Lambda Calculus – Notes

Lambda calculus is an abstract way of representing a programming language, commonly known as the *smallest universal programming language* there is. The lambda calculus is equivalent to a single-tape Turing machine, where all abstractions are composed of functions. The way function are representing are through the syntax:

$$\text{function} := \lambda < argument > . < expression >$$

where the argument and expression are composed of placeholder variables. For example, the identity function $f(x) = x$ can be represented in the lambda calculus as

$$(\lambda x.x)$$

We can also apply these functions to particular arguments, for example, the identity function evaluated at y would be

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

Notice that when functions are being evaluated, the value gets "substituted" into the function. For this reason, we use the notation $[y/x]$ to denote that y is being substituted in for x:

$$(\lambda x.x)y \to [y/x]x \to y$$

Free vs. Bound Variables

Bound variables are variables that are being bound by a λ operator. For a variable to be bound, it must be the argument of a function. **Free variables** are variables that do not have this property. For example, in $(\lambda x.x)y$, x is a bound variable, and y is a free variable.

There are some cases in which a variable can both be bound and free; freedom is always relative to an expression. For example, in the expression $(\lambda x.xy)(\lambda y.y)$, y is both a bound variable (in the function $\lambda y.y$) and a free variable (relative to the entire expression).

Reduction

Our ultimate goal is to take these expressions and have ways to reduce them into smaller pieces. There are three main ways:

α-reduction

Note that the exact letters used in a particular function don't matter; in fact, we could have used z instead in the identity function: $\lambda z.z$. Substituting letters like this is known as **α-reduction**, as particular variable names don't matter as long as you change it throughout an entire abstraction.

However, we must worry about **name capture**, changing the scope of a variable. The two functions $\lambda y.\lambda z.y$ and $\lambda y.\lambda y.y$ demonstrate this. The first is a function that returns a constant mapping function to the variable y. The second returns the identity function. We must be careful when we are performing α-reduction so as to not cause this from happening.

β-reduction

β-reduction is very straightforward; it is when you evaluate a function at an argument expression.

$$(\lambda x.M)z \to M[z/x]$$

Arithmetic

We can do arithmetic using lambda calculus. To do this, we define the function $\lambda s.(\lambda x.x)$ (the function that returns the identity function) to be zero. Then, we can define the rest of the natural numbers as follows: 1 is the function that returns $x \to f(x)$, 2 is the function that returns $x \to f(f(x))$, and so on.

$$1 \coloneqq \lambda s.(\lambda z.s(z))$$
$$2 \coloneqq \lambda s.(\lambda z.s(s(z)))$$
$$3 \coloneqq \lambda s.(\lambda z.s(s(s(z))))$$

From here, we can define a few operators that take on lambda function forms and can be interpreted in a clean way:

- The **successor function** take an argument x and outputs $x + 1$:

  $$\text{succ} \equiv \lambda n.\lambda f.\lambda x.f(nfx)$$

  The intuition behind this function is that given a lambda representation of an integer m, which is a function returning a function composed m times, n in the successor function is the dummy variable for this input. But note that nfx will return $f^n(x)$, because if $n = \lambda s.\lambda z.s^n(z)$, then

  $$nfx = (\lambda s.\lambda z.s^n(z))fx = (\lambda z.f^n(z))x = f^n(x)$$

  Thus, from the extra f composition in the successor function, it will output $f^{n+1}(x)$.

- The **addition function** adds two numbers, taking advantage of the fact that $f^{m+n}(x) = f^m(f^n(x))$:

  $$\text{plus} \equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$