Несколько задая. Стратегии вычислений. Fix

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

20 сентября 2018 г.

Outline

Несколько задачек про прокрастинирующие вычисления

- Построить бесконечный список чисел Фибоначчи
  Подсказка:

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

- Построить список простых чисел (решето Эратосфена)
- Дан (бесконечный) список (бесконечных) списков. Сложить все элементы в один большой список. Ограничение: если элемент стоит в списке с конечным номером на позиции с конечным номером, то в результирующем списке он должен быть наблюдаем на конечной позиции.

Жесткая задача про прокрастинирующие вычисления

Есть бинарное дерево со значениями только в листях и типом

Жесткая задача про прокрастинирующие вычисления

Есть бинарное дерево со значениями только в листях и типом

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Дано дерево типа `Tree Int`. Нужно построить новое дерево такой же структуры, складывая в листья минимальный элемент исходного дерева

- за 2 прохода
- за 1 проход

Логика высказываний

Формулы бывают:

- Переменные: a, b, c ...
- Две логические константы
- Отрицания формул: $\neg a$, $\neg\neg b$ ...
- Конъюнкции двух формул: $a \wedge b$, $\neg b \wedge c$ ...
- Дизъюнкции двух формул: $a \vee b$, $\neg b \vee (c \vee d)$ ...
- Импликации двух формул: $a \supset b$, $\neg b \supset (c \supset d)$ ...

Логика высказываний

Формулы бывают:

- Переменные: a, b, c ...
- Две логические константы
- Отрицания формул: $\neg a$, $\neg\neg b$ ...
- Конъюнкции двух формул: $a \wedge b$, $\neg b \wedge c$ ...
- Дизъюнкции двух формул: $a \vee b$, $\neg b \vee (c \vee d)$ ...
- Импликации двух формул: $a \supset b$, $\neg b \supset (c \supset d)$ ...

Можно *интерпретировать* синтаксические конструкции выше, придав смысл переменным, отрицанию и бинарным операциям. Google: логика высказываний, булева алгебра, таблица истинности.

Упражнения про логику высказываний

- Опишите тип данных `data Formula ...=`
- Дана формула и окружение, которое связывает переменные. Упростите формулу пока упрощается
- Построить ДНФ по произвольной формуле
- Построить КНФ по произвольной формуле
- Дана формула с переменными. Проверить, что она истинная для любых значений переменных
  - Можно решать перебором
  - Можно придумать что-то хитрое

Outline

Ещё один мини-язык Λ с арифметикой

Выражения в мини-языке бывают:

- Константы: 1,2,3, и т.д.
- Именованные значения (“переменные”): a,b,x,y, и т.д.
- Оператор сложения $+$: $a + 1$, $x + y + x$, и т.д.
- Функции от одного аргумента: $(\lambda x \to x + 1)$, $(\lambda x \to \lambda y \to x + y)$, и т.д.
- Применение одного выражения к другому: $(\lambda x \to x + x)(2 + 3)$, $((\lambda x \to \lambda y \to x + y)2)(3 + 5)$, и т.д.

Ещё один мини-язык Λ с арифметикой

Выражения в мини-языке бывают:

- Константы: 1,2,3, и т.д.
- Именованные значения (“переменные”): a,b,x,y, и т.д.
- Оператор сложения $+$: $a + 1$, $x + y + x$, и т.д.
- Функции от одного аргумента: $(\lambda x \to x + 1)$, $(\lambda x \to \lambda y \to x + y)$, и т.д.
- Применение одного выражения к другому: $(\lambda x \to x + x)(2 + 3)$, $((\lambda x \to \lambda y \to x + y)2)(3 + 5)$, и т.д.

Упражнение: напишите алгебраический тип для языка Λ.

Ещё один мини-язык Λ с арифметикой

Выражения в мини-языке бывают:

- Константы: 1,2,3, и т.д.
- Именованные значения (“переменные”): a,b,x,y, и т.д.
- Оператор сложения $+$: $a + 1$, $x + y + x$, и т.д.
- Функции от одного аргумента: $(\lambda x \to x + 1)$, $(\lambda x \to \lambda y \to x + y)$, и т.д.
- Применение одного выражения к другому: $(\lambda x \to x + x)(2 + 3)$, $((\lambda x \to \lambda y \to x + y)2)(3 + 5)$, и т.д.

Упражнение: напишите алгебраический тип для языка Λ.

? Если мы хотим написать интерпретатор Λ, то как он должен работать?

Call-by-value

Также известно как *stict* или *eager*. Применяется в большинстве известных языков программирования.

Описывает как вычислять применение a к b.

- Слева направо: вычисляем a, затем b, затем применяем.
- Справа налево: b, a, затем применяем.

Замечание: если a посчитано до конца, то разница отсутствует.

Call-by-value: примеры

Слева направо:

- $((\lambda x \to \lambda y \to x + y)2)(3 + 5)$
- $(\lambda y \to 2 + y)(3 + 5)$
- $(\lambda y \to 2 + y)8$
- $2 + 8$
- 10

Справа налево:

- $((\lambda x \to \lambda y \to x + y)2)(3 + 5)$
- $((\lambda x \to \lambda y \to x + y)2)8$
- $(\lambda y \to 2 + y)8$
- $2 + 8$
- 10

В чистых языках не важно как вычислять: справо налево или слева направо, ответ одинаковый.

Call-by-name: пример

Не вычисляем аргменты функций. Когда они понадобятся мы подставляем аргумент как он есть туда. где он используется.

- $((\lambda x \to \lambda y \to x + y + y)2)(3 + 5)$
- $(\lambda y \to 2 + y + y)(3 + 5)$
- $2 + (3 + 5) + (3 + 5)$
- $2 + 8 + (3 + 5)$
- А теперь допустим, что $+$ вычисляется слева направо
- $10 + (3 + 5)$
- $10 + 8$
- 18

Call-by-name: пример и не из мини-языка, и не из Haskell

- $(\lambda x \to x; x)$ (print "hi")
- (print "hi"); (print "hi")
- (print "hi")
- ()

? Что напечатается?

Вычисления ... особого рода

Упражнение прямо на лекции: давайте посчитаем выражение тремя способами

$$(\lambda y \to yy)(\lambda x \to xx)$$

Вычисления ... особого рода

Упражнение прямо на лекции: давайте посчитаем выражение тремя способами

$$(\lambda y \to yy)(\lambda x \to xx)$$

Считаем, считаем, считаем... и получаем

$$(\lambda x \to xx)(\lambda x \to xx)$$

То же самое, с точностью до переименования
Считаем дальше и зависаем

Одно и то же, при трех стратегиях.

Всегда ли одинаковый ответ?

Положим $loop$ это $(\lambda x \to xx)(\lambda x \to xx)$ и рассмотрим

$$(\lambda x \to \lambda y \to yy)(loop)$$

Всегда ли одинаковый ответ?

Положим $loop$ это $(\lambda x \to xx)(\lambda x \to xx)$ и рассмотрим

$$(\lambda x \to \lambda y \to yy)(loop)$$

? Что будет, если посчитать с помощью CBV?

Всегда ли одинаковый ответ?

Положим $loop$ это $(\lambda x \to xx)(\lambda x \to xx)$ и рассмотрим

$$(\lambda x \to \lambda y \to yy)(loop)$$

? Что будет, если посчитать с помощью CBV?
? А если CBN?

Всегда ли одинаковый ответ?

Положим $loop$ это $(\lambda x \to xx)(\lambda x \to xx)$ и рассмотрим

$$(\lambda x \to \lambda y \to yy)(loop)$$

? Что будет, если посчитать с помощью CBV?
? А если CBN?

$$(\lambda y \to y)$$

CBN завершается, а CBV – нет

Так может CBV строго лучше CBN?

$$(\lambda y \to yy)(big)$$

Применяем call-by-name:

Так может CBV строго лучше CBN?

$$(\lambda y \to yy)(big)$$

Применяем call-by-name:

$$(big)(big)$$

Так может CBV строго лучше CBN?

$$(\lambda y \to yy)(big)$$

Применяем call-by-name:

$$(big)(big)$$

CBV считает аргумент один раз. CBN – два.

Вообще, если аргумент функции встречается больше одного раза, то CBN будет вычислять его больше одного раза. Нехорошо.

Так может CBV vs. CBN vs. call-by-need (lazy, прокрастинирующий)

Иногда CBN завершается, а CBV – нет.
Иногда CBV не вычисляет что-то, что CBN вычисляет.
Иногда CBN не вычисляет что-то 2,3 или больше раз, когда CBV только один.

Ленивые вычисления

- Если что-то посчитали – запоминаем результат на будущее. Избегает повторных вычислений
- Завершается, если CBN завершается
- Вычисляет аргументы максимум один раз

? Так может call-by-need – это то, что нужно всегда использовать?

call-by-need a.k.a. прокрастинирующие

- Создание отложенного вычисления типа T зачастую сводится к созданию функции `()-> T`.
- Требует некоторых усилий, использует некоторое количество памяти. На больших программах может стать важным

```
xs = [1,2,3,..... big list, ...]
n = foldl (+) 0 xs
```

call-by-need a.k.a. прокрастинирующие

- Создание отложенного вычисления типа T зачастую сводится к созданию функции `()-> T`.
- Требует некоторых усилий, использует некоторое количество памяти. На больших программах может стать важным

```
xs = [1,2,3,..... big list, ...]
n = foldl (+) 0 xs
```

Список `xs` занимает много памяти, но его нельзя объявить мусором, потому что элементы используются при вычислении `n`.

call-by-need a.k.a. прокрастинирующие

- Создание отложенного вычисления типа T зачастую сводится к созданию функции `()-> T`.
- Требует некоторых усилий, использует некоторое количество памяти. На больших программах может стать важным

```
xs = [1,2,3,..... big list, ...]
n = foldl (+) 0 xs
```

Список `xs` занимает много памяти, но его нельзя объявить мусором, потому что элементы используются при вычислении `n`.

Но существует функция `foldl'`!
Упражнение: попробуйте поскладывать большие списки чисел (`[1..1000000]`) с помощью foldl, foldr, foldl'. Когда лучше использовать `foldl` вместо `foldl'`?

Упражнение: вычисление выражения из языка Λ

Задачи такого рода были в том году на экзамене.

```
evalL:: [(String, L)] -> L -> Maybe L
```

Нужно действовать по аналогии с примером про вычислитель арифметики с операциями сложения и умножения. Из стратегий вычислений (CBV, CBN, прокрастинирующую) выберите какая нравится.

N.B. Там есть **грабли**.

Outline

Неподвижная точка

Рассмотрим $f : A \to A$.
x – это *неподвижная точка*, если $f(x) = x$.

Неподвижная точка в Haskell:

```
fix f = let {x = f x} in x
```

ну или

```
fix f = f (fix f)
```

Пример

```
fix :: (t -> t) -> t
fix f = f (fix f)

last :: [a] -> Maybe a
last [] = Nothing
last [x] = Just x
last (x:xs) = last xs
```

Пример

```
fix :: (t -> t) -> t
fix f = f (fix f)

last :: [a] -> Maybe a
last [] = Nothing
last [x] = Just x
last (x:xs) = last xs

last2 :: [a] -> Maybe a
last2 = fix g
  where
    g _ []     = Nothing
    g _ [x]    = Just x
    g r (_:xs) = r xs
```

Упражнения про fix для функций

Реализовать через `fix`:

- `map`
- Вычисление n-го числа Фибоначчи
- Для данного числа `n` посчитайте `cos (cos (cos ..... n)...)` пока результат (а именно `0.7390851332151607`) не перестанет изменяться.

Fixpoint для типов (1/2)

Почти связный список

```
data ListG a r = Nil | Cons a r
```

Что по сути описывает либо пустой список, либо ячейку памяти со значением типа `a` и “дыркой”.

```
type List2 a r = ListG a (ListG a r)
type List3 a r = ListG a (List2 a r)
type List4 a r = ListG a (List3 a r)
```

Fixpoint для типов (2/2)

```
type RingF_{n+1} a = RingF (RingF_{n+1} a)

newtype Fix f = Fix (f (Fix f))
Fix :: f (Fix f) -> Fix f

type List a = Fix (ListG a)
```

Теперь можно конструировать значения типа `List a`:

Fixpoint для типов (2/2)

```
type RingF_{n+1} a = RingF (RingF_{n+1} a)

newtype Fix f = Fix (f (Fix f))
Fix :: f (Fix f) -> Fix f

type List a = Fix (ListG a)
```

Теперь можно конструировать значения типа `List a`:

```
(Fix Nil) :: List a
(Fix $ Cons 1 $ Fix Nil) :: List Int
(Fix $ Cons 1 $ Fix $ Cons 2 $ Fix Nil) :: List Int
   :: List Int
```