Notes on Undecidability

Foundations of Computer Science

Fall 2017

Church-Turing Thesis

What the multitape Turing machine example from the notes from last lecture shows is that the definition of decidable languages is quite robust: extensions to Turing machines do not add any expressive power to Turing machines. Intuitively, standard Turing machines are powerful enough to be able to simulate any variant of Turing machines.

In fact, Turing machines are powerful enough, as far as we know, to simulate any model of computation, or any programming language. This thesis is known as the *Church-Turing thesis*:

The Church-Turing Thesis: Any feasible model of computation can be simulated by Turing machines.

The notion *feasible* is kept vague, partly because it cannot really be defined. But it captures the idea of a model of computation that is, for instance, physically realizable—it does not depend on the ability to perform an infinite amount of computation in a finite amount of time, or requiring an infinite amount of space.

Because *feasible* cannot be defined, the Church-Turing thesis is not a theorem. Nevertheless, we have a lot of evidence that the thesis is true.

While it may seem surprising at first blush that a Turing machine can in fact simulate any model of computation, it really should not be. It should be clear, for starters, that computers can simulate any model of computation. It then only remains to argue that a Turing machine can simulate a computer. But a computer is really just a CPU with a bunch of attendant hardward to make it interact with the world, and a CPU is really just a sort of Turing machines. Putting it the other way around, it is not difficult to imagine simulating a CPU with a Turing machine: the alphabet of the Turing machine includes `0` and `1`, which are the basic values that a CPU works with; the registers of the CPU can live on the tape; the finite number of gates of the CPU can be modeled by the finite control of the Turing machine; and the memory that the CPU has access to, which is finite, can be against stored on the tape. While a CPU addresses memory by indexing into it, the Turing machine would have to move the tape head to the appropriate cell corresponding to the memory location, but

the difference is merely one of efficiency.[1]

The Church-Turing thesis makes the distinctions between decidable languages and languages that are *not* decidable relevant for the working programmer. Recall that languages over an alphabet Σ are just functions from Σ^* to the Booleans. A language A is decidable if it can be accepted by a total Turing machine; in other words, a function A from Σ^* to the Booleans is decidable if you can come up with a total Turing machines that says yes to a string w exactly when the function A maps w to `true`. A language is not decidable (we'll call those undecidable) if there is no total Turing machine that accepts it. Because of the Church-Turing thesis, if a language is undecidable, then not only is there no total Turing machine that accepts it, there is also no program in any of your favorite programming languages that implements the function corresponding to the language: if there was, then you could also write a Turing machine that simulates that program, and this would contradict undecidability. Thus, if a language is undecidable, there can be no program that implements the corresponding function, period. The notion of decidability, even though it seems like a technical notion having to do with Turing machines only, in fact impacts programming in general. Being able to determine what languages are decidable and which aren't determines which problems can be solved using computer programs, and which can't.

Undecidability

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 $\{u \in \Sigma^* \mid f(u) = 1\}$ is decidable.

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).

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

[1] The Church-Turing thesis never claimed that Turing machines can simulate any model of computation *efficiently*, just that they can simulate it.

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 bijection $f : \mathbb{N} \longrightarrow 2\mathbb{N}$ 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.

It is easy to show that $\mathbb{N} \preceq \mathbb{Q}^+$, where $\mathbb{Q}^+$ is the set of nonnegative rational numbers. To show that $\mathbb{Q}^+ \preceq \mathbb{N}$, imagine putting all positive rational numbers in an infinite two-dimensional matrix with rational number i/j at the cell in column i and row j. We can now associate a natural number with each rational number traversing the array in diagonal bands, where band k lists all rational number of the form i/j with $i + j = k$. We treat zero specially. Formally, we can define $f : \mathbb{Q}^+ \longrightarrow \mathbb{N}$ by taking

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

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 $0, 1, 3, 6, 10, 15, 21, \ldots$. Function f is one-to-one, so $\mathbb{Q}^+ \preceq \mathbb{N}$. By property (ii) above, $\mathbb{N} \approx \mathbb{Q}^+$.

I will let you figure out how this generalizes to $\mathbb{N} \approx \mathbb{Q}$.

Not every infinite set is equipollent to $\mathbb{N}$. A classic argument can be used to show that $\mathbb{N} \prec \mathbb{R}$, the set of real numbers.

Rather than show the latter, though, here's a result that will be more useful for us.

Cantor's Theorem: For any set A, we have $A \prec 2^A$, where 2^A is the set of subsets of A.

Notation $\wp(A)$ is also used for 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 bijection $f : A \longrightarrow 2^A$. 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:

- 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, $A \not\approx 2^A$. □

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

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

We can now show that there must be an undecidable language. Let $T(\Sigma)$ be the set of Turing machines over alphabet Σ. We establish that

$$T(\Sigma) \prec 2^{\Sigma^*}$$

where of course 2^{Σ^*} is the set of all languages over alphabet Σ.

First, we show that $T(\Sigma) \preceq \mathbb{N}$. For this, it suffices to encode every Turing machines M into a natural number. We identify Turing machines that differ only by the names of the states.

Intuitively, if $M = (Q, \Sigma, \Gamma, \sqcup, \vdash, \delta, s, acc, rej)$, without loss of generality, we can take $Q = \{1, \ldots, n\}$, and if $\Gamma = \{a_1, \ldots, a_k\}$, we can encode every symbol as an integer in $\{1, \ldots, k\}$, taking $1, \ldots, |\Sigma|$ to be the symbols from Σ under a fixed by arbitrary order, and taking $\sqcup = |\Sigma| + 1$ and $\vdash = |\Sigma| + 2$. This means that we can represent the non-transition parts of M using numbers n, k, s, *acc*, and *rej*. For the transition function, note that δ can be described by the list of every tuple (p, a, q, b, d) describing the transition relation, where each tuple is made up of natural numbers, and with nk tuples total (one for each choice of state and symbol). Thus, we can represent the transition function using $5nk$ natural numbers, and therefore we can represent M with natural numbers n, k, s, *acc*, *rej*, and the $5nk$ natural numbers describing the transition function. We can take these $5nk + 5$ natural numbers and encode them uniquely in a single natural number using a Goedel encoding: if we let $2, 3, 5, 7, 11, 13, 17, 19, \ldots$ be an enumeration of distinct primes, then we can represent a tuple $(n_1, n_2, n_3, \ldots, n_l)$ as $2^{n_1}3^{n_2}5^{n_3}7^{n_4}\ldots$. (Unique factorization of natural numbers gives us that different Turing machines yield different natural numbers.)

This means that to every Turing machine M we can associate a distinct natural number, giving us a one-to-one map from $T(\Sigma)$ to $\mathbb{N}$. So $T(\Sigma) \preceq \mathbb{N}$.

Next, we can show that $\mathbb{N} \preceq \Sigma^*$. That's actually pretty easy. Since $\Sigma \neq \varnothing$, then there is an $a \in \Sigma$. The map $f : \mathbb{N} \longrightarrow \Sigma^*$ given by $f(n) = a^n$ is clearly one-to-one. So $\mathbb{N} \preceq \Sigma^*$.

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

$$T(\Sigma) \preceq \mathbb{N} \preceq \Sigma^* \prec 2^{\Sigma^*}$$

and transitivity gives us $T(\Sigma) \prec 2^{\Sigma^*}$.

By property (iii), this means that there is no onto function $T(\Sigma) \longrightarrow 2^{\Sigma^*}$. Consider the function $L : T(\Sigma) \longrightarrow 2^{\Sigma^*}$ that associates to every Turing machine the language it accepts — that L cannot be onto. Therefore, there must be a language $A \in 2^{\Sigma^*}$ such that there is no M with $L(M) = A$. That is, A is undecidable.

The Halting Problem

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. I'm not going to give the description of such a Turing machine U, but I'll put some pointers on the course web page. Note that this is no stranger than writing, say, a Python interpreter in Python. The key here is that one Turing machine can simulate other Turing machines.

To do so, U must take a Turing machine as input, that is, we need a way to encode Turing machines as input of a given alphabet. For the sake of concreteness, let's fix $\Sigma = \{0, 1\}$ here. (The undecidability result does not depend on the alphabet.) It is clear to our computer scientist intuition that we can represent a Turing machine as a string of bits. We need to be able to read off the various components of an encoded Turing machine, but we can do so by encoding each component separately. We've done so with natural numbers above. It's a simple matter to do so with strings of bits. The details of the encoding are unimportant, so I'll just assume that if M is a Turing machine, there is an encoding $\widehat{M}$ of that Turing machine. Similarly, let $\langle u, v \rangle$ be an encoding of the pair u and v in such a way that we can recover both u and v from the encoding.

Thus, universal Turing machine U takes as input strings of the form $\langle \widehat{M}, w \rangle$ — encodings of a pair of a Turing machine encoding and a string, and simulates M with input w.

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

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

I claim that *HP* is undecidable, that is, there is no total Turing machine M that accepts *HP*. It turns out to be a similar argument than the one used for Cantor's Theorem. We argue by contradiction: assume it *is* decidable, and derive an absurdity.

By way of contradiction, assume *HP* is decidable. That means we have a total Turing machine K that accepts *HP*. That is, K accepts $\langle \widehat{M}, w \rangle$ when M halts on w, and rejects when M does not halt on w.

Using K, construct another Turing machine I as follows:

On input x:

1. Run U with input $\langle \widehat{K}, \langle x, x \rangle \rangle$
2. If U rejects, accept
3. If U accepts, go into an infinite loop

Clearly, we can implement I as a Turing machine if we have K and U. Since I is a Turing machine, it has an encoding $\widehat{I}$. We can also ask whether I halts on a given input. The input we care about? $\widehat{I}$ of course!

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

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

- Say I does not halt on input $\widehat{I}$. By definition of I, this happens only when U accepts $\langle \widehat{K}, \langle \widehat{I}, \widehat{I} \rangle \rangle$. But U is a universal Turing machine, so U accepts when K accepts $\langle \widehat{I}, \widehat{I} \rangle$. But K is the Turing machine deciding *HP*, and it accepts when I halts on input $\widehat{I}$. But we said I does not halt on $\widehat{I}$. That's absurd.

Either way, we get an absurdity. So our assumption that *HP* is decidable, that is, that K exists, must be wrong. There is no such K. So *HP* is undecidable.