λ Calculus – Church Numerals and Lists

Michael Smith

May 13, 2004

1 Church Numerals

In the pure λ calculus, there is no concept of a 'number' – the only things we get to play around with are function definitions and applications. We therefore need to define the natural numbers in terms of *functions*. If we hark back to Peano's postulates, all we need is:

- A **zero** function – i.e. $\texttt{zero}(x_1, \ldots, x_n) = 0$.
- A **successor** function – i.e. $\texttt{succ}(n) = n + 1$.

Notice here how these correspond to the definition of *primitive recursive functions*. Our λ calculus clearly allows us to model the other primitives very easily (projection, composition and primitive recursion) if we felt inclined to do so.

We now have the problem of how to translate the above into λ terms. Remember, however, that we are looking for functions that are themselves numerals – not ones that generate numerals for us. To this end, we do not need to define a successor function – we could just pick any arbitrary function and apply it n times to add n to its argument. So the only problem is in defining zero. Again though, this isn't a problem, as zero is just the successor function applied *zero* times to its argument (which can be arbitrary). So we define the natural numbers as:

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.fx \\ 2 &= \lambda f.\lambda x.f(fx) \\ &\vdots \end{aligned}$$

Note that x and f are our 'zero' and 'successor' functions respectively, but we can't leave them as free variables (given that they are arbitrary), which is why we need the λ bindings. These are known as *Church numerals*.

Now that we've got numbers, we can define some arithmetic expressions. Let's start simple. Suppose we have a Church numeral n, and want to add '1' to it – all we need to do is apply f one more time to it. The only thing we need to remember to do is to strip the λ's from n, and shove them back on again at the end:

$$\texttt{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

Similarly, adding two numbers together is nice and easy, since we just need to replace the x in one of them by the whole of the second number. So to add n_1 and n_2:

$$\texttt{add} = \lambda n_1.\lambda n_2.\lambda f.\lambda x.n_1 f(n_2 fx)$$

Multiplication also turns out to be easy, as in order to get n_1 copies of n_2, just put n_2 into each f in n_1. The idea is simple - we just need to juggle things round a bit so that the x in each copy of n_2 disappears (this is done by passing in $n_2 f$, which is a function that takes one argument, x, and so chains the new number together by itself):

$$\texttt{mult} = \lambda n_1.\lambda n_2.\lambda f.n_1(n_2 f)$$

Just to round things off, exponentiation is the easiest of the lot! All we need to do is pass n_1 into n_2, causing it to be multiplied by itself n_2 times. So $n_1^{n_2}$ is:

$$\texttt{exp} = \lambda n_1.\lambda n_2.n_2 n_1$$

Consider all the operations we've done so far - they've all produced *bigger* numbers than we started with. So, you ask, what about operations like division, or subtraction? It may be of comfort to know that these operations can be done, but we should first consider a much simpler task – that of calculating the predecessor of a number (i.e. subtracting 1). As a first guess, you might come up with a function like the following:

$$\texttt{wrong\_pred} = \lambda n.\lambda f.\lambda x.(\lambda a.\lambda b.b)(nfx)$$

You might be under the impression that the $\lambda a.\lambda b.b$ will 'eat up' the first f in n, giving us the required answer. So why is this not the case? Simply because f is *not* the first argument to this expression. Remember the brackets – we cannot say that $\theta(f(fx)) = \theta(f)(fx)$, since we would be magically moving the first argument of f to the second argument of θ. The actual result of the above is to consume the *whole number*.

Demoralised by our failure, how can we come up with a solution to this problem? If we think a little, it's possible to imaging crawling along the number, counting how many f's we go past (by building up a new number). If we can do this, then why not, in addition to counting the f's, remember how many f's we had previously counted. When we get to the end of n, we're then holding both n *and* $n-1$. Take a look at this:

$$(0,0) \to (0,1) \to (1,2) \to (2,3)\ldots$$

Starting with the pair $(0,0)$, we just apply n transitions to get to the pair $(n-1,n)$. So we're looking to give the number n the pair $(0,0)$ as its x argument, and a function that corresponds to these transitions as its f argument. Clearly this function just needs to take (a,b) as its argument and produce $(b,b+1)$.

The only thing left to say before we write this predecessor function down, is that we need a way of encoding pairs. If you are worried about just assuming that such a representation exists, let me point out that the following functions will work, where `pair` is the constructor, and `#1` and `#2` are the projections:

$$\begin{aligned} \texttt{pair} &= \lambda a.\lambda b.\lambda f.fab \\ \#1 &= \lambda p.p(\lambda a.\lambda b.a) \\ \#2 &= \lambda p.p(\lambda a.\lambda b.b) \end{aligned}$$

We can now, without cause for concern, write down the predecessor function:

$$\texttt{pred} = \lambda n.\#1\ (n\ [\lambda p.\texttt{pair}\ (\#2\ p)\ (\texttt{succ}(\#2\ p))]\ [\texttt{pair}\ 0\ 0])$$

By substituting the transition function for f in n, it gets applied n times to the pair $(0,0)$ (x). Thus the only thing left to do is to extract the first item in the pair at the end, which is (happily) $n-1$!

Notice that we can now construct a subtraction function – just pass in `pred` as f, and n_1 as x, to n_2 in order to subtract 1 from n_1, n_2 times. To compute $n_1 - n_2$:

$$\texttt{subtract} = \lambda n_1.\lambda n_2.n_2\ \texttt{pred}\ n_1$$

2 Lists

We can now extend these ideas to consider a representation of lists. Instead of just applying a function f multiple times to x, we can imagine passing each f a value a_n. This means that we can construct lists of values, in the following way:

$$\begin{aligned} [] &= \lambda f.\lambda x.x \\ a :: [] &= \lambda f.\lambda x.fax \\ b :: a :: [] &= \lambda f.\lambda x.fb(fax) \\ &\vdots \end{aligned}$$

Notice that this is exactly what the *fold left* functional does in ML, if we pass the function in as f.

As this form is using the same structure as the Church numerals, similar functions are going to apply. For example, the `cons` operation is a lot like `succ` , except that we have to add a value to the list as well. To construct a::l:

$$\texttt{cons} = \lambda a.\lambda l.\lambda f.\lambda x.fa(lfx)$$

Another simple operation is that to extract the head of the list, although this doesn't really have an analogy with the numerals. All we need to do is let f be a function that returns its first argument, and ignores the rest (we could also be sneaky and let x be the empty list, so that we return a vaguely sensible result if we try to extract the head of an empty list). To find the head of l:

$$\texttt{head} = \lambda l.l(\lambda a.\lambda b.a)x$$

As expected, the interesting case is the analogy to `pred`– that of extracting the tail of the list. The expression is pretty much identical, except that we start with the pair $([], [])$, and have to remember to hold onto the list values as we reconstruct it and its tail. To find the tail of l:

$$\texttt{tail} = \lambda l.\#1\ (l\ [\lambda p.\lambda q.\texttt{pair}\ (\#2\ q)\ (\lambda f.\lambda p.f\ p\ (\#2\ q\ f\ p))]\ [\texttt{pair nil nil}])$$

It's interesting to note that this representation of a list has certain functions (such as a map, or a fold function) that are easy to implement with it, whereas others (such as tail) turn out to be much harder than expected. Conversely, if we were to represent a list as a nested sequence of pairs, the tail operation would be trivial, whereas doing something like map would be quite tricky. Clearly there is a trade-off to be had between the various ways of representing list structures in the λ calculus.