Introduction to lambda calculus
Part 1

Antti-Juhani Kaijanaho

19 January 2017*

The lambda calculus was invented by Alonzo Church in the early 1930s as an attempted fundamental theory for mathematics (Church 1932, 1933). It failed in that mission, but accidentally became something much more interesting. Church (1936) and Alan Turing (1937) independently proved that the *Entscheidungsproblem* – that is, the problem of determining whether a particular formula of first-order predicate logic is a theorem – is undecidable. Church used the lambda calculus, while Turing used his new theory of computing machines (now called Turing machines). Turing also proved that the lambda calculus can do everything that Turing machines can, and vice versa.

The lambda calculus became – no doubt at least partially due to the influence of Strachey (2000) – a sort of basic mathematical theory of programming languages, and also has been used as the basis for many functional programming languages (such as Scheme, ML, and Haskell).

Good expositions of modern lambda calculus were written by Church (1985) and Curry and Feys (1968). The definitive reference of the mathematical theory of the lambda calculus was written by Barendregt (1984).

1 Untyped lambda calculus

1.1 Syntax

The abstract syntax of pure untyped lambda calculus is very simple:

*Minor changes made afterward. Last changed 2017-01-20 11:52:21+02:00.

$$
\begin{array}{rcl}
x, y, z & \in & \textbf{Var} \\
t, u & \in & \textbf{Term} \\
t, u & ::= & x \\
 & | & t\,u \\
 & | & \lambda x \cdot t
\end{array}
$$

Thus, there are only three operations: using a variable, calling a one-parameter function, and constructing a (nameless) one-parameter function from a term and a variable to name the function's parameter. Traditionally, in lambda calculus, a function call is called an *application* and constructing a function is called *abstraction*. We can extend it by adding, for example, literal constants and arithmetical operations, in which case the language is no longer pure.

In this document, I will be using the following concrete phrase-structure syntax for the pure lambda calculus:

```
<term> ::= <term1>
         | λ <variables> . <term>
<term1> ::= <term2>
          | <term1> <term2>
<term2> ::= <variable>
          | ( <term> )
<variables> ::= <variable> | <variables> <variable>
```

As to the lexical syntax, in this document variables are single letters possibly with subscripts or primes added (so x, x_1, and x' are distinct variables); all other lexemes are λ, the period (which I usually write on the centerline, not at the bottom of the line), and the parentheses.

In an implementation it makes more sense to allow multi-character variable names as usual, since subscripts are not easy to write in a character string. Similarly, an implementation will probably replace λ with a reserved word (such as `fun` or `lambda`).

This gramar specifies that abstraction has a lower precedence than application; this is usual. Thus $\lambda x \cdot fx$ means the same as $\lambda x \cdot (fx)$. It is also standard that parentheses are not required around the argument in function calls. Thus we write fx, not $f(x)$ (though the latter is also allowed by the syntax). If arithmetic is added, abstraction should have a lower precedence and application higher precedence than the usual arithmetic operations (I will leave modifying the grammar as an exercise to the reader).

Notice that this concrete syntax allows specifying more than one parameter variable in a single abstraction. This is simple syntactic sugar: $\lambda xyz \cdot t$

In S-expression syntax (following Lisp), every term must be written inside parentheses, and even the list of parameter names is put inside parentheses; the period is omitted. There are two exceptions to this rule: we do not put parentheses around single variables, and we omit left-associative parentheses from nested applications; thus, we write (fxy) instead of $(((f)(x))(y))$, and we write $\lambda xyz \cdot t$ as $(\lambda(xyz)t)$. Further, in an S-expression syntax, all operators are written as functions that precede their arguments. For example, we would write $\lambda xy \cdot (x+y)/2$ as $(\lambda(xy)(/(+xy)2))$.

I will leave defining the exact syntax as an exercise for the reader.

Aside 1: S-expression syntax for lambda calculus

means the same as $\lambda x \cdot \lambda y \cdot \lambda z \cdot t$. Since abstraction, according to the concrete syntax, associates to the right, this is the same as $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot t))$. A function computing the average of two numbers is written as $\lambda xy \cdot (x+y)/2$.

Other concrete syntaxes are possible. One could, for example, follow Haskell and use $\backslash$ instead of λ and $\to$ instead of $\cdot$, or follow Lisp and use S-expressions (see Aside 1). Many theoreticians, in contrast, disallow giving multiple parameters in one abstraction and omit the period, and make abstraction to have a higher precedence than appliation (so they would write $(\lambda x \cdot x)a$ as λxxa and $\lambda x \cdot fx$ as $\lambda x(fx)$).

1.2 Denotational semantics

Suppose that there is some universe of values that we are interested in; let us call it $\mathbf{D}$.[1] Let us suppose that every value in $\mathbf{D}$ may be interpreted as a function $\mathbf{D} \to \mathbf{D}$. Then we may define the meaning of a pure untyped lambda calculus term as a function $(\mathbf{Var} \to \mathbf{D}) \to \mathbf{D}$; that is, a pure term stands for, or *denotes*, a value so long as all the variables occurring free in it stand for some values. Formally, we define an evaluation function as follows:

$$\mathbf{E} : \mathbf{Term} \to (\mathbf{Var} \to \mathbf{D}) \to \mathbf{D}$$

$$\mathbf{E}\,[\![x]\!]\,\sigma = \sigma\,([\![x]\!]) \tag{1}$$

$$\mathbf{E}\,[\![tu]\!]\,\sigma = \mathbf{E}\,[\![t]\!]\,\sigma\,(\mathbf{E}\,[\![u]\!]\,\sigma) \tag{2}$$

$$\mathbf{E}\,[\![\lambda x \cdot t]\!]\,\sigma = f, \text{ where } f : \mathbf{D} \to \mathbf{D} \tag{3}$$
$$f(z) = \mathbf{E}\,[\![t]\!]\,(\sigma[x := z])$$

[1] The theory (Scott and Strachey 1971) of what sorts of sets of values may be used for $\mathbf{D}$ is complicated and I will not discuss it here. I will, however, note that the theory implies the existence of an "undefined" value, written as $\bot$ and pronounced *bottom* (in Finnish *pohja*). When interpreted as a function, $\bot(x) = \bot$ for all $x \in \mathbf{D}$.

Example 1 *Let us compute the denotation of* $\lambda x \cdot x$*:*

$$\begin{aligned}
\mathbf{E}\,[\![\lambda x \cdot x]\!]\,\sigma &= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{by}\ (3)\\
&\qquad\qquad f(z) = \mathbf{E}\,[\![x]\!]\,(\sigma[x := z]) \\
&= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{by}\ (1)\\
&\qquad\qquad f(z) = (\sigma[x := z])\,(x) \\
&= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{simplify}\\
&\qquad\qquad f(z) = z
\end{aligned}$$

Thus, $\lambda x \cdot x$ *denotes the identity function* $f(x) = x$*.*

Example 2 *Let us compute the denotation of* $\lambda x \cdot a$*:*

$$\begin{aligned}
\mathbf{E}\,[\![\lambda x \cdot a]\!]\,\sigma &= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{by}\ (3)\\
&\qquad\qquad f(z) = \mathbf{E}\,[\![a]\!]\,(\sigma[x := z]) \\
&= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{by}\ (1)\\
&\qquad\qquad f(z) = (\sigma[x := z])\,(a) \\
&= f,\ \textit{where}\ f\colon \mathbf{D} \to \mathbf{D} && \textit{simplify}\\
&\qquad\qquad f(z) = \sigma(a)
\end{aligned}$$

Thus, $\lambda x \cdot a$ *denotes the constant function* $f(x) = a$*, where the value of* a *is given by the environment.*

Example 3 *Let us compute the denotation of* xx*:*

$$\begin{aligned}
\mathbf{E}\,[\![xx]\!]\,\sigma &= \mathbf{E}\,[\![x]\!]\,\sigma(\mathbf{E}\,[\![x]\!]\,\sigma) && \textit{by}\ (2)\\
&= \sigma(x)(\sigma(x)) && \textit{by}\ (1)
\end{aligned}$$

Thus, xx *denotes the application of whatever* x *denotes to itself.*

It is rule (2) that demands that every value in $\mathbf{D}$ must be interpretable as a function $\mathbf{D} \to \mathbf{D}$. So, for example, if we include nonnegative integers in $\mathbf{D}$, we must define what it means to call a nonnegative integer. One popular definition states that $0xy = y$ and $(n{+}1)xy = x(nxy)$ for all $n = 0, 1, 2, 3, \ldots$; that is, for example, $3xy = x(x(xy))$. I will presume this definition in this document.

Exercise 1 *Extend the abstract syntax and denotational semantics to allow nonnegative integer constants as well as integer addition and multiplication. Use your definitions to compute the value of*

$$(\lambda xy \cdot x \times 2 + y)\,3\,4$$

References

Barendregt, H. P. (1984). *The Lambda Calculus. Its Syntax and Semantics*. Revised. Studies in logic and the foundation of mathematics 103. Amsterdam: Elsevier.

Church, Alonzo (1932). "A Set of Postulates for the Foundation of Logic". In: *The Annals of Mathematics, Second Series* 33.2, pp. 346–366. DOI: 10.2307/1968337.

— (1933). "A Set of Postulates for the Foundation of Logic (Second Paper)". In: *The Annals of Mathematics, Second Series* 34.4, pp. 839–864. DOI: 10.2307/1968702.

— (1936). "An unsolvable problem of elementary number theory". In: *American Journal of Mathematics* 58.2, pp. 345–363.

— (1985). *The Calculi of Lambda Conversion*. Princeton University Press.

Curry, Haskell B. and Robert Feys (1968). *Combinatory Logic*. Vol. 1. North-Holland.

Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13, pp. 11–49. DOI: 10.1023/A:1010000313106.

Scott, Dana and Christopher Strachey (1971). *Toward a Mathematical Semantics for Computer Languages*. Tech. rep. PRG-6. Oxford University Computing Laboratory Programming Research Group. URL: http://repository.readscheme.org/ftp/papers/plsemantics/oxford/Scott-Strachey_PRG06.pdf.

Turing, A. M. (1937). "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. 2nd ser. 42.1, pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.