Структуры данных

Дмитрий Халанский

05 октября 2020 г.

type — просто синоним

```
type TId a = a
type MyInt = Int
type ConstInt a = Int
type K a b = a
```

Тогда `TId Int = Int`, `MyInt = Int`, `ConstInt String = Int`.
Частичное применение типовых аргументов не работает: `K Int` нельзя использовать в программе.

Типы-суммы как наследование

Haskell:

```haskell
data Bool = True | False

(&&) a b = case a of
             True -> b
             False -> False
```

Примерный аналог на Python:

```python
class Bool:
  pass

class BTrue(Bool):
  pass

class BFalse(Bool):
  pass

def booleanAnd(a, b):
  if isinstance(a, BTrue):
    return b
  elif isinstance(a, BFalse):
    return BFalse()
```

Различие: “наследников” точно больше не появится, так что можно писать программы перечислением всех вариантов.

Типы-суммы как размеченные объединения

Haskell:

```haskell
data IntOrDouble = IODInt Int
                 | IODDouble Double

getInt :: IntOrDouble -> Int
getInt (IODInt i) = i
getInt (IODDouble d) =
  fromInteger (round d)
```

Примерный аналог на C:

```c
#include <math.h>
enum IntOrDoubleTag {
    INT_OR_DOUBLE_TAG_INT,
    INT_OR_DOUBLE_TAG_DOUBLE
};
typedef struct {
    enum IntOrDoubleTag tag;
    union {
        struct {
            int i;
        } intContents;
        struct {
            double d;
        } doubleContents;
    } contents;
} IntOrDouble;
int getInt(IntOrDouble i) {
    switch (i.tag) {
        case INT_OR_DOUBLE_TAG_INT:
            return i.contents
                .intContents.i;
        case INT_OR_DOUBLE_TAG_DOUBLE:
            return (int)round(
                i.contents
                    .doubleContents.d);
    }
}
```

newtype

```
newtype A a b c = Ac ([a], [b], c)
```

почти эквивалентно

```
data A a b c = Ac !([a], [b], c)
```

newtype предпочтителен в плане производительности: в скомпилированной программе он не встречается.

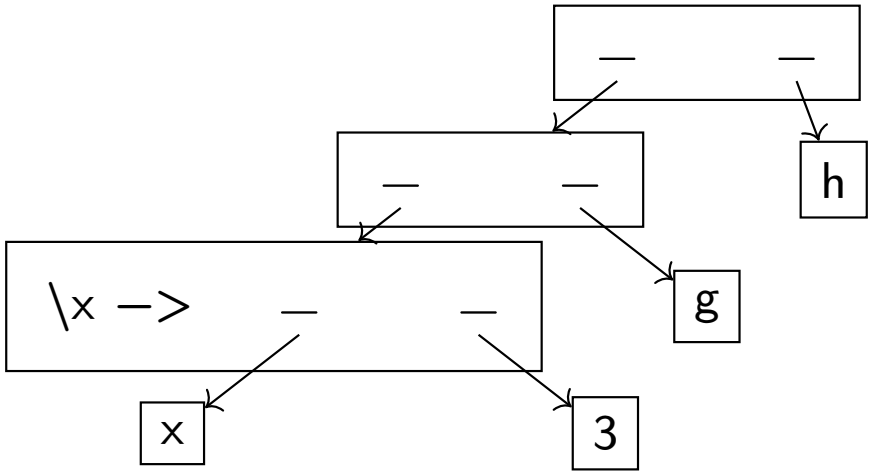
Аппроксимация модели вычислений Haskell

- 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 приводит к тому, что все вхождения редуцируемой переменной заменяются ссылкой на thunk, соответствующий аргументу.

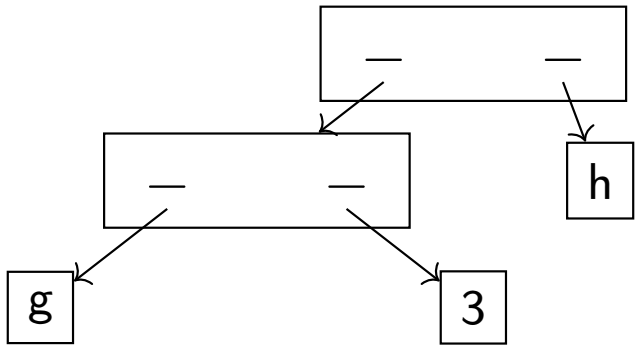
Это не совсем правда, но хорошая отправная точка.

Пример структуры thunk

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



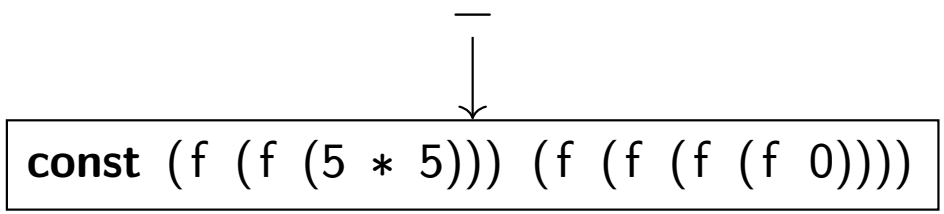
Видим аппликацию. Нормализуем сначала тело. Видим редекс. Сокращаем:



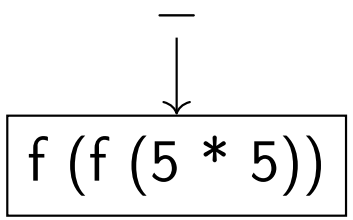
Длинный пример редуцирования

Возьмём f a = a + a.

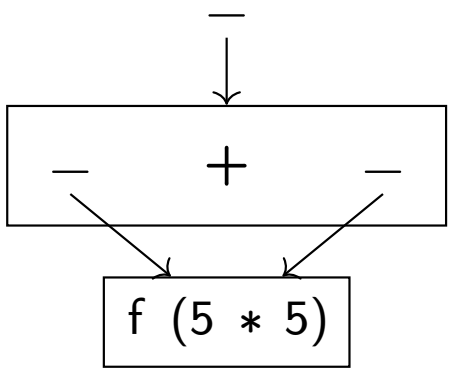
1. Вызовем **const** (f (f (5 * 5))) (f (f (f (f 0)))). Получаем ящик:



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

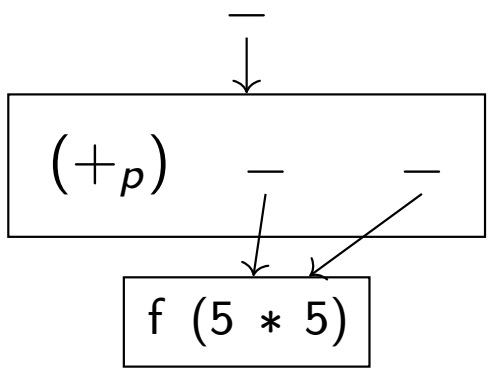


3. Раскрывая этот ящик по определению f, получим



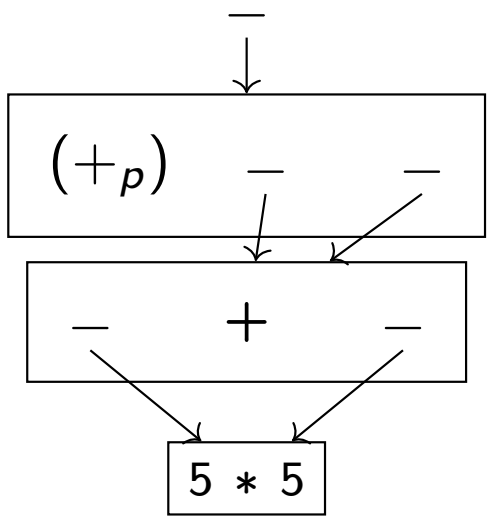
Длинный пример редуцирования

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.



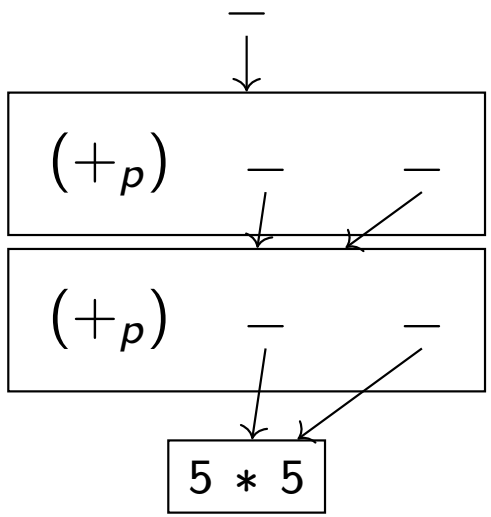
Длинный пример редуцирования

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



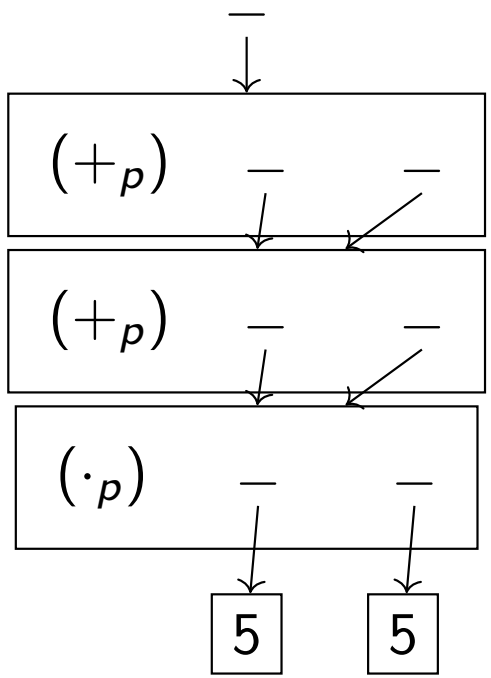
Длинный пример редуцирования

6. Снова раскроем $+$.



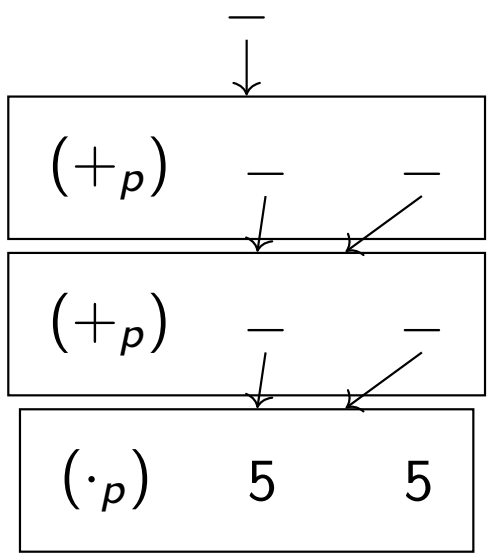
Длинный пример редуцирования

7. Раскроем $*$.



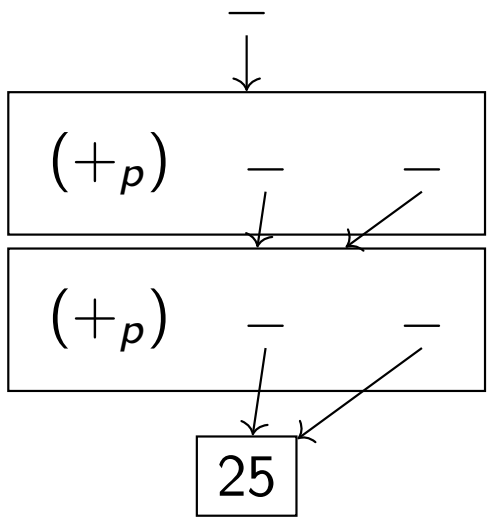
Длинный пример редуцирования

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



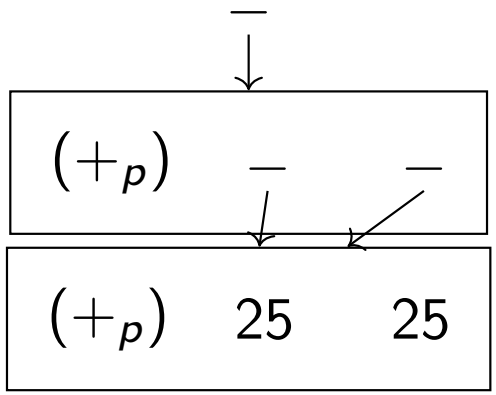
Длинный пример редуцирования

9. Вычислим.

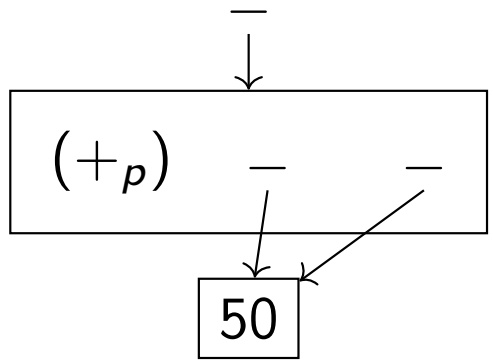


Длинный пример редуцирования

10. Подставим.

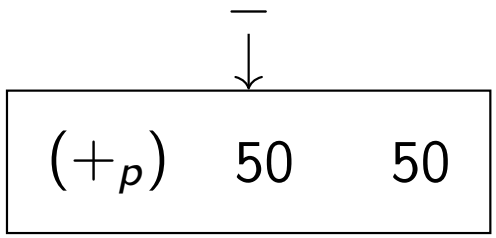


11. Вычислим.

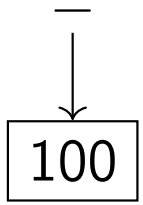


Длинный пример редуцирования

12. Подставим.



13. Вычислим.



14. Подставим!

100

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.