Math 29: Universal Machines

April 13th, 2022

1 The Universal Machine

Last time, we discussed how to code register machines in such a way that we can computably determine what the machine coded by a given number does and simulate its behavior. This process can then, in effect, run any machine by having access to the number which codes it. This is called a **univeral machine**, and the fact that Turing and register machines compute the same functions means there are examples of universal machines for both types.

We will not give a full example of a universal machine, as the existence is the most important part. One should think about universal machines as similar to operating systems like Windows, MacOS, or Linux. Users of an operating system can install programs in the form of binary files, i.e. large numbers in binary. The operating system can then decode those files into a set of instructions to then run. We can even create new programs and run them inside the operating system at run-time, which is something we can do through a universal machine as well.

We will fix a universal machine U (either register or Turing - it doesn't matter). We assume that the machine takes a single input $k = p(e, n)$, which runs the machine coded by e on input n. (In the coding we gave, not all numbers were codes for machines. We can assume anything not given as a code is a code for the machine that diverges on every input.) Then we will use the notation

$$U(k) = U(\langle e, n \rangle) = \varphi_e(n)$$

to represent running the e-th machine with input n. As with all partial functions, we use $\varphi_e(n) \uparrow$ to mean φ_e diverges on input n, i.e. the machine never stops running. If it does stop running and outputs m, we write $\varphi_e(n) \downarrow = m$. As our pairing function is computable, we can abuse notation and allow φ_e to take in any finite number k of inputs as opposed to specifying that it decodes one input into k.

The following theorem justifies why we allow partial computable functions and do not require them to be total.

Theorem 1. *There is no machine T such that the T-functions indexed by e (i.e. $f_e : \omega \to \omega$ defined via $f(n) = T(\langle e, n \rangle)$) are exactly the total computable functions.*

Proof: Suppose for the sake of contradiction that there were such a T. Then we defined a total computable function as follows. Define $g : \omega \to \omega$ via $g(n) = T(\langle n, n \rangle) + 1$. It is not hard to see that g is a total computable function. To compute it, we run the program T on input $\langle n, n \rangle$. Since T lists out exactly the total computable functions, it halts on all inputs, so we can wait for it to converge, then add one. (If it were not total, then $g(n) \uparrow$ whenever $T(\langle n, n \rangle) \uparrow$.)

However, since $T(\langle e, n \rangle)$ lists out all of the total computable functions, there must be some k such that $g(n) = T(\langle k, n \rangle)$ for all n. (Otherwise, g would be a total computable function not in the list.) But then we have that

$$g(k) = T(\langle k, k \rangle) + 1 = T(\langle k, k \rangle)$$

This is a contradiction, so there can be no such T.

This may remind you of Cantor's diagonalization proof that the real numbers have a different cardinality than the natural numbers. **Diagonalization** is a very important idea in computability theory, which we will see multiple times. Essentially, if you want to ensure that some list cannot contain every object of a certain type, you exploit the list to construct a new object which cannot be in the list.

While there may not be a list of the total computable functions, then

Lemma 2. *(The Padding Lemma) For any e, there are infinitely many k such that $\varphi_e = \varphi_k$. In fact, there is a total computable, increasing function f such that, for all n, $\varphi_{f(n)} = \varphi_e$.*

Proof: Fix e. Then let M be some register machine such that $\#M = e$, and let R_i be a register unused by M. Then for each k, we can defined M_k to be the register machine which starts with k R_i^+ nodes followed by an R_i^--node which empties R_i, and when empty, runs M.

Then for all k, $\#M_{k+1} > \#M_k$, and M_k computes the same function as M. Finally, define $f(n) = \#M_n$, which is total computable because our coding of register machines is effective.

Lemma 3. *There is a total computable function $s : \omega^2 \to \omega$ such that*

$$\varphi_e(x_0, \ldots, x_k) = \varphi_{s(e,x_k)}(x_0, \ldots, x_{k-1})$$

Proof: Given e and x_k, let $s(e, x_k)$ return the code of the register machine M which takes in $x_0, \ldots, x_{k-1}$ and starts with x_k-many R_k^+ nodes, then runs U with inputs e, x_0, $\ldots$, x_k.

s is total computable because our coding of machines is effective, and we can compute its code because our coding of machines is effective.

Theorem 4. *(The s-m-n Theorem) For any n and m, there is a total computable function $s_n^m : \omega^{n+1} \to \omega$ such that, for all e,*

$$\varphi_e(y_0, \ldots, y_{m-1}, x_{n-1}, \ldots, x_0) = \varphi_{s_n^m(e,x_0,\ldots,x_{n-1})}(y_0, \ldots, y_{m-1})$$

Proof: We argue by induction on n. $n = 1$ was just the previous lemma. Now suppose there is such an s_n^m for all m, and consider $n+1$. Define s_{n+1}^m via $s_{n+1}^m(e, x_0, \ldots, x_n) = s(s_n^{m+1}(e, x_0, \ldots, x_{n-1}), x_n)$. Then

$$\varphi_{s_{n+1}^m(e,x_0,\ldots,x_n)}(y_0, \ldots, y_{m-1}) = \varphi_{s(s_n^{m+1}(e,x_0,\ldots,x_{n-1}),x_n)}(y_0, \ldots, y_{m-1})$$

By the previous lemma, this is equal to

$$\varphi_{s_n^{m+1}(e,x_0,\ldots,x_{n-1})}(y_0, \ldots, y_{m-1}, x_n)$$

By the induction hypothesis, then, this is equal to

$$\varphi_e(y_0, \ldots, y_{m-1}, x_n, \ldots, x_0)$$

as desired. Thus we have shown the theorem for all n by induction, and the code for s_n^m can be computed because our coding of machines is effective.

A sequence of functions $\{f_e\}_{e\in\omega}$ is **uniformly computable** if there is a total computable function g such that, for all n and e, $g(e,n) = f_e(n)$. That is, there is a single machine which can compute every function using indices for them. The universal machine witnesses that the partial computable functions are uniformly computable, whereas the lemma above shows that the total computable functions are not.

Uniformly computable functions allow us to define computable functions using infinitely many other computable functions, as there is actually a single program which can simulate all of them at once. We shall exploit this fact often, particularly when it comes to the listing of all partial computable functions.

Lemma 5. *There is a partial computable function g which cannot be extended to a total computable function. I.e., there is partial computable g such that for all total computable functions f, there is n with $g(n)\downarrow \neq f(n)$.*

Proof: Define $g(n)$ to be $\varphi_n(n)+1$. Notice that $g(n)\downarrow$ if and only if $\varphi_n(n)\downarrow$.

Now suppose for the sake of contradiction that there is some total computable function f such that $f(n) = g(n)$ whenever $g(n)\downarrow$. Then $f = \varphi_e$ for some e, in which case

$$f(e) = \varphi_e(e) = g(e) = \varphi_e(e) + 1$$

$\varphi_e(e)\downarrow$ because f is total. But this is a contradiction, therefore no such f can exist.

Notice the following:

Lemma 6. *If the domain of φ_e is computable, then it can be extended to a total computable function.*

Proof: Suppose the domain of φ_e is computable with characteristic function g. Then define $f : \omega \to \omega$ via

$$f(n) = \begin{cases} \varphi_e(n) & \text{if } g(n) = 1 \\ 0 & \text{if } g(n) = 0 \end{cases}$$

Then f is total because g is, and $\varphi_e(n)\downarrow$ whenever $g(n) = 1$. (As this means n is in the domain of φ_e.) Furthermore, it is computable because g is.

A set X is said to be **computably enumerable** (or c.e.) if it is the domain of φ_e for some e. Then we write $X = W_e$. The universal machine gives us not only an enumeration of the partial computable functions, but of the c.e. sets as well. Notice that, by combining the above two lemmas, we see that there is a c.e. set which is **not** computable. Specifically, the domain of any partial computable function which cannot be extended to a total computable function is computably enumerable but not computable.

The c.e. sets are incredibly important for computability theory. They are sets which can be listed out by an algorithm, but there is no way in general to know if a number will never show up in the list. We will see that our primary examples of non-computable sets are c.e., and that they have a rich structure.

Lemma 7. *A set is computable if and only if it and its complement are c.e.*

Proof. Homework 3 Question 2. □

Lemma 8. *Prove that every c.e. set contains a computable subset.*

Proof. Homework 3 Question 3. □