

Введение в функциональное программирование на языке Haskell

Григорий Волков

2022

ИСП РАН

- Функция в математике, например: $f(x) = x^3$

- Функция в математике, например: $f(x) = x^3$
 - ставит в соответствие аргументу некоторый результат и всё
- А в программировании?

- Функция в математике, например: $f(x) = x^3$
 - ставит в соответствие аргументу некоторый результат и всё
- А в программировании?
 - может не завершиться
 - `int calc() { while (true) {}; return 0; }`

- Функция в математике, например: $f(x) = x^3$
 - ставит в соответствие аргументу некоторый результат и всё
- А в программировании?
 - может не завершиться
 - `int calc() { while (true) {}; return 0; }`
 - может иметь *побочные эффекты*
 - `int calc() { printf("hi!\n"); return 1; }`

- Функция в математике, например: $f(x) = x^3$
 - ставит в соответствие аргументу некоторый результат и всё
- А в программировании?
 - может не завершиться
 - `int calc() { while (true) {}; return 0; }`
 - может иметь *побочные эффекты*
 - `int calc() { printf("hi!\n"); return 1; }`
- Суть парадигмы ФП в двух словах — приблизиться к математическим функциям

- Функция в математике, например: $f(x) = x^3$
 - ставит в соответствие аргументу некоторый результат и всё
- А в программировании?
 - может не завершиться
 - `int calc() { while (true) {}; return 0; }`
 - может иметь *побочные эффекты*
 - `int calc() { printf("hi!\n"); return 1; }`
- Суть парадигмы ФП в двух словах — приблизиться к математическим функциям
- Язык Haskell: «чистые» функции + продвинутая система типов

Зачем приближаться к математическим функциям?

Зачем приближаться к математическим функциям?

- Ссылочная прозрачность (referential transparency)!
 - т.е. можно рассуждать о программе как о системе перезаписи

Зачем приближаться к математическим функциям?

- Ссылочная прозрачность (referential transparency)!
 - т.е. можно рассуждать о программе как о системе перезаписи
 - Огромный потенциал для анализа и оптимизации программ
 - Можно кэшировать общие подвыражения, параллелить вычисление подвыражений

Зачем приближаться к математическим функциям?

- Ссылочная прозрачность (referential transparency)!
 - т.е. можно рассуждать о программе как о системе перезаписи
 - Огромный потенциал для анализа и оптимизации программ
 - Можно кэшировать общие подвыражения, параллелить вычисление подвыражений
- Явные рассуждения об эффектах

Зачем приближаться к математическим функциям?

- Ссылочная прозрачность (referential transparency)!
 - т.е. можно рассуждать о программе как о системе перезаписи
 - Огромный потенциал для анализа и оптимизации программ
 - Можно кэшировать общие подвыражения, параллелить вычисление подвыражений
- Явные рассуждения об эффектах
 - Разграничение доступных возможностей в разных местах кода
 - Например «в этой функции можно только вычислять и писать в лог»

- Консоль GHCi онлайн: <https://www.tryhaskell.org/>
- Компилятор + консоль онлайн: <https://replit.com/languages/haskell>

- Консоль GHCi онлайн: <https://www.tryhaskell.org/>
- Компилятор + консоль онлайн: <https://replit.com/languages/haskell>
- Домашнее задание: установить инструментарий :)
 - Инструмент сборки stack: <https://haskellstack.org>
 - Сам управляет версиями компилятора
 - VS Code: расширение Haskell
 - Другие редакторы с поддержкой LSP: настраиваем haskell-language-server вручную

```
$ ghci
```

```
Prelude> 123 + 456
```

```
579
```

```
Prelude> 111 + 222 > (123 + 456) || 1 < 0
```

```
False
```

Кажется, все как обычно.

```
$ ghci
Prelude> 123 + 456
579
Prelude> 111 + 222 > (123 + 456) || 1 < 0
False
```

Кажется, все как обычно.

```
Prelude> True != False
```

```
<interactive>:11:6: error:
  • Variable not in scope: (!=) :: Bool -> Bool -> t
  • Perhaps you meant one of these:
    '>=' (imported from Prelude), '==' (imported from Prelude),
    '/=' (imported from Prelude)
```

Только неравенство пишется как /=, а не !=!

- `Variable not in scope: (!=) :: Bool -> Bool -> t`

Когда мы использовали несуществующий оператор `!=`, интерпретатор не нашёл переменную `(!=)` типа `Bool -> Bool -> t` в текущей области видимости.

- `Variable not in scope: (!=) :: Bool -> Bool -> t`

Когда мы использовали несуществующий оператор `!=`, интерпретатор не нашёл переменную `(!=)` типа `Bool -> Bool -> t` в текущей области видимости.

Операторы — не встроенные особые конструкции, а просто функции!
Попробуем вызвать привычным образом?

- `Variable not in scope: (!=) :: Bool -> Bool -> t`

Когда мы использовали несуществующий оператор `!=`, интерпретатор не нашёл переменную `(!=)` типа `Bool -> Bool -> t` в текущей области видимости.

Операторы — не встроенные особые конструкции, а просто функции!
Попробуем вызвать привычным образом?

```
Prelude> (&&)(True, False)
```

```
<interactive>:16:5: error:
```

- `Couldn't match expected type 'Bool'`
 with actual `type '(Bool, Bool)'`

Ой: мы создали *пару* (кортеж из 2 элементов) ?!

На самом деле вызов функции выглядит так:

```
Prelude> (&&) True False
```

```
False
```

```
Prelude> max 4 2
```

```
4
```

На самом деле вызов функции выглядит так:

```
Prelude> (&&) True False
```

```
False
```

```
Prelude> max 4 2
```

```
4
```

И наоборот, обычную функцию тоже можно использовать как оператор (в инфиксной записи):

```
Prelude> 2 `max` 4
```

```
4
```

Почему вызов выглядит как `max 4 2`? Почему типы выглядят как `Bool -> Bool -> Bool`? Что происходит??

Почему вызов выглядит как `max 4 2`? Почему типы выглядят как `Bool -> Bool -> Bool`? Что происходит??

Функция двух аргументов — на самом деле функция одного аргумента, возвращающая функцию одного аргумента, возвращающая результат!

```
max :: Integer -> (Integer -> Integer)
```

```
max 4 :: Integer -> Integer
```

```
max 4 2 :: Integer
```

Почему вызов выглядит как `max 4 2`? Почему типы выглядят как `Bool -> Bool -> Bool`? Что происходит??

Функция двух аргументов — на самом деле функция одного аргумента, возвращающая функцию одного аргумента, возвращающая результат!

```
max :: Integer -> (Integer -> Integer)
```

```
max 4 :: Integer -> Integer
```

```
max 4 2 :: Integer
```

```
Prelude> (max 4) 2
```

```
4
```

```
Prelude> :t +d (max 4)
```

```
(max 4) :: Integer -> Integer
```

Это сделано для удобства *частичного применения*.

Частичное применение удобно для передачи функций в качестве аргументов другим функциям.

Рассмотрим работу со списками.

Функция `map` применяет функцию ко всем элементам списка:

```
Prelude> map (+ 1) [12, 34, 56]  
[13, 35, 57]
```

Частичное применение удобно для передачи функций в качестве аргументов другим функциям.

Рассмотрим работу со списками.

Функция `map` применяет функцию ко всем элементам списка:

```
Prelude> map (+ 1) [12, 34, 56]  
[13,35,57]
```

Кстати, с помощью синтаксиса `..` можно генерировать диапазоны:

```
Prelude> map (max 4) [1..10]  
[4,4,4,4,5,6,7,8,9,10]
```

В стандартной библиотеке много функций для работы со списками, например:

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
Prelude> [True, False] ++ [False, True]
[True,False,False,True]
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> take 2 [11,22,33,44,55]
[11,22]
Prelude> concat [[1,2],[],[3],[4,5,6]]
[1,2,3,4,5,6]
```

Операции со списками

В стандартной библиотеке много функций для работы со списками, например:

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
Prelude> [True, False] ++ [False, True]
[True,False,False,True]
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> take 2 [11,22,33,44,55]
[11,22]
Prelude> concat [[1,2],[],[3],[4,5,6]]
[1,2,3,4,5,6]
```

Но вернёмся к map...

Определяем функции

Чтобы (например) передать `map` произвольное выражение, нужно определить свою функцию. Анонимные функции задаются лямбда-выражениями:

```
Prelude> map (\x -> x * 2 + x * 3) [1,2,3,4,5]
```

```
[5,10,15,20,25]
```

```
Prelude> (\x -> x * 2 + x * 3) 5
```

```
25
```

Определяем функции

Чтобы (например) передать `map` произвольное выражение, нужно определить свою функцию. Анонимные функции задаются лямбда-выражениями:

```
Prelude> map (\x -> x * 2 + x * 3) [1,2,3,4,5]  
[5,10,15,20,25]  
Prelude> (\x -> x * 2 + x * 3) 5  
25
```

Чтобы функция перестала быть анонимной, мы можем локально связать её с названием с помощью конструкции `let .. in`:

```
Prelude> let myfunc = \x -> x * 2 + x * 3 in map myfunc [1,2,3,4,5]  
[5,10,15,20,25]
```

Определяем функции

Чтобы (например) передать `map` произвольное выражение, нужно определить свою функцию. Анонимные функции задаются лямбда-выражениями:

```
Prelude> map (\x -> x * 2 + x * 3) [1,2,3,4,5]  
[5,10,15,20,25]  
Prelude> (\x -> x * 2 + x * 3) 5  
25
```

Чтобы функция перестала быть анонимной, мы можем локально связать её с названием с помощью конструкции `let .. in`:

```
Prelude> let myfunc = \x -> x * 2 + x * 3 in map myfunc [1,2,3,4,5]  
[5,10,15,20,25]
```

Синтаксический сахар позволяет опустить лямбда-синтаксис:

```
Prelude> let myfunc x = x * 2 + x * 3 in map myfunc [1,2,3,4,5]  
[5,10,15,20,25]
```

Кстати, можно и так:

```
Prelude> map (\x -> x 123) [ (3 *), (* 2), (+ 1)  
                             , (`mod` 6), \x -> (x + 1) * (x - 1) ]  
[369,246,124,3,15128]
```


Кстати, можно и так:

```
Prelude> map (\x -> x 123) [ (3 *), (* 2), (+ 1)  
                             , (`mod` 6), \x -> (x + 1) * (x - 1) ]  
[369,246,124,3,15128]
```

«Functions are first-class citizens»: их можно не только передавать как аргументы другим функциям, но и хранить в любых структурах данных.

Задаём функции

В консоли GHCi можно также глобально определить функцию для всей последующей сессии:

```
Prelude> myfunc x = x * 2 + x * 3
```

```
Prelude> myfunc 32
```

```
160
```

```
Prelude> map myfunc [1..10]
```

```
[5,10,15,20,25,30,35,40,45,50]
```

Задаём функции

В консоли GHCi можно также глобально определить функцию для всей последующей сессии:

```
Prelude> myfunc x = x * 2 + x * 3
```

```
Prelude> myfunc 32
```

```
160
```

```
Prelude> map myfunc [1..10]
```

```
[5,10,15,20,25,30,35,40,45,50]
```

В .hs файле определения на верхнем уровне как правило даются с явной аннотацией типов:

```
myfunc :: Integer -> Integer
```

```
myfunc x = x * 2 + x * 3
```

Условные выражения

Конструкция `if` является выражением, подобно тернарному оператору `?:` в императивных языках:

```
Prelude> if 2 > 1 then 222 else 111  
222
```

Условные выражения

Конструкция `if` является выражением, подобно тернарному оператору `?:` в императивных языках:

```
Prelude> if 2 > 1 then 222 else 111  
222
```

Теперь мы можем, например, определять рекурсивные функции с условием завершения:

```
fib :: Integer -> Integer  
fib x = if x == 0 || x == 1 then x else fib (x - 2) + fib (x - 1)
```

Условные выражения

Конструкция `if` является выражением, подобно тернарному оператору `?:` в императивных языках:

```
Prelude> if 2 > 1 then 222 else 111
222
```

Теперь мы можем, например, определять рекурсивные функции с условием завершения:

```
fib :: Integer -> Integer
fib x = if x == 0 || x == 1 then x else fib (x - 2) + fib (x - 1)

Prelude> fib 4
3
Prelude> map fib [0..10]
[0,1,1,2,3,5,8,13,21,34,55]
```

Условные выражения

Конструкция `if` является выражением, подобно тернарному оператору `?:` в императивных языках:

```
Prelude> if 2 > 1 then 222 else 111  
222
```

Теперь мы можем, например, определять рекурсивные функции с условием завершения:

```
fib :: Integer -> Integer  
fib x = if x == 0 || x == 1 then x else fib (x - 2) + fib (x - 1)  
  
Prelude> fib 4  
3  
  
Prelude> map fib [0..10]  
[0,1,1,2,3,5,8,13,21,34,55]
```

Рекурсия — базовый приём в функциональном программировании, однако на практике мы чаще используем её скрыто, внутри библиотечных функций вроде `map`.

Ту же функцию можно записать компактнее, практически в виде математических уравнений:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib x = fib (x - 2) + fib (x - 1)
```


Ту же функцию можно записать компактнее, практически в виде математических уравнений:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib x = fib (x - 2) + fib (x - 1)
```

На самом деле это эквивалентно использованию конструкции case:

```
fib a = case a of
    0 -> 0
    1 -> 1
    x -> fib (x - 2) + fib (x - 1)
```

Сопоставление с образцом

case — pattern matching («сопоставление с образцом»).

В отличие от switch в Си-подобных языках, это *выражение*, которое «разбирает» входное значение. Ветви могут как сравнивать с константами, так и связывать имена. Даже во вложенных значениях:

```
Prelude> case (1, [2, 3, 4]) of (_, [x, y, 4]) -> x * y  
6
```

Сопоставление с образцом

case — pattern matching («сопоставление с образцом»).

В отличие от switch в Си-подобных языках, это *выражение*, которое «разбирает» входное значение. Ветви могут как сравнивать с константами, так и связывать имена. Даже во вложенных значениях:

```
Prelude> case (1, [2, 3, 4]) of (_, [x, y, 4]) -> x * y
6
```

```
calc23 :: [Integer] -> Integer
calc23 [x, y] = x * 2 + y * 2
calc23 [x, 1, y] = x * 3 + y * 3
calc23 _ = 0
```

```
*Test> calc23 [2,2]
8
```

```
*Test> calc23 [2,1,2]
12
```

```
*Test> calc23 [2,2,2]
0
```

```
twice f val = (f val, f val)
```

Какой тип у этой функции?

```
twice f val = (f val, f val)
```

Какой тип у этой функции?

```
Prelude> :t twice
```

```
twice :: (t -> b) -> t -> (b, b)
```

Компилятор автоматически вывел наиболее общий тип. Этой функции всё равно, значениями каких типов оперирует *f*, поэтому мы видим *параметры*, или *переменные* (с маленькой буквы) вместо конкретных типов.

```
twice f val = (f val, f val)
```

Какой тип у этой функции?

```
Prelude> :t twice
```

```
twice :: (t -> b) -> t -> (b, b)
```

Компилятор автоматически вывел наиболее общий тип. Этой функции всё равно, значениями каких типов оперирует *f*, поэтому мы видим *параметры*, или *переменные* (с маленькой буквы) вместо конкретных типов.

Не говорите слова «generic» и «template», они оба имеют другие значения в мире Haskell :)

Такие функции называются полиморфными.

Параметрический полиморфизм и его интересные последствия

Самый глубокий слайд на сегодня.

Параметрический полиморфизм и его интересные последствия

Самый глубокий слайд на сегодня.

Сколько реализаций существует у функции такого типа?

```
id :: a -> a
```


Параметрический полиморфизм и его интересные последствия

Самый глубокий слайд на сегодня.

Сколько реализаций существует у функции такого типа?

```
id :: a -> a
```

Одна, очевидная.

```
id x = x
```

Нам просто *неоткуда* больше взять результат типа `a`, кроме как из аргумента!!

Параметрический полиморфизм и его интересные последствия

Самый глубокий слайд на сегодня.

Сколько реализаций существует у функции такого типа?

```
id :: a -> a
```

Одна, очевидная.


```
id x = x
```

Нам просто *неоткуда* больше взять результат типа `a`, кроме как из аргумента!!

Хотя, я вас немного обманул. На самом деле можно взять значение `undefined / error "msg"`, которое имеет любой тип (`undefined :: a`) и выкидывает исключение при означивании.

```
id _ = undefined
```

Но всё же, по сигнатуре функции зачастую можно однозначно понять, что она делает. Поэтому популярен поиск по типам (Hoogle / Hayoo), разрабатываются инструменты синтеза кода (Synquid).



set:stackage
Search

Packages

- is:exact
- base
- ghc
- haskell-gi-base
- relude
- xmonad-contrib
- Cabal
- base-prelude
- rio
- numeric-prelude
- pqueue
- dimensional
- ghc-lib-parser
- rebase
- numhask
- stack
- LambdaHack
- mixed-types-num
- brittany
- distribution-opensuse
- fakory
- hledger-web

Results

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

`map :: (a -> b) -> [a] -> [b]`
base Prelude Data.List GHC.Base GHC.List GHC.OldList, ghc GHC.Prelude, haskell-gi-base Data.GI.Base.ShortPrelude, relude Relude.List.Reexport, xmonad-contrib XMonad.Config.Prime
map f xs is the list obtained by applying f to each element of xs, i.e.,

`map :: (a -> b) -> [a] -> [b]`
Cabal Distribution.Compat.Prelude.Internal, base-prelude BasePrelude, rio RIO.List RIO.Prelude, numeric-prelude NumericPrelude NumericPrelude.Base, pqueue Data.PQueue.Max, dimensional Numeric.Units.Dimensional.Prelude, ghc-lib-parser GHC.Prelude, rebase Rebase.Prelude, numhask NumHask.Prelude, stack Stack.Prelude, LambdaHack Game.LambdaHack.Core.Prelude
Game.LambdaHack.Core.Prelude, mixed-types-num Numeric.MixedTypes.PreludeHiding, brittany Language.Haskell.Brittany.Internal.Prelude, distribution-opensuse OpenSuse.Prelude, fakory Fakory.Prelude, hledger-web Hledger.Web.Import, tonalude Tonalude
map f xs is the list obtained by applying f to each element of xs, i.e.,

`strictMap :: (a -> b) -> [a] -> [b]`
ghc GHC.Utils.Misc, ghc-lib-parser GHC.Utils.Misc

`map :: (a -> b) -> [a] -> [b]`
prelude-compat Data.List2010 Prelude2010

`map :: (a -> b) -> [a] -> [b]`
llvm-hs-pure LLVM.Prelude
O(n). map f xs is the list obtained by applying f to each element of xs, i.e.,

Рис. 1: Hoogle

Какого типа функция `max` на самом деле?

```
Prelude> :t max
```

```
max :: Ord a => a -> a -> a
```

Она тоже полиморфная, но у неё есть *контекст*, содержащий *ограничение*: тип `a` должен быть «упорядочиваемым» (`Ord`).

Какого типа функция `max` на самом деле?

```
Prelude> :t max
```

```
max :: Ord a => a -> a -> a
```

Она тоже полиморфная, но у неё есть *контекст*, содержащий *ограничение*: тип `a` должен быть «упорядочиваемым» (`Ord`).

Числовые константы, кстати, тоже полиморфны:

```
Prelude> :t 123
```

```
123 :: Num p => p
```

Забегая вперёд скажем, что эти слова `Ord`, `Num` — названия *классов типов*.

В стандартной библиотеке определён оператор:

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

```
f $ x = f x
```

Зачем?

В стандартной библиотеке определён оператор:

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

Зачем?

```
Prelude> take 2 (filter (> 10) (map (* 2) [1..10]))  
[12,14]  
Prelude> take 2 $ filter (> 10) $ map (* 2) [1..10]  
[12,14]
```

Он определён с наименьшим приоритетом и соответственно предназначен для избавления от скобок. Таким образом можно писать последовательности трансформаций данных подобно UNIX pipes, только справа налево.

Спасибо за внимание!