ISSN:0975-9646

Lambda Calculus

Sampa Das
Assistant Professor., Computer Science and Engineering, WBUT ,
Siliguri Institute Of Technology, Darjeeling, India

Abstract: : In this work, we present preliminary study of Lambda Calculus in the field of computability .Originally developed in order to study some mathematical properties of effectively computable functions, this formalism has provided a strong theoretical foundation for the family of functional programming languages. We show how to perform some arithmetical computations using the lambda calculus and how to dene recursive functions, even though functions in lambda calculus are not given names and thus cannot refer explicitly to themselves.

Keywords: ***Variables, Syntax, Semantic.***

1. INTRODUCTION

The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions[1].Lambda calculus(also written as λ)is a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application using variable binding and substitution .The name is derived from Greek letter lambda(λ) . Lambda Calculus has played an important role in the development of theory programming language. Counterparts to lambda Calculus. We have several options regarding types in the lambda calculus.

Untyped lambda calculus. In the untyped lambda calculus, we never specify the type of any expression. Thus we never specify the domain or co-domain of any function. This gives us maximal flexibility. It is also very unsafe, because we might run into situations where we try to apply a function to an argument that it does not understand.

Simply-typed lambda calculus. In the simply-typed lambda calculus, we always completely specify the type of every expression. This is very similar to the situation in set theory. We never allow the application of a function to an argument unless the type of the argument is the same as the domain of the function.

Thus, terms such as f (f) are ruled out, even if f is the identity function.

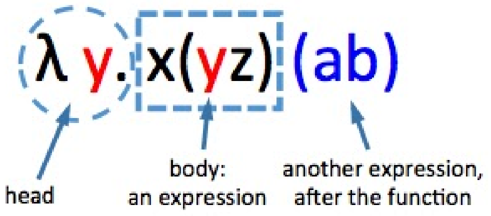

Fig: Lambda Function
Source: http://www.cs.unr.edu/~sushil/class/326/notes /wk7.1.html

Polymorphic typed lambda calculus. This is an intermediate situation, where we may specify, for instance, that a term has a type of the form X → X for all X, without actually specifying X[2].

2.EXPRESSING LAMBDA CALCULUS

The lambda calculus is a formal language. The expressions of the language are called lambda terms, and we will give rules for manipulating them.

A. THE SYNTAX

The syntax of the lambda calculus defines some expressions as valid lambda calculus expression and some as invalid, just as some strings of characters are valid C programs and some are not. A valid lambda calculus expression is called a "lambda term" .The following three rules give an inductive definition that can be applied to build all syntactically valid lambda terms:

• A variable, x , is itself a valid lambda term
• If t is a lambda term, and is a variable ,then (λx.t) is a lambda term (called a lambda abstraction);
• If t and s are lambda terms, then (ts) is a lambda term (called an application).

Nothing else is a lambda term. Thus a lambda term is valid if and only if it can be obtained by repeated application of these three rules. However, some parentheses can be omitted according to certain rules. For example, the outer most parentheses are usually not written.

a. The Lambda Variables

When a variable occurs in the body of an abstraction that uses the variable in its parameter, the occurrence of this variable inside the body is called bound, because the occurrence is bound to this binder abstraction. If a variable occurs at a position such that there is no such outer binding abstraction, the occurrence of the variable is called free. For example, in term (λx.x) y, y is free. In λy.xy, x is free. In term λx.x x, all occurrences of x inside the body are bound. In (λx.x)x, first occurrence of x (inside the body) is bound, and the second occurrence is free. In λz.λx.λy.x y z, the occurrences of x, y and z in the body have their respective binder abstractions. After reducing(λz.λx.λy.x y z)a, where a is just a variable term, we obtain (λx.λy.xya), in which, a is a free variable.

Free and bound variables

In calculus all names are local to definitions. In the function λx:x we say that x is bound" since its occurrence in the body of the dentition is preceded by x. A name not preceded by a is called a free variable". In the expression (λx:xy) the variable x is bound and y is free.

In the expression

(λx:x) (λy:yx)

the x in the body of the first expression from the left is bound to the first λ. The y in the body of the second expression is bound to the second λ and the x is free. It is very important to notice that the x in the second expression is totally independent of the x in the first expression.

Formally we say that a variable <name> is free in an expression if one of the following three cases holds:

* <name> is free in <name>.

* <name> is free in λ <name1 > : <exp> if the identifier <name>!<name1 > and <name> is free in <exp>.

* <name> is free in E1E2 if <name> is free in E1 or if it is free in E2. A variable <name> is bound if one of two cases holds:

* <name> is bound in <name1 > : <exp> if the identifier <name>=<name1 >or if <name> is bound in <exp>.

* <name> is bound in E1E2 if <name> is bound in E1 or if it is bound in E2.

It should be emphasized that the same identifier can occur free and bound in the same expression. In the expression

(λx:xy) (λy:y)

the first y is free in the parenthesized sub expression to the left. It is bound in the sub expression to the right. It occurs therefore free as well as bound in the whole expression [4].

b. LAMBDA (λ) REDUCTION

The key notion of the λ-calculus is that it is possible to arrive at logically equivalent expression by means of a process called λ-reduction. In the usual case, λ-reduction is actually a combination of three distinct reduction operations, each of which is discussed below. The key operation, the one that does the heavy lifting, is called β-reduction, and that is operation we will discuss first. By the way, some people say "λ-conversion" instead of λ-reduction; others reserve "λ-conversion" to refer specifically to a single step in a series of reductions.

1. β - REDUCITON

Nothing happens until a λ-binding form occurs in construction with an argument, thus: ((λ var body) argument) Once a λ-based binding form occurs with an argument like this, it is possible to reduce the expression to a simpler form by means of β-reduction (sometimes with the help of α-reduction and η-reduction). The main idea of β-reduction is to replace every free occurrence of the variable "var" in "body" with "argument". For instance, in the form below, both occurrences of "var" in the body are free. Consequently, after β-reduction, both occurrences get replaced with "argument", and the result is significantly simpler than the original expression.

The beta reduction is perhaps the most intuitive reduction, but is also the most important in analyzing lambda expressions. It is, in essence, a direct substitution, and it goes something like this:

((λ (x) . BODY) a) ->β BODY [x λa]= BODY

Lets look at this step by step.

((\ (x) . BODY) a)

As you should already know, this first part defines a function which takes an argument 'x' and puts it into 'BODY'. In this instance, we are looking to pass 'a' to that function. This is the expression on which we want to perform our beta reduction.

->β BODY [x λ a] = BODY

Here is our beta reduction. First we explicit say we are doing a beta reduction, then the BODY [x\a] part means "In 'BODY', I am substituting all instances of 'x' with 'a'". Because there is no 'x' in BODY, no substitution is actually made. After this (or on the next line) you write the result.

2. α - REDUCITON

Unfortunately, applying β-reduction indiscriminately can cause trouble when the body contains binding operators. Following the rules of β-reduction, we replace "x" with "z" for a result of "(λ y1 (z y1))", and this result is correct. Note that in the result the λ-operator binds exactly one variable in the body, and the other variable remains free. But consider what happens in this closely parallel but slightly different situation:

((λ x (λ y (x y))) y) β-reduction: substitute "y" in for "x" in the body "(λ y (x y))" ==>(λ y (y y)) [Wrong result!]))", in which the λ-operator binds two variables, not just one, which is not correct. The argument variable "y" is said to have been `captured' by the inner λ-operator; another commonly-used expression for this kind of situation is `variable collision'. The solution is to make use of alphabetic variants. Roughly, two expressions are "alphabetic variants" if they are identical except perhaps for the choice of variable symbols. For instance, the following expressions are alphabetic variants of one another:

(λ x (λ y (x (+ y x))))

(λ z (λ y (z (+ y z))))

To create an alphabetic variant for an expression of the form "(λ var body)", simply replace each free occurrence of "var" in the expression with "new", where "new" is a variable symbol not occurring anywhere in "body". (Since expressions have finite length, as long as there is an infinite supply of variable symbols, it will always be possible to find a suitable variable to serve the role of "new".) This transformation is called α-reduction. The crucial property of the reduced form is that each λ operator binds the same number of variables in the same positions within its body.[3].

An alpha reduction is hardly a reduction at all. It could be otherwise called, "renaming," but must be done with some care to make sure that one does not alter the scope of a variable when renaming it. Lets look at an example:

(λ (x) (+ x y))

In this expression, x is used as part of a function. It can be said that x is bound to the scope of the lambda expression, but y is free. You should agree that this is equivalent to:

(λ z.(+ z y))

If you wanted to do this formally, it would be called an alpha reduction, and would be written as:

(λ x.(+ x y)) ->α (λ (z) (+ z y))

3. $_{\eta}$- Reduction

Eta reductions are used to eliminate useless variables in abstractions. Let's look at an example:

$$(\lambda\, x.(\text{BODY } x)) \to_{\eta} = \text{BODY}$$

In this abstraction, whenever the function is to be used, any argument will simply be passed to BODY. Hence, this function is, in essence, equal to BODY. One caveat of eta abstraction is that a name conflict must be avoided. What this means is if x is a free variable (i.e., not scoped in a function abstraction) in BODY itself, then this eta reduction is not valid, as it would be changing the scope of x. In order to avoid this, an alpha reduction must be performed before the eta reduction. .Eta reductions are very rarely used, and in most applications of the lambda calculus, it can be avoided completely.[5] Eta reductions are used to eliminate useless variables in abstractions. Let's look at an example:

$$(\lambda\, x.(\text{BODY } x)) \to\eta = \text{BODY}$$

In this abstraction, whenever the function is to be used, any argument will simply be passed to BODY. Hence, this function is, in essence, equal to BODY. One caveat of eta abstraction is that a name conflict must be avoided. What this means is if x is a free variable (i.e., not scoped in a function abstraction) in BODY itself, then this eta reduction is not valid, as it would be changing the scope of x. In order to avoid this, an alpha reduction must be performed before the eta reduction.

Eta reductions are very rarely used, and in most applications of the lambda calculus, it can be avoided completely.

3.APLLICATION OF LAMBDA CALCULUS

ARITHMATIC

We expect from a programming language that it should be capable of doing arithmetical calculations. Numbers can be represented in lambda calculus starting from zero and writing “suc(zero)" to represent 1, “suc(suc(zero))" to represent 2, and so on. In the lambda calculus we can only define new functions. Numbers will be defined as functions using the following approach: zero can be defined as

$$\lambda\, s{:}(\,\lambda\, z{:}z)$$

This is a function of two arguments s and z. We will abbreviate such expressions with more than one argument as

$$\lambda\, sz{:}z$$

It is understood here that s is the first argument to be substituted during the evaluation and z the second. Using this notation, the first natural numbers can be defined as

1 = λ sz:s(z)
2 = λ sz:s(s(z))
3 λ sz:s(s(s(z)))
and so on.

a.Addition

Adding numbers can be understood as automating the successor function. If we want to add 5 to the number 3, this can be interpreted as using the successor function five times on 3. (Or the other way around, because 3+5 = 5+3.) Fortunately, our way of writing numbers has this automation already built in. As we have pointed out above, evaluating a number n means that we replicate the expression after it n times. If the expression after it is the successor function, it will be spelled out n times, and if we resolve it, then the successor function will be applied n times to the number expression after it.[6]

3+5 = 3S5 = λ sz.s(s(s(z))) (λ abc.b(abc)) λ xy.x(x(x(x(x(y)))))

If you feel playful, you may try and see that it resolves properly to

8: λ xy.x(x(x(x(x(x(x(x(y)))))))) [7]

b. Multiplication

A similarly clever function yields multiplication:

MULTIPLY :⇔ λ abc.a(bc)

This function takes two arguments, for instance like this:

2 x 3 = MULTIPLY 2 3 :⇔ (λ abc.a(bc)) (λ sz.s(s(z))) (λ xy.x(x(x(y))))
= λ c.(λ sz.s(s(z)))((λ xy.x(x(x(y))))c)
= λ cz.((λ xy.x(x(x(y))))c)(((λ xy.x(x(x(y))))c)(z))
= λ cz.(λ y.c(c(c(y)))) (c(c(c(z))))
= λ cz.c(c(c(c(c(c(z)))))) = 6

we see that our multiplication function takes its two arguments (2 and 3) and arranges them like this:

MULTIPLY 2 3 = (λ abc.a(bc)) 2 3 = λ c.2(3c)

Resolving this gives us λ cz.(3c)(3c(z)). This is equivalent to applying the second c three times to the z: c(c(c(z))), and applying the first c three times to that result: c(c(c(c(c(c(z)))))). Together with the function head λ cz, it conveniently results in 6 (i.e., six times the application of the first argument to the second).The best way to get rid of any remaining bewilderment will be if you take a piece of paper and a pen and try a few multiplications yourself. [6]

4.CONCLUSION

The Lambda Calculus cannot do all of mathematics, because many mathematical problems have no solution, and many mathematical formulas cannot be computed (which is not the same thing). The Lambda Calculus can also be used to compute neural networks with arbitrary accuracy, by expressing the strengths of the connections between individual neurons, and the activation values of the neurons as numbers, and by calculating the spreading of activation through the network in very small time.

REFERENCE

1. H. P. Barendregt. The Lambda Calculus, its Syntax and Semantics.NorthHolland, 2nd edition, 1984
2. J.-Y. Girard, Y. Lafont, and P. Taylor. Proofs and Types. Cambridge University Press, 1989.
3. The New York University “Lambda Tutorial” [Online4. Gerold]

4. G. Michaelson, An Introduction to Functional Programming through Lambda Calculus,Addition-wesley,Wokingham,1998
5. http://www.cs.unr.edu/~sushil/class/326/notes/wk7.1.html
6. G. Revesz, Lambda-Calculus Combinators and Functional Programming, Cambridge University Press, Cambridge, 1988, chapters 1-3.
7. P. M, Kogge, The Architecture of Symbolic Computers, McGraw-Hill, New York,1991, chapter 4.

AUTHOR PROFILE

Sampa Das received the B.E. degrees in Computer Science Engineering from National Institute of Technology in 2007 and pursuing M Tech from Sikkim Manipal Institute of Technology. Working as a Asst. Prof. in Siliguri Institute Of Technology since 2008 to till now.