Lambda Calculus

Topics

1. Programming language classification: functional, imperative, pure, impure, object oriented

   Simplest to study mathematically is functional programming, it is a core of other languages, well related to math.

2. Functions have been key in mathematics since the 1700's.

   From the study of motion, the idea of a function emerged. By 1673 Leibniz (ancestor of most computer scientists) used the terms "function", "constant", "variable", "parameter".

   Euler 1755- New definition of function: "If some quantities depend on others in such a way as to undergo variation when the latter are varied, then the former are called *functions* of the latter"

   Dirichlet 1827 defines common notations

   $$y = f(x)$$
   $$y = x^2,$$

   but not precise, Bourbaki uses $x \mapsto x^2$

3. The move toward set theory in 1908 led to an effort to code all mathematical concepts as sets. Students are probably familiar with functions as *single valued* relations, relation $R(x, y)$ is a set of ordered pairs, a subset of $A \times B$

   For exmample $y = x^2$ on numbers $\{<0,0>, <1,1>, <2,4>, <3,0>, ...\}$, if $<a,b>$ $, <a, b'>$ appear, then $b = b'$.

4. We don't use this definition, we want a function to be a *rule* of correspondence given by an algorithm.

Church	1932	A set of postulates for the foundations of mathematics[1].
	1940	He captured this with his Lambda Calculus. [2]

We define the *pure λ-calculus* as a starting point. Its syntax is given as a collection (type) of λ-terms, inductively defined. There are these variants.

Definition 1 *Thompson book* Def 2.1
There are 3 kinds of λ-expressions:

- Variables $v_0, v_1, v_2, ...$
- Applications (e_1, e_2) for e_1, e_2 λ-expressions
- Abstractions $(\lambda x.e)$ for x a var, e a λ-expression

Definition 2[1] *λ-terms*

- Variables $x_1, x_2, ...$
- $(\lambda x M)$
- (NM)

Syntactic conventions for abbreviations:

C1. Application binds more tightly than abstraction.

$$\lambda x.xy \text{ means } (\lambda x.(xy)) \textbf{ not } ((\lambda x.x)y)$$

C2. Application associates to the left.

$$xyz \text{ means } ((xy)z)$$

C3. $\lambda x_1.\lambda x_2.\lambda x_3.e$ means $(\lambda x_1.(\lambda x_2.(\lambda x_3.e)))$

Note there are variations in the literature that we will read.

Definition 3 From Stenlund *Combinators λ-Terms and Proof Theory*, D. Reidel 1972, p.11, Ch 1 §4

- A variable
- (Possibly constants)
- (a, b) application, write $a_1a_2...a_n$ for $(..((a_1a_2)a_3)...)$
- $\lambda x.a$

Since there is so much variation and chance for ambiguity, we introduce an unambiguous definition using abstract syntax, a key idea from early work that led to Lisp. It's from one of the seminal papers. This is by John McCarthy (1963) [3].

[1] Definition 2 comes from the "Barendregt Bible", *The Lambda Calculus, its Syntax and Semantics*, N-H 1981

Definition 4 *Abstract syntax* for the Lambda Calculus - λ-terms

- Variables $x, y, z, x_1, y_1, z_1, ...$
- Abstraction $\lambda(x.t)$ $\quad t$ is a λ-term, x is a variable
- Application $ap(f; a)$ $\quad f, a$ λ-terms

The identity function	Applying the identity function to itself
Thompson $(\lambda x.x)$	$(\lambda x.x)(\lambda x.x)$
Barendregt $(\lambda x.x)$	$(\lambda xx)(\lambda xx)$
Stenlund $\lambda x.x$	$(\lambda x.x\lambda x.x)$
Abstract $\lambda(x.x)$	$ap(\lambda(x.x); \lambda(x.x))$

These definitions are all *inductive*. Thompson does not mention this. Barendregt mentions it in a footnote. Stenlund is explicit. It is clear in the abstract syntax based on defining other mathematical expressions, such as arithmetic expressions: *exp*

- Variables $x, y, z, x_1, y_1, z_1, ...$
- Constants $0, 1$
- $add(exp, exp)$
- $mult(exp, exp)$

$0,\ 1,\ add(0,0),\ mult(0,0),\ mult(0,1), ..., add(add(0,0), add(0,1)), ...$

In the Coq and Nuprl programming languages, types can be defined inductively. The Coq type for the lambda calculus is this:

$$\text{inductive term: Type} =$$
$$|var\ \ (v : var)$$
$$|lam\ (v : var)(t : term)$$
$$|ap\ (t : term)(t : term)$$

Subterms

Free Variables

$$Free(x) = x$$

$$Free(\lambda(x.b)) = Free(b) - \{x\}$$

$$Free(ap(f; a)) = Free(f) \cup Free(a)$$

Equality

α-Equality

Substitution $e[a/x]$
à la Barendregt: *with variable convention*: all bound variables are chosen different from the free variables.

$$x[a/x] = a$$
$$y[a/x] = y \quad \text{if } x \neq y$$
$$\lambda(y.b)[a/x] = \lambda(y.b[a/x])$$
$$ap(f;t)[a/x] = ap(f[a/x];t[a/x])$$

See lecture notes from Lecture 2, 2010 for an account of "safe substitution" (2.2) that allows us to safely substitute *open terms*. Why is this important?

In normal use of λ-terms and in programming languages, open terms have meaning with reference to some *context* or environment. We don't want to break that link by having the binding operator, $\lambda(x.\_\_)$, *capture* the external link.

Typically in mathematics, say calculus, we can't apply a function to itself! So $(x\,x)$ as a term and $(\lambda x.x\,\lambda x.x)$ are not common.

Here is a simple λ-term that does not appear in ordinary mathematics and might seem crazy:

$$\lambda(x.ap(x;x)) \quad \text{also written as} \quad \lambda x.xx$$

Even more strange from CS6110 lecture notes:

$$\Omega = ap(\lambda(x.ap(x;x)); \lambda(x.ap(x;x))$$
$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

References

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics, second series*, 33:346–366, 1932.

[2] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[3] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.