CS314. Функциональное программирование

Лекция 10. Исполнение программ на языке Haskell

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

20 октября 2017 г.

Содержание

Строгая функция

Определение

Функция одного аргумента f называется строгой, если она не завершается или завершается с ошибкой в случаях, когда либо вычисление её аргумента зацикливается, либо значение аргумента не определено.

Строгая функция

Определение

Функция одного аргумента f называется строгой, если она не завершается или завершается с ошибкой в случаях, когда либо вычисление её аргумента зацикливается, либо значение аргумента не определено.

Обозначения

- $\perp$ (bottom, «дно»)— неопределённое значение (например, незавершающееся вычисление или `undefined`);
- f — строгая, если $f\ \perp = \perp$.

Примеры строгих функций

Функция (+1)

```
ghci> undefined + 1
*** Exception: Prelude.undefined
ghci> (length $ repeat 1) + 1
...
```

Функция head

```
ghci> head undefined
*** Exception: Prelude.undefined
```

Примеры нестрогих функций

Сокращённое вычисление булевых функций

```
ghci> True || undefined
True
ghci> False && undefined
False
ghci> True && undefined
*** Exception: Prelude.undefined
```

Константная функция

```
ghci> const 1 undefined
1
```

Примеры нестрогих функций

```
f x y
  | x > 0 =  x
  | otherwise = y
```

```
ghci> f 5 undefined
5
ghci> f undefined 5
*** Exception: Prelude.undefined
```

- f — нестрогая по второму аргументу и строгая по первому.

Нестрогая семантика: выводы

- Большинство языков программирования имеют строгую семантику.
- Даже в строгих языках такие конструкции, как условная операция или булевы выражения, обычно реализуются с нестрогой семантикой.
- Язык Haskell имеет нестрогую семантику, поэтому функции в нём по умолчанию нестрогие, значение аргумента вычисляется только при необходимости.
- Нестрогая семантика реализуется посредством «ленивого» вычисления.
- Нестрогая семантика является важнейшим фактором, влияющим на эффективность исполнения программ на языке Haskell.

Содержание

«Частичная нестрогость» head

```
ghci> head undefined
*** Exception: Prelude.undefined
ghci> head (1:undefined)
1
```

- В строгих языках список `1:undefined` неопределён.
- В языке Haskell указания первого элемента списка достаточно для вычисления результата функции `head`, при этом нет необходимости вычислять всё остальное.

Содержание

Задумки

```
let (x, y) = (length [1..5], reverse "olleh") in ...
```

- Значения `x` и `y` не вычисляются, вместо них в памяти сохраняются «задумки» (thunk, отложенное вычисление) — структуры данных, содержащие указания (рецепты), что нужно сделать для вычисления значения при возникновении такой необходимости.

Задумки

```
let (x, y) = (length [1..5], reverse "olleh") in ...
```

- Значения `x` и `y` не вычисляются, вместо них в памяти сохраняются «задумки» (thunk, отложенное вычисление) — структуры данных, содержащие указания (рецепты), что нужно сделать для вычисления значения при возникновении такой необходимости.

```
let z = (length [1..5], reverse "olleh") in ...
```

- Здесь само `z` оказывается задумкой, пара не подвергается деконструкции и вычислению значений компонентов.

Частичное вычисление задумки

```
let z      = (length [1..5], reverse "olleh")
    (n, s) = z
in ...
```

- При вычислении второй строки кода z деконструируется из задумки до пары задумок n и s.
- Между n и s с одной стороны и z с другой организуется связь (sharing = разделение).

Частичное вычисление задумки

```
let z      = (length [1..5], reverse "olleh")
    (n, s) = z
    'h':ss = s
in ...
```

- Происходит деконструкция списка во втором компоненте пары:
  1. s = *thunk* : *thunk*
  2. s = 'h' : *thunk*
- Значение ss — задумка, более глубокое вычисление не требуется.

Уровни вычисления значения (4, [1, 2])

1. thunk
2. (thunk, thunk)
3. (4, thunk)
4. (4, 1:thunk)
5. (4, 1:2:thunk)
6. (4, 1:2:[ ])

- Уровень 1 — невычисленное значение.
- Значения на уровнях 2–5 находятся в слабой головной нормальной форме.
- Значение на уровне 6 находится в нормальной форме.

Вычисление до требуемого уровня

- Использование аргумента в теле функции обычно требует его вычисления до некоторого уровня.
- Для head достаточно выделения первого элемента списка:

```
head (x:thunk) = x
```

Вычисление до требуемого уровня

- Использование аргумента в теле функции обычно требует его вычисления до некоторого уровня.
- Для head достаточно выделения первого элемента списка:

```
head (x:thunk) = x
```

- Функция length должна выяснить количество элементов, но значения самих элементов для неё несущественны:

```
length (thunk : thunk : thunk : [ ]) = 3
```

Вычисление до требуемого уровня

- Использование аргумента в теле функции обычно требует его вычисления до некоторого уровня.
- Для head достаточно выделения первого элемента списка:

```
head (x:thunk) = x
```

- Функция length должна выяснить количество элементов, но значения самих элементов для неё несущественны:

```
length (thunk : thunk : thunk : [ ]) = 3
```

- Функции show необходимо вычисление до нормальной формы:

```
show "hello" = "hello"
```

Строгость и вычисление до требуемого уровня

- Можно говорить об уровне строгости.
- Функцию называют более строгой, если для вычисления её результата требуется более глубокий уровень вычисления значения аргумента.
- Уровни строгости для функций нескольких аргументов могут отличаться по аргументам.
- Функцию называют ленивой по своему аргументу, если она не требует вычисления его значения хотя бы на один уровень (до слабой головной нормальной формы), и строгой в противном случае.
- Некоторым функциям для вычисления результата необходимо вычисление аргумента до нормальной формы.

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
⇒ head $ map thunk thunk
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
⇒ head $ map thunk thunk
⇒ head $ map (2 *) (1:thunk)
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
⇒ head $ map thunk thunk
⇒ head $ map (2 *) (1:thunk)
⇒ head $ (2 * 1) : map (2 *) thunk
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
⇒ head $ map thunk thunk
⇒ head $ map (2 *) (1:thunk)
⇒ head $ (2 * 1) : map (2 *) thunk
⇒ 2 * 1
```

Пример исполнения программы с учётом ленивости

```
head $ map (2 *) [1 .. 10]
⇒ head thunk
⇒ head $ map thunk thunk
⇒ head $ map (2 *) (1:thunk)
⇒ head $ (2 * 1) : map (2 *) thunk
⇒ 2 * 1
⇒ 2
```

Содержание

Хвостовая рекурсия

Определение

Говорят, что для некоторой рекурсивной функции имеет место хвостовая рекурсия, если результатом функции является результат её рекурсивного вызова.
Если результат рекурсивного вызова подвергается дополнительной обработке, то рекурсия не является хвостовой.

Хвостовая рекурсия

```
f 0 = 1
f n = f (n-1)
```

Нехвостовая рекурсия

```
f 0 = 1
f n = 1 + f (n-1)
```

Аккумулирующий параметр

Пример: сумма чисел от 1 до n (с хвостовой рекурсией)

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = sumN' (acc+n) (n-1)
```

- `acc` — аккумулирующий параметр (накапливает значение суммы между рекурсивными вызовами).

Аккумулирующий параметр

Пример: сумма чисел от 1 до n (с хвостовой рекурсией)

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = sumN' (acc+n) (n-1)
```

- `acc` — аккумулирующий параметр (накапливает значение суммы между рекурсивными вызовами).

Пример: сумма чисел от 1 до n (без хвостовой рекурсии)

```
sumN 0 = 0
sumN n = n + sumN (n-1)
```

Оптимизация хвостовой рекурсии в строгих языках

- Большое число рекурсивных вызовов приводит к переполнению стека, так как каждый рекурсивный вызов создаёт запись активации.
- Хвостовая рекурсия позволяет реализовать рекурсивные вызовы в одной и той же записи активации (меняются только значения параметров).
- Фактически хвостовая рекурсия может заменяться компилятором на цикл.

Вычисление с оптимизацией

```
sumN' 0 5 = sumN' 5 4 = sumN' 9 3 = sumN' 12 2 =
= sumN' 14 1 = sumN' 15 0 = 15
```

Глубокая рекурсия в Haskell

```
ghci> sumN 100000000
```

Глубокая рекурсия в Haskell

```
ghci> sumN 100000000
```

```
sumN n = sumN' 0 n
 where
  sumN' acc 0 = acc
  sumN' acc n = sumN' (acc+n) (n-1)
```

- Долгие вычисления и значительный расход памяти.

```
sumN 0 = 0
sumN n = n + sumN (n-1)
```

- Переполнение стека.

Аккумулирующие параметры в условиях ленивости

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = sumN' (acc+n) (n-1)
```

- Аккумулирующий параметр при каждом вызове сохраняется в виде задумки.

Аккумулирующие параметры в условиях ленивости

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = sumN' (acc+n) (n-1)
```

- Аккумулирующий параметр при каждом вызове сохраняется в виде задумки.

```
sumN' 0 5 = sumN' (0+5) 4 = sumN' ((0+5)+4) 3 =
= sumN' (((0+5)+4)+3) 2 = sumN' ((((0+5)+4)+3)+2) 1 =
= sumN' (((((0+5)+4)+3)+2)+1) 0 = (((((0+5)+4)+3)+2)+1) =
= ... = 15
```

Аккумулирующие параметры в условиях ленивости

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = sumN' (acc+n) (n-1)
```

- Аккумулирующий параметр при каждом вызове сохраняется в виде задумки.

```
sumN' 0 5 = sumN' (0+5) 4 = sumN' ((0+5)+4) 3 =
= sumN' (((0+5)+4)+3) 2 = sumN' ((((0+5)+4)+3)+2) 1 =
= sumN' (((((0+5)+4)+3)+2)+1) 0 = (((((0+5)+4)+3)+2)+1) =
= ... = 15
```

- Почему нет задумок по второму аргументу?

Функция seq

```
seq :: a -> b -> b
```

- По определению функция `seq` — строгая по первому аргументу:
  - `⊥ `seq` b = ⊥`
  - `a `seq` b = b`
- при вызове `seq` необходимо убедиться, что значение первого аргумента не является неопределённым, в случае с числом для этого его необходимо вычислить.

Функция seq

```
seq :: a -> b -> b
```

- По определению функция `seq` — строгая по первому аргументу:
  - `⊥ `seq` b = ⊥`
  - `a `seq` b = b`
- при вызове `seq` необходимо убедиться, что значение первого аргумента не является неопределённым, в случае с числом для этого его необходимо вычислить.

```
ghci> undefined `seq` 5
*** Exception: Prelude.undefined
ghci> (undefined, undefined) `seq` 5
5
```

Функция seq: более сложный случай

```
ghci> seq ((\True x -> "qq") undefined) 123
???
ghci> seq ((\True -> \x -> "qq") undefined) 123
???
```

Функция seq: более сложный случай

```
ghci> seq ((\True x -> "qq") undefined) 123
123
ghci> seq ((\True -> \x -> "qq") undefined) 123
*** Exception: Prelude.undefined
```

Функция seq: более сложный случай

```
ghci> seq ((\True x -> "qq") undefined) 123
123
ghci> seq ((\True -> \x -> "qq") undefined) 123
*** Exception: Prelude.undefined
```

- В первом случае имеет место частичное применение функции, поэтому сопоставление паттерна `True` с параметром `undefined` не производится (всё равно второго параметра ещё нет), но уже ясно что результат не является неопределённым (результат — это функция).

Функция seq: более сложный случай

```
ghci> seq ((\True x -> "qq") undefined) 123
123
ghci> seq ((\True -> \x -> "qq") undefined) 123
*** Exception: Prelude.undefined
```

- В первом случае имеет место частичное применение функции, поэтому сопоставление паттерна `True` с параметром `undefined` не производится (всё равно второго параметра ещё нет), но уже ясно что результат не является неопределённым (результат — это функция).
- Во втором случае применение полное, поэтому для выяснения, определён ли результат, необходимо выполнить сопоставление `True` с `undefined`, поэтому генерируется исключение.

Пример с суммой чисел

Использование seq

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = acc `seq` sumN' (acc+n) (n-1)
```

Пример с суммой чисел

Использование seq

```
sumN n = sumN' 0 n
  where
    sumN' acc 0 = acc
    sumN' acc n = acc `seq` sumN' (acc+n) (n-1)
```

Расширение BangPatterns

```
{-# LANGUAGE BangPatterns #-}

sumN n = sumN' 0 n
  where
    sumN' !acc 0 = acc
    sumN' !acc n = sumN' (acc+n) (n-1)
```

Содержание

Левая и правая свёртки

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

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

Левая и правая свёртки

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

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

- Хвостовая рекурсия и аккумулирующий параметр в левой свёртке.
- Нехвостовая рекурсия и потенциально большой стек рекурсивных вызовов в правой свёртке.

Суммирование с помощью правой свёртки

```
ghci> foldr (+) 0 [1..1000000]
*** Exception: stack overflow
```

Суммирование с помощью правой свёртки

```
ghci> foldr (+) 0 [1..1000000]
*** Exception: stack overflow
```

```
foldr (+) 0 [1..1000000]
  ⇒ 1+(foldr (+) 0 [2..1000000])
  ⇒ 1+(2+(foldr (+) 0 [3..1000000]))
  ⇒ 1+(2+(3+(foldr (+) 0 [4..1000000])))
```

- Множество рекурсивных вызовов и переполнение стека.

Суммирование с помощью левой свёртки

```
foldl (+) 0 [1..n]
  ⇒ foldl (+) (0+1) [2..n]
  ⇒ foldl (+) ((0+1)+2) [3..n]
  ⇒ foldl (+) (((0+1)+2)+3) [4..n]
```

- Хвостовая рекурсия, аккумулирующий параметр и большой объём динамической памяти для хранения задумок.

Строгая левая свёртка (Data.List.foldl')

```
foldl' (+) 0 [1..n]
  ⇒ foldl' (+) (0+1) [2..n]
  ⇒ foldl' (+) (1+2) [3..n]
  ⇒ foldl' (+) (3+3) [4..n]
  ⇒ foldl' (+) (6+4) [5..n]
  ⇒ ...
```

Строгая левая свёртка (Data.List.foldl')

```
foldl' (+) 0 [1..n]
  ⇒ foldl' (+) (0+1) [2..n]
  ⇒ foldl' (+) (1+2) [3..n]
  ⇒ foldl' (+) (3+3) [4..n]
  ⇒ foldl' (+) (6+4) [5..n]
  ⇒ ...
```

```
foldl' _ z [ ]     = z
foldl' f z (x:xs) = z `seq` foldl' f (f z x) xs
```

```
foldl  _ z [ ]     = z
foldl  f z (x:xs) = foldl f (f z x) xs
```

Левая свёртка с ленивой комбинирующей функцией

```
(?) :: Int -> Int -> Int
_ ? 0 = 0
x ? y = x*y
```

```
okey = foldl (?) 1 [2,3,undefined,5,0]

boom = foldl' (?) 1 [2,3,undefined,5,0]
```

Левая свёртка с ленивой комбинирующей функцией

```
(?) :: Int -> Int -> Int
_ ? 0 = 0
x ? y = x*y
```

```
okey = foldl (?) 1 [2,3,undefined,5,0]

boom = foldl' (?) 1 [2,3,undefined,5,0]
```

- В первом случае строится задумка `(((((1 ? 2) ? 3) ? undefined) ? 5) ? 0)`, результат вычисления которой — 0.
- Во втором случае происходит умножение на `undefined` и генерируется исключение.

Содержание

Рекурсия, ограниченная конструктором данных

```
map _ [ ]     = [ ]
map f (x:xs) = f x : map f xs
```

- Последняя операция в теле функции — конструктор списка `(:)`.
- Результат вызова функции может участвовать в ленивых вычислениях, например, с `head` и другими частично строгими функциями, не дожидаясь полного вычисления.

«Правильная» рекурсия в foldr

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

- Если комбинирующая функция f является конструктором списка, то рекурсивный вызов может быть отложен до появления в нём необходимости (то есть может вообще никогда не произойти).

Содержание

Распространённые идиомы ленивых вычислений

- Разделение генерации и фильтрации данных (будут сгенерированы только те данные, которые необходимы для вычисления ответа).

```
prune . generate
```

```
take 3 . sort -- зависит от алгоритма сортировки
```

Распространённые идиомы ленивых вычислений

- Разделение генерации и фильтрации данных (будут сгенерированы только те данные, которые необходимы для вычисления ответа).

```
prune . generate
```

```
take 3 . sort -- зависит от алгоритма сортировки
```

- Повторное использование кода.

Распространённые идиомы ленивых вычислений

- Разделение генерации и фильтрации данных (будут сгенерированы только те данные, которые необходимы для вычисления ответа).

```
prune . generate
```

```
take 3 . sort -- зависит от алгоритма сортировки
```

- Повторное использование кода.
- Бесконечные структуры данных и «завязывание узлов».

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Недостатки

- Хранение задумок (даже значения примитивных типов хранятся как задумки, поскольку их значением может быть `undefined`).
- Написание высокоэффективной программы требует владения сложной техникой (анализ строгости, unboxed-значения и управление порядком вычислений).

Ленивые вычисления — хорошо или плохо?

Ленивые вычисления — хорошо или плохо?

Разнообразие языков программирования — это прекрасно!

Содержание

Сложность основных операций обработки списков

- `head` — $O(1)$
- `tail` — $O(1)$
- `(:)` — $O(1)$
- `(!! n)` — $O(n)$
- `length` — $O(n)$
- `last` — $O(n)$
- `(++)` — $O(n)$, где n — длина первого списка
- `fold* step` — $O(n) \times$ сложность step
- `sort` — $O(n \log n)$

Сложность основных операций обработки списков

- `head` — $O(1)$
- `tail` — $O(1)$
- `(:)` — $O(1)$
- `(!! n)` — $O(n)$
- `length` — $O(n)$
- `last` — $O(n)$
- `(++)` — $O(n)$, где n — длина первого списка
- `fold* step` — $O(n)$ $\times$ сложность step
- `sort` — $O(n \log n)$

Пример для свёрток: обращение списка

```
reverse'  = foldr (\x acc -> acc ++ [x]) []   -- O(n^2)
reverse'' = foldl (\acc x -> x:acc)      []   -- O(n)
```

Сложность операций с учётом ленивости

Какова сложность функции minEl?

```
minEl = head . sort
```

Сложность операций с учётом ленивости

Какова сложность функции minEl?

```
minEl = head . sort
```

В языке со строгим вычислением

$$O(1) + O(n \log n) = O(n \log n)$$

Сложность операций с учётом ленивости

Какова сложность функции minEl?

```
minEl = head . sort
```

В языке со строгим вычислением

$$O(1) + O(n \log n) = O(n \log n)$$

В Haskell

- Зависит от реализации `sort`.

Сложность операций с учётом ленивости

Какова сложность функции minEl?

```
minEl = head . sort
```

В языке со строгим вычислением

$$O(1) + O(n \log n) = O(n \log n)$$

В Haskell

- Зависит от реализации `sort`.
- Вроде бы $O(n)$.

Сложность операций с учётом ленивости

Какова сложность функции minEl?

```
minEl = head . sort
```

В языке со строгим вычислением

$$O(1) + O(n \log n) = O(n \log n)$$

В Haskell

- Зависит от реализации `sort`.
- Вроде бы $O(n)$.
- Если реализация sort использует рекурсию, ограниченную конструктором данных (например, mergesort), то точно $O(n)$.

Пример: среднее значение элементов списка

Версия 1

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

Пример: среднее значение элементов списка

Версия 1

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

Хотя

- сложность: $O(n)$

Пример: среднее значение элементов списка

Версия 1

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

Хотя

- сложность: $O(n)$

Но

- два прохода по списку
- список должен храниться в памяти целиком

Пример: среднее значение элементов списка

Версия 1

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

Хотя

- сложность: $O(n)$

Но

- два прохода по списку
- список должен храниться в памяти целиком

Версия 2

```
mean :: [Double] -> Double
mean xs = s / fromIntegral n
  where
    (n, s)        = foldl' k (0, 0) xs
    k (!n, !s) x = (n+1, s+x)
```

Литература

- Саймон Марлоу. Параллельное и конкурентное программирование на языке Haskell (первый раздел главы 2 — Ленивые вычисления и слабая головная нормальная форма).
- Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell (глава 25 — Профилирование и оптимизация).