Лямбда исчисление

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

24 октября 2019 г.

В этих слайдах

1. Исчисление
2. Стратегии вычислений

ACHTUNG!

Любых слайдов абсолютно недостаточно, чтобы разобраться в теме.

Даже если Вы напишите с нуля интерпретатор λ-исчисления, гарантировать полного понимания невозможно.

Читайте умные книжки, например [4].

Правила вывода в исчислении

Пусть дан некоторый язык L, с помощью которого записываются P_i и C.

$(n+1)$-местные правила вывода имеют форму

$$\frac{P_1 \qquad \dots \qquad P_n}{C}$$

- P_i – посылки (premises)
- C_i – заключение (conclusion)

По смыслу означает «если и P_1, и P_2, ..., и P_n, то C»

Исчисление

Состоит из

- непустого множества аксиом
- множества правил вывода

Определение

Аксиома – это правило вывода без посылок

Формальное определение можно прочитать в книжке Герасимова [5].

Пример исчисления. Дифференциальное исчисление

Языком L будет язык задания функций (который вообще-то надо формально определять, но не будем)

$$\frac{}{sin'(x) = cos(x)}\ \text{sin} \qquad \frac{}{cos'(x) = -sin(x)}\ \text{cos} \qquad \frac{}{x' = 1}\ \text{pow} \qquad \frac{}{c' = 0, \text{если } c \in N}\ \text{const}$$

$$\frac{f'(x) = u(x) \qquad g'(x) = v(x)}{((f \cdot g)(x))' = u(x) \cdot g(x) + f(x) \cdot v(x)}\ \text{mul}$$

$$\frac{f'(x) = u(x) \qquad g'(x) = v(x)}{(f(g(x)))' = u(g(x)) \cdot v(x)}\ \text{cmps}$$

Дифференциальное исчисление. Пример вывода

$$\dfrac{\dfrac{}{sin'(x) = cos(x)}\,\text{sin} \qquad \dfrac{\dfrac{}{cos'(x) = -sin(x)}\,\text{cos} \qquad \dfrac{\dfrac{}{2' = 0}\,\text{const} \quad \dfrac{}{x' = 1}\,\text{pow}}{(2 \cdot x)' = 0 \cdot x + 2 \cdot 1}}{cos'(2x) = -sin(2x) \cdot 2}}{(sin(x) \cdot cos(2x))' = cos(x) \cdot cos(2x) + sin(x) \cdot (-sin(2x) \cdot 2)}$$

На вывод можно смотреть как на **доказательство** того, что производная действительно посчитана правильно.

Результат вывода можно было бы упростить и дальше, но у нас недостаточно правил вывода для этого.

Язык Λ-выражений

В начале нужно выбрать язык $\mathcal{V}$ имен переменных. Традиционно используются два варианта:

- ✔ Непустая последовательность букв: `[a-z]([a-z])*`
- ☐ Натуральные числа

Из алфавита строятся слова(предложения) языка. Алфавитом для Λ будет $\{(,),\to,\lambda\} \cup \mathcal{V}$

Слова в языке Λ (*лямбда выражения* или *лямбда термы*) строятся по следующим грамматическим правилам

$$\begin{aligned} \langle \mathrm{expr} \rangle &::= \langle \mathrm{varname} \rangle \mid (\lambda \to \langle \mathrm{expr} \rangle) \mid (\langle \mathrm{expr} \rangle \langle \mathrm{expr} \rangle) \\ \langle \mathrm{varname} \rangle &::= \mathcal{V} \end{aligned}$$

Т.е. λ-выражение это или вхождение переменной, либо λ-*абстракция*, либо *применение* (аппликация).

В язык входят скобки, но они часто опускаются.

$$E_1 E_2 \ldots E_n \sim (\ldots(E_1 E_2) \ldots E_n)$$

Например $(\lambda x \to x)$ это функция `id` из Haskell. Первый x – *аргумент* λ-абстракции, второй x – *тело*.

Имена аргументов и переменных не несут существенного смысла, т.е.

$$(\lambda x \to x) \equiv (\lambda y \to y) \equiv (\lambda z \to z) \equiv (\lambda t \to t)$$

Синтаксис $A \equiv B$ означает, что A – это синоним B.

Подстановка

Функции можно применять к выражениям, например $(\lambda x \to x)y$ вычисляется путём замены в теле λ-абстракции аргумента абстракции на y, т.е.

$$(\lambda x \to x)y = [y/x]x = y$$

Запись $[A/x]B$ означает, что надо подставить A вместо всех вхождений x в B.

Также встречаются другие нотации:

$$[A/x]B = B[x \mapsto A] = [x \mapsto A]B = B[x := A]$$

Свободные и связанные вхождения переменных

Формально:

- Имя n свободно в λ-выражении n.
- Имя n свободно в $(\lambda x \to E)$, если имя $n \neq x$ и n свободно в E.
- Имя n свободно в λ-выражении (MN) если либо оно свободно в M, либо оно свободно в N.

Пример

$$(\lambda x \to xy)(\lambda y \to y)$$

N.B. В одном λ-выражении одно и то же имя может входить и свободно, и связанно.

N.B. В некотором смысле λ – это квантор.

Подстановки

Корректный пример:

$$\mathfrak{I}\mathfrak{I} = (\lambda x \to x)(\lambda x \to x) = (\lambda x \to x)(\lambda z \to z) =$$

$$= [(\lambda z \to z)/x]x = (\lambda z \to z) = \mathfrak{I}$$

Подстановки

Корректный пример:

$$\mathfrak{I}\mathfrak{I} = (\lambda x \to x)(\lambda x \to x) = (\lambda x \to x)(\lambda z \to z) =$$

$$= [(\lambda z \to z)/x]x = (\lambda z \to z) = \mathfrak{I}$$

Некорректный пример:

$$(\lambda x \to (\lambda y \to xy))y \neq (\lambda y \to yy)$$

А нужно делать так

$$(\lambda x \to (\lambda y \to xy))y = (\lambda x \to (\lambda t \to xt))y = (\lambda t \to yt)$$

Основная идея:

- Если у нас есть $(\lambda x \to e)E$, то мы заменяем все *свободные* вхождения x на E.
- Если при замене свободная переменная в E вдруг становится связанной, то мы переименовываем связанную переменную в e перед выполнением подстановки.

Пример:

$$(\lambda x \to (\lambda y \to (x(\lambda x \to xy))))y$$

В $(\lambda y \to (x(\lambda x \to xy)))$ только первый x может быть заменен. Но перед подстановкой необходимо переименовать y в теле на новое имя t.

$$[y/x](\lambda t \to (x(\lambda x \to xt))) = (\lambda t \to (y(\lambda x \to xt)))$$

Редексы (*REDucible EXpressions*)

Редекс – это подвыражение (подтерм) вида $((\lambda x \to e)e_2)$

Стратегия вычислений – способ, по которому мы выбираем какие редексы и в каком порядке будет упрощать (вычислять, β-редуцировать).

Будем обозначать как $e \longrightarrow_\beta e'$ β-редукцию терма e в e'.

Редекс находится *левее*, если его λ в записи левее.

Редекс считается *самый левый внешний(leftmost outermost)*, если он самый левый и не содержится ни в каком другом редексе.

Редекс считается *самый левый внутренний(leftmost innermost)*, если он самый левый и не содержит ни какого другого редекса.

Нормальные формы

У нас четыре возможности

- Редуцируем ли под абстракциями? (да/нет)
- Редуцируем ли аргументы перед подстановкой? (да/нет)

Редуцируем аргументы?	Редуцируем под абстракциями?	
	Да(strong)	Нет(weak)
Да(strict)	Normal form $E ::= \lambda x \to E \mid x E_1 \dots E_n$	Weak Normal form $E ::= \lambda x \to e \mid x E_1 \dots E_n$
Нет(lazy)	Head normal form $E ::= \lambda x \to E \mid x e_1 \dots e_n$	Weak head normal form $E ::= \lambda x \to e \mid x e_1 \dots e_n$

В таблице E_j – это выражение в соответствующей нормальной форме, а e_i – произвольный λ-терм.

Порядков редукции бывает много...[2]

1. Call-by-Name
2. Normal Order
3. Call-by-Value
4. Applicative Order
5. Hybrid Applicative Order
6. Head Spine Reduction
7. Hybrid Normal Order

И ещё есть оптимизации связанные с мемоизацией(кешированием) нормальных форм подвыражений.
Так Call-by-Name + кеширование = Call-by-Need (Haskell)

Call-By-Name $\rightarrow$ Weak Head Normal Form

Редуцирует **самый левый внешний** редекс, который **не под абстракцией**. Например, $(\lambda x \to (\lambda y \to M)N)$ уже в WHNF, потому что единственный редекс $(\lambda y \to M)N$ под абстракцией.

$$\frac{}{x \xrightarrow{cbn} x}\ \text{Var} \qquad \frac{}{(\lambda x \to e) \xrightarrow{cbn} (\lambda x \to e)}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{cbn} (\lambda x \to e) \qquad [e_2/x]e \xrightarrow{cbn} e'}{(e_1 e_2) \xrightarrow{cbn} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbn} e_1' \neq (\lambda x \to e)}{(e_1 e_2) \xrightarrow{cbn} (e_1' e_2)}\ \text{App-non-abs}$$

CBN может посчитать 1 аргумент несколько раз по сравнению с CBV.

Call-by-Value $\to$ Weak Normal Form

Редуцирует **самый левый внутренний** редекс, который **не под абстракцией**.
Например, в $(\lambda x \to (\lambda y \to U)V)((\lambda z \to M)N)$ самый левый внутренний – это $(\lambda y \to U)V$, но редуцироваться первым будет $((\lambda z \to M)N)$.

$$\frac{}{x \xrightarrow{cbv} x}\ \text{Var} \qquad \frac{}{(\lambda x \to e) \xrightarrow{cbv} (\lambda x \to e)}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{cbv} (\lambda x \to e) \qquad e_2 \xrightarrow{cbv} e_2' \qquad [e_2'/x]e \xrightarrow{cbv} e'}{(e_1 e_2) \xrightarrow{cbv} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbv} e_1' \neq (\lambda x \to e) \qquad e_2 \xrightarrow{cbv} e_2'}{(e_1 e_2) \xrightarrow{cbv} (e_1' e_2')}\ \text{App-non-abs}$$

Стандарт для большинства языков программирования.

Нормальной формы может не быть!

Определение

Нормализация – процесс поиска соответствующей нормальной формы с помощью применения β-редукции согласно соответствующей стратегии

Пример: комбинатор $\Omega = (\lambda x \to xx)(\lambda x \to xx)$

$$(\lambda x \to xx)(\lambda x \to xx) \xrightarrow{cbv} [(\lambda x \to xx)/x](xx) \xrightarrow{cbv} (\lambda x \to xx)(\lambda x \to xx) \xrightarrow{cbv} \dots$$

CBN vs. CBV

Call-by-Name чаще завершается

$$(\lambda xy \to y)\Omega \xrightarrow{cbv} \text{расходится}$$

$$(\lambda xy \to y)\Omega \xrightarrow{cbn} (\lambda y \to y)$$

Но Call-by-Name иногда вычисляет аргументы больше одного раза

$$(\lambda x \to (Ax)(Bx))((\lambda y \to y)C) \xrightarrow{cbn} (A((\lambda y \to y)C))\ (B((\lambda y \to y)C))$$

$$(\lambda x \to (Ax)(Bx))((\lambda y \to y)C) \xrightarrow{cbv} (\lambda x \to (Ax)(Bx))C \xrightarrow{cbv} (AC)(BC)$$

Applicative Order $\to$ Normal Form

Редуцирует **самый левый внутренний** редекс, и **под абстракцией тоже**. Например, в $(\lambda x \to (\lambda y \to U)V)((\lambda z \to M)N)$ самый левый внутренний – это $(\lambda y \to U)V$.

$$\frac{}{x \xrightarrow{ao} x}\ \text{Var} \qquad \frac{e \xrightarrow{ao} e'}{(\lambda x \to e) \xrightarrow{ao} (\lambda x \to e')}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{ao} (\lambda x \to e) \qquad e_2 \xrightarrow{ao} e_2' \qquad [e_2'/x]e \xrightarrow{ao} e'}{(e_1 e_2) \xrightarrow{ao} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{ao} e' \neq (\lambda x \to e) \qquad e_2 \xrightarrow{ao} e_2'}{(e_1 e_2) \xrightarrow{ao} (e_1' e_2')}\ \text{App-non-abs}$$

N.B. Аппликативный порядок совершает больше редукций и выдает более простой ответ по сравнению с CBV, но не гарантирует, что редукция завершится.

"Другой" Applicative Order

Иногда (в википедии или книге SICP [7]) аппликативным порядком называют Call-By-Value, где явно упорядочивают порядок вычисления фактических аргументов у λ-абстракций.

Согласно[2] это аппликативным порядком не является.

Короче говоря, стратегии с редукцией под абстракциями (applicative order, normal order) в программировании не используются.

Normal Order $\to$ Normal Form

Сначала редуцирует **самый левый внешний** редекс. Встретив применение $(e_1 e_2)$ вначале пытается редуцировать e_1 как CBN. Если не получилась абстракция – принимается за аргументы.

$$\frac{}{x \xrightarrow{nor} x}\ \text{Var} \qquad \frac{e \xrightarrow{nor} e'}{(\lambda x \to e) \xrightarrow{nor} (\lambda x \to e')}\ \text{Abs}$$

$$\frac{e_1 \xrightarrow{cbn} (\lambda x \to e) \qquad [e_2/x]e \xrightarrow{nor} e'}{(e_1 e_2) \xrightarrow{nor} e'}\ \text{App-abs}$$

$$\frac{e_1 \xrightarrow{cbn} e' \neq (\lambda x \to e) \qquad e_1' \xrightarrow{nor} e_1'' \qquad e_2 \xrightarrow{nor} e_2'}{(e_1 e_2) \xrightarrow{ao} (e_1'' e_2')}\ \text{App-non-abs}$$

N.B. Нормальный порядок сочетает две стратегии, позволяет получить более простые результаты, чем CBN. Чаще завершается, чем AO.

Нумералы А.Чёрча

$0 \sim (\lambda sx \to x)$
$1 \sim (\lambda sx \to sx)$
$2 \sim (\lambda sx \to s(sx))$
и т.д.

Функция successor (следующее число): $S \equiv (\lambda wyx \to y(wyx))$

$$S0 \equiv (\lambda wyx \to y(wyx))(\lambda fx \to x) \xrightarrow{ao} (\lambda yx \to y((\lambda fz \to z)yx)) \xrightarrow{ao}$$
$$\xrightarrow{ao} (\lambda yx \to y((\lambda z \to z)x)) \xrightarrow{ao} (\lambda yx \to yx) \equiv 1$$

Сложение m и n: m раз применить функцию S к n

$$2S3 \equiv \big(\lambda sz \to s(sz)\big)\big(\lambda wyx \to y(wyx)\big)\big(\lambda uv \to u(u(uv))\big) \xrightarrow{ao}$$
$$\xrightarrow{ao} \big(\lambda wyx \to y(wyx)\big)\big((\lambda wyx \to y(wyx))(\lambda uv \to u(u(uv)))\big) \equiv SS3$$

If-then-else

Логические операции

$$T \equiv (\lambda xy \to x) \qquad\qquad F \equiv (\lambda xy \to y)$$

$$\neg \equiv \big(\lambda x \to x(\lambda uv \to v)(\lambda ab \to a)\big) \equiv (\lambda x \to xFT)$$

Пример:

$$\forall f, a : \quad 0fa \equiv (\lambda sz \to z)fa = a$$

$$\forall a : \quad Fa \equiv (\lambda xy \to y)a = (\lambda y \to y) \equiv I$$

If-then-else с условием равенства нулю

$$Z \equiv (\lambda x \to xF\neg F)$$

Для $N = 0 : \; Z\,0 \equiv (\lambda x \to xF\neg F)0 = 0F\neg F = \neg F = T$

Для $N \neq 0 : \; ZN \equiv (\lambda x \to xF\neg F)N = NF\neg F = IF = F$

А ещё...

Можно описать умножение, ''предыдущее число'', вычитание.
Упражнение: нагуглите и разберитесь как они работают.

Мы не можем в Λ-языке ссылаться сами на себя явно, можем ли мы писать рекурсивные программы?

Мы не можем в Λ-языке ссылаться сами на себя явно, можем ли мы писать рекурсивные программы?

Таки да! Вот Y-комбинатор, великий и ужасный ©

$$Y \equiv \big(\lambda f \to (\lambda x \to f(xx))(\lambda x \to f(xx))\big)$$

Основное свойство

$$YR = (\lambda x \to R(xx))(\lambda x \to R(xx)) \xrightarrow{nor} R\big((\lambda x \to R(xx))(\lambda x \to R(xx))\big) = R(YR)$$

Для чисел Чёрча таким образом можно написать, например, факториал [1].

Операционные семантики big-step & small-step

Операционная семантика показывает как программа исполняется.

Big-step (операционная семантика большого шага)

- Вычисляет подвыражение программы за один (большой) шаг
- Смотря на неё вполне понятно, как написать интерпретатор
- Для некоторых случаев (например, программа с параллелизмом) в ней нельзя описать семантику, так как шаг слишком большой

Small-step (операционная семантика малого шага)

- Вычисляет подвыражение до какой-то степени (например, 1 редукцию) и возвращает наверх результат
- Относительно просто понять, что происходит
- Реализация работает медленнее, чем big-step семантика

Ещё бывает денотационная семантика, которая пытается придать программе смысл в виде математического отображения.

Проблема останова (1/2)

Вопрос: можем ли мы написать алгоритм, который будет брать на вход произвольную λ-абстракцию и аргумент, и говорить посчитается ли для них нормальная форма.

Положим наши программы либо зависают, либо выдают значение `true`.

Положим существуюет гипотетеическая $Halting(P, w)$, которая всегда завершается, и возвращает `true`, если (Pw) редуцируется в `true`, иначе $Halting(P, w)$ возвращает `false`.

Покажем от противного, что $Halting$ не может существовать.

Проблема останова (2/2)

Вопрос: во что отредуцируется E, в `true` или в `false` ?

$$E = Halting\big(\lambda m \to not(Halting(m,m)),\quad \lambda m \to not(Halting(m,m))\big)$$

Если E редуцируется в `true`, то применим функцию $\lambda m \to not(Halting(m,m))$ к аргументу $\lambda m \to not(Halting(m,m))$ и получим

$$not(Halting\big(\lambda m \to not(Halting(m,m)),\quad \lambda m \to not(Halting(m,m))\big)) = \neg E$$

что является отрицание истинного факта выше.

Если E редуцируется в `false`, то это означает, $Halting$ иногда зависает, что противоречит определению функции $Halting$. □

Безымянное представление через индексы де Брёйна (de Bruijn)

Идея

- Заводим глобальный контекст Γ, где взаимно однозначно сопоставляем каждому натуральному числу имея переменной.
- Связанные переменные представляем числом $k > 0$. Оно означает, что переменная связывается k-й охватывающей лямбдой.
- Свободная переменная x представляются в виде суммы Γx и глубины её местоположения внутри терма в λ абстракциях.

Пример: $\Gamma = \{b \mapsto 0, a \mapsto 1, z \mapsto 2, y \mapsto 3, x \mapsto 4\}$

- $x(yz) \equiv 4(3\ 2)$
- $(\lambda w \to yw) \equiv (\lambda \to 4\ 0)$
- $(\lambda w \to yx) \equiv (\lambda \to \lambda \to 6)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (1/2)

Пример: $[1 \mapsto s](\lambda \to 2) = [x \mapsto s](\lambda y \to x)$

Когда s проникнет под абстракцию, то надо будет "сдвинуть" некоторые индексы переменных, но не все, например, если $s = 2(\lambda \to 0)$ (т.е. $s = z(\lambda w \to w)$), то надо сдвинуть 2, а не 0.

Определение

Сдвиг терма t на d позиций с отсечкой c (обозначается $\uparrow_c^d (t)$)

- $\uparrow_c^d (k) = \begin{cases} k, & \text{если } k < c \\ k + d, & \text{если } k \geq c \end{cases}$
- $\uparrow_c^d (\lambda \to t_1) = \lambda \to \uparrow_{1+c}^d (t_1)$
- $\uparrow_c^d (t_1 t_2) = \uparrow_c^d (t_1) \uparrow_c^d (t_2)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (2/2)

Определение

Подстановка терма s вместо переменной номер j (обозначается $[j \mapsto s]t$)

- $[j \mapsto s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{в противном случае} \end{cases}$
- $[j \mapsto s](\lambda \to t_1) = (\lambda \to [(j+1) \mapsto \uparrow^1_0 s]t_1)$
- $[j \mapsto s](t_1 t_2) = ([j \mapsto s]t_1 [j \mapsto s]t_2)$

Упражнения на подстановку с индексами де Брёйна

- $[b \mapsto a](b(\lambda x \to \lambda y \to b))$
- $[b \mapsto a(\lambda z \to a)](b(\lambda x \to b))$
- $[b \mapsto a](\lambda b \to b\ a)$
- $[b \mapsto a](\lambda a \to b\ a)$

Ссылки I

- Демки на Haskell
  Gitlab repo
- Demonstrating Lambda Calculus Reduction
  Peter Sestoft
  PDF
- Типы в языках программирования. 1й том.
  Бенджамин Пирс
- Lambda-Calculus and Combinators, an Introduction
  J. ROGER HINDLEY & JONATHAN P. SELDIN
  PDF
- Курс математической логики и теории вычислимости
  Герасимов А.С.
  PDF

Ссылки II

A Tutorial Introduction to the Lambda Calculus
Raúl Rojas
PDF

Structure and Interpretation of Computer Programs
Abelson, Harold and Sussman, Gerald Jay and with Julie Sussman
PDF