Функциональное программирование
Лекция 4. Введение в Haskell

Денис Николаевич Москвин

ИТМО, магистратура JB SE

28.09.2020

План лекции

План лекции

Язык Haskell: основные факты

- Haskell — *чистый* функциональный язык программирования с «*ленивой*» семантикой и полиморфной *статической* типизацией.
- Сайт языка: `https://www.haskell.org/`
- Назван в честь американского логика и математика Хаскелла Б. Карри.
- Первая реализация: 1990 год.
- Текущий стандарт языка: Haskell 2010: `https://www.haskell.org/onlinereport/haskell2010/`
- Стандарт определяет языковые конструкции в терминах трансляции в Haskell Kernel.
- Девиз (неофициальный): *Avoid Success at All Costs!*

Реализация Haskell

- Основная реализация: GHC.
  Последние версии 8.6.5/8.8.4/8.10.2.
- Включает интерпретатор GHCi.
- Упаковка библиотек в пакеты и дистрибуция: Cabal (или Stack).
- Хранилище пакетов: Hackage (или Stackage).
- Установка: Haskell Platform 8.6.5 (Windows) или ghcup (Linux и MacOS) .
- Инструменты поиска по документации: Hoogle или Hayoo.
- Стандартная библиотека
  - в узком смысле: то, что описано в Haskell Report;
  - в широком смысле: то, что поставляется с GHC.
- Для нашего курса настоятельно рекомендуется использовать версию GHC $\geqslant$ 7.10.

Крэш-старт

Создаём файл `Hello.hs` содержащий:

```
main = putStrLn "Hello, world!"
```

Затем запускаем интерпретатор GHCi, загружаем файл и вызываем определенную в нем функцию `main`

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/
Prelude> :load Hello
[1 of 1] Compiling Main  ( Hello.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello, world!
*Main>
```

`Prelude` — стандартный модуль языка Haskell, всегда подгружаемый по умолчанию.

План лекции

Выражения

- Выражения строятся из литералов (и переменных) с помощью функций, операторов и конструкторов данных.
- Интерпретатор позволяет вычислять их значения:

```
GHCi> 13 * 3 + 3
42
GHCi> "Hello " ++ 'w' : "orld!"
"Hello world!"
GHCi> not False
True
GHCi> exp 1
2.718281828459045
GHCi> reverse "Hello"
"olleH"
```

- При вызове функций круглые скобки вокруг аргументов не нужны.

Конструкторы данных: кортéжи

- *Кортéжи* (tuples) хранят значения произвольных типов, разделенные запятыми и заключенные в круглые скобки.

```
(42,"Hello world!")

(True,'c',3.141592653589793)
```

- Минимальный допустимый размер кортéжа равен 2.
- Стандарт говорит, что максимальный размер должен быть не меньше 15, в GHC — 62.
- Кортéжи разных размеров относятся к разным типам, никаких неявных преобразований между ними нет.

Конструкторы данных: списки

- *Списки* (lists) отличаются от кортéжей тем, что тип их элементов должен быть одним и тем же.

```
[1,2,3]

['H','e','l','l','o']
```

- Длина списка неограниченна, в том числе имеются одноэлементные списки и пустой список `[]`.
- Тип списка не зависит от его длины и полностью определяется типом элементов.
- Строковые литералы являются синтаксическим сахаром для списка символов:

```
GHCi> ['H','e','l','l','o']
"Hello"
```

Объявления: связывание переменных

- Знак равенства задает *связывание* (binding): имя слева связывается с выражением справа

```
x = 42                    -- глобальное
aBC = let z = x + y       -- глобальное (aBC), локальное (z)
  in z ^ 2                -- отступ (layout rule)
y = 7 + 3                 -- глобальное
```

Первый символ идентификатора должен быть в нижнем регистре.

- В GHCi связывание тоже допустимо

```
GHCi> answer = 39 + 3
GHCi> answer
42
```

- Иногда про связывание переменных говорят как про объявление констант или функций нулевой арности.

Объявления: связывание образцов (pattern binding)

Связываться равенством может несколько переменных сразу

```
GHCi> (x,y) = ('A',21*2)
GHCi> x
'A'
GHCi> y
42
GHCi> [u,v,w] = reverse "Hi!"
GHCi> w
'H'
GHCi> [p,q,r] = reverse "Hello!"
GHCi> q
*** Exception: Non-exhaustive patterns in [p, q, r]
```

Образцы — это конструкторы данных, в которых вместо значений подставлены переменные.

Объявления: функциональное связывание (functional binding)

Равенство может задавать функцию (function binding). Ниже `foo` связывается глобально, а `x` и `y` — локально.

```
foo x y = 10 * x + y        -- определение foo

fortyTwo = foo 2 22         -- применение foo
```

Допустимо использовать лямбда-выражения для определения функций.

```
foo  x y = 10 * x + y           -- комбинАторный стиль
foo' x   = \y -> 10 * x + y     -- смешанный стиль
foo''    = \x y -> 10 * x + y   -- лямбда-стиль
```

Все три приведенные определения эквивалентны.

Частичное применение и бесточечный стиль

- Три способа определить логарифм по основанию 2:

```
lg x = logBase 2 x          -- комбинАторное определение

lg'  = \x -> logBase 2 x    -- определение через лямбды

lg'' = logBase 2            --  бесточечный стиль
```

- Последний способ, в котором имя связывается с частично примененной функцией, называется *бесточечным* (pointfree).
- Смысл термина: в определении отсутствует точка применения функции — ее аргумент.
- Переход от второго равенства к третьему ни что иное как η-редукция.

Образцы как формальные аргументы функции

При определении функции мы можем в качестве формальных параметров использовать образцы.

```
fst (x,y) = x
snd (x,y) = y
```

При вызове происходит подстановка фактических значений вместо формальных параметров-переменных:

```
GHCi> fst ("Hello",2128506)
"Hello"
GHCi> snd ("Hello",2128506)
2128506
```

Допускается вложенность образцов произвольной глубины:

```
GHCi> fstOfSnd (x,(y,z)) = y
GHCi> fstOfSnd ('z',(33,True))
33
```

Иммутабельность

Связывание происходит единожды (в данной лексической области видимости).

```
z = 1                -- ok, связали
z = 2                -- ошибка
q q = \q -> q        -- ok, но...

p p p = p                -- ошибка
p = \p p -> p            -- ошибка
p = \p -> (\p -> p)      -- ok
```

В интерпретаторе повторное связывание допустимо.

```
GHCi> z = "Hello"
GHCi> z = 2
GHCi> z
2
```

Отступы

При переносе кода объявления на следующую строку отступ должен быть больше, чем отступ начала этого объявления.

```
roots a b c =
  (                                          -- начало пары
    (- b - sqrt (b^2-4*a*c)) / (2*a),-- первый элемент
    (- b + sqrt (b^2-4*a*c)) / (2*a) -- второй элемент
  )                                          -- конец пары
nRoots a b c = -- начало нового (глобального) объявления
  if b^2-4*a*c > 0
  then 2
  else if b^2-4*a*c == 0
  then 1
  else 0
```

При обнулении отступа начинается новое объявление (функциональное связывание).

Рекурсия

Рекурсивное определение содержит имя определяемой функции в ее теле. Корректная реализация рекурсии должна содержать достижимое терминирующее условие.

```
factorial0 n = if n > 1
               then n * factorial0 (n - 1)
               else 1

factorial1 n = if n == 0
               then 1
               else n * factorial1 (n - 1)
```

Чем отличается поведение этих реализаций?
Какая из них лучше?

Ошибки времени исполнения

Имеется специальное значение $\bot$ (основание, дно, bottom), маркирующее ошибку времени исполнения.
Библиотечная константа `undefined` — пример «реализации» $\bot$:

```
GHCi> undefined
*** Exception: Prelude.undefined
```

Другая реализация $\bot$:

```
bot = 1 + bot
fortyTwos = 42 : fortyTwos
```

Вторая функция — пример «продуктивной» расходимости:

```
GHCi> take 5 fortyTwos
[42,42,42,42,42]
```

«Аккуратная» версия факториала

```
factorial n =
  if n < 0
  then error "factorial: negative argument"
  else if n > 1
       then n * factorial (n-1)
       else 1
```

Функция error это гибкая версия undefined с настраеваемым сообщением об ошибке:

```
GHCi> factorial (-3)
*** Exception: factorial: negative argument
```

Еще более аккуратным был бы подход с перехватываемыми исключениями, позже мы его изучим.

Техника аккумулирующего параметра

Исходная реализация рекурсивной функции

```
factorial0 n = if n > 1
               then n * factorial0 (n-1)
               else 1
```

Альтернативная реализация, использующая аккумулятор

```
factorial' n = helper 1 n
helper acc n = if n > 1
               then helper (acc * n) (n - 1)
               else acc
```

Версия с аккумулятором часто позволяет сохранить линейную по числу рекурсивных вызовов сложность.

Конструкция where...

Конструкция `where` позволяют обеспечить локальное связывание вспомогательных конструкций.

```
roots' a b c = ((- b - sd) / denom, (- b + sd) / denom)
  where {sd=sqrt discr; discr=b^2-4*a*c; denom=2*a}

roots'' a b c = ((- b - sd) / denom, (- b + sd) / denom)
  where sd = sqrt discr
        discr = b ^ 2 - 4 * a * c
        denom = 2 * a
```

Допускается связывание не только переменных, но и функций

```
factorial'' n' = helper 1 n'
   where helper acc n = if n > 1
                        then helper (acc * n) (n - 1)
                        else acc
```

Выражение let...in...

Выражение `let...in...` отличается от `where...` порядком следования блоков, в которых новые имена связываются и используются.

```
roots''' a b c =
  let sd = sqrt discr
      discr = b ^ 2 - 4 * a * c
      denom = 2 * a
  in ((- b - sd) / denom, (- b + sd) / denom)

factorial''' m =
  let helper acc n =
        if n > 1
        then helper (acc * n) (n - 1)
        else acc
  in helper 1 m
```

Предохранители (Guards)

Предохранители просматриваются сверху вниз до первого истинного

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

factorial''''' n' =
  let helper acc n | n > 1      = helper (acc * n) (n - 1)
                   | otherwise = acc
  in helper 1 n'
```

Конструкция `where` может быть общей для предохранителей

```
nRoots' a b c | d > 0  = 2
              | d == 0 = 1
              | d < 0  = 0
   where d = b ^ 2 - 4 * a * c
```

Система модулей

- Программа состоит из набора модулей.
- Модули позволяют управлять пространствами имён.
- Инкапсуляция через списки экспорта и импорта.

Пример модуля

```
module A (foo, bar) where
import B (f, g, h)
foo = f g
bar = ...
bas = ...
```

- Конфликты имён разрешаются через полные имена

Квалифицированный импорт

```
import qualified B (f, g, h)
foo = B.f B.g
```

План лекции

Операторы

- *Оператор* — это комбинация из одного или более символов

```
! # $ % & * + . / < > ? @ ^ | - ~     = \     :
```

- Все операторы *бинарные* и *инфиксные*.
- Исключение: унарный префиксный минус, который всегда ссылается на `Prelude.negate`.
- Операторы, начинающиеся на двоеточие, должны быть конструкторами данных.
- Пример: оператор для суммы квадратов

```
a *+* b = a ^ 2 + b ^ 2
```

```
GHCi> 3 *+* 4
25
```

Инфиксная и префиксная нотация

- Операторы могут определяться и использоваться в префиксном (функциональном) стиле.
- Функции, в свою очередь, могут определяться и использоваться в инфиксном (операторном) стиле.

```
(*+*) a b = a ^ 3 + b ^ 3
x `plusminus` y = (x + y, x - y)
```

```
GHCi> (*+*) 2 3
35
GHCi> 2 *+* 3
35
GHCi> plusminus 4 3
(7,1)
GHCi> 4 `plusminus` 3
(7,1)
```

Проблема приоритета и ассоциативности

Чему равны значения выражений?

```
1 *+* 2 + 3

1 *+* 2 *+* 3
```

Проблема приоритета и ассоциативности

Чему равны значения выражений?

```
1 *+* 2 + 3

1 *+* 2 *+* 3
```

Инфиксные операторы требуют определения

- *приоритета*: какой оператор из цепочки выполнять первым;
- *ассоциативности*: какой оператор из цепочки выполнять первым при равном приоритете.

Приоритет и ассоциативность (fixity)

С помощью объявлений `infixl`, `infixr` или `infix` задаётся приоритет и ассоциативность операторов и функций.

```
infixl 6  **+*, *****
```

Теперь введённые нами операторы левоассоциативны и имеют тот же приоритет, что и обычный оператор сложения.
Задача: расставьте скобки и вычислите

```
1 **+* 2 + 3
3 + 1 **+* 2 * 3
```

Функциям тоже можно задавать приоритет

```
infix 5 `plusminus`
```

```
GHCi> 5 + 3 `plusminus` 6 * 2
(20,-4)
```

Приоритет стандартных операторов

```
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`
```

- В GHCi можно подглядеть, набрав `:info (&&)`.
- Применение имеет наивысший (10) приоритет.

Стандартный оператор ($)

- Оператор ($) задаёт применение, но с наименьшим возможным приоритетом

```
infixr 0  $
f $ x  =  f x
```

- Используется для элиминации избыточных скобок:

```
f (g x) ≡ f $ g x

f (g x (h y)) ≡ f $ g x (h y) ≡ f $ g x $ h y
```

- Из примера ясна причина правоассоциативности.
- ($) используют также для передачи аппликации в ФВП.

(Метаоператор ≡ мы будем использовать для обозначения операционной эквивалентности двух выражений.)

Стандартный оператор (.)

Оператор (.) задаёт композицию функций

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

Например, выражение (^2) . (+5) — это функция, прибавляющая 5 к своему аргументу, а затем возводящая результат в квадрат:

```
GHCi> (^2) . (+5) $ 1
36
GHCi> (^2) . (+5) $ 2
49
```

Сечения

- Операторы на самом деле просто функции и, поэтому, допускают частичное применение.
- *Сечения* (sections) — синтаксический сахар для частичного применения как к левому, так и к правому аргументу.
- Левое сечение:

```
(2 *+*) ≡ (*+*) 2 ≡ \y -> 2 *+* y
```

- Правое сечение:

```
(*+* 3) ≡ \x -> x *+* 3
```

- Наличие скобок при задании сечений обязательно, это часть их синтаксиса.

План лекции

Каждое выражение имеет тип

- Базовые типы:
  - `Bool` — булево значение;
  - `Char` — символ Юникода;
  - `Int`, `Integer` — целые числа;
  - `Float`, `Double` — числа с плавающей точкой;
  - `type1 -> type2` — тип функции;
  - `(type1, type2, ..., typeN)` — тип кортежа, `N>1`;
  - `()` — единичный тип, с одной константой `()`;
  - `[type1]` — тип списка с элементами типа `type1`.
- В GHCi для определения типа используют команду `:type`.
- Можно явно указывать тип выражения (`42 :: Integer`).
- Типы списка, кортежа и функции можно записывать в префиксной нотации

```
GHCi> [1,2,3] :: [Double]
[1.0,2.0,3.0]
GHCi> [1,2,3] :: [] Double
[1.0,2.0,3.0]
```

Устройство и использование типа

Булев тип представляет собой перечисление (enumeration)

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

Здесь `Bool` — *конструктор типа*,
а `True` и `False` — *конструкторы данных*.
Их имена должны начинаться с символа в верхнем регистре.
Можно задавать функции несколькими равенствами:

```
not          :: Bool -> Bool
not True     =  False
not False    =  True
```

Конструкторы данных в левой части называются *образцами*,
при вызове функции происходит *сопоставление с образцом*.
Объявление типа необязательно, но приветствуется.

Функции нескольких переменных

```
foo :: Int -> (Int -> Int)
foo x y = 10 * x + y
```

Функциональная стрелка *правоассоциативна*:

```
Int -> Int -> Int ≡ Int -> (Int -> Int)
```

Применение ассоциативно влево: `foo 4 2 == (foo 4) 2`.
Конструкция `foo 4` — это частично применённая функция.

```
GHCi> :t foo 4
foo 4 :: Int -> Int
GHCi> bar = foo 4
GHCi> :t bar
bar :: Int -> Int
GHCi> bar 2
42
```

Функции нескольких переменных и образцы

Переменные тоже являются образцами, это позволяет не писать здесь 4 уравнения:

```
(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) x    y    = False
```

Можно использовать подчеркивание (wildcard), если передаваемые аргументы не требуются для реализации тела:

```
(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _    _    = False
```

Параметрический полиморфизм

```
GHCi> k x1 x2 = x1
GHCi> :type k
k :: p1 -> p2 -> p1
```

В стрелочный тип входят не конкретные типы (должны начинаться с символа в верхнем регистре), а *переменные типа*. Можем применять к любым типам

```
GHCi> :type k 'x'
k 'x' :: p2 -> Char
GHCi> :type k "ABC"
k "ABC" :: p2 -> [Char]
GHCi> :type k 'x' False
k 'x' False :: Char
GHCi> k 'x' False
'x'
```

System F

```
k :: p1 -> p2 -> p1
k x1 x2 = x1
```

Все переменные *типа* находятся под неявным квантором всеобщности.
Его можно сделать явным; в GHC Core реализована System F.

```
GHCi> :set -XTypeApplications -fprint-explicit-foralls
GHCi> :t k
k :: forall {p1} {p2}. p1 -> p2 -> p1
GHCi> :t k @Char
k @Char :: forall {p2}. Char -> p2 -> Char
GHCi> :t k @Char 'x'
k @Char 'x' :: p2 -> Char
GHCi> :t k @Char @Bool
k @Char @Bool :: Char -> Bool -> Char
```

Специальный полиморфизм

Классы типов позволяют наложить специальные ограничения на полиморфный тип

```
GHCi> bas x y = 10 * x + y
GHCi> :t bas
bas :: Num a => a -> a -> a
```

Контекст `Num a` накладывает на тип a ограничения: для него должны быть определены операторы сложения, умножения и т.п.

`Int` и `Double` — *представители* (instanses) класса типов `Num`:

```
GHCi> bas (2 :: Int) (3 :: Int)
23
GHCi> bas (2 :: Double) (3 :: Double)
23.0
GHCi> bas 'y' 'z'
<interactive> error:
   * No instance for (Num Char) arising from a use of `bas'
```

Типы функций высших порядков

Функции высших порядков — функции, имеющие стрелочные аргументы.

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Еще примеры стандартных ФВП

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

infixl 1 &                    -- Data.Function
(&) :: a -> (a -> b) -> b
(&) = flip ($)
```

```
GHCi> 3 & (*10) & (+12)
42
```

Типы функций высших порядков

```
GHCi> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
GHCi> :info curry
curry :: ((a, b) -> c) -> a -> b -> c
                    -- Defined in `Data.Tuple'
GHCi> :info uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
                    -- Defined in `Data.Tuple'
GHCi> :t uncurry foo
uncurry foo :: (Int, Int) -> Int
GHCi> :t uncurry bas
uncurry bas :: Num c => (c, c) -> c
GHCi> (uncurry bas) (2,3)
23
```

В последнем примере работает механизм `default`.