Lambda Calculus

1 Introduction

Let us describe imagine a function

$$f : \mathbb{Z} \to \mathbb{Z} \tag{1}$$
$$f(x) = 3x \tag{2}$$

In various programming languages we may write this as

```
def triple(x):
    return 3*x

triple::Integer->Integer
triple x = 3*x

int triple(int x):
    return 3*x
```

In general a function take one argument and returns one body. We write a function in lambda calculus as

$$\lambda[argument].[body] \tag{3}$$

f will be written in the lambda calculus as $f = \lambda x.(3x)$. To apply this function we say that $f5 = (\lambda x.(3 * x))5 = (3 * 5) = 15$.

In lambda calculus, we don't have an arithmatic as axioms; the only thing we have is λ, a symbol that allows us to construct functions. But what do these functions take as arguments and return? Other functions, of course. Why would we do this? You will see.

2 Lambda terms

The syntax of lambda calculus is particularly simple. We define a lambda term as:

1. a variable, x, is a lambda term
2. if t is a lambda term, and x is a variable, then $\lambda x.t$ is a lambda term (called a lambda abstraction)
3. if t and s are lambda terms, then ts is a lambda term (called an application)

A lambda abstraction $\lambda x.t$ represents an anonymous function that takes a single input, evalutes t with x bound to the input, and then returns t.

An application ts can be thought of as calling t with s as the input.

As a final point, the "=" sign is not part of lambda calculus but we use it to say that two lambda terms are equivalent.

3 Simple examples

The Identity function, $Ix = x$, is defined as $I = \lambda x.x$

The Constant function, C_r that always returns r, is defined as $C_r = \lambda x.r$

What does the function $T = \lambda a.\lambda b.a$ do?

[Currying]

4 Church numerals

Functions in LC take other functions as arguments and return other functions. This by itself is not very fun, so we must begin embedding data into functions. Let's start by embedding numbers.

The higher-order function that represents natural number n is a function that maps any function f to its n-fold composition f^n. Let us derive it.

$$nf = f^n \tag{4}$$
$$nfx = f^n x \tag{5}$$
$$n = \lambda fx.f^n x \tag{6}$$

so

$$0 = \lambda fx.x \tag{7}$$
$$1 = \lambda fx.fx \tag{8}$$
$$2 = \lambda fx.f(fx) \tag{9}$$
$$3 = \lambda fx.f(f(fx)) \tag{10}$$
$$\tag{11}$$

or $nfx = f^n x$.

let us derive the S or successor function. We observe that $Sn = \lambda fx.f^{(n+1)}x = \lambda fx.f(f^n x) = \lambda fx.f(nfx)$; doing an abstraction $S = \lambda nfx.f(nfx)$.

the + or addition function and × or multiplication function is

$$+ = \lambda mnfx.mf(nfx) \tag{12}$$
$$\times = \lambda mnfx.m(nf)x \tag{13}$$

we have the amazing result that

$$exp(m, n) = \lambda mn.nm \tag{14}$$

Interesting observation: in Peano arithmetic we would have axiomatically defined "zero" and "successor", and let zero = z, one = sz, two = ssz etc. We don't have s and z but we have the really powerful λ that allows us to abstract over z and s. Cool eh?

5 Booleans

The construction of booleans is a little unusual but not illogical. We use booleans to make control flow decisions, the most important of which is the ITE or if-then-else control structure that you are all familiar with. Hence if b is a boolean we declare it to have the property that

$$bxy \tag{15}$$

evaluates to x if b is true and y otherwise. Then

$$T = \lambda ab.a \tag{16}$$
$$F = \lambda ab.b \tag{17}$$

ITE is trivial to define as:

$$ITE = \lambda bpq.bpq \tag{18}$$

it is also obvious that

$$NOT = \lambda pab.pba \tag{19}$$
$$AND = \lambda pq.pqF \tag{20}$$
$$OR = \lambda pq.pTq \tag{21}$$

for example $(ORpq) = (pTq)$ which is "T if p else q", which is a correct definition of OR.

6 Recursion

As it stands lambda expressions cannot refer to themselves, hence we cannot write recursive functions. Or can we not?

I shall write this section in Lisp syntax to make it clearer. We will use the good old function fac.

```
fac =
(lambda (n)
    (if (= n 0)
        1
        (* n (fac (- n 1)))))
```

Well fib is a recursive function so we can't define it as it is in the lambda calculus. To define it we first define

```
F =
(lambda (f n)
    (if (= n 0)
        1
        (* n (f (- n 1)))))
```

what is (F fac)? it is a function of one variable,

```
(F fac) =
(lambda (n)
    (if (= n 0)
        1
        (* n (fac (- n 1)))))
```

We see that (F fac) = fac. fac is the fixed point of F.

The Y combinator takes F and returns the fixed point of F, YF, such that FYF = YF. Egad! We use lambda abstraction (on YF) to write $Y = \lambda F.FYF$ but shit it uses recursion...

7 Fixed point combinator

Recall we wish to find a combinator Y such that YF = FYF, in other words YF is the fixed point of F.

The solution to this is one of the most beautiful results of LC. To get a glimpse of it we take a little detour.

I want to introduce the terminology $\to$ to mean "reduces to"; we will define it more carefully later but for now it means "apply the substitution", so $(\lambda x.xx)y \to yy$. We have $a \to b \implies a = b$.

Consider

$$\omega = (\lambda x.xx)(\lambda x.xx) \tag{22}$$

Obviously $\omega = \omega$ but we have the rather curious property that $\omega \to \omega$. A small modification leads to

$$\begin{aligned} YA &= (\lambda x.xx)(\lambda x.Axx) && (23)\\ &\to (\lambda x.Axx)(\lambda x.Axx) && (24)\\ &\to A(\lambda x.Axx)(\lambda x.Axx) && (25)\\ &= AYA && (26) \end{aligned}$$

so

$$Y = \lambda A.(\lambda x.Axx)(\lambda x.Axx) \tag{27}$$

$$\begin{aligned} Z &= \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)) && (28)\\ Y_k &= (LLLLLLLLLLLLLLLLLLLLLLLLLL) && (29)\\ L &= \lambda abcdefghijklmnopqstuvwxyzr.(r(thisisafixedpointcombinator)) && (30) \end{aligned}$$

8 Reduction rules

9 Completeness and Incompleteness theorems