Курс: Функциональное программирование

Лекция 4. Введение в Haskell

Денис Николаевич Москвин

14.10.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

Язык Haskell (1)

Haskell — *чистый* функциональный язык программирования с «*ленивой*» семантикой и полиморфной *статической* типизацией.

Сайт языка: http://haskell.org

Назван в честь американского логика и математика Хаскелла Карри.

Язык Haskell (2)

- Стандарт языка: Haskell 2010
- Основная реализация: GHC
- Хранилище пакетов: HackageDB
- Упаковка библиотек в пакеты и дистрибуция: Cabal
- Среда разработки: Haskell Platform
(GHC + Cabal + лучшие библиотеки + вспомогательные инструменты)

Крэш-старт

1. Инсталлируем Haskell Platform.

2. Создаём файл `hello.hs` содержащий:

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

3. Компилируем в исполняемый файл...

```
$ ghc --make hello
```

...или запускаем в интерпретаторе GHCi

```
$ ghci hello.hs
```

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

Связывание (1)

Знак равенства задаёт связывание (binding)

«Переменая» (слева) связывается со значением (справа)

```
x = 2                 -- глобальное
y = 42                -- глобальное
foo = let z = x + y -- глобальное (foo) локальное (z)
      in print z    -- отступ (layout rule)
```

- Первый символ должен быть в нижнем регистре
- В GHCi `let` используют для глобального связывания

```
Prelude> let fortyTwo = 42
Prelude> fortyTwo
42
```

Связывание (2)

Равенство может задавать функцию

(`add` связывается глобально, а `x` и `y` — локально)

```
add x y = x + y          -- определение add
add' x = \y -> x + y
add''  = \x y -> x + y

fortyTwo = add 40 2      -- вызов add

oops = print add 1 2
good = print (add 1 2)
```

Связывание (3)

Иммутабельность: связывание происходит единожды

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

Независимость от порядка:

```
fortyTwo = add 40 2  -- вызов add
add x y = x + y      -- определение add
```

Ленивость:

```
Prelude> let k = \x y -> x
Prelude> k 42 undefined
42
```

Рекурсия

Факториал в C.

Цикл и изменяемые переменные:

```
long factorial (int n)
{
  long res = 1;
  while (n > 1)
    res *= n--;
  return res;
}
```

Факториал в Haskell.

Рекурсия и повторное связывание переменной в новой области видимости

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

Хвостовая рекурсия

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

Это менее эффективно, чем цикл на C — на каждом шаге рекурсии монтируется новый кадр стека (stack frame).

Однако имеется оптимизация *хвостовой* рекурсии — преобразование её в цикл. Приведём рекурсю к хвостовой:

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

Стандартная техника для достижения хвостового вызова — вспомогательная функция с аккумулирующим параметром.

Конструкция `where` и выражение `let ... in`

Конструкция `where` позволяет обеспечить локальное связывание вспомогательных конструкций.

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

Выражение `let ... in` используют с той же целью

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

Предохранители (Guards)

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

```
f x y | y > z  = ...
      | y == z = ...
      | y < z  = ...
   where z = x * x
```

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

Каждое выражение имеет тип

Базовые типы:

- `Bool` — булево значение
- `Char` — символ Юникода
- `Int` — целое фиксированного размера
- `Integer` — целое произвольного размера
- `type1->type2` — тип функции
- `(type1,type2,...,typeN)` — тип кортежа
- `()` — 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)`

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
```

Объявление типа необязательно, но приветствуется.

Каррирование и частичное применение

```
mult :: Integer -> (Integer -> Integer)
mult x1 x2 = x1 * x2
```

Применение функции к аргументам происходит последовательно, по одному: `mult 2 3 == (mult 2) 3`.

В GHCi:

```
*Fp04> :type mult 2
mult 2 :: Integer -> Integer
*Fp04> let foo = mult 2
*Fp04> :type foo
foo :: Integer -> Integer
*Fp04> foo 3
6
```

Конструкция `mult 2` — это ***частично применённая*** функция.

Параметрический полиморфизм

Возможная реализация комбинатора **K** на Haskell

```
*Fp04> let k = \x y -> x
*Fp04> :type k
k :: t -> t1 -> t
```

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

```
*Fp04> :type k 'x' False
k 'x' False :: Char
```

Все переменные типа находятся под (неявно подразумеваемым) квантором всеобщности `k :: forall t t1. t -> t1 -> t`

Ограниченная квантификация

Классы типов позволяют наложить специальные ограничения на полиморфный тип

```
*Fp04> :type add
add :: Num a => a -> a -> a
```

Контекст `Num a` накладывает на тип `a` ограничения, заданные в классе типов `Num`: для него должны быть определены операторы сложения, умножения и т.п. Типы `Int` и `Double` — представители класса типов `Num`; можно применять к ним `add`

```
*Fp04> add (2::Int) (3::Int)
5
*Fp04> add (2.0::Double) (3.0::Double)
5.0
```

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

Система модулей

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

```
module A (foo, bar) where
import B
import C (f, g, h)

foo = f g
bar = ...
bas = ...
```

Конфликты имён разрешаются через полные имена

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

Команды GHCi `:module` и `:load`

- Команда `:load` отвечает за загрузку модуля.
- Команда `:module` управляет областью видимости.

```
Prelude> :load Fp04
[1 of 1] Compiling Fp04             ( Fp04.hs, interpreted )
Ok, modules loaded: Fp04.
*Fp04> isUpper 'A'
<interactive>:1:1: Not in scope: `isUpper'
*Fp04> :module +Data.Char
*Fp04 Data.Char> isUpper 'A'
True
*Fp04 Data.Char> :module -Data.Char
*Fp04>
```

Модуль `Prelude` всегда в области видимости (пока его явно не выгрузили).

Hoogle

Hoogle — это Google для Haskell.

Позволяет осуществлять поиск по API стандартных библиотек.

- Переходим на http://www.haskell.org/hoogle/
- Вводим, например, `digitToInt`
- Смотрим описание
- Можем посмотреть исходный код

План лекции

- Язык Haskell
- Основы программирования
- Базовые типы
- Система модулей
- Операторы и сечения

Операторы

Оператор — это комбинация из одного или более символов

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

Все операторы *инфиксные* и *бинарные* (исключение: унарный префиксный минус, который всегда ссылается на `Prelude.negate`)

Определим, например, оператор для суммы квадратов

```
a *+* b = a * a + b * b

res = 3 *+* 4
```

Инфиксная и префиксная нотация

Операторы могут определяться и использоваться в префиксном (функциональном) стиле. Для этого оператор заключают в круглые скобки:

```
(**+**) a b = a * a * a + b * b * b

res1 = (**+**) 2 3   -- ==35
res2 = 2 **+** 3     -- ==35
```

Функции, в свою очередь, могут определяться и использоваться в инфиксном (операторном) стиле

```
x `plus` y = x + y

res3 = 2 `plus` 3    -- ==5
res4 = plus 2 3      -- ==5
```

Проблема приоритета и ассоциативности

Чему равны значения выражений?

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

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

Приоритет и ассоциативность (fixity)

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

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

Теперь введённые нами операторы левоассоциативны и имеют тот же приоритет, что и обычный оператор сложения.

Расставьте скобки и вычислите

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

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 (&&)`.
- Аппликация имеет наивысший (10) приоритет.

Сечения

Операторы на самом деле просто функции и, значит, допускают частичное применение.

Левое сечение

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

Правое сечение

```
(**+ 3) == \x -> x **+ 3
```

Скобки обязательно должны присутствовать!

Стандартный оператор ($)

Оператор ($) задаёт аппликацию, но с наименьшим возможным приоритетом

```
infixr 0  $
($)    :: (a -> b) -> a -> b
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 последнего ясна причина правоассоциативности.

Бесточечный стиль

В Haskell можно сделать η-редукцию в определении функции. Если

```
foo = \x -> bar x
```

или, что то же самое

```
foo x = bar x
```

то можно написать определение `foo` в *бесточечном стиле*

```
foo = bar
```