Chapter 2

Lambda Calculus: History and Syntax

1 A Brief History

In 1928, Hilbert and Ackerman posed a challenge: devise an algorithm that takes as input a first-order logic statement and determines whether that statement is valid or not. Soon after, Alonzo Church, then Professor at the Department of Mathematics in Princeton, started working on this problem. His approach was to research the notion of "function" and create based on this notion a logical system that is sufficient for all of mathematics. Lambda calculus emerged out of this research, also with contributions from Church's students Kleene and Rosser. This research led to Church's 1936 paper showing that an algorithm as desired Hilbert and Ackerman's does not exist . His solution was to formulate a term in Lambda Calculus and show that there is no way to determine whether that term has a closed form (more precisely β-normal form). About one year later, Turing published his paper , where he establish the same result but using different techniques that are based on "computing machines", and proved that his and Church's approach were equivalent.

Church and Turing. Church and Turing's results are like two sides of a coin. Church's result is all about abstraction offers a mathematical language in which computation can be expressed. Turing's result is all about implementation: it convincingly describes how to implement computation.

2 Abstract Syntax of Lambda Calculus

There are at least several ways to define the syntax of Lambda Calculus. In this section, we go through these different ways, partly because they introduce some basic techniques that we shall use in this course.

Definition 2.1 (Inductive Definition)**.** Let V be a countable set V of variables. We define the abstract syntax for lambda calculus inductively as follows. $\mathcal{T}$ is the *least* set of the terms that satisfy the following.

1. if $x \in V$ then $x \in \mathcal{T}$
2. if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$ then $t_1 t_2 \in \mathcal{T}$
3. if $x \in V$ and $t \in \mathcal{T}$ then $\lambda x.t \in \mathcal{T}$
4. $\mathcal{T}$ is the "least" set verifying the above properties

Each term in $\mathcal{T}$ is called a ***lambda term***.

Example 2.1. Example lambda terms include

- x,
- $\lambda x.x$, and
- $\lambda x.x\ y$.

Definition 2.2 (Lambda Abstraction and Application)**.** The term $\lambda x.x$ is called ***(lambda) abstraction***, and the term $t_1\ t_2$ is known as ***application***.

An intuitive way of thinking of $\lambda x.t$ is as a function that takes x and computes the result in its body t.

Abstract versus Concrete Syntax. This inductive definition of lambda calculus is an *abstract syntax*: it defines the set of properly parsed terms (i.e., abstract syntax trees).

It does not tell us how to read a lambda term as written in *concrete syntax*. For example, given $\lambda x.x\ y$, we can parse it as $(\lambda x.x\ y)$ or $(\lambda x.x)y$. Similarly, we can parse $t_1\ t_2\ t_3$ as $(t_1\ t_2)\ t_3$ or $t_1\ (t_2\ t_3)$.

Disambiguation Conventions. We will use parenthesis to aid in parsing (to disambiguate the syntax). To minimize parenthesis, we will have the following conventions:

1. Application associates to the left.
2. The body of a lambda terms extends as far as right as possible.

With this convention $\lambda x.x\ y$ is parsed as $\lambda x.(x\ y)$ and $t_1\ t_2\ t_3$ is parsed as $(t_1\ t_2)\ t_3$.

Definition 2.3 (Inference Rules)**.** We can define the abstract syntax of Lambda Calculus by using inference rules. Given a countable set of variables V, the set of lambda terms is defined as follows.

1. $\dfrac{x \in V}{x \in \mathcal{T}}$

2. $\dfrac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T}}{t_1\ t_2 \in \mathcal{T}}$

3. $\dfrac{x \in V \qquad t \in \mathcal{T}}{\lambda x.t \in \mathcal{T}}$

Exercise 2.1. Convince yourself that the two definitions above, the a nd a re equivalent.

Definition 2.4 (BNF Style)**.** We can define the syntax of Lambda Calculus by using the BNF style. Assuming that x ranges over a countable set of variables, the set of lambda terms t is defined as follows.

$$t ::= x \mid t_1 t_2 \mid \lambda x.t$$

Exercise 2.2. Suppose that our syntax allow us to write natural numbers and add them. For example, we may have terms like this $1, 2, +\ 1\ 2$.

Now, what does the following lambda terms do?

1. $\lambda x.+\ x\ 1$,
2. $\lambda x.\lambda y.+\ x\ y$,
3. $\lambda x.\lambda y.\lambda z.z(+\ x\ y)$.

Summary. What is remarkable about Lambda Calculus is that just a one line definition suffices to define all of computation. We don't need to talk about "tapes", "cells" "instructions", "states", etc, which are needed even in the most basic definition of Turing Machine. Not only it is elegant, but it is also powerful: it allows expressing sophisticated algorithms clearly and concisely.

Note. Even many modern programming languages today lack the features of Lambda Calculus. Though many are also busy trying to add them (which is not always easy). For example, the C++17 has now support for Lambda expressions as outlined in this document from the C++ Standard Committee.

For good measure, perhaps they could have waited a few more years to match the 100th year invention of Lambda Calculus!

3 Bound and Free Variables

Definition 2.5 (Parameters and Binding)**.** Consider a lambda abstraction of the form $\lambda x.t$, which denotes a function. We refer to the variable x as the ***formal parameter*** or more simply as the ***parameter*** of the abstraction (function), and say that the abstractions ***binds*** x.

Definition 2.6 (Bound and Free Variables)**.** The occurrence of a variable is ***bound*** if there is an enclosing lambda abstraction that binds the variable, and is ***free*** otherwise.

Example 2.2. Consider the lambda abstraction, $\lambda x.x + y$. In order to evaluate the function, for a particular argument x, we need to know the value of y. The variable y in this case is free and x is bound; λ binds x.

Example 2.3. Some example lambda abstractions and their bound and free variables.

1. $\lambda x.x$ is the identity function. Here x is a bound variable.
2. $\lambda x.y$ is the constant "y" function. Here y is a free variable.
3. $\lambda x.x\ y$ has x as a bound variable and y as a free variable.
4. $\lambda x.(\lambda y.x\ y)$ has both variables bound.
5. $x\ \lambda x.x + 1$. In this term the first occurrence of x is free and the second is bound.

Definition 2.7 (Free Variables)**.** The set of ***free variables*** of a term t, written as $FV(t)$, is defined as

1. if $t = x$ then $FV(x) = \{x\}$, or
2. if $t = t_1\ t_2$ then $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$, and
3. if $t = \lambda x.t$ then $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Definition 2.8 (Closed and Open Terms)**.** A term t is closed if $FV(t) = \varnothing$. Otherwise t is open.

4 Alpha Conversion and Alpha Equivalence

In any abstraction, the formal parameter acts simply as a placeholder and therefore we can change as long as we don't create a "clash" with a free variable. For example, consider the abstraction $\lambda x.t$, where x is the argument. We can rename x to any variable, say y, as long as y is not a free variable in t.

Definition 2.9 (Alpha Conversion and Alpha Equivalence)**.** An α***-conversion*** or α***-variation*** of a lambda term t is another term t' where the a bound variable x of t is renamed to another variable $y \notin FV(t)$. We say than two terms that are reducible to each other by α-conversions are ***alpha equivalent*** and denote α-equivalent terms t_1 and t_2 as $t_1 =_\alpha t_2$.

Example 2.4.
1. $\lambda x.x =_\alpha \lambda y.y$.
2. $\lambda x.y \neq_\alpha \lambda y.y$.
3. $\lambda x.\lambda y.\lambda z.x\ y\ z =_\alpha \lambda z.\lambda y.\lambda x.z\ y\ x$.
4. $\lambda x.t =_\alpha \lambda y.[y/x]t$ if $y \notin FV(t)$.

Exercise 2.3. Show that $=_\alpha$ is an equivalence relation.