Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Побудова інтерпретатора безтипового лямбда числення мовою програмування Haskell

Текстова частина до курсової роботи

за спеціальністю «Комп’ютерні науки та інформаційні технології» - 122

Керівник курсової роботи

доцент, кандидат наук

Проценко В. С.

____

(Підпис)

“__” __________________ 2020 року

Виконала студентка КНІТ-4

Крисан О. А.

“__” __________________ 2020 року

Київ 2020

Тема: Побудова інтерпретатора безтипового лямбда числення мовою програмування Haskell

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітки
1	Отримання теми курсової роботи	15.10.2019	
2	Пошук тематичних джерел	16.01.2020	
3	Вивчення знайденої інформаціїї про безтипове лямбда числення	15.03.2020	
4	Написання теоретичної частини курсової роботи	26.03.2020	
5	Написання практичної частини курсової роботи	10.04.2020	
6	Перевірка курсової керівником	17.04.2020	
7	Редагування та створення презентації	18.04.2020	
8	Захист курсової роботи	25.04.2020	

Студент Крисан О. А.

Керівник Проценко В. С.

“________” ____________________

Зміст

Анотація 3

Вступ 4

Роділ 1. Безтипове лямбда-числення 6

- **1.1 Основи** 6
- **1.2. Абстрактний і конкретний синтаксис** 7
- **1.3. Змінні і метазмінні** 8
- **1.4. Область видимості** 8
- **1.5. Операційна семантика** 9

Розділ 2. Програмування на мові лямбда-числення 13

- **2.1. Функції з кількома аргументами** 13
- **2.2. Булевські константи Черча** 13
- **2.3. Пари** 15
- **2.4. Числа Черча** 16
- **2.5 Розширене обчислення** 18
- **2.6 Рекурсія** 20
- **5.7 Представлення** 22
- **5.8 Синтаксис** 23

Розділ 3. Інтерпретатор безтипового лямбда-числення мовою програмування Haskell 25

- **3.1 Терми та константи** 25
- **3.2 Функція виводу** 26
- **3.3. Функція зсуву** 27
- **3.4. Функція підстановки:** 27
- **3.5. Функція обчислення:** 28

Висновки 29

Список літератури 31

Анотація

Курсова робота присвячена дослідженню безтипового лямбда числення, як базової мови програмування, а також реалізації алгоритму побудови інтерпретатора мовою Haskell.

Також детально розібрано теоретичну базу безтипового лямбда числення як з математичної точки зору, так і з точки зору мови програмування.

Вступ

Безтипове лямбда-числення слугує основою на якій базується більшість систем типів, які існують у наш час.

В середині 60-х років Пітер Ландин відмітив, що складну мову програмування можна вивчати сформулювавши для цього невелике базове числення, яке формулює самі суттєві механізми мови і доповним його набором зручних похідних конструкцій, поведінка яких описується шляхом перекладу їх на мову базового числення. В якості базової мови Ландин використав лямбда-числення, формальну систему винайдену Алонсо Черчем, де все обчислення зводиться до елементарних операції. Важливість лямбда-числення базується на тому, що його можна одночасно розглядати як просту мову програмування і як математичний об'єкт про який можна доводити певні твердження.

Лямбда-числення можна розширити декількома способами. По-перше, часто буває зручно додати особливий конкретний синтаксис для чисел, кортежів, записів тощо. Поведінку яких можна змоделювати і в базовій мові. Цікавіше додати більш складні властивості. Такі властивості досить важко додавати.

В першій частині буде розглянуто теоретичну частину безтипового лямбда-числення з математичної точки зору. Буде введено поняття терму, розглянуті ассоціативність різних операції. Також дано визначення поняттям редекса та бета-редукції. Розглянуті різні її види, та де вони використовуються.

В другій частині буде розглянуто теоретичну частину безтипового лямбда-числення як мови програмування. Введено поняття функцій від

багатьох аргументів та рекурсії. Розглянуті числа та булеві константи Черча. Введено означення підстановки.

В третій частині буде побудовано базовий інтерпретатор для лямбда-числення на основі мови програмування Haskell. Розглянуті основні принципи редукції лямбда-виразу, розібран алгоритм написання інтерпретатора.

Розділ 1. Безтипове лямбда-числення

1.1 Основи

Процедурна(або функціональна) абстракція є головною властивістю майже всіх мов програмування. Замість того, щоб переписувати одне й те саме обчислення кожен раз ми пишемо процедуру або функцію, яка робить це обчислення в загальному вигляді, в залежності від одного чи декількох іменованих параметрів, а тоді при необхідності викликаємо цю процедуру, в кожному випадку задаючи значення параметрів.

Лямбда-числення реалізує такий спосіб визначення та застосування функцій в чистому вигляді. В λ-численні все є функціями : аргументи, які функції приймають також функції і результат, який вертає функція, також функція. Синтаксис λ-числення визначає 3 вида термів:

1. Змінна x - це терм
2. Абстракція змінної x в термі t1 (λx.t1) також терм
3. Застосування терма t1 до терма t2 також терм(t1 t2)

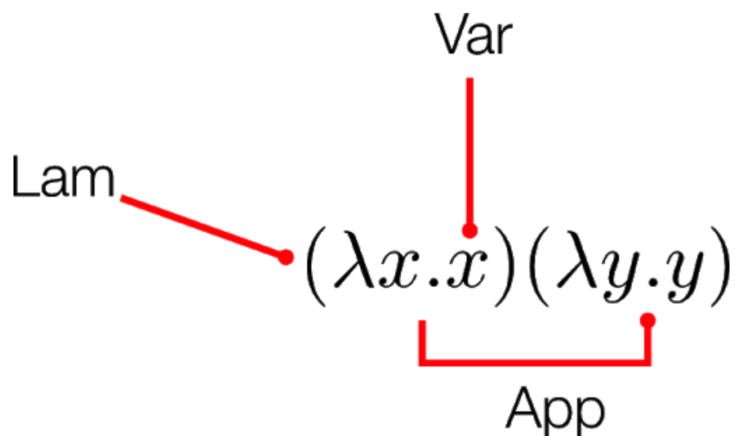

Рис. 1 Види термів

1.2. Абстрактний і конкретний синтаксис

Під час обговорення синтаксису мов програмування корисно розрізняти два рівня структури.

Конкретний (або поверхневий) синтаксис мови відноситься до рядків символів, безпосередньо які читають і пишуть програмісти. Абстрактний синтаксис - це набагато простіше внутрішнє уявлення програм у вигляді розмічених дерев (вони називаються абстрактними синтаксичними деревами або АСД). Подання у вигляді дерева робить структуру термів безпосередньо очевидною, і тому його природно використовувати для складної обробки, яка потрібна як при строгому визначенні мов (і доведенні їх властивостей), так і всередині компіляторів і інтерпретаторів.

Перетворення з конкретного в абстрактний синтаксис відбувається в два етапи. Спочатку лексичний аналізатор (або лексер) переводить послідовність символів, написаних програмістом, в послідовність лексем - ідентифікаторів, ключових слів, коментарів, символів пунктуації, і т. п. Лексичний аналізатор прибирає коментарі і вирішує питання, пов'язані з пробільними символами, угодами про заголовних/малих літер буквах, а також форматах для числових і символьних констант. Після цього синтаксичний аналізатор перетворює послідовність лексем в абстрактне синтаксичне дерево. При синтаксичному аналізі угоди про пріоритет і асоціативності операторів допомагають зменшити необхідність захаращення поверхневого уявлення програм дужками, явно вказують структуру складових виразів.

Граматики, подібні до тієї, яку було преведено для лямбда-термів, повинні розглядатися як опису дозволених видів дерев, а не послідовностей лексем або символів. Зрозуміло, коли ми будемо записувати терми в

прикладах, визначеннях, теоремах і доказах, нам доведеться висловлювати їх в конкретній, лінійного запису, але ми завжди будемо мати на увазі відповідні абстрактні синтаксичні дерева.

Щоб уникнути зайвих дужок, для запису лямбда-термів в лінійній формі будемо застосовувати дві угоди. По-перше, застосування ліво-асоціативне, тобто, s t u позначає те ж дерево, що (s t) u.

По-друге, ми абстракція забирає все до чого може дотягнутися, так що, наприклад, λx. λy. x y x означає те ж саме, що λx. (λy. ((x y) x))

1.3. Змінні і метазмінні

Ще одна тонкість в наведеному визначенні синтаксису стосується використання метазмінни. Будемо продовжувати використовувати метазмінну t (а також s і u, з індексами внизу або без них) як позначення довільного терма. Аналогічним чином, x (а також y і z) заміщає довільну змінну. Зауважимо, що тут x - це метазмінна, значеннями якої є інші змінні! На жаль, число коротких імен обмежена, і нам буде потрібно іноді використовувати x, y і т. Д. Для позначення змінних об'єктного мови. Втім, з контексту завжди буде ясно, що мається на увазі. Наприклад, у реченні «Терм λx. λy. x y має вигляд λz. s, де z = x, а

s = λy. x y »z і s - імена метазмінних, а x і y - імена змінних об'єктного мови.

1.4. Область видимості

Останнє, що нам потрібно роз'яснити в синтаксисі лямбда-числення, - область видимості змінних.

Входження змінної x називається зв'язаним, якщо воно знаходиться в тілі t абстракції λx. t. (Точніше, воно пов'язане цієї абстракцією. Ми можемо

також сказати, що λx - зв'язує визначення з областю видимості t.) Входження x вільно, якщо воно знаходиться в позиції, в якій воно не пов'язане ніякої вищерозміщеної абстракцією змінної x. Наприклад, входження x в x y і λy. x y вільні, а входження x в λx. x і λz. λx. λy. x (y z) пов'язані. В (λx. X) x перше входження x пов'язано, а другу вільну.

Терм без вільних змінних називається замкнутим; замкнуті терми називають також комбінаторами. Найпростіший комбінатор, званий функцією тотожності,

id = λx. x;

не виконує ніяких дій, а просто повертає свій аргумент.

1.5. Операційна семантика

У своїй чистій формі лямбда-числення не має ніяких вбудованих констант і елементарних операторів - ні чисел, ні арифметичних операцій, ні умовних виразів, ні записів, ні циклів, ні послідовного виконання виразів, ні введення-виведення, і т. д. Єдиний засіб для «Обчислення» термів - застосування функцій до аргументів (які самі є функціями). Кожен крок обчислення полягає в тому, що в термі-застосуванні, в якому лівий член є абстракцією, пов'язана змінна в тілі цієї абстракції замінюється на правий член. Записується це так

(λx.t12) t2 -> [x -> t2] t12

де [x -> t2] t12 означає «терм, що отримується з t12 шляхом заміни всіх вільних входжень x на t2 ». Наприклад, терм (λx. X) y за один крок обчислення переходить в y, а терм (λx. X (λx. X)) (u r) переходить в u r (λx. x). Терм виду (λx. t12) t2 називається редексом (reducible expression, «скорочує

вираз»), а операція переписування редекса відповідно до зазначеного правилом називається бета-редукцією.

Кожна стратегія визначає, які редекси в термі можуть спрацювати на наступному кроці.

При повній бета-редукції в будь-який час може спрацювати будь-який редекс. На кожному кроці ми вибираємо який-небудь редекс десь всередині обчислюється терма, і проводимо крок редукції.

Розглянемо, наприклад, терм

(λx.x) ((λx.x) (λz. (λx.x) z))

При повній бета-редукції можна, наприклад, почати з самого внутрішнього редекса, потім обробити проміжний, а потім - зовнішній:

id (id (λz. id z))

-> id (id (λz.z))

-> id (λz.z)

-> λz.z

• При стратегії нормального порядку обчислень завжди спочатку скорочується найлівіший, самий

зовнішній редекс. При такій стратегії вказаний терм оброблявся б так:

id (id (λz. id z))

->id (λz. id z)

->λz. id z

-> λz.z

При такій стратегії (а також всіх перерахованих нижче) ставлення обчислення насправді є частковою функцією: кожен терм t за крок переходить не більше ніж в один терм t1.

Стратегія виклику по імені ще більш сувора: вона не дозволяє проводити редукцію всередині абстракцій. Починаючи з того ж самого терма, перші дві редукції проводяться так само, як і при нормальному порядку обчислень, але потім зупиняються і λz. id z вважається нормальною формою:

id (id (λz. id z))

->id (λz. id z)

->λz. id z

Варіанти виклику по імені використовувалися в деяких добре відомих мовах, а саме в Алгол-60 і Haskell. У Haskell, насправді, використовується оптимізована версія, відома як виклик за потребою, в якій замість того, щоб переобчислювати аргумент при кожному використанні, при першому обчисленні все входження аргументу замінюються значенням, і таким чином зникає необхідність обчислювати його заново наступного разу. При такій стратегії потрібно підтримувати деякий поділ структур даних між уявленнями термів під час виконання - по суті, виходить ставлення редукції на графах абстрактного синтаксису, а не на деревах.

У більшості мов використовується стратегія виклику за значенням. Скорочуються тільки самі зовнішні редекси, і, крім того, редекс спрацьовує тільки в тому випадку, якщо його права частина вже зведена до значення - замкнутому терму, який вже обчислений і не може бути скорочений далі.

Стратегія виклику за значенням строга в тому сенсі, що аргументи функції завжди обчислюються, незалежно від того, використовуються вони в тілі функції чи ні. З іншого боку, несуворі (Або ліниві) стратегії обчислення - виклик на ім'я або по необхідності, - обчислюють тільки ті аргументи, які дійсно використовуються.

Розділ 2. Програмування на мові лямбда-числення

Лямбда-числення - значно потужніший формалізм, ніж здається при першому погляді на його крихітне визначення. У цьому розділі ми продемонструємо кілька стандартних прикладів програмування в цьому формалізмі.

2.1. Функції з кількома аргументами

Зауважимо для початку, що в лямбда-числення не вбудована підтримка функцій з кількома аргументами. Зрозуміло, її було б неважко додати, однак того ж самого результату простіше досягти через функції вищого порядку, які повертають функції в якості результату. Припустимо, у нас є терм s з двома вільними змінними x і y, і ми хочемо написати таку функцію f, яка видавала б для кожної пари аргументів (v, w) результат підстановки v замість x і w замість y. Ми пишемо не f = λ (x, y) .s, як ми зробили б це в більш багатому мовою, а f = λx.λy.s. Це означає, що f є функція, яка, отримавши значення v для параметра x, видає функцію, яка, отримавши значення w для параметра y, видає потрібний результат. Після цього ми по черзі застосовуємо f до аргументів, отримуючи запис f v w (т. е., (f v) w), яка переходить в ((λy. [x->s]) w), і далі в [y->w][x->v]s. Таке перетворення функцій з кількома аргументами в функції вищого порядку називається карування в честь Хаскелла Каррі.

2.2. Булевські константи Черча

Ще одна мовна конструкція, легко кодується в лямбда-численні - булевскі значення і

умовні вирази. Визначимо терми true і false таким чином:

true =λt.λf. t;

false =λt.λf. f;

Можна вважати, що терми true і false представляють булевскі значення «істина» і «брехня» в тому сенсі, що з їх допомогою ми можемо виконувати операцію перевірки булевського значення на істинність. А саме, ми можемо визначити комбінатор test, такий, що test b v w переходить в v, якщо b одно true, і в w, якщо b одно false.

test = λl. λm. λn. l m n;

Комбінатор test майже нічого не робить: test b v w просто переходить в b v w. По суті, саме булеве значення є умовним виразом: воно приймає два аргументи і вибирає з них або перший (якщо це true), або другий (якщо це false). Наприклад, терм test true v w редукується таким чином:

test true v w

= (λl.λm.λn. l m b) true v w

-> (λm.λn. true m b) v w

-> (λn. true v b) w

-> true v w

= (λt.λf. t) v w

-> (λf. v) w

-> v

Нескладно також визначити у вигляді функцій такі булеві оператори як логічна кон'юнкція:

and = λb. λc. b c false;

Тобто, and - це функція, яка, отримавши два булевских значення b і c, повертає c, якщо b = true і false, якщо b = false; таким чином, and b c видає true, якщо і b, і c рівні true, і false, якщо або b, або c - false.

Наведемо також приклад для or і not:

or = λb. λc. b true c;

not = λb. b false true;

2.3. Пари

За допомогою булевських констант ми можемо закодувати пари значень у вигляді термів:

pair =λf.λs.λb. b f s;

fst =λp. p true;

snd =λp. p false;

Це означає, що pair v w - функція, яка, будучи застосована до булевського значення b, застосовує b до v і w. За визначенням булевських констант, при такому виклику результатом буде v, якщо b дорівнює true, і w, якщо b дорівнює false, так що функції першої та другої проекції fst і snd можна отримати, просто подавши в пару відповідні булеві значення. Ось перевірка твердження snd (pair v w) ->* w:

```
   snd (pair v w)
=  snd ((λf. λs. λb. b f s) v w)
-> snd ((λs. λb. b v s) w)
-> snd (λb. b v w)
=  (λp. p false) (λb. b v w)
```

-> (λb. b v w) false

-> false v w

->* v.

2.4. Числа Черча

Подання чисел у вигляді лямбда-термів не набагато складніше, ніж те, що ми вже бачили. Числа Черча c0, c1, c2, і т. Д. Можна визначити таким чином:

c0=λs.λz. z;

c1=λs.λz. s z;

c2=λs.λz. s (s z);

c3=λs.λz. s (s (s z));

...

А саме, кожне число n представляється комбінатором cn, які приймають два аргументи, s і z («функція слідування «нуль»), і застосовує s до z n раз. Як і у випадку з булевськими константами і парами, таке кодування перетворює числа в активні сутності: число n представляється функцією, яка щось робить n раз. Якщо згадати представлення терму false то можна помітити те, що їх представлення на мові лям'бда числення співпадає, таке зустрічається досить частно в мовах асемблера, а також в низькорівневих мовах типу С, де 0 і false теж представляються однаково.

Функцію слідування на числах Черча можна визначити так:

succ = λn. λs. λz. s (n s z)

Або:

succ2=λn. λs. λz. n s(s z);

Терм succ - це комбінатор, який приймає число Черча n і повертає інше число Черча, - тобто, повертає функцію, яка приймає аргументи s і z, і багаторазово застосовує s до z. Потрібне число застосувань s до z ми отримуємо, спочатку передавши s і z в якості аргументів n, а потім явним чином застосувавши s ще раз до результату.

Аналогічним чином, додавання на числах Черча можна здійснювати термом plus, який отримує в якості аргументів два числа Черча, m і n, і повертає ще одне число Черча, тобто функцію, яка бере аргументи s і z, застосовує s к z n раз (передаючи s і z в якості аргументів n), а потім застосовує ще раз до результату:

plus = λm. λn. λs. λz. m s (n s z)

Для реалізації множення використовується ще один трюк: оскільки plus приймає аргументи по одному, застосування його до одного аргументу n дає функцію, яка додає n до будь-якого аргументу.

mult = λm. λn. m (plus n) c0

Або можна визначити так:

mult = λn . λm . λs . λz . n (m s) z

Степінь :

power=λn . λm . λs . λz . m n s z

Щоб перевірити, чи є число Черча нулем, потрібно знайти якусь пару аргументів, яка поверне цю інформацію, а точніше , потрібно застосувати число до пари термів z і s так, щоб застосування s до z один або більше разів

давало false, а відсутність застосування давало true. Зрозуміло, що в якості z потрібно просто взяти true. Для ss використаємо функцію, яка ігнорує свій аргумент і завжди повертає false:

iszero =λm. m (λx. false) true;

2.*

Приклад як можна визначити список:

nil=λhh.λtt. tt;

cons=λh.λt.λhh.λtt. hh h (t hh tt);

head=λl. l(λh.λt. h) false;

tail=λl.fst(l(λx.λp. pair(snd p)(cons x(snd p)))(pair nil nil));

isnull=λl. l (λh.λt. false) true;

2.5 Розширене обчислення

Ми впевнились, що булеві значення, числа та операції над ними можуть бути закодовані за допомогою чистого лямбда-числення. Строго кажучи, всі потрібні нам програми можна писати, не виходячи за рамки цієї системи. Однак при роботі з прикладами часто буває зручно включити до неї елементарні булеві значення і числа (а може, й інші типи даних). У випадках, коли нам потрібно абсолютно точно вказати, з якою системою ми в даний момент працюємо, для чистого лямбда-числення будемо використовувати позначення λ, а для системи, в яку додано булеві та арифметичні вирази з – позначення λNB.

В λNB по суті є дві різні реалізації булевих значень і дві реалізації чисел: справжніх та так як ми закодували вище, і можна обирати між ними при

написанні програм. Звісно, між цими двома реалізаціями нескладно написати перетворення. Щоб перевести булеве значення по Черчу в елементарне булеве значення, потрібно застосувати його до значень true` і false`:

realbool =λb. b true` false`;

Для зворотнього перетворення використовується умовний вираз:

churchbool =λb. if b then true else false;

Можемо вбудувати ці перетворення у операції вищого порядку. Ось перевірка на рівність для чисел Черча, повертаюча справжнє логічне значення:

realeq =λm.λn. (equal m n) true` false`;

Таким же чином ми можемо перетворити число Черча у відповідне елементарне число, застосувавши його до succ і 0:

realnat =λm. m (λx. succ x) 0;

Ми не можемо застосувати m до succ, так як сам по собі запис succ не має синтактичного змісту: ми визначили арифметичні вирази так, що succ завжди має застосовуватися до чогось. Цю умову ми обійдемо, обгорнув succ у маленьку функцію, котра завжди повертає succ від свого аргумента.

Причини, за якими елементарні булеві та арифметичні значення стають корисні при роботі з прикладами, в основному зв'язані з чергою обрахувань. Розглянемо, наприклад, терм succ c_1 . Виходячи із наведеного вище обговорення, ми могли б очікувати, що він має давати число Черча c_2 після обчислення. Насправді цього не відбувається:

succ c1;

-> λs.λz. s ((λs'.λz'. s' z') s z))

Цей терм містить у собі редекс, який при обчисленні привів би нас (за два кроки) до c_2 , проте згідно з правилами виклику за значенням ми не маємо на це права, оскільки редекс знаходиться всередині лямбда-абстракції.

Ніякої фундаментальної проблеми тут немає: терм, що виходить при обчисленні succ c_1 , очевидним чином поведінково еквівалентний c_2 в тому сенсі, що застосування цього терма до пари аргументів v і w завжди дасть той же результат, що і застосування c_2 до тих же аргументів. Проте залишкове обчислення ускладнює перевірку, що наша функція scc веде себе як треба. У випадку більш складних арифметичних обчислень складність ще зростає.

2.6 Рекурсія

Згадаймо, що терм, який не може просунутися далі згідно відношенню обчислення, називається нормальною формою. Цікаво, що у деяких термів немає нормальної форми. Наприклад, розбігающийся комбінатор

omega = (λx. x x) (λx. x x);

містить тільки один редекс, але крок обчислення цього редекса дає в результаті знову omega! Про терми, які не мають нормальної форми, кажуть, що вони розбігаються.

Комбінатор omega можна узагальнити до корисного терма, так званого комбінатором нерухомої точки, з допомогою якого можна визначати рекурсивні функції, наприклад, factorial.

fix =λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));

Подібно omega, комбінатор fix має складну структуру з повторами; дивлячись на визначення, важко зрозуміти, як він працює. Ймовірно,

отримати інтуїтивне уявлення про його поведінку зручніш за все, розглянувши його дію на конкретному прикладі. Припустимо, ми хочемо написати рекурсивне визначення функції вигляду h =<тіло, що містить h> – тобто, побудувати таке визначення, в якому права частина використовує ту саму функцію, яку ми визначаємо, як у визначенні факторіала. Ідея полягає в тому, щоб рекурсивне визначення «розгорталося» там, де воно зустрінеться. Наприклад, факторіалу інтуїтивно відповідає визначення:

```
if n=0 then 1
        else n * (if n-1=0 then 1
                else (n-1) * (if n-2=0 then 1
                        else (n-2) * ... ))
```

або, в термінах чисел Черча,

```
if realeq n c0 then c1
        else times n (if realeq (prd n) c0 then c1
                else times (prd n)
                        (if realeq (prd (prd n)) c0 then c1
                                else times (prd (prd n)) ...))
```

Цього можна досягти за допомогою комбінатора fix, спочатку визначивши g = <тіло, що містить f>, а потім h = fix g. Наприклад, функцію факторіала можна визначити через

```
g =λfct. λn. if realeq n c0 then c1 else (times n (fct (prd n)));
```

factorial = fix g

5.7 Представлення

Перш, ніж зайнятися формальним визначенням лямбда-числення, слід поставити ще одне, останнє питання: що, строго кажучи, означає твердження, що числа Черча представляють звичайні числа?

Щоб відповісти на це питання, згадаємо, що таке звичайні числа.

• константа 0,

• операція iszero, що відображає числа на булеві значення,

• дві операції succ і pred, що відображають числа на числа.

Кодування за Черчем подає кожен з цих елементів у вигляді лямбда-терма (тобто, функції):

• Терм c_0 подає число 0.

• Терми succ і prd представляють арифметичні операції succ і pred, в тому сенсі, що, якщо t є представленням числа n, то succ t при обчисленні подає число n + 1, а prd t подає n - 1 (або 0, якщо n дорівнює 0).

• Терм iszero представляє операцію iszero, в тому сенсі, що, якщо t є відображенням 0,то iszero t дає при обчисленні true, а якщо t є ненульове число, то iszero t дає false. Збираючи всі ці твердження разом, уявімо, що у нас є програма, яка обробляє деякі складні чисельні обчислення і видає булевий результат. Якщо ми замінимо всі числа і арифметичні операції лямбда-термами, які їх представляють, і запустимо отриману програму, ми отримаємо той самий результат. Таким чином, з точки зору впливу на

остаточні результати програм, немає ніякої різниці між справжніми числами і їх уявленнями по Черчу.

5.8 Синтаксис

Абстрактна граматика, як визначає терми повинна розглядатися як скорочений запис індуктивної певної множини абстрактних синтаксичних дерев.

Означення [Терми]: Нехай є зліченна множина імен змінних V. Множина термів – це найменша множина T , така, що:

1. $x \in T$ для всіх $x \in V$;
2. Якщо $t_1 \in T$ і $x \in V$, то $\lambda x.t_1 \in T$;
3. Якщо $t_1 \in T$ і $t_2 \in T$, то $t_1\ t_2 \in T$.

Можна дати просте індуктивне визначення множини вільних змінних, що зустрічаються в термі.

Означенн: Множина вільних змінних терма t (записується FV(t)) визначається так:

$FV(x) = \{x\}$;

$FV(\lambda x.\ t_1) = FV(t_1) \setminus \{x\}$;

$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$

Означення [Підстановка]:

$[x \to s]x \quad = s$

$[x \to s]y \quad = y$ $\quad$ *,якщо* yx

$[x \to s](y.t1) \quad = y.[x \to s]t1$ $\quad$ *,якщо* yx *і* $y\ FV(s)$

$$[x \rightarrow s](t1\ t2) \qquad = ([x \rightarrow s]t1)([x \rightarrow s]t2)$$

Розділ 3. Інтерпретатор безтипового лямбда-числення мовою програмування Haskell

Побудова інтерпретатора буде відбуватися за алгоритмом запропонованим Б. Пірсом.

3.1 Терми та константи

Спочатку необхідно визначитися з типами, потрібно визначити що таке Терм. Визначаємо за означенням. Де Var - змінна, Abs - абстракція, App - застосування терму

```
data Term  = Var Int Int
           | App Term Term
           | Abs String Term
               deriving (Eq, Show)
```

Представленням змінної є число - її індекс де Брауна. Подання абстракції містить тільки терм для тіла абстракції. Застосування містить два терми, один з яких застосовується до іншого.

Для цілей відладки корисно зберігати в кожній змінній додаткове число для перевірки цілісності. В цьому другому числовому полі завжди містить повна довжина контексту, в якому зустрілася ця змінна.

Незважаючи на те, що всередині терми представляються через індекси де Брауна, користувачеві, вони, очевидно, в такому вигляді показуватися не повинні: під час читання слід перетворювати терми з звичайного уявлення в безіменне, а під час друку перетворювати їх назад в звичайну форму . В цьому немає нічого складного, проте робити це зовсім наївним чином (скажімо, породжуючи для всіх змінних абсолютно нові імена) не варто,

оскільки тоді імена пов'язаних змінних в друкованих термах ніяк не будуть пов'язані з іменами вихідної програми. Це ускладнення можна подолати, якщо зберігати при кожній абстракції рядок-підказку з ім'ям пов'язаної змінної.

3.2 Функція виводу

Основна робота з термами (зокрема, підстановка) нічого особливого з цими рядками робити не буде: вони просто переносяться в результат в вихідної формі без будь-якої турботи про співпадіння імен, захоплення змінних тощо. Коли функції виводу потрібно створити нове ім'я для зв'язаної змінної, вона спочатку намагається використовувати підказку; якщо виявиться, що це ім'я суперечить якому-небудь імені, яяке вже використовується в поточному контексті, процедура роздруківки буде намагатися створити схоже ім'я, додаючи до імені штрихи, поки, зрештою, не знайдеться ім'я, яке не використовується в даному контексті.

Функція виводу нашого терму виглядає так:

```
printtm :: Context -> Term -> String

printtm ctx (Var x n)= indexToName ctx x

printtm ctx (Abs x t) =let (ctx',x') = pickFreshName ctx x

                         in "(l." ++ x' ++ " " ++ printtm ctx' t ++ ")"

printtm ctx (App t1 t2) = "(" ++ printtm ctx t1 ++ " " ++ printtm ctx t2 ++ ")"
```

В ній використовується контекс, список рядків і відповідних зв’язувань(які реалізовані тривіально):

```
data Binding = NameBind deriving (Show)
```

```
type Context = [(String, Binding)]
```

Функція pickFreshName - шукає ім'я, яке не використовується в даному контексті за допомогою підказки.

Функція indexToName - знаходить рядкове ім'я змінної по його індексу.

3.3. Функція зсуву

```
termShift :: Int -> Term -> Term
termShift d term= walk 0 term
  where walk c t = case t of
                Abs x t1   -> Abs x (walk (c + 1) t1)
                App t1 t2  -> App (walk c t1) (walk c t2)
                Var x n   -> if x >= c then Var (x + d) (n + d) else Var x (n + d)
```

Внутрішній зсув тут представлений викликом внутрішньої функції walk c t. Оскільки d не змінюється всередині, немає потреби передавати її в кожен виклик walk: коли це потрібно в варіанті зі змінною всередині walk, ми просто використовуємо зовнішнє зв'язування d.

3.4. Функція підстановки:

```
termSubst :: Int -> Term -> Term -> Term
```

Підстановка [j->s]t терма s замість змінної з номером j в термі t записується тут у вигляді termSubst j s t. Єдина відмінність від визначення підстановки полягає в тому, що в зсув s проводиться відразу, в гілці Var, замість того, щоб зсувати s на одиницю при кожному проході через зв'язування. При цьому виходить, що аргумент j у всіх викликах walk один і той самий, і у внутрішньому визначенні його можна опустити.

3.5. Функція обчислення:

```
eval1 :: Context -> Term -> Maybe Term

eval :: Context -> Term -> Term

eval ctx t = case eval1 ctx t of

        Nothing -> t

        Just t' -> eval ctx t'
```

Функція однокрокового обчислення прямо кодує правила обчислення, але при цьому, крім терма, приймає додатковим аргументом контекст ctx.

Було реалізовано стратегію виклику по значенню, тому у випадку (l.x x) y редукція не буде відбуватися і ми рахуємо це нормальною формою.

3.6. Приклади редукції:

Початкові дані: ctx1 = [], term1 = App (Abs "x" (Var 0 0)) (Abs "x" (Var 0 0))

*Lambda> printtm ctx1 term1

"((l.x x) (l.x x))"

*Lambda> printtm ctx1 (eval ctx1 term1)

"(l.x x)"

Початкові дані: ctx1 = [], term2 = App (App (Abs "y" (Var 0 0)) (Abs "x" (Var 0 0))) (Abs "z" (Var 0 0))

*Lambda> printtm ctx1 term2

"(((l.y y) (l.x x)) (l.z z))"

*Lambda> printtm ctx1 (eval ctx1 term2)

"(l.z z)"

Висновки

В результаті проведеної роботи було розглянуте поняття безтипового лямбда числення, як мови програмування. Дані основні означення та описані механізми роботи редукції та підстановки(програмний варіант редукції). Було запропоновано синтаксис, за допомогою якого було описано безтипове лямбда-числення.

Було розроблено досить простий застосунок мовою програмування Haskell, який інтерпретує лямбда-числення. На вхід ми отримуємо терм та контекст(для визначення імен змінних). На виході отримуємо максимально редукований терм. Якщо на вхід ми подаємо терм описаний відповідною структурою на мові Haskell, то на виході ми вже транслюєму цю структуру до звичного математичного вигляду лямбда-функцій.

Реалізовані були базові операції, такі як зсув, підстановка та обчислення. Які поступово один за одним застосовувалися до вхідного терму та в результаті було отримано потрібний результат редукції. Було реалізовано стратегію виклику по значенню.

Отримані результати показують, що навіть примітивними операціями можна отримувати досить складні функції з багатьма аргументами, які ми звикли використовувати в мовах програмування високого рівня. Лямбда-числення є повним по Тюрингу, тому будь-які програми, які реалізуються на машині Тюринга можуть бути реалізовані механізмами безтипового лямбда-числення.

Отже, безтипове-лямбда числення є не тільки потужним математичним базисом, але і потужною примітивною мовою програмування

Список використаної літератури

1. Benjamin C. Pierce, Types and Programming Languages, ISBN 0-262-16209-1. C. 31 – 51
2. Henk Barendregt, Erik Barendsen. Introduction to Lamda Calculus. Жовтень 1994, с 6-9
3. В. А. Башкин. Лямбда-исчисление, ЯрГУ 2018. С. 6-20
4. Harold Abelson and Gerald Sussman. Structure and Interpretation of Computer Programs. The MIT Press, New York, 1985. C 7-15