БЕЛЕЖКИ ПО λ-СМЯТАНЕ И ТЕОРИЯ НА ДОКАЗАТЕЛСТВАТА

ТРИФОН ТРИФОНОВ

Увод

Този курс се състои от две части: λ-смятане и теория на доказателствата. В първата част на курса ще бъдат разгледани основите на λ-смятането като модел на функционални изчисления, ще бъдат разгледани свойствата на термовата редукция и ще бъде въведено понятието за синтактичен тип. Втората част ще е посветена на една от четирите "колони" на математическата логика: теорията на доказателствата. Ще бъде разгледана логическата страна на понятието формално доказателство и ще бъдат разгледани няколко системи за изразяване на формални доказателства. Ще бъде обърнато специално внимание на системи за интерактивно построяване и обработка на доказателства и ролята на доказателствата като "сертификати за коректност" на компютърни програми. Ще бъде показана тясната връзка между системите за изразяване на доказателства и типовото λ-смятане. Ще бъдат разгледани по-сложни типови системи и някои от техните приложения в компютърната лингвистика и в извличане на функционални програми от доказателства.

Ще считаме, че читателят се е срещал със следните понятия и идеи:

- частична и тотална функция (теоретико-множествена дефиниция)
- машина на Тюринг
- дефиниция на изчислима по Тюринг функция
- кодиране на машини на Тюринг като числа, изброимост на изчислимите функции
- неизброимост на функциите над естествени числа
- стоп-проблем на Тюринг
- други примери за неизчислими функции:
  - проверка за тоталност/недефинираност
  - проверка за теорема/тавтология
  - теорема на Райс
- разрешими и неразрешими множества
- полуразрешими множества
- необходимост от разглеждане не само на тотални, но и на частични изчислими функции (несъществуване на универсална тотална изчислима функция)
- примитивно рекурсивни функции + оператор за неограничено търсене (минимизация, μ-оператор) = частично-рекурсивни функции
- разликата между разрешимост и конструктивност (има ли в π поне n поредни седмици)

Дата: 8 април 2016 г.

- неконструктивни доказателства (парадоксът за пияниците, $a, b \in \mathbb{R} \setminus \mathbb{Q}, a^b \in \mathbb{Q}$)
- теорема на Кнастер-Тарски за неподвижната точка

Нотации за равенство

В изложението ще използваме няколко различни нотации за равенство, описани по-долу:

Ще използваме	за да изразим, че
$A := B$	синтактичният обект A се дефинира чрез израза B
$A \equiv B$	A и B съвпадат като синтактични обекти
$A = B$	A и B са равни в дадена логическа система
$A \overset{\alpha}{=} B$	термът A се получава от терма B чрез преименуване на свързаните променливи
$A \overset{\beta}{=} B$	термовете A и B имат общ β-редукт
$A \overset{\eta}{=} B$	термовете A и B имат общ η-редукт
$A \overset{\beta\eta}{=} B$	термовете A и B имат общ $\beta\eta$-редукт

1. Безтипово λ-смятане

λ-смятането е теория на математическите функции, разглеждаща ги като *синтактични правила* вместо от теоретико-множествена гледна точка (като множества от двойки). Създадено от Alonzo Church през 1932–33 г. Замислено е като фундаментална математическа теория на функциите, но е доказано логически противоречива от Stephen Kleene и John Rosser през 1935 г.

През 1936 г. Church се концентрира върху изчислителния аспект на λ-смятането. В същата година Kleene показва, че функциите, определими чрез λ-смятане са точно общите рекурсивни функции на Jacques Herbrand и Kurt Gödel, а самият Alan Turing през 1936–37 г. показва, че λ-определимите функции съвпадат с тези, изчислими с машина на Тюринг. Така се оказва, че три независими опита да бъдат формализирани "ефективно" изчислимите функции водят до еквивалентни модели. Обобщението на тези наблюдения е тезисът на Church-Turing, който твърди, че интуитивното понятие за автоматично изчислима функция съвпада с формалната математическа дефиниция на изчислима функция (чрез която и да е от еквивалентните формулировки на това понятие).

λ-смятането може да се разглежда като концептуален език за програмиране и като такъв дава основите на функционалния стил на програмиране.

1.1. Синтаксис на λ-смятането.

Нека считаме, че разполагаме с изброим набор от синтактични променливи V, които означаваме с малки латински букви $x, y, z, \ldots$. Азбуката на λ-смятането се състои от всички променливи, символа λ и скоби. λ-термовете са синтактични обекти, които представляват думи в азбуката на λ-смятането построени по три проста правила, зададени по-долу.

Дефиниция 1.1 (λ-термове)**.** Дефинираме индуктивно множеството от λ-термове Λ:

(1) Ако $x \in V$, то $x \in \Lambda$ (*променлива*).
(2) Ако $M, N \in \Lambda$, то $(MN) \in \Lambda$ (*апликация* или *прилагане* на терма M към терма N).

(3) Ако $x \in V$ и $M \in \Lambda$, то $(\lambda_x M) \in \Lambda$ (*абстракция* или *построяване* на функция с аргумент x и тяло M).

Идеята зад апликацията е да моделира прилагане на функцията M над аргумент N. Идеята на абстракцията е да моделира генериране на функция от израза M чрез абстрахиране на променливата x като функционален аргумент.

Бележка 1.1 (Нотация). Ще записваме $(\dots((M_1M_2)M_3)\dots M_n)$ съкратено като $M_1 \dots M_n$ или като $\vec{M}$, т.е. ще считаме, че апликацията е лявоасоциативна операция. Ще записваме $(\lambda_{x_1}(\lambda_{x_2}(\lambda_{x_3}\dots(\lambda_{x_n}M)\dots)))$ накратко като $\lambda_{x_1,x_2,\dots,x_n}M$ или просто $\lambda_{\vec{x}}M$. Ще пропускаме най-външните скоби. Ще бележим $M \equiv N$, в случай, че два терма съвпадат синтактично, т.е. като поредица от символи.

Пример 1.1. Няколко примера за λ-термове:

(1) $I := \lambda_x x$ — идентитет
(2) $\lambda_x y$ — "константна" функция
(3) $\lambda_{x,y}x$, $\lambda_{x,y}y$ — лява и дясна проекция
(4) $\lambda_{x,y}yx$ — функция, който прилага втория си аргумент над първия
(5) $x(\lambda_y x(yz)(\lambda_t xyt))$

Дефиниция 1.2 (Свободни променливи)**.** Нека $M \in \Lambda$, дефинираме множеството $\mathsf{FV}(M) \subseteq V$ от свободни променливи на M по индукция по построението на M:

(1) $\mathsf{FV}(x) := \{x\}$
(2) $\mathsf{FV}(M_1M_2) := \mathsf{FV}(M_1) \cup \mathsf{FV}(M_2)$
(3) $\mathsf{FV}(\lambda_x N) := \mathsf{FV}(N) \setminus \{x\}$

Дефиниция 1.3 (Затворени термове, комбинатори)**.** Терма $M \in \Lambda$ наричаме *затворен* или *комбинатор*, ако $\mathsf{FV}(M) = \emptyset$.

Дефиниция 1.4 (Свързани променливи)**.** Нека $M \in \Lambda$, дефинираме множеството $\mathsf{BV}(M) \subseteq V$ от свързани променливи на M по индукция по построението на M:

(1) $\mathsf{BV}(x) := \emptyset$
(2) $\mathsf{BV}(M_1M_2) := \mathsf{BV}(M_1) \cup \mathsf{BV}(M_2)$
(3) $\mathsf{BV}(\lambda_x N) := \mathsf{BV}(N) \cup \{x\}$

Пример 1.2. Нека $M := \lambda_{x,y}xyz$, $N := x(\lambda_x yx)z$. Тогава $\mathsf{FV}(M) = \{z\}$, $\mathsf{BV}(M) = \{x, y\}$, $\mathsf{FV}(N) = \{x, y, z\}$, $\mathsf{BV}(N) = \{x\}$.

Основна синтактична трансформация в λ-смятането е заместването на свободна променлива с друг λ-терм.

Дефиниция 1.5 (Субституция)**.** Нека $M, N \in \Lambda$, $x \in V$. Дефинираме индуктивно по построението на M субституцията на x с N в M, която ще отбелязваме с $M[x \mapsto N]$.

(1) $x[x \mapsto N] := N$
(2) $y[x \mapsto N] := y$ за $y \not\equiv x$
(3) $(M_1M_2)[x \mapsto N] := (M_1[x \mapsto N])(M_2[x \mapsto N])$
(4) $\lambda_x P[x \mapsto N] := \lambda_x P$
(5) $(\lambda_y P)[x \mapsto N] := \lambda_y(P[x \mapsto N])$ за $y \not\equiv x$ и $x \notin P$ или $y \notin \mathsf{FV}(N)$

Важно е да обърнем внимание, че поради допълнителното условие $y \notin \mathsf{FV}(N)$ в последния случай от горната дефиниция, операцията субституция не винаги е дефинирана. Така например, не сме дефинирали кой терм отговаря на прилагането на субституцията $[x \mapsto y]$ към терма $\lambda_y x$. Ако не бяхме включили условието свързаната променлива да не е сред свободните променливи на терма, с който заместваме, то бихме получили като резултат $\lambda_x x$. Интуитивно, това не би било коректно, понеже така бихме успели чрез синтактична операция да променим качествено "константната" функция, представяна от терма $\lambda_y x$ на функцията "идентитет", представяна от терма $\lambda_x x$. Подобна аномалия обикновено се нарича "прихващане на променливи" и не е феномен присъщ само за λ-смятането. Прихващането на променливи се среща при други езици с квантори, какъвто е например езикът на предикатното смятане от първи ред. Дефиницията избягва проблематичните случаи за сметка на тоталността на операцията субституция, т.е. възможността произволна субституция да е винаги приложима над произволен терм.

Технически, това решение на проблема не е оптимално, тъй като би наложило да правим проверка за дефинираност при всяко прилагане на субституция. Затова често се прилага друга стратегия, известна като "използване на свежи свързани променливи". Тя се базира на интуитивното разбиране, че конкретното име на свързана променлива не е съществено за семантиката на терма. Така например, можем да си мислим, че всички термове от вида $\lambda_x x, \lambda_y y, \lambda_z z \ldots$ представят една и съща по смисъл функция, а именно идентитет. Формално може да се дефинира релацията на еквивалентност $\overset{\alpha}{=}$, която е в сила за термове, които се различават синтактично само по имената на свързаните си променливи.

Задача 1.1. ***(5 т.)*** *Да се дефинира формално с индукция операция "преименуване на свързана променлива", която по даден терм $M \in \Lambda$, променлива $x \in \mathsf{BV}(M)$ и променлива $y \notin \mathsf{FV}(M) \cup \mathsf{BV}(M)$, дефинира нов терм M_y^x, който представлява резултата от заменянето на всички* свързани *срещания на x в M с y.*

Дефиниция 1.6 (Вариант)**.** Ако $M_y^x \equiv N$ за някои $x, y \in V$, то казваме че N е *вариант* на M.

Дефинираме релацията $\overset{\alpha}{=}$ като рефлексивно, симетрично и транзитивно затваряне на операцията "е вариант на". Формално:

(1) $M \overset{\alpha}{=} M$.
(2) Ако $M_y^x \equiv N$ за някои $x, y \in V$, то $M \overset{\alpha}{=} N$.
(3) Ако $M \overset{\alpha}{=} N$, то $N \overset{\alpha}{=} M$.
(4) Ако $M \overset{\alpha}{=} N$ и $N \overset{\alpha}{=} P$, то $M \overset{\alpha}{=} P$.

Очевидно дефинираната релация е релация на еквивалентност. Лесно се вижда, че с преименуване на свързаните променливи винаги можем да достигнем до терм, в който множествата на свързаните и свободните променливи нямат сечение.

Твърдение 1.1. *Нека $M \in \Lambda$. Можем да намерим $M' \overset{\alpha}{=} M$, така че $\mathsf{BV}(M) \cap \mathsf{FV}(M) = \emptyset$.*

Доказателство. Нека $\mathsf{BV}(M) = \{x_1, \ldots, x_n\}$. Нека си изберем крайно множество от променливи $\{y_1, \ldots, y_n\} \subseteq V \setminus (FV(M) \cup \mathsf{BV}(M))$. Това е възможно

тъй като $\mathsf{FV}(M)$ и $\mathsf{BV}(M)$ са крайни, а V е безкрайно. Дефинираме $M' := (\ldots(M^{x_1}_{y_1})^{x_2}_{y_2}\ldots)^{x_n}_{y_n}$. □

По подобен начин може да се провери, че с подходящо преименуване можем да осигурим, че операцията субституция ще бъде дефинирана.

Задача 1.2. ***(8 т.)*** *Да се покаже, че операцията субституция може да се разглежда като тотална с точност до релацията $\overset{\alpha}{=}$. Формално, нека считаме че са дадени произволни $x \in V$ и $N \in \Lambda$. Да се покажат следните две свойства:*

(1) *За всяко $M \in \Lambda$, съществува $M' \overset{\alpha}{=} M$, така че $M'[x \mapsto N]$ е дефинирана.*
(2) *Ако $M \overset{\alpha}{=} M' \in \Lambda$, $N \overset{\alpha}{=} N' \in \Lambda$ и $M[x \mapsto N]$ и $M'[x \mapsto N']$ са дефинирани едновременно, то $M[x \mapsto N] \overset{\alpha}{=} M'[x \mapsto N']$.*

Формално можем да разсъждаваме по следния начин. Нека разгледаме фактормножеството $\Lambda_{/\overset{\alpha}{=}}$, т.е. вместо отделни λ-термове разглеждаме класове на еквивалентност от λ-термове относно релацията $\overset{\alpha}{=}$. Съгласно Задача 1.2 за такива класове на еквивалентност операцията субституция е коректно дефинирана и тотална.

Бележка 1.2. Curry предлага решение на проблема с прихващането на свободни променливи, при което преименуването на свързани променливи е част от дефиницията на субституция. В настоящото разглеждане правим решението на две стъпки, за да може по-ясно да се види нуждата от такова преименуване.

Това решение на проблема с прихващането на променливи е "отдолу-нагоре", т.е. дефинираме частична операция за субституция и показваме как тя може да бъде превърната в тотална. Алтернативен подход би бил "отгоре-надолу", т.е. разглеждането на тотална операция субституция и ограничаването ѝ с допълнително условие за коректност.

Задача 1.3. ***(8 т.)*** *Да разгледаме алтернативна дефиниция на операцията субституция $M[x \rightsquigarrow N]$, която е дефинирана по същия начин като $M[x \mapsto N]$, с изключение на условието (5), което е променено по следния начин:*

(5) $(\lambda_y P)[x \rightsquigarrow N] := \lambda_y(P[x \rightsquigarrow N])$ *за* $y \not\equiv x$

Казваме, че операцията субституция е коректна, *ако $\mathsf{FV}(N) \cap \mathsf{BV}(M) = \emptyset$. Да се покаже, че:*

(1) *ако $M[x \rightsquigarrow N]$ е коректна, то $M[x \mapsto N]$ е дефинирана $M[x \mapsto N] \equiv M[x \rightsquigarrow N]$;*
(2) *има случай, в който $M[x \mapsto N]$ е дефинирана, но $M[x \rightsquigarrow N]$ не е коректна;*
(3) *за всяко $M \in \Lambda$ можем да намерим $M' \overset{\alpha}{=} M$, така че $M'[x \rightsquigarrow N]$ да е коректна.*

След като се уверихме, че има коректно и формално решение за избягване на прихващането на променливи, за удобство ще въведем следните две конвенции, предложени от Barendregt:

(1) Отъждествяваме синтактично термове, които са α-еквивалентни, т.е. считаме че релацията $\overset{\alpha}{=}$ е "вградена" в синтактичното равенство $\equiv$.

(2) Ако в даден момент разглеждаме някакво множество от термове, ще считаме че сме избрали такива представители, че нито една от свързаните променливи няма да съвпада с нито една от свободните.

Задача 1.4. ***(5 т.)*** *Да се направи програмна реализация на операцията субституция, която при нужда преименува свързаните променливи по подходящ начин при прилагане, за да осигури коректност.*

Интуитивно е ясно, че субституцията оказва влияние само върху свободните променливи на даден терм. Формално, това може да бъде изразено чрез следната Лема.

Лема 1.1. *Нека $M, N \in \Lambda$ и $x \notin \mathsf{FV}(M)$. Тогава $M[x \mapsto N] \equiv M$.*

Доказателство. Доказателството провеждаме с индукция относно дефиницията на M.

(1) Нека $M \equiv y \in V$. Тъй като $x \notin \mathsf{FV}(M) = \{y\}$, то $x \not\equiv y$. Тогава по дефиниция $M[x \mapsto N] \equiv M$.
(2) Нека $M \equiv (M_1 M_2)$. Тъй като $x \notin \mathsf{FV}(M) = \mathsf{FV}(M_1) \cup \mathsf{FV}(M_2)$, то по индукционно предположение имаме, че $M_i[x \mapsto N] \equiv M_i$ за $i = 1, 2$. Тогава
$$(M_1 M_2)[x \mapsto N] \equiv (M_1[x \mapsto N])(M_2[x \mapsto N]) \equiv M_1 M_2.$$
(3) Нека $M \equiv \lambda_y M'$ и $y \notin \mathsf{FV}(N)$. Ако $y \equiv x$, имаме твърдението по дефиниция. Нека сега $y \not\equiv x$. Тъй като $\mathsf{FV}(M) = \mathsf{FV}(M') \setminus \{y\}$, то $x \notin \mathsf{FV}(M')$ и следователно
$$(\lambda_y M')[x \mapsto N] \equiv \lambda_y M'[x \mapsto N] \equiv \lambda_y M'.$$

□

Лесно може да се види, че не е задължително две субституции да комутират, т.е. че има значение в какъв ред ще бъдат изпълнени те.

Задача 1.5. ***(2 т.)*** *Да се покажат $M, N, P \in \Lambda$ и $x, y \in V$ такива, че $M[x \mapsto N][y \mapsto P] \not\equiv M[y \mapsto P][x \mapsto N]$.*

При определени условия, все пак редът на субституциите може да бъде обърнат в някакъв смисъл. Тъй като понятието за субституция лежи в основата на дефиниране на изчислителния аспект на λ-смятането, възможността за обръщане на реда на субституциите ще ни позволи по-късно да правим разсъждения за последователността, в която функциите в даден λ-терм се оценяват.

Лема 1.2 (за субституцията)**.** *Нека $M, N, P \in \Lambda$, а $x \not\equiv y \in V$ и $x \notin \mathsf{FV}(P)$. Тогава $M[x \mapsto N][y \mapsto P] \equiv M[y \mapsto P][x \mapsto N[y \mapsto P]]$.*

Доказателство. Провеждаме индукция по дефиницията на M.

(1) Нека $M \equiv x$. Тогава по дефиниция имаме:
$$x[x \mapsto N][y \mapsto P] \equiv N[y \mapsto P] \equiv x[y \mapsto P][x \mapsto N[y \mapsto P]].$$
(2) Нека $M \equiv y$. Тъй като $x \not\equiv y$, по дефиниция имаме:
$$y[x \mapsto N][y \mapsto P] \equiv y[y \mapsto P] \equiv P,$$
$$y[y \mapsto P][x \mapsto N[y \mapsto P]] \equiv P[x \mapsto N[y \mapsto P]].$$
Но тъй като по условие $x \notin \mathsf{FV}(P)$, от Лема 1.1 получаваме, че $P[x \mapsto N[y \mapsto P]] \equiv P$.

(3) Нека $M \equiv z$, така че $x \not\equiv z \not\equiv y$. Тогава по дефиниция имаме:

$$z[x \mapsto N][y \mapsto P] \equiv z \equiv z[y \mapsto P][x \mapsto N[y \mapsto P]].$$

(4) Нека $M \equiv (M_1 M_2)$. По индукционно предположение (ИП) Лемата е вярна за M_1 и M_2. Тогава:

$$\begin{array}{lll} (M_1M_2)[x \mapsto N][y \mapsto P] & \overset{\text{(по дефиниция)}}{\equiv} & (M_1[x \mapsto N][y \mapsto P]) \\ & & (M_2[x \mapsto N][y \mapsto P]) \\ & \overset{\text{(по ИП)}}{\equiv} & (M_1[y \mapsto P][x \mapsto N[y \mapsto P]]) \\ & & (M_2[y \mapsto P][x \mapsto N[y \mapsto P]]) \\ & \overset{\text{(по дефиниция)}}{\equiv} & (M_1M_2)[y \mapsto P][x \mapsto N[y \mapsto P]]. \end{array}$$

(5) Нека $M \equiv (\lambda_z M')$, така че Лемата е изпълнена за M'. Можем да считаме, че $x \not\equiv z \not\equiv y$. Също така, можем да допуснем, че $z \notin \mathsf{FV}(N) \cup \mathsf{FV}(P)$ Тогава имаме:

$$\begin{array}{lll} (\lambda_z M')[x \mapsto N][y \mapsto P] & \overset{\text{(по дефиниция)}}{\equiv} & \lambda_z(M'[x \mapsto N][y \mapsto P]) \\ & \overset{\text{(по ИП)}}{\equiv} & \lambda_z(M'[y \mapsto P][x \mapsto N[y \mapsto P]]) \\ & \overset{\text{(по дефиниция)}}{\equiv} & (\lambda_z M')[y \mapsto P][x \mapsto N[y \mapsto P]]. \end{array}$$

□

Задача 1.6. *(2 т.) Да се покаже, че условията $x \notin \mathsf{FV}(P)$ и $x \not\equiv y$ в Лема 1.2 са съществени, т.е. че съществуват $M, N, P \in \Lambda$, така че $M[x \mapsto N][y \mapsto P] \not\equiv M[y \mapsto P][x \mapsto N[y \mapsto P]]$, ако някое от двете условия е нарушено.*

1.2. **Безименни λ-термове.** Конвенцията за имплицитно преименуване на свързаните променливи, която въведохме по-горе, позволява удобна интуитивна работа с λ-термовете, но не е много удобна за програмна реализация. Причината за това е, че се налага за сравнение на термове да се реализира релацията $\overset{\alpha}{=}$. Макар че такава реализация би могла да стане за линейно време, за нея се налага използване на структура от данни от тип речник, която поддържа информация за съответствието между имената на свързаните променливи. За поддържането на такава структура би било нужна памет линейна по дължината на терма.

Това неудобство би могло да се избегне, ако изберем подходящ синтаксис за λ-термове, който е еквивалентен на дефинирания по-горе, но при който релацията $\overset{\alpha}{=}$ се превежда до синтактично равенство. В този раздел ще представим една идея на de Bruijn, която предлага дефиниция на "каноничен" синтаксис на λ-смятането, в която променливите се представят с числови индекси вместо с произволни имена.

За да представим интуицията зад идеята на безименното представяне на термове, е удобно да си мислим за λ-термовете като графи. Можем да си представим, че на всеки λ-терм съответства абстрактно синтактично дърво, което има три вида възли, някои от които имат етикети:

(1) променливи, които нямат наследници, т.е. могат да са само листа. Етикетите на променливите са техните имена.
(2) апликации, които имат точно две поддървета, съответстващи на функцията и нейния аргумент

(3) λ-възли, които съответстват на абстракции и имат точно един наследник (тялото на функция). Етикетите на λ-възлите са имената на променливите, които те свързват.

В тези дървета можем да добавим "обратни" ребра, които представят свързаните променливи. Такива ребра ще започват от листа x и ще завършват във възел λ_x, но само ако x е сред наследниците на λ_x (т.е. x е в обхвата на λ_x). В така получените графи, понятието $\overset{\alpha}{=}$ съответства на съществуването на изоморфизъм на съответните графи такъв, който да запазва типа на възлите и етикетите на свободните променливи, т.е. на възлите от тип "променлива", от които няма изходящи ребра.

Задача 1.7. ***(8 т.)*** *Да се дефинира формално понятието "граф, съответстващ на λ-терм" и да се докаже, че два λ-терма са α-еквивалентни тогава и само тогава, когато съответните им графи са изоморфни.*

Тъй като етикетите на свързаните променливи и на λ-възлите служат само за построяване на обратните ребра, те могат да бъдат изтрити след построяването им. Сега въпросът за намиране на подходящ каноничен синтаксис на λ-смятането се свежда то това да се измисли еднозначен механизъм за описване на обратните ребра в графа. Стандартният синтаксис на λ-смятането използва "абсолютно" адресиране на λ-възлите в графа чрез глобално дефинирани имена и това води до нуждата от въвеждане на α-еквивалентност. Алтернативата е използването на "относително" адресиране на λ-възлите относно даден възел променлива. Има различни възможности за това, например:

(1) еднозначно номериране на всички λ-възли на графа с някакво каноничо обхождане на подлежащото му абстрактно синтактично дърво (например ляво-корен-дясно)
(2) номериране на λ-възлите по пътя до променливата започвайки от корена в посока към листото
(3) номериране на λ-възлите по пътя до променливата започвайки от листото в посока към корена

Не е без значение кой от гореописаните подходи ще изберем, тъй като искаме относителното адресиране да е максимално устойчиво при синтактични трансформации над λ-терма. Такива синтактични трансформации са, например:

- прилагане на субституция
- разглеждане на подтерм
- конструиране на нов терм

Решението 1 е най-неустойчивото: при него се налага преименуване и при трите синтактични трансформации изброени по-горе. Решението 2 е устойчиво относно прилагане на субституция, но не и при отделяне на подтерм или конструиране на нов терм. Накрая, решението 3 е устойчиво и при трите трансформации, затова ще използваме именно него. Така всяка свързана променлива ще представяме с естествено число, което представя номера на λ-възела, към когото е свързана, броейки отвътре навън и започвайки от 0.

Бележка 1.3. В оригиналната си статия de Bruijn предлага решенията 2 и 3, но с времето решението 3 се е утвърдило като практически по-удачно.

Пример 1.3.

- $\lambda_x x$ се представя като $\lambda 0$,

- $\lambda_x\lambda_y y$ се представя като $\lambda\lambda 0$,
- $\lambda_x\lambda_y x$ се представя като $\lambda\lambda 1$,
- $\lambda(\lambda 0)(\lambda 1)$ отговаря на $\lambda_x(\lambda_y y)(\lambda_z x)$,
- $\lambda_x x(\lambda_z z(\lambda_y xyz)x)$ се представя като $\lambda 0(\lambda 0(\lambda 201)1)$.

Все още е необходимо да решим как ще представяме свободните променливи. При тях трябва да изберем такова представяне, което позволява лесното свързване на променливи с еднакви имена при конструиране на нов терм. Поточно, ако имаме терм M в който имаме няколко свободни променливи с име x, искаме да можем лесно да построим терма $\lambda_x M$. Тъй като всяко срещане на x може да е скрито под различен брой λ-абстракции, това означава, че всяко срещане на x трябва да бъде заменено с число, което съответства точно на броя λ, зад които то е скрито, т.е. на номера на λ-абстракция, която ще бъде поставена отвън на терма. В общия случай, една свободна променлива x трябва да е номерирана с такова число n, така че за всички срещания на x разликата между n и броя на λ-абстракциите, зад които е скрито срещането на x да е константно число i. Това число i ще наричаме *индекс на de Bruijn* на променливата x или просто *индекс на* x. Така ако имаме даден безименен терм M, то λM ще свърже променливата с индекс 0, $\lambda\lambda M$ ще свърже променливите с индекси 0 и 1 и т.н.

Пример 1.4. Нека свободната променлива y има индекс 0, а свободната променлива z има индекс 2. Тогава

- термът $\lambda_x y$ се представя като $\lambda 1$
- термът $y(\lambda_x xyz)z$ се представя като $0(\lambda 013)2$,
- термът $(\lambda_x xy(\lambda_u zu(\lambda_v vy))z)$ се представя като $(\lambda 01(\lambda 40(\lambda 03))3)$.
- термът $\lambda(\lambda 0((\lambda 1)21))1$ отговаря на $\lambda_x(\lambda_t t((\lambda_u t)yx))y$.

Вече сме готови да дадем формална дефиниция на безименни термове.

Дефиниция 1.7 (Безименни термове, Λ_n, Λ^*)**.** С едновременна индукция за всяко n дефинираме множествата Λ_n от безименни термове с не повече от n различни свободни променливи.

(1) $i \in \Lambda_n$ за всяко естествено число i, за което $0 \leq i < n$.
(2) Ако $M, N \in \Lambda_n$, то $(MN) \in \Lambda_n$ е апликацията на M над N.
(3) Ако $M \in \Lambda_{n+1}$, то $\lambda M \in \Lambda_n$ е абстракцията над променливата с индекс 0 в M.

С $\Lambda^* := \bigcup_{n=0}^{\infty} \Lambda_n$ отбелязваме множеството на всички безименни λ-термове.

Може да бъде формално проверено, че синтаксисът на безименните термове е изоморфен на синтаксиса на термове с имена с точност до α-еквивалентност. За тази цел могат да бъдат дефинирани две изображения $\#(\cdot)$ и $\flat(\cdot)$, които преобразуват безименен терм в терм с имена на променливите и обратно. За дефиницията на изображенията ни е необходима помощна дефиниция, която да задава съответствие между имена на променливи от V и индекси на свободни променливи.

Дефиниция 1.8 (Контекст от имена)**.** Крайна редица от различни променливи $\Gamma := x_{n-1}, \ldots, x_0 \in V$ ще наричаме *контекст от имена*. Дължината на контекста ще бележим с $|\Gamma| := n$, а множеството $\operatorname{dom}\Gamma := \{x_{n-1}, \ldots, x_0\}$ ще наричаме домейн на контекста. При даден контекст Γ, ще казваме, че променливата x_i има индекс i.

Дефиниция 1.9. Нека $X \subseteq V$ е множество от променливи. Дефинираме $\Lambda_X := \{M \in \Lambda : \mathsf{FV}(M) \subseteq X\}$.

Задача 1.8. ***(15 т.)*** *Да се дефинират фамилиите от изображения* $\#_\Gamma : \Lambda_{|\Gamma|} \to \Lambda_{\mathrm{dom}\,\Gamma}$ *и* $\flat_\Gamma : \Lambda_{\mathrm{dom}\,\Gamma} \to \Lambda_{|\Gamma|}$ *за даден контекст от имена* $\Gamma := x_{n-1}, \ldots, x_0$, *които извършват превод между термове с имена и безименни термове. Да се покаже, че*

(1) $\#_\Gamma(\flat_\Gamma(M)) \equiv M$ *за всеки терм* $M \in \Lambda_{\mathrm{dom}\,\Gamma}$
(2) $\flat_\Gamma(\#_\Gamma(M)) \overset{\alpha}{=} M$ *за всеки терм* $M \in \Lambda_{|\Gamma|}$

Пример 1.5. В Пример 1.4 беше използван контекст от вида $\Gamma := z, u, y$. Така $\#_\Gamma(\lambda_1) \overset{\alpha}{=} \lambda_x y$, а $\flat_\Gamma(y(\lambda_x xyz)z) \equiv 0(\lambda 013)2$.

Задача 1.9. ***(15 т.)*** *Да се реализира програма, която позволява въвеждането и извеждането на* λ*-термове в два формата: с имена* (Λ) *и без имена* (Λ^*) *на променливите. За преобразуването между двата формата да се използва автоматично генериран контекст от имена от вида* $\Gamma := x_{n-1}, \ldots, x_0$.

Следващата задача е да дефинираме субституция при безименните λ-термове. Субституцията ще има вида $[k \mapsto M]$, където $k \in N$ е индекс на свободна променлива, а $M \in \Lambda^*$ е безименен терм. Прихващането на свободни променливи от λ-абстракции в аспекта на безименните λ-термове е невъзможно, тъй като по дефиниция всяко срещане на свободна променлива е означено с число, поголямо от номера на всяка λ-абстракция, под която е скрито. Въпреки това, трябва да обърнем внимание, че при субституцията е възможно свободните променливи в M да бъдат поставени под различен брой λ-абстракции. Затова преди да дадем дефиницията за субституция, трябва да се подготвим със синтактичен инструмент $\uparrow^d$, който позволява изместването на индексите на свободните променливи с някакво естествено число $d \in \mathbb{N}$. Такова изображение ще дефинираме индуктивно; но то трябва да бъде дефинирано така, че да увеличава с d само числата, съответстващи на свободни променливи в оригиналния терм. Това може да бъде постигнато чрез добавянето на още един параметър $c \in \mathbb{N}$, който "помни" броят на λ-абстракциите, през които сме преминали. Така ще знаем, че числата, които са по-малки от c съответстват на свързани променливи и не трябва да бъдат изместени с d.

Дефиниция 1.10 (Изместване)**.** Дефинираме $\uparrow_c^d (M) : \Lambda_n \to \Lambda_{n+d}$ с индукция по построението на терма $M \in \Lambda_n$.

(1) $\uparrow_c^d (k) := \begin{cases} k, & \text{за } 0 \le k < c, \\ k+d, & \text{за } c \le k < n. \end{cases}$
(2) $\uparrow_c^d (MN) := (\uparrow_c^d (M))(\uparrow_c^d (N))$
(3) $\uparrow_c^d (\lambda M) := \lambda \uparrow_{c+1}^d (M)$

Дефинираме $\uparrow^d (M) := \uparrow_0^d (M)$.

Дефиниция 1.11 (Субституция на безименни термове)**.** Нека $M, N \in \Lambda_n$ и $k \in \mathbb{N}$. С индукция по M дефинираме субституцията $M[k \mapsto N] \in \Lambda_n$.

(1) $k[k \mapsto N] := N$
(2) $i[k \mapsto N] := i$ за $i \neq k$
(3) $(M_1 M_2)[k \mapsto N] := (M_1[k \mapsto N])(M_2[k \mapsto N])$
(4) $(\lambda M)[k \mapsto N] := \lambda(M[k+1 \mapsto \uparrow^1 (N)])$

Пример 1.6. Нека отново приемем, че променливата y има индекс 0, а променливата z има индекс 2. Тогава субституцията

$$(y(\lambda_x xyz)z)[y \mapsto zy] \equiv zy(\lambda_x x(zy)z)z$$

се представя като

$$\begin{array}{rcl}(0(\lambda 013)2)[0 \mapsto 20] & \equiv & 20((\lambda 013)[0 \mapsto 20])2 \\ & \equiv & 20(\lambda(013[1 \mapsto \uparrow^1 (20)]))2 \\ & \equiv & 20(\lambda 0(31)3)2.\end{array}$$

Пример 1.7. Субституцията

$$y(\lambda_x xy(\lambda_u uxy))[y \mapsto z(\lambda_v vyz)] \;\equiv\; z(\lambda_v vyz)(\lambda_x x(z(\lambda_v vyz))(\lambda_u ux(z(\lambda_v vyz))))$$

се представя като

$$M := 0(\lambda 01(\lambda 012))[0 \mapsto 2(\lambda 013)] \;\equiv\; 2(\lambda 013)(\lambda(01(\lambda 012))[1 \mapsto \uparrow^1 (2(\lambda 013))]).$$

Тъй като $\uparrow^1 (2(\lambda 013)) \equiv 3(\lambda \uparrow^1_1 (013)) \equiv 3(\lambda 024)$, получаваме

$$M \equiv\; 2(\lambda 013)(\lambda 0(3(\lambda 024))(\lambda(012)[2 \mapsto \uparrow^1 (3(\lambda 024))])).$$

И отново, понеже $\uparrow^1 (3(\lambda 024)) \equiv 4(\lambda 035)$, окончателно имаме

$$M \equiv\; 2(\lambda 013)(\lambda 0(3(\lambda 024))(\lambda 01(4(\lambda 035)))).$$

Задача 1.10. ***(15 т.)*** *Да се провери, че двете дефиниции за субституции са съгласувани относно изображенията $\#_\Gamma$ и $\flat_\Gamma$. За целта, нека фиксираме контекст от имена $\Gamma := x_{n-1}, \ldots, x_0$. Да се покаже, че*

(1) $\#_\Gamma(M)[x_i \mapsto \#_\Gamma(N)] \overset{\alpha}{=} \#_\Gamma(M[i \mapsto N])$ *за произволни $M, N \in \Lambda_n$,*
(2) $\flat_\Gamma(M)[i \mapsto \flat_\Gamma(N)] \equiv \flat_\Gamma(M[x_i \mapsto N])$ *за произволни $M, N \in \Lambda_{\mathrm{dom}\,\Gamma}$.*

Задача 1.11. ***(5 т.)*** *Да се направи програмна реализация на субституция над безименни термове.*

1.3. **Редукции.** След като вече въведохме синтаксиса на λ-смятането, ще обърнем внимание на неговата изчислителна семантика. За целта ще дефинираме няколко редукции, които дефинират как един λ-терм може да се сведе до друг терм със същата семантика от изчислителна гледна точка. От формална гледна точка редукциите представляват бинарни релации $R \subseteq \Lambda^2$.

Първата и основна редукция, която ще разгледаме е β-редукцията. Ако разгледаме операцията "абстракция" като конструктор или генератор на функции, а операцията "апликация" като деструктор или консуматор на функции, то можем да си мислим, че тези операции са дуални една на друга. β-редукцията дефинира как тези две операции си взаимодействат. Нека имаме конструирана функция $\lambda_x M$ с аргумент x и тяло N и нека приложим тази функция над някакъв λ-терм N. Резултатът от това прилагане би следвало да замести "формалния" параметър x в тялото на функцията M с "фактическия" параметър N. Полученият резултат е термът $M[x \mapsto N]$, за който казваме, че е получен с β-редукция от терма $(\lambda_x M)N$ и отбелязваме

$$(\lambda_x M)N \xrightarrow{\beta} M[x \mapsto N].$$

Терм от вида $(\lambda_x M)N$ наричаме *β-редекс*, а терма $M[x \mapsto N]$ наричаме негов *β-редукт*. В даден λ-терм може да има едновременно много β-редекси на произволна дълбочина. За да формализираме какво означава това, ще разгледаме понятието *подтерм*.

Дефиниция 1.12 (Подтерм)**.** Дефинираме индуктивно релацията "M е подтерм на N", което отбелязваме $M \leq N$ за $M, N \in \Lambda$.

(1) $M \leq M$.
(2) Ако $\lambda_x M \leq N$, то $M \leq N$.
(3) Ако $MN \leq P$, то $M \leq P$ и $N \leq P$.

Пример 1.8. Термът $x(\lambda_y xy)$ има 5 различни подтерма: x, y, xy, $\lambda_y xy$ и $x(\lambda_y xy)$.

Задача 1.12. ***(3 т.)*** *Докажете, че релацията $\leq$ е частична наредба, т.е. че е рефлексивна, транзитивна и антисиметрична.*

Искаме да дефинираме β-редукцията така, че тя да може да редуцира произволен подтерм на даден терм M, който е β-редекс. За да направим това, ще дефинираме малко по-общото понятие λ-*затваряне*.

Дефиниция 1.13 (λ-затваряне)**.** Нека е дадена бинарна релация над λ-термове $R \subseteq \Lambda^2$. Дефинираме индуктивно релацията R^λ, която наричаме λ-*затваряне* на R, по следния начин:

(1) Ако $(M, N) \in R$, то $(M, N) \in R^\lambda$.
(2) Ако $(M, N) \in R^\lambda$, $P \in \Lambda$ и $x \in V$, то
   - $(MP, NP) \in R^\lambda$,
   - $(PM, PN) \in R^\lambda$,
   - $(\lambda_x M, \lambda_x N) \in R^\lambda$.

Ако $R^\lambda = R$, казваме че R е λ-*съвместима*.

Интуитивно, два терма M и N са в релация R^λ ако те съвпадат синтактично с изключение на два техни съответни подтерма $M' \leq M$ и $N' \leq N$, които са в релация R.

Пример 1.9. Релацията $\equiv$ е λ-съвместима.

Задача 1.13. ***(2 т.)*** *Дайте друг пример за λ-съвместима релация.*

Задача 1.14. ***(2 т.)*** *Докажете, че R^λ е λ-съвместима за произволна релация R.*

Пример 1.10. Нека $x \in V$ е променлива и нека $R = \{(x, P) : P \in \lambda\}$. Тогава $(M, N) \in R^\lambda$ тогава и само тогава, ако N може да се получи от M чрез заместване на едно срещане (свободно или свързано) на променливата x с някакъв терм P. Например, $(xy, (\lambda_z z)y) \in R^\lambda$, $(\lambda_{x,y} x, \lambda_{x,y} y) \in R^\lambda$, но $(xx, yy) \notin R^\lambda$.

Задача 1.15. ***(5 т.)*** *Да се докаже, че $(M, N) \in R^\lambda$ тогава и само тогава, когато съществуват терм P, променлива $x \notin \mathsf{FV}(MN) \cup \mathsf{BV}(MN)$ и два подтерма $M' \leq M$ и $N' \leq N$, така че*

(1) $P[x \mapsto M'] \equiv M$
(2) $P[x \mapsto N'] \equiv N$
(3) $(M', N') \in R$.

Вече сме готови да дадем дефиниция β-редукция.

Дефиниция 1.14 ($\xrightarrow{\beta}$, β-редукция)**.** Нека разгледаме релацията

$$\beta := \left\{ \left((\lambda_x M)N, M[x \mapsto N] \right) : M, N \in \Lambda, x \in V \right\}.$$

β-редукцията дефинираме като λ-затваряне на β, т.е. $\xrightarrow{\beta} := \beta^\lambda$. Обратната релация $\xleftarrow{\beta} := \xrightarrow{\beta}^{-1} := \left\{ (M, N) : N \xrightarrow{\beta} M \right\}$ наричаме β-експанзия.

Пример 1.11.

- $(\lambda_x x)y \xrightarrow{\beta} y$
- $\lambda_z(\lambda_{x,y}x)(\lambda_u u)(\lambda_u uzu)z \xrightarrow{\beta} \lambda_z(\lambda_u u)z \xrightarrow{\beta} \lambda_z z$
- $(\lambda_x xz)((\lambda_y zy)z) \xrightarrow{\beta} (\lambda_x xz)(zz) \xrightarrow{\beta} zzz$
- $(\lambda_x xz)((\lambda_y zy)z) \xrightarrow{\beta} (\lambda_y zy)zz \xrightarrow{\beta} zzz$

Бележка 1.4 (β-редукция на безименни λ-термове). Дефиницията на β-редукция при безименните λ-термове интуитивно би трябвало да изглежда по следния начин: $(\lambda M)N \xrightarrow{\beta} M[0 \mapsto N]$. Но като се вгледаме внимателно, забелязваме, че тази дефиниция е некоректна. Например, нека да приемем че променливата y отговаря на индекс 1. Тогава $(\lambda_x xy)y \xrightarrow{\beta} yy$ се превежда до $(\lambda 02)1 \xrightarrow{\beta} 11 \not\equiv 12 \equiv (02)[0 \mapsto 1]$. Проблемът идва от това, че премахвайки една λ-абстракция от λM би трябвало числата, съответстващи на свободните променливи да се намалят с единица, с изключение на току-що освободената променлива с индекс 0, която от своя страна ще бъде заместена с N. Този проблем може най-лесно да се реши, ако числата съответстващи на всички свободни променливи в N се увеличат с единица, след това се извърши субституцията и най-накрая свободните променливи отново се намалят с единица, т.е. $(\lambda M)N \xrightarrow{\beta} \uparrow^{-1}(M[0 \mapsto \uparrow^1 N])$.

Задача 1.16. ***(3 т.)*** *Да се дефинира формално релацията $\xrightarrow{\beta}$ за Λ^* и да се докаже, че двете β-редукции са съгласувани, т.е. за произволен контекст от имена Γ*

(1) $\flat_\Gamma(M) \xrightarrow{\beta} \flat_\Gamma(N)$, *ако* $M, N \in \Lambda$, $\mathsf{FV}(MN) \subseteq \operatorname{dom}\Gamma$ *и* $M \xrightarrow{\beta} N$.

(2) $\#_\Gamma(M) \xrightarrow{\beta} P$, *ако* $M, N \in \Lambda_{|\Gamma|}$, $M \xrightarrow{\beta} N$ *и* $P \overset{\alpha}{=} \#_\Gamma(N)$.

Релацията $\xrightarrow{\beta}$ още се нарича едностъпкова редукция, понеже извършва редукция само над един редекс. Нейно обобщение е многостъпковата β-редукция, дефинирана по-долу. Преди това ще припомним операциите рефлексивно, симетрично и транзитивно затваряне на релации.

Дефиниция 1.15 (Рефлексивно, симетрично и транзитивно затваряне)**.** Нека R е бинарна релация. Разглеждаме нова релация R', дефинирана чрез следните индуктивни клаузи:

(B) $R \subseteq R'$, т.е. ако $(x, y) \in R$, то $(x, y) \in R'$ (съгласуваност с R)
(R) $(x, x) \in R'$ (рефлексивно затваряне)
(S) Ако $(x, y) \in R'$, то $(y, x) \in R$ (симетрично затваряне)
(T) Ако $(x, y) \in R'$ и $(y, z) \in R'$, то $(x, z) \in R'$ (транзитивно затваряне).

Можем да дефинираме R' като комбинираме базовата клауза (B) с произволно непразно подмножество от клаузите (R), (S) и (T).

Дефиниция 1.16 ($\xrightarrow{\beta}\!\!\!\!\!\rightarrow$, многостъпкова β-редукция)**.** $\xrightarrow{\beta}\!\!\!\!\!\rightarrow$ дефинираме като рефлексивно и транзитивно затваряне на $\xrightarrow{\beta}$.

β-редукцията поражда релация на еквивалентност между λ-термовете, която наричаме β-еквивалентни.

Дефиниция 1.17 ($\overset{\beta}{=}$, β-еквивалентност)**.** $\overset{\beta}{=}$ дефинираме като рефлексивно, симетрично и транзитивно затваряне на $\xrightarrow{\beta}$.

Бележка 1.5. Алтернативно, бихме могли да дефинираме $\overset{\beta}{=}$ като симетрично и транзитивно затваряне на $\overset{\beta}{\twoheadrightarrow}$.

Интуитивно, два терма M и N са β-еквивалентни ако има крайна редица от редукции и експанзии, с които от M може да се получи N.

Пример 1.12. Нека $I := \lambda_x x$, $K := \lambda_{x,y} x$, $K^* := \lambda_{x,y} y$, $S := \lambda_{x,y,z} xz(yz)$ и нека $M, N, P \in \Lambda$ са произволни термове. Тогава:

- $IM \overset{\beta}{=} M$
- $KMN \overset{\beta}{=} M$
- $K^*MN \overset{\beta}{=} N$
- $SMNP \overset{\beta}{=} MP(NP)$
- $SKK \overset{\beta}{=} I$
- $SKS \overset{\beta}{=} I$
- $SK \overset{\beta}{=} K^*$

Името "β-редукция" е малко подвеждащо, понеже изпълнението на една редукционна стъпка може да доведе до нарастване на дължината на терма.

Пример 1.13. $(\lambda_{x,f} fxxxx)(yzuvw) \overset{\beta}{\to} \lambda_f f(yzuvw)(yzuvw)(yzuvw)(yzuvw)$

Интересно е да видим колко може да нарасне даден терм след β-редукция. За целта нека дефинираме понятието "дължина на терм".

Дефиниция 1.18 (дължина на терм)**.** Нека $M \in \Lambda$, дефинираме $|M|$ (дължината на M) с индукция по M:

- $|x| := 1$
- $|MN| := |M| + |N|$
- $|\lambda_x M| := |M| + 1$.

Преди да разгледаме как нараства даден терм при β редукция, нека първо намерим оценка на $|M[x \mapsto N]|$. В най-лошия случай броят на срещанията на x в M ще е равен на $|M|$ (ако $M \equiv xx \dots x$). Тогава $|M[x \mapsto N]| \in O(|M|\,|N|)$.

Сега нека $M \overset{\beta}{\to} N$. В най-лошия случай, $M \equiv (\lambda_x P)Q$, а $N \equiv P[x \mapsto Q]$. Най-лоша оценка ще получим, ако $|P|$ и $|Q|$ зависят линейно от $|M|$. Прилагайки горната оценка получаваме $|N| \in O(|M|^2)$.

Съответно, ако изпълним n-кратно β-редукция, така че

$$M_0 \overset{\beta}{\to} M_1 \overset{\beta}{\to} \dots M_{n-1} \overset{\beta}{\to} M_n,$$

ще имаме $|M_n| \in O(|M_0|^{2^n})$.

Тази оценка изглежда прекалено висока и затова е логичен въпросът дали тя реално се достига. Факт е, че съществуват термове, чиято n-кратна β-редукция води до подобно експоненциално увеличение на дължината, но такъв пример ще покажем по-късно.

Също така е любопитно да видим дали е възможно един терм да нараства неограничено много. Нека преди това да си отговорим на по-простия въпрос: възможно ли е да имаме безкрайна редица от β-редукции?

Оказва се, че това е възможно, при това доста лесно, макар и не напълно интуитивно. Да разгледаме комбинатора $\omega := \lambda_x xx$ и да дефинираме $\Omega := \omega\omega$. Очевидно $\Omega \overset{\beta}{\to} \Omega$, което води до безкрайна редица от β-редукции. Както

ще видим по-късно, комбинаторът Ω всъщност изпълнява ролята на безкраен цикъл.

Интуицията зад ω не е съвсем ясна на пръв поглед, понеже е трудно да си представим функция, която се изпълнява над себе си. В термините на машини на Тюринг, обаче, това съвсем не е необичайно: всяка машина на Тюринг може да се кодира с естествено число, което да бъде записана на лентата на същата машина на Тюринг, което ще доведе на изпълнение на машината на Тюринг над собствения си код!

Сега вече е лесно да си представим терм, който при β-редукция нараства неограничено.

Пример 1.14. Нека $\omega_3 := \lambda_x xxx$. Тогава имаме

$$\omega_3\omega_3 \xrightarrow{\beta} \omega_3\omega_3\omega_3 \xrightarrow{\beta} \omega_3\omega_3\omega_3\omega_3 \xrightarrow{\beta} \dots$$

Комбинаторите S и K, които разгледахме по-горе, имат специална роля. Оказва се, че всеки Λ-терм има β-еквивалентен терм образуван само от апликации на комбинаторите S и K. В това може да ни увери следната Лема.

Лема 1.3. *Нека $M, N \in \Lambda$. Тогава*

(1) $\lambda_x x \overset{\beta}{=} SKK$,
(2) $\lambda_x M \overset{\beta}{=} KM$, *ако* $x \notin \mathsf{FV}(M)$,
(3) $\lambda_x MN \overset{\beta}{=} S(\lambda_x M)(\lambda_x N)$.

Доказателство. (1) $SKK \overset{\beta}{=} \lambda_z Kz(Kz) \overset{\beta}{=} \lambda_z z \overset{\alpha}{=} \lambda_x x$.

(2) $KM \equiv (\lambda_{x,y} x)M \overset{\beta}{=} \lambda_y M$, като можем да считаме, че $y \notin \mathsf{FV}(M)$.

(3) $S(\lambda_x M)(\lambda_x N) \overset{\beta}{=} \lambda_z(\lambda_x M)z((\lambda_x N)z) \overset{\beta}{=} \lambda_z M[x \mapsto z]N[x \mapsto z] \overset{\alpha}{=} \lambda_x MN$. □

Лема 1.3 ни дава алгоритъм за превеждане на произволни λ-термове до термове, състоящи се само от S и K като прилагаме Лемата "отвътре-навън".

Пример 1.15.

$$K^* \equiv \lambda_x \lambda_y y \overset{\beta}{=} \lambda_x SKK \overset{\beta}{=} K(SKK).$$

$$\begin{aligned}\lambda_x\lambda_y yx &\overset{\beta}{=} \lambda_x S(\lambda_y y)(\lambda_y x) \overset{\beta}{=} \lambda_x S(SKK)(Kx)\\ &\overset{\beta}{=} S(\lambda_x S(SKK))(\lambda_x Kx) \overset{\beta}{=} S(K(S(SKK)))(S(\lambda_x K)(\lambda_x x))\\ &\overset{\beta}{=} S(K(S(SKK)))(S(KK)(SKK)).\end{aligned}$$

Дефиниция 1.19 (Апликативни термове)**.** Дефинираме множеството от апликативни λ-термове $A\Lambda \subseteq \Lambda$ индуктивно по следния начин

(1) Ако $x \in V$, то $x \in A\Lambda$.
(2) Ако $M, N \in A\Lambda$, то $MN \in A\Lambda$.

Бележка 1.6. Системата, която се състои от апликативни термове съставени от комбинаторите S и K с редукциите $KMN \xrightarrow{\beta} M$ и $SMNP \xrightarrow{\beta} MP(NP)$ е създадена от Curry под името *комбинаторна логика*.

Задача 1.17. ***(5 т.)*** *Нека k и s са две фиксирани променливи от V. Да се дефинира изображение $\Phi : \Lambda \Rightarrow A\Lambda$, което превежда произволен λ-терм в комбинаторна логика. Да се покаже, че за произволно $M \in \Lambda$:*

(1) $\mathsf{FV}(\Phi(M)) = \mathsf{FV}(M) \cup \{k, s\}$ *и*

(2) $M \overset{\beta}{=} \Phi(M)[k \mapsto K][s \mapsto S]$.

Екстра кредит: ***(2 т.)*** *Да се направи програмна реализация на изображението Φ.*

Релацията $\overset{\beta}{=}$ може да се разглежда като *операционна еквивалентност* на термове, т.е. два терма са β-еквивалентни, ако чрез поредица от изчисления можем да стигнем от единия до другия.

Дефиниция 1.20 (Операционна еквивалентност)**.** Казваме, че в λ-смятането два терма M и N са *операционно еквивалентни*, ако $M \overset{\beta}{=} N$ и бележим $\lambda \models M = N$.

Операционното еквивалентност е вид *интенсионално равенство*, т.е. равенство на термовете, което е породено от тяхната вътрешна структура. В математиката има по-силен вид равенство, наречено *екстенсионално равенство*, което представлява равенство относно външното проявление на термовете, т.е. относно тяхното поведение като математически функции. Например, очевидно $\lambda_x fx \overset{\beta}{\neq} f$, но за произволен терм M имаме, че $(\lambda_x fx)M \overset{\beta}{=} fM$. Ектенсионалното равенство в λ-смятането традиционно се задава с така наречените η-редукция и η-еквивалентност.

Дефиниция 1.21 ($\overset{\eta}{\to}$, η-редукция)**.** Нека разгледаме релацията

$$\eta := \{(\lambda_x Mx, M) : \ M \in \Lambda, x \notin \mathsf{FV}(M)\}\,.$$

η-редукцията дефинираме като λ-затваряне на η, т.е. $\overset{\eta}{\to} := \eta^{\lambda}$.

Дефиниция 1.22 ($\overset{\eta}{=}$, $\overset{\beta\eta}{=}$)**.** Релацията $\overset{\eta}{=}$ (η-еквивалентност) наричаме рефлексивното, симетричното и транзитивното затваряне на η-редукцията $\overset{\eta}{\to}$.

Релацията $\overset{\beta\eta}{=}$ ($\beta\eta$-еквивалентност) наричаме симетричното и транзитивното затваряне на обединението на релациите $\overset{\beta}{=}$ и $\overset{\eta}{=}$.

Лесно може да се види, че $\overset{\beta\eta}{=}$ описва точно екстенсионалното равенство между термове.

Дефиниция 1.23 (екстенсионално равенство)**.** Казваме, че M и N са екстенсионално равни и бележим $\lambda + \mathrm{ext} \models M = N$, ако

(1) $\lambda \models M = N$,

(2) за произволно $x \notin \mathsf{FV}(MN)$ е вярно, че $\lambda + \mathrm{ext} \models Mx = Nx$.

Задача 1.18. ***(3 т.)*** *Да се докаже, че релацията $\lambda + \mathrm{ext} \models M = N$ е λ-съвместима релация на еквивалентност.*

Твърдение 1.2. *$M \overset{\beta\eta}{=} N$ точно тогава, когато $\lambda + \mathrm{ext} \models M = N$.*

Доказателство. ($\Leftarrow$) Индукция по $M \overset{\beta\eta}{=} N$. Тъй като релацията $\lambda + \mathrm{ext} \models M = N$ е релация на еквивалентност и освен това е λ-съвместима, можем да разгледаме само базовите случаи.

Нека $M\beta N$. Тогава по дефиниция $\lambda \models M = N$, откъдето $\lambda + \text{ext} \models M = N$.

Нека $M\eta N$. Тогава $M \equiv \lambda_x Nx$ за някое $x \notin \mathsf{FV}(N)$. Нека $y \notin \mathsf{FV}(MN)$. Тогава очевидно $My \overset{\beta}{=} Ny$, откъдето $\lambda + \text{ext} \models My = Ny$, а оттам и $\lambda + \text{ext} \models M = N$.

($\Rightarrow$) Индукция по $\lambda + \text{ext} \models M = N$.

Нека $\lambda \models M = N$. Тогава $M \overset{\beta}{=} N$, откъдето очевидно $M \overset{\beta\eta}{=} N$.

Нека $\lambda + \text{ext} \models Mx = Nx$ *за произволно* $x \notin \mathsf{FV}(MN)$. Нека $x \notin \mathsf{FV}(MN)$. Тогава по индукционно предположение имаме:

$$M \overset{\eta}{=} \lambda_x Mx \overset{\beta\eta}{=} \lambda_x Nx \overset{\eta}{=} N.$$

□

1.4. **Изчисления в λ-смятането.** Вече споменахме резултата на Turing, според който λ-смятането като изчислителен модел е еквивалентен на машините на Тюринг. В този раздел ще дадем идея как могат да се пишат изчислими функции в λ-смятането. За разлика от други изчислителни модели, в λ-смятането нямаме "цифрова информация", която да бъде обработвана, например естествени числа в двоичен запис, както при машините на Тюринг. Въпреки това, езикът на λ-смятането, макар и прост, е достатъчен, за да кодира естествени числа. Тук ще разгледаме едно популярно кодиране, предложено от Church. Основната идея при него е, че броенето може да се представи като брой последователни прилагания на една и съща функция над даден аргумент.

Дефиниция 1.24 (n-кратно прилагане на функция)**.** Нека $n \in \mathbb{N}$ и $f, x \in \Lambda$. Дефинираме терма $f^n x \in \Lambda$ с индукция по n:

(1) $f^0 x := x$,
(2) $f^{n+1} x := f(f^n x)$.

Дефиниция 1.25 (Нумерали на Church)**.** За всяко естествено число $n \in \mathbb{N}$ дефинираме комбинатора $c_n := \lambda_{f,x} f^n x$, който представя n.

Пример 1.16. По-долу са дадени някои от нумералите на Church:

- $c_0 := \lambda_{f,x} x \equiv \lambda_f I \equiv K^*$
- $c_1 := \lambda_{f,x} fx \overset{\eta}{=} \lambda_f f \equiv I$
- $c_5 := \lambda_{f,x} f(f(f(f(fx))))$

След като имаме представяне на естествените числа, вече можем да пишем и термове, които отговарят на изчислими функции над естествени числа.

Пример 1.17. Да потърсим комбинатор c_S такъв, че за произволно $n \in \mathbb{N}$ имаме, че $c_S c_n \overset{\beta}{=} c_{n+1}$. Термът c_S трябва да е такъв, че да преобразува n-кратното прилагане на функция в $n+1$-кратно такова. Това може да се случи по два различни начина:

- $c_S := \lambda_{n,f,x} f(nfx)$, или
- $c_S := \lambda_{n,f,x} nf(fx)$.

Очевидно и в двата случая $c_S c_n \overset{\beta}{=} \lambda_{f,x} f^{n+1} x$. В първия вариант на дефиницията това е по-лесно да се докаже, понеже се следва точно дефиницията на $f^n x$.

Интуитивно, нумералите на Church можем да разглеждаме като функции от по-висок ред, които моделират цикли с фиксиран брой итерации. Друг начин за разглеждане на нумералите на Church е като рекурсивни схеми, които приемат аргумент f, който описва стъпката на рекурсията и аргумент x, който описва базата на рекурсията. Това ни дава и идея как да програмираме по-сложни числови функции над естествени числа.

Пример 1.18. Да потърсим комбинатор c_+, който моделира събирането над естествени числа. Формално, искаме за произволни $m, n \in \mathbb{N}$ да имаме $c_+ c_m c_n \overset{\beta}{=} c_{m+n}$.

Идеята за реализация е подобна на тази на комбинатора c_S. За да постигнем $m+n$-кратно прилагане на функция, трябва да направим последователно m- и n-кратно прилагане. Отново, поради комутативността на събирането, можем да направим две подобни реализации:

- $c_+ := \lambda_{m,n,f,x} mf(nfx)$, или
- $c_+ := \lambda_{m,n,f,x} nf(mfx)$.

Задача 1.19. ***(2 т.)*** *Да се докаже формално, че c_+ наистина моделира операцията събиране.*

Програмна реализация на нумерали на Church можем да направим на произволен динамично типизиран език за програмиране, който реализира анонимни функции (closures) и има възможност за построяване на функции от произволно висок ред.

Например, на езика Scheme, реализацията на нумерали на Church би изглеждала така:

```
(define (repeat n f x) (if (= n 0) x (f (repeat (- n 1) f x))))
(define (c n) (lambda (f) (lambda (x) (repeat n f x))))
```

С лекота могат да бъдат реализирани и комбинаторите c_S и c_+:

```
(define cs (lambda (n) (lambda (f) (lambda (x) (f ((n f)
   x))))))
(define c+ (lambda (m) (lambda (n)
            (lambda (f) (lambda (x) ((m f) ((n f) x))))))))
```

Езиците за програмиране, които поддържат анонимни функции, изпълняват форма на β-редукция и ни позволяват да извършваме реално пресмятания с нумерали на Church. Повече от тях, обаче, не дават възможност за "извеждане" на телата на функциите. Например, на Scheme получаваме следния резултат:

```
> (c+ (c 2))
#<procedure>
```

За да можем да "видим" резултата от пресмятането, е необходимо да можем да преобразуваме нумерал на Church в естествено число (обратното преобразуване вече се дава от Scheme функцията `c`. Как да реализираме функция `print`, така че `(print (c n))` да извежда числото n? Отново ще използваме, че `(c n)` моделира n-кратна рекурсия и така ще получим n като n-кратно добавяне на единица към числото 0:

```
(define (print n) ((n (lambda (x) (+ x 1))) 0))
```

Така, вече имаме

```
> (print (c 2))
2
> (print (cs (c 2)))
3
> (print ((c+ (c 5)) (c 3)))
8
```

Пример 1.19. Следвайки идеята за нумералите като рекурсивни схеми, можем да предложим алтернативни дефиниции на c_+:

- $c_+ := \lambda_{m,n} n c_S m$, т.е. събиране на m с n дефинираме като n-кратно прибавяне на единица към m, или
- $c_+ := \lambda_{m,n} m c_S n \overset{\eta}{=} \lambda_m m c_S$, т.е. събиране с m дефинираме като m-кратно прибавяне на единица.

Разликата с предишните реализации е, че този път използваме рекурсията над m на по-високо ниво, а именно над нумерали, а не над аргументи x.

Задача 1.20. ***(2 т.)*** *Да се докаже коректността на горните дефиниции на* c_+.

Задача 1.21. ***(3 т.)*** *Следвайки идеята за реализацията на* `print` *можем да дефинираме* $c_I := \lambda_n n c_S c_0$.

(1) *Да се докаже, че за произволно* $n \in \mathbb{N}$ *е изпълнено* $C_I c_n \overset{\beta}{=} c_n$.

(2) *Вярно ли е, че* $C_I \overset{\beta\eta}{=} I$*? Да се докаже или да се покаже контрапример.*

Пример 1.20. Да потърсим комбинатор c_*, който представя функцията умножение над нумерали, т.е. за всеки две числа $m, n \in \mathbb{N}$ имаме $c_* c_m c_n \overset{\beta}{=} c_{mn}$. Идеята тук отново е да използваме рекурсия, както и при c_+, но този път на по-високо ниво от това. За да постигнем повторение на дадена функция mn пъти ще направим m-кратно повторение на n-кратното повторение на функцията. Разбира се, симетричната дефиниция също е възможна поради комутативността на умножението. Така можем да дефинираме

- $c_* := \lambda_{m,n,f} m(nf)$, или
- $c_* := \lambda_{m,n,f} n(mf)$.

Ако пък решим да използваме рекурсия от по-висок ред, т.е. над нумерали, бихме могли да използваме функцията c_+ и да дефинираме

- $c_* := \lambda_{m,n} m(c_+ n)c_0$, т.е. m-кратно добавяне на n, или симетрично
- $c_* := \lambda_{m,n} n(c_+ m)c_0$, т.е. n-кратно добавяне на m.

Използвайки дефиницията на c_+ като $\lambda_m m c_S$, можем да дефинираме направо

- $c_* := \lambda_{m,n} m(nc_S)c_0$, или симетрично
- $c_* := \lambda_{m,n} n(mc_S)c_0$.

Задача 1.22. ***(5 т.)*** *Да се докаже коректността на всички дадени по-горе дефиниции на* c_*.

Пример 1.21. Като още един пример за дефиниране на аритметична операция ще разгледаме операцията степенуване. Да потърсим комбинатор $c_{\exp}$ такъв, че за произволни $m, n \in \mathbb{N}$ и $m > 0$ имаме, че $c_{\exp} c_m c_n \overset{\beta}{=} c_{m^n}$. Този път е по-лесно да дадем дефиницията, в която използваме рекурсия над нумерали.

Наистина m^n можем да представим като n-кратно умножение на единицата с m. Така имаме

- $c_{\exp} := \lambda_{m,n} n(c_* m)c_1$

Нека се опитаме да дадем "директна" дефиниция на $c_{\exp}$. За целта, нека да си представим, че имаме функция f, която искаме да повторим m^n пъти. Нека започнем от самата f и да разгледаме операция g_m, която при подадено f^k, т.е. k-кратно повторение на f получава f^{km}, т.е. km-кратно повторение на f. Тогава $c_{\exp}$ би имало вида $c_{\exp} := \lambda_{m,n,f} n g_m f$, т.е. n пъти умножаваме броя на прилаганията на f по m, започвайки от само едно прилагане. Сега остава само да намерим операцията g_m. Това, което остава да съобразим е, че с m-кратно повторение на f^k всъщност ще получим f^{km}. Но m-кратно повторение на функция можем да получим точно с нумерала c_m! С други думи, операцията g_m не е нищо повече от нумерала c_m. Така окончателно можем да дефинираме

- $c_{\exp} := \lambda_{m,n,f} nmf \overset{\eta}{=} \lambda_{m,n} nm$.

Задача 1.23. ***(2 т.)*** *Да се докаже, че горните дефиниции на $c_{\exp}$ са коректни.*

Задача 1.24. ***(2 т.)*** *Да се дефинира комбинатор c_{hyp}, за който*

$$c_{\mathrm{hyp}} c_m c_n \overset{\beta}{=} c_p, \text{ където } p = \underbrace{m^{m^{\cdot^{\cdot^{m}}}}}_{n}.$$

Екстра кредит: ***(1 т.)*** *Да се направи програмна реализация.*

Като следваща стъпка, ще въведем представяне на булеви стойности, отново предложено от Church. Дефинираме

- $c_{\mathsf{tt}} := \lambda_{x,y} x \equiv K$ — булева истина
- $c_{\mathsf{ff}} := \lambda_{x,y} y \equiv K^*$ — булева лъжа

Можем да дефинираме условен оператор "if-then-else", т.е. комбинатор $c_?$, за който е изпълнено:

- $c_? c_{\mathsf{tt}} MN \overset{\beta}{=} M$
- $c_? c_{\mathsf{ff}} MN \overset{\beta}{=} N$

Лесно се вижда, че $c_? := I$ изпълнява горните условия, т.е. самите c_{tt} и c_{ff} изпълняват ролята на условния оператор.

Не е трудно да дефинираме и основните логически операции:

- $c_\neg := \lambda_p p c_{\mathsf{ff}} c_{\mathsf{tt}}$ (отрицание)
- $c_\wedge := \lambda_{p,q} pq c_{\mathsf{ff}}$ (конюнкция)
- $c_\vee := \lambda_{p,q} p c_{\mathsf{tt}} q \overset{\eta}{=} \lambda_p p c_{\mathsf{tt}}$ (дизюнкция)

Можем да дефинираме и предикати над естествени числа.

Пример 1.22.

- $c_{=0} := \lambda_m m(\lambda_p c_{\mathsf{ff}}) c_{\mathsf{tt}}$ реализира сравнение с 0
- $c_{/2} := \lambda_m m c_\neg c_{\mathsf{tt}}$ реализира проверка за четност

Има функции, които не са толкова лесни за реализация. Например, не е тривиално да се дефинира комбинатор c_P, за който $c_P c_n \overset{\beta}{=} c_{n-1}$ за $n > 0$. Не е ясно как може да "пропуснем" едно от прилаганията на функцията. Също така, не е очевидно как ще се дефинира комбинатор $c_!$, за който $c_! c_n \overset{\beta}{=} c_{n!}$. В този

случай проблемът е, че на всяка стъпка от итерацията трябва да се извършва различна операция, а именно умножение с число, което зависи от *номера* на стъпката.

Преди да покажем как се реализират тези функции, ще направим малко отклонение. Нека видим как можем да кодираме наредени двойки. Търсим тройка от комбинатори $c_{\langle\rangle}, c_{\llcorner}, c_{\lrcorner}$, съответно служещи за конструиране на наредена двойка и на намиране на нейната лява и дясна компонента. Формално, искаме да са изпълнени следните условия:

- $c_{\llcorner}(c_{\langle\rangle}MN) \overset{\beta}{=} M$
- $c_{\lrcorner}(c_{\langle\rangle}MN) \overset{\beta}{=} N$

Идеята за дефиниране на комбинаторите идва от наблюдението, че

- $\lambda_{x,y}c_{\llcorner}(c_{\langle\rangle}xy) \overset{\beta\eta}{=} K \equiv c_{\mathtt{tt}}$ и
- $\lambda_{x,y}c_{\lrcorner}(c_{\langle\rangle}xy) \overset{\beta\eta}{=} K^* \equiv c_{\mathtt{ff}}$.

Така можем да дефинираме:

- $c_{\langle\rangle} := \lambda_{x,y,z}zxy$
- $c_{\llcorner} := \lambda_p pc_{\mathtt{tt}}$
- $c_{\lrcorner} := \lambda_p pc_{\mathtt{ff}}$

С помощта на механизма на наредените двойки можем да симулираме многоаргументни функции. В частност, можем да използваме нумералите, за да правим едновременна рекурсия над няколко параметъра. Това ни позволява, например, да запазваме между отделните рекурсивни стъпки не само текущия резултат, но и номера на стъпката.

Сега вече сме готови да дефинираме комбинатора c_P, който намира предшественик на даден нумерал c_n за $n > 0$. Идеята е вместо да се опитваме да пресмятаме директно c_{n-1}, да пресметнем наредената двойка $\langle c_n, c_{n-1}\rangle$, като така поддържаме памет за "предишното число". Ако на дадена стъпка сме пресметнали двойката $z := \langle c_i, c_{i-1}\rangle$, следващата стъпка $\langle c_{i+1}, c_i\rangle$ можем да получим като $c_{\langle\rangle}(c_S(c_{\llcorner}z))(c_S(c_{\lrcorner}z))$, или — още по-просто — като $c_{\langle\rangle}(c_S(c_{\llcorner}z))(c_{\llcorner}z)$. Дясната компонента на наредената двойка ще бъде систематично "изхвърляна" до последната стъпка, когато всъщност тя ще представлява крайния резултат.

Окончателно, можем да дефинираме:

$$c_P := \lambda_n c_{\lrcorner}\big(n(\lambda_z c_{\langle\rangle}(c_S(c_{\llcorner}z))(c_{\llcorner}z))(c_{\langle\rangle}c_0c_0)\big).$$

Задача 1.25. ***(3 т.)*** *Да се докаже, че*

- $c_P c_0 \overset{\beta}{=} c_0$,
- $c_P c_n \overset{\beta}{=} c_{n-1}$ *за* $n > 0$.

Пример 1.23. С подобен подход можем да дефинираме и функция, която пресмята факториел. В този случай на всяка стъпка ще пресмятаме наредената двойка $\langle c_n, c_{n!}\rangle$.

$$c_! := \lambda_n c_{\lrcorner}\big(n(\lambda_z c_{\langle\rangle}(c_S(c_{\llcorner}z))(c_*(c_S(c_{\llcorner}z))(c_{\lrcorner}z)))(c_{\langle\rangle}c_0c_1)\big).$$

Забелязваме, че в горния пример имаме повтарящия се израз $c_S(c_{\llcorner}z)$. Удобно би било, ако можехме да направим полагане $y := c_S(c_{\llcorner}z)$. В общия случай, ако искаме да положим $y := N$ в израза M, можем да използваме β-редекса

$(\lambda_y M)N$, стига $y \notin \mathsf{FV}(M)$. С този трик, дефиницията ще изглежда по следния начин:

$$c_! := \lambda_n c_\lrcorner\big(n(\lambda_z(\lambda_y c_{\langle\rangle} y(c_* y(c_\lrcorner z)))(c_S(c_\llcorner z)))(c_{\langle\rangle} c_0 c_1)\big).$$

Задача 1.26. ***(3 т.)*** *Да се докаже коректността на една от горните дефиниции на $c_!$.*

Задача 1.27. ***(2 т.)*** *Да се дефинират комбинатори $c_=$ и $c_<$, за които за произволни $m, n \in \mathbb{N}$.*

- $c_= c_m c_n \overset{\beta}{=} c_{m=n}$
- $c_< c_m c_n \overset{\beta}{=} c_{m<n}$.

Екстра кредит: ***(1 т.)*** *Да се направи програмна реализация.*

Задача 1.28. ***(3 т.)*** *Да се дефинират комбинатори c_{quot} и c_{rem}, така че за произволни $m, n \in \mathbb{N}$:*

- $c_+(c_*(c_{\text{quot}} c_m c_n)c_n)(c_{\text{rem}} c_m c_n) \overset{\beta}{=} c_m$,
- $c_<(c_{\text{rem}} c_m c_n)c_k \overset{\beta}{=} c_{\mathtt{tt}}$.

Екстра кредит: ***(2 т.)*** *Да се направи програмна реализация.*

Задача 1.29. ***(3 т.)*** *Да се дефинират комбинатори $c_/$ и c_{prime}, така че за произволни $m, n \in \mathbb{N}$:*

- $c_/ c_m c_n \overset{\beta}{=} c_{\exists_k(km=n)}$;
- $c_{\text{prime}} c_n \overset{\beta}{=} c_{\neg\exists_{k,l>1}(kl=n)}$.

Екстра кредит: ***(2 т.)*** *Да се направи програмна реализация.*

Задача 1.30. ***(1 т.)*** *Да се дефинират комбинатори c_-, така че за произволни $m, n \in \mathbb{N}$:*

- $c_- c_m c_n \overset{\beta}{=} c_{m \dot{-} n}$, *където* $m \dot{-} n := \max(m - n, 0)$.

Екстра кредит: ***(1 т.)*** *Да се направи програмна реализация.*

Задача 1.31. ***(8 т.)*** *Да се предложи дефиниция на списъци в безтиповото λ-смятане. Да се реализират комбинатори реализиращи стандартните функции* `map`, `foldr` *и* `filter`.

Екстра кредит: ***(5 т.)*** *Да се направи програмна реализация.*

Възможността за използването на наредени двойки разшири множеството от числови функции, които можем да дефинираме. Въпреки това, все още нямаме инструментариума да дефинираме всички изчислими функции. Причината за това е, че нумералите на Church ни дават възможност само за рекурсия, чиито брой стъпки е известен предварително, така наречената *примитивна рекурсия*. Необходим ни е инструмент, с който да можем да дефинираме произволни рекурсивни функции, включително и такива, които не винаги дават резултат.

За целта ще разгледаме следната важна теорема за неподвижната точка.

Теорема 1.1 (за неподвижната точка)**.** *Нека $F \in \Lambda$ е произволен терм. Тогава съществува терм X, който е неподвижна точка на F, т.е. $FX \overset{\beta}{=} X$. Нещо повече, съществува комбинатор $Y \in \Lambda$, който намира неподвижната точка на произволен терм F, т.е. за произволен $F \in \Lambda$ имаме $F(YF) \overset{\beta}{=} YF$.*

Доказателство. Търсим комбинатор Y такъв, че $YF \overset{\beta}{\twoheadrightarrow} F(YF)$, т.е. който след прилагане над аргумент F и β-редукция не само репродуцира себе си, но и прилага аргумента си F над себе си.

Вече видяхме пример за саморепродуциращ се терм: Ω. По негово подобие разглеждаме терма $\omega_F := \lambda_x F(xx)$ и дефинираме $Y := \lambda_F \omega_F \omega_F$. Така имаме

$$YF \equiv \omega_F \omega_F \equiv (\lambda_x F(xx))\omega_F \overset{\beta}{\to} F(\omega_F \omega_F) \equiv F(YF).$$

Полагаме $X := YF$. □

Бележка 1.7 (Парадокс на Curry). Комбинаторът Y е бил измислен от Curry, обаче в друг контекст. С негова помощ той е доказал противоречивостта на λ-смятането, разглеждано като логика. Идеята на доказателството е следната. Нека дефинираме $c_\to := \lambda_{p,q} pq c_{\mathtt{tt}} \overset{\eta}{=} \lambda_p c_\vee (c_\neg p)$. Нека сега фиксираме произволен терм M и да разгледаме терма $N := \lambda_x c_\to x M$, т.е. функцията транслираща дадено условие x до истинността на твърдението "x влече M". По Теоремата за неподвижната точка можем да намерим X такова, че $X \overset{\beta}{=} c_\to X M$. Ако допуснем, че $X \overset{\beta}{=} c_{\mathtt{ff}}$, то $c_\to X M \overset{\beta}{=} c_{\mathtt{tt}}$, което е противоречие (както ще видим в следващия раздел). Ако пък $X \overset{\beta}{=} c_{\mathtt{tt}}$, тогава $c_\to X M \overset{\beta}{=} M \overset{\beta}{=} c_{\mathtt{tt}}$, което също е противоречие ако изберем $M := c_{\mathtt{ff}}$. Това означава, че $X \overset{\beta}{\neq} c_{\mathtt{tt}}$ и $X \overset{\beta}{\neq} c_{\mathtt{ff}}$, т.е. λ-смятането не може да бъде разглеждано като логика, понеже в него има термове, които нямат логическа стойност.

Бележка 1.8 (Комбинатор на Тюринг). Комбинаторът Y далеч не е единственото решение на уравнението $F(YF) = YF$. Alan Turing е предложил комбинатора $\Theta := (\lambda_{x,F}(xxF))(\lambda_{x,F} F(xxF))$, за който може лесно да се провери, че $\Theta F \overset{\beta}{\to} F(\Theta F)$. Можете ли да се сетите за друг комбинатор със същото свойство?

Задача 1.32. ***(5 т.)*** *Да се дефинира комбинатор A, който реализира функцията на Ackermann, т.е. за който*

- $A\, c_0\, c_n \overset{\beta}{=} c_{n+1}$,
- $A\, c_{m+1}\, c_0 \overset{\beta}{=} A\, c_m\, c_1$,
- $A\, c_{m+1}\, c_{n+1} \overset{\beta}{=} A\, c_m\, (A\, c_{m+1}\, c_n)$.

Екстра кредит: ***(2 т.)*** *Да се направи програмна реализация.*

Задача 1.33. ***(5 т.)*** *Да се дефинира комбинатор M, който симулира операцията "минимизация", т.е. ако t е комбинатор, за който съществува число n, такова че*

(1) $tc_n \overset{\beta}{=} c_0$

(2) $\forall_{m<n} \exists_k (tc_m \overset{\beta}{=} c_{k+1})$,

то $Mt \overset{\beta}{=} c_n$.

Екстра кредит: ***(2 т.)*** *Да се направи програмна реализация.*

Нумералите на Church ни даваха възможност да реализираме функции с рекурсия по естествените числа, а с помощта на Y можем да реализираме функции с обща, т.е. не непременно примитивна рекурсия. Като пример, нека да разгледаме рекурсивната дефиниция на числата на Fibonacci:

$$fib(n) = \begin{cases} 0, & \text{ако } n = 0, \\ 1, & \text{ако } n = 1, \\ fib(n-1) + fib(n-2), & \text{ако } n \geq 2. \end{cases}$$

В този случай за изчислението на n-тото число на Fibonacci използваме не само предшестващото го число, но и неговия предшественик. Нека разгледаме дясната страна на дефиницията на функцията fib като функция от по-висок ред (оператор) Φ приемаща като параметър произволна функция f над естествени числа:

$$\Phi(f)(n) = \begin{cases} 0, & \text{ако } n = 0, \\ 1, & \text{ако } n = 1, \\ f(n-1) + f(n-2), & \text{ако } n \geq 2. \end{cases}$$

Формално, можем да си мислим, че функцията fib е решение на уравнението $\Phi(f) = f$, т.е. fib е неподвижна точка на оператора Φ.

Задача 1.34. ***(2 т.)*** *Да се покаже, че уравнението $\Phi(f) = f$ притежава единствено решение.*

Следователно, един начин да дефинираме λ терм c_{fib} изчисляващ функцията fib е да дефинираме оператора Φ и да положим $c_{fib} := Y\Phi$.

Термът Φ може да се дефинира по следния начин:

$$\Phi := \lambda_f \lambda_n c_{=0} n c_0 \Big(c_{=0}(c_P n) c_1 \Big(c_+(f(c_P n))\big(f(c_P(c_P n))\big)\Big)\Big).$$

Да проследим една възможна редица от редукции на $c_{fib}c_2$, като подчертаваме редексите, които редуцираме.

$$
\begin{array}{rcl}
c_{fib}c_2 & \equiv & Y\Phi c_2 \\
& \equiv & \underline{(\lambda_F\omega_F\omega_F)}\Phi c_2 \\
& \overset{\beta}{\to} & \underline{(\lambda_x\Phi(xx))\omega_\Phi}c_2 \\
& \overset{\beta}{\to} & \Phi(\omega_\Phi\omega_\Phi)c_2 \\
& \equiv & \Big(\lambda_f\lambda_n c_{=0}nc_0\Big(c_{=0}(c_Pn)c_1\Big(c_+(f(c_Pn))\big(f(c_P(c_Pn))\big)\Big)\Big)\Big)(\omega_\Phi\omega_\Phi)c_2 \\
& \overset{\beta}{\to} & \Big(\lambda_n c_{=0}nc_0\Big(c_{=0}(c_Pn)c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pn))\big(\omega_\Phi\omega_\Phi(c_P(c_Pn))\big)\Big)\Big)\Big)c_2 \\
& \overset{\beta}{\to} & \underline{c_{=0}c_2}c_0\Big(c_{=0}(c_Pc_2)c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{\mathsf{ff}}c_0\Big(c_{=0}(c_Pc_2)c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{=0}(\underline{c_Pc_2})c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & \underline{c_{=0}c_1}c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{\mathsf{ff}}c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_+(\omega_\Phi\omega_\Phi(\underline{c_Pc_2}))\big(\omega_\Phi\omega_\Phi(\underline{c_P(c_Pc_2)})\big) \\
& \overset{\beta}{\twoheadrightarrow} & c_+(\omega_\Phi\omega_\Phi c_1)\big(\omega_\Phi\omega_\Phi c_0\big) \\
& \equiv & c_+(\underline{(\lambda_x\Phi(xx))\omega_\Phi}c_1)\big(\underline{(\lambda_x\Phi(xx))\omega_\Phi}c_0\big) \\
& \overset{\beta}{\to} & c_+(\Phi(\omega_\Phi\omega_\Phi)c_1)\big(\Phi(\omega_\Phi\omega_\Phi)c_0\big).
\end{array}
$$

Видяхме как сведохме изчислението на $c_{fib}c_2$ до изчислението на сумата на $c_{fib}c_1 \overset{\beta}{=} \Phi(\omega_\Phi\omega_\Phi)c_1$ и на $c_{fib}c_0 \overset{\beta}{=} \Phi(\omega_\Phi\omega_\Phi)c_0$. По-долу е показана редукцията на всеки един от тези два терма.

$$
\begin{array}{rcl}
\Phi(\omega_\Phi\omega_\Phi)c_1 & \equiv & \Big(\lambda_f\lambda_n c_{=0}nc_0\Big(c_{=0}(c_Pn)c_1\Big(c_+(f(c_Pn))\big(f(c_P(c_Pn))\big)\Big)\Big)\Big)(\omega_\Phi\omega_\Phi)c_1 \\
& \overset{\beta}{\twoheadrightarrow} & \underline{c_{=0}c_1}c_0\Big(c_{=0}(c_Pc_1)c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_1))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_1))\big)\Big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{\mathsf{ff}}c_0\Big(c_{=0}(c_Pc_1)c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_1))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_1))\big)\Big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{=0}(\underline{c_Pc_1})c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_1))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_1))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & \underline{c_{=0}c_0}c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_1))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_1))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_{\mathsf{tt}}c_1\Big(c_+(\omega_\Phi\omega_\Phi(c_Pc_2))\big(\omega_\Phi\omega_\Phi(c_P(c_Pc_2))\big)\Big) \\
& \overset{\beta}{\twoheadrightarrow} & c_1.
\end{array}
$$

$$\begin{aligned}
\Phi(\omega_\Phi\omega_\Phi)c_0 &\equiv \underline{\left(\lambda_f\lambda_n c_{=0} n c_0\left(c_{=0}(c_P n)c_1\left(c_+(f(c_P n))\left(f(c_P(c_P n))\right)\right)\right)\right)(\omega_\Phi\omega_\Phi)c_0} \\
&\overset{\beta}{\twoheadrightarrow} \underline{c_{=0}c_0 c_0\left(c_{=0}(c_P c_0)c_1\left(c_+(\omega_\Phi\omega_\Phi(c_P c_0))\left(\omega_\Phi\omega_\Phi(c_P(c_P c_0))\right)\right)\right)} \\
&\overset{\beta}{\twoheadrightarrow} \underline{c_{\mathtt{tt}}c_0\left(c_{=0}(c_P c_0)c_1\left(c_+(\omega_\Phi\omega_\Phi(c_P c_0))\left(\omega_\Phi\omega_\Phi(c_P(c_P c_0))\right)\right)\right)} \\
&\overset{\beta}{\twoheadrightarrow} c_0.
\end{aligned}$$

Така в крайна сметка получаваме

$$c_{fib}c_2 \overset{\beta}{\twoheadrightarrow} c_+(\Phi(\omega_\Phi\omega_\Phi)c_1)\left(\Phi(\omega_\Phi\omega_\Phi)c_0\right) \overset{\beta}{\twoheadrightarrow} c_+c_1c_0 \overset{\beta}{\twoheadrightarrow} c_1.$$

Добре е да се обърне внимание, че за да получим крайния резултат трябваше внимателно да подбираме реда на редукциите, които изпълняваме. Това е наложително, понеже сляпо редуциране, например, на най-вътрешния редекс би ни довело до безкрайна редица, тъй като по дефиниция $\omega_\Phi\omega_\Phi \overset{\beta}{\to} \Phi(\omega_\Phi\omega_\Phi)$. В много езици за програмиране, например Scheme, оценяването на изрази се извършва по т. нар. апликативна стратегия, наречена още "извикване по стойност" (call-by-value). При тази стратегия една функция се извиква само когато аргументите ѝ са примитивни, т.е. "стойности". В термините на λ-смятането, можем да си мислим за стойностите като нередуцируеми термове, т.е. термове в *нормална форма*. Ако искаме да редуцираме само редекси, чиито аргументи са нередуцируеми, това всъщност означава, че трябва всеки път да избираме за редукция възможно най-вътрешен редекс. За съжаление, Както отбелязахме по-горе, прилагането на тази стратегия върху терм дефиниран с помощта на комбинатора Y със сигурност няма да ни доведе до резултат. Затова не бива да ни учудва, че наивна реализация на горния терм на Scheme води до нетерминиращо оценяване:

```
(define Y
  (lambda (F)
    ((lambda (x) (F (x x)))
     (lambda (x) (F (x x))))))

(define Phi
  (lambda (f)
    (lambda (n)
      (((c=0 n) (c 0))
       (((c=0 (cp n)) (c 1))
        ((c+ (f (cp n))) (f (cp (cp n))))))))))

> (print ((Y Phi) (c 2)))
...
```

За да се справим с този проблем е да имаме механизъм, с който да "забраняваме" определени редукции. Такава възможност ни се дава от факта, че в езика Scheme, както и в повечето езици за програмиране, λ-абстракцията е примитивен (семантичен), а не съставен (синтактичен) обект. Това е така, понеже

поради съображения за ефективност е добре телата на функциите не се оценяват (редуцират) преди да им е подаден конкретен параметър. Затова, когато искаме да "отложим" дадена редукция, можем да заменим даден терм M с η-еквивалентния му терм $\lambda_x Mx$, където x е "свежа" променлива, т.е. $x \notin \mathsf{FV}(M)$. Тъй като η-еквивалентността в λ-смятането съответсва на екстенсионално равенство, термът $\lambda_x Mx$ ще има същото поведение като M. Поради правилата за оценяване в Scheme, обаче, редукциите в M няма да се изпълняват, когато той се намира зад λ.

За да избегнем безкрайното редуциране на терма Y е достатъчно да разгледаме негов вариант, в който "безкрайният цикъл" xx е скрит под λ:

$$Z := \lambda_F(\lambda_x F(\lambda_y xxy))(\lambda_x F(\lambda_y xxy)).$$

Съответната реализация на Scheme изглежда по следния начин:

```
(define Z
  (lambda (F)
    ((lambda (x) (F (lambda (y) ((x x) y))))
     (lambda (x) (F (lambda (y) ((x x) y)))))))

> (print ((Z Phi) (c 2)))
...
```

За съжаление, се оказва че блокирането на редукциите в терма Y не е достатъчно. Нека отново разгледаме внимателно редукцията на терма $c_{fib}c_2$, за да определим къде стратегията за редуциране на най-вътрешен редукт води до безкрайна редица от редукции. Можем да забележим, че по интуиция при оценяването на условията за "дъно" на рекурсията сме използвали стандартната при всички езици за програмиране стратегия за оценяване на разклонение: първо оценяваме условието и след това, в зависимост от неговата стойност, оценяваме *само единия* от двата клона за изчисление. Такава последователност на оценяване е разумна: в крайна сметка ако оценяваме и двата клона на програмата, то не бихме имали никакво разклонение на изчислителния процес. Именно по тази причина конструкциите за разклонение в Scheme (`if`, `cond`, `case`) са *специални форми*, които се оценяват по различна от стандартната апликативна стратегия. Прилагането на специална стратегия при разклонения и от изключително значение при рекурсивно дефинирани функции. Докато при функции без рекурсивно извикване оценяването на двата клона на изчисление само ще доведе до излишна работа, при рекурсивно дефинирани функции оценяването и на "дъното" и на "стъпката" довежда до изчисление, което не завършва!

За да избегнем това, ще приложим отново трика с η-експанзията: но този път върху клона на програмата, който съответства на рекурсивната стъпка. Разглеждаме следната алтернативна дефиниция на оператора Φ:

$$\Psi := \lambda_f \lambda_n c_{=0} n c_0 \Big(c_{=0}(c_P n) c_1 \Big(\lambda_y c_+ (f(c_P n)) \big(f(c_P(c_P n)) \big) y \Big) \Big).$$

Оказва се, че това вече е достатъчно, за да упътим интерпретатора Scheme към работеща последователност от редукции на c_{fib}:

```
(define Psi
  (lambda (f)
```

```
    (lambda (n)
      (((c=0 n) (c 0))
       (((c=0 (cp n)) (c 1))
        (lambda (y)
          (((c+ (f (cp n))) (f (cp (cp n)))) y)))))))

> (print ((Z Psi) (c 2)))
1
```

1.5. **Конфлуентност и нормализация.** Вече видяхме потенциала на λ-смятането като изчислителен модел и ролята на β-редукцията като изчислителна стъпка. Началото на даден изчислителен процес се дава при прилагане на функция (построена чрез λ-абстракция) над някакъв терм за аргумент. Така се образува β редекс, който може да бъде редуциран. В процеса на редуциране могат да се появят още β-редекси. Процесът продължава докато не получим терм, който не може да бъде редуциран.

Дефиниция 1.26 (Нормална форма)**.** Казваме, че термът M е в *(β-)нормална форма*, ако той не може да бъде (β-)редуциран, т.е. $\neg\exists_{N\in\Lambda} M \xrightarrow{\beta} N$. В такъв случай ще отбелязваме $M \not\xrightarrow{\beta}$.

Пример 1.24. Термовете $K, K^*, I, S, c_n, \omega$ са в нормална форма. Термът Ω не е в нормална форма, тъй като $\Omega \xrightarrow{\beta} \Omega$. Термът YF не е в нормална форма, тъй като по дефиниция $YF \xrightarrow{\beta} F(YF)$.

Дефиниция 1.27 (Нормална форма на терм)**.** Казваме, че термът N е *нормална форма на терма* M, ако $M \xrightarrow{\beta} N \not\xrightarrow{\beta}$. Ако за даден M такъв терм N не съществува, казваме че M *няма нормална форма*. Терм, който има нормална форма се нарича *нормализируем*.

Пример 1.25. c_{10} е нормална форма на терма $c_S c_9$, а c_8 е нормална форма на терма $c_3 c_2$.

В този раздел ще разгледаме следните въпроси:

(1) Има ли термове, които нямат нормална форма?
(2) Ако терм има единствена нормална форма, всяка редица от редукции ли води до нея?
(3) Има ли термове, които имат повече от една нормална форма?
(4) Има ли стратегия за редукция, която винаги води до нормална форма (ако има такава)?
(5) Можем ли алгоритмично да определим дали един терм има нормална форма?

Лесно можем да се уверим, че отговорът на въпрос 1 е отрицателен. Наистина, термът Ω не е нормализируем, понеже $\Omega \xrightarrow{\beta} \Omega$ е единствената възможна редукционна стъпка.

Възможностите за нормализация на даден терм могат да се представят нагледно в графичен вид чрез понятието "граф на редукциите".

Дефиниция 1.28 (Граф на редукциите)**.** Нека е даден терм M. *Граф на редукциите* на M наричаме графа $G(M) := \left\langle R(M), \xrightarrow[M]{\beta} \right\rangle$, където

- върхове на графа са всички редукти на M, т.е. термовете, до които може да се редуцира M: $R(M) := \left\{ N : \ M \overset{\beta}{\twoheadrightarrow} N \right\}$
- ребра на графа са всички възможни редукции между термовете във $R(M)$, т.е. $\overset{\beta}{\underset{M}{\rightarrow}} := \left\{ \langle N, P \rangle \ : \ N, P \in R(M) \wedge N \overset{\beta}{\rightarrow} P \right\}$. С други думи, $\overset{\beta}{\underset{M}{\rightarrow}}$ представлява релацията $\overset{\beta}{\rightarrow}$ ограничена до върховете на графа: $\overset{\beta}{\underset{M}{\rightarrow}} := \overset{\beta}{\rightarrow} \cap R(M)^2$.