Lambda Calculus

Programming Languages

Haritz Puerto-San-Roman
Univeristy of Malaga
haritzpuerto94@gmail.com

Felipe Sulser-Larraz
Univeristy of Malaga
felipesulser@gmail.com

ABSTRACT

This document is the report created to support the presentation about λ calculus. In this document, we will proceed to present untyped λ calculus in a simple way. Afterwards, several examples are shown in order to show how reductions are made. Finally several problems and their solutions are presented in an incremental fashion.

With this document we do not wish to explain λ-calculus in a deep and thorough way but in a more simple and exercise-driven approach.

1. INTRODUCTION

λ calculus was invented by Church in 1928 and was first published in 1932. It is a formal system designed to investigate functions and recursion, i.e. the foundations of mathematics. The original system was shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser who developed the Kleene–Rosser paradox. In 1936 Church published just the portion relevant to computation, what is now called the untyped lambda calculus. In 1940, he also introduced a computationally weaker, but logically consistent system, known as the simply typed lambda calculus.

The syntax of λ calculus is very simple:

$$e ::= x \mid \lambda x.e \mid ee$$

Being:

- x a variable
- λ x . e is a λ abstraction (function). x is the argument and e is the body.
- ee is a λ application.

We call the set of all λ-terms Λ.

Example

Which of these expressions are right?

- λ (x.x)
- (λ(x. y)
- λx.(xy)
- (λx.x)y

The last two expressions are right.

Once we know its syntax, we are ready to learn its rules. We will start with the beta rule.

1.1 Beta rule

This reduction describes the function application rule.

$$(\lambda x.e1)e2- > e1[x/e2]$$

e2 is passed to the function.

Example
Assume sqr and 3 are defined:
$(((\lambda f.(\lambda x.(f(f(x))))sqr)3)$
$= ((\lambda x.(sqr(sqr(x)))3)$
$= (sqr(sqr\ 3))$
$= (sqr\ 9)$
$= 81$
Example
$(\lambda x.(\lambda z(xz)))y$
$= \lambda z.(yz)$

Example A currified function $(\lambda x.\lambda y.xy)z\ \lambda y.zy \rightarrow$ this is like currying in haskell!

λ-calculus uses static scoping:
Question:
Does $(\lambda x.x(\lambda x.x))z$ equals to $(\lambda x.x(\lambda y.y))z$?
Yes, both x's are bound to a λ. x can be anything.

1.2 Alfa conversion:

This is simply renaming a bound variable.

$$\lambda v.E \rightarrow \lambda w.E[v \rightarrow w]$$

This is means: $\lambda x.x = \lambda y.y$

Example
$\lambda y.(\lambda f.fx)y \xrightarrow{a} \lambda z.(\lambda f.fx)z \xrightarrow{a} \lambda z.(\lambda g.gx)z$

1.3 Eta-Conversion

An η-conversion is adding or dropping of abstraction over a function. Let's see it with an example:
If x does not appear in f, then
(λ x. f x) g = f g
Extensive use of η-reduction can lead to Pointfree programming

1.4 Pointfree Programming

It is very common for functional programmers to write functions as a composition of other functions, never mentioning the actual arguments they will be applied to. For example, compare:

sum = foldr (+) 0

with:

sum' xs = foldr (+) 0 xs

1.5 Conventions

Looking at λ terms can be very hard to decipher, so we will omit outer parenthesis whenever possible and we will use association to the left.

$$(\lambda x.(\lambda y.yx)) = \lambda x.\lambda y.yx$$

And instead of that, we will write:

$$\lambda xy.yx$$

Let's see a full example:

$$(\lambda x.xy)(\lambda z.z)w = (\lambda z.z)yw = yw$$

1.6 Combinators

A λ-term M is a called a combinator if $FV(M) = \emptyset$. The following λ-terms are examples of combinators.

- $I = \lambda x.x$
- $K = \lambda xy.x$
- $S = \lambda xyz.xz(yz)$
- $\omega = \lambda x.xx$
- $\Omega = \omega\omega$
- $Y = \lambda f.(\omega(\lambda x.f(xx)))$

1.7 Beta normal form

We say that a term λ is in β normal form if it cannot be β-reduced. A term has a β normal form if it β reduces to a term that has a β normal form.

I is in β-nf. Ω does not have a β-nf.

$KI\Omega$ not in β-nf but it has one, namely I.

1.8 Logical values, Tuples and numbers

Let's define true and false values:

$$\text{T} = \lambda tf.t$$
$$\text{F} = \lambda tf.f$$

T is a function that takes 2 arguments and returns the first one.
F is a function that takes 2 arguments and returns the last one.

Let's see an example of T and F
if T then e1 else e2 = T e1 e2 = ($\lambda tf.t$) e1 e2 = e1
if F then e1 else e2 = F e1 e2 = ($\lambda tf.f$) e1 e2 = e2

Now, let's define AND, OR and NOT.

AND = λ xy . xyF
OR = λ xy . xTy
NOT = λ x . xFT

Let's see some examples of these definitions:

Assume e1 = T
AND e1 e2 => e1 e2 F => T e2 F => e2
OR e1 e2 => e1 T e2 => T T e2 => T
NOT e1 => e1 F T => T F T => F

Assume e1 = F
AND e1e2 => e1 e2 F => F e2 F => F
OR e1 e2 => e1 T e2 => F T e2 => e2
NOT e1 => e1 F T => F F T = T

How can we represent a tuple?
pair = λxyb . b x y
fst = λp. p T
snd = λp. p F

Let's see how it works:

fst (pair e1 e2) => (pair e1 e2) T => (λb . b e1 e2) T => T e1 e2 => e1

There are many ways to represent numbers using λ calculus. We will use the following:
0 = λ f x . x
1 = λ f x . fx
2 = λ f x . f(fx)
n = λ f x . f...f(fx) There are n 'f'

Now let's define the successor function:
SUCC := λ nfx.f (n f x)

Let's see some examples:

SUCC 0 = SUCC (λ f x. x) = (λn f x. f (n f x)) (λf x. x) = λf x . f ((λf x. x) f x)) = λ f x . f(x) = 1
SUCC 1 = SUCC (λfx.fx) = (λn f x. f (n f x)) (λf x.fx) = λf x . f((λf x.fx) f x) = λf x.f (f x) = 2

Once we have the successor function we want the predecessor function. In order to define it we need the auxiliary function next.
next = λp . pair (snd p) (add (snd p) 1)
next (pair a b) = pair b (b+1)

It can be shown that by applying next to pair 0 0 exactly n times, we obtain pair (n-1) n

$$\text{PRED} := \lambda\text{n . fst } (next^n \text{ (pair 0 0))}$$

PRED 1 = (λn . fst ($next^n$ (pair 0 0))) 1 = fst (pair 0 1) = 0

Later on, we will need another function related with numbers. This function is called zero?. It tells us if a number is zero or not.

$$\text{zero?} = \lambda\text{nxy . n } (\lambda\text{z.y) x}$$

Let's see that it does work.

Zero? 0 = (λnxy . n (λz.y) x) (λfx . x) = λxy. (λfx. x) (λz.y) x = λxy. x = T
Zero? 1 = (λnxy . n (λz.y) x) (λfx . fx) = λxy. (λfx. fx) (λz.y) x = λxy. (λz.y) x = λxy. y = F

2. FIXED POINTS

2.1 Theorem. Fixed points exists.

For every M $\in \Lambda$ there exists X $\in \Lambda$ such that M X = X, that is X is a fixed point of M .

We claim that YM is a fixed point of M.

YM = (λx . M (xx))(λx . M (xx))
= M ((λx . M (xx))(λx . M (xx)))
= M (YM)

Let's see an application.

$$add\ n\ m = \begin{cases} m & if \quad n = 0 \\ add(n-1)(m+1) & otherwise \end{cases}$$

ADD = λxy. (Zero? x) (y) (ADD (Pred x) (Succ y))

There is a problem here. In the definition of add we are referencing add so let's abstract out add.

ADD = λpxy. (Zero? x) (y) (p (Pred x) (Succ y)) ADD
Q = λpxy. (Zero? x) (y) (p (Pred x) (Succ y))
ADD is a fixed point of Q
ADD = YQ

Now, ADD is not used in its definition.

Let's check its behaviour:

ADD n m = YQ n m = Q(YQ) n m = Q (ADD) n m = (Zero? n) (m) (ADD (Pred n) (Succ m))

Now, let's see a few more theorems.

2.2 Godel Numbering

There exists an effect enumeration of λ-terms. For M $\in \lambda$ we write #M to denote the Godel number of M . We write [#M] to stand for the λ-term representing #M .

2.3 Another important theorem

For every λ-term F there is a λ-term X such that F [#X] = X.
Proof
All recursive functions are λ-definable by the Church-Turing Thesis. By the effectiveness of our numbering, there is a term N such that:
N[#M] = [#[#M]]
Furthermore, there is a term A such that
A[#M][#N] = [#(M N)]

Now, let's take W = λn. F(An (N n))

X = W[#W] = F(A [#W](N[#W])) = F(A[#W]([#[#M]])) = F([#(W[#W])]) = F([#X])

3. DECISION PROBLEM

Alonzo Church proved that there is no term that decides whether two terms have the same normal form.

He reduced this problem to asking whether a given term has a normal form, and then showed this problem can't be answered using a λ-term. We will only show this proof.

Theorem. There is no lambda term, M, such that

$$M\ n = \begin{cases} 0 & if\ Godel\ number\ n\ has\ a\ \beta nf \\ 1 & otherwise \end{cases}$$

Proof
Let's suppose there is such M.
Let's define G = λ n. Zero?(M n) Ω I
As we shown before, there is an X such that:
G[#X] = X

Let's suppose x has a β-nf.
M[#X] = 0 => G[#X] = Zero? (0) ΩI = Ω = X => X has no β-nf.
We have reached a contradiction!
Let's suppose x has no a β-nf.
M[#X] = 1 => G[#X] = Zero? (1) ΩI = I = X => X has β-nf.
We have reached a contradiction!
We can conclude that there is no such M.

4. TURING COMPLETENESS

We will show the equivalence to μ-recursive functions.

Constant function: $f(x_1, ...x_k) = n$
Successor function: $S(x) = f(x) = x + 1$
Projection function: $P(i, k) = f(x_1, ...x_k) = x_i$

4.1 Operators

1. Composition operator
2. Primitive recursion
3. Minimalisation

For this demonstration I will assume that the reader knows the concepts of the operators in the μ-recursive language.

If this is not the case, please read the μ-recursive functions chapter from the book Automata Theory and Formal Languages.

Constant function is straightforward and we have already given a definition of successor funtion.
And the Projection function?
Projection: $f(x_1, ..., x_k) = x_i$
In lambda terms:
$\lambda x_1, ..., x_k.x_i$

4.2 Operators

Composition

Definition 9.5: *Composition operation*

Given $m > 0$, $k \geq 0$ and the functions:

$$g : \mathbf{N}^m \rightarrow \mathbf{N}$$

$$h_1, h_2, ..., h_m : \mathbf{N}^k \rightarrow \mathbf{N}$$

If the Function f: $\mathrm{N}^k \rightarrow \mathrm{N}$ is:

$$f(\underline{n}) = g(h_1(\underline{n}), h_2(\underline{n}), ..., h_m(\underline{n}))$$

then we say **that f is the composition of g *with* $h_1, h_2, ..., h_m$.**

We will denote $f(\underline{n}) = g(h_1, h_2, ..., h_m)(\underline{n})$, or simply $f = g(h_1, h_2, ..., h_m)$.

Composition in λ calculus
In terms of lambda calculus:
$\lambda gh1h2...hmn1n2...nk.$
$g(h1\ n1\ n2...nk)(h2\ n1....nk)...(hm\ n1\ n2...nk)$

Primitive recursion

Definition 9.6: *Recursively defined function*

Given $k \geq 0,$ and the functions:

$g : \mathrm{N}^k \rightarrow \mathrm{N}$
$h : \mathrm{N}^{k+2} \rightarrow \mathrm{N}$

The Function f: $\mathrm{N}^{k+1} \rightarrow \mathrm{N}$ such that is defined as:

$$f(\underline{n},m) = \begin{cases} g(\underline{n}) & \text{if } m = 0 \\ h(\underline{n},m-1,f(\underline{n},m-1)) & \text{if } m > 0 \end{cases}$$

we say that **f is defined recursively by g and h.**

We will denote it as $f(\underline{x}) = \langle g / h \rangle(\underline{x})$, or simply $f = \langle g / h \rangle$.

Primitive Recursion in λ calculus
We will use the previously explained Y combinator.
$\lambda g\ h\ n1\ n2...nk.$
$Y(\lambda fm.iszero\, m(f\, n1\, n2...nk)\ (g\, n1\, n2...nk)(prec\, m)(f(prec\, m))))$

Minimalisation

Definition 9.7: *Unbounded Minimalization*

Given $k \geq 0$ and the Function:
$g : \mathrm{N}^{k+1} \rightarrow \mathrm{N}$

If the Function f: $\mathrm{N}^k \rightarrow \mathrm{N}$ is:

$$f(\underline{n}) = \begin{cases} minimum(A) & if\ A \neq \varnothing \quad \wedge \quad \forall t \leq minimum(A) \quad g(\underline{n},t) \in \mathrm{N} \\ \uparrow & \text{otherwise} \end{cases}$$

where $A = \{ t \in \mathrm{N} \ / \ g(\underline{n},t) = 0 \}$ and $\underline{n} \in \mathrm{N}^k$

then we say that f is obtained from g by unbounded minimalization.
We will denote it as $f(\underline{n}) = \mu[g](\underline{n})$, or simply $f = \mu[g]$.

Note: The symbol "$\uparrow$" means that the Function, for that input vector ($\underline{n}$), verifies that: $\underline{n} \notin Dom(f)$. That is, the Function diverges ("it is not defined") for that input .

Minimalisation in λ calculus
Again, we'll use the Y combinator.
$\lambda g\ n1\ n2...nk.$
$(Y.(\lambda h\ x.zero?(g\ x1\ x2...xk\ x)x(h(succ\ x)))zero)$

We have just proven that λ-calculus is at least as powerful as μ recursive functions. In order to prove that it is turing complete, we must show that λ-calculus is not more powerful than μ recursive functions.

5. CONCLUSIONS

We have provided:

1. Syntax definition
2. Rules of derivation and conversion
3. Simple data structures and Church's encoding
4. Recursion in Lambda Calculus
5. Decision Problem in Lambda Calculus
6. Equivalence for the Turing completeness in Lambda Calculus

6. REFERENCES

[1] Gunther 2010, *Lambda Calculus and the Decision Problem*
[2] Roger Hindley and Jonathan Seldin, *Introduction to Combinators and Lambda calculus*
[3] Prof. Sungwoo Park (POSTECH), *CSE-321 Programming Languages, Untyped Lambda Calculus*
[4] Raul Rojas (FU Berlin), *A Tutorial Introduction to the Lambda Calculus*
[5] Ramos-Jimenez, G. (UMA) *Automata Theory and Formal Languages*