A non-Turing-recognizable language

1 Overview

Thus far in the course we have seen many examples of decidable languages—they include all of the regular and context-free languages, as well as many other interesting examples. A few examples of undecidable languages came up when we discussed languages *about* context-free grammars, such as this language:

$$\mathrm{EQ}_{\mathrm{CFG}} = \{\langle G, H\rangle \,:\, G \text{ and } H \text{ are context-free grammars with } L(G) = L(H)\}.$$

We did not *prove* that any of these languages are undecidable, however.

The goal for this lecture is to prove that a particular language is not decidable. In fact, we will prove something stronger, which is that a particular language is not Turing-recognizable. The proof will turn out to be remarkably simple, and almost identical to Cantor's proof that $\mathcal{P}(\mathbb{N})$ is uncountable from the very first lecture of the course. This proof uses a technique called *diagonalization*, and we will see why this name is used.

2 Encodings of Turing machines

Before we can describe a non-Turing-recognizable language, we need to briefly discuss *encodings* of Turing machines. For the sake of this particular discussion, and for the rest of this lecture, we will restrict our attention to Turing machines whose input alphabet is $\Sigma = \{0, 1\}$.

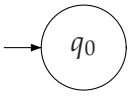
So, consider the collection of all 1DTMs having input alphabet Σ. The Turing machines in this class can have any number of states, any tape alphabet that includes 0, 1, and the blank symbol B, any possible transition function, and so on. Of course, we do not really want to consider two Turing machines to be different if they function in exactly the same way, and differ only in the *names* assigned to their states or tape symbols, so we will assume that the Turing machines in this class have state sets of the form $Q = \{q_0, \ldots, q_m\}$ for some $m \geq 0$ and tape alphabets of the form $\Gamma = \{\tau_0, \ldots, \tau_k\}$ for some $k \geq 2$ (where $\tau_0 = B$, $\tau_1 = 0$, $\tau_2 = 1$, and $\tau_3, \ldots, \tau_k$ are any additional tape symbols that might be used). Having made these simple assumptions, there are certainly a countable number of Turing machines in the class we are considering.

We will assume that we have picked some encoding scheme that lets us take any Turing machine M from our class and encodes it as a binary string $\langle M\rangle \in \Sigma^*$. If you are interested in a particular encoding scheme, you can find one in the text in Section 9.1.2; but the specifics of the scheme are not really important for this lecture. All that we really need to assume right now is that every Turing machine M in our class has an encoding—which must surely be true if our encoding is a sensible one.

It could, by the way, be the case that some binary strings do not encode a Turing machine at all, or that some Turing machines could be encoded in multiple ways, using our chosen encoding scheme. There might be some reason to avoid encoding schemes that have either of these properties for other purposes, but for this lecture we do not care about these things.

Now, if someone hands you a string $w \in \Sigma^*$, there are two possibilities: either w is a valid encoding of a 1DTM M with input alphabet Σ, or it is not. With this in mind, we will define for *every* string $w \in \Sigma^*$ a 1DTM M_w in the following way.

1. If w is a valid encoding of a 1DTM M having input alphabet Σ (i.e., $w = \langle M \rangle$), then we define M_w to be this Turing machine M.
2. If w is not a valid encoding of any 1DTM having input alphabet Σ, then M_w is defined to be the 1DTM that looks like this:



   Note that there is nothing special about this particular Turing machine—we've just chosen something simple and arbitrary to make sure that M_w is a well-defined Turing machine for every possible string $w \in \Sigma^*$.

So now, if we list all of the 1DTMs that we get in this way:

$$M_\varepsilon,\ M_0,\ M_1,\ M_{00},\ M_{01},\ M_{10},\ M_{11},\ M_{000},\ \ldots \tag{1}$$

then we get a list containing *every* 1DTM whose input alphabet is Σ. (It could be that some Turing machines appear many times in the list, but this is fine.)

3 A NON-TURING-RECOGNIZABLE LANGUAGE

Now we are ready to define our first undecidable language. It will, in fact, be non-Turing-recognizable. It is as follows:

$$L_d = \{w \in \Sigma^* \,:\, w \notin L(M_w)\}\,.$$

Here, the Turing machines M_w are as we defined them in the previous section. This means that the language L_d depends upon the choice of an encoding scheme for Turing machines. If you started with a different encoding, you would have a different language—but its interesting properties (in particular, that it is not Turing-recognizable) would not change.

By the way, the subscript "d" stands for "diagonal", which will make more sense later in the lecture.

Theorem 1. *The language L_d is not Turing-recognizable.*

Proof. Assume toward contradiction that L_d is Turing-recognizable. By the definition of Turing recognizability, this means that $L_d = L(M)$ for some 1DTM M having input alphabet Σ. We know that the list (1) defined above includes every such Turing machine, so we may take $w \in \Sigma^*$ to be the lexicographically first string such that $M_w = M$, so that $L_d = L(M_w)$.

We now ask ourselves whether or not this particular string w is or is not contained in L_d. We have

$$w \in L_d \overset{\text{(a)}}{\Leftrightarrow} w \notin L(M_w) \overset{\text{(b)}}{\Leftrightarrow} w \notin L_d,$$

where the equivalence (a) follows from the definition $L_d = \{w \in \Sigma^* \,:\, w \notin L(M_w)\}$ and the equivalence (b) follows from the fact that $L_d = L(M_w)$.

We have proved that $w \in L_d \Leftrightarrow w \notin L_d$, which is a contradiction. It therefore follows that L_d is not Turing-recognizable. □

You should notice, as I stated in the beginning of the lecture, that this proof is almost identical to Cantor's proof that $\mathcal{P}(\mathbb{N})$ is uncountable, which we saw in the first lecture of the course.

You might also notice that the proof has very little to do with Turing machines. If, for instance, you had defined an encoding scheme for DFAs instead of Turing machines, so that (1) was instead a list of all possible DFAs, then the above proof would succeed in constructing a non-regular language rather than a non-Turing-recognizable one. (That would be less interesting, though, because we have other ways to prove languages are not regular.)

4 Why it is called diagonalization

The proof technique that the above proof represents is called *diagonalization*. Let us now discuss where this name comes from.

Imagine first drawing an infinitely large table whose rows are labelled by the 1DTMs

$$M_\varepsilon,\ M_0,\ M_1,\ M_{00},\ M_{01},\ M_{10},\ M_{11},\ M_{000},\ \ldots$$

and whose columns are labelled by the strings in Σ^* (in lexicographic order). In the entry indexed by each pair (M_w, x), place a 1 if M_w accepts x, and a 0 otherwise. In other words:

$$\text{Entry } (M_w, x) \text{ is } \begin{cases} 1 & \text{if } x \in L(M_w) \\ 0 & \text{if } x \notin L(M_w). \end{cases}$$

For instance, the upper left-hand corner of this table might look like this:

	ε	0	1	00	01	10	11	$\cdots$
M_ε	0	0	0	0	0	0	0	$\cdots$
M_0	1	0	1	0	1	0	1	$\cdots$
M_1	1	1	1	0	1	1	1	$\cdots$
M_{00}	0	0	0	0	1	0	0	$\cdots$
M_{01}	1	1	0	0	1	0	1	$\cdots$
M_{10}	1	0	0	1	0	0	0	$\cdots$
M_{11}	0	0	1	1	0	1	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Of course, the entries in this hypothetical picture are meaningless—I did not pick an encoding scheme for Turing machines and test them on different inputs to figure out the entries in the table. But, if you really did choose an encoding scheme as discussed in Section 2, there would be well-defined entries in the table.

Now, consider a particular row in the table—for instance, the one indexed by the Turing machine M_{10}. You get an infinite sequence of 0's and 1's that completely determines the language accepted by M_{10}. Using the hypothetical entries in the above picture, we would have that M_{10} accepts ε, 00, and maybe many other strings whose columns are not pictured. This sequence of 0's and 1's is called the *characteristic sequence* of the language $L(M_{10})$.

What we are after is a language whose characteristic sequence is different from *all* of the rows in the table. If we have such a language, we know it cannot be Turing-recognizable, because the rows in the table give us the characteristic sequences for every possible Turing-recognizable language (over the alphabet Σ). This is because every possible 1DTM with input alphabet Σ appears in the list (1), and therefore indexes at least one row of the table.

So, how can we build such a sequence? One answer is to look at the *diagonal* entries of the table:

	ε	0	1	00	01	10	11	$\cdots$
M_ε	**0**	0	0	0	0	0	0	$\cdots$
M_0	1	**0**	1	0	1	0	1	$\cdots$
M_1	1	1	**1**	0	1	1	1	$\cdots$
M_{00}	0	0	0	**0**	1	0	0	$\cdots$
M_{01}	1	1	0	0	**1**	0	1	$\cdots$
M_{10}	1	0	0	1	0	**0**	0	$\cdots$
M_{11}	0	0	1	1	0	1	**1**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

If we *flip* each of these entries, we will obtain an infinite sequence of bits that differs from every row in the table. In the example table above, we obtain the sequence

$$1\,1\,0\,1\,0\,1\,0\,\cdots$$

when we flip the diagonal entries. The reason why this sequence differs from every row in the table is as follows:

1. The sequence is different from the *first* row, because the two sequences must differ in the *first* position (because we chose the first bit of our new sequence to be the negation of the first bit of the first row).
2. The sequence is different from the *second* row, because the two sequences must differ in the *second* position.
3. Similar reasoning for every other row. The new sequence differs from row k in the k-th position.

So, we have obtained the characteristic sequence of a language that is not Turing-recognizable, because it differs from every row in the table. If we think for a moment, then we realize that the

language given by this sequence, which we obtained by flipping the diagonal entries of the table, is more succinctly described as follows:

$$\{w \in \Sigma^* \,:\, w \notin L(M_w)\}\,.$$

This is precisely the language L_d.

Hopefully you now see why the technique is called diagonalization. It is a very simple and general idea—if you have an infinite table like the one above, it allows you to construct a sequence that differs from every row in the table. The entries don't necessarily need to be bits, and the rows and columns can be indexed by anything we want. This turns out to be a very important proof technique in theoretical computer science and mathematics that can be applied in several interesting settings.