# What is the Lambda Calculus? - My Notes

In 1928, David Hilbert (1862 - 1943) posed the Entscheidungsproblem. This problem asks for an algorithm which decides (namely, answers yes or no to) whether or not a given statement of first-order logic is provable from some set of given axioms.

In 1936, Alan Turing (1912 - 1954) and Alonzo Church (1903 - 1995) independently published papers showing that the Entscheidungsproblem is unsolvable (that there is no such algorithm). Turing's method used his work on so-called Turing machines. He reduced a different problem, the Halting problem, to the Entscheidungsproblem, and then showed that the Halting problem was unsolvable, thus showing the Entscheidungsproblem was also unsolvable. Church used the $\lambda$-calculus and showed that there was no algorithm that decides whether two $\lambda$-expressions are equivalent, another problem that reduces to the Entscheidungproblem.

Both of these proofs rely on one key unproven assumption, often called the Church-Turing thesis: that Hilbert's use of the word "Algorithm" was a Turing machine (or equivalently a $\lambda$-expression). More precisely, Church and Turing both posited that any procedure which could be carried out by a human using pencil and paper (ignoring resource limitations) could be carried out by a turing machine or a $\lambda$-expression.

While Turing's machines and Church's $\lambda$-calculus are equivalent universal models for computation (as was proved by Turing in the cited paper, we will see what exactly it means for the two models to be equivalent later in the talk when we discuss computable functions), Turing machines became the theoretical foundation for modern computers due to the fact that the theoretical Turing machine is something that is very straightforward to build in real life, and so Alan Turing is a much more popular dude in the literature than Alonzo Church (for some numbers to back this up, Turing's wikipedia page has 240 notes, while Church's only has 10).

In this talk, we will be combining the methods of Church and Turing, first looking at the basic structure of the $\lambda$-calculus (Church's idea) and then giving a concise proof of the undecidability of the Halting problem (Turing's idea) with the $\lambda$-calculus as the underlying machinery rather than Turing machines.

I'd also quickly like to add that I'll be using the word "function" to talk about $\lambda$-expressions. This is convention and means something more like a function in a programming language than a mathematical function. In fact, another way to think of the $\lambda$-calculus is as a simple programming language in and of itself.

## 1: $\lambda$-expressions

**Definition:** A $\lambda$-expression is a string of one of the following forms:

| | |
|---|---|
| $x$ | some variable |
| $\lambda x.M$ | where $M$ is a $\lambda$-expression (function abstraction) |
| $AB$ | where $A$ and $B$ are $\lambda$-expressions (function application) |

**Notes:** function application is left-associative: $ABC \equiv (AB)C$.

We can add parentheses for clarity or to provoke right-associative behavior: $ABC \not\equiv A(BC)$.

**Examples:** $\lambda x.x$ $\lambda x.y$ $\lambda x.\lambda y.xy$ $(\lambda x.(\lambda y.xy))(\lambda x.y)(\lambda x.x)$ $\lambda x.(\lambda y.(\lambda z.xzyz))$

**Notation:** Often $\lambda x(\lambda y.M)$ is shortened to $\lambda xy.M$. However, we should keep in mind that there are actually two nested $\lambda$-expressions in $\lambda xy.M$.

## 2: $\lambda$-calculus operations and $\beta$-normal form

We use the notation $[y/x]$ to denote substitution of all instances of $x$ in a string for $y$. For example, $[w/z]z \equiv w$, $[w/z]xzy \equiv xwy$, and $[a/b](\lambda x.b) \equiv \lambda x.a$. A $\lambda$-expression can be changed (reduced) by one of the two operations:

| | |
|---|---|
| $\lambda x.M \to \lambda y.[y/x]M$ | $\alpha$-conversion: simply to avoid name collisions |
| $(\lambda x.M)A \to [A/x]M$ | $\beta$-reduction: computation of a function application |

**Examples:** $\lambda x.(\lambda y.xy) \to \lambda w.(\lambda y.wy)$ $\lambda x.(\lambda yz.zyx) \to \lambda x.(\lambda yw.wyx)$ $\lambda x.((\lambda y.y)x) \to \lambda x.x$ $(\lambda fx.f(f(fx)))gy \to (\lambda x.g(g(gx)))y \to g(g(gy))$ $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$ $(\lambda x.y)z \to y$

**Definition:** A $\lambda$-expression is in **$\beta$-normal form** if no $\beta$-reduction operation can be performed. For example, $\lambda x.x$ and $y$ are in $\beta$-normal form while $(\lambda x.x)y$ is not because $(\lambda x.x)y \to [y/x]x \equiv y$ via $\beta$-reduction.

**Definition:** A $\lambda$-expression **halts** if, after a finite number of operations, it reaches a $\beta$-normal form. For example, $(\lambda x.x)(\lambda x.x)$ halts while $(\lambda x.xx)(\lambda x.xx)$ doesn't.

**Notes:** When using $\alpha$-conversion, there must not be any $y$ in $M$ (i.e. we cannot create a name collision).

It is often reasonable to think of a $\lambda$-expression in $\beta$-normal form as a function (algorithm) acting on a $\lambda$-expression.

We apply the $\beta$-reduction rule "outside-in" rather than "inside-out" meaning we apply the leftmost $\lambda$ first.

## 3: Boolean algebra in the $\lambda$-calculus

We can define the Boolean values "True" and "False" in the following way:

$$\mathbf{T} \coloneqq \lambda xy.x \qquad\qquad \mathbf{F} \coloneqq \lambda xy.y$$

And we can define logical operations as follows:

| | |
|---|---|
| $\vee \coloneqq \lambda xy.x\mathbf{T}y$ | Logical "or" |
| $\wedge \coloneqq \lambda xy.xy\mathbf{F}$ | Logical "and" |
| $\neg \coloneqq \lambda x.x\mathbf{FT}$ | Logical "not" |

For example: $\neg\mathbf{T} \to \mathbf{TFF} \to \mathbf{F}$ $\wedge\mathbf{TF} \to \mathbf{TFF} \to \mathbf{F}$ $\vee\mathbf{TF} \to \mathbf{TTF} \to \mathbf{T}$

## 4: Arithmetic in the $\lambda$-calculus (Church numerals)

We can define the natural numbers in the following way:

$\mathbf{0} \coloneqq \lambda fx.x$ $\mathbf{1} \coloneqq \lambda fx.fx$ $\mathbf{2} \coloneqq \lambda fx.f(fx)$ $\mathbf{3} \coloneqq \lambda fx.f(f(fx))$ ...

Notice that $\mathbf{n}f$ $\beta$-reduces to a function which applies $f$ $n$ times to its argument.

We can define some mathematical operations on the natural numbers as follows:

| | |
|---|---|
| $S \coloneqq \lambda n.(\lambda fx.nf(fx))$ | Successor function |
| $+ \coloneqq \lambda nm.nSm$ | Addition |
| $\times \coloneqq \lambda nm.n(+m)\mathbf{0}$ | Multiplication |

For example: $S\mathbf{2} \to \lambda fx.\mathbf{2}f(fx) \to \lambda fx.f(f(fx)) \equiv \mathbf{3}$ $+(\mathbf{3})(\mathbf{2}) \to \mathbf{3}S\mathbf{2} \to S(S(S\mathbf{2})) \to S(S\mathbf{3}) \to S\mathbf{4} \to \mathbf{5}$ $\times(\mathbf{3})(\mathbf{2}) \to \mathbf{3}(+\mathbf{2})\mathbf{0} \to (+\mathbf{2})((+\mathbf{2})((+\mathbf{2})\mathbf{0})) \to (+\mathbf{2})((+\mathbf{2})\mathbf{2}) \to (+\mathbf{2})\mathbf{4} \to \mathbf{6}$

It is often useful to have an operator which checks if a given number is zero, returning a Boolean value:

$$Z \coloneqq \lambda n.n\mathbf{F}\neg\mathbf{F}$$

For example: $Z\mathbf{0} \to \mathbf{0F}\neg\mathbf{F} \to \neg\mathbf{F} \to \mathbf{T}$ $Z\mathbf{1} \to \mathbf{1F}\neg\mathbf{F} \to \mathbf{F}\neg\mathbf{F} \to \mathbf{F}$ $Z\mathbf{3} \to \mathbf{3F}\neg\mathbf{F} \to \mathbf{F}(\mathbf{F}(\mathbf{F}\neg))\mathbf{F} \to \mathbf{F}$

## 5: Computable functions

**Definitions:**

A function $f : \mathbb{N}^k \to \mathbb{N}$ is $\lambda$-*computable* if there is a $\lambda$-expression $F$ so that $F\mathbf{n}_1 \dots \mathbf{n}_k \to^* \mathbf{m}$ iff $f(n_1, \dots, n_k) = m$.

A set $A \subseteq \mathbb{N}^k$ is $\lambda$-*recognizable* if there is a $\lambda$-expression $L$ so that $L\mathbf{n}_1 \dots \mathbf{n}_k \to^* \mathbf{T}$ iff $(n_1, \dots, n_k) \in A$.

If there is an $L$ as above so that $L\mathbf{n}_1 \dots \mathbf{n}_k$ halts for every $(n_1, \dots, n_k) \in \mathbb{N}^k$, then $A$ is $\lambda$-*decidable*.

## 6: Encodings of $\lambda$-expressions

We will need to slightly restrict our definition of a $\lambda$-expression by only allowing variable names to be $x$ or $x$ followed by any number of $'$s: $x$, $x'$, $x''$, $x'''$, etc. We will also require that $\lambda$-expressions be written out "in full" (i.e. in terms of only the six basic symbols required, which are listed in the table below, and with only one variable per $\lambda$). Now we can encode any $\lambda$-expression by a natural number by translating symbols to digits as follows:

| Symbol | $\lambda$ | . | $x$ | $'$ | ( | ) |
|---|---|---|---|---|---|---|
| Digit | 1 | 2 | 3 | 4 | 5 | 6 |

So, for example, our + algorithm defined earlier, when written out in our more restrictive notation, looks like this:

$$\lambda x.\lambda x'.x(\lambda x''.\lambda x'''.\lambda x''''.x''x'''(x'''x''''))x'$$

Which means its encoding, denoted $< + >$, is 13213423513442134442134444234434445344434446634.

## 7: The Halting problem

Given any $\lambda$-expression $M$ and church numeral $\mathbf{w}$, can we decide if $M\mathbf{w}$ halts? More precisely, is the set

$$\{(< M >, w) \mid M\mathbf{w} \text{ halts}\} \subseteq \mathbb{N}^2$$

$\lambda$-decidable? As it turns out, the answer is no. If some $\lambda$-expression $H$ $\lambda$-decides this set then we can define a new $\lambda$-expression $G \coloneqq \lambda m.Hmm((\lambda x.xx)(\lambda x.xx))y$. Now, does $G < G >$ halt? Well, if $G < G >$ halts then we have $H < G >< G > \twoheadrightarrow^* \mathbf{T}$, so $G < G > \twoheadrightarrow H < G >< G > ((\lambda x.xx)(\lambda x.xx))y \twoheadrightarrow^* \mathbf{T}((\lambda x.xx)(\lambda x.xx))y \to (\lambda x.xx)(\lambda x.xx)$, which of course does not halt. On the other hand, if $G < G >$ does not halt then $H < G >< G > \twoheadrightarrow^* \mathbf{F}$ since $H$ $\lambda$-decides the halting set. this means $G < G > \twoheadrightarrow H < G >< G > ((\lambda x.xx)(\lambda x.xx))y \twoheadrightarrow^* \mathbf{F}((\lambda x.xx)(\lambda x.xx))y \to y$ which is in $\beta$-normal form, showing that $G < G >$ halted. This contradiction shows that $G$ (and thus $H$) cannot exist.

# Extras:

| | |
|---|---|
| $Y \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ | (fixed-point combinator: recursion) |
| $P \coloneqq \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$ | (Predecessor) |
| $- \coloneqq \lambda nm.nPm$ | (Subtraction) |
| $\leq \coloneqq \lambda nm.Z(-nm)$ | (Less than or equal to comparison) |
| $\div \coloneqq \lambda nm.Y(\lambda fx.(\leq mx)(+\mathbf{1}(f(-xm)))\mathbf{0})n$ | (Division) |
| $! \coloneqq \lambda n.Y(\lambda fx.(Zn)\mathbf{1}(\times n(f(Px))))n$ | (Factorial) |
| $B \coloneqq \lambda n.Y(\lambda fx.(Zx)\mathbf{0}((Z(Px))\mathbf{1}(+(f(Px))(f(P(Px))))))n$ | (n-th fibonacci number) |