Introduction to Lambda Calculus

Dr. K.R. Chowdhary, Professor,
Depatment of Computer Science and Engineering,
Faculty of Engineering, JNV university, Jodhpur,
Email: kr.chowdhary@acm.org.

March 19, 2013

Abstract

This note provides a brief introduction to Lambda calculus, relates it to a turing machine, and recursive functions, provides examples as how functions can be used define computation, instead of storing intermediate results. The note also suggests its application in programming language design, and compilers.

1 Introduction

What is a function? In modern mathematics, the prevalent notion is that of "functions as graphs": each function f has a fixed domain X and codomain Y, and a function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$. Two functions $f, g : X \to Y$ are considered equal if they yield the same output on each input, i.e., $f(x) = g(x)$ for all $x \in X$. This is called the *extensional* view of functions, because it specifies that the only thing observable about a function is how it maps inputs to outputs.

However, before the 20th century, functions were rarely looked at in this way. An older notion of functions as that of "functions as rules". In this view, to give a function means to give a rule for how the function is to be calculated. Often, such a rule can be given by a formula, for instance, the familiar $f(x) = x^2$ or $g(x) = sin(e^x)$ from calculus. As before, two functions are *extensionally* equal if they have the same input-output behavior; but now we can also speak of another notion of equality: two functions are *intensionally* equal if they are given by (essentially) the same formula.

When we think of functions as given by formulas, it is not always necessary to know the domain and codomain(range) of a function. Consider for instance the function $f(x) = x$. This is, of course, the *identity function*. We may regard it as a function $f : X \to X$ for any set X.

In most of cases, the "functions as graphs"paradigm is the most elegant and appropriate way of dealing with functions. Graphs define a more general class of functions, because it includes functions that are not necessarily given by a rule.

In computer science, the "functions as rules"paradigm is often more appropriate. Think of a computer program as defining a function that maps input to output. Most computer programmers (and users) do not only care about the extensional behavior of a program (which inputs are mapped to which outputs), but also about how the output is calculated: How much time does it take? How much memory and disk space is used in the process? How much communication bandwidth is used? These are intensional questions having to do with the particular way in which a function was defined.

2 Reduction and functional programming

A functional program consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. Reduction consists of replacing a part P of E by another expression P according to the given rewrite rules. In schematic notation

$$E[P] \to E[P'],$$

provided that $P \to P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called normal form $E*$ of the expression E, consists of the output of the given functional program. An example is as given below.

$$\begin{aligned} (7+4)*(8+5*3) &\to 11*(8+5*3) \\ &\to 11*(8+15) \\ &\to 11*23 \\ &\to 253 \end{aligned}$$

In this example the reduction rules consist of the 'tables' of addition and of multiplication on the numerals. Also symbolic computations can be done by reduction. For example

first of (sort (append ('dog', 'rabbit') (sort (('mouse', 'cat'))))) →
→ first of (sort (append ('dog', 'rabbit') ('cat', 'mouse')))
→ first of (sort ('dog', 'rabbit', 'cat', 'mouse'))
→ first of ('cat', 'dog', 'mouse', 'rabbit')
→ 'cat'.

The necessary rewrite rules for *append* and *sort* can be programmed easily in a few lines. Functions like append given by some rewrite rules are called *combinators*.

Reduction systems usually satisfy the Church-Rosser property, which states that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced as follows:

$$\begin{aligned}(7+4)*(8+5*3) &\rightarrow (7+4)*(8+15)\\ &\rightarrow 11*(8+15)\\ &\rightarrow 11*23\\ &\rightarrow 253\end{aligned}$$

or even by evaluating several expressions at the same time:

$$\begin{aligned}(7+4)*(8+5*3) &\rightarrow 11*(8+15)\\ &\rightarrow 11*23\\ &\rightarrow 253.\end{aligned}$$

3 Lambda Calculus

The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions. Let us begin by looking at another well-known language of expressions, namely arithmetic. Arithmetic expressions are made up from variables $(x, y, z, \dots)$, numbers $(1, 2, 3, \dots)$, and operators $\{+, -, \times\}$ etc. An expression such as $x + y$ stands for the result of an addition (as opposed to an instruction to add, or the statement that something is being added).

The great advantage of this language is that expressions can be nested without any need to mention the intermediate results explicitly. So for instance, we write

$$A = (x + y) \times z^2,$$

and not the following:
let $w = x + y$, then let $u = z^2$, then let $A = w \times u$.

The latter notation would be tiring and cumbersome to manipulate. The *lambda calculus* extends the idea of an expression language to include functions. Where we normally write

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the statement that x is being mapped to x^2). As in arithmetic, we use parentheses to group terms. It is understood that the variable x is a local variable in the term $\lambda x.x^2$. Thus, it does not make any difference if we write $\lambda y.y^2$ instead. A local variable is also called a *bound* variable.

3.1 Application and abstraction

The first basic operation of the λ-calculus is application. The expression $F.A$ or FA denotes the data F considered as algorithm applied to the data A considered as input. This can be viewed in two ways: either as the process of computation FA or as the output of this process. The first view is captured by the notion of conversion and even better of reduction; the second by the notion of models (semantics).

The theory is *type-free*: it is allowed to consider expressions like FF , that is F applied to itself. This will be useful to simulate recursion. The other basic operation is abstraction. If $M \equiv M[x]$ is an expression containing ('depending on') x, then $\lambda x.M[x]$ denotes the function $x \to M[x]$.

Application and abstraction work together in the following intuitive formula.

$$\begin{aligned}(\lambda x.2 * x + 1)3 &\to 2 * 3 + 1 \\ &\to 7.\end{aligned}$$

That is, $(\lambda x.2*x+1)3$ denotes the function $x \to 2*x+1$ applied to the argument 3 giving $2 * 3 + 1$ which is 7. In general we have

$$(\lambda x.M[x])N = M[N].$$

This last equation is preferably written as

$$(\lambda x.M)N = M[x := N]$$

where $[x := N]$ denotes substitution of N for x. It is remarkable that although above is the only essential axiom of the λ-calculus, the resulting theory is rather involved.

3.2 Composition of Functions

One advantage of the lambda notation is that it allows us to easily talk about higher-order functions, i.e., functions whose inputs and/or outputs are themselves functions. An example is the operation $f \to f \circ f$ in mathematics, which takes a function f and maps it to $f \circ f$, the composition of f with itself. In the lambda calculus, $f \circ f$ is written as

$$\lambda x.f(f(x)),$$

and the operation that maps f to $f \circ f$ is written as

$$\lambda f.\lambda x.f(f(x)).$$

The evaluation of higher-order functions can get somewhat complex; as an example, consider the following expression:

$$((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5)$$

Convince yourself that this evaluates to 625. Another example is given in the following exercise.

3.3 Functions of more arguments

Functions of several arguments can be obtained by iteration of application. Intuitively, if $f(x, y)$ depends on two arguments, one can define

$$F_x = \lambda y.f(x, y),$$

$$F = \lambda x.F_x.$$

Then

$$(F_x)y = F_x y = f(x, y).$$

This last equation shows that it is convenient to use association to the left for iterated application: $FM_1 \dots M_n$ denotes

$$(\dots((FM_1)M_2)\dots M_n).$$

The equation above becomes

$$Fxy = f(x, y).$$

Dually, iterated abstraction uses association to the right: $\lambda x_1 \dots x_n.f(x_1, \dots, x_n)$ denotes

$$\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.f(x_1, \dots, x_n)))).$$

4 Lambda Calculus and Computability

In the 1930's, several people were interested in the question: what does it mean for a function $f : \mathbb{N} \to \mathbb{N}$ to be computable? An informal definition of computability is that there should be a *pencil-and-paper* method allowing a trained person to calculate $f(n)$, for any given n. The concept of a pencil-and-paper method is not so easy to formalize. Three different researchers attempted to do so, resulting in the following definitions of computability:

1. Turing defined an idealized computer we now call a *Turing machine*, and postulated that a function is computable (in the intuitive sense) if and only if it can be computed by such a machine.

2. Godel defined the class of *general recursive functions* as the smallest set of functions containing all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion). He postulated that a function is computable (in the intuitive sense) if and only if it is general recursive.

3. Church defined an idealized programming language called the *lambda calculus*, and postulated that a function is computable (in the intuitive sense) *if and only if* it can be written as a lambda term.

It was proved by Church, Kleene, and Turing that all three computational models were equivalent to each other, i.e., each model defines the same class of computable functions. Whether or not they are equivalent to the "intuitive"notion of computability is a question that cannot be answered, because there is no formal definition of "intuitive computability". The assertion that they are in fact equivalent to intuitive computability is known as the Church-Turing thesis.

5 Connections to computer science

The lambda calculus is a very idealized programming language; arguably, it is the simplest possible programming language that is Turing complete. Because of its simplicity, it is a useful tool for defining and proving properties of programs. Many real-world programming languages can be regarded as extensions of the lambda calculus. This is true for all functional programming languages, a class that includes Lisp, Scheme, Haskell, and ML(Meta Language). These languages combine the lambda calculus with additional features, such as data types, input/output, side effects, udpateable memory, object orientated features, etc. The lambda calculus provides a vehicle for studying such extensions, in isolation and jointly, to see how they will affect each other, and to prove properties of programming language (such as: a well-formed program will not crash).

The lambda calculus is also a tool used in compiler construction.