CSCI3390-Lecture 6: An Undecidable Problem

September 21, 2018

1 Summary

- The language L_{TM} recognized by the universal Turing machine is not decidable. Thus there is no algorithm that determines, yes or no, whether a given Turing machine accepts a given input string. This is proved by a variant of the Cantor diagonal argument, originally used to show that there are uncountable infinite sets.
- As a consequence, we are able to show that a large number of basic questions about the behavior of Turing machines, and therefore about the behavior of computer programs, are also undecidable. The main tool for proving this is to *reduce* the problem that we want to show is undecidable to a problem we already know to be undecidable. We'll discuss reductions in the next lecture.

2 The main result: L_{TM} is not decidable.

We showed that the language

$$L_{TM} = \{< \mathcal{M}, w >: \mathcal{M} \text{ accepts } w\}$$

is Turing-recognizable. When you strip away the formal description, L_{TM} is the decision problem:

Input: A Turing machine $\mathcal{M}$ and a string $w \in \{0,1\}^*$,

Output: Yes, if $\mathcal{M}$ accepts w, no if $\mathcal{M}$ does not accept w.

The fact that L_{TM} is Turing-recognizable translates to: there is an algorithm for this problem that always answers 'Yes' when that is the correct output, and never answers incorrectly. This is simply the procedure of simulating $\mathcal{M}$ on the string w, which you carried out in the first assignment.

However, this algorithm does not completely solve the decision problem, because it may fail to answer, by running forever. We want to know if there is an algorithm that always answers 'Yes' when $\mathcal{M}$ accepts w, and always answers 'No' when $\mathcal{M}$ does not accept w.

We will show that no such algorithm exists. That is,

Theorem 1 *L_{TM} is undecidable.*

Before we proceed to the proof, let's think about what a proof that L_{TM} is *decidable* would look like. Suppose we could prove something along the lines of

3 Cantor's Diagonal Argument

You could skip right ahead to the proof of Theorem 1. This section is partly historical background, partly motivation for the argument.

Below is a *listing* of the elements of several infinite sets: the natural numbers, the integers, ordered pairs of integers, and strings of bits.

$$\mathbf{N}\text{: } 0, 1, 2, 3, \ldots$$

$$\mathbf{Z}\text{: } 0, 1, -1, 2, -2, 3, -3 \ldots$$

$$\mathbf{Z} \times \mathbf{Z}\text{: } (0,0), (0,1), (0,-1), (1,0), (-1,0), (0,2), (0,-2), (1,1), (1,-1), \ldots$$

$$\{0,1\}^* : \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$$

(In case it's not clear, the method for listing the elements of $\mathbf{Z} \times \mathbf{Z}$ is to list all those pairs such that the sum of the absolute values of the components is 0, then those whose sum is 1, then 2, *etc.* There are only finitely many pairs in each of these groups.)

An infinite set X whose elements can be listed this way is said to be a *countable* set. Another, more formal way to say this is that there is a bijective function $f : \mathbf{N} \to X$.

Are there any *uncountable* sets? The answer is yes. The mathematician Georg Cantor, who invented the modern theory of sets, gave several proofs of this fact.

The one we give here was published in 1891. If X is a set, the $\mathcal{P}(X)$ denotes the *power set* of X, the set of all subsets of X. For example,

$$\mathcal{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Cantor's result is

Theorem 2 *Let X be a set. There is no onto function $f : X \to \mathcal{P}(X)$.*

If X is finite, this theorem is trivial, because $\mathcal{P}(X)$ contains more elements than X. If X is an infinite set, this gives meaning to one infinite set having 'more elements than' another infinite set. Observe that if Y is a countable set, then we can list all its elements

$$y_0, y_1, y_2, \ldots,$$

and thus there is an onto function $f : \mathcal{N} \to Y$ defined by $f(j) = y_j$. So, in particular, the above theorem implies that $\mathcal{P}(\mathbf{N})$ is an uncountable set.

Proof of Theorem 2. Suppose to the contrary that such an onto function $f : X \to \mathcal{P}(X)$ exists. Define

$$Z = \{z \in X : z \notin f(z)\}.$$

Z is a subset of X, so since f is onto, there is some $y \in X$ such that $f(y) = Z$. Is y an element of $f(y)$? If $y \in f(y)$, then by the definition of Z, $y \notin Z = f(y)$, and if $y \notin f(y)$, then again, by the definition of Z, $y \in f(y)$. So we get the paradoxical conclusion that y is an element of $f(y)$ if and only if it isn't. But there's really no paradox, just a contradiction. Our original assumption that f is onto must be false, so no such onto function exists.

This theorem tells us something about Turing machines and decidability as well. Every Turing machine can be encoded by a bit string, so the set of Turing machines is countable. Consider the function g that maps a Turing machine $\mathcal{M}$ with input alphabet $\{0,1\}$ to the language $L \subseteq \{0,1\}^*$ that it recognizes. Since $\mathcal{P}(\{0,1\}^*)$ is uncountable, g cannot be an onto function, so there must be some languages (in fact, almost all languages) that are not Turing-recognizable, and therefore not decidable. But this observation is not terribly informative, since it does not provide an example of an undecidable problem.

Cantor's argument was used by Bertrand Russell in 1901 to formulate *Russell's Paradox*: Let Z be the set of all sets that are not elements of themselves. Then $Z \in Z$ if and only if $Z \notin Z$. The upshot is that no system of logic or set

theory should allow one to define this set Z—the problem was that the logician Frege had just published a system that allows one to do precisely that. So this was a big deal in the history of mathematical logic.

Russell explained the paradox this way: Imagine a group of men that includes a barber. The barber shaves precisely the men in the group who do not shave themselves. So who shaves the barber? If he shaves himself, then by this criterion, he doesn't shave himself, and if he doesn't, he does. (The resolution of the 'paradox' here is that no such group of men, with such a barber, can possibly exist.)

4 Proof of Theorem 1.

Suppose L_{TM} is decidable: that is, we have a Turing machine $\mathcal{V}$ that decides whether a given Turing machine accepts a given input string. In particular, we can decide if a given Turing machine $\mathcal{M}$ accepts its *own encoding* $< \mathcal{M} >$. In other words, we would have that the language

$$\{< \mathcal{M} >: \mathcal{M} \text{ accepts } < \mathcal{M} >\}$$

is decidable. Let $\mathcal{V}_2$ be a TM that decides it. Now we tweak $\mathcal{V}_2$ simply by interchanging its 'accept' and 'reject' states, and get another TM $\mathcal{V}_3$. Let's summarize what $\mathcal{V}_3$ does on a given input u:

- If u is not the encoding of any Turing machine, the $\mathcal{V}_2$ rejects u, so $\mathcal{V}_3$ accepts u.
- If $u =< \mathcal{M} >$ for some TM $\mathcal{M}$, and $\mathcal{M}$ accepts u, then $\mathcal{V}_2$ accepts u, so $\mathcal{V}_3$ rejects u.
- If $u =< \mathcal{M} >$ for some TM $\mathcal{M}$, and $\mathcal{M}$ does not accept u, then $\mathcal{V}_2$ rejects u, so $\mathcal{V}_3$ accepts u.

We now ask 'who shaves the barber?' Does $\mathcal{V}_3$ accept its own encoding $u =< \mathcal{V}_3 >$? The second rule above says that if $\mathcal{V}_3$ accepts u, then it rejects u, and if it doesn't accept u, then it accepts u. So we get the same sort of paradox.

This means that $\mathcal{V}_3$ cannot exist, but then neither can $\mathcal{V}_2$, nor $\mathcal{V}$. So L_{TM} is undecidable.

5 A non-Turing-recognizable language

We showed earlier that a language is decidable if and only if both it and its complement are Turing recognizable. Since L_{TM} is Turing recognizable but not decidable, its complement is not Turing-recognizable. In other words, the following problem is not Turing-recognizable:

> *Input:* A Turing machine $\mathcal{M}$ and a string $w \in \{0,1\}^*$,
>
> *Output:* Yes, if $\mathcal{M}$ does not accept w, no if $\mathcal{M}$ accepts w.

That is, there is not even an algorithm for this question that always answers 'Yes' when that is the correct answer.

6 Perspective

Theorem 1, and the related problems we discuss below, although stated about Turing machines, are true in any computational model that is equivalent in power to Turing machines. This includes any conventional programming language. So, for example, there is no Java program to determine if a given Turing machine accepts a given input, and there is no Java program to determine if a given Java function

```
boolean f(int n)
```

returns `true` on a given integer argument. Similarly, the claim below that there is no algorithm to determine if two Turing machines have the same behavior (accept the same strings) implies that there is no algorithm to determine if two Python functions behave identically.

7 A collection of undecidable problems about Turing machines

All of the problems below are undecidable. We will see how to deduce this from the undecidability of L_{TM} in the next lecture.

7.1 Halting Problem, version 1

Input: A Turing machine $\mathcal{M}$ and a string $w \in \{0,1\}^*$.
Output: Yes if $\mathcal{M}$ eventually halts when started on input w, No otherwise.

7.2 Halting Problem, version 2

Input: A Turing machine $\mathcal{M}$.
Output: Yes if $\mathcal{M}$ eventually halts regardless of the input string (*i.e.,* $\mathcal{M}$ is free of infinite loops), No otherwise (*i.e.,* if $\mathcal{M}$ runs forever on some input).

7.3 Nonemptiness

Input: A Turing machine $\mathcal{M}$.
Output: Yes if there is some $w \in \{0,1\}^*$ accepted by $\mathcal{M}$, no otherwise.

7.4 Finiteness.

Input: A Turing machine $\mathcal{M}$.
Output: Yes if the set of strings accepted by $\mathcal{M}$ is finite, No if $\mathcal{M}$ accepts infinitely many strings.

7.5 Equivalence

Input: Two Turing machines $\mathcal{M}, \mathcal{N}$.
Output: Yes $\mathcal{M}$ and $\mathcal{N}$ accept exactly the same strings, no otherwise.

7.6 Minimality

Input: A Turing machine $\mathcal{M}$.
Output: Yes if $\mathcal{M}$ is the smallest Turing machine that recognizes the set of strings it accepts, No if there is a smaller machine that recognizes the same language.

Here you have to say what exactly you mean by the size of a Turing machine. One way to do this is to fix an encoding scheme, and use the length of the encoding of $\mathcal{M}$ as the size of $\mathcal{M}$.