LAMBDA CALCULUS

1. Background

λ-calculus is a formal system with a variety of applications in mathematics, logic, and computer science. It examines the concept of functions as processes, rather than the usual set-theoretic notion of functions in extension, and it provides an alternative model for computable functions. Functional programming languages like Haskell and ML owe a great deal to lambda calculus for their existence. λ-calculus is itself, in some sense, a very simple programming language, although it would be impractical to implement for most purposes. However, as a programming language, what it lacks in practicality, it makes up for in simplicity, and since it is equivalent to most computer languages, it provides a great way to study their properties.

λ-calculus was invented in the 1930's by Alanzo Church and Stephen Kleene. Originally, in addition to being used to study function theory, λ-calculus was intended to provide a foundation for mathematics, but this system was shown to be inconsistent. Despite this, Church reduced λ-calculus to its functional aspect, which is the system we use today.

2. Description of the Language

We will assume familiarity with the basics of λ-calculus, but here we will include a brief description of the language and transformation rules.

The language of λ-calculus, Λ, can be defined recursively as follows:

λ-terms. *$T \in \Lambda$ iff one of the following holds:*

1. *T is a member of a countable set of variables $\{a, b, c, \ldots, x_1, x_2, \ldots\}$.*
2. *T is of the form (MN) where M and N are in Λ.*
3. *T is of the form $(\lambda X.Y)$ where X is a variable and Y is in Λ.*

Λ is the smallest language with this property.

An element of the form (MN) is called an application, and an element of the form $\lambda X.Y$ is called an abstraction. In order to reduce parentheses we will assume application is left-associative. Also, instead of saying "T is of the form M" we will write "T $\equiv$ M".

Example. *The following are all examples of λ -terms:*

1. $\lambda p.p\lambda x.y$
2. $x\lambda m.\lambda n.nm$
3. $x\lambda mn.nm$ *-whenever we have consecutive abstractions as in the previous example, we can concatenate them like this.*
4. $(\lambda p.p(\lambda xy.x))(\lambda m.mab)$
5. $(\lambda x.xx)(\lambda y.y)$

Example (5) is an instancs of a redex, which is an application with an abstraction on the left. We interpret abstractions $\lambda x.P$ as descriptions of what they do to arguments. The term P describes explicitly what the abstraction will return, using the variable x to represent the term applied to the abstraction. So the abstraction in example (5) takes a term x and outputs 2 copies of it. Hence it takes $(\lambda y.y)$ and returns $(\lambda y.y)(\lambda y.y)$, and we say "$(\lambda x.xx)(\lambda y.y)$ contracts to $(\lambda y.y)(\lambda y.y)$," or more succinctly, "$(\lambda x.xx)(\lambda y.y) \rhd_1 (\lambda y.y)(\lambda y.y)$." If P takes 0 or more contractions to get to the term Q, then we say "P reduces to Q," or "$P \rhd Q$."

We say two terms P and Q are β-equivalent, or "$P =_\beta Q$" if they both reduce to a common term. For our purposes, we will drop the β subscript.

3. λ-definability, Church Numerals and Recursion

In order to show that λ-calculus is actually useful, we will show that we can encode natural numbers and many of the basic arithmetic operations into the language. There are many possible ways to do this, but we will use the conventional system put forth by Church:

Church Numerals. *For every natural number n, we associate with it a λ-term $\overline{n}$ of the form*

$$\overline{n} \equiv \lambda xy.x^n y,$$

where x^n denotes x repeated n times.

So we see that each number n, as a Church numeral, takes in two arguments and applies the first argument n times to the second argument. This will make it easy to define basic arithmetic operations, since many of them, like multiplication and exponentiation, are simply repeated applications of a particular function.

Definition. *A function $\phi : \mathbb{N}^k \to \mathbb{N}$ is $\underline{\lambda - definable}$ if there is a λ-term F such that for every n_1, n_2,..., $n_k \in \mathbb{N}$, we have that*

$$F\overline{n_1} \ldots \overline{n_k} = \overline{\phi(n_1, \ldots, n_k)}.$$

That is, F is the same as ϕ, except that it operates on the Church numerals.

All of the basic operations on natural numbers are λ-definable. We can build them up quite easily due to the form of the Church numerals:

$$\begin{aligned} [\text{inc}] &\equiv \lambda n.\lambda xy.x(nxy) \\ [\text{add}] &\equiv \lambda mn.m[\text{inc}]\overline{0} \\ [\text{mul}] &\equiv \lambda mn.m([\text{add}]n)\overline{0} \\ [\text{exp}] &\equiv \lambda mn.n([\text{mul}]m)\overline{1} \end{aligned}$$

In [add], [mul], and [exp], we see that the abstractions take a given Church numeral, replace the repeated part by a given function, and replace the end part by a starting value. For instance, in [mul], the repeated part of one of the inputs m is replaced by the function ([add]n) that adds n to a given input, and the end part is replaced by the starting value 0. So we see that [mul] implements multiplication as repeated addition.

In order to define more complex functions, it is useful to create a pairing operator and corresponding selectors. Before, in [add], [mul], and [exp], we could only replace the repeating part of a number by unary operators and replace the end part by a single value. With pairing operators and selectors, however, we could effectively pass multiple values through each repeated function. So we define the [cons], [car], and [cdr] operators (paying tribute to the LISP naming conventions):

$$\begin{aligned} [cons] &\equiv \lambda abm.mab \\ [car] &\equiv \lambda p.p(\lambda xy.x) \\ [cdr] &\equiv \lambda p.p(\lambda xy.y) \end{aligned}$$

We see that the pairing operator [cons] is specifically designed with selectors [car] and [cdr] in mind:

$$\begin{aligned} [car]([cons]AB) &\equiv (\lambda p.p(\lambda xy.x))((\lambda abm.mab)AB) \\ &\rhd (\lambda p.p(\lambda xy.x))(\lambda m.mAB) \\ &\rhd (\lambda m.mAB)(\lambda xy.x) \\ &\rhd (\lambda xy.x)AB \\ &\rhd A \end{aligned}$$

It is clear that [cdr] works the same way, except at the last step it returns B instead of A.

So now we can make some more complicated functions, like [fac], which returns the factorial of a given number. Typically, in a program, the factorial function is defined iteratively, with a counter variable that multiplies an accumulated value on each step. Now with our pairing operators, we can create [fac] in an analoguous way, replacing the end part of the input number n by the pair (1,1), and replacing the repeating part of n by the function that takes a pair (a,b) and returns the pair (a+1,a*b). Applied n times, the function will return (n+1,n!), so we need to take the [cdr] at the end. Thus [fac] can be defined as follows:

$$\begin{aligned} \text{[fac-iter]} &\equiv (\lambda p.\text{[cons]}(\text{[inc]}(\text{[car]}p))(\text{[mul]}(\text{[car]}p)(\text{[cdr]}p))) \\ \text{[fac]} &\equiv \lambda n.\text{[cdr]}(n\text{[fac-iter]}(\text{[cons]}\overline{1}\ \overline{1})) \end{aligned}$$

Continuing on with our development of a useful set of operators, a logical next step would be to create some form of conditional operator, something essential to just about any programming language. Suppose we would like to implement an if-then-else construct that performs a test, and depending on the outcome, returns one of two specified outputs. That is, we want to create an operator [if-then-else] that takes 3 inputs A, B, and C, where A holds a boolean value, and B and C are the two possible outputs. Boolean values can be represented many ways in lambda calculus, but we will represent it with our given Church numerals, where $\overline{0}$ represents false and $\overline{1}$ represents true. So to begin, our function will take B and C, and pair them together with [cons]. Then we will want to find some way to apply [car] to this pair if A is $\overline{1}$ and [cdr] if A is $\overline{0}$. To do this, we can use the fact that $\overline{0}$ ignores its first input, whereas $\overline{1}$ does not:

$$[\text{if-then-else}] \equiv \lambda abc.a(\lambda x.[car])[cdr]([cons]bc)$$

We can see with a simple reduction that this function works:

$$\begin{aligned}
[\text{if-then-else}]0AB &\equiv 0(\lambda x.[car])[cdr]([cons]AB) \\
&\rhd [cdr]([cons]AB) \\
&\rhd B
\end{aligned}$$

$$\begin{aligned}
[\text{if-then-else}]1AB &\equiv 1(\lambda x.[car])[cdr]([cons]AB) \\
&\rhd (\lambda x.[car])[cdr]([cons]AB) \\
&\rhd [car]([cons]AB) \\
&\rhd A
\end{aligned}$$

So far we have been avoiding a rather important operator, the predecessor operator, and now we have enough machinery to define it. Since we do not have negative numbers, we will have the predecessor operator take 0 to 0.

$$\begin{aligned}
[\text{pred-iter}] &\equiv (\lambda p.[cons]\overline{1}([\text{if-then-else}]([car]p)([inc]([cdr]p))\overline{0})) \\
[\text{pred}] &\equiv \lambda n.[cdr](n[pred-iter]([cons]\overline{0}\ \overline{0}))
\end{aligned}$$

We see that the [pred] operator is similar to [fac] in that it replaces the repeating part of the input number n by a function that operates on and returns pairs. In this case, we replace the end part of n by the pair (0,0) and apply to it n times the function that takes a pair (a,b) and returns (1,0) if a=0 and (1,b+1) if a=1.

With [pred] we can now define [monus] which subtracts n from m if n≤m and returns 0 otherwise:

$$[\text{monus}] \equiv \lambda mn.n[pred]m$$

Now that we have shown many of the basic arithmetic operations to be λ-definable, we may ask what other natural number functions are definable. Clearly, not all can be defined, since the set of functions from $\mathbb{N}^k$ to $\mathbb{N}$ is uncountable and the set of λ-terms is countable. It can be shown quite easily that all primitive recursive functions are definable, by introducing the recursion operator. But this is far from a desirable set of functions, since there are many computable functions that are not primitive recursive, like Ackerman's function, for instance. Although we will not prove it here, Kleene had shown that every total recursive function is λ-definable. We will use this fact in the next section to derive a general undecidability theorem.

4. The Undecidability Theorem

The first undecidability proof in mathematics was given by Alonzo Church in 1936, using λ-calculus. The following proof is a more general form, but proves the same result, that there is no algorithm to decide whether or not two λ-terms are equivalent.

To begin, we will make a few major assumptions. First, we assume that there is an algorithm that defines an injective map $\#:\Lambda \to \mathbb{N}$ that associates with every λ-term a natural number. This is called a Godel numbering of the terms, which

allows us to perform computations within λ-calculus on other λ-terms. We also assume that these terms are numbered so that there are total recursive functions τ and ν such that

$$\tau(\#A, \#B) = \#(AB) \text{ and } \nu(n) = \#\overline{n}$$

Before we state the theorem, we begin with a few definitions.

Definition. *Let A and B be disjoint sets of natural numbers. They are called recursively separable iff there is a total recursive function ϕ such that*

$$\begin{aligned} n \in A &\implies \phi(n) = 1 \\ n \in B &\implies \phi(n) = 0 \end{aligned}$$

We say two sets of terms A and B are recursively separable if the sets of Godel numbers associated with A and B are recursively separable.

Definition. *A set of terms A is closed under equality if*

$$P \in A \text{ and } P = Q \implies Q \in A$$

One more thing: let $[\![X]\!]$ denote $\overline{\#X}$.

Scott-Curry Undecidability Theorem. *No pair of non-empty sets of terms closed under equality is recursively seperable.*

Proof. Suppose not. Then there is a total recursive function ϕ , and a pair of non-empty sets of terms $\mathcal{A}$ and $\mathcal{B}$ such that ϕ separates $\mathcal{A}$ and $\mathcal{B}$. Since every recursive function is definable in λ-calculus, let F be the λ-term defining ϕ. That is,

$$X \in \mathcal{A} \implies F[\![X]\!] = \overline{1}, \tag{1}$$

$$X \in \mathcal{B} \implies F[\![X]\!] = \overline{0}. \tag{2}$$

Now let T and V be the terms which define the functions τ and ν. Then we have that

$$T[\![X]\!][\![Y]\!] = [\![XY]\!], \tag{3}$$

$$V\overline{n} = [\![\overline{n}]\!] \tag{4}$$

Now in order to find a contradiction, we will construct, using any two terms A $\in \mathcal{A}$ and B $\in \mathcal{B}$ we want, another term J with the property that

$$F[\![J]\!] = \overline{1} \implies J = B, \tag{5}$$

$$F[\![J]\!] = \overline{0} \implies J = A. \tag{6}$$

Clearly, both of these statements cannot be true. If $F[\![J]\!] = \overline{1}$, then J must be in $\mathcal{B}$, but this implies by the definition of F that $F[\![J]\!] = \overline{0}$. Likewise, if $F[\![J]\!] = \overline{0}$, we conclude by the same reasoning that $F[\![J]\!] = \overline{1}$, so we reach a contradiction. Now to construct J, we will want it to have the following property:

$$J = [\text{if-then-else}](F[\![J]\!])BA \tag{7}$$

If we construct such a J, then we see that if $F[\![J]\!] = 1$, then $J = [\text{if-then-else}]1BA = B$, and if $F[\![F]\!] = 0$, then $J = [\text{if-then-else}]0BA = A$. So let

$$J \equiv H[\![H]\!] \text{ where } H \equiv \lambda y.[\text{if-then-else}](F(Ty(Vy))BA$$

Now we perform a simple β reduction to see that J has the desired property (7):

$$\begin{aligned} J &= H[\![H]\!] \\ &= (\lambda y.[\text{if-then-else}](F(Ty(Vy)))BA)[\![H]\!] \\ &= [\text{if-then-else}](F(T[\![H]\!](V[\![H]\!])))BA \\ &= [\text{if-then-else}](F(T[\![H]\!][\![[\![H]\!]]\!]))BA \\ &= [\text{if-then-else}](F([\![H[\![H]\!]]\!]))BA \\ &= [\text{if-then-else}](F([\![J]\!]))BA \end{aligned}$$

□

References

[1] Barendregt, H. P., *The Lambda Calculus*, North-Holland Publishing Company, New York, NY, 1981.

[2] Hindley, J. R., Seldin, J. P., *Introduction to Combinators and λ-Calculus*, Cambridge University Press, New York, NY, 1986.