C++ Templates - Leveraging Turing Complete Behaviour of Lambda Calculus

Vibhu Yadav, Dartmouth College

July 7, 2016

Abstract

The aim of this project is to introduce C++ Template programming and implementation of Lambda Calculus using templates. The project aims to follow the work already done in this field and explore the bright side on implementing such systems. We will also explore Lambda calculus and already existing C++ libraries that make C++ behave as a functional programming language. The project aims to identify new implementations that can take the benefit of such a behaviour and suggest some future work in that direction. Given the time constraints and the difficulty of such an exploration this paper will try to give a brief overview of the process while not skimming on important concepts and help the reader to pick up the required concepts and implement such functions and use cases.

1 Introduction

A piece of code written in any language in modern times tends to be Turing complete in itself. Most languages are turing complete except some data languages like XML, HTML, JSON, S-expressions or some regular languages which are generated by regular expressions and which are recognized by finite automata. Lambda calculus is a mathematical tool to show computation and is also turing complete. Lambda calculus can also be treated as the most basic functional programming as most of the functions used today in the modern languages can be described in lambda calculus. This makes a very strong point to use it in C++ template programming(as it is turing complete, accidentally proven) to make C++ behave like a functional programming language. One may question why is it necessary to do something like this when there are many functional languages out there. Well, few of the benefits of using C++ is that it is a widely accepted programming language used in system programming, low level embedded hardware, large systems with constraints and is also fast, flexible and efficient language. Another benefit lies to the programmer who would want to use functional programming and also use C++ classes and object oriented programming approach together. This could make a special case use of using such a technology to leverage both sides of the programming. There have been some attempts in the recent past to use this and this paper will mostly explore what has been done in this area, give a brief overview, implement some uses cases and point towards a future direction of where such a technology can go hence forward and improve if at all[11][4].

2 Turing Completeness

In computer science theory a turing complete machine needs to a form of conditional jump and be able to perform such operation by being able to save states and produce an output. In theory it can have infinite storage and a very long runtime but in practice some of the process might adhere to turing completeness like most modern computers but it may not be possible to provide sufficiently large storage. The concept is named after English mathematician Alan Turing[9]. As we will discuss in the next section Lambda calculus is a classic example of a Turing complete. Lambda based functional languages(Also discussed in the following sections) need the ability to abstract functions over arguments(for example lambda abstraction) and the ability to apply functions to those arguments(like lambda reduction).
3  Lambda Calculus

The $\lambda$ calculus is almost the smallest, universal and basic computer language possible. It was developed by Alonzo Church in 1930s in order to formulate the concept of effective computability in Mathematics. The $\lambda$ calculus has only one transformation rule of variable substitution and a only one function definition schema. Any computable function can be expressed and calculated to generate an output in a linear fashion in $\lambda$ calculus. Thus this property also makes Lambda Calculus equivalent to a Turing machine.[8][6]

Lambda Calculus can compute almost everything that can be computed because of its Turing complete behaviour as discussed above. A line of symbols is called an expression in lambda calculus. It can be represented in a following manner,

$$(\lambda x.x y)(a b)$$

(1)

where $\lambda$ and .(dot) are used together to write functions with the variable written between these boths. After them comes the expression or the body of the equation. This can be followed by multiple expressions and Parenthesis. The parenthesis are written to identify the expression that belongs together and to make the equations look cleaner. When two variables have the same name they are same otherwise there is no real significance variable alone. Also because of these parenthesis property the above equation can also be written as

$$(((\lambda x.(x y))a)b)$$

(2)

The $\lambda x$ is also called the head of the expression and contains the variable to be operated upon. Thus a simple cut and paste operation will replace x above by ab. So the above equation will become,

$$(a b y)$$

(3)

In general functions are expressions and functions can contains many expressions and hence functions in lambda calculus can contain other functions. Variables in the hear are also called the bound variables. Unmentioned variables are free variables. Following the tutorials on Lambda calculus, I will try to explain more concepts here[1],

3.1  Numbers

Defining numbers in lambda calculus leverages the successive operations on previous calculated numbers or generally speaking expressions.

Let’s say zeros is,

$$0 = (\lambda s z.z)$$

(4)

The expression above throws away the first expression after it and keep evaluates the second one because s is the bounded variable that is not replaced by anything as the body that follows it, does not contain s and only z remains.

Other numbers can thus be calculated as follows,

$$1 = (\lambda s z.s(z))$$
$$2 = (\lambda s z.s(s(z)))$$
$$3 = (\lambda s z.s(s(s(z))))$$

Obviously this is a cumbersome technique to write really big numbers. There are obviously methods like recursion i.e. automating the successor function n times to get nth number or writing functions as expressions to compute big numbers etc., but I won’t go into too much detail on this. Let’s move on to find how can we use these numbers to actually perform some arithmetic operations on them.
3.2 Addition

Knowing the above number calculator we can evaluate sum of two numbers. Let’s take an example, and add number 2 and 3. This will be written in λ calculus as follows,

\[
2 + 5 = 2S5 = 7 \\
= (\lambda sz.s(s(z)))(\lambda abc.b(abc))(\lambda sz.s(s(s(s(z))))) \\
= (\lambda sz.s(s(s(s(s(z)))))�\]

where (\lambda abc.b(abc)) is nothing but the additive function.

3.3 Multiplication

A multiplication function in λ calculus is written as (\lambda abc.a(bc)). As an example let’s consider the following,

\[
2 * 4 = M24 = 8 \\
= (\lambda abc.a(bc))(\lambda sz.s(s(z)))(\lambda sz.s(s(s(z)))) \\
= (\lambda e.((\lambda sz.s(s(s(z))))e)) \\
= (\lambda cz.((\lambda sz.s(s(s(z))))((\lambda sz.s(s(s(z))))c))c)) \\
= (\lambda cz.(c(c(c(c(c(c(c(c(c)))))))))) = 8
\]

Below is also the power function,

\[
Power(b, e) = (\lambda b.(\lambda e.e)b)
\]

3.4 Subtraction

The concept of subtraction involves in knowing that a successor of a number is the next state of that number. In other words x+1 will be the next state of x and there should be a method to store such a state. This is a very long method as each times one has to iterate n times backwards and store the numbers. So even though this might be very inefficient, but remember the condition for a Turing machine. In an infinite space universe and where time is not important this holds true for even the largest number one can think of.

3.5 Other Logical and Conditional Operators

I am going to list all the logical and conditional operators I tried while following the tutorial,

\[
TRUE = (\lambda xy.x) \\
FALSE = (\lambda xy.Y) \\
NOT = (\lambda a.a(\lambda bc.c((\lambda de.d))) \\
AND = (\lambda ab.ab(\lambda xy.y)) \\
OR = (\lambda ab.a(\lambda xy.x)b) \\
IS - ZERO(n) = n FALSE NOT FALSE \\
IS - ZERO(0) = (\lambda sz.z) FALSE NOT FALSE \\
= NOT FALSE \\
= TRUE \\
GREATER - OR - EQUAL(n, m) = (\lambda nm.IS - ZERO(nPm)) \\
EQUAL(n, m) = (\lambda nm.AND(IS - ZERO(nPm)IS - ZERO(mPn)))
\]
the P above in last two operation stands for the Predecessor, i.e. n times the predecessor function on m and vica-versa. If n is exactly the same as m, the result will be 0. Also useful is a tool I found online which can help in lambda reduction calculation[2].

4 C++ Templates

Templates are a feature of C++ that helps creating functions, classes and variables(C++14 for the later) to work with generic data types. This helps the programmer to reuse such a template with various data types and not to rewrite for each. It’s a utility that we will use to leverage lambda calculus and give C++ an inclination towards how functional languages behave[10]. Some programmers also use templates for creating macros and thus are expanded in compile time. The complexity and overhead increases as these templates size increase. As in our case where we practiced implementing these templates using the lambda calculus, there size was pretty large. Here is code snippet that shows the declaration of a C++ function template followed by a class template.

```cpp
// Function Template
template <class type> ret-type func-name(parameter list)
{
    // body of function
}

// Class Template
template <class type> class class-name {
    // body of the class
}
```

4.1 Template Specialization

To have a different implementation for a template when a specific type is passed in the parameter of the template argument. This is also called a template specialization. It can be inferred much like operator overloading concept.

```cpp
// class template
template <class T> class myclass { //body };  
// class specialization
template <> class myclass <char> { //body }; 

int main () {
    myclass<int> myint (10);
    myclass<char> mychar ('m');
}
```

5 Functional Programming in C++

Using all the concepts learned and used above we can leverage the turing complete behaviour of C++ templates to implement lambda calculus. There have been some other attempts in order to recreate functional programming in C++ as described above. One of the early approaches was developing a C++ library for the same. The library is called FC++[4][3]. Exploration and use for this library proved as useful source of introduction for this project. Additional implementation in terms of matrices and machine learning tools in this library could be a beneficial addition for a functional approach in C++[7]. Adopting these kind of libraries might not be tempting for a functional programmer but could be useful for those kind of programmers who might want to exploit the object oriented approach mixed with functional approach.
For the most part of the project I tried implementing Principal Component Analysis of list of numbers and other functions using the templates provided[5]. Because some of the data can be too long such implementation using Lambda Calculus is not efficient and there more modern languages like Haskell’s implementation of State Monad will make more sense. This again is very hard concept to understand but is more efficient in many ways. I also explored the FC++ library and many of its features, though it lacked implementation of lambda calculus which could be a future work. I looked into details of currying in lambda calculus and its implementation.

6 Conclusion

Even though Lambda calculus has its merits when implemented in C++ and can make it behave more like other functional languages it is still deficient in the sense that, some of even easy operations might require a lot of effort and can be inefficient. Debugging the problems was also hard. Given the complexity of identifying the Turing Completeness, Lambda Calculus, State Automata and State Monads, the conceptualization of this area is not trivial and requires deep and proper understanding of the functional languages and its operations. But then again libraries like FC++ and Boost can be improved in order to support these lambda calculus operations. This can also be a future work for this Project.

References