `aboba_int` теперь чиселко
```

Пример

```haskell
module Main (main) where

main :: IO()
main = do
    putStrLn "Enter your name:"
    name <- getLine
    putStrLn $ "Hello, " ++ name ++ "!"
    putStrLn "Enter your age:"
    arg <- getLine
    let age = read arg :: Int
    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`: $\lambda p\ .\ p\ False\ True$ (Вместо p подставляем абстракцию `True` или `False`, внутри которой меняем аргументы местами)

`And`: $\lambda p\ .\ \lambda q\ .\ p\ q\ False$

`Or`: $\lambda p\ .\ \lambda q\ .\ p\ True\ q$

`IfThenElse`: $\lambda p\ .\ \lambda t\ .\ \lambda e\ .\ p\ t\ e$ ($p == True\ \Rightarrow$ вернется t, 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` = $\lambda n\ .\ n\ (\lambda x\ .\ False)\ True$

`isZero(0)` $= (\lambda n\ .\ n\ (\lambda x\ .\ False)\ True)\ (\lambda s\ .\ \lambda z\ .\ z)\ =\ (\lambda s\ .\ \lambda z\ .\ z)(\lambda x\ .\ False)\ True\ =\ True$

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

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

`isZero` = $\lambda x\ .\ x\ True$

`isZero(0)` = $(\lambda y\ .\ y)\ True = True$

`isZero(2)` = $[\ False,\ [\ False,\ \lambda y\ .y\ ]\ ]\ True = False$

`prev` = $\lambda x\ .\ x\ False$

`prev(0)` = $False$

`prev(5)` = $[False, d_4]\ False = d_4$

Функции

Базисные

Функции вида $f\ :\ N^k \to N$

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

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

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

Композиция

$g \ : \ N^m \to N$

$f_1, \ldots, f_m \ : \ N^n \to N$

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

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

$H = \lambda x_1 \ldots \ \lambda x_n \ . \ G \ (F_1 \ x_1 \ \ldots \ x_n) \ \ldots \ (F_m \ x_1 \ \ldots \ x_n) \Longleftrightarrow h(x_1, \ldots, x_n)$

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.

$F \rightsquigarrow f$

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

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)

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 зациклиться на каком-то вызове.

ЧРФ = ПРФ + μ

Th. (о реализуемости)

$f \ : \ N^k \ \to \ N$ -- частичная функция, представима в виде λ-терма $f(n_1, \ldots, n_k) = m$

f вычислима $\Longleftrightarrow$ f = $((False \ d_{n_1}) \ d_{n_2} \ldots) \underset{\beta}{\to} d_m$

Th. (о неподвижной точке λ-терма)

$\forall F \ \exists \ V \ : \ V \underset{\beta}{=} F \ V$

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)

$\exists Y\ :\ \forall F\quad Y\ F \underset{\beta}{=} F\ (Y\ F)$

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

Th. (первая теорема рекурсии)

$\forall M\ \exists F\ :\ F \underset{\beta}{=} M[f/F]$

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

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

Практика 22-11-03

Хотим, чтобы у каждого листа был уникальный ключ

```haskell
relabel :: Tree a -> Int -> (Tree (Int, a), Int)
relabel (Leaf x) i = (Leaf (i, x), i + 1)
relabel (Node l r) i =
    let (l', i') = relable l i in
    let (r', i'') = relable r i' in
    (Node (l', r'), i'')

ghci> relabel tree 0
```

Спрячем элемент, отвечающий за счётник

```haskell
type Counter a = Int -> (a, Int)

relabel :: Tree a -> Counter (Tree (Int, x))
relabel (Leaf x) = \i -> (Leaf (i, x), i + 1)
relabel (Node l r) =
    relabel l `next` \l' ->
    relabel r `next` \r' ->
    ret $ Node l' r'

next :: Counter a -> (a -> Counter b) -> Counter b
next f g = \i ->
    let (a, i') = f i in
    g a i'

ret :: a -> Counter a   -- "return"-функция
ret x = \i -> (x, i)
```

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

```haskell
import Control.Monad.State

type State s a = s -> (a, s)

relabel :: Tree a -> Int -> Tree (Int, a)
relabel tree i =
    evalState (go tree) i
    where
        go :: Tree a -> State Int (Tree (Int, a))
        go (Leaf x) = do
            i <- get
            pur (i + 1)
            return (Leaf (i, x))
        go (Node l r) = do
            l' <- go l
            r' <- go r
            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 счётник состояния в 2 раза

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

22-11-07 \todo

Теорема Чёрча — Россера

$$U \underset{\beta}{\to} M,\ U \underset{\beta}{\to} N$$

$$\exists K\ :\ M \underset{\beta}{\to} K,\ N \underset{\beta}{\to} K$$

```
      U
     / \
    /   \
   M     U
    \   /
     \ /
      K
```

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

Правила вывода

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

Практика 22-11-10

class Functor

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

fmap

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

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

`<$>` -- аналог для `fmap`

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

pure

превращает объект в инстанс данной структуры.

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a

instance Applicative Maybe where
  pure :: a -> Maybe a
  pure x = Just x

instance Applicative [] where
  pure :: a -> [a]
  pure x = [x]
```

<*>

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

```haskell
class (Functor f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b


instance Applicative Maybe where
  <*> :: Maybe (a -> b) -> Maybe a -> Maybe b
  <*> Nothing _ = Nothing
  <*> (Just f) x = fmap f x    --    <=>    <*> _ Nothing = Nothing
                               --           <*> (Just f) (Just x) = Just (f x)
```

`(Just f)` -- контейнер с функцией `f`.

```haskell
instance Applicative [] where
  (<*>) :: [a -> b] -> [a] -> [b]
  <*> 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

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

Не сработает, ожидается список функций

Рабочие варианты:

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

Хотим создать что-то нестандартное...

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

```haskell
data Ziplist a = ZipList [a] deriving (Show)

instance Functor ZipList where
    fmap :: (a -> b) -> ZipList a -> ZipList b
    fmap f (ZipList xs) = ZipList $ fmap f xs

instance Applicative ZipList where
    pure :: a -> ZipList a
    pure x = ZipList $ repeat x  -- бесконечный список повторяющихся `x`

    (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
    ZipList fs <*> ZipList xs = ZipList $ zipWith $ fs xs
```

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

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

```haskell
module Parser where

-- 1 + 2 - 3  ==>  BinOp Plus (Number 1) (BinOp Minus (Number 2) (Number 3))
dara Expr = BinOp Op Expr Expr
          | Number Int

newtype Parser a = Parser  { runParser :: String -> Maybe (String, a) }
Parser :: String -> a

charA :: Parser Char  -- считывает одну заглавную букву A
charA = Parser $ \input ->

```

22-11-14 \todo

Типизация

по Чёрчу -- явно прописывается тип: X^{α}

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

Терм в контексте

$$\underbrace{x_1 : \alpha_1,\ \ldots, x_n : \alpha_n}_{r} \vdash u : \beta$$

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

Правила типизирования

картинка 1

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

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, по 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 рандомное значение, мы проверяем это свойство.

Используется библиотека `Hedgehog`.

Пример

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

```haskell
module Test.List where

import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import Test.Tasty
import Test.Tasty.Hedgehog

import Data.List (sort)
import List

-- Условие: в списке все элементы не больше, чем максимальное
prop_maximum :: Property
prop_maximum = property $ do
  list <- forAll $ genList 1 100
  let maxValue = maximumValue list
  assert (all (<= maxValue) list)

genInt :: Gen Int
genInt = Gen.Int (Range.constant (-10) 10)
-- генерируем чиселки от -10 до 10

genList :: Gen [Int]  -- возвращает сгенерированный список интов
genList = Gen.List (Range.constant 1 10) genInt
--          тип       кол-во эл-тов     как генерируем
--       структуры     (от 1 до 10)      элементы



-- сами тесты
props :: [TestTree]
props =
  [ 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 релузьтаты с более эффективной.

Пример:

`reverse` глупый за $O(n^2)$

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

`reverse` умный за $O(n)$

```
fastReverseList :: [a] -> [a]
 = go []
 where
   go acc [] = acc
   go acc (h : t) = go (h : acc) t
```

```
module Test.List where

import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import Test.Tasty
import Test.Tasty.Hedgehog

import List


genInt :: Gen Int
genInt = Gen.Int (Range.constant (-10) 10)

genList :: Gen [Int]
genList = Gen.List (Range.constant 1 10) genInt
```

```haskell

-- Проверяем, что дважды обернув список получаем исходный
prop_reverseList :: Property
prop_reverseList = property $ do
  list <- forAll $ genList 1 100
  reverseList (reverseList list) === list

-- Проверяем, что быстрая версия обращения списка работает так же, как эталонная
prop_fastReverseList :: Property
prop_fastReverseList = property $ do
  list <- forAll genList
  reverseList list === fastReverseList list

props :: [TestTree]
props =
  [ testProperty "Reversing list twice gets the input list" prop_reverseList
  , testProperty "Fast reverse gives the same " prop_fastReverseList
  ]
```

Пример: проверка min/max элемента в отсортированном списке

```haskell
module List where

-- простая реализация отсортированного списка за O(n)
class Ranged t where
    maxVal :: Ord a => t a -> a
    minVal :: Ord a => t a -> a

instace Ranged [] where
    maxVal xs = go (head xs) (tail xs)
    where
        go curMax [] = curMax
        go curMax (h : t) | h < curMax = go h t
                          | otherwise = go curMax t

    minVal xs = go (head xs) (tail xs)
    where
        go curMin [] = curMin
        go curMin (h : t) | h < curMin = go h t
                          | otherwise = go curMin t


-- умная реализация
newtype SortedList a = Sorted { getSorted :: [a] } derivind (Show, Eq)

instance Ranged SortedList where
    maxVal = last . getSorted
    minVal = head . getSorted
```

```haskell
module Test.List where

import Hedgehog
import qualified Hedgehog.Gen as Gen
import qualified Hedgehog.Range as Range
import Test.Tasty
import Test.Tasty.Hedgehog

import Data.List (sort)
import List
```

```haskell


genInt :: Gen Int
genInt = Gen.Int (Range.constant (-10) 10)

genList :: Gen [Int]
genList = Gen.List (Range.constant 1 10) genInt

genSortedList :: Gen (SortedList Int)
genSortedList = do
    xs <- genList
    return $ Sorted (sort xs)


prop_minSorted :: Property
prop_minSorted = property $ do
    list <- forAll $ genSortedList
    minVal list ===
```

Генерируем арифметическое выражение

```haskell
data Expr = BinOp Op Expr Expr
          | Number Int
          deriving (Show, Eq)

data Op = Plus
        | Minus
        | Mult
        | Div
        | Pow
        deriving (Show, Eq)


-- выбирает элемент из множества
genOp :: Gen Op
genOp = Gen.element [Plus, Minus, Mult, Div, Pow]

-- хотим создавать выражение рекурсивно (BinOp) или нерекурсивно (Number)
-- `Gen.choice` выбирает, что из этого генирировать
genExpr  = Gen.recursive Gen.choice [numGen] [binOpGen]
    where
        numGen = Number <$> Gen.Int (Range.linearFrom 0 (-10) 10)
        binOpGen = do
            op <- genOp
            Gen.subterm2 genExpr genExpr (BinOp op)
--                              (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`.