Semester Project Report

Lambda Calculus and Turing Machine Equivalence

Author:

Anurag Sharma

Indian Institute of Technology Kanpur.

Supervisor:

Prof. Mohua Banerjee

Indian Institute of Technology, Kanpur

April 20, 2017

Contents

1 Introduction

1.1 Motivation of the Project

Church developed the lambda calculus in the 1930s as a theory of functions that provides rules for manipulating functions in a purely syntactic manner. His motivation to develop a theory of functions came from the Hilberts Entscheidungsproblem challenge.

In 1936 motivated by the same problem Alan Turing published his paper, - "On computable numbers with an application to the Entscheidungsproblem". In this paper he introduces Turing machines and gave a different definition of the computable functions. [3] Turing also proved that the expressive power of Church's Lambda Calculus and his own Turing Machines is essentially same. Both of the theories take a very different view on the "computability" of functions but capture the same class of functions. The goal of this project is to understand the equivalence between Church's λ-calculus and Turing Machines.

1.2 What is a function?

The study of lambda calculus should start with understanding two different ways of viewing functions in modern mathematics. One of the common way is to see function as a graph which has a domain X and a codomain Y. Here a function $f : X \to Y$ is a set of tuples so that for every $x \in X$, there exactly one $y \in Y$ such that $(x, y) \in f$. Two functions are equal if they give same output for the same input irrespective of their mathematical formulation. This is called the *extensional* view of functions.

A different view of functions is to see them as mathematical formula, something which can be computed using pen and paper. Here the functions are understood as rules. Equality of function in this viewpoint becomes exact similarity between their mathematical description. This is called *intensional* view of the functions [2].

2 The lambda calculus

The lambda calculus is a theory of functions as formulas. The main ideas are applying a function to an argument and forming functions by abstraction. The syntax of basic λ-calculus is sparse which makes it an elegant and focused notation for representing functions.

Functions and arguments are considered at same level in λ-calculus. This results intensional theory of functions as rules of computation, contrasting with an extensional theory of functions as sets of ordered pairs.

Compared to a Turing machine, lambda calculus does not care about the details of the machine evaluating it. The lambda calculus in its pure form is untyped and does not concern itself about the types of expressions at all. Computation in λ-calculus is all about computation in the form of variable substitution.

In the next few sections we will study about the syntax and reduction rules of λ-calculus and build enough tools to understand the equivalence between λ-calculus and Turing machine computability.

3 The untyped lambda calculus

3.1 Syntax

The objects of study in the λ-calculus are the terms of the language, and ideas of convertibility or equality between pairs of these terms.
As a starting point, we assume the existence of a countable set of variables.

Definition. Under the assumption of infinite (but countable) set V of variables denoted by $x, y, z \ldots$, the set of lambda terms is given by the following Backus-Naur Form[1]:

$$\text{Lambda terms}: \quad M, N ::= x \,|\, (MN) \,|\, (\lambda x.M)$$

A term of the untyped λ-calculus is finite string made up of the symbols "λ","(",")",","." and variable identifiers from V. Let Λ^* be the set of strings (finite sequences) made using the the special symbols and variables of the language. The set of lambda terms is the smallest subset $\Lambda \subseteq \Lambda^*$ such that:

- Whenever $x \in V$ then $x \in \Lambda$
- Whenever $M, N \in \Lambda$ then $(MN) \in \Lambda$
- Whenever $x \in V$ and $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$

$\lambda x.M$ is called *abstraction* of M and (MN) is called *application* of M on N

Convention. 1. Omit the outermost parenthesis. Example: (MN) should be written as MN. 2. Applications is left associative. Example: MNP means $(MN)P$. 3. The body of abstraction (i.e the scope of λ (lambda) after the . (dot)) extends as much to the right as possible. 4. When writing multiple abstractions, they can be contracted. Example: $\lambda x.\lambda y.\lambda z.$ can be written as $\lambda.xyz$ in condensed form.

3.2 Free and Bound variables

In lambda calculus, in the function definition $\lambda x.x$ the variable x in the body of the definition (the second x) is bound because its first occurrence in the definition is λx. A variable that is not bound in a lambda term is said to be free in that term. For example in the function $(\lambda x.xy)$, the variable x in the body of the function is bound and the variable y is free. Every variable in a lambda expression is either bound or free. Bound and free variables have quite a different status in lambda terms.

3.2.1 Examples

Consider the expression $(\lambda x.x)(\lambda y.xy)$. The variable x in the body of the leftmost expression is bound to the first lambda. The variable y in the body of the second expression is bound to the second lambda but the variable x in the second lambda term is free. The second x is independent of the first x.

Now consider the expression $(\lambda x.xy)(\lambda y.yz)$. In this expression the free variables are y and z. Although y is bound in the second lambda term but since it is free in the first laambda term therefore we consider it free for the term as a whole.

3.2.2 Formal definition of Free variables

The set of free variables (FV) of a term M is defined such that:

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

$$FV(MN) = FV(M) \cup FV(N)$$

$$FV(\lambda x.M) = FV(M) \backslash \{x\}$$

3.3 α equivalence

Before we discuss α equivalence, we should first understand the rules of renaming the variables in λ calculus.

3.3.1 Renaming

If x and y are the variables in the lambda term M, we define $M\{y/x\}$ as the resulting term of renaming x as y in the lambda term M. Following rules apply for renaming the variables in λ-calculus:

$$\begin{aligned} x\{y/x\}) &\equiv y \\ z\{y/x\}) &\equiv z \quad \text{if} \quad z \neq x \\ (MN)\{y/x\} &\equiv (M\{y/x\})(N\{y/x\}) \\ (\lambda x.M)\{y/x\} &\equiv \lambda y.(M\{y/x\}) \\ (\lambda z.M)\{y/x\} &\equiv \lambda z.(M\{y/x\}) \quad \text{if} \quad z \neq x \end{aligned}$$

We are now in a position to define α equivalence class.

Definition. Two terms M and N are said to be α-convertible if one can derive the same term from both purely by renaming bound variables to new variables. If two terms are α-convertible we also write them as $M \equiv_\alpha N$. These terms are understood to be identical at the syntactic level.

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\}) \quad \text{for all variables } y \text{ that do not occur in } M$$

3.4 Substitution

In the renaming operation just defined, we replace a variable by another variable in a lambda term. Now we turn to a different operation called *substitution*. This allows us to replace a variable by a lambda term while unrolling the abstraction. Let $M[N/x]$ denote the result of substituting N for

x in M. Following rules guide the substitution for the lambda terms:

$$\begin{aligned}
x[N/x] &\equiv N \\
y[N/x] &\equiv y \quad \text{if} \quad y \neq x \\
(MP)[N/x] &\equiv (M[N/x])(P[N/x]) \\
(\lambda x.M)[N/x] &\equiv \lambda x.M \\
(\lambda y.M)[N/x] &\equiv \lambda y.(M[N/x]), \quad \text{if } x \neq y \text{ and } y \notin FV(N) \\
(\lambda y.M)[N/x] &\equiv \lambda z.(M\{z/y\}[N/x]), \quad \text{if } x \neq y,\ y \in FV(N), \text{ and } z \text{ is fresh.}
\end{aligned}$$

Remarks. The above rules are a bit complicated than the rules of renaming because of the following reasons: 1. We have to only replace free variables. The names of bound variables are immaterial, and should not affect the result of a substitution. 2. We have to avoid unintended capture of free variables.

4 $\beta -$ reduction

4.1 Introduction

Evaluating lambda terms by plugging arguments into functions in abstraction terms is called β-reduction. A term of the form $(\lambda x.M)N$ where abstraction is applied to another lambda term is called β**-redux**. The term to which it reduces that is $M[N/x]$ is called β**-reduct**. A lambda term without any β-reduxes is said to be in *β-normal form.*

Definition. The notion of reduction called β-reduction is the relation β which is the set [6]

$$\{((\lambda x.M)N, M[N/x]) \mid M, N \in \Lambda\}$$

The one step β reduction ($\rightarrow_\beta$) is the compatible closure of this relation i.e it satisfies following:

$$(\beta) \qquad \frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]}$$

$$(\mathbf{cong}_1) \qquad \frac{M \rightarrow_\beta M^{'}}{MN \rightarrow_\beta M^{'}N}$$

$$(\mathbf{cong}_2) \qquad \frac{N \rightarrow_\beta N^{'}}{MN \rightarrow_\beta MN^{'}}$$

$$(\zeta) \qquad \frac{M \rightarrow_\beta M^{'}}{(\lambda x.M) \rightarrow_\beta (\lambda x.M^{'})}$$

Definition. We write $M \twoheadrightarrow_\beta N$ if M reduces to N in zero or more steps i.e $\twoheadrightarrow_\beta$ is defined to be the reflexive transitive closure of $\rightarrow_\beta$.[2]

4.2 β equivalence

Finally we define β equivalence by allowing reduction steps and inverse reduction steps i.e by making $\rightarrow_\beta$ symmetric. Formally β equivalence is defined to be the reflexive, symmetric and transitive closure of $\rightarrow_\beta$, i.e, the smallest equivalence relation containing $\rightarrow_\beta$ relation defined above.

5 A Fixed-Point Theorem

The ability to apply a term to itself leads to recursive properties of the λ-calculus. We shall see one of these properties now. Suppose f is a function. We say that f has a fixed point x if $f(x) = x$. For example $f(x) = x^2$ has 0 and 1 as fixed points. Similarly every point in the domain of $f(x) = x$ is a fixed point of f whereas $f(x) = x + 1$ has no fixed point.

Theorem. In the untyped lambda calculus, every term F has a fixed point.
Proof[2]. Suppose $A = \lambda xy.y(xxy)$, and define $\Theta = AA$. If F is any lambda term then $N(= \Theta F)$ is a fixed point of F. This is claim follows from the following calculation:

$$\begin{aligned} N &= \Theta F \\ &= AAF \\ &= (\lambda xy.y(xxy))AF \\ &\twoheadrightarrow_\beta F(AAF) \\ &= F(\Theta F) \\ &= FN \end{aligned}$$

The term $\Theta = AA$ used is called Turing's fixed point combinator. Another well known fixed point combinator is *Curry's "paradoxical combinator*: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. The fixed point in λ-calculus allow us to solve equations. The above theorem tells us that we can always solve such equations, which have arbitrary right hand side and x as left hand side, in lambda calculus.

6 Computation with Lambda Calculus

We now show that with minimal syntax, the λ-calculus is a powerful language. We start with numbers, and numbers arent part of the syntax. Therefore we show that numbers, and functions, can be encoded by terms and that as much power as one can expect from a programming language is available to compute with these numbers (principally because of the existence of fixed point combinators).

For each natural number n, we define a lambda term $\overline{n}$, called the n^{th} *Church numeral*, as $\overline{n} = \lambda fx.f^n x$. The first few Church numerals look like this:

$$\begin{aligned}\overline{0} &= \lambda fx.x \\ \overline{1} &= \lambda fx.fx \\ \overline{2} &= \lambda fx.f(fx) \\ &\dots\end{aligned}$$

Now using these as our "numbers" we can define basic arithmetic operations on them again by using just lambda terms.

6.1 Example of basic arithmetic functions

6.1.1 Successor function

succ $= \lambda nfx.f(nfx)$
Applying it to $\overline{n}$

$$\begin{aligned}\textbf{succ}\ \overline{n} &= (\lambda nfx.f(nfx))(\lambda fx.f^n x) \\ &\to_\beta \lambda fx.f((\lambda fx.f^n x)fx) \\ &\twoheadrightarrow_\beta \lambda fx.f(f^n x) \\ &= \lambda fx.f(f^{n+1}x) \\ &= \overline{n+1}\end{aligned}$$

6.1.2 Addition function

add $= \lambda nmfx.nf(mfx)$

Applying it to $\overline{1}$ and $\overline{2}$

$$\begin{aligned}
\overline{1} &= \lambda fx.fx \\
\overline{2} &= \lambda fx.f(fx) \\
\textbf{add}\ \overline{1}\ \overline{2} &= \lambda nmfx.nf(mfx)\overline{1}\ \overline{2} \\
&\to_\beta \lambda fx.\overline{1}f(\overline{2}fx) \\
&\to_\beta \lambda fx.(\lambda fx.fx)f((\lambda fx.ffx)fx) \\
&\twoheadrightarrow_\beta \lambda fx.(\lambda fx.fx)ffffx) \\
&\to_\beta \lambda fx.fffx \\
&= \overline{3}
\end{aligned}$$

6.1.3 Multiplication function

mult $= \lambda nmf.n(mf)$

Applying it to $\overline{1}$ and $\overline{2}$

$$\begin{aligned}
\overline{1} &= \lambda fx.fx \\
\overline{2} &= \lambda fx.f(fx) \\
\textbf{mult}\ \overline{1}\ \overline{2} &= \lambda nmf.n(mf)\overline{1}\ \overline{2} \\
&\to_\beta \lambda mf.\overline{1}(mf)\overline{2} \\
&\to_\beta \lambda f.\overline{1}(\overline{2}f) \\
&\to_\beta \lambda f.(\lambda fx.fx)((\lambda fx.ffx)f) \\
&\to_\beta \lambda f.(\lambda fx.fx)(\lambda x.ffx) \\
&\twoheadrightarrow_\beta \lambda f.(\lambda x.ffx) \\
&= \overline{2}
\end{aligned}$$

6.1.4 Boolean functions

Here we define two lambda terms which will encode the truth values "true" and "false".

TRUE $= \lambda xy.x$

FALSE $= \lambda xy.y$

and $= \lambda ab.ab$**FALSE**

Applying **and** to **TRUE** and **TRUE**

$$\begin{aligned}
\textbf{TRUE} &= \lambda xy.x \\
\textbf{and TRUE TRUE} &= (((\lambda ab.ab\textbf{FALSE})\textbf{TRUE})\textbf{TRUE}) \\
&\twoheadrightarrow_\beta (\lambda b.\textbf{TRUE}\ b\ \textbf{FALSE})\textbf{TRUE} \\
&\rightarrow_\beta \textbf{TRUE TRUE FALSE} \\
&= (\textbf{TRUE TRUE})\textbf{FALSE} \\
&= ((\lambda xy.x)(\lambda xy.x))\textbf{FALSE} \\
&\twoheadrightarrow_\beta (\lambda y.\textbf{TRUE})\textbf{FALSE} \\
&\rightarrow_\beta \textbf{TRUE}
\end{aligned}$$

Similarly if we apply **and** to other combinations we get the desired results. Following can be verified easily:

$$\textbf{and TRUE FALSE} \twoheadrightarrow_\beta \textbf{FALSE}$$

$$\textbf{and FALSE FALSE} \twoheadrightarrow_\beta \textbf{FALSE}$$

$$\textbf{and FALSE TRUE} \twoheadrightarrow_\beta \textbf{FALSE}$$

On the similar note we see some other conditionals defined in λ-calculus. These conditionals are further used to build programs in the language to do manipulations like calculation factorial, computing fibonacci sequences etc.

or $= \lambda ab.aab$

if_then_else $= \lambda x.x$

not $= \lambda pab.pba$

7 Turing and his machines

In the previous sections we discussed λ-calculus and familiarised ourselves with the basic operations of defining functions and evaluating lambda terms. Now we will focus on a different structure which Turing developed to tackle Hilbert's challenge.

7.1 Entscheidungsproblem (decision problem)

In 1928 David Hilbert posed a challenge:

Hilberts Entscheidungsproblem [3]

Is there an algorithm which when fed any statement in the formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peanos axioms for arithmetic, using the usual rules of first-order logic?

It is important to note the usefulness of such an algorithm because using it we can answer questions like

$$\forall k > 1 \; \exists \, p \; , \; q \; ((2k = p + q) \wedge prime(p) \wedge prime(q))$$

(where prime(p) is a suitable arithmetic statement that p is a prime number). This is Goldbach's Conjecture [5] (every strictly positive even number is the sum of two primes).

A few years later it was proved that such an algorithm can not exist by the work of Church and Turing in 1935-36. While working towards the proof of Entscheidungsproblem both Turing and Church developed a different notion to define "computable" function. It was later shown by Turing that both the notions were equivalent and the expressive power of Church's λ-calculus is same as that of Turing Machines. To understand the equivalence it is important to understand basic terminology of Turing machines. In the next section, we will go over the basics of Turing machine and then build towards an equivalence between λ calculus and Turing machines.

7.2 Computing Machines (Turing Machines)

Informally Turing machines work with a tape and a 'read-write' head. The tape extends infinitely in both directions, each cell of which holds one symbol. In his original paper[3], Turing defines two type of cells viz. F-square (figures) and E-squares (erasable). These squares extend indefinitely on both the sides of the tape and are places alternately. F-squares are used to mark figures (actual

useful computation) and E-squares are used as a scratchpad. At each step, there is a current position where the machine can read the symbol at the current position and write a symbol in that square. At each step, the machine is in one of a finite number of states. Symbols on the tape include both the input symbols and a finite collection of auxillary symbols, including a special blank symbol. At every step, all but finitely many of the tape cells hold the blank symbol (i.e most of the tape is empty).

The machine updates as follows: Depending on the current state and the currently scanned input symbol, the machine erases the symbol and writes a new symbol into that cell. (The new symbol could be the same as the one that was erased.) It moves the current position one cell to the left or right, and changes to a new state.

m-configuration (current state) and the scanned symbol define the configuration of the machine. If a machine prints two kinds of symbols, of which the *first kind* (called figures) consists entirely of 0 and 1 (the others being called symbols of the *second kind*), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m-configuration* , the subsequence of the symbols printed by it which are of the first kind will be called the sequence computed by the machine [3].
Following definitions are taken from Turing's paper: [3].

7.2.1 Circular and Circle-free machines

If a computing machine always writes down only a finite number of symbols of the first kind, it will be called circular. Otherwise it is said to be circle-free. A machine will be circular if it reaches a configuration from which there is no possible move i.e it gets stuck.

7.2.2 Computable sequences and numbers

A sequence is said to be computable if it can be computed by a circle-free machine. A number is computable if it differs by an integer from the number computed by a circle-free machine.

8 Proof of equivalence

Now for the proof of the equivalence between λ-definability[8] and Turing computability, Turing in his paper [4] extends the notion of λ-definability to λ-K-definability. This modified form of λ-definability, known as λ-K-definability is shown to be equivalent to Turing machines and and a proof λ-definable function is λ-K-definable is trivial

8.1 λ-K-definability

There will be three differences from the normal λ-definable functions. 1. For λ-K-definable we use only one kind bracket [ ], istead of three, {}, ( ), [ ]; 2. We use x, $x^{|}, x^{||} \ldots$ as variables instead of an indefinite infinite list of the single symbols 3. Change in the form of condition (ii) of immediate transformability, not affecting the definition of convertibility except in form.

8.2 Immediate transformability

i M is of the form $\lambda.V[X]$ and N is $\lambda.U[Y]$ where Y is obtained from X by replacing the variable V by the variable U throughout, provided that U does not occur in X.[4]

ii M is of the form $[\lambda.V[X]][Y]$ where V is a variable and N is obtained by substituting Y for V throughout X. This is to be subject to the restriction that if W be either V or a variable occurring in Y, then λW must not occur in X.[4]

iii N is immediately transformable into M by (ii).[4]

8.3 Outline

The well-formed-formulae in λ-calculus representing an integer n will be denoted by N_n. Since in the lambda calculus the numerals are defined for natural numbers whereas Turing machine prints only 0 and 1, we will associate 0 with N_1 and 1 with N_2. More formally, we say that a sequence γ whose n^{th} figure is $\phi_\gamma(n)$ is λ-definable or effectively calculable if $1+\phi_\gamma(n)$ is a λ-definable function of n.[7] That is to say that there is a well-formed-formula M_γ such that following holds true for all integers n:

$$\{M_\gamma\}(N_n) \twoheadrightarrow_\beta N_{1+\phi_\gamma(n)}$$

i.e $\{M_\gamma\}(N_n)$ is convertible into $\lambda xx^{|}.x(x(x^{|}))$ or into $\lambda xx^{|}.x(x^{|})$ according as the n^{th} figure of γ is 1 or 0.

Now the proof to show that every λ-definable sequence γ is computable goes by construction of a Turing machine which computes γ. The construction of such a machine say $\mathcal{L}$ relies on the sub-machine $\mathcal{L}_1$, which if supplied with a well-formed-formula, M obtains any formula into which M is convertible. For exact construction of $\mathcal{L}_1$ many subroutines of functions like comparing, enumerating, pasting, copying etc. are required. Details of these are too tedious to be noted down here. Turing in his paper [4] constructs such a machine and an interested reader is referred to see the details there.

Assuming we have such a machine $\mathcal{L}_1$ as described earlier, then it is easy to construct a machine $\mathcal{L}_2$ which successively obtains all the formulae into which M is convertible. Now the machine $\mathcal{L}$ is such that it has n-sections each for computing the n^{th} figure of the sequence γ. The first stage in each of the section is the formation of $\{M_\gamma\}(N_n)$. This formula is sent to $\mathcal{L}_2$ which subsequently obtains each formula to which it can be converted. Each formula to which $\{M_\gamma\}(N_n)$ can be converted eventually appears on the tape, and is compared to $\lambda xx^{|}.x(x(x^{|}))$ or $\lambda xx^{|}.x(x^{|})$. Based on the verdict a suitable figure 1 or 0 respectively is printed on the tape. By hypothesis $\{M_\gamma\}(N_n)$ is convertible into one of the formulae N_2 or N_1 therefore n^{th} section of the machine $\mathcal{L}$ finishes and writes down the n^{th} figure of the sequence γ.

Declaration

I/We hereby declare that the work presented in the project report entitled ..... contains my own ideas in my own words. At places, where ideas and words are borrowed from other sources, proper references, as applicable, have been cited. To the best of our knowledge this work does not emanate or resemble to other work created by person(s) other than mentioned herein. The work was created on this ..... day of ..... 20..

Name Signature

Date:

References

[1] Knuth, Donald E. "Backus normal form vs. backus naur form." Communications of the ACM 7.12 (1964): 735-736.

[2] Selinger, Peter. "Lecture notes on the lambda calculus." arXiv preprint arXiv:0804.3434 (2008).

[3] Turing, Alan M. "On computable numbers, with an application to the Entscheidungsproblem." Proceedings of the London Mathematical Society 2.1 (1937): 230-265.

[4] Turing, Alan M. "Computability and λ-definability." The Journal of Symbolic Logic 2.4 (1937): 153-163.

[5] Wang, Yuan, ed. The Goldbach Conjecture. Vol. 4. World Scientific, 2002.

[6] Ker, A.D.: Lambda Calculus. Course Notes. Oxford University, Oxford (2003)

[7] Church, Alonzo. "An unsolvable problem of elementary number theory." American Journal of Mathematics 58.2 (1936): 345-363.

[8] Kleene, Stephen Cole. "A theory of positive integers in formal logic. Part I." American Journal of Mathematics 57.1 (1935): 153-173.