The λ-Calculus

Guest Lecture

Jason Zhang

May 2025

Code++

Table of contents

Introduction

Motivation

We want to understand a form of computation involved in many different forms of programming, especially *declarative* and *functional* programming. Here are some languages that either rely or use elements of the λ-calculus.

1. Haskell
2. Lisp
3. Python
4. Scala
5. Java
6. Much More

Learning about the underlying logic concerning this topic is very important for computer science and mathematics. If you are planning to major in CS you will probably learn about this. Learning about these topics can help with developing new algorithms and design patterns.

Imperative vs. Declarative

In imperative focused languages (C++, C, etc.), there is modification of **state**. For example, consider the simple procedure $x \leftarrow x + 1$. After the procedure runs, x takes on a new value. Imperative languages have their computation modeled by the **Turing Machine**. The Turing Machine runs on an infinite tape of symbols, changing the symbols and moving along it depending on how it was coded.

On the other hand, declarative languages (Haskell, Scala, etc.) are usually stateless. That means, if we assign x to be 3 then it will stay 3. One underlying computation method for functional programming is the **λ-calculus**. It is not the only one.

Theorem (Church, Turing, Kleene)

The λ-calculus and the Turing Machine are equivalent in computational strength.

The Language

Atomic Expressions

There are three notable atomic expressions. There may be a <term> identifier within the expressions. This means that any other valid expression can go inside, allowing us to recursively build expressions.

1. x
2. $\lambda x.<\text{term}>$
3. $(<\text{term}> <\text{term}>)$

Understanding Atomic Expressions

The three previous atomic expressions is defined as follows.

1. Any variable symbol (usually lowercase). Examples x, y, z, etc.
2. The x is again any variable symbol. This defines a function that takes in a parameter x. Therefore, in more familiar mathematical notation, we have $f(x) = <\text{term}>$.
3. This denotes function application. Suppose we see $(M\ N)$. In familiar notation, this reads as $M(N)$. Realize that N could also be a function, and M could return another function.

Basic Rules

β-Reduction

There is one important fundamental way to get from one true expression to another. This is known as **β-reduction**.

β-Reduction

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

This essentially means that all instances of x in M are replaced with N.

Examples

This concept should not be too foreign, we use it all the time! Consider this function $f(x) = x^2 + 2x + 1$. Let us try $f(3)$. We plug in 3 into all instances of x and achieve $f(3) = 3^2 + 2(3) + 1$.

Suppose we had $((\lambda x.x^2 + 2x + 1)\ 3)$. The expression evaluates to $3^2 + 2(3) + 1$ by β-reduction.

Alert

While this example is nice, it should be noted that in real λ-calculus we are not free to use things like $+$ or even 3. We must define everything.

α-conversion

While not as important, it is still important to note.

α-conversion

$(\lambda x.M[x]) \rightarrow_\alpha (\lambda y.M[y])$

This simply means we can change the names of variables around as long as we change it everywhere. There are some ways of doing λ-calculus without needing α-conversion.

Functions

Functions Functions Functions

Functions are **anonymous** in λ-calculus, meaning they don't have names. Although you could give them names if you really wanted to.

One other important thing is that pretty much everything is a function. Yes, that includes numbers. There is a notion of applying the function 172 onto $\times$ and it would give you something. The result probably doesn't make sense, but the computation never made sense anyways.

Multiple Arguments and Currying

Sometimes, we will work with functions that have multiple arguments. This is already possible without any modifications. Suppose instead of writing $f(x, y)$, you had $(g(x))(y)$ where $g(x)$ returns a function that has the value of x fixed. The equivalent in λ-calculus is left as an exercise.

Writing this can be a laborious task. Thankfully, we can cheat by doing what is known as "currying". We will simply modify our language and write two argument functions as $\lambda xy.M$. Function applications will be $((\lambda xy.[\ldots])\ M\ N)$ where M replaces x first and then N replaces y via β-reduction.

Booleans

We will begin with a simple exercise in regular boolean algebra.

1. What does $\top$ and $\bot$ mean?
2. What do the following operations do $\wedge, \vee, \neg$?
3. Name a universal logic gate.
4. Consider the if statement of the form $p \to q$. Rewrite this using only logic gates.

This is important when we construct our logic in λ-calculus.

True and False

We will define the True and False functions as follows.

True ($\top$)

$\lambda xy.x$

False ($\bot$)

$\lambda xy.y$

The motivation of which will become clear later. We will use *TRUE* and *FALSE* in place of each function to make it less verbose.

Negation

We will construct our negation function as follows.

Not ($\neg$)

$\lambda x.(x \;\; \mathit{FALSE} \;\; \mathit{TRUE})$

We will call this *NOT* for obvious reasons.

An example application onto *TRUE*.
$((\lambda x.(x \;\; \mathit{FALSE} \;\; \mathit{TRUE}))\; \mathit{TRUE}) \to_\beta (\mathit{TRUE} \;\; \mathit{FALSE} \;\; \mathit{TRUE}) \to_\beta \mathit{FALSE}$. This works because *TRUE* picks out the first element, which is *FALSE*.

Exercises:

1. Rewrite *NOT* using only λ-expressions.
2. Verify that $((\lambda x.(\mathit{NOT} \;\; (\mathit{NOT} \;\; x)))\; M) \to_\beta M$ assuming M is either *TRUE* or *FALSE*.
3. Create the other logic gates.

Numbers

We will define numbers using the *Church Encoding*.

Church Encoding

For any number $n \in \mathbb{N}$, $n = \lambda fx.(\underbrace{f\ (f \ldots (f}_{n \text{ times}}\ x) \ldots))$.

This definition will be useful later. This definition does not define numbers, but simply n times applying a function. A special case of this function is a number.

Successor

The function succ informally means $+1$.

Using this, any number n is equivalent to $n\ \mathrm{succ}\ 0$ since the function succ is applied to 0 n times.

Note that succ does have a formal definition using purely the λ-calculus.

A Reminder

Remember that everything here is a function! So we could compute some crazy things such as $+ \times +$ (this notation is how you would usually write it). However, doing this takes a very long time.

Recursion

There is no function that can reference itself (directly call itself), but there are functions that can do basically the same thing.

We can choose between the Y-combinator or the Θ combinator. These are basically fixed point functions. A fixed point in regular math occurs at $x = c$ if $f(c) = c$.

Likewise, $\Theta\ f \rightarrow_\beta f\ (\Theta\ f)$, as an example.

Applications and Beyond

Applications

As previously noted, there are many applications of the λ-calculus. It is the foundation of languages like Haskell.

On the other hand, it comes quite handy in languages like Python. Consider the following expression:

```
lambda x : x + 1
```

This is just $\lambda x.x + 1$. Additionally, knowing the λ-calculus allows for a strong background in discrete mathematics (a course required in most colleges for CS) and gives you better knowledge on creating new algorithms.

Higher Order

One last thing to note is that there are many variations and extensions of the λ-calculus. Here is a list:

1. Typed lambda calculus
2. System F
3. Kappa calculus
4. More!