# Math 29: Noncomputability

April 18th, 2022

## 1 Noncomputable Functions

A (partial) function is **not computable** exactly when one might think it would be, i.e. if there is no $\varphi_e$ which witnesses that it is computable. As we have seen with the recursion theorem, it is not that easy to diagonalize out of the partial recursive functions. However, cardinality assures us that almost all functions are not computable: the cardinality of total functions from $\omega$ to $\omega$, i.e. the cardinality of $\omega^\omega$, is equal to that of the real numbers, which by Cantor's diagonalization is not countable. However, our coding of machines is an injection from the set of machines into the set of naturals, so there can only be countably many machines. Thus there are uncountably many functions, but countably many computable ones.

Notice that being not computable is a property of infinite objects: finite objects can be hard-coded into a machine.

Last week, we saw that there was a partial computable function $f$ which could not be extended to a total computable function. However, there are infinitely many ways to extend it to a total function: for every $n$ such that $f(n)\downarrow$, define $g(n) = f(n)$. If $f(n)\uparrow$, then choose $g(n)$ however you like. In fact, there are uncountably many total functions which extend $f$: The complement of the domain of $f$ is countable, as a subset of $\omega$, so given any function $h \in \omega^\omega$, we can define $g(n)$ to be $h(k)$, whenever $n$ is the $k$-th element of the complement of the domain of $f$.

## 2 Noncomputable Sets

Often, we will be more concerned with noncomputable sets: sets whose characteristic and principle functions are not computable. (As shown last week, it is enough for one of them to not be computable.) For the above example, we showed that a partial computable function can be extended to a total computable function if its domain is computable. Therefore, the domain of $f$ must be an example of a noncomputable set. In fact, given a noncomputable total function, we can always get a noncomputable set from it.

**Lemma 1.** *The graph of a (partial) function $f$, $graph(f)$, is the set $\{\langle n, k\rangle : f(n) = k\}$. If $f$ is total, $graph(f)$ is computable if and only if $f$ is.*

**Proof:** Suppose $f$ is computable. Since it is total, we can compute the characteristic function of $graph(f)$ as follows: given $\langle n, k\rangle$, calculate $f(n)$. If $f(n) = k$, return one. If it is not, then return 0.

Suppose that $graph(f)$ is computable. Given $n$, we wish to compute $f(n)$. Start querying $\chi_{graph(f)}$ for $\langle n, 0\rangle$, $\langle n, 1\rangle$, $\langle n, 2\rangle$, and so on. If it ever returns 1, then return $k$. If it returns 0, continue to the next number. Because $f$ is total, there will be some $k$ such that $\chi_{graph(f)}(\langle n, k\rangle) = 1$, so this process will halt in finite time. Thus $f$ is computable via this process.

Applying this lemma also gives us infinitely many noncomputable sets, one from each function we obtained above.

The majority of computability theory actually concerns the study of noncomputable sets, moreso than the computable ones. Computable sets are relatively predictable, and they only contain information that we know how to determine for ourselves. By contrast, we shall see that noncomputable sets can be incredibly complicated, with some containing more information than others and some containing almost no information at all. Whether or not a set is computable is equivalent to thinking about whether or not the problem of determining membership in that set is **solveable**.

Often, we will define a set which contains information we might be interested in knowing, and then we will prove that the set is not computable. This will mean that there's no way for us to ever actually write down an algorithm to compute this information. The following theorem is a very important example. An **index set** is a set $X$ such that, for all $e$ and $k$, if $\varphi_e = \varphi_k$ then $e \in X$ if and only if $k \in X$. In other words, an index set is a set of partial computable functions coded as a set of natural numbers: as each function has infinitely many codes, then each of them needs to be in the set, otherwise it is ambiguous which functions are in or out.

**Theorem 2.** *(Rice's Theorem) The only computable index sets are $\emptyset$ and $\omega$.*

**Proof:** Suppose not. Then there is a computable index set $X$ which is not $\omega$ or $\emptyset$. As $X$ is neither $\omega$ nor $\emptyset$, there must be $a$ and $b$ such that $a \in X$ and $b \notin X$. Then the following function is computable for a given $n$ and $m$ because $X$ is:

$$f(x) = \begin{cases} a & \text{if } x \notin X \\ b & \text{if } x \in X \end{cases}$$

As $f$ is a total computable function, it has a fixed point $e$ by the recursion theorem. That is, there is $e$ such that $\varphi_e = \varphi_{f(e)}$. Therefore, if $e \in X$, then

$$\varphi_e = \varphi_{f(e)} = \varphi_b$$

and, if $e \notin X$, then

$$\varphi_e = \varphi_{f(e)} = \varphi_a$$

However, both of these contradict the fact that $X$ is an index set, as in the former case $b \notin X$ but $e \in X$, and in the latter $a \in X$ but $e \notin X$.

Essentially, this says that we cannot compute any nontrivial class of computable functions. If we describe some collection of computable functions, then unless it is everything or nothing, there is no algorithm which can determine membership by looking at a code for that function. Even if we come up with some other effective way of coding functions, such as binary .exe files as opposed to indices in a specific universal machine, then Rice's theorem would still apply: we'd be able to computably interpret between the coding we defined and any effective coding of functions, so a computation of the index set in the new coding would translate to one in the old coding.

Notice that this says something interesting about practical computation. Consider the case where we want to designate certain programs, or functions, as malicious and the others as safe. This is an index set: we don't care which code for the function we have because its behavior is what determines malice. But any program attempting to ascertain which other programs are malicious and which are not is doomed to have either false positives or false negatives, i.e. safe programs are classified as malicious or malicious ones are categorized as safe. Because of this, it is impossible to make future-proof antivirus software. For similar reasons, it is very difficult to entirely automate content moderation without input from humans.

# 3 Priority Arguments

A common technique for proving that certain sets exist is a **priority construction.** In a priority construction, we have some over-arching goal we want to achieve by satisfying a list of **requirements.** The goal is usually something quite complicated, while each requirement is easier to satisfy. The goal will be achieved if each requirement is satisfied.

This is best seen with a concrete example. Suppose our goal is to build a c.e. set $A$ which is not computable. Then our requirements will focus on a single, specific Turing machine to ensure that it does not witness that our set is computable. I.e., we'll have the requirements $R_i$ for each $i \in \omega$, which say "if $\varphi_i$ is the characteristic function of a computable set, then $\varphi_i \neq \chi_A$." To meet the goal, we now need to describe how we can meet each of the requirements, then describe how these requirements can be met without conflicting with one another.

Let's first consider how to meet a single $R_i$. If we ever see $\varphi_i(n)\downarrow$, then we can ensure that $\varphi_i \neq \chi_A$ by setting $\chi_A(n) = 1 - \varphi_i(n)$. In terms of building a c.e. set, where we need to list elements, this means listing $n$ if $\varphi_i(n) = 0$ and not listing it otherwise. If $\varphi_i$ isn't total, then we will win by default since it cannot be the characteristic function of a set.

Now we need to describe how to make sure these requirements work in parallel with one another. In general, we can't hope for two $R_i$'s to be satisfied on the same $n$: if $\varphi_i(n) \neq \varphi_j(n)$, then we can't necessarily hope that we'll be able to set $\chi_A$ to avoid both of them. To fix this, we can pick separate $n$'s for each $i$. We can do this in a multitude of ways, but the simplest is to assign $i$ to $i$. That is, put $i$ in $A$ if and only if $\varphi_i(i)\downarrow = 0$. Then we know that $A$ cannot be computable, because if it were with index $e$ for its characteristic function, then $\varphi_e(e)\downarrow$ would lead us to a contradiction.

In this simple example, each requirement could be treated relatively independently. However, in more complicated constructions that we will see, trying to satisfy one requirement will hurt our attempts to satisfy others, and we won't be able to simply assign different numbers to each requirement to fix the issue. This is where the **priority** part of a priority construction comes in: Each requirement will be assigned some level of priority, with only finitely many requirements having higher priority. Then during the construction, higher priority requirements are allowed to **injure** lower priority ones. This will require us to describe how requirements recover from injury to ensure they are satisfied at the end of the construction.

Notice that the above construction was simple to the point where we could easily describe the set in question: $\{i : \varphi_i(i)\downarrow = 0\}$. Usually this will not be the case, but this particular set is an example of a very important class of sets: a

**halting set**, a set of indices based on some type of convergence information. For reasons similar to Rice's theorem, halting sets will broadly be noncomputable, and are perhaps the most famous examples of such. We will discuss these further in the next class.