Свёртки/ленивые вычисления

Дмитрий Халанский

19 октября 2020 г.

1. Изучение Haskell
   - Как?
   - Зачем?
   - Почему?
2. Свёртки
3. Свёртка списка
4. Модель вычислений
5. Утечки памяти

1. Изучение Haskell
   - Как?
   - Зачем?
   - Почему?
2. Свёртки
3. Свёртка списка
4. Модель вычислений
5. Утечки памяти

Что есть в Haskell

- Переменные;
- Аппликации;
- Абстракции;
- `error`;
- Несколько примитивных типов (числа и т.п.);
- Типы-суммы и сопоставление с образцом;
- Типы-записи и проекции;
- Классы типов;
- Несколько встроенных конструкций для взаимодействия с операционной системой.

Haskell — *очень* простой язык. Важный принцип его разработки — как можно меньше магии на стороне языка. На самом деле магия есть, но её приходится искать специально.

Чего нет в Haskell

В Haskell нет:

- Обработки исключений;
- Присваиваний;
- Классов с полями;
- Перегрузки операторов;
- Коллекций;
- Глобального состояния;
- Циклов.

Но всё это можно написать *на* Haskell и пользоваться этим, как если бы оно было в языке изначально.

Императивная программа на Haskell

```
program = do tell "Beginning␣execution"
             v <- newIORef 0
             forM_ [1..10] $ \i -> do
               vValue <- readIORef v
               tell $ "v␣is" ++ show vValue
               writeIORef v (i + vValue)
             readIORef v
```

Минус по сравнению с Python: дичайший синтаксис.

Плюс по сравнению с Python: ноль магии, в каждое определение можно заглянуть.

Как понимать Haskell

Неправильно Принимать на веру лекции и практики и пытаться по примерам нащупать, как этим пользоваться. Этот подход неизбежен для языков с волшебными правилами (например, `https://docs.python.org/3/reference/datamodel.html#special-method-names` или правила перегрузки функций в C++ можно только выучить, но не понять).

Правильно Открыть определение и почитать; может, поредуцировать руками простые примеры. Например, по опыту прошлых лет, понять трансформеры монад можно только тогда, когда посмотришь на то, какими конкретными значениями всё представляется и что именно делает каждая функция. Лучший способ осознать что-то с курса ФП — сесть и реализовать то же самое самостоятельно.

1. Изучение Haskell
   - Как?
   - Зачем?
   - Почему?
2. Свёртки
3. Свёртка списка
4. Модель вычислений
5. Утечки памяти

Влияние на индустрию

- Даже в Java-коде встречаются `map`, `foldr` и прочие функциональные комбинаторы.
- Функции высших порядков вообще встречаются уже более-менее везде.
- Сопоставление с образцом уже есть в Scala и C#, скоро будет в Java.
- Аппликативные функторы и монады (скоро изучим) тоже иногда встречаются в коде или *сильно* помогают его пониманию.
- Параметрический полиморфизм и автоматический вывод типов тоже потихоньку приходит в используемые языки: Swift, Rust.
- Моноиды полезны для описания распределённых вычислений. Например, многопользовательские текстовые редакторы на технологии CRDT целиком основаны на концепции моноидов...

Вывод: не напороться на ФП можно, но сложно. Надо его не бояться.

1. Изучение Haskell
   - Как?
   - Зачем?
   - Почему?
2. Свёртки
3. Свёртка списка
4. Модель вычислений
5. Утечки памяти

Зачем всем сдалось это ФП?

Вкратце: типичный день программиста — чтение полотен кода и попытки их понять.

- Глядя на map, легко понять, что это map и в нём точно нет проблем. Чтобы осознать цикл со счётчиком, нужно внимательно прочитать заголовок цикла, проверить, не меняется ли счётчик в теле, перепроверить тип счётчика (unsigned/signed), и только потом убедиться, что всё, что делает некий цикл, — это отображение одного массива на другой. Проще говоря, ФП — это набор мощных абстракций, которые позволяют мыслить о коде на уровне выше и не задумываться о мелочах.
- Помимо этого, ФП даёт возможности по построению своих собственных абстракций, которые внутри могут прятать нетривиальную логику и высокую производительность, но дают простой интерфейс. Алгебра (моноиды, группы, полугруппы, функторы и всё такое) применительно к программированию — наука как раз о том, как делать интуитивн понятные интерфейсы. Чудесная книга по теме: `https://algebradriven.design/`.

1 Изучение Haskell
- Как?
- Зачем?
- Почему?

2 Свёртки

3 Свёртка списка

4 Модель вычислений

5 Утечки памяти

Две интерпретации **foldr**

- Функция с сигнатурой

  ```
  foldr :: Foldable f => (a -> b -> b) -> b -> f a -> b
  ```

  Такая функция аналогична понятию коллекций и foreach-циклов в других языках.

  ```
  state = initial_state
  for a in lst:
    state = update_state(a, state)
  ```

  эквивалентно

  ```
  foldr update_state initial_state lst
  ```

- Функция-*свёртка*, которая заменяет везде один конструктор на одну функцию, а другой — на другую и “схлопывает” результат. Это эквивалентно паттерну проектирования “Visitor” из ООП.

Свёртка списков

Возьмём и поменяем местами аргументы **foldr**:

```
foldrAlt :: [a] -> (a -> b -> b) -> b -> b
foldrAlt xss f ini = foldr f ini xss
```

Тогда

```
foldrAlt [1, 2, 3] = \f a -> f 1 (f 2 (f 3 a))
```

Что это?

Свёртка списков

Возьмём и поменяем местами аргументы **foldr**:

```
foldrAlt :: [a] -> (a -> b -> b) -> b -> b
foldrAlt xss f ini = foldr f ini xss
```

Тогда

```
foldrAlt [1, 2, 3] = \f a -> f 1 (f 2 (f 3 a))
```

Что это? Кодирование списков в лямбда-исчислении! Следствие: всё, что можно сделать с помощью сопоставления с образцом, гарантированно можно выразить через свёртку.

```
myTail :: [a] -> Maybe [a]
myTail = fst . foldr (\x (_, xs) -> (Just xs, x:xs))
                     (Nothing, [])
myHead :: [a] -> Maybe a
myHead = foldr (\x b -> Just x) Nothing
myNull :: [a] -> Bool
myNull = foldr (\_ _ -> False) True
```

Свёртка Maybe

Напомним определение:

```
data Maybe a = Nothing | Just a
```

Хотим завести функцию, которая будет **Nothing** заменять на одну функцию, а **Just** — на другую. Какой её тип?

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Какая реализация?

Свёртка Maybe

Напомним определение:

```
data Maybe a = Nothing | Just a
```

Хотим завести функцию, которая будет **Nothing** заменять на одну функцию, а **Just** — на другую. Какой её тип?

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Какая реализация?

```
maybe b f Nothing = b
maybe b f (Just a) = f a
```

Всё, что возможно сделать с сопоставлением с образцом, можно переписать через эту функцию.

```
myInit = foldr (\x b -> Just $ case b of
                                Nothing -> []
                                Just xs -> (x:xs)) Nothing
myInit' = foldr (\x b -> Just $ maybe [] (x:) b) Nothing
-- foldr (\x -> Just . maybe [] (x:)) Nothing
```

Свёртка Nat

```
data Nat = Zero | Suc Nat
```

Хотим завести функцию, которая будет Zero заменять на одну функцию, а Nat — на другую. Тип и реализация:

Свёртка Nat

```haskell
data Nat = Zero | Suc Nat
```

Хотим завести функцию, которая будет Zero заменять на одну функцию, а Nat — на другую. Тип и реализация:

```haskell
foldNat :: a -> (a -> a) -> Nat -> a
foldNat a f Zero = a
foldNat a f (Suc n) = f $ foldNat a f n
```

Переставим аргументы:

```haskell
foldNatAlt n s z = foldNat z s n
```

Тогда

```haskell
foldNatAlt (Suc (Suc (Suc Zero))) = \s z -> s (s (s z))
```

Что это?

Свёртка Nat

```haskell
data Nat = Zero | Suc Nat
```

Хотим завести функцию, которая будет Zero заменять на одну функцию, а Nat — на другую. Тип и реализация:

```haskell
foldNat :: a -> (a -> a) -> Nat -> a
foldNat a f Zero = a
foldNat a f (Suc n) = f $ foldNat a f n
```

Переставим аргументы:

```haskell
foldNatAlt n s z = foldNat z s n
```

Тогда

```haskell
foldNatAlt (Suc (Suc (Suc Zero))) = \s z -> s (s (s z))
```

Что это?Числа Чёрча — это свёрнутые натуральные числа.

Свёртка (,)

Как свернуть пару?

Свёртка (,)

Как свернуть пару?

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a, b) = f a b
```

1 Изучение Haskell
- Как?
- Зачем?
- Почему?

2 Свёртки

3 Свёртка списка

4 Модель вычислений

5 Утечки памяти

Разворот списка

Как максимально просто реализовать разворот списка?

Разворот списка

Как максимально просто реализовать разворот списка?

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Перепишем в вид, который показывали на лекции:

```
reverse [] = []
reverse (y:ys) = (\x a -> a ++ [x]) y (reverse ys)
```

Выразим через **foldr**:

Разворот списка

Как максимально просто реализовать разворот списка?

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Перепишем в вид, который показывали на лекции:

```
reverse [] = []
reverse (y:ys) = (\x a -> a ++ [x]) y (reverse ys)
```

Выразим через **foldr**:

```
reverse = foldr (\x a -> a ++ [x]) []
```

Конкатенация

Попробуем бездумно, чисто механически оттранслировать сопоставление с образцом в свёртку:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Перепишем явно через сопоставление с образцом:

```
(+++) xs = case xs of
              [] -> \ys -> ys
              (x:xs) -> \ys -> x : ((xs +++) ys)
```

Перепишем через **foldr**:

```
(+++) = foldr (\x a -> \ys -> x : a ys) id
-- foldr (\x a -> (x :) . a) id
-- foldr (\x -> ((x :) .)) id
```

На самом деле есть куда более простая реализация:

```
xs +++ ys = foldr (:) ys xs
```

Разворот списка 2

Более быстрый разворот списка:

```
reverse = go []
  where go acc [] = acc
        go acc (x:xs) = go (x:acc) xs
```

go не имеет нужный вид: рекурсивный вызов не имеет вид go acc xs.

Что делать?

Разворот списка 2

Более быстрый разворот списка:

```
reverse = go []
  where go acc [] = acc
        go acc (x:xs) = go (x:acc) xs
```

go не имеет нужный вид: рекурсивный вызов не имеет вид go acc xs. Что делать?Переставим аргументы:

```
reverse = flip go []
  where go [] = \acc -> acc
        go (x:xs) = \acc -> go xs (x:acc)
```

Теперь рекурсивный вызов имеет вид go xs. Через **foldr**:

```
reverse = flip go []
  where go = foldr (\x a -> \acc -> a (x:acc)) id
```

Наконец, получили почти то же самое, что конкатенация:

```
reverse xs = foldr (\x a acc -> a (x:acc)) id xs []
-- foldr (\x -> (. (x:))) id xs []
```

1 Изучение Haskell
- Как?
- Зачем?
- Почему?

2 Свёртки

3 Свёртка списка

4 Модель вычислений

5 Утечки памяти

Аппроксимация модели вычислений Haskell

- Каждое вычисление кладётся в специальный ящик (thunk) и лежит там.
- Связь с вводом-выводом (например, печать в консоль) заставляет thunk доходить до нормальной формы.
- Нормализуем по нормальной стратегии.
- β-редукция:
  1. После того, как нужное число аргументов передано для проверки шаблона, идёт сверху вниз по списку шаблонов в поисках соответствия.
  2. При удачном соответствии заменяет связанные в шаблоне переменные на указатели на thunk'и соответствующих частей сопоставленной структуры.

Это не совсем правда, но хорошая отправная точка.

Подробнее про β-редукции (1)

- `(\x y -> y) undefined 3`

Подробнее про β-редукции (1)

- `(\x y -> y) undefined 3`

  Шаблон от двух аргументов. Приняли два аргумента, сопоставили. Сопоставление успешное, $[x \leftarrow \bot, y \leftarrow 3]$. Тело — y, что равно 3.

- `(\x -> (\y -> y)) undefined 3`

Подробнее про β-редукции (1)

- `(\x y -> y) undefined 3`

  Шаблон от двух аргументов. Приняли два аргумента, сопоставили. Сопоставление успешное, $[x \leftarrow \bot, y \leftarrow 3]$. Тело — y, что равно 3.

- `(\x -> (\y -> y)) undefined 3`

  Ответ: 3. Шаблон от одного аргумента. Приняли один аргумент, сопоставили. Сопоставление успешное, $[x \leftarrow \bot]$. Осталось тело — функция `\\y -> y`. Редуцируем дальше. Это шаблон от одного аргумента. Принимаем аргумент, успешно сопоставляем, $[y \leftarrow 3]$. Тело — y, что равно 3.

Подробнее про β-редукции (2)

- `(\(Just 3) -> id) Nothing`

Подробнее про β-редукции (2)

- `(\(Just 3) -> id) Nothing`

  Ответ: вылет. Приняли один аргумент, который нужен для проверки шаблона, попытались сопоставить, не вышло.

- `(\(Just 3) y -> y) Nothing`

Подробнее про β-редукции (2)

- `(\(Just 3) -> id) Nothing`

  Ответ: вылет. Приняли один аргумент, который нужен для проверки шаблона, попытались сопоставить, не вышло.

- `(\(Just 3) y -> y) Nothing`

  Ответ: функция от одного аргумента, которая вылетит при вызове. Для проверки шаблона нужно проверить два аргумента, а у нас их ещё нет. Дадут второй — проверим первый и вылетим.

- `(\x y -> let (Just z) = x in y) (Nothing :: Maybe ()) 3`

Подробнее про β-редукции (2)

- `(\(Just 3) -> id) Nothing`

  Ответ: вылет. Приняли один аргумент, который нужен для проверки шаблона, попытались сопоставить, не вышло.

- `(\(Just 3) y -> y) Nothing`

  Ответ: функция от одного аргумента, которая вылетит при вызове. Для проверки шаблона нужно проверить два аргумента, а у нас их ещё нет. Дадут второй — проверим первый и вылетим.

- `(\x y -> let (Just z) = x in y) (Nothing :: Maybe ()) 3`

  Ответ: 3. Принимаем два аргумента, сопоставляем с шаблонами, всё проходит. x не будет сопоставляться с образцом.

Что приводит к поиску WHNF (1)

- Поиск WHNF `seq a b` приведёт к нахождению WHNF как `b`, так и `a`.

Что приводит к поиску WHNF (1)

- Поиск WHNF `seq a b` приведёт к нахождению WHNF как `b`, так и `a`.

```
seq undefined 3
a = map succ ['a' .. 'z']
:sprint a
seq a 3
:sprint a
```

- Вычисление WHNF экземпляра структуры данных, строгого по какому-то аргументу конструктора, попытается вычислить WHNF аргумента;

Что приводит к поиску WHNF (1)

- Поиск WHNF `seq a b` приведёт к нахождению WHNF как `b`, так и `a`.

```
seq undefined 3
a = map succ ['a' .. 'z']
:sprint a
seq a 3
:sprint a
```

- Вычисление WHNF экземпляра структуры данных, строгого по какому-то аргументу конструктора, попытается вычислить WHNF аргумента;

```
newtype D = D Int -- strict
data D' = D' !Int
data D'' = D'' Int
seq (D undefined) 3 -- crash
seq (D' undefined) 3 -- crash
seq (D'' undefined) 3 -- ok
```

Что приводит к поиску WHNF (2)

- Для вычисления WHNF аппликации нужно вычислить WHNF тела:

Что приводит к поиску WHNF (2)

- Для вычисления WHNF аппликации нужно вычислить WHNF тела:

```
seq (undefined 3) 4 -- crash
```

- Если аргументов для сопоставления с образцом подано недостаточно, то такая аппликация уже в WHNF; если же их достаточно, то WHNF вычислит WHNF аргументов настолько, насколько нужно для проверки соответствия шаблонам.

Что приводит к поиску WHNF (2)

- Для вычисления WHNF аппликации нужно вычислить WHNF тела:

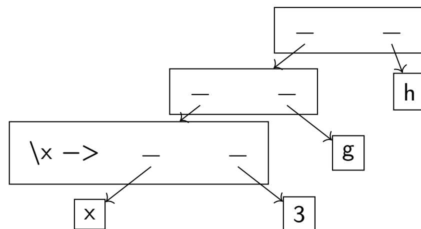
```
seq (undefined 3) 4 -- crash
```

- Если аргументов для сопоставления с образцом подано недостаточно, то такая аппликация уже в WHNF; если же их достаточно, то WHNF вычислит WHNF аргументов настолько, насколько нужно для проверки соответствия шаблонам.

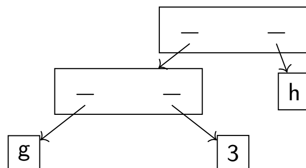
```
seq ((\Nothing 3 -> 5) undefined) 4 -- ok
seq ((\Nothing -> 5) undefined) 4 -- crash
seq ((\Nothing -> 5) undefined) 4 -- crash
seq ((\(x:xs) -> x) (3:undefined)) 4 -- ok
(\(Just x:xs) -> 3) (Just undefined:undefined) -- ok
(\(_:[]) -> 3) [undefined] -- ok
```

Пример структуры thunk

“(\x −> x 3) g h” (*максимально условно*) представляется как



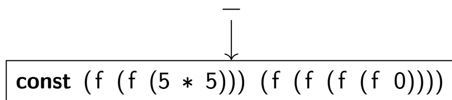
Видим аппликацию. Нормализуем сначала тело. Видим редекс. Сокращаем:



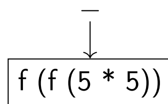
Длинный пример редуцирования

Возьмём `f a = a + a`.

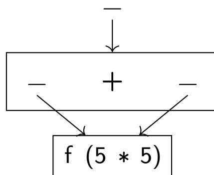
1. Вызовем **const** `(f (f (5 * 5))) (f (f (f (f 0))))`. Получаем ящик:



2. Раскрывая **const** по определению, получаем ящик:

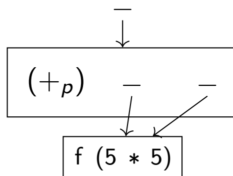


3. Раскрывая этот ящик по определению `f`, получим



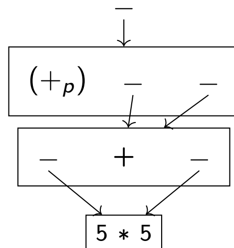
Длинный пример редуцирования

4. Раскрывая определение $+$, обнаруживаем, что это некий $(+_p)$ — функция, написанная уже не на языке Haskell, а являющаяся частью его среды исполнения и складывающая настоящие числа.



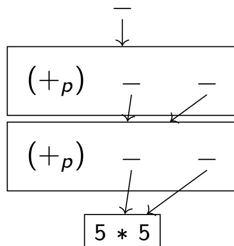
Длинный пример редуцирования

5. $(+_p)$ строга в своих аргументах. Нам нужно вычислить их до нормальной формы. Раскроем ящик.



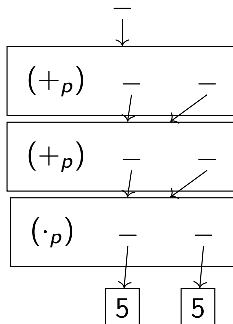
Длинный пример редуцирования

6. Снова раскроем $+$.



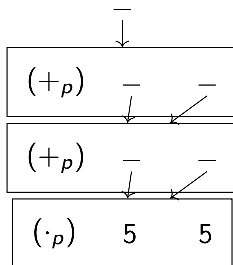
Длинный пример редуцирования

7. Раскроем $*$.



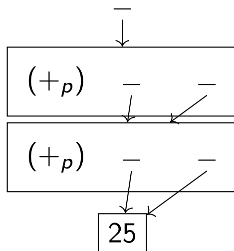
Длинный пример редуцирования

8. 5 — значение в нормальной форме. Подставим его в вызов.



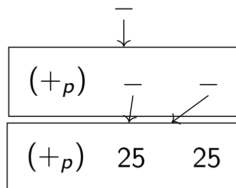
Длинный пример редуцирования

9. Вычислим.

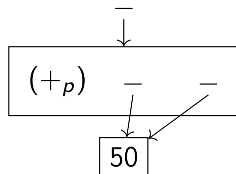


Длинный пример редуцирования

10. Подставим.



11. Вычислим.



Длинный пример редуцирования

12. Подставим.

$$\boxed{(+_p) \quad 50 \quad 50}$$

13. Вычислим.

$$\boxed{100}$$

14. Подставим!

100

1 Изучение Haskell
- Как?
- Зачем?
- Почему?

2 Свёртки

3 Свёртка списка

4 Модель вычислений

5 Утечки памяти

Пример с лекции

Подробно объяснено по ссылке
`https://wiki.haskell.org/Foldr_Foldl_Foldl'`.

scanl

Напишите такую функцию `scanl`, что

```
scanl (*) 1 []                 == [1]
scanl (*) 1 [1, 2, 3]          == [1, 1, 2, 6]
scanl (+) 1 [1, 2, 3]          == [1, 2, 4, 7]
scanl (++) "e" ["a", "b", "c"] == ["e","ea","eab","eabc"]
```

scanl

Напишите такую функцию scanl, что

```
scanl (*) 1 []                  == [1]
scanl (*) 1 [1, 2, 3]           == [1, 1, 2, 6]
scanl (+) 1 [1, 2, 3]           == [1, 2, 4, 7]
scanl (++) "e" ["a", "b", "c"] == ["e","ea","eab","eabc"]

scanl f a [] = [a]
scanl f a (x:xs) = a : scanl f (a `f` x) xs
```

last

Напишите такую функцию `last`, что

```
last [1, 2, 3] = 3
```

last

Напишите такую функцию `last`, что

```
last [1, 2, 3] = 3

last (x:[]) = x
last (x:xs) = last xs
```

Утечка

Нормальная функция:

```
length $ show $ scanl (+) 0 [1..10000000]
```

“Посчитать длину строкового представления списка частичных сумм списка чисел от 1 до 10000000”. Нормально считается и возвращает 143459631.

Упростим задачу: посчитаем длину даже не всего списка, а только последнего элемента.

```
length $ show $ last $ scanl (+) 0 [1..10000000]
```

Падает из-за того, что кончилась память (???).

Отслеживаем утечку (1)

Посмотрим с точки зрения thunk'ов. Забудем пока про `length . show`.

```
last (scanl (+) 0 [1, 2, 3]) =
case scanl (+) 0 [1, 2, 3] of
  (x:[]) -> x
  (_:xs) -> last xs =
case 0 : scanl (+) (0 + 1) [2, 3] of
  (x:[]) -> x
  (_:xs) -> last xs =
case 0 : (0 + 1) : scanl (+) ((0 + 1) + 2) [3] of
  (x:[]) -> x
  (_:xs) -> last xs =
last ((0 + 1) : scanl (+) ((0 + 1) + 2) [3]) =
```

Отслеживаем утечку (2)

```
last ((0 + 1) : scanl (+) ((0 + 1) + 2) [3]) =
case (0 + 1) : scanl (+) ((0 + 1) + 2) [3] of
  (x:[]) -> x
  (_:xs) -> last xs =
case (0 + 1) : ((0 + 1) + 2) :
        scanl (+) (((0 + 1) + 2) + 3) [] of
  (x:[]) -> x
  (_:xs) -> last xs =
last (((0 + 1) + 2) : scanl (+) (((0 + 1) + 2) + 3) [])
```

Отслеживаем утечку (3)

```
last (((0 + 1) + 2) : scanl (+) (((0 + 1) + 2) + 3) []) =
case ((0 + 1) + 2) : scanl (+) (((0 + 1) + 2) + 3) [] of
  (x:[]) -> x
  (_:xs) -> last xs =
case ((0 + 1) + 2):[(((0 + 1) + 2) + 3)] of
  (x:[]) -> x
  (_:xs) -> last xs =
last [(((0 + 1) + 2) + 3)] =
case [(((0 + 1) + 2) + 3)] of
  (x:[]) -> x
  (_:xs) -> last xs =
((0 + 1) + 2) + 3
```

Отслеживаем утечку (выводы)

Чтобы вернуть даже WHNF, `last` приходится проходить по всему списку, тем самым генерируя его. При этом он никак не форсирует вычисление самих элементов.
В противовес этому, вычисление

```
length $ show $ scanl (+) 0 [1..10000000]
```

проходит так (если интересно, смотрите в исходник `show` для списков):

(Длина списка с аккумулятором)

Считаем, что вычисление длины списка определено, например, так:

```
length = length' 0
    where length' n [] = n
          length' n (_:xs) = (length' $! 1 + n) xs
```

Отслеживаем отсутствие утечки

```
length $ show $ scanl (+) 0 [1..10000000] =
length' 0 $ "[" + show (scanl (+) 0 [1..10000000]) =
length' 1 $ show (0 : scanl (+) (0+1) [2..10000000]) =
length' 1 $ show 0 ++ ","
            ++ show (scanl (+) (0+1) [2..10000000] =
length' 1 $ "0," ++ show (scanl (+) (0+1) [2..10000000] =
length' 3 $ show (scanl (+) (0+1) [2..10000000]) =
length' 3 $ show ((0+1) : scanl (+) ((0+1)+2) [3..10000000]) =
length' 3 $ show (0+1) ++ ","
            ++ show (scanl (+) ((0+1)+2) [3..10000000]) =
length' 3 $ "1," ++ show (scanl (+) (1+2) [3..10000000]) =
length' 5 $ show (scanl (+) (1+2) [3..10000000]) = ...
```

Как видим, на предпоследнем шаге 0+1 сократилось.