Lecture 12: Undecidable Languages

Ryan Bernstein

1 Introductory Remarks

- Assignment 3 is available online, and is due a week from today (05/19)

1.1 Recapitulation

On Tuesday, we built a lot of Turing machines. These Turing machines decided languages based on encodings of other machines or objects. They included languages like:

- $\text{A}_{\text{DFA}} = \{\langle D, w\rangle \mid D \text{ is a DFA that accepts } w\}$
- $\text{E}_{\text{DFA}} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \emptyset\}$
- $\text{ALL}_{\text{DFA}} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \Sigma^*\}$
- $\text{A}_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a context-free grammar capable of generating } w\}$

The key point here was that, since Turing machines are algorithms, we could use any algorithm that we've seen before as a step when constructing a machine M. This includes things like conversion to Chomsky normal form or between machines equivalent in power, but it also includes the Turing machines that we're constructing themselves. Once we've constructed as a machine, we can use that machine as a subroutine to solve larger problems, just as we can with programs, functions, and libraries that we write while programming.

By constructing Turing machines that decided all of these languages, we showed that they were Turing-decidable. Today, we'll be looking at languages that are *not* Turing-decidable. To do this, we'll need to start by showing that such languages actually exist.

2 Countable and Uncountable Infinity

First, a return to CS 250. In CS 250, we discussed the idea of infinite sets. We broke infinite sets into two categories: countably infinite and uncountably infinite. Countably infinite sets are those that can be put into a one-to-one mapping with the natural numbers or some subset thereof. Obviously, the natural numbers themselves fall into this category. We can also see that $\mathbb{Z}$ is countably infinite by generating a function $f : \mathbb{Z} \to N$:

$$f(x) = \begin{cases} 2x - 1 & x > 0 \\ -2x & x \leq 0 \end{cases}$$

This generates a mapping like the following:

$$\begin{array}{c|ccccccc} x & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\ f(x) & 6 & 4 & 2 & 0 & 1 & 3 & 5 \end{array}$$

What's the point of creating such a mapping? Countably infinite languages are so named because we can enumerate all of their members by counting. Once we've mapped the set to the natural numbers, we can reach every member of the set simply by starting at zero and counting up. While this enumeration may never finish, we will *eventually* hit any arbitrary element.

Uncountably infinite sets, then, are sets that *cannot* be mapped to the natural numbers. The canonical example of this is the set of all infinite-length binary strings. We call this set uncountably infinite because of an argument called Cantor's Diagonalization. This assumes that we have some enumeration of all infinite-length binary strings, like so:

$$\begin{array}{rccccccc} s_1 = & 0 & 1 & 0 & 1 & 0 & 0 & ... \\ s_2 = & 1 & 1 & 0 & 1 & 0 & 0 & ... \\ s_3 = & 0 & 1 & 1 & 0 & 1 & 1 & ... \\ ... & & & & & & & \\ s_n = & 1 & 1 & 1 & 0 & 1 & 0 & ... \\ ... & & & & & & & \end{array}$$

We can generate another binary string s_d by inverting the first character of s_1, the second character of s_2, and so on.

$$\begin{array}{rccccccc} s_1 = & \underline{0} & 1 & 0 & 1 & 0 & 0 & ... \\ s_2 = & 1 & \underline{1} & 0 & 1 & 0 & 0 & ... \\ s_3 = & 0 & 1 & \underline{1} & 0 & 1 & 1 & ... \\ ... & & & & & & & \\ s_n = & 1 & 1 & 1 & 0 & 1 & 0 & ... \\ ... & & & & & & & \end{array}$$

$$s_d = 100...$$

Because s_d differs from *every string in this enumeration* at the point of diagonalization, we know that s_d will not have already been a member of this enumeration at any point. In other words, our enumeration that lists all possible infinite-length binary strings cannot have generated s_d, even though it is itself an infinite-length binary string.

The important part of this distinction, for our purposes, is that even though countable and uncountable sets can both have infinite cardinalities, an uncountably infinite set has many, many more members than does a countably infinite one.

3 Turing Machines and Languages

Now that we've established the difference between countable and uncountable infinity, we can make some statements about Turing machines and languages.

Theorem The set of all Turing machines is countably infinite.

We say that a set is countably infinite if we can establish a one-to-one mapping between it and some subset of the natural numbers. We learned in CS 250 that we can encode any object that can be represented with finite precision using only the alphabet $\{0, 1\}$. We saw examples of the things that this allowed on Tuesday.

If we let $\langle G \rangle$ be the encoding of a Turing machine using $\{0, 1\}$, we can also interpret $\langle G \rangle$ as a (probably very large) binary representation of a natural number n. By enumerating the natural numbers, we can therefore enumerate the binary representation of every possible Turing machine.

The set of all Turing machines is therefore countably infinite.

Theorem Σ^* is countably infinite.

We can use similar logic to show that Σ^* is countably infinite. We can enumerate all of the strings in Σ^* by generating them in what's known as *shortlex order*. Shortlex order orders strings first by length and then by standard lexicographical (i.e. dictionary) order. Mapping the strings in $\{0, 1\}^*$ to the natural numbers might then look something like this:

Index	0	1	2	3	4	5	6	...
String	ϵ	0	1	00	01	11	000	...

Theorem The set of all languages is uncountably infinite.

We can now show that the set of all languages is uncountable. We define a language L as a subset of Σ^*, which means that every element of Σ^* is either present or absent in L. We can therefore draw L as a bit vector parallel to the enumeration of Σ^* that we just created. Each bit in this vector represents the presence or absence in L of the corresponding element of Σ^*. As an example, let $A = \{w \in \{0, 1\}^* \mid w \text{ contains an even number of zeros}\}$. We can represent A as a bit vector parallel to Σ^* like this:

Index	0	1	2	3	4	5	6	...
String	ϵ	0	1	00	01	11	000	...
Present in A?	1	0	1	1	0	1	0	...

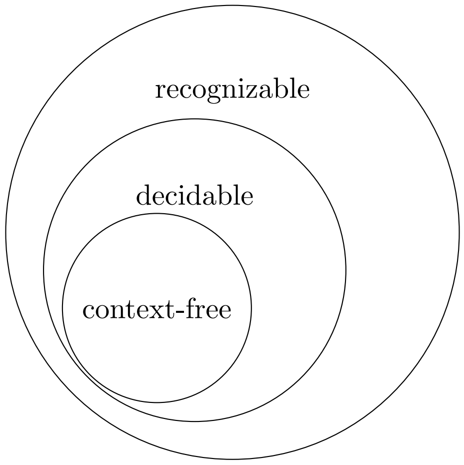
We can generalize this logic to say that any powerset of a countably infinite set is uncountable.

Conclusion There are many, many more languages than Turing machines.

Since the set of all languages is uncountably infinite and the set of all Turing machines is countable, there are many more languages than there are Turing machines. This means that there must be some languages for which no Turing machine exists. We can therefore conclude that there exist languages that are *not* Turing-recognizable.

4 $\mathbf{A_{TM}}$

We've drawn our language hierarchy like this, with Turing-recognizable languages as a strict superset of Turing-decidable ones:



As a reminder, our distinction was this:

A Turing machine M *decides* a language L if and only if:

1. $\forall s \in L$, M Accepts s
2. $\forall s \notin L$, M Rejects s

A Turing machine M *recognizes* a language L if and only if:

1. $\forall s \in L$, M Accepts s

We haven't actually shown that there's a meaningful distinction here. To prove that the set of all Turing-recognizable languages is a proper superset of the set of all Turing-decidable languages, we'll examine the language A_{TM}.

Let $\mathrm{A}_{\mathrm{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and M Accepts $w\}$.

4.1 A_{TM} is Turing-Recognizable

Building a recognizer for A_{TM} is pretty simple. We can use exactly the same strategy that we used for languages like $\mathrm{A}_{\mathrm{DFA}}$ and $\mathrm{A}_{\mathrm{NFA}}$.

```
M = "On input ⟨M, w⟩:
    1. Simulate M on w.
        • If M accepts w, ACCEPT
        • If M rejects w, REJECT"
```

This works just fine for DFAs, NFAs, or PDAs. But since Turing machines have the option of looping forever rather than entering q_{accept} or q_{reject}, we have a third case to take into account.

Note that we did *not* use the word "otherwise" or "else" (e.g. "If M accepts w, Accept. Otherwise, Reject") to determine when to Reject. If we're building a *decider* for a language, it's fine to use these words, because we know that every step should terminate. Since the simulation of M on w may never terminate, though, we cannot use this same strategy when constructing a recognizer.

4.2 $\mathrm{A_{TM}}$ is Not Turing-Decidable

Showing that $\mathrm{A_{TM}}$ is undecidable is a bit trickier. First, we'll assume that $\mathrm{A_{TM}}$ is decidable. Then there is a machine, $M_{A_{TM}}$, that decides it.

We can now construct a machine D as follows:

$D =$ "On input $\langle M \rangle$:

1. Simulate $M_{A_{TM}}$ on $\langle M, \langle M \rangle \rangle$.
    - If $M_{A_{TM}}$ accepts $\langle M, \langle M \rangle \rangle$, REJECT
    - If $M_{A_{TM}}$ rejects $\langle M, \langle M \rangle \rangle$, ACCEPT"

D takes an encoding of a machine as input. It uses $M_{A_{TM}}$ to determine whether or not that machine accepts an encoding of itself. If so, D rejects the machine. If not, D accepts.

What happens if we run D with input $\langle D \rangle$?

- If D accepts $\langle D \rangle$, then D must REJECT $\langle D \rangle$
- If D rejects $\langle D \rangle$, then D must ACCEPT $\langle D \rangle$.

Clearly, this is paradoxical. Since the construction of D was enabled by our assumption that $M_{A_{TM}}$ existed and this resulted in a contradiction, we can conclude that our assumption was incorrect, and $\mathrm{A_{TM}}$ is undecidable.

4.3 $\mathrm{A_{TM}}$'s Implications for Recognizability

We can now conclude that the set of recognizable (but not decidable) languages contains at least one member. $\mathrm{A_{TM}}$ is not only the first language in this class that we've seen, but also the most important.

Assume that $\mathrm{A_{TM}}$ was decidable, but that some other undecidable language B was Turing-recognizable. Then there would exist a machine M_B that recognized (but did not decide) B.

We could clearly construct a decider for B by running $M_{A_{TM}}$ on $\langle M_B, w \rangle$. In other words, if A_{TM} was decidable, then *every* Turing-recognizable language would also be decidable.

5 Recognizability vs. Decidability

Theorem If L is Turing-recognizable and $\overline{L}$ is Turing-recognizable, then L is Turing-decidable.

If L and $\overline{L}$ were both Turing-recognizable, we could construct a decider for L like so:

Let M_L be a machine that recognizes L and $M_{\overline{L}}$ be a machine that recognizes $\overline{L}$. We'll create a nondeterministic machine that decides L. $M =$ " On input w:

1. Simulate M_L and $M_{\overline{L}}$ in parallel.
    - If M_L accepts w, ACCEPT
    - If $M_{\overline{L}}$ accepts w, REJECT"

Since we know that either $w \in L$ or $w \in \overline{L}$ and we have recognizers for both languages, we can create a decider for L by determining which machine recognizes w. As a corollary, we can say that if L is recognizable but not decidable, then $\overline{L}$ is *not* recognizable. Otherwise, the class of decidable languages would equivalent to the class of recognizable languages, which we've just shown to be false.

This lets us conclude the following: Since $\mathrm{A_{TM}}$ is recognizable, but not decidable, $\overline{\mathrm{A_{TM}}}$ must be undecidable. This will be important later, but for now, we can see that it validates the theorem that we proved using diagonalization at the beginning of the lecture: there exist languages that are not Turing-recognizable, and that therefore exist outside of all of the circles on our language hierarchy.

6 Reductions

We've now proven the existence of two undecidable languages ($\mathrm{A_{TM}}$ and $\overline{\mathrm{A_{TM}}}$) and one unrecognizable language ($\overline{\mathrm{A_{TM}}}$). The proof for both of these was a bit involved, and required us to generate a paradox.

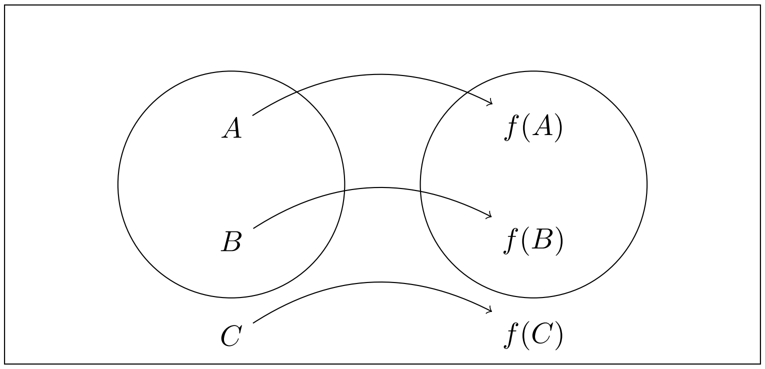
The good news is that we need not do this for *every* undecidable or unrecognizable language; we have a strategy that we can use to place languages in these categories. The bad news is that this process is probably the single most difficult concept in this course.

We say that a problem A is *reducible* to another problem B if a solution to B can be used to solve A.

To get even more specific, we say that a language A is *mapping reducible* to a language B if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that:

1. $\forall s(s \in A \to F(s) \in B)$
2. $\forall s(s \notin A \to F(s) \notin B)$

In other words, F is a function whose result is in B if and only if its input was in A:



There's one element that we've glossed over, which is the definition of a computable function. We say that $f : \Sigma^* \to \Sigma^*$ is a computable function if there exists a Turing machine M that, on input w, halts with $f(w)$ on its tape.

If A is mapping reducible to B, we write $A \leq_m B$. To show that this is the case, we need only construct a Turing machine F that:

- Given some $w \in A$ outputs an $s \in B$
- Given some $w \notin A$ outputs an $s \notin B$

6.1 Creating Mapping Reductions

Since A_{TM} is our prototypical undecidable language, we'll show that another language B is undecidable by creating a mapping that shows that $\text{A}_{\text{TM}} \leq_m B$. This shows that if we could solve B, we could also solve A_{TM}.

All such mapping reductions work by implying the existence of a machine $M_{A_{TM}}$ that decides A_{TM}. As such, all mapping reductions use the same proof by contradiction, written below:

Proof Assume that B is Turing-decidable. Then there exists a machine M_B that decides it.

We can create construct a machine $M_{A_{TM}}$ that decides A_{TM} as follows:

```
$M_{A_{TM}}$ = "On input $\langle M, w \rangle$:
  1. Run $M_B$ on $F(\langle M, w \rangle)$.
       • If $M_B$ accepts $F(\langle M, w \rangle)$, ACCEPT
       • REJECT"
```

Since we've already shown that A_{TM} is undecidable, we've reached a contradiction. Our assumption is therefore false, and B is undecidable.

6.2 The Halting Problem

Many of you were in David Lu's section of Computer Science 251, for which I was a TA. During my lectures on program correctness, we discussed the Halting Problem: given an arbitrary input, does this program halt, or loop infinitely? We called the Halting Problem a canonical example of an uncomputable problem. We will prove that the Halting Problem is undecidable now, using a mapping reduction.

Proof Let $\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a Turing machine and } M \text{ halts on input } w\}$. Show that HALT_{TM} is undecidable.

To do this, we'll demonstrate that $\text{A}_{\text{TM}} \leq_m \text{HALT}_{\text{TM}}$. Doing this is as simple as constructing a Turing machine that, given an input $\langle M, w \rangle$ produces an output $\langle M', x \rangle$ such that M' halts on x if and only if M accepts w.

```
$F$ = "On input $\langle M, w \rangle$:
  1. Construct a machine $M'$ as follows: $M'$ = "On input $x$:
       (a) Simulate $M$ on $w$.
             • If $M$ accepts $w$, ACCEPT $x$
             • If $M$ rejects $w$, enter an infinite loop."
  2. Output $\langle M', w \rangle$."
```

Note that F creates M' and outputs it without ever running it. That means that F will always halt, even if the simulation step in M' does not.

What is $L(M')$? We have three cases to consider.

Case 1: M accepts w

If M accepts w, then M' accepts x, regardless of what x actually is. This means that $L(M') = \Sigma^*$. Since M' accepts all inputs, M' halts on all inputs.

This means that when we output $\langle M', w\rangle$, we've output a machine and a word on which that machine will halt. This in turn means that $\langle M', w\rangle \in \textsc{Halt}_{\text{TM}}$.

Case 2: M rejects w

If M rejects w, M' enters an infinite loop, regardless of the value of x. This means that M' loops on all inputs. Outputting $\langle M', w\rangle$ is therefore outputting a machine and a word on which that machine will *not* halt. In this case, $\langle M', w\rangle \notin \textsc{Halt}_{\text{TM}}$.

Case 3: M loops on w

In this case, the simulation step of M' will never terminate, and M' will loop regardless of the value of x. This means that M' does not halt on w, and $\langle M', w\rangle \notin \textsc{Halt}_{\text{TM}}$.

This means that the *only* case in which we output a machine and a word on which that machine halts will be Case 1, when M accepts w. As discussed before, this allows us to create a following decider for A_{TM}.

Assume that $\textsc{Halt}_{\text{TM}}$ is decidable. Then there is a Turing machine H that decides it.

$M_{A_{TM}}$ = "On input $\langle M, w\rangle$:

1. Run H on $F(\langle M, w\rangle)$.
    - If H accepts $F(\langle M, w\rangle)$, ACCEPT
    - REJECT

Since we know that A_{TM} is undecidable, we've reached a contradiction. We can therefore conclude that our initial assumption was false, and that $\textsc{Halt}_{\text{TM}}$ is undecidable.