Lecture 25: λ Calculus

MATH230

Te Kura Pāngarau | School of Mathematics and Statistics
Te Whare Wānanga o Waitaha | University of Canterbury

Outline

Lambda Calculus

The lambda calculus is a formal language with rules of formatation, manipulation, and simplification of strings called λ-expressions.

It is the process of “simplification” - to be known as β-reduction - that is the process of computation in this model.

Some of the ideas of the lambda calculus go back to Gottlob Frege, but Alonso Church took those ideas and developed the theory proper through a series of papers in the 1930s.

This (with its type theory counterparts) has fascinating links to proofs in first-order logic and through this to program and proof verification.

Grammar of Lambda Calculus

Lambda calculus has countably many variables $x, y, z, a, b, c, \ldots$

λ-expressions are constructed from variables using the following three mechanisms.

Variables: $x, y, z, \ldots$

Application: $(t\ u)$ for λ-expressions t, u

Abstraction: $(\lambda x.\ t)$ for variable x and λ-expression t

Examples

g

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

$((\lambda x.\ x\ x)\ g)$

Notation Conventions

As with well-formed formulae in logic, we may opt to drop brackets. If you do, then all λ-expressions will be interpreted under the following conventions.

Application associates to the left e.g.
$t\ u\ v = (t\ u)\ v$

Abstraction associates to the right e.g.
$\lambda x.\lambda y.\lambda z.\ t = \lambda x.(\lambda y.(\lambda z.\ t))$

Application binds tighter than abstraction e.g.
$\lambda x.\ t\ u = \lambda x.\ (t\ u) \neq (\lambda x.\ t)u$

If you find brackets useful and clarifying, then feel free to keep them.

Interpreting λ-expressions

All λ-expressions are to be thought of as functions.

Functions are first class citizens in the λ-calculus.

Application, $(t\ u)$, is thought of as applying t to the input u.

Abstraction $\lambda x.\ u$ is a function which takes input into x and substitutes that value in every (free) instance of x in the λ-expression u.

This means we have analogous free, bound, and substitution definitions as discussed for first-order logic and quantifiers.

Free & Bound Variables

The λ abstraction operator bounds instances of its variable within the body of the abstraction expression. If x is a variable and e is a λ-expression, then x is bound in any sub-expression of the form

$$\lambda x.\ e$$

Any variable in e, other than x, is said to be free in the above sub-expression.

Examples

$(\lambda x.\lambda y.\ yx)$

$(\lambda z.zx)(x)$

$(\lambda x.\lambda y.\ yx)((\lambda z.zx)(x))$

Substitution Rules

When we substitute one λ-expression N into another λ-expression M for a variable x we replace all *free occurences* of x in M with N: this is denoted $M[x := N]$.

Substitution of λ-expressions is defined inductively as follows

$$x[x := N] = N$$

$$y[x := N] = y \text{ when } y \neq x$$

$$(M_1\ M_2)[x := N] = (M_1[x := N]\ M_2[x := N])$$

$$(\lambda x.\ e)[x := N] = \lambda x.\ e$$

$$(\lambda y.\ e)[x := N] = \lambda y.\ e[x := N]$$

As with substitution of well-formed formulae in first-order logic, this formal definition looks more complicated than the intuitive idea of substituting one term for another.

α-reduction

We do not want to make free variables bound when substituting.

Bound variables are *dummy variables* in the sense that their name is not important. Compare the functions

$$f(x) = x^2 \qquad\qquad f(t) = t^2$$

The fact that one is written in terms of x, while the other is in terms of t does not change the fact that these functions *do* the same thing: square their input.

We are free to rename bound variables in λ-expressions. This process is called α-reduction.

Examples

$\lambda x.\ x =_\alpha \lambda y.\ y$

$(\lambda x.\ (\lambda y.\ yx))\ y =_\alpha$

β-reduction

This is intended to capture the idea that application (MN) of one expression to another is to be thought of as applying the function M to the input N.

Furthermore, the abstractions $\lambda x.\ e$ are intended to be interpreted as functions which take in an x and substitute this into e.

$$(\lambda x.\ e)\ M =_\beta e[x := M]$$

λ-expressions of the form $\lambda x.\ e$ are called β-redexes.

Computation is: evaluating recursive functions, carrying out instructions for a Turing machine, and now computing β-reductions on λ-expressions.

Example

Perform a step of β-reduction on the following β-redex

$$(\lambda x.\ x)(\lambda y.\ y(\lambda z.\ zw))$$

Example

Perform a step of β-reduction on the following β-redex

$$((\lambda x.\ \lambda y.\ yx)\ f)\ g$$

Example

Perform a step of β-reduction on the following β-redex

$$(\lambda x.\ xxx)(\lambda x.\ xxx)$$

Multivariable Functions?

Multivariable functions are abundant in mathematics. Construction rules for λ-expressions only allow for the construction of unary functions. What gives? Currying means this actually isn't a problem.

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

Further Reading

Here are some recommended reading to follow up on the lecture content. Some are freely available online.

Type Theory and Functional Programming, *Simon Thompson*

- Stanford Encyclopedia Articles:
  1. The Lambda Calculus
  2. Type Theory
  3. Church's Type Theory

Church's original papers are worth visiting, although more work than Turing's paper.
An Unsolvable Problem of Elementary Number Theory, *Alonso Church*

A Note on the Entscheidungsproblem, *Alonso Church*