Basics

October 6, 2023

Contents

1 Rules

Lambda Calculus sounds more daunting than it is. There are only a few rules to the calculus of functions.

Notation
A function $f(x) \to x^2$ can be expressed as $\lambda x.x^2$. λ denotes the beginning of an anonymous function. These functions are function closures. Every λ can take one argument, which precedes a ".", after which is the definition of the function.

Some notations allow shorthand for functions to take multiple arguments. λxyz is shorthand for $\lambda x \lambda y \lambda z.(...)$. Note that the outer function λx returns another function $\lambda y \lambda z.(...)$. This is a partially applied function, or a curried function. Any use of the variable x in the returned function is preserved.

While in pure expression form there is no such thing as a named function, productive notation demands it. $f = \lambda x.x$ names the identity function as f.

Composition
To pass an argument to a function, or compose a function within another function, simply write them consecutively. fx passes x to f. This operation is fully left associative, so $fxyz$ passes x to f, y to the result of $f(x)$, and z to the result of $f(x)(y)$. This expression can therefore be expressed as $f(x)(y)(z)$ or $((fx)y)z$ all the same.

Scope
The arguments within a function are bound to that function, any use within that funciton or any subfunctions are bound to the state of the outer most argument, arguments cannot be reused in subfunctions or overwritten in any way.

Equivalency
There are a variety of equivalency notations. Alpha equivalence: $=_\alpha$ denotes functional equality between expressions.

$$\lambda x\lambda y\lambda z.zxy =_\alpha \lambda a\lambda b\lambda t.tab$$

Beta reduction or beta equivalence denotes equivalence in structurally differing expressions, as a sign of a reduction or progression of the functions evaluation.

$$(\lambda a\lambda b\lambda z.zab)(1)2 \rightarrow_\beta \lambda z.(1)2$$

2 Computation

This is a mathematically sound and computationally pure expression of programs. It is a Turing equivalent system. The most atomic unit is a function, everything function is immutable, everything is parallel, and the only operation is function composition. From these simple rules we can build any programatic structure present in modern programming languages and any form of digital computation whatsoever. This is also the theory behind any functional programming dialect.

3 Basic Definitions

With that, lets begin with a small example to demonstrate how this system begins show it's Turing complete potential.We can define true and false as functions which take two arguments and return the first or second one respectively.

$$\mathrm{T} = \lambda x \lambda y.x$$

$$\mathrm{F} = \lambda x \lambda y.y$$

It doesnt really matter what this means beyond the fact that we can use them to derive distinct behavior, which we very easily can. Some people like to define explicit conditional branching functions, such as if.

$$\mathrm{if} = \lambda c \lambda a \lambda b.cab$$

If we pass β equivalent boolean function as a condition and two other functions for the branches, we get the following reductions.

$$\mathrm{if}(\mathrm{T})ab =_\beta \mathrm{T}ab =_\beta a$$

$$\mathrm{if}(\mathrm{F})ab =_\beta \mathrm{F}ab =_\beta b$$

So we can see that passing true returns the first function, and passing false returns the second function. This is however simply a more expressive notation over simply passing the conditions to a boolean directly.

Boolean operators follow.

$$\mathrm{and}_b = \lambda x \lambda y.xy\mathrm{F}$$

$$\mathrm{or}_b = x \lambda y.x\mathrm{T}y$$

$$\mathrm{not}_b = \lambda x.x\mathrm{FT}$$

$$\mathrm{eq}_b = \lambda x \lambda y.xy(\mathrm{not}_b y)$$

$$\mathrm{xor}_b = \mathrm{neq}_b = \lambda x \lambda y.\mathrm{not}_b(\mathrm{eq}_b xy)$$

We can also begin to define basic data structures such as a pair, the notation of which I like to shorten to express a 2 item vector.

$$\upsilon_2 = \lambda a \lambda b \lambda z.zab$$

Adding items to a pair is as simple as calling the function with arguments you want to add.

$$\upsilon_2(1)(2) \rightarrow_\beta \lambda z.(z1)2$$

Retrieving these items from the pair structure is now trivial, pass a boolean to the function. This is a common operation, so to increase expressiveness we will define renamed versions of booleans to use for datastructure navigation with notation borrowed from relational algebra.

$$\pi_1 =_\alpha \mathrm{T}$$

$$\pi_2 =_\alpha \mathrm{F}$$

Project 1, and project 2 will be the standard pair indexing operations we will use from here on out.

$$a = \upsilon_2(5)(7)$$

$$a\pi_1 \rightarrow_\beta 5$$

$$a\pi_2 \rightarrow_\beta 7$$

4 Recursion

To achieve recursion for repeated operation, we can use one of many combinators. The most standard one being

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

which is an expression which infinitely passes itself to itself. We can mutate this expression to achieve more useful recursive functions, such as the Y-combinator.

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$