Functional Programming Theory & Applications

Draft Version

Avinash Bharti

June 15, 2019

Contents

List of Figures

Part I.

1. Lambda Calculus

1.1. Considering the Origins of Lambda Calculus

- Alonzo Church originally created lambda calculus in the 1930s, which is before the that computers were available. Lambda Calculus explores the theoretical basis for what it means to compute. Alozo Church worked with people like Haskell Curry, Kurt Gödel, Emil Post and Alan Turing to create a definition for algorithms.
- Church's motivation in creating lambda calculus was to prove that Hilbert's Entscheidungsproblem or decision problem `https://en.wikipedia.org/wiki/Entscheidungsproblem`.
- The goals of lambda calculus are to study the interaction of functional abstraction and function application from an abstract, purely mathematical perspective.
- *Functional abstraction* begins by breaking a particular problem into series of steps.
- *Functional application* is the act of applying the function to an argument and obtaining a value as output.
- Basic properties of lambda calculus :-
  1. Lambda calculus uses only functions - no other data or other types (no Strings, integers, Booleans, or other types found in programming languages today).
  2. Lambda calculus has no state or side effects.
  3. The order of evaluation is irrelevant.
  4. All functions are unary, taking just one argument.

1.2. Understanding the Rules

Lambda calculus performs three operations :-

- Creating functions to pass as variables
- binding a variables to the expression (abstraction)
- applying a function to an argument

1.2.1. Working with variables

When considering variables in lambda calculus, the variables is a place-holder (in mathematical sense) and not a container for values (in the programming sense). Variables provide the basis of the inductive (the inference of general laws from specific instances) definition of lambda terms.

Variables may be untyped or simple typed. The typing doesn't actually indicate a kind of data. Rather, it defines how to interpret the lambda calculus.

1.2.2. Using application

If M and N are lambda terms, the combination MN is also a lambda term. In this case, M generally refers to a function and N generally refers to an input to that function.Lambda calculus is left associated by default . EFG means E is applied to F and F is applied to G. Also math associativity doesn't apply in lambda calculus. Examples :-

- $(x) \rightarrow x+1$:- The statement as that variable x is mapped to $x+1$.

- $(x, y) \to x^2 + y^2$:- The statement is read is saying that tuple (x, y) is mapped to $x^2 + y^2$. But as mentioned earlier lambda calculus allows functions to have just one input , so this should be $x \to (y \to x^2 + y^2)$. The transition of the code so that each function has only one argument is called *curring*.

1.2.3. Using abstraction

The term abstraction derives from the creation of general rules and concepts based on the use and classification of specific examples.

- Abstracting untyped lambda calculus : When E is a lambda term and x is a variable , $\lambda x.E$ is a lambda term. An abstraction is definition of a function, but doesn't invoke the function. Examples :
  1. $f(x) = x + 1$:- the lambda abstraction for this function is $\lambda x.x + 1$
  2. $f(x) = x^2 + y^2$:- $\lambda x.x^2 + y^2$. The lambda calculus has no concept of a variable declaration, therefore the variable y is considered a function isn't yet defined , not a variable deceleration. If y is variable then $\lambda x.(\lambda y.x^2 + y^2)$
- Abstracting simply-typed calculus :- The abstraction process for simply-typed lambda calculus follows the same pattern. The term *types* doesn't refer to string, integer or Boolean - the types used by programming paradigms. The *type* refers to mathematical definition of the function's *domain* and *range* which is represented by $A \to B$. A number of lambda calculus extensions also rely on simple typing including: products, co-products, natural numbers (System T) and some types of recursion (such as Programming Computable Functions, or PCF). Examples :
  1. $\lambda x : \nu.x + 1$:- In this case , the parameter x has a type of ν . This is the Church style of notation. However, in many cases we need to define the type of the function as whole , which requires the Curry-style notation. Therefore the alternative method is $(\lambda x.x + 1) : \nu \to \nu$
  2. When working with multi-parameter inputs, we must curry the function. Like $\lambda x : \nu(\lambda y : \nu.x^2 + y^2)$ in Church style ,
     $(\lambda x : \nu.(\lambda y : \nu.x^2 + y^2)) : \nu \to \nu \to \nu$ in Curry style.

1.3. Performing Reduction (Conversion) Operations

Reduction or conversion is the act of expressing a lambda function in its purest, simplest form and ensuring that no ambiguity exists in its meaning. There are three kinds of reductions to perform various tasks in lambda calculus:

- α alpha
- β beta
- η eta

1.3.1. Considering α-reduction

The act of renaming variables is called *α-conversion* or *α-reduction.* Two functions are *α-equivalent* when they have same result.
$\lambda x.x + 1$
$\lambda a.a + 1$
The following two functions aren't *α-equivalent*; rather they're two functions:
$\lambda x.(\lambda y.x^2 + y^2)$
$\lambda x.(\lambda x.x^2 + x^2)$

1.3.2. Considering β-reduction

The concept of *β-reduction or conversion* is important because it helps simplify lambda functions, sometimes with the help of *α-conversion.* Few definitions :-

- Bound variables :- $\lambda x.x+1$ in this x is bound variable.
- Unbound or free variables :- $\lambda x.x^2+y^2$, here y is free or unbound variable. Unbound variables always remain after any sort of reduction as an unsolved part.

The basic idea behind β*-reduction* is very simple.
$((\lambda x.x+1)z)$ is lambda expression . The result of β*-reduction* appears like this $(z+1)[x:=z]$. Let take another example :-
$(((\lambda x.(\lambda y.x^2+y^2))2)4)$
$(((\lambda x.(\lambda y1.x^2+y1^2))2)4)$
$((\lambda y1.2^2+y1^2)4)[x:=2]$
$(2^2+4^2)[y1:=4]$
2^2+4^2
20

1.3.3. Considering η-reduction

The full implementation of lambda calculus provides a guarantee that the reduction of $(\lambda x.Px)$, in which no argument is applied to x and P doesn't contain x as an unbound (free) variable , results in P. This is the definition of the η*-reduction.*
Note:- η*-reduction we will take up later after some practice of* α*-reduction and* β*-reduction* .

2. Category Theory

to be updated

3. Type Theory

to be updated

4. Logic

to be updated

5. Algorithms

to be updated

Part II.

6. Lambda Calculus with Haskell

to be updated

7. Category Theory with Haskell

to be updated

8. Type Theory with Haskell

to be updated

9. Logic with Haskell

to be updated

10. Algorithms with Haskell

to be updated

Part III.

11. Lambda Calculus with Scala

to be updated

12. Category Theory with Scala

to be updated

13. Type Theory with Scala

to be updated

14. Logic with Scala

to be updated

15. Algorithms with Scala

to be updated

Part IV.

16. Projects in Haskell

to be updated

17. Projects in Scala

to be updated