Undecidable Languages

Foundations of Computer Science, Fall 2018

Remember that a language $A \subseteq \Sigma^*$ is decidable (or Turing-decidable when we want to be more precise) when there is a *total* Turing machine M that accepts A. By the Church-Turing thesis, we can say that a decision problem $d : \Sigma^* \longrightarrow \{1, 0\}$ is computable when its associated language $\{u \in \Sigma^* \mid f(u) = 1\}$ is decidable.

Existence of undecidable languages

A language is said to be *undecidable* when it is not decidable. A simple counting argument shows that there has to be at least one undecidable language. Intuitively, there are more languages (over alphabet Σ) than there are Turing machines (over alphabet Σ). For simplicity, here, we'll focus only on alphabet $\Sigma = \{0, 1\}$, but clearly, the argument generalizes to arbitrary alphabets.

Because there are infinitely many languages and infinitely many Turing machines, making such a statement precise requires a definition of "size" for infinite sets.

We first need a few definitions about functions.

A function $f : A \longrightarrow B$ is *one-to-one* (or injective) if it maps distinct elements of A into distinct elements of B (that is, if $a \neq b$, then $f(a) \neq f(b)$).

A function $f : A \longrightarrow B$ is *onto* (or surjective) if every element of B is in the image of A under f (that is, if for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$).

A function $f : A \longrightarrow B$ is a *one-to-one correspondance* (or bijective) if it is both one-to-one and onto.

Definition: Two sets A and B are *equipollent* (have the same size), written $A \approx B$, if you can match every element of A with a distinct element of B, and vice versa. Formally, $A \approx B$ when there exists a *one-to-one correspondence* (a bijective function) $f : A \longrightarrow B$.

Definition: Set A is "no bigger than" set B, written $A \preceq B$, if you can match every element of A with a distinct element of B. Formally, $A \preceq B$ if there $A \approx C$ for some $C \subseteq B$, or equivalently if there exists a one-to-one function $f : A \longrightarrow B$.

Definition: $A \prec B$ if $A \preceq B$ but $A \not\approx B$.

Properties:

(i) Both $\approx$ and $\preceq$ are transitive.

(ii) (Cantor-Bernstein) If $A \preceq B$ and $B \preceq A$ then $A \approx B$.

(iii) If $A \prec B$, then there is no onto function $f : A \longrightarrow B$.

Property (i) is pretty easy to show. Property (ii) is much more challenging. Property (iii) can be proved from property (ii).

If $\mathbb{N}$ is the set of natural numbers and $2\mathbb{N}$ the set of even natural numbers, then the bijective function $f : \mathbb{N} \longrightarrow 2\mathbb{N}$ given by $f(n) = 2n$ shows that $\mathbb{N} \approx 2\mathbb{N}$. Similarly, you can show that $\mathbb{N} \approx 2\mathbb{N} + 1$, where $2\mathbb{N} + 1$ is the set of odd natural numbers. The bijective function

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ -(n+1)/2 & \text{if } n \text{ is odd} \end{cases}$$

shows that $\mathbb{N} \approx \mathbb{Z}$. With a bit more effort, we can show that $\mathbb{N} \approx \mathbb{Q}^+$, the set of all non-negative rational numbers: establishing $\mathbb{N} \preceq \mathbb{Q}^+$ is trivial, while establishing $\mathbb{Q}^+ \preceq \mathbb{N}$ uses the one-to-one function f defined by:

$$\begin{aligned} &f(0) = 0 \\ &f(n/m) = \Delta(n+m-2) + m - 1 \quad \text{for n¿0} \end{aligned}$$

where $\Delta(k)$ is the kth triangular number, defined by $\Delta(0) = 0$ and $\Delta(n+1) = \Delta(n) + n$. the first triangular numbers are $0, 1, 3, 6, 10, 15, 21, \ldots$.[1] By property (ii) above, these two facts yield $\mathbb{N} \approx \mathbb{Q}^+$. A trick similar to the one used for $\mathbb{Z}$ can be used to show $\mathbb{N} \approx \mathbb{Q}$.

Not every infinite set is equipollent to $\mathbb{N}$. A classic diagnolization argument can be used to show that $\mathbb{N} \prec \mathbb{R}$, the set of real numbers. Rather than show that result, though, the following more general result showing a hierarchy of non-equipollent sets will be more useful. Given a set A, let 2^A be the powerset of A, that is, the set of all subsets of A. (The powerset 2^A is sometimes denoted $\wp(A)$.)

Cantor's Theorem: For any set A, we have $A \prec 2^A$.

Proof: Clearly, $A \preceq 2^A$, by taking $f : A \longrightarrow 2^A$ to be $f(x) = \{x\}$.

To show that $A \not\approx 2^A$, we argue by contradiction. Suppose that there *were* a bijective function $f : A \longrightarrow 2^A$. I'll show that this assumption leads to an absurdity.

Construct the following set:

$$A_0 = \{x \in A \mid x \notin f(x)\}$$

This a well-defined subset of A. Therefore, because f is onto, there must exist $a_0 \in A_0$ such that $f(a_0) = A_0$.

Now, does $a_0 \in A_0$? There are only two possibilities, yes or no. Neither works:

[1]Imagine putting all non-negative rational numbers in an infinite two-dimensional matrix with rational number i/j at the cell in column i and row j. Function f associates a natural number with each rational number by traversing the array in diagonal bands, where band k lists all rational number of the form i/j with $i + j = k$.

– If $a_0 \in A_0$, then by definition of a_0, $a_0 \notin f(a_0)$, that is, $a_0 \notin f(a_0) = A_0$.

– If $a_0 \notin A_0$, then by definition of a_0, $a_0 \in f(a_0)$ (otherwise, a_0 would be in A_0) and thus $a_0 \in f(a_0) = A_0$.

Either way, we get an absurdity. So our assumption that there is a bijection f cannot be. Thus, $A \not\approx 2^A$. $\square$

Cantor's Theorem means, in particular, that we get an infinite tower of sets of increasing infinite sizes:

$$\mathbb{N} \prec 2^{\mathbb{N}} \prec 2^{2^{\mathbb{N}}} \prec 2^{2^{2^{\mathbb{N}}}} \prec \dots$$

We can now show that there must be an undecidable language. Let $T(\{0,1\})$ be the set of Turing machines over alphabet $\{0,1\}$. We only need to establish that

$$T(\{0,1\}) \prec 2^{\{0,1\}^*} \tag{1}$$

where of course $2^{\{0,1\}^*}$ is the set of all languages over $\{0,1\}$. Why does this help? Consider the function $L : T(\Sigma) \longrightarrow 2^{\Sigma^*}$ that associates to every Turing machine the language it accepts. By property (iii), equation 1 says that there is no onto function $T(\Sigma) \longrightarrow 2^{\Sigma^*}$, and therefore L is not onto. That L is not onto simply means that there is a language $A \in 2^{\Sigma^*}$ such that there is no M with $L(M) = A$, and in particular no total M: language A is undecidable.

We establish 1 in a few steps. First, we show that $T(\{0,1\}) \preceq \mathbb{N}$. For this, it suffices to encode every Turing machines M into a natural number. This encoding turns out to be handy later, so let's spend some time fleshing it out. We identify Turing machines that differ only by the names of their states.

If $M = (Q, \{0,1\}, \Gamma, \llcorner\!\lrcorner, \vdash, \delta, s, \mathit{acc}, \mathit{rej})$, without loss of generality, we can take $Q = \{1, \dots, n\}$, with $s = 1$, $\mathit{acc} = 2$, and $\mathit{rej} = 3$. Also without loss of generality, we can take $\vdash = 2$, $\llcorner\!\lrcorner = 3$, and Γ to be of the form $\{0, 1, 2, 3, \dots\}$: symbols in the tape alphabet that are not $0, 1$ (from the fixed input alphabet) can be anything, really, and we can take them to be natural numbers > 1. This means, really, that to complete describe the Turing machine M, we only need to describe the transitions. Each transition is of the form $\delta(n_1, n_2) = (n_3, n_4, n_5)$ where n_1, n_3 are number representing the states, n_2, n_4 are numbers representing the symbols, and $n_5 \in \{1, 2\}$ where 1 represents L and 2 represents R. Each individual transition therefore can be encoded as a string

$$0^{n_1}10^{n_2}10^{n_3}10^{n_4}10^{n_5} \tag{2}$$

and the entire Turing machine be encoded as a string

$$111\ \mathit{code}_1\ 11\ \mathit{code}_2\ 11\ \dots\ 11\ \mathit{code}_k\ 111 \tag{3}$$

where each code_i is a string of the form 2, and each transition of M is encoded by one of the code_i

Every string over $\{0,1\}$ can be interpreted for the code at most one Turing machine. (Why?) Many strings over $\{0,1\}$ are not the code of any Turing machine, and these are easy to identify. We write $\langle M\rangle$ for the encoding over $\{0,1\}$ of Turing machine M.

By treating the string $\langle M\rangle$ as the binary representation of a natural number, we can associate to every Turing machine M a distinct natural number, giving us a one-to-one map from $T(\{0,1\})$ to $\mathbb{N}$. So $T(\{0,1\}) \preceq \mathbb{N}$.

Next, we can show that $\mathbb{N} \preceq \{0,1\}^*$. That's actually pretty easy. The map $f : \mathbb{N} \longrightarrow \{0,1\}^*$ given by $f(n) = 0^n$ is clearly one-to-one. So $\mathbb{N} \preceq \{0,1\}^*$.

Cantor's Theorem gives us that $\{0,1\}^* \prec 2^{\{0,1\}^*}$. So we have

$$T(\{0,1\}) \preceq \mathbb{N} \preceq \{0,1\}^* \prec 2^{\{0,1\}^*}$$

and transitivity gives us $T(\{0,1\}) \prec 2^{\{0,1\}^*}$, as required.

Universal Turing machines

The argument above shows that there must be at least one undecidable language. It doesn't help us identify one.

To do so, first we need one of Turing's main result about his machines: universality. It is possible to develop a *universal* Turing machine U that takes as input an encoding of a Turing machine M and an input string w and simulates running Turing machine M on input string w, accepting when M accepts, rejecting when M rejects, and looping when M loops. This may loop weird at first, but really it is no weirder than writing a Python interpreter in Python.

We use the code defined by 3, and write $\langle M\rangle w$ for the encoding of machine M followed by w. Note that we can always recognize w in $\langle M\rangle w$ as the string following the second block of 111.

The easiest way to define Turing machine U is to start with a 3-tapes Turing machine U' with input alphabet $\{0,1\}$ and tape alphabet $\{0,1,\vdash,\llcorner\!\lrcorner\}$.[2] The first tape is the input tape, and the tape head on that tape is used to look up the transitions to make when given input $\langle M, w\rangle$. (Recall that the encoding $\langle M\rangle$ records all the transitions of the Turing machine using codes of the form 2.) The second tape of U' will simulate the tape of M. Multiple cells of the tape will simulate single cell of U', since we need to encode the tape alphabet of M using esentially $0,1$. We can use 0^n to represent symbol n, and separate the cells using 1. The third tape of U' holds the state M, with state n represented as 0^n. The behavior of U' is as follows:

1. Check the format of tape 1 to make sure its prefix describes the encoding of a Turing

[2]This description is taken from Hopcroft and Ullman *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley.

machine (checking it starts with 111, that is has a sequence of codes of the form 2 separated by 11 and ended by 111, etc). Reject if not.

2. Initialize tape 2 to contain w, the portion of the input beyond the second block of 111, suitably encoded to represent the symbols of the tape alphabet of M. Initialize tape 3 to hold 0, the initial state of M.

3. if tape 3 holds 00, accept; if tape 3 holds 000, reject.

4. Let j be the encoded symbol currently scanned by tape head 2, and let 0^i be the current content of tape 3. Scan tape 1 from left to the second block 111, looking for a substring beginning 110^i10^j1. If no such string is found, reject. If such a code is found, let be $0^i10^j10^k10^l10^m$. Put 0^k on tape 3, put l on tape 2 in place of j (possibly shifting the tape left and right if needed), and move the head in direction m. Go back to step 3.

It is straightforward, if tedious, to check that U' accepts $\langle M, w\rangle$ when M accepts w and rejects $\langle M, w\rangle$ when M rejects w.

We know from previous lectures that a three-tapes Turing machine can be simulated by a one-tape Turing machine, and we can take U to be the one-tape Turing machine simulating U'.

The Halting Problem

We say that a Turing machine M halts on input w if M either accepts or rejects w, we don't care which. Basically, M does not spin forever on input w.

Consider the following language, called the *Halting Problem*:

$$HP = \{\langle M\rangle, w \mid M \text{ halts on input } w\}$$

I claim that HP is undecidable, that is, there is no total Turing machine M that accepts HP. We argue by contradiction: assume HP is decidable, and derive an absurdity.

Assume HP is decidable. That means we have a total Turing machine K that accepts HP. That is, K accepts $\langle M\rangle w$ when M halts on w, and rejects $\langle M\rangle w$ when M does not halt on w. Let $\langle K\rangle$ be the encoding of K.

Using K, construct another Turing machine I as follows:

```
On input $x$:
  1. Run $U$ with input $\langle K\rangle xx$
  2. If $U$ rejects, accept
  3. If $U$ accepts, go into an infinite loop
```

Clearly, we can implement I as a Turing machine since we know $\langle K\rangle$. We can just modify U to rewrite $\langle K\rangle xx$ on its input tape and modify the reject and accept states of U. Since I

is a Turing machine, it has an encoding $\langle I \rangle$, which is just a string over $\{0, 1\}$. We can also ask whether I halts on any given input. Let's be devious and ask whether I halts on input $\langle I \rangle$!

There are only two possibilities. Either I halts on input $\langle I \rangle$, or it does not. Neither makes sense.

– Say I halts on input $\langle I \rangle$. By definition of I, this happens only when U rejects $\langle K \rangle\langle I \rangle\langle I \rangle$. Since U is a universal Turing machine, U rejects when K rejects $\langle I \rangle\langle I \rangle$. But K is the Turing machine deciding HP, and it rejects exactly when I *does not* halt on $\langle I \rangle$. But we said I halts on $\langle I \rangle$. That's absurd: it can't do both.

– Say I does not halt on input $\langle I \rangle$. By definition of I, this happens only when U accepts $\langle K \rangle\langle I \rangle\langle I \rangle$. But U is a universal Turing machine, so U accepts when K accepts $\langle I \rangle\langle I \rangle$. But K is the Turing machine deciding HP, and it accepts when I *does* halt on input $\langle I \rangle$. But we said I does not halt on $\langle I \rangle$. That's absurd: it can't do both.

Either way, we get an absurdity. So our assumption that HP is decidable must be wrong. There is not total Turing machine K that accepts HP, and HP is undecidable.