Lambda Calculus and the Decision Problem

William Gunther

November 8, 2010

Abstract

In 1928, Alonzo Church formalized the λ-calculus, which was an attempt at providing a foundation for mathematics. At the heart of this system was the idea of function abstraction and application. In this talk, I will outline the early history and motivation for λ-calculus, especially in recursion theory. I will discuss the basic syntax and the primary rule: β-reduction. Then I will discuss how one would represent certain structures (numbers, pairs, boolean values) in this system. This will lead into some theorems about fixed points which will allow us have some fun and (if there is time) to supply a negative answer to Hilbert's Entscheidungsproblem: is there an effective way to give a yes or no answer to any mathematical statement.

1 History

In 1928 Alonzo Church began work on his system. There were two goals of this system: to investigate the notion of 'function' and to create a powerful logical system sufficient for the foundation of mathematics. His main logical system was later (1935) found to be inconsistent by his students Stephen Kleene and J.B.Rosser, but the 'pure' system was proven consistent in 1936 with the Church-Rosser Theorem (which more details about will be discussed later).

An application of λ-calculus is in the area of computability theory. There are three main notions of computability: Hebrand-Gödel-Kleene recursive functions, Turing Machines, and λ-definable functions. Church's Thesis (1935) states that λ-definable functions are exactly the functions that can be "effectively computed," in an intuitive way. It was proved by Kleene (1936) and Alan Turing (1937) that all three of these notions are equivalent. The thesis today is generally known as the Church-Turing Thesis, and Turing machines are widely studied and, ostensibly the most powerful of the three systems.

The first achievement of λ-calculus was to provide a negative answer to Hilbert's Decision Problem. In 1928, Hilbert asked if there is an effective way to answer whether some mathematical assertion was true. A proof or counterexample need not be given, merely a yes or no answer in some "effective" way. In 1936 Church proved that there was no effective way (using the model of λ-calculus) to determine whether a given λ-term has a β-normal form. The more widely known answer was given the the next year by Turing proved there was no effective way (using the model of Turing Machines) to determine whether a Turning Machine halts on a given input.

2 Basic Definitions and Syntax

2.1 Lambda Terms and β-Reduction

The following symbols will be used in our definition of λ-terms.

1) $($, $)$	precedence symbols
2) λ	abstraction operator
3) $x_0, x_1, \ldots$	variable symbols

Definition. Define Λ, the set of λ-terms, as the smallest set such that:-

- If x is a variable symbol, then $x \in \Lambda$.

- if $M \in \Lambda$ and x is a variable symbol then $(\lambda x \centerdot M) \in \Lambda$
- If $M, N \in \Lambda$ then $(MN) \in \Lambda$.

Example.

- $(\lambda x \centerdot (\lambda y \centerdot ((yx)z)))$ is a λ-term
- $(x\lambda z)w\lambda$ is not a λ-term

The theory on λ-terms, which we will denote as λ, is the smallest set which contains all terms which are considered equal under some rules. For instance, one rule would be reflexivity, ie. for all M, $M = M$. Another is transitivity and symmetry.

Apart from these rather trivial rules, we have applicative rules, ie. for all M, N, Z if $M = N$ then $MZ = NZ$ and $ZM = ZN$. This is saying if two functions are equal, then they should return the same value on any equal input (the converse is not necessarily true; this is call η-conversion, and will not be discussed). It is also saying if two data is equal, then it should be regarded the same when inputted in any function. These are all nice, but there is one principal rule called β-conversion which give our theory power.

Definition (β-conversion)**.** For all $M, N \in \Lambda$, we have $(\lambda x \centerdot M)N = M[x := N]$ where $M[x := N]$ is the term where instances of x in M are replaced by N.

Example.

- $(\lambda x \centerdot x)(\lambda y \centerdot y) = x[x := (\lambda y \centerdot y)] = (\lambda y \centerdot y)$
- $(\lambda x \centerdot (\lambda y \centerdot x + y))3 = (\lambda y \centerdot x + y)[x := 3] = (\lambda y \centerdot 3 + y)$

2.2 Conventions and Variables

Notice above, that looking at λ terms can be very hard to decipher. So we'll often omit outer parentheses whenever possible. We'll also use association to the left. That is, instead of writing

$$((MN)O)P$$

We'll just write

$$MNOP$$

We can also represent functions of two variables.

$$\lambda x \centerdot \lambda y \centerdot yx$$

Notice that the above expects for two inputs. On given it's first input, it really only returns another function, which expects another call. This idea is known as *Currying* and is used in λ-calculus to express functions with more than one variable. As a short hand, instead of writing the above, we will write

$$\lambda xy \centerdot yx$$

Definition (Free variable)**.** For M a λ-term, we say x occurs free in M if x occurs in M and it is not bound in the scope of a λ operator. More formally, $\mathrm{FV}(M)$ is defined by induction on the complexity of M.

If $M = x$ then $\mathrm{FV}(M) = \{x\}$.
If $M = \lambda x \centerdot N$ then $\mathrm{FV}(M) = \mathrm{FV}(N) \setminus \{x\}$.
If $M = AB$ then $\mathrm{FV}(M) = \mathrm{FV}(A) \cup \mathrm{FV}(B)$

Another notational "convention" (although it's a bit more than that) is α-conversion, where we associate terms that can be achieved from a change of bound variable. For instance

$$(\lambda x \centerdot x) = (\lambda y \centerdot y)$$

Note that we could associate these terms in our theory, but by convention we choose to just associate them on the syntactic level to make our lives a little simpler. The formal rule for α conversion is as follows:

Definition (α-conversion)**.** For all $M \in \Lambda$, $\lambda x \centerdot M$ is considered the same term as $\lambda y \centerdot M[x := y]$ assuming that $y \notin \mathrm{FV}(M)$.

What could go wrong if $y \in \mathrm{FV}(M)$?

Example. Consider $M = \lambda x \centerdot xy$, Note that $Mx = xy$. Then try to change the bound variable of M to y. Then $M = \lambda y \centerdot yy$. But then $Mx = xx$, and $xx \neq xy$.

2.3 Combinators and Examples

Let's do a few examples of β-reduction before moving on.

Example.

$$\begin{aligned}(\lambda x\mathbin{.}xy)(\lambda z\mathbin{.}z)w &= (\lambda x\mathbin{.}xy)(\lambda z\mathbin{.}z)w\\ &= (\lambda z\mathbin{.}z)yw\\ &= (\lambda z\mathbin{.}z)yw\\ &= yw\\ &= yw\end{aligned}$$

Definition. A λ-term M is a called a combinator if $\mathrm{FV}(M) = \emptyset$.

Here are some standard combinators:-

- $\mathbf{I} = \lambda x\mathbin{.}x$
- $\mathbf{K} = \lambda xy\mathbin{.}x$
- $\mathbf{S} = \lambda xyz\mathbin{.}xz(yz)$
- $\boldsymbol{\omega} = \lambda x\mathbin{.}xx$
- $\boldsymbol{\Omega} = \boldsymbol{\omega}\boldsymbol{\omega}$
- $\mathbf{Y} = \lambda f\mathbin{.}(\boldsymbol{\omega}(\lambda x\mathbin{.}f(xx)))$

Example.

$$\begin{aligned}\mathbf{S}(\lambda xy\mathbin{.}\mathbf{K}y)\mathbf{IK} &= \mathbf{S}(\lambda xy\mathbin{.}\mathbf{K}y)\mathbf{IK}\\ &= (\lambda xy\mathbin{.}\mathbf{K}y)\mathbf{K}(\mathbf{IK})\\ &= (\lambda xy\mathbin{.}\mathbf{K}y)\mathbf{K}(\mathbf{IK})\\ &= (\lambda y\mathbin{.}\mathbf{K}y)(\mathbf{IK})\\ &= (\lambda y\mathbin{.}\mathbf{K}y)(\mathbf{IK})\\ &= (\lambda y\mathbin{.}\mathbf{K}y)\mathbf{K}\\ &= (\lambda y\mathbin{.}\mathbf{K}y)\mathbf{K}\\ &= \mathbf{KK}\end{aligned}$$

Definition. We say that a term λ is in β normal form if it cannot be β-reduced. A term has a β normal form if it β reduces to a term that has a β normal form.

Example. $\mathbf{I}$ is in β-nf. $\boldsymbol{\Omega}$ does not have a β-nf. $\mathbf{K I \Omega}$ not in β-nf but it has one, namely $\mathbf{I}$

Theorem (Church). *There is no term G that decides for every $M \in \Lambda$ whether M has a β-nf.*

3 Data Structures

3.1 Boolean Structures

The first question is how do we represent boolean values, namely true or false.

Definition. Define the following combinators:

$$\begin{aligned}\mathbf{T} &= \lambda xy\mathbin{.}x\\ \mathbf{F} &= \lambda xy\mathbin{.}y\end{aligned}$$

Notice $\mathbf{T} = \mathbf{K}$ from above. Why are these definitions the useful ones? They allow us to easy do IF THEN ELSE statements.

Example. $A, B \in \Lambda$ then $MAB = A$ if $M = \mathbf{T}$ and $MAB = B$ if $M = \mathbf{F}$. Thus this is the expression IF M THEN A ELSE B (sort of; it's more like IF M THEN A; ELSE IF (NOT M) THEN B; ELSE who knows?)

3.2 Pairing

Now that we have boolean expression, it would be helpful to be able to pair two terms so that either can be recovered.

Definition. For $M, N \in \Lambda$ define

$$[M, N] = \lambda x\,.\,xMN$$

Note we can recover either M or N.

$$[M, N]\mathbf{T} = M$$
$$[M, N]\mathbf{F} = N$$

3.3 Numbers

Now we would like to be able to define a representation of the natural numbers $\mathbb{N}$.

Definition. Define $\ulcorner 0 \urcorner = \mathbf{I} = \lambda x\,.\,x$. Then assuming that we have define $\ulcorner n \urcorner$ for $n \in \mathbb{N}$ define

$$\ulcorner n+1 \urcorner = [\mathbf{F}, \ulcorner n \urcorner]$$

It is important to note that this is not the only way to represent numbers in λ-terms. In fact, this is not the way Church represented numbers. This was is a variation of representation of the natural numbers first used by Kleene. The benefit of this system is that the basic operations (successor, predecessor, and test for zero) are all very easy to define.

Definition. We define the following operations:

$$\begin{aligned} \mathbf{Succ} &= \lambda yx\,.\,x\mathbf{F}y \\ \mathbf{Pred} &= \lambda x\,.\,x\mathbf{F} \\ \mathbf{Zero?} &= \lambda x\,.\,x\mathbf{T} \end{aligned}$$

Let's just make sure these do what we expect:

$$\begin{aligned} \mathbf{Succ}\ulcorner n \urcorner &= \lambda x\,.\,x\mathbf{F}\ulcorner n \urcorner \\ &= [\mathbf{F}, \ulcorner n \urcorner] \\ &= \ulcorner n+1 \urcorner \\ \mathbf{Pred}\ulcorner n+1 \urcorner &= \ulcorner n+1 \urcorner\mathbf{F} \\ &= (\lambda x\,.\,x\mathbf{F}\ulcorner n \urcorner)\mathbf{F} \\ &= \ulcorner n \urcorner \\ \mathbf{Zero?}\ulcorner 0 \urcorner &= (\lambda x\,.\,x\mathbf{T})\mathbf{I} \\ &= \mathbf{T} \\ \mathbf{Zero?}\ulcorner n+1 \urcorner &= (\lambda x\,.\,x\mathbf{T})(\lambda y\,.\,y\mathbf{F}\ulcorner n \urcorner) \\ &= (\lambda y\,.\,y\mathbf{F}\ulcorner n \urcorner)\mathbf{T} \\ &= \mathbf{F} \end{aligned}$$

Note: We are not testing to see whether an arbitrary term is $\ulcorner 0 \urcorner$ or not, but whether a number is a term or not. There is a very big distinction. If you feed these functions garbage (ie. things not numbers) then you will get garbage (ie. things not boolean values or numbers).

4 Fixed Points

4.1 Paradoxical Combinator

Church's Thesis says that all recursive functions are representable and λ-terms. The question is: how do you do recursion? This is where fixed points come in. In my definition of the standard combinators above, there's one $\mathbf{Y}$ which is called the Paradoxical Combinator discovered by Curry. To remind you,

$$\mathbf{Y} = \lambda f\,.\,((\lambda x\,.\,f(xx))(\lambda x\,.\,f(xx)))$$

This is an example of a fixed point combinator, so named because of the next theorem

Theorem (Fixed Points Exist)**.** *For every $M \in \Lambda$ there exists $X \in \Lambda$ such that $MX = X$, that is X is a fixed point of M.*

Proof. We claim that $\mathbf{Y}M$ is a fixed point of M. To show this, we work backwards.

$$\begin{aligned}\mathbf{Y}M &= (\lambda x\boldsymbol{.}M(xx))(\lambda x\boldsymbol{.}M(xx))\\ &= M((\lambda x\boldsymbol{.}M(xx))(\lambda x\boldsymbol{.}M(xx)))\\ &= M(\mathbf{Y}M)\end{aligned}$$

□

At first glance, this doesn't seem like an overly powerful theorem. Next, however, we will discuss the power of fixed points is in self-reference and recursion.

4.2 Fixed Points: The Point

To show you the power of fixed point combinators, it's probably best to just show you an application. Let us build an adding function. that is, we want to build a term P such that

$$P\ulcorner n\urcorner\ulcorner m\urcorner = \ulcorner n+m\urcorner$$

Well, what we would like is to define

$$P\ulcorner n\urcorner\ulcorner m\urcorner = \begin{cases}\ulcorner m\urcorner & \text{if } n=0\\ P\ulcorner n-1\urcorner\ulcorner m+1\urcorner & \text{otherwise}\end{cases}$$

Let's rewrite this using our successor, predecessor, and test for zero combinators and "abstract ou" our bound variables.

$$P = \lambda xy\boldsymbol{.}\big(\mathbf{Zero}?x\big)\big(y\big)\big(P(\mathbf{Pred}x)(\mathbf{Succ}y)\big)$$

This is a perfectly adequate recursive way to define the sum of two terms. The problem: in our definition of P we reference P. So, we abstract out P:

$$P = \underbrace{\Big(\lambda pxy\boldsymbol{.}\big(\mathbf{Zero}?x\big)\big(y\big)\big(P(\mathbf{Pred}x)(\mathbf{Succ}y)\big)\Big)}_{Q} P$$

What do you see? P is a fixed point of Q. So, define P to be

$$P = \mathbf{Y}\Big(\lambda pxy\boldsymbol{.}\big(\mathbf{Zero}?x\big)\big(y\big)\big(P(\mathbf{Pred}x)(\mathbf{Succ}y)\big)\Big)$$

Let's see exactly what this does.

$$\begin{aligned}P\ulcorner n\urcorner\ulcorner m\urcorner &= YQ\ulcorner n\urcorner\ulcorner m\urcorner\\ &= Q(YQ)\ulcorner n\urcorner\ulcorner m\urcorner\\ &= Q(P)\ulcorner n\urcorner\ulcorner m\urcorner\\ &= (\mathbf{Zero}?\ulcorner n\urcorner)\ (\ulcorner m\urcorner)\ \ (P(\mathbf{Pred}\ulcorner n\urcorner)(\mathbf{Succ}\ulcorner m\urcorner))\end{aligned}$$

This is how we wanted P to behave. Now we need only show that P actually behaves like the plus operation. To prove this, we just do induction on the first input.

For the case where $n = 0$, we have $P\ulcorner 0\urcorner\ulcorner m\urcorner = \ulcorner m\urcorner$

For $n > 0$, $P\ulcorner n\urcorner\ulcorner m\urcorner = P\ulcorner n-1\urcorner\ulcorner m+1\urcorner = \ulcorner(n-1)+(m+1)\urcorner$ by induction, which is $\ulcorner n+m\urcorner$

This exemplifies why fixed points are important; they allow us to make recursive definitions and do self-reference.

4.3 Another Example

Earlier in the semester, Brian Kell told us what a what a Turing machine was. In an example, he constructed a Turing machine that will take an input a tape with left and right parentheses and output either True or False depending on whether the parentheses are properly formatted. We shall do the same with a λ-expression.

First, everything in λ-calculus is a λ-term. So we need to make definitions for what [and] in λ-terms. Let's define

$$\begin{aligned} [&= \ulcorner 1 \urcorner = \lambda x \mathbin{.} x(\lambda yz \mathbin{.} z)(\lambda w \mathbin{.} w) \\ ] &= \ulcorner 2 \urcorner = \lambda x \mathbin{.} x(\lambda yz \mathbin{.} z)(\lambda q \mathbin{.} q(\lambda rs \mathbin{.} s)(\lambda t \mathbin{.} t)) \\ X &= \ulcorner 0 \urcorner = \lambda x \mathbin{.} x \end{aligned}$$

And we will expect input to be a sequence of [and] ending with an X (X is our "end of string" character). Our term is going to evaluate the input term by term, so it is useful to have some term we can throw in case of an exception (ie. we find out the parentheses are not properly formatted in the middle of the expression). We'll call our expression B, and define it as

$$B = \mathbf{Y}\lambda bx \mathbin{.} (\mathbf{Zero}?x)Fb$$

Note this does what we expects; it ignores the rest of the input and then outputs $\mathbf{F}$.

$$B[\,[\,]\,X = B[\,]\,X = B]\,X = BX = \mathbf{F}$$

Now for the heart of the matter. Our term will actually take in two values. One will be a counter (that we control) that tells us the number of opening parentheses we have seen and haven't matched up. The other is the next term of the sequence. The idea is that if we ever get to a right parenthesis and do not have a left one to match it up, we will throw an exception. Otherwise, if we make it to the end and the number of left parentheses that have been unmatched is 0 then the expression was valid. Here's the term

$$((\lambda a \mathbin{.} (\lambda b \mathbin{.} a(bb))(\lambda b \mathbin{.} a(bb)))(\lambda hnp \mathbin{.} ((\lambda x \mathbin{.} x(\lambda st \mathbin{.} s))p)(((\lambda x \mathbin{.} x(\lambda st \mathbin{.} s))n)(\lambda st \mathbin{.} s)(\lambda st \mathbin{.} t))(((\lambda x \mathbin{.} x(\lambda st \mathbin{.} s))((\lambda q \mathbin{.} q(\lambda st \mathbin{.} t))p))$$
$$(h((\lambda qr \mathbin{.} r(\lambda st \mathbin{.} t)q)n))(((\lambda x \mathbin{.} x(\lambda st \mathbin{.} s))n)((\lambda a \mathbin{.} (\lambda b \mathbin{.} a(bb))(\lambda b \mathbin{.} a(bb)))\lambda bx \mathbin{.} ((\lambda x \mathbin{.} x(\lambda st \mathbin{.} s))x)(\lambda st \mathbin{.} t)b)(h((\lambda q \mathbin{.} q(\lambda st \mathbin{.} t))n))))))(\lambda w \mathbin{.} w)$$

Hm... That might not have been as clear as I thought it would be. Let me try to display it a little neater.

$$\left(\mathbf{Y}\lambda hnp \mathbin{.} \Big(\mathbf{Zero}?p\Big)\Big((\mathbf{Zero}?n)\mathbf{TF}\Big)\bigg(\Big(\mathbf{Zero}?(\mathbf{Pred}p)\Big)\Big(h(\mathbf{Succ}n)\Big)\Big((\mathbf{Zero}?n)B\big(h(\mathbf{Pred}n)\big)\Big)\bigg)\right)\ulcorner 0 \urcorner$$

Obvious, right? Probably not. Here it is with a little more explanation.

```
H = Yλhnp .
        (Zero?p) (                      is input X:
                (Zero?n) (                 is # of unmatched left parens 0?:
                        T                       return true
                ) (                        else:
                        F                       return false
                )
        ) (                              else:
                (Zero?(Pred p)) (          is input [ :
                        (h(Succ n))             call h on next input with n+1 unmatched [
                ) (                        else (it's a ]):
                        (Zero?n) (              is there no left parens to match it with:
                              B                       throw exception
                        ) (                     else
                              h(Pred n)               call h on next input with n-1 unmatched [
                        )
                )
        )
```

Then make our function $P = H\ulcorner 0 \urcorner$. One can check that this works.

4.4 Another Fixed Point Theorem

Theorem (Gödel Numbering)**.** *There exists an effect enumeration of λ-terms. That is, an assignment of terms M to $\mathbb{N}$. We call this number the Gödel numering of Λ. For $M \in \Lambda$ we write $\#M$ to denote the Gödel number of M. We write $\ulcorner \#M \urcorner$ to stand for the λ-term representing $\#M$.*

Here's another fixed point theorem that will help us answer the decision problem:

Theorem. *For every F there is an X such that $F\ulcorner \#X \urcorner = X$.*

Proof. All recursive functions are λ-definable by the Church-Turing Thesis. By the effectiveness of our numbering, there is a term $\mathbf{N}$ such that

$$\mathbf{N}\ulcorner \#M \urcorner = \ulcorner \# \ulcorner \#M \urcorner \urcorner$$

Furthermore, there is a term $\mathbf{A}$ such that

$$\mathbf{A}\ulcorner \#M \urcorner \ulcorner \#N \urcorner = \ulcorner \#(MN) \urcorner$$

We think of $\mathbf{N}$ as giving us the Gödel number of a term, and $\mathbf{A}$ as giving us the Gödel number of the application of two terms.

Take $W = \lambda n\mathbf{.}F(\mathbf{A}n(\mathbf{N}n))$. Then let $X = W\ulcorner \#W \urcorner$

$$\begin{aligned} X &= W\ulcorner \#W \urcorner \\ &= F(\mathbf{A}\ulcorner \#W \urcorner(\mathbf{N}\ulcorner \#W \urcorner)) \\ &= F(\mathbf{A}\ulcorner \#W \urcorner(\ulcorner \# \ulcorner \#W \urcorner \urcorner)) \\ &= F(\ulcorner \#(W\ulcorner \#W \urcorner) \urcorner) \\ &= F(\ulcorner \#X \urcorner) \end{aligned}$$

□

5 Answer to the Decision Problem

Alonzo Church proved that there is no term that will decide whether two terms have the same normal form. He reduced this problem to asking whether a given term has a normal form, and then showed this problem can't be answered using a λ-term. We will just show this part of the proof, as it still gives a negative answer to the Decision Problem, which is all we're interested in.

Another thing to note is that this is not the classical proof.

Theorem. *There is no lambda term M such that*

$$M\mathbf{n} = \begin{cases} \ulcorner 0 \urcorner & \textit{if term with Gödel number } n \textit{ has a } \beta\textit{-nf} \\ \ulcorner 1 \urcorner & \textit{otherwise} \end{cases}$$

Note: What I'm using to separate the two isn't so important. I chose to use 0 and 1 because that is done in the classical proof, but one can see quick modification will work for **F** and **T** which is probably the more natural candidate.

Proof. Suppose there was such an M. Then define

$$G = \lambda n\mathbf{.Zero}?(Mn)\ \mathbf{\Omega}\ \mathbf{I}$$

By the previous fixed point theorem, there is a term X such that

$$G\ulcorner \#X \urcorner = X$$

Then,

$$\begin{aligned} X \text{ has a } \beta\text{-nf} &\implies M\ulcorner \#X \urcorner = 0 \\ &\implies G\ulcorner \#X \urcorner = \mathbf{\Omega} \\ &\implies X = \mathbf{\Omega} \\ &\implies X \text{ has no } \beta\text{-nf} \end{aligned}$$

Similarly:

$$\begin{aligned} X \text{ has no } \beta\text{-nf} &\implies M\ulcorner \# X \urcorner = 1 \\ &\implies G\ulcorner \# X \urcorner = \mathbf{I} \\ &\implies X = I \\ &\implies X \text{ has a } \beta\text{-nf} \end{aligned}$$

□

An important thing to note is this shows that one cannot separate (using a term) a subset of Λ that is closed under equality (in this case, it was all terms having a β-nf). Why does it need to be closed under equality? Because the fixed point doesn't give us exactly what we want, it gives us something β equal to it.

There's really no modifications to the proof to get this result, except changing $\mathbf{I}$ to a representative from that class, and $\mathbf{\Omega}$ for it's compliment.