Lecture 15: Undecidability I

built on 2021/02/23 at 09:18:07

1 Decidability Is Pretty Robust

Although our starting point was the Turing machine, we have mentioned before that decidability power is pretty much the same for all (reasonable) variants of TMs. For example, all of the following have the same power:

- a single one-sided and two-sided infinite tape.
- the ability to stay put in addition to moving left or right
- restriction to only the binary alphabet (as opposed to an arbitrary finite Σ)
- one tape vs. a finite number of tapes

AND this is the same as Python, Java, C, Haskell, etc. etc. This also includes our favorite λ-calculus. Keep in mind, **they do differ in terms of efficiency**—in many cases, I don't mean just constant differences but some significant differences (e.g., polynomial differences in the running time, or exponential differences compared to, say, a quantum computer).

1.1 Church-Turing Thesis

All previously mentioned models have been proved to have the same decidability power as a TM. Amazingly, there have not been candidates of potentially implementable and natural computation models that are strictly more powerful than the TM. Back in 1936, Church and Turing put forward a bold statement, known today as the *Church-Turing Thesis*:

> **Church-Turing Thesis**: Any natural/reasonable notion of computation can be simulated by a Turing machine.

When describing Turing machines, we could explicitly describe all states and transitions (low-level) or like what we have been doing (medium-level), describe in plain English how that TM will operate, leaving the "obvious" detail out. By alluding to the Church-Turing thesis, we can give a super high level description: once we describe an algorithm (in e.g., pseudocode), there must exist a TM that executes that algorithm. In this class, we'll strive to be somewhere between medium and super high.

1.2 Encoding of Objects As Strings

In so far, we describe our machines as formal mathematical objects. It is possible to represent such a machine as a string in our chosen alphabet, similar to how we are able to serialize (or JSONify) an object in modern programming languages.

For an alphabet Σ (of your choosing), let $\langle M\rangle_{\Sigma*}$, or $\langle M\rangle$ for short, denote the encoding of an object M as a string in Σ^*. Here are some examples:

- Say you have a Turing machine M, $\langle M\rangle \in \Sigma^*$ is the encoding of the TM M.
- Say you have an NFA machine N, $\langle N\rangle \in \Sigma^*$ is the encoding of the NFA N.
- Say you have a pair of Turing machines (M_1, M_2), $\langle M1, M2\rangle \in \Sigma^*$ is the encoding of (M_1, M_2).
- Say you have a pair (M, x) representing a Turing machine M and a string x, $\langle M, x\rangle$ is the encoding of (M, x).

1.3 DFAs Are Easy

Using such encoding, we can define:

$$\begin{aligned}
\mathsf{ACCEPT}_{\mathsf{DFA}} &= \{\langle D, x\rangle \mid D \text{ is a DFA that accepts } x\} \\
\mathsf{SELF\text{-}EAT}_{\mathsf{DFA}} &= \{\langle D\rangle \mid D \text{ is a DFA that accepts } \langle D\rangle\} \\
\mathsf{EMPTY}_{\mathsf{DFA}} &= \{\langle D\rangle \mid D \text{ is a DFA that accepts nothing, i.e. } L(D) = \varnothing\} \\
\mathsf{EQUIV}_{\mathsf{DFA}} &= \{\langle D_1, D_2\rangle \mid D_1, D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}
\end{aligned}$$

You should convince yourself that all these languages are decidable. For ACCEPT and SELF-EAT, we can simply simulate running the DFA. It will terminate for sure and either accepts or rejects the input. How do we decide EMPTY? This is simply the graph problem: can you reach any of the accepting states? We have an algorithm for that, so it is decidable.

The most interesting one is EQUIV. There are many ways to go about this. Here's one that not only is interesting but will demonstrate a technique that will be useful later on:

Lemma 1.1 *$\mathsf{EQUIV}_{\mathsf{DFA}}$ is decidable.*

Proof: The input to $\mathsf{EQUIV}_{\mathsf{DFA}}$ is a pair (D_1, D_2). Our task is to decide whether D_1 and D_2 recognize the same language. Let's build a Turing machine M that performs the following steps:

1. Create an NFA $D' = (L(D_1) \cap L(D_2)) \cup (L(D_2) \cap L(D_1))$. (This is simply the symmetric difference $L(D_1)\Delta L(D_2)$.)
2. Ask the decider for $\mathsf{EMPTY}_{\mathsf{DFA}}$. If D' is accepted, then say YES; otherwise say NO.

■

1.4 Reductions

The proof above illustrates an important, powerful technique in computer science: using a solution to one problem as a subroutine to solve another. In technical lingo, we say language A reduces to language B if it is possible to decide A using an algorithm/TM for deciding B as a subroutine (obviously, we'll have to do some thinking to convert it back and forth). Notationally, we write

$$A \leqslant_T B$$

This literally means A is **no harder than** B. This is true because you can use B to solve A.

This also means:

Suppose $A \leqslant_T B$. If B is decidable, then so is A.

Case in point: $\mathsf{EQUIV}_{\mathsf{DFA}} \leqslant \mathsf{EMPTY}_{\mathsf{DFA}}$ and $\mathsf{EMPTY}_{\mathsf{DFA}}$ is decidable. So $\mathsf{EQUIV}_{\mathsf{DFA}}$ must well be decidable.

The contrapositive of the following statement is:

Suppose $A \leqslant_T B$. If A is undecidable, then so is B.

2 Undecidability

A few lectures ago, we said a language L is *decidable* if there is a TM M that halts on every $x \in \Sigma^*$ and when it halts, correctly answers whether an $x \in \Sigma^*$ belongs to L. The idea of undecidability is the opposite of decidability: we wish to show that no Turing machine has such capabilities.

2.1 Evidence Of Undecidability

Consider the binary alphabet $\Sigma = \{0, 1\}$. Is every language on Σ decidable? Probably not. Here's an argument:

> There are finitely many TMs: This is because every TM can be encoded as a string in Σ^* and $|\Sigma^*| = |\mathbb{N}|$.
>
> But the set of all languages on Σ^* (i.e. 2^{Σ^*}) is strictly larger than $\mathbb{N}$.

To paraphrase the above argument, the set of all decidable languages is countable (well countably infinite). But the set of all languages is uncountable.

Essentially, almost all of the languages out there are undecidable. And it's not just the weird, contrived languages—many natural languages (ones that we wish to solve) are undecidable.

Case in point: Checking if two programs P_1 and P_2 do the same thing (efficiency aside). This would be really cool to have for verifying specification, program optimization, etc. etc. But as we will show, this is not decidable.

2.2 A Simple Undecidable Language

Consider the following simple problem, known as the *Halting problem*:

> Given a program P, is it going to terminate? More formally, let

$$\mathsf{HALT} = \{\langle M, x\rangle \mid M \text{ is a TM that terminates on } x\}.$$

In 1936, Turing showed that the Halting problem is undecidable.

Theorem 2.1 *The language*

$$\mathit{HALT} = \{\langle M, x\rangle \mid M \textit{ is a TM that terminates on } x\}.$$

is undecidable.

Proof Idea: Suppose for a contradiction that HALT is decidable, so there is a TM M_{HALT}. We'll come up with another TM that confuses the heck out of M_{HALT}.

Our supposed M_{HALT}:

- Input: an encoding of a TM
- Output: whether the input TM is decidable.

We'll build a new TM (a "CONFUSER"). Given as input $\langle M\rangle$:

1. CONFUSER executes $M_{\mathsf{HALT}}(\langle M, \langle M\rangle\rangle)$
2. If this call accepts, it enters an infinite loop. If this call rejects, it halts.

In other words, CONFUSER loops if $M(\langle M\rangle)$ halts—and halts if $M(\langle M\rangle)$ loops. **Our contradiction:** Does CONFUSER($\langle$CONFUSER$\rangle$) halt or loop? If M_{HALT} predicted it'd loop, it halts; and if M_{HALT} predicted it'd halt, it loops. Absurd!

If you remember Cantor's diagonalization argument, this is the same thing: Remember that the set of all TM's is countable, so there is a function $f : \mathbb{N} \to \{\text{TMs}\}$ that lists them all. Hence, we have our familiar table. The rows list all possible Turing machines; the columns, the encodings of such Turing machines. The entry i, j (row i, column j) shows M_{HALT}'s prediction for whether M_i is going to halt when given as input $\langle M_j\rangle$.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\langle M_5\rangle$	$\langle M_6\rangle$
M_1	y	n	n	y	y	y
M_2	n	y	y	y	y	y
M_3	n	y	n	n	n	y
M_4	n	y	n	y	n	y
$\vdots$						
CONFUSER	n	n	y	n	...	

We obtain a contradiction in the same way as in our previous Cantor's diagonalization proof: CONFUSER is a TM, so it must be among the M's. Say it is M_k, but then this is absurd because CONFUSER does the exact opposite of what M_{HALT} predicts about $\langle M_k, \langle M_k\rangle\rangle$.

This proves that given some code, it is impossible to determine if it terminates (in general). In fact, assuming the Church-Turing thesis, this is unsolvable by any algorithm, any mechanism, any human being, anything in this world and any (physical) world we can imagine.