Brady Dean

11/12/18

- Created by Alonzo Church in the 1930s.

Lambda Calculus is made of expressions that are defined recursively:

$$< expression > := < name > \mid < function > \mid < application >$$
$$< funciton > := \lambda < name > . < expression >$$
$$< application > := < expression >< expression >$$

- The left hand side of an application is the function and the right hand side is the argument.
- This is untyped lambda calclulus.
- Everything is a function, even numbers.
    - Numbers are encoded by the number of times a function is applied to itself.
    - 0 = λs.λz. z the function s is applied to the argument z zero times
    - 1 = λs.λz. s z the function s is applied once
    - 2 = λs.λz. s (s z) the function s is applied twice
- A lambda is also known as an anonymous function.
- The only two keywords are lambda and dot.
- Function application associates to the left.
- Identifiers that do not appear in the head are free variables.
    - $(\lambda x. xy)$ – y is a free variable
    - An identifier is free if it is unbound from an expression.
- All identifiers are local to their function.
    - $(\lambda x. x)(\lambda x. xy)$ – These are two distinct x's.
- If substituting E brings an unbound into a bound expression, the variable is renamed.
- One way of making this distinction properly is to rename bound variables during substitution, making sure to always give them unique names. This is called α-conversion and expressions that only differ in bound variable names are considered α-equivalent or even completely equivalent.
- $(\lambda x. M)N = M[x \coloneqq N]$
- Application associates to the left
- Abstraction associates to the right
- Beta Reduction – Process to evaluate expressions
    - The argument expression is bound to the input variable
    - All occurrences of the bound variable are replaced
    - The head is removed
    - $\big(\lambda xyz. xz(yz)\big)(\lambda mn. m)(\lambda p. p)$
      $\big(\lambda yz. (\lambda mn. m)z(yz)\big)(\lambda p. p)$

$$\left(\lambda z.(\lambda mn.m)z\left((\lambda p.p)z\right)\right)$$
$$\left(\lambda z.(\lambda n.z)\left((\lambda p.p)z\right)\right)$$
$$(\lambda z.z)$$

- Alpha Conversion protects from name collisions during application
  - $(\lambda x.xx)(\lambda x.x) \to (\lambda x.xx)(\lambda x'.x')$
  - $(\lambda x.xx)(\lambda x.x) \to (\lambda x.xx)(\lambda y.y)$
  - Occurs when both expressions share the same variable name. Only one needs to change.
- Common functions
  - $I = (\lambda x.x)$
  - $Y = (\lambda f.\left(\lambda x.f\ (x\ x)\right)\left(\lambda x.f(x\ x)\right))$
  - $K = (\lambda xy.x)$
  - $(\lambda xy.y)$
- Lambda calculus is typically evaluated in normal order (leftmost expression first)
- There is also Applicative order where the arguments to functions are evaluated first. This is seen in Lisp
- Call by Need is similar to normal order but expressions are not evaluated until needed. This is known as Lazy Evaluation in Haskell
- Beta-reduction may be done in any order according to the Church-Rosser property so implementations may parallelize the process to increase efficiency.