

Лекция 8: ленивость

Функциональное программирование на Haskell

Алексей Романов

4 апреля 2018

МИЭТ

Передача по значению

- Как вычисляется выражение вида $f(g(x, y), h(z))$ в привычных языках?

Передача по значению

- Как вычисляется выражение вида $f(g(x, y), h(z))$ в привычных языках?
- Сначала вычисляются аргументы $g(x, y)$ и $h(z)$ (гарантированно в этом порядке или нет), потом их значения передаются в f .
- Это называется передачей аргументов по значению.
- Для вычисления выражения нужно вычислить все подвыражения.
- Исключения в C-подобных языках?

Передача по значению

- Как вычисляется выражение вида $f(g(x, y), h(z))$ в привычных языках?
- Сначала вычисляются аргументы $g(x, y)$ и $h(z)$ (гарантированно в этом порядке или нет), потом их значения передаются в f .
- Это называется передачей аргументов по значению.
- Для вычисления выражения нужно вычислить все подвыражения.
- Исключения в C-подобных языках?
- `&`, `|`, `?` `:.`

Передача по значению

- Как вычисляется выражение вида $f(g(x, y), h(z))$ в привычных языках?
- Сначала вычисляются аргументы $g(x, y)$ и $h(z)$ (гарантированно в этом порядке или нет), потом их значения передаются в f .
- Это называется передачей аргументов по значению.
- Для вычисления выражения нужно вычислить все подвыражения.
- Исключения в C-подобных языках?
- `&&`, `||`, `?` `:`.
- В них вычисляется первый операнд, а второй (и третий для `?` `:`) — только если необходимо.

Передача по имени

- Как можно сделать по-другому?

Передача по имени

- Как можно сделать по-другому?
- Макросы в C — каждое использование аргумента заменяется на переданное выражение (а не на его значение).
- И это рекурсивно: макросы в этом определении тоже будут заменены на своё определение.
- Это передача по имени (почти).

Передача по имени

- Как можно сделать по-другому?
- Макросы в C — каждое использование аргумента заменяется на переданное выражение (а не на его значение).
- И это рекурсивно: макросы в этом определении тоже будут заменены на своё определение.
- Это передача по имени (почти).
- У настоящей передачи по имени нет таких проблем со скобками, как в макросах C.

Передача по имени

- Как можно сделать по-другому?
- Макросы в C — каждое использование аргумента заменяется на переданное выражение (а не на его значение).
- И это рекурсивно: макросы в этом определении тоже будут заменены на своё определение.
- Это передача по имени (почти).
- У настоящей передачи по имени нет таких проблем со скобками, как в макросах C.
- Аналогичное поведение переменных в командной строке и скриптах Windows и Linux.

Сравнение на примерах 1

- Если рассмотреть

$$x = 2 + 2$$

$$y = x + x$$

то при передаче по значению будет вычислено

Сравнение на примерах 1

- Если рассмотреть

$$x = 2 + 2$$

$$y = x + x$$

то при передаче по значению будет вычислено

$$x = 4$$

$$y = 4 + 4$$

а при передаче по имени —

Сравнение на примерах 1

- Если рассмотреть

$$x = 2 + 2$$

$$y = x + x$$

то при передаче по значению будет вычислено

$$x = 4$$

$$y = 4 + 4$$

а при передаче по имени —

$$x = 2 + 2$$

$$y = (2 + 2) + (2 + 2)$$

Сравнение на примерах 2

- `if1(x, y, z) = if x then y else z`

`if1(true, 1, 1 / 0)`

Что случится при передаче по значению? А по имени?

Сравнение на примерах 2

- `if1(x, y, z) = if x then y else z`

`if1(true, 1, 1 / 0)`

Что случится при передаче по значению? А по имени?

- По значению: ошибка.

Сравнение на примерах 2

- `if1(x, y, z) = if x then y else z`

`if1(true, 1, 1 / 0)`

Что случится при передаче по значению? А по имени?

- По значению: ошибка.
- По имени: 1.

Какой порядок лучше?

- Плюсы передачи по имени:

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений (представьте, что в предыдущем примере вместо 1 / 0 вызов сложной функции).

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений (представьте, что в предыдущем примере вместо 1 / 0 вызов сложной функции).
- Главный минус:

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений (представьте, что в предыдущем примере вместо $1 / 0$ вызов сложной функции).
- Главный минус:
 - Можно сделать очень много лишних вычислений.

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений (представьте, что в предыдущем примере вместо $1 / 0$ вызов сложной функции).
- Главный минус:
 - Можно сделать очень много лишних вычислений.
 - Как только мы используем какую-то переменную более одного раза

Какой порядок лучше?

- Плюсы передачи по имени:
 - Можно избежать ошибок.
 - Можно избежать лишних вычислений (представьте, что в предыдущем примере вместо $1 / 0$ вызов сложной функции).
- Главный минус:
 - Можно сделать очень много лишних вычислений.
 - Как только мы используем какую-то переменную более одного раза (если переменные в её определении не изменились).

Передача по необходимости

- В Haskell используется передача по необходимости:
- Как передача по имени, но значение каждой переменной вычисляется только один раз.

Передача по необходимости

- В Haskell используется передача по необходимости:
- Как передача по имени, но значение каждой переменной вычисляется только один раз.
- Это имеет смысл потому, что все переменные неизменяемы.

Передача по необходимости

- В Haskell используется передача по необходимости:
- Как передача по имени, но значение каждой переменной вычисляется только один раз.
- Это имеет смысл потому, что все переменные неизменяемы.
- При этом значения тоже не обязательно вычислять «до конца»: мы хотим, чтобы `length [1/0, 1/0, 1/0]` возвращало 3.

Передача по необходимости

- В Haskell используется передача по необходимости:
- Как передача по имени, но значение каждой переменной вычисляется только один раз.
- Это имеет смысл потому, что все переменные неизменяемы.
- При этом значения тоже не обязательно вычислять «до конца»: мы хотим, чтобы `length [1/0, 1/0, 1/0]` возвращало 3.
- Ленивость в том или ином виде есть во многих языках (пример: `System.Lazy<T>` в .NET).
- Но в Haskell она встроена очень глубоко.

Нормальная форма

- Выражение находится в нормальной форме, если там нет ничего, что можно вычислить (редуцировать).
- Примеры:

Нормальная форма

- Выражение находится в нормальной форме, если там нет ничего, что можно вычислить (редуцировать).
- Примеры:

42

(2, "hello")

$\lambda x. x + 1$

- Примеры выражений не в нормальной форме:

Нормальная форма

- Выражение находится в нормальной форме, если там нет ничего, что можно вычислить (редуцировать).
- Примеры:

42

(2, "hello")

\x -> (x + 1)

- Примеры выражений не в нормальной форме:

1 + 2

(\x -> x + 1) 2

"he" ++ "llo"

(1 + 1, 2 + 2)

let x = 1 in x

Теоремы о нормальных формах в лямбда-исчислении

- В лямбда-исчислении можно доказать:
- Два порядка вычисления одного выражения не могут привести к разным значениям (нормальным формам).
- Но может случиться, что один из них даёт значение, а другой — нет.
- Если какой-то порядок приводит к значению, то передача по имени и по необходимости тоже к нему приведут.

Слабая заголовочная нормальная форма

- Выражение находится в слабой заголовочной нормальной форме (WHNF), если это:
 - Лямбда.
 - Литерал.
 - Конструктор данных, возможно с аргументами.
 - Функция от n аргументов с $m < n$ аргументами.

Аргументы здесь — любые выражения

Слабая заголовочная нормальная форма

- Выражение находится в слабой заголовочной нормальной форме (WHNF), если это:
 - Лямбда.
 - Литерал.
 - Конструктор данных, возможно с аргументами.
 - Функция от n аргументов с $m < n$ аргументами.

Аргументы здесь — любые выражения (а для НФ они тоже должны быть в НФ).

- Примеры:

Слабая заголовочная нормальная форма

- Выражение находится в слабой заголовочной нормальной форме (WHNF), если это:
 - Лямбда.
 - Литерал.
 - Конструктор данных, возможно с аргументами.
 - Функция от n аргументов с $m < n$ аргументами.

Аргументы здесь — любые выражения (а для НФ они тоже должны быть в НФ).

- Примеры:

```
(1 + 1, 2 + 2)      -- (,) (1 + 1) (2 + 2)
\x -> 2 + 2 + x
(1/0) : take 4 [1..]
(+) 1
```

Отложенные вычисления

- В Haskell во время исполнения переменная может указывать не только на значение, а на невычисленное выражение (thunk).

- Если мы напишем

```
x = (length [1,3], 1/0)
```

то `x` вначале указывает именно на thunk.

- После этого

```
case x of  
  (y, z) -> ...
```

требуется вычислить внешний конструктор `x` (привести к WHNF), и в памяти будет

Отложенные вычисления

- В Haskell во время исполнения переменная может указывать не только на значение, а на невычисленное выражение (thunk).

- Если мы напишем

```
x = (length [1,3], 1/0)
```

то `x` вначале указывает именно на thunk.

- После этого

```
case x of  
  (y, z) -> ...
```

требуется вычислить внешний конструктор `x` (привести к WHNF), и в памяти будет

```
x: (*thunk1*, *thunk2*)  *thunk1*: length [1,3]  
y: *thunk1*              *thunk2*: 1/0  
z: *thunk2*
```

- Если это

```
case x of
```

```
  (y, z) -> print y
```

то y тоже потребуется вычислить, и получим

- Если это

```
case x of  
  (y, z) -> print y
```

то `y` тоже потребуется вычислить, и получим

```
x: (2, *thunk2*)           *thunk1*: 2  
y: 2                       *thunk2*: 1/0  
z: *thunk2*
```

:print

- Мы можем увидеть частично вычисленные выражения в GHCi с помощью команд `:print` и `:sprint`:

```
ghci> x = (length [1,3], 1/0 :: Double)
```

```
ghci> :sprint x
```

```
x = _
```

```
ghci> case x of (y, z) -> 0
```

```
0
```

```
ghci> :sprint x
```

```
x = (_,_)
```

```
ghci> case x of (y, z) -> y
```

```
2
```

```
ghci> :sprint x
```

```
x = (2,_)
```

:print

- Мы можем увидеть частично вычисленные выражения в GHCi с помощью команд `:print` и `:sprint`:

```
ghci> x = (length [1,3], 1/0 :: Double)
```

```
ghci> :sprint x
```

```
x = _
```

```
ghci> case x of (y, z) -> 0
```

```
0
```

```
ghci> :sprint x
```

```
x = (_,_)
```

```
ghci> case x of (y, z) -> y
```

```
2
```

```
ghci> :sprint x
```

```
x = (2,_)
```

- Также посмотрите `ghc-vis` (для старых GHC).

Примеры

- Ещё примеры:

Примеры

- Ещё примеры:
- `square x = x*x`
`square (square (1+1))`
- `length (take 2 (filter even [1..]))`

Примеры

- Ещё примеры:
- `square x = x*x`
`square (square (1+1))`
- `length (take 2 (filter even [1..]))`
- `(&&) :: Bool -> Bool -> Bool`
`True && x = x`
`False && x = False`

ведёт себя как в C автоматически.

Примеры

- Ещё примеры:
- `square x = x*x`
`square (square (1+1))`
- `length (take 2 (filter even [1..]))`
- `(&&) :: Bool -> Bool -> Bool`
`True && x = x`
`False && x = False`
ведёт себя как в C автоматически.
- Как определить вариант, который начинает вычисление с правого операнда?

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0
```

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
sum (x:xs) acc = sum xs (x + acc)

sum [1..100] 0 ~\~> sum 1:[2..100] 0
```

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>
sum [2..100] (1 + 0)
```

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0  $\rightsquigarrow$  sum 1:[2..100] 0  $\rightsquigarrow$ 
```

```
sum [2..100] (1 + 0)  $\rightsquigarrow$  sum 2:[3..100] (1 + 0)
```

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>
sum [2..100] (1 + 0) ~> sum 2:[3..100] (1 + 0) ~>
sum [3..100] (2 + (1 + 0))
```


Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>
sum [2..100] (1 + 0) ~> sum 2:[3..100] (1 + 0) ~>
sum [3..100] (2 + (1 + 0)) ~> ... ~>
(100 + ... + (1 + 0))
```

Минусы ленивости

- Вроде бы можно гарантировать, что ленивые вычисления всегда делают не больше шагов, чем энергичные.
- Но каждый шаг занимает больше времени.
- Thunks могут занимать больше памяти, чем результат их вычисления (а могут и меньше).
- Классический пример:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs (x + acc)
```

```
sum [1..100] 0  $\rightsquigarrow$  sum 1:[2..100] 0  $\rightsquigarrow$ 
```

```
sum [2..100] (1 + 0)  $\rightsquigarrow$  sum 2:[3..100] (1 + 0)  $\rightsquigarrow$ 
```

```
sum [3..100] (2 + (1 + 0))  $\rightsquigarrow$  ...  $\rightsquigarrow$ 
```

```
(100 + ... + (1 + 0))  $\rightsquigarrow$  4950
```

Минусы ленивости

- Из-за этого `foldl` — ловушка, практически всегда нужно либо `foldr`, либо `foldl'` (будет позже).

Минусы ленивости

- Из-за этого `foldl` — ловушка, практически всегда нужно либо `foldr`, либо `foldl'` (будет позже).
- С другой стороны, заметьте, что в этом примере список `[1..100]` никогда не появился целиком в памяти!

Минусы ленивости

- Из-за этого `foldl` — ловушка, практически всегда нужно либо `foldr`, либо `foldl'` (будет позже).
- С другой стороны, заметьте, что в этом примере список `[1..100]` никогда не появился целиком в памяти!
- Ещё один интересный пример:
`length (take 2 (filter (< 1) [1..]))`.

Минусы ленивости

- Из-за этого `foldl` — ловушка, практически всегда нужно либо `foldr`, либо `foldl'` (будет позже).
- С другой стороны, заметьте, что в этом примере список `[1..100]` никогда не появился целиком в памяти!
- Ещё один интересный пример:
`length (take 2 (filter (< 1) [1..]))`.
- Что же можно сделать?

Минусы ленивости

- Из-за этого `foldl` — ловушка, практически всегда нужно либо `foldr`, либо `foldl'` (будет позже).
- С другой стороны, заметьте, что в этом примере список `[1..100]` никогда не появился целиком в памяти!
- Ещё один интересный пример:
`length (take 2 (filter (< 1) [1..]))`.
- Что же можно сделать?
- Часто компилятор может оптимизировать ленивые вычисления (если знает, что это не изменит результата).
- Или...

Строгие вычисления в Haskell: seq

- Мы можем управлять ленивостью явно.
- Базовый инструмент для этого: «волшебная» (её нельзя реализовать на самом Haskell)
функция `seq :: a -> b -> b`.

Строгие вычисления в Haskell: seq

- Мы можем управлять ленивостью явно.
- Базовый инструмент для этого: «волшебная» (её нельзя реализовать на самом Haskell) функция `seq :: a -> b -> b`.
- `seq e1 e2` сначала приводит `e1` к WHNF, а после этого возвращает `e2`.

Строгие вычисления в Haskell: seq

- Мы можем управлять ленивостью явно.
- Базовый инструмент для этого: «волшебная» (её нельзя реализовать на самом Haskell) функция `seq :: a -> b -> b`.
- `seq e1 e2` сначала приводит `e1` к WHNF, а после этого возвращает `e2`.
- На практике всегда `e1` — переменная в `e2`.
- Пример использования:
$$f \$! x = seq\ x\ (f\ x)$$
- Это аналог `$`, только вычисляющий аргумент до WHNF перед вызовом функции.

sum через seq

- Теперь можем определить sum так:

sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = let a1 = x + acc in  
                  a1 `seq` sum xs a1
```

- И процесс вычисления:

```
sum [1..100] 0 ~> sum 1:[2..100] 0
```

sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = let a1 = x + acc in  
                  a1 `seq` sum xs a1
```

- И процесс вычисления:

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>  
let a1 = 1 + 0 in a1 `seq` sum [2..100] a1
```

sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = let a1 = x + acc in  
                  a1 `seq` sum xs a1
```

- И процесс вычисления:

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>
```

```
let a1 = 1 + 0 in a1 `seq` sum [2..100] a1 ~>
```

```
sum [2..100] 1 ~> sum 2:[3..100] 1
```

sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = let a1 = x + acc in  
                  a1 `seq` sum xs a1
```

- И процесс вычисления:

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>
```

```
let a1 = 1 + 0 in a1 `seq` sum [2..100] a1 ~>
```

```
sum [2..100] 1 ~> sum 2:[3..100] 1 ~>
```

```
let a1 = 2 + 1 in a1 `seq` sum [3..100] a1
```


sum через seq

- Теперь можем определить sum так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = sum xs $! x + acc
```

- Или так:

```
sum [] acc = acc
```

```
sum (x:xs) acc = let a1 = x + acc in  
                  a1 `seq` sum xs a1
```

- И процесс вычисления:

```
sum [1..100] 0 ~> sum 1:[2..100] 0 ~>  
let a1 = 1 + 0 in a1 `seq` sum [2..100] a1 ~>  
sum [2..100] 1 ~> sum 2:[3..100] 1 ~>  
let a1 = 2 + 1 in a1 `seq` sum [3..100] a1  
~> ... ~> 4950
```

foldl'

- Мы знаем, что сумма списка — частный случай свёртки.
- Для функций `product`, `minimum`, `maximum` имеем ровно те же проблемы!
- Можем определить вариант `foldl` (более общая версия в `Data.Foldable`):

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = let a' = f a x
                    in a' `seq` foldl' f a' xs
```

foldl'

- Мы знаем, что сумма списка — частный случай свёртки.
- Для функций `product`, `minimum`, `maximum` имеем ровно те же проблемы!
- Можем определить вариант `foldl` (более общая версия в `Data.Foldable`):

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = let a' = f a x
                    in a' `seq` foldl' f a' xs
```

```
sum xs = foldl' (+) 0 xs
product xs = foldl' (*) 1 xs
minimum xs = foldl1' min xs
```

...

Строгие переменные в образцах

- В `foldl'` параметр асс вычисляется только до WHNF. Если мы вычисляем среднее так

```
mean :: [Double] -> Double
mean xs = s' / fromIntegral l'
  where (s', l') = foldl' step (0, 0) xs
         step (s, l) a = (s + a, l + 1)
```

то в `s` и `l` снова накапливаются вычисления!

Строгие переменные в образцах

- В `foldl'` параметр асс вычисляется только до WHNF. Если мы вычисляем среднее так

```
mean :: [Double] -> Double
mean xs = s' / fromIntegral l'
  where (s', l') = foldl' step (0, 0) xs
        step (s, l) a = (s + a, l + 1)
```

то в `s` и `l` снова накапливаются вычисления!

- Включив `BangPatterns`, сделаем их строгими:

```
step (!s, !l) a = (s + a, l + 1)
```

- Это превращается в

```
step (s, l) a = let s' = s + a; l' = l + 1
                 in s' `seq` l' `seq` (s', l')
```

- Правила перевода **довольно сложные**.

Строгие поля в типах данных

- Другой способ решить ту же проблему:

```
data SPair a b = SPair !a !b
```

```
mean :: [Double] -> Double
```

```
mean xs = s' / fromIntegral l'
```

```
where SPair s' l' = foldl' step (0, 0) xs
```

```
    step (SPair s l) a = SPair (s + a) (l + 1)
```

Строгие поля в типах данных

- Другой способ решить ту же проблему:

```
data SPair a b = SPair !a !b
```

```
mean :: [Double] -> Double
```

```
mean xs = s' / fromIntegral l'
```

```
where SPair s' l' = foldl' step (0, 0) xs
```

```
      step (SPair s l) a = SPair (s + a) (l + 1)
```

- Восклицательный знак означает, что любое вычисление значения с этим конструктором вычислит и эти поля (до WHNF).
- То есть конструктор работает как

```
SPair a b = a `seq` b `seq` SPair' a b
```


где `SPair'` — конструктор, который получится без строгих полей.

- [Why Functional Programming Matters](#) (статья, написанная до появления собственно Haskell)
- [All About Strictness](#)
- [The Incomplete Guide to Lazy Evaluation](#)
- [Laziness в Haskell Wikibook](#)