Lecture 18: Undecidability III

built on 2021/03/04 at 09:41:01

1 Recap: What Is Hard?

What is undecidable? Apparently, everything!

Our starting point in this journey was HALT. A couple of lectures ago, we showed that HALT—whether a TM M would halt on input x—is undecidable.

From this starting point, we used reduction techniques to prove that many other problems are also undecidable. For example, we have seen that both of the languages below are undecidable:

- $\mathsf{ACCEPT} = \{\langle M, x\rangle \mid M \text{ is a TM which accepts } x\}$.
- $\mathsf{ALL} = \{\langle M\rangle \mid M \text{ accepts all strings}\}$.
- $\mathsf{EMPTY} = \{\langle M\rangle \mid M \text{ accepts nothing}\}$

2 Fancy Reduction

Let's now prove the following theorem:

Theorem 2.1 *EMPTY $\leqslant_T$ HALT:*

Proof: Suppose there is a TM M_{HALT} that decides HALT. We give a description of a TM for deciding EMPTY. Given $\langle M\rangle$:

1. Write down the description $\langle M'\rangle$, where M' does this: For $t = 0, 2, 3, \dots, \infty$, run M on each string of length at most t for t steps. If any of them terminates and accepts, then terminate and accept.
2. Call M_{HALT} on $\langle M', \varepsilon\rangle$. Accept if it rejects; and reject if it accepts.

■

Notice that the Turing machine M' is, in a way, using the Dovetailing trick we saw earlier. This is used because we don't know how long a program will run on a particular input—and we want to run this program on all possible input. The belief is that, if the given TM accepts nothing, M' should keep on going forever. But if TM turns out to accept something, M' will *eventually* halt.

At this point, I should probably give you a rule of thumb of recognizing when a language is undecidable:

Deciding if an arbitrary TM (or program) has an "interesting" property is usually undecidable.

There is, in fact, a rather general theorem for that. Rice's theorem, which appears as an extra-credit problem on your assignment, makes such a statement.

2.1 Optional: A Simple Yet Undecidable Problem

In so far, everything that we showed to be undecidable was concerned with Turing machines or automata. We'll briefly look at a problem that doesn't involve automata but is undecidable, though it appears simple at first glance.

Consider the following puzzle. We have a collection of dominoes of various types. Each is marked with two strings, one at the top and one at the bottom, so a domino looks like

$$\left[\frac{\mathsf{ab}}{\mathsf{b}}\right]$$

A collection of dominoes looks like a set of various-typed dominoes

$$C = \left\{\left[\frac{\mathsf{abc}}{\mathsf{c}}\right], \left[\frac{\mathsf{b}}{\mathsf{ca}}\right], \left[\frac{\mathsf{ca}}{\mathsf{a}}\right], \left[\frac{\mathsf{a}}{\mathsf{ab}}\right]\right\}$$

We have an infinite supply of each type of domino.

The *Post correspondence problem* (PCP) is to make a list of these dominoes (allowing repetitions) such that the string we obtain from reading the top line is the same as the string we obtain from reading the bottom line. For example, using the collection C above, we can find a match as follows:

$$\left[\frac{\mathsf{a}}{\mathsf{ab}}\right]\left[\frac{\mathsf{b}}{\mathsf{ca}}\right]\left[\frac{\mathsf{ca}}{\mathsf{a}}\right]\left[\frac{\mathsf{a}}{\mathsf{ab}}\right]\left[\frac{\mathsf{abc}}{\mathsf{c}}\right].$$

The language PCP is defined to be

$$\mathsf{PCP} = \{\langle C\rangle \mid C \text{ is a collection of dominoes that have an arrangement with matching top and bottom}\}$$

Theorem 2.2 *PCP is undecidable.*

Proving this theorem is beyond the scope of this course, but here is a general outline: Given a TM M, make a set of dominoes that only have matches if the execution of M ends in the accepting state, hence showing $\mathsf{ACCEPT} \leqslant_T \mathsf{PCP}$. More specifically: the string along the top and bottom will be a computation history of the Turing machine's computation. This means it will list a string describing the initial state, followed by a string describing the next state, and so on until it ends with a string describing an accepting state. The top portion starts out lagging behind the bottom by one execution step, and they'll only come to an agreement at the very end.

3 Recursion Theorem

Having seen the fixed-point theorem for λ-calculus, the following theorem shouldn't be much of a surprise!

Theorem 3.1 *Let T be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a Turing machine R that computes a function $r : \Sigma^* \to \Sigma^*$ such that for every w,*

$$r(w) = t(\langle R\rangle, w)$$

We have seen an application of this theorem, though stated in λ-calucus lingo. Let $t(F, n)$ be the function

```
if (n==0) return 1
else return n*(F(pred(n)))
```

This theorem says there is a function r (you know it as fact) that computes $t(r, w)$.

How do you prove such a thing? Let's go back to λ-calculus.

Proof: Let t be represented as a λ-term $\bar{t}$. This means $\bar{t}\,M\,N$ would compute $t(M, N)$. We're going to construct R by appealing to the fixed-point theorem: Every term F has a fixed-point N, where $FN =_\beta N$.

Recall that by setting $A = \lambda xy.\, y(xxy)$ and $\Theta = AA$, we have that $N = \Theta F$ is a fixed point for F. Therefore, we make R to be $\Theta\bar{t}$.

To show that this has the right property, consider that for any λ-term P, we have

$$\begin{aligned} RP &= (\Theta\bar{t})P \\ &\to \bar{t}(\Theta\bar{t})P \\ &= \bar{t}RP \end{aligned}$$

■

So What? Firstly, this proves that recursion doesn't add any intrinsic ability that a TM/λ-calculus didn't have without it.

Secondly, we have a slightly nicer proof for ACCEPT (and a few other things):

Theorem 3.2 *ACCEPT is undecidable.*

Proof: (An Alternative Proof.) Suppose for a contradiction that there is a machine M that decides ACCEPT. Consider the following machine D:

Input: $w \in \Sigma^*$

- Obtain, via recursion theorem, its own description $\langle D \rangle$.
- Run $M(\langle D, w \rangle)$
- Do the opposite of what M predicts.

Running B on w would do the opposite of what M predicts. Therefore, we can't decide ACCEPT. ■

4 Optional: Self-Reference

Consider the following task, which appears unsurmountable at first:

> Write a program that prints out its own source code.

To be more concrete, you want to write `Self.java` such that after compilation (`javac Self.java`), the execution (`java Self`), without the ability to read `Self.java`, will print out the source code of `Self.java`. This property is known as *self-reproducing*.

Obstacle #1: A direct challenge to writing such a program is the mere fact that if you edit `Self.java`, the changes have to be reflected in `Self.java` because it has to print out its own source code when executed (without reading `Self.java` at run time).

Warm-Up: In general, it is easy to write a program that prints out the source code of a program (other than itself). Indeed, there is a program Q where

Input: w
Output: source code of a program that outputs w

An Implication: This means, if we have `Moo.java`, the output of $Q((newFile(\text{"}Moo.java\text{"})).read())$ is a valid source program that when run, will print out w, which is the source code of `Moo.java`. Note: the reading of `Moo.java` was done when Q constructs the program, not when the generated program is run.

Obstacle #2: Since we don't seem to have an idea how to implement `Self.java` directly, we'll attempt it split it into two components: A and B.

- The goal of component A is to print out the source code for B. Hence, to generate A, we'll use Q to print out the description of B. This part is really simple, but it requires that A must be run after the source code for B is all set.
- The goal of component B is to print out the source code for A. There is a slight problem here. B can't be defined to be the output of $Q(A)$ because that would be circular: we needed the source of B to generate A. The crucial problem is, *how can B obtain the source of B?* Here's what we can do: we make B take in the source of a program, say, m. (This will be eventually given by A, which has the source of B.) Then, B is a program that does the following:
  1. Compute $Q(m)$—a program that when run, outputs m
  2. Put the result of $Q(m)$ next to m, making it a complete program

3. Output this description

Amazingly, the following Python code does it (original code credit: http://blog.amir.rachum.com/blog/2012/08/05/self-printing-programs-in-python/, but adapted to Python 3 for this course):

```
s = r"print('s = r\"{}\"'.format(s)+ '\n'+ s)"
print('s = r\"{}\"'.format(s)+ '\n'+ s)
```

The first line is practically A: it is a program that outputs a B when run. The output is stored in s. The second line is B, which has both steps: the step that generates $Q(m)$ and the step that generates prints m itself. Together, this forms a self-reproducing program.

There is nothing special about Python. We can do the same in λ-calculus, our primitive computational model. Let's follow the recipe outlined above.

$$
\begin{aligned}
\mathsf{Q} &:= \lambda w.\lambda x.\, w \\
\mathsf{A} &:= \mathsf{QB} \\
&\to (\lambda w.\lambda x.\, w)\mathsf{B} \\
&\to \lambda x.\, \mathsf{B} \\
\mathsf{B} &:= \lambda m \lambda y.\, m(\mathsf{Q}my) \\
&\to^* \lambda my.\, mm
\end{aligned}
$$

We'll define SELF that runs A to print out the "source" of B and then run B on its own code. The specification for SELF is

$$\mathsf{SELF}\, M \to^* \mathsf{SELF}.$$

We can satisfy it by using

$$\mathsf{SELF} := \lambda z.\, \mathsf{B}(\mathsf{A}z),$$

which can be simplified as follows:

$$
\begin{aligned}
\lambda z.\, \mathsf{B}(\mathsf{A}z) &\to^* \lambda z.\, (\lambda my.\, mm)\mathsf{B} \\
&\to^* \lambda z.\, \mathsf{BB}.
\end{aligned}
$$

It is not hard to see that for any λ-term M,

$$\mathsf{SELF}\, M \to \mathsf{BB} \to (\lambda my.\, mm)\mathsf{B} \to \lambda y.\, \mathsf{BB} \equiv_\alpha \mathsf{SELF}.$$

5 Concluding Remarks

Many interesting, real-world problems turn out to be undecidable. The list goes on and on. We'll conclude with a few interesting ones:

Mortal Matrices: The input are *two* 21×21 matrices called A and B. The question then is, whether we can multiply A and B together (multiple times in any order) so that the final result is the 0 matrix.

It was shown in 2007 that this too is undecidable.

Hilbert's 10th Problem: The input is a multivariate polynomial with integer coefficients.

- The question "Does it have an integer root?" is undecidable.
- However, the quetion "Does it have a real root" is decidable.
- More interestingly, the question "Does it have a rational root" remains open: we don't know if it is decidable or undecidable.

  Exercise: Show that $\mathsf{EQUIV}_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1$ and M_2 are TMs s.t. $L(M) = L(M')\}$ is undecidable. (*Hint:* show that $\mathsf{ALL} \leqslant_T \mathsf{EQUIV}$.)