Untyped Lambda Calculus Notes

Yipeng Liu

Last updated: **October 27, 2024**

Contents

1. Basics

Lambda calculus is a formal system for expressing computations. Untyped lambda calculus, can be seen as a computation model alternative to Turing machines, in which all computation is reduced to function application and abstraction. It is also a simple programming language used widely as a core language in many complex programming language implementations. It is so simple yet powerful enough to express any computation as mathematical object, in which properties can be proved easily. Extension to basic lambda calculus brings complex feature such as mutable reference, exception handling, and type system etc.

1.1. Introduction

The key idea is that in lambda calculus, **everything** is a function. Arguments passed to functions are function, and result returned by a function is also function. For example, assuming that the language has integer, `if` expression, and recursion (those features can be formulated as functions, we will see!), then `factorial` can be defined as follows:

$$\text{factorial} = \lambda n.\ \text{if } (n == 0)\ 1\ (n * \text{factorial } (n - 1))$$

Mathematically, the factorial function is defined as:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n - 1) & \text{otherwise} \end{cases}$$

The abstraction, $\lambda n.\ ...$, is the same as writing a function that takes an argument `n` and returns the result of the expression.

1.2. Syntax

The syntax of untyped lambda calculus consists of only three type of terms: a *variable* x, an *abstraction* of a *variable* from a term t, written as $\lambda x.\ t$, and an *application* of term a term t_1 to another term t_2, written as $t_1 t_2$. Formally:

Terms — Definition

$$\begin{aligned} t \Coloneqq\ & x && \text{(variable)} \\ \mid\ & \lambda x.\ t && \text{(abstraction)} \\ \mid\ & t\ t && \text{(application)} \end{aligned}$$

We usually use t, s, u, etc. to denote an arbitrary term. Similarly, x, y, z, etc. are used to denote arbitrary variables. Applications are left-associative, so $t\ s\ u$ is the same as $((t\ s)\ u)$. Body of abstraction is extended as far as possible, as the result $\lambda x.\ \lambda y.\ x\ y$ is the same as $\lambda x.\ (\lambda y.\ (x\ y))$.

1.3. Scope

An occurrence of a variable x in a term t is said to be *bound* if it is within the scope of an abstraction $\lambda x.\ t$. On the other hand, an occurrence of a variable x is said to be *free* if it appears in a term t but is not bound. For example, occurrences of x in $\lambda y.\ x$ and $x\ y$ are free, while those in $\lambda x.\ \lambda y.\ (x\ y)$ and $\lambda x.\ x$ are bound. In $(\lambda x.\ x)\ x$, the first occurrence of x is bound, while the second is free.

Free Variables — Definition

The set of *free* variables of a term t is denoted as $\text{FV}(t)$:

$$\begin{aligned} \text{FV}(x) &= \{x\} \\ \text{FV}(\lambda x.\ t) &= \text{FV}(t) \setminus \{x\} \\ \text{FV}(t\ s) &= \text{FV}(t) \cup \text{FV}(s) \end{aligned}$$

1.4. Evaluation

Pure lambda calculus does not have constants or primitives – we can not perform arithmetic operations, loops, or printing. The only thing we can do is to apply functions to arguments.

1.4.1. Beta Reduction

We call a term of the form $(\lambda x.\ t)\ s$ a *redex* (reducible expression), and the process of replacing the redex with the result of the application is called *beta reduction*:

$$(\lambda x.\ t)s \xrightarrow{\beta} [x \mapsto s]t$$

where $[x \mapsto s]t$ denotes the result of replacing all **free** occurrences of x in t with s. For example, $(\lambda x.\ x)\ y$ evaluates to y, and $(\lambda x.\ x\ (\lambda x.\ x))\ s$ evaluates to $s\ (\lambda x.\ x)$. Formally, we define the capture-avoiding substitution as follows:

Substitution — Definition

$$\begin{aligned}
&[x \mapsto s]x &&= s \\
&[x \mapsto s]y &&= y && \text{if } x \neq y \\
&[x \mapsto s](\lambda y.\ t) &&= \lambda y.\ [x \mapsto s]t && \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(s) \\
&[x \mapsto s](t\ u) &&= [x \mapsto s]t\ [x \mapsto s]u
\end{aligned}$$

1.4.2. Alpha Conversion

Consider $\lambda x.\ x$ and $\lambda y.\ y$ – do they represent the same function? Yes! Although variable names are different, the meaning of the terms are the same. We say that two terms are *alpha-equivalent* if they are the same up to renaming of bound variables. And it is intuitive that variable names can be changed without changing the meaning of the term. This operation is called *alpha-conversion*:

$$(\lambda x.\ t) \xrightarrow{\alpha} (\lambda y.\ [x \mapsto y]t) \quad \text{if } y \notin \mathrm{FV}(t)$$

where x in the binder is replaced with y in the body of the abstraction.

Sometimes we need to rename variables to avoid variable capture in evaluation. For example, let's evaluate the following term:

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

Notice that we can not replace x in the inner lambda with $(\lambda x.\ x\ y)$ directly, because y is already bound in the inner lambda, but free in the outer term. Then we need to do *alpha-conversion*, or rename y in the inner lambda to avoid variable capture. Let's rename y to z. Then the term becomes:

$$(\lambda x.\ (\lambda y.\ x\ y))\ (\lambda x.\ x\ y) \xrightarrow{\alpha} (\lambda x.\ (\lambda z.\ x\ z))\ (\lambda x.\ x\ y)$$

Now we can do beta reduction, replacing x in the inner lambda with $\lambda x.\ x\ y$:

$$(\lambda x.\ (\lambda z.\ x\ z))\ (\lambda x.\ x\ y) \xrightarrow{\beta} (\lambda z.\ (\lambda x.\ x\ y)\ z) \xrightarrow{\beta} \lambda z.\ z\ y$$

When picking new name for variables, we need to make sure that the new variable name is not already used in the term, i.e. it does not capture any free variables in the term. For example, it is **incorrect** to rename y to z in $\lambda z.\ \lambda y.\ (z\ y)$, because z is *free* in $(z\ y)$.

1.4.3. Eta Reduction

Eta reduction captures the nature of function extensionality: two functions are equal if they give the same result for all arguments. For example, $\lambda x.\ f x$ is the same as f if x is not free in f. Formally, we define eta reduction as:

$$\lambda x.\, t\ x \xrightarrow{\eta} t \quad \text{if } x \notin \mathrm{FV}(t)$$

1.4.4. Evaluation Strategy

When evaluating a term, we can choose different strategies to perform beta reduction, i.e. which redex to reduce first. A term is *strongly normalizing* if every reduction sequence terminates in a normal form, and *weakly normalizing* if there exists a reduction sequence that terminates in a normal form, but not all reduction sequences terminate. It has been proven that in untyped lambda calculus, normalizing terms always have a unique normal form, but not all terms are normalizing. Thus, different evaluation strategies may lead to different results. Let's consider several evaluation strategies:

Full beta reduction Any redex can be reduced at any time.

Consider the following term, which has three redexes:

$$\underline{(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))}$$
$$(\lambda x.\ x)\ \underline{((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))}$$
$$(\lambda x.\ x)\ \Big((\lambda x.\ x)\ \Big(\lambda z.\ \underline{(\lambda x.\ x)\ z}\Big)\Big)$$

We can reduce the innermost first, and then do the middle one, and finally the outermost one:

$$\begin{aligned}\underline{(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))} &\longrightarrow \underline{(\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)}\\ &\longrightarrow \underline{(\lambda x.\ x)\ (\lambda z.\ z)}\\ &\longrightarrow \lambda z.\ z\end{aligned}$$

Normal order reduction Always reduce the leftmost, outermost redex first.

Let's do the same term with normal order reduction:

$$\begin{aligned}\underline{(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))} &\longrightarrow \underline{(\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)}\\ &\longrightarrow \lambda z.\ \underline{(\lambda x.\ x)\ z}\\ &\longrightarrow \lambda z.\ z\end{aligned}$$

We get the same result using different reduction sequences!

Call by name Do normal order reduction, but do not reduce inside abstractions.

$$\begin{aligned}\underline{(\lambda x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z))} &\longrightarrow \underline{(\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)}\\ &\longrightarrow \lambda z.\ (\lambda x.\ x)\ z\end{aligned}$$

This is the normal form! We can not reduce it further because we don't reduce inside abstractions.

Call by value Only reduce outermost redexes and a redex is reduced only when its right-hand side is in normal form.

$$\begin{aligned}(\lambda x.\ x)\ \underline{(\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)} &\longrightarrow \underline{(\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)}\\ &\longrightarrow \lambda z.\ (\lambda x.\ x)\ z\end{aligned}$$

Note that we can not reduce further $(\lambda z.\ (\lambda x.\ x)\ z)$, the right-hand side after the first step, as it is in normal form. We don't reduce inside abstractions as in call by name. Call by value is strict in the sense that arguments are evaluated before the function is applied. Sometimes it fails to find the normal form, even if it exists.

We have seen that in call by name and call by name, $\lambda z.\ (\lambda x.\ x)\ z$ is regarded as normal form, while there is still a redex in it. On the other hand, in normal order reduction, since we can reduce the redex inside the abstraction, we get the normal form $\lambda z.\ z$. If a term has beta normal form (no beta reduction possible), then normal order reduction will find it. The term above evaluates to different results depending on the evaluation strategy, but eventually they all lead to the same normal form as the one yielded by normal order reduction after some reductions. This renders *Church-Rosser theorem*, or the *confluence property*, which states that if a term can be reduced to two different normal forms, then those normal forms are equivalent up to alpha-conversion.

1.4.5. Operational Semantics

Operational semantics is a formal way to describe how the evaluation of a term proceeds. Let's adopt call by value evaluation strategy. We can define the evaluation relation $t \to t'$:

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2}\ \text{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 t_2 \longrightarrow v_1 t_2'}\ \text{E-App2}$$

$$\frac{}{(\lambda x.\ t_{12}) v_2 \longrightarrow [x \mapsto v_2] t_{12}}\ \text{E-AppAbs}$$

v denotes a value, which is a term that cannot be reduced further. In our case, a value is an abstraction $\lambda x.\ t$. E-App1 applies to any application whose left-hand side is not a value, while E-App2 applies to any application whose right-hand side is not a value. That means, E-App1 is always applied before E-App2 – we use E-App1 to reduce t_1 to a value, then use E-App2 to reduce t_2 to a value, and finally use E-AppAbs to reduce the application. The evaluation order is determined by the rules! If we instead use call by name, we can define the evaluation relation as follows:

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2}\ \text{E-App1}$$

$$\frac{}{(\lambda x.\ t_{12}) t_2 \longrightarrow [x \mapsto t_2] t_{12}}\ \text{E-AppAbs}$$

Can you see the difference? In call by name, we do not reduce the right-hand side before applying the function. This is where lazy evaluation comes from! Moreover, in E-AppAbs we no longer require the right-hand side to be a value.

2. Programming in Lambda Calculus

Now that we have a basic understanding of lambda calculus, we can start programming in it! We will see how to define various data structures and functions using lambda calculus.

2.1. Functions of more arguments

Observe that lambda calculus only allows functions of one argument, but functions of more arguments can be obtained by *currying*. For example, a function of two arguments, $f(x, y)$, can be represented as

$(\lambda x.\ (\lambda y.\ f\ x\ y))$. Removing unnecessary parentheses, we can write it as $\lambda x.\ \lambda y.\ f\ x\ y$. A function taking two arguments is a function that takes one argument and returns a function that takes another argument!

2.2. Church Boolean

Boolean values and conditionals can be easily defined in lambda calculus. We define `true` and `false` as follows:

$$\begin{aligned} \text{true} &= \lambda x.\ \lambda y.\ x \\ \text{false} &= \lambda x.\ \lambda y.\ y \end{aligned}$$

And we define `if` function as:

$$\text{if} = \lambda b.\ \lambda x.\ \lambda y.\ b\ x\ y$$

This function does not do much: it just applies two arguments to the boolean value. Let's see how it works:

$$\begin{aligned} \text{if true}\ u\ v &= \underline{(\lambda b.\ \lambda x.\ \lambda y.\ b\ x\ y)\ \text{true}}\ u\ v \\ &\longrightarrow \underline{(\lambda x.\ \lambda y.\ \text{true}\ x\ y)\ u}\ v \\ &\longrightarrow \underline{(\lambda y.\ \text{true}\ u\ y)}\ v \\ &\longrightarrow \text{true}\ u\ v \\ &= \underline{(\lambda x.\ \lambda y.\ x)\ u}\ v \\ &\longrightarrow \underline{(\lambda y.\ u)}\ v \\ &\longrightarrow u \end{aligned}$$

if true $u\ v$ evaluates to u! Similarly, we can define other boolean functions:

$$\begin{aligned} \text{and} &= \lambda b.\ \lambda c.\ b\ c\ \text{false} \\ \text{or} &= \lambda b.\ \lambda c.\ b\ \text{true}\ c \\ \text{not} &= \lambda b.\ b\ \text{false true} \end{aligned}$$

Those definitions are not unique, and we can define them in many ways. The insight here is that a church boolean is a function that takes two arguments and returns one of them – `true` returns the first argument, and `false` returns the second argument. Keeping this in mind, it is easy to see that why the above definitions make sense. For example, the function `and`, takes two boolean values b and c, and returns c if b is `true`, and `false` otherwise. Thus, it returns `true` if both b and c are `true`, and `false` if either of them is `false`.

2.3. Pairs

Using boolean, we can define pairs as well.

$$\begin{aligned} \text{pair} &= \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s \\ \text{fst} &= \lambda p.\ p\ \text{true} \\ \text{snd} &= \lambda p.\ p\ \text{false} \end{aligned}$$

Let $p = \text{pair}\ u\ v$, then fst p evaluates to u, and snd p evaluates to v. Let's try:

$$\begin{aligned}
\text{fst}\ (\text{pair}\ u\ v) &= \text{fst}\ \Big(\underline{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s) u}\ v\Big) \\
&\longrightarrow \text{fst}\ \Big(\underline{(\lambda s.\ \lambda b.\ b\ u\ s)\ v}\Big) \\
&\longrightarrow \text{fst}\ (\lambda b.\ b\ u\ v) \\
&= \underline{(\lambda p.\ p\ \text{true})\ (\lambda b.\ b\ u\ v)} \\
&\longrightarrow \underline{(\lambda b.\ b\ u\ v)\ \text{true}} \\
&\longrightarrow \underline{\text{true}\ u\ v} \\
&\overset{*}{\longrightarrow} u
\end{aligned}$$

Do you see how the definition of boolean helps us to define pairs?

2.4. Church Numerals

Natural numbers can be represented in lambda calculus as *Church numerals*:

$$\begin{aligned}
c_0 &= \lambda f.\ \lambda x.\ x \\
c_1 &= \lambda f.\ \lambda x.\ f\ x \\
c_2 &= \lambda f.\ \lambda x.\ f\ (f\ x) \\
c_3 &= \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \\
&\ldots \\
c_n &= \lambda f.\ \lambda x.\ f^n\ x
\end{aligned}$$

A Church numeral c_n is a function that takes two arguments, a function f and a value x, and applies f to x n times. For example, c_2 is a function that takes a function f and a value x, and applies f to x twice.

Successor

We define the successor function as:

$$\text{succ} = \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$$

It takes a Church numeral n and returns another Church numeral that represents the successor of n, i.e. applying f to x one more time than n does. Let's try succ c_2:

$$\begin{aligned}
\text{succ}\ c_2 &= \underline{(\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x))\ c_2} \\
&\longrightarrow \lambda f.\ \lambda x.\ f\ (c_2\ f\ x) \\
&\longrightarrow \lambda f.\ \lambda x.\ f\ \Big(\underline{(\lambda f.\ \lambda x.\ f\ (f\ x))\ f}\ x\Big) \\
&\longrightarrow \lambda f.\ \lambda x.\ f\ \Big(\underline{(\lambda x.\ f\ (f\ x))\ x}\Big) \\
&\longrightarrow \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \\
&= c_3
\end{aligned}$$

Addition

Then we define addition:

$$\text{plus} = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

It takes two Church numerals m and n, and returns another Church numeral that represents the sum of m and n. If we had $(m\ f\ x)$, it would apply f to x m times. Now we have $(m\ f\ (n\ f\ x))$, which applies f to $(n\ f\ x)$ m times, where $(n\ f\ x)$ applies f to x n times. Thus, it applies f to x $m + n$ times!

Multiplication

What about multiplication?

$$\text{times} = \lambda m.\ \lambda n.\ m\ (\text{plus}\ n)\ c_0$$

Here we use another trick: plus n is a function that takes Church numeral m and returns $m + n$. Passing it as the first argument to m will apply it m times on c_0 – that is, it will add n to 0 m times, which is $m * n$. alternatively, we can define multiplication as:

$$\text{times} = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x$$

Can you see the similarity between the two definitions?

Exponentiation

For m^n, we define:

$$\exp = \lambda m.\ \lambda n.\ n\ m$$

A very short definition that looks weird. To see how it works, let's suppose n is a Church numeral. Then we have

$$\begin{aligned} n\ f\ x &= f^n\ x \\ n\ f &= f^n \qquad (\eta\text{ -reduction}) \end{aligned}$$

Let's replace f with m:

$$n\ m = m^n$$

We discover that $n\ m$ is the same as m^n! Thus our definition does the right thing.

Predecessor

Surprisingly, predecessor is the hardest function to define. It is not clear how to remove one application of f from x. One trick is to use a pair of numbers $(n - 1, n)$ to represent n, and then we can extract the predecessor from the pair.

$$\begin{aligned} \text{zz} &= \text{pair}\ c_0\ c_0 \\ \text{ss} &= \lambda p.\ \text{pair}\ (\text{snd}\ p)\ (\text{succ}\ (\text{snd}\ p)) \\ \text{pred} &= \lambda n.\ \text{fst}\ (n\ \text{ss}\ \text{zz}) \end{aligned}$$

When `ss` is applied to a pair, it returns a pair of the second element of the pair and the successor of the second element. Thus, `ss` is a function that takes a pair $(n - 1, n)$ and returns $(n, n + 1)$. Then, `pred` takes a Church numeral n and returns the first element of the pair $(n - 1, n)$, which is $n - 1$.

2.5. Recursion

There is no built-in recursion in lambda calculus, because when defining a term, we can not refer itself in its body. If we do not have express loop/recursion in a language, then the language can not express any computation. Here in untyped lambda calculus, we can define recursive functions using fixed-point combinators.

Recall that in untyped lambda calculus, some terms are not normalizing, i.e. they do not have a normal form. Consider the following term called the *Omega combinator*:

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

It has only one redex, and it reduces to itself:

$$\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)} \longrightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

No matter how many times we apply beta reduction, we will never reach a normal form. Let's consider another term called the *Y combinator*:

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

It is hard to see what this term does by looking at its definition. It is also called *fixed-point combinator*, and it can be used to define recursive functions in lambda calculus. Let's try applying Y to some function g:

$$\begin{aligned} Yg &= \underline{(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g} \\ &\longrightarrow \underline{(\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))} \\ &\longrightarrow g\ (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)) \\ &= g\ (Yg) \end{aligned}$$

Yg evaluates to $g\ (Yg)$! However, in the setting of call by value, Yg diverges because it tries to evaluate Yg before applying g to it. There is another fixed-point combinator called the *Z combinator* that works in call by value:

$$Z = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

Actually it is an eta expansion of Y, and because of the argument v, $Z\ g$ in the right-hand side will not be evaluated eagerly in call by value language. Let's see how it works:

$$\begin{aligned} Z\ g\ v &= \underline{(\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y)))\ g}\ v \\ &\longrightarrow \underline{(\lambda x.\ g\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ g\ (\lambda y.\ x\ x\ y))}\ v \\ &\longrightarrow g\ (\lambda y.\ (\lambda x.\ g\ (\lambda k.\ x\ x\ k))(\lambda x.\ g\ (\lambda k.\ x\ x\ k))\ y)\ v \\ &\overset{\eta}{\longrightarrow} g\ (\lambda x.\ g\ (\lambda k.\ x\ x\ k))(\lambda x.\ g\ (\lambda k.\ x\ x\ k))\ v \\ &= g\ (Z\ g)\ v \end{aligned}$$

It behaves the same way as Yg. In real world programming languages, fixed point combinators are usually called `fix`. To define a recursive function, first we define a function $g = \lambda f.\ \ldots$ where the body contains a reference to f, and then we apply `fix` to g to get the recursive function. For example, to define the factorial function, we can define g as:

$$\begin{aligned} &g = \lambda f.\ \lambda n.\ \text{if}\ (n == c_0)\ \text{then}\ c_1\ \text{else}\ (n * f(n-1)) \\ &\text{factorial} = \text{fix}\ g \end{aligned}$$

assuming that we have the multiplication, subtraction, equality, and if expression. Now we gained the ability to define recursive functions in lambda calculus!

3. References

- Pierce, B. C. (2002). Types and Programming Languages
- Felleisen, M., & Flatt, M. (2006). Programming Languages and Lambda Calculi