Non-Computable Functions in the Abacus Model

Foundations of Computer Science, Fall 2018

We show that there is at least one function $\mathbb{N} \longrightarrow \mathbb{N}$ that is not computable, according to the abacus model we introduced last time.

To do that, we first need a bit of mathematical *legerdemain.* First, note that we can taken any k-tuple of natural numbers, and encode it with a single natural number in such a way that we can recover the components of the tuple from that single natural number.It's easier to proceed by examples.

Consider the tuple (n_1, n_2). We can encode that tuple using a single natural number $2^{n_1}3^{n_2}$. To recover n_1 and n_2 given that natural number, we simply need to find the unique prime factorization and extract out n_1 and n_2 as the number of times 2 and 3 are factors, respectively.

Consider the tuple (n_1, n_2, n_3, n_4). We can encode that tuple using a single natural number $2^{n_1}3^{n_2}5^{n_3}7^{n_4}$. To recover n_1, n_2, n_3, and n_4 given that natural number, we again find the unique prime factorization and extract out n_1, n_2, n_3, and n_4 as the number of times 2, 3, 5, and 7 are factors.

In the general case, we can encode a k-tuple using the first k prime numbers, in a similar way. We call this a *Gödel encoding* of a tuple of natural numbers.

Using this trick, we can encode any flow diagram (a program in our abacus model) using a single natural number that uniquely describes that flow diagram. Recall that we can represent a flow diagram using a sequence of 5-tuples, where each tuple consists of a node name, a location name, an operation to be performed at the location, and two node names to go to when the operation succeeds and fails, respectively. Clearly, we can use natural numbers to represent node names, and we can also use natural numbers for location names. (We used names like X and Y in class, but we can also use 0, 1, 2, and not lose any generality.) For the operation, we can use 1 for $+$, 2 for $-$, and 3 for no operation. In other words, we can represent a flow diagram with a sequence of 5-tuples of natural numbers. Since a flow diagram is finite, this is just a tuple of 5-tuples of natural numbers. Using the above Gödel encoding, we can encode that tuple of 5-tuples into a tuple t of natural numbers, by encoding each 5-tuple into its own natural number. We can then take that tuple of natural numbers t and encode it in turn into a single natural number. By factoring, we can regain the encoding of the 5-tuples, and then factoring each of those we can recover the content of all those 5-tuples.

So given any natural number n, either n is the encoding of a flow diagram with one input location as described above, or it's not possible to decode n into a sensical flow diagram with one input location. For each natural number n, let p_n be the flow diagram such that the encoding of p_n is n, and let p_n be the simple flow diagram that moves from **start** to **stop** immediately. Thus, for every natural number n, p_n is a well defined flow diagram. Moreover, for any flow diagram p with one input location, there is a number n such that $p = p_n$: just encode p to get that n!

For a flow diagram p with one input location, let $[p]$ represent the function $\mathbb{N} \longrightarrow \mathbb{N}$ that p calculates.

We have all the ingredients now. I claim that the following function $W : \mathbb{N} \longrightarrow \mathbb{N}$ is not computable — that is, there is no flow diagram that calculates W. Function W is defined as follows:

$$W(n) = [p_n](n) + 1$$

That is, function W at argument n is the same as the value of the function that p_n calculates applied to n, plus 1. This is a perfectly well-defined function. (Though we can debate its usefulness beyond being an example here.)

The argument that W is not computable is super easy. Suppose it *were* computable. Then there would be a flow diagram P (necessarily with one input location) that calculates W. Let's look at P more carefully. Since P is a flow diagram, you can encode it. Say it has code K. Then $P = p_K$ as per the above remark. Since P calculates W, that means that $[P](x) = W(x)$ for all x. But look at this:

$$\begin{aligned} W(K) &= [p_K](K) + 1 \\ &= [P](K) + 1 \qquad \text{(since } p_K = P\text{)} \\ &\neq [P](K) \end{aligned}$$

This makes no sense, we have both that $W(K) = [P](K)$ and $W(K) \neq [P](K)$! That can't be. Since this all follows from our supposition that a P exists that computes W, this means that no such P exists.

(This kind of argument is called a diagonalization argument, made famous by Cantor in the late 19th century.)