λ Calculus

Abstract

This series of notes is intended as an introduction to the λ- calculus. But it is most likely a series of ramblings as I go about learning it as I intend to include all of my explorations in it as well. For example, I go over implementing a simple evaluator (and perhaps a better one later for practical use). This is a more revised and complete edition of my notes for the Math+ mentorship program hosted by the University of Toronto. With a special thanks to my mentor, Luke, and to Peter Selinger whose book on the topic is the source of much of this series.

Introduction

Syntax

The λ-calculus has a formal syntax which can be described using the following definitions. The first one is easier to understand but we might use the second one later.

Definition 1. *Given an infinite set, V, of variables, and let Λ be the set of all λ-expressions. Then:*

- *For $x \in V$, then $x \in \Lambda$*
- *For $M, N \in \Lambda$ then $(MN) \in \Lambda$*
- *For $x \in V$, and $M \in \Lambda$, then $\lambda x.M \in \Lambda$*

Definition 2. *Given an infinite set V of variables. The set of all λ-terms, Λ, is given by the following BNF:*

$$M, N ::= x \mid (MN) \mid \lambda x.M$$

These might be too difficult to understand. So I will attempt to describe them in words. There are three kinds of λ-expressions.

1. *Variables.* The set V in the pervious definitions. For example x, y, etc.
2. *Combinators/Functions.* A function, it takes an input and returns some value based on it. Denoted by λ[args].[expr]. For example $\lambda x.x$.
3. *Applications.* Applying two λ expressions to eachother. Denoted by MN. For example $(\lambda x.x)(\lambda y.y)$.

Evaluation

Before we can get to any of the interesting aspects of the λ-calculus, we must learn how to evaluate λ-expressions. We'll go over that in this section and — by taking a page out of Gerald Sussman's book— implement an evaluator to do these for us.

α-EQUIVALENCE

We wish to see whether two λ-terms are equal or not. In traditional mathematics we might say that two functions with identical domains and codomains are equal. However in the λ-calculus we have no such concepts. So we have to compare the rules by which our input is manipulated into our desired output. If this is identical in two λ-terms, then we can say that they are equal. Let such an operator be called α-equivalence.

Definition 3. *An occurance of a variable, x, in $\lambda x.N$ is said to be bound. And the corresponding λx is called a binder. A variable that's not bound is called free. The set of all free variables of a term is defined as below:*

- $FV(x) = \{x\}$,
- $FV(MN) = FV(M) \cup FV(N)$,
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$

If we attempt to formally define this concept, we quickly reach a problem. $\lambda x.x$ and $\lambda y.y$ are clearly expressing the same rule. Which is to say that they only differ in their bound variable. Informally we may call two λ-terms that only differ in their bound variables to be α-equivalent. This too, is hard to define formally. We need a *renaming* operation to account for differing bound variables. Such an operation is defined as follows.

Definition 4. *For some variables x and y, and a term M. $M\{y/x\}$– renaming x as y– is as follows:*

- $x\{y/x\} \equiv y$
- $z\{y/x\} \equiv z$ *if* $x \neq z$
- $(MN)\{y/x\} \equiv (M\{y/x\})(N\{y/x\})$
- $(\lambda x.M)\{y/x\} \equiv \lambda z.(M\{y/x\})$ *if* $x \neq z$

Now we are fully capable of defining α-equivalence. The following is essentially a formal writing of our previous definition.

Definition 5. *α-equivalence is the smallest conguent relation $=_\alpha$ on λ-terms, such that for all terms M and all variables $y \notin M$,*

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\})$$

β-reduction

In the previous section, we defined a renaming operation to replace a variable in a λ-term. Which in turn allowed us to define what it means for two λ-terms to be equal to one another. In this section we wish to discuss how we might simplify λ-expressions. Lets call this β-reduction.

In normal mathematics, evaluating $f(x) = x^2$ at $x = a$ is quite simple. You substitute all instances of x with a and calculate the results. Unlike last time, this works in the λ-calculus as well. So before we can formally define β-reduction, we must first define substitution, which allows us to replace a variable by a λ-term. There are two problems with defining such an operation.

1. We should only replace *free* variables. The names of bound variables are out of our scope and should not be changed. For example $x(\lambda xy.x)[N/x]$ is $N(\lambda xy.x)$ and not $N(\lambda xy.N)$
2. We need to avoid unintended "capture" of free variables. For example, let $M \equiv \lambda x.yx$ and $N \equiv \lambda z.xz$. Note that x is free in N but bound in M. If we do $M[N/y]$ naively we get $\lambda x.Nx = \lambda x.(\lambda z.xz)x$. However, since x is bound only to M, the x in M and the x in N are not the same. So we must rename the bound variables before the substitution.

$$M[N/y] = (\lambda x'.yx')[N/y] = \lambda x'.Nx' = \lambda x'.(\lambda z.xz)x'$$

Definition 6. *The substitution of N for free occurances of x in M, in symbols $M[N/x]$, is defined as follows:*

- $x[N/x] \equiv N$
- $y[N/x] \equiv y$ *if* $x \neq y$
- $(MP)[N/x] \equiv (M[N/x])(P[N/x])$
- $(\lambda x.M)[N/x] \equiv \lambda x.M$
- $(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x])$ *if* $x \neq y$ *andy* $\notin FV(N)$
- $(\lambda y.M)[N/x] \equiv \lambda y'.(M\{y'/y\}[N/x])$ *if* $x \neq y, y \notin FV(N)$ *and* y' *is fresh*

A term of the form $(\lambda x.M)N$ is called a *β-redex*. It reduces to $M[N/x]$, which is called the *reduct*. A λ-term without any β-redexs is in *β-normal form*. For example:

$$\begin{aligned}(\lambda x.y)((\lambda z.zz)(\lambda w.w)) &\to_\beta (\lambda x.y)((\lambda w.w)(\lambda w.w)\\ &\to_\beta (\lambda x.y)(\lambda w.w)\\ &\to_\beta y\end{aligned}$$

And since y has no redexes it is in normal form. We could've also just looked at $\lambda x.y$ and realized that all the arguments are uselss. The key take aways are (1) reducing a redex can create new redexes, (2) reducing a redex can delete some other redexes, (3) the number of steps can vary. However, not all terms evaluate to a normal form. Some can just keep reducing forever without reaching a normal form.

If M and M' are terms such that $M \twoheadrightarrow_\beta M'$ and if M' is in normal form we say that M evaluates to M'. Now we are able to formally define β-reduction.

Definition 7. *We define single-step β-reduction to be the smallest relation $\to_\beta$ on terms satisfying:*

$$(\beta) \qquad \frac{}{(\lambda x.M) \to_\beta M[N/x]}$$

$$(cong_1) \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$(\zeta) \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

$$(cong_2) \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}$$

Definition 8. *We write $M \twoheadrightarrow_\beta M'$ if M reduces to M' in zero or more* *steps. Formally, $\twoheadrightarrow_\beta$ is defined to be the reflexive transitive closure of $\rightarrow_\beta$, i.e., the smallest reflexive transitive relation containing $\rightarrow_\beta$.*

And by allowing $\rightarrow_\beta$ to be symmetric, we can define β-equivalence.

Definition 9. *We write $M =_\beta M'$ if M can be transformed into M' by zero or more reduction steps and/or inverse reduction steps. Formally. $=_\beta$ is defined to be the reflec symmetric transitive closure of $\rightarrow_\beta$.*

Reperesenting Data

This chapter is an outline of how you might represent data in the λ-calculus. Two types, booleans and the natural numbers are presented here. For more, check the appendices.

BOOLEANS

Booleans are quite simple to implement in the λ-calculus. We want to find "switches" in the λ-calculus. Functions that can only be in two states. For example, function that takes two arguments, must either return the first, or the second. So let's define $\mathbf{T} = \lambda xy.x$ and $\mathbf{F} = \lambda xy.y$.

Using these simple definition we can build our familiar logic gates. For example, the not function reverses it's input. And since we can pick what we return using **T** and **F**, it is quite easy to define not.

$$\mathbf{not} = \lambda a.a\mathbf{FT}$$

a is either **T**, or **F**. If a is **T**, then we want to let our first argument to a be **F** (since **F** is the inverse of **T**). And if a is equal to F, our second argument should be **T**.

Exercise 1. *Find **and**, **or** functions that work with our representation of booleans.*

Exercise 2. *Find an alternative encoding for booleans. Find the corresponding logic gates.*

Exercise 3. *Implement three-valued logic*[1] *in the λ-calculus.*

NATURAL NUMBERS

The implementation of the natural numbers is very similar to the peano axioms. It is merely the application of a function multiple times.

Definition 10. *A number, $n \in \mathbb{N}$, is represented in the λ-calculus as a function that applies it's first argument n times to the next. Such numbers are called* ***Church Numerals****. For example:*

$$\begin{aligned} 0 &= \lambda so.s \\ 1 &= \lambda so.so \\ 2 &= \lambda so.s(so) \\ 3 &= \lambda so.s(s(so)) \\ 4 &= \lambda so.s(s(s(so)) \\ n &= \lambda so.s^n(o) \end{aligned}$$

We can also define a "$f(x) = x+1$" function. Also called a successor function. Remember that adding one is the samething as applying s to o one more time in $\lambda so.s^n(o)$.

Theorem 1 (Successor Function)**.** *Let $S = \lambda fmx.m(fmx)$. For all Church numerals $N = \lambda so.s^n(o)$, $SN = \lambda so.s^{n+1}(o)$.*

[1] Logic with three values instead of two.

Proof. This is a simple induction proof. Our base case is $(\lambda fmx.m(fmx))(\lambda s$ $\lambda mx.m((\lambda so.o)mx) \rightarrow_\beta \lambda mx.mx$ which is equal to 1. Then our inductive step is:

$$\begin{aligned}(\lambda fmx.m(fmx))(\lambda so.s^n(o)) &\rightarrow_\beta \lambda mx.m((\lambda so.s^n(o))mx) \\ &\rightarrow_\beta \lambda mx.m(m^n(x)) \\ &\twoheadrightarrow_\beta \lambda mx.m^{n+1}(x)\end{aligned}$$

Q.E.D.

Exercise 4. *(a) Prove $\lambda nmfx.nf(mfx)$ is addition. (b) Prove that $MN \twoheadrightarrow_\beta M \times N$. (c) Prove that $\lambda nmf.n(mf)$ is multiplication.*

Let's also cover how we might use booleans in combination with church numerals. For example, let's define a function that will return **T** if it's input is 0.

Theorem 2. *The function* **zero**? $= \lambda nxy.n(\lambda x.y)x$ *will return* ***T*** *iff* $n = \lambda so.o$.

Proof. The key here is to realize that if n is $\lambda so.o$, then it will ignore $\lambda x.y$ and return $\lambda xy.x$ which is what we want. However if x is not 0, it will apply $\lambda x.y$ to x a number of times, which won't matter because $\lambda x.y$ returns y regardless of it's argument. Meaning that the output of the whole function would be $\lambda xy.y$ which is also what we want. Q.E.D.

Exercise 5. *Prove theorem 2 inductively.*

Exercise 6. *Create an equality combinator that will return* ***T*** *iff it's two arguments are equal Church numerals. And* ***F*** *if otherwise.*

Exercise 7. *(a) Implement a pair data structure in the λ-calculus with functions to retrieve each element of the pair. (b) Implement the integers in the λ-calculus along with the appropriate functions (+, −, etc...) (c) Implement the rationals in the λ-calculus along with all apropriate functions.*

Solution. (a) Our **pair** data structure is: **pair** $= \lambda pqc.cpq$. To retrieve the first element, we can use the **fst** function: **fst** $= \lambda p.p\mathbf{T}$ and we can use the **snd** function to retrieve the second element: **snd** $= \lambda p.p\mathbf{F}$.

(b) Integers are simply signed naturals. Therefore we will use a pair to represent them. **int** $= \lambda sn.(\mathbf{pair}\ sn)$. Where s is the sign and n is a natural number.

(c) Rationals are defined as $\left\{\frac{p}{q} \mid \forall p, q \in \mathbb{Z}\right\}$. and so we can represent them as pairs. **rat** $= \lambda nd.(\mathbf{pair}\ nd)$. Where n is the numerator and d is the denominator.

Q.E.D.

Fixed Points and Recursion

Definition 11. *A **fixed-point**, is some x, such that $f(x) = x$. In λ-calculus notation, this would be $FX =_\alpha X$.*

Theorem 3 (Turing Fixed-Point Combinator)**.** *In the untyped λ-calculus, every term, F, has a fixed point.*

Proof. Let $A = \lambda xy.y(xxy)$, and define $\Theta = AA$. Suppose F is any

λ-term, and let $N = AAF = \Theta F$. Therefore:

$$\begin{aligned} N &= \Theta F \\ &= (\lambda xy.y(xxy))AF \\ &\twoheadrightarrow_\beta F(AAF) \\ &= FN \end{aligned}$$

Θ is known as the *Turing Fixed Point Combinator*. Q.E.D.

Fixed points are rather powerful tools. Finding a fixed point is equivalent to solving the equation $x = f(x)$. And since we can do it with any function, we can solve the stated equation for all λ-terms. For example, the factorial function is usually defined recursively as follows:

$$\begin{aligned} \text{factorial}(0) &= 1 \\ \text{factorial}(n) &= n \times \text{factorial}(n-1) \end{aligned}$$

The equivalent λ-term would then be

$$\textbf{fact} = \lambda n.\textbf{if}\,(\textbf{zero}?\ n)\ 1\ (*\ n\ (\textbf{fact}(\textbf{pred}\ n)))$$

However, since fact is defined in terms of itself, we don't really know what it *really* is. So we can use the fixed-point combinator to deduce it. Notice that:

$$\textbf{fact} = (\lambda fn.\textbf{if}\,(\textbf{zero}?\ n)\ 1\ (*\ n\ (f\,(\textbf{pred}\ n))\textbf{fact}$$

Meaning that fact is a fixed-point of $(\lambda fn.\textbf{if}\,(\textbf{zero}?\ n)\ 1\ (*\ n\ (f\,(\textbf{pred}\ n))$, or in other words, $\textbf{fact} = \Theta(\lambda fn.\textbf{if}\,(\textbf{zero}?\ n)\ 1\ (*\ n\ (f\,(\textbf{pred}\ n))$

Exercise 8. *Implement the fibonacci numbers in the λ-calculus. They are defined as follows:* $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

Solution. As we have done before, we can write an equation for **fib** in terms of itself, then we can use the Turing combinator to solve for **fib**.

$$\mathbf{fib} = \lambda n.\mathbf{if}(\mathbf{zero?}\ n)0(\mathbf{if}(\mathbf{zero?}(\mathbf{pred}\ n))1$$
$$(+(\mathbf{fib}(\mathbf{pred}\ n))(\mathbf{fib}(\mathbf{pred}\ (\mathbf{pred}\ n)))))$$

And solving for **fib** gives us

$$\Theta(\lambda n.\mathbf{if}(\mathbf{zero?}\ n)0(\mathbf{if}(\mathbf{zero?}(\mathbf{pred}\ n))1$$
$$(+(f(\mathbf{pred}\ n))(f(\mathbf{pred}\ (\mathbf{pred}\ n))))))$$

Q.E.D.

Exercise 9. *Implement a test for primality in the λ-calculus. Including any functions not yet defined.*