ECS 120 Lesson 18 – Decidable Problems, the Halting Problem

Oliver Kreylos

Friday, May 11th, 2001

In the last lecture, we had a look at a problem that we claimed was not solvable by an algorithm – the problem whether a given `C` program prints the message "Hello World" and then exits. Today we are going to look at a very similar problem and prove formally that there exists no Turing Machine that can decide it.

1 Problems and Languages

Let us start by defining some terminology for problems. A problem P is a question that can be asked about entities E of some type (Turing Machines, integers, graphs, etc.) and answers either "yes" or "no." For example, the "Hello World" problem can be applied to `C` programs and asks whether an entity E (a certain `C` program) prints "Hello World" and then exits. We denote the answer that problem P gives about entity E as $P(E)$. If $P(E)$ is "yes," we call E an *instance* of P.

At first glance, it seems that the theory about languages we have investigated so thoroughly cannot be applied to reason about problems. A language is merely a set of words over some alphabet, whereas a problem is a question about some entity, that can either be answered "yes" or "no." As it turns out, this difference is not fundamental, but just a matter of notation. We have already seen that any entity of some type we can reason about mathematically can also be converted into words over some alphabet by the encoding mechanism discussed earlier; this means that any entity E can be converted to a word $\langle E \rangle \in \Sigma^*$ for some alphabet Σ. We can now define a language $L(P)$ for problem P as follows: $L(P) := \left\{ \langle O \rangle \in \Sigma^* \;\middle|\; E \text{ is an instance of } P \right\}$, where

Σ is the alphabet needed to encode entity O. In other words, the language for a problem P consists of exactly those words that are encodings of instances of P, i. e., that are encodings of entities for which P's answer is "yes." Being a set of words, $L(P)$ is a language in the usual sense.

We can now directly apply language theory to solving problems: If and only if P is a problem that can be solved by an algorithm, i. e., there exists an algorithm that always terminates and either answers "yes" or "no," then the language $L(P)$ must be Turing-decidable: Given any word $w \in \Sigma^*$, there is a Turing Machine $D(P)$ that can decode $w = \langle E \rangle$ back to an entity E, and can then apply the algorithm to answer P to that entity. An algorithm comes up with an answer after a finite number of steps; therefore, $D(P)$ can accept or reject the word after a finite number of steps. This means that $D(P)$ is a decider for $L(P)$.

2 The Halting Problem Revisited

We defined the halting problem as the question whether a given Turing Machine M halts on input word w. Using the conversion strategy from above, we convert the halting problem to a language $H_{\text{TM}} := \left\{ \langle M, w \rangle \;\middle|\; \text{TM } M \text{ halts on input } w \right\}$. The words in H_{TM} are encodings of Turing Machines and the input words they halt on. Let us first look at a slightly modified version of the halting problem: Let $A_{\text{TM}} := \left\{ \langle M, w \rangle \;\middle|\; \text{TM } M \text{ accepts } w \right\}$ be the language associated with the *acceptance problem*, the question whether a Turing Machine M accepts an input word w.

We will now prove that there can be no algorithm that solves the accepting problem, in other words, there can be no Turing Machine that decides A_{TM}. We already know, however, that there is a Turing Machine that *recognizes* A: A Universal Turing Machine as defined in the last lecture. Thus, our proof will also show that the class of decidable languages is a proper subset of the class of recognizable languages.

Since we can not try all possible algorithms and prove for each that it does not solve the acceptance problem, we have to prove the non-existence of such an algorithm by contradiction: We first assume that there *is* a deciding Turing Machine for A_{TM}, and then we show that this assumption violates known properties of Turing Machines. To form this contradiction, we use a strategy called *diagonalization*, invented by Georg Cantor in 1873 to show that the set of real numbers, $\mathbf{R}$, is uncountable. To develop that idea, we

take a short detour through the theory of infinite sets.

3 Comparing Infinite Sets

Since the number of elements of infinite sets cannot be counted, it is not possible to compare the sizes of two infinite sets by counting their elements and comparing the numbers. Instead, we compare two infinite sets by pairing up their elements: If any element of the first set is paired with exactly one element of the second set, and vice versa, then the sets must have the same number of elements. In practice, for example, the number of students in a class can be compared to the number of chairs in a classroom by observing that, if no chair is empty, no student is standing, and each chair is occupied by exactly one student, the numbers of chairs and students must be identical.

More formally, a function $f\colon X \to Y$ is called *injective* or *one-to-one* if no two elements in X are mapped to the same element in Y: $\forall x_1, x_2 \in X : x_1 \neq x_2 \Longrightarrow f(x_1) \neq f(x_2)$. A function $f\colon X \to Y$ is called *surjective* or *onto* if every element in Y is the function value of some element in X: $\forall y \in Y \exists x \in X : f(x) = y$. A function that is both injective and surjective is called *bijective* or a *correspondence*. Two sets X and Y are the *same size*, if and only if there exists a bijective function $f\colon X \to Y$. This definition of "same size" leads to some surprising facts about infinite sets:

- The set of even integers E is a proper subset of the set of integers $\mathbf{Z}$, $E \subset \mathbf{Z}$, but they are of the same size: The function $f\colon \mathbf{Z} \to E, \quad n \mapsto 2n$ is a bijection.
- The set of integers $\mathbf{Z}$ is a proper subset of the set of rational numbers $\mathbf{Q}$, $\mathbf{Z} \subset \mathbf{Q}$, but they are of the same size: There is a process, called *Cauchy enumeration* or *dovetailing*, that pairs integers and rational numbers by a bijective function.

These are two examples of an important class of infinite sets: Those that have the same size as the set of integers $\mathbf{Z}$, called *countably infinite* sets. The name countably infinite stems from the fact that elements of those sets can be paired with (positive) integers; therefore, each element of such a set can be assigned a unique order number: The elements of these sets can be enumerated in order. The enumeration never ends, but any element of a countably infinite set will be enumerated sooner or later. Some other examples of countably infinite sets are:

- Any infinite language[1] over some alphabet Σ.
- The set of all possible Turing Machines.
- The set of all Turing-recognizable languages, i. e., the class CH–0.

To make talking about sets that are either finite or countably infinite easier, these are commonly called *countable*.

3.1 Uncountable Sets

The examples above might lead to the impression that every infinite set is countable; after all, the sets of integers and rational numbers are very different, yet of the same size. As it turns out, there are sets which are truly "bigger" than the set of integers; it is not possible to assign an integer number to each of their elements. The first set proven to be *uncountable* was the set of real numbers $\mathbf{R}$; the proof by Georg Cantor uses his diagonalization method.

The diagonalization idea is based on the fact that each real number can be written as an infinite decimal fraction[2]. We now assume that the set of real numbers is countable; that way, every number $x \in \mathbf{R}$ is associated with a positive integer n. We can now create a real number D by picking as the i-th decimal digit of D the i-th decimal digit of the real number associated with the integer i. If we wrote all real numbers into an (infinite) table, as in Table 1, with the numbers ordered by their associated integers down the rows and their digits accross the columns, the number D would consist of all the diagonal elements of that table. Hence the name diagonalization method.

From D we create another number N by changing each digit of D to some other digit, for example replacing each digit d_i by $(d_i + 5) \bmod 10$. This number N must also be a real number, so it must occur somewhere in the table of real numbers, say at position k. But then the k-th digit of N was chosen to be different from the k-th digit of the number in position k; this means the number N can appear nowhere in the table. Therefore our assumption that the table contains all real numbers must have been wrong, which means that there are more real numbers than integers. This result

[1] Finite languages like $A = \{\texttt{ababba}\}$ are not countably infinite; they are finite and have less elements than $\mathbf{Z}$.

[2] If some real number x has a finite decimal expansion, say $x = 0.1234$, then it is written as $x = 0.12340000\ldots$. By definition, these two decimal fraction have the same value.

Order	Digits						
1.	$\underline{3}$	4	2	1	0	7	…
2.	1	$\underline{3}$	0	0	9	9	…
3.	2	0	$\underline{1}$	3	8	4	…
4.	1	3	9	$\underline{8}$	3	3	…
5.	3	2	4	7	$\underline{5}$	9	…
6.	2	5	4	9	4	$\underline{5}$	…
⋮				⋮			
m.	3	3	1	8	5	5	…
⋮				⋮			

Table 1: A hypothetical table of all real numbers, and the number D created by picking all diagonal elements. Since D is a real number, it must occur somewhere in the table, say in position m.

shows that the set or real numbers is truly "bigger" than the set of integers. To write down this fact, we generalize the cardinality notation $|S|$. If S is a finite set, $|S|$ still denotes the number of its elements. If S is infinite however, there can be no specific cardinality; for infinite sets, the $|\cdot|$ notation is only used to compare cardinalities of sets:

- $|A| = |B|$, if A and B are "same size," i. e., if there exists a bijective function $f\colon A \to B$.
- $|A| \leq |B|$, if there exists an injective function $f\colon A \to B$.
- $|A| \geq |B|$, if there exists a surjective function $f\colon A \to B$.
- $|A| < |B|$ iff $|A| \leq |B|$ and not $|A| = |B|$.
- $|A| > |B|$ iff $|A| \geq |B|$ and not $|A| = |B|$.

These definitions lead to some properties which make the comparisons work as normal comparisons of numbers:

- $|A| = |A|$ for any set A (finite or infinite).
- $|A| \leq |B| \wedge |A| \geq |B| \implies |A| = |B|$.
- $|A| \leq |B| \iff |B| \leq |A|$.

3.2 The ℵ Hierarchy of Infinite Sets

This notation allows us to compare the sizes of infinite sets. We have already seen two classes of infinity: $|\mathbf{Z}|$ and $|\mathbf{R}|$. The question is: Are these the only two classes, or are there more? To answer this, we need another theorem from set theory:

Theorem 1 *For any set S, $|S| < |\mathcal{P}(S)|$.*

Proof We first show that $S \leq \mathcal{P}(S)$ by giving the function $f\colon S \to \mathcal{P}(S)$, $a \mapsto \{a\}$. This function is injective: For every $a, b \in S$, $a \neq b \Longrightarrow \{a\} \neq \{b\}$. We now have to show that not $|S| = |\mathcal{P}(S)|$, i. e., that there is no bijection between S and $\mathcal{P}(S)$. Again, we prove this by contradiction based on the diagonalization method: Assume that there is a bijective function $f\colon S \to \mathcal{P}(S)$. From f we construct a "diagonal" set $D := \left\{ a \in S \;\middle|\; a \in f(a), \right\} \subset S$, and then a "negated" set $N = \left\{ a \in S \;\middle|\; a \notin f(a) \right\} \subset S$. N is a subset of S, i. e., it is an element of the power set $\mathcal{P}(S)$. We assumed that f was a bijection, therefore there must be an $a \in S$ such that $f(a) = N$. Now there are two possible cases:

- $a \in N$. In this case, by definition of N, $a \notin f(a) = N \Longrightarrow a \notin N$.
- $a \notin N$. In this case, by definition of N, $a \in f(a) = N \Longrightarrow a \in N$.

Both of these cases are contradictions in themselves, therefore the assumption that f is a bijection must have been wrong, and there can be no such bijection. Thus, $|S| \neq |\mathcal{P}(S)|$.

From this theorem, we can conclude that there are *infinitely many* different classes of infinity: $|S| < |\mathcal{P}(S)| < \left|\mathcal{P}\big(\mathcal{P}(S)\big)\right| < \cdots$ Since these classes are important, they have special names, based on the first character ℵ ("aleph") of the Hebrew alphabet:

Base Case $\aleph_0 := |\mathbf{Z}|$.

Inductive Case For all $n \geq 0$: If $|S| = \aleph_n$, then $\aleph_{n+1} := |\mathcal{P}(S)|$.

From our definition and the last theorem, we know that $\aleph_0 < \aleph_1 < \aleph_2 < \cdots$ define an infinite hierarchy of infinite sets. By coincidence, the cardinality of the set of real numbers is $\aleph_1$, i. e., $|\mathbf{R}| = |\mathcal{P}(\mathbf{Z})|$ (we are not going to prove that here).

This result also shows, by using that Σ^* is countable and that thus $|\mathcal{P}(\Sigma^*)| = \aleph_1$, there are exactly as many different languages as there are real numbers. We already said that $|\text{CH–0}| = \aleph_0$; therefore, there are "infinitely many more" languages not in CH–0 than there are in CH–0; in other words, the most general class of languages we have been looking at so far is only a very small subset of all possible languages. Since there is generally no "finite" way of specifying languages not in CH–0, they are usually beyond our comprehension and not treated by language theory.

We now finish this detour from our discussion and resume to show that the acceptance problem for Turing Machines is not decidable.

4 The Undecidability of the Acceptance Problem

To prove this claim, we first assume that it is decidable, and then use a diagonalization method to form a contradiction. If A_{TM} is decidable, there must be a deciding Turing Machine H that accepts it. We can write the function performed by H on an input word $\langle M, w\rangle$ in the following way:

$$H(\langle M, w\rangle) := \begin{cases} \text{accept}, & \text{if } M \text{ accepts } w \\ \text{reject}, & \text{otherwise} \end{cases}$$

Based on H, we now construct a "diagonal" machine D. The idea is, that while H asks whether a Turing Machine M accepts *any* input word w, D specializes this question in the sense that it asks whether M accepts a special input word: the encoding of M itself.

This idea of feeding a Turing Machine its own encoding as input word seems weird at first. It does occur in practice, however: One common example are programming language compilers; these are routinely written in their own languages. If a user of `gcc`, the GNU project `C` compiler, wants to upgrade her compiler, she would download the newest source code from the project web page and compile the new compiler using her current one. Essentially, the compiler is fed with its own description (the source code) and generates a new compiler. More generally, one can think that an encoding of a Turing Machine is just some word, and that a Turing Machine can be presented with any input word. If a Turing Machine's input alphabet is not capable of representing its own encoding, one could simply add characters to

the input alphabet, and let the machine reject when encountering such an added character. This would not change the Turing Machines behaviour and its language at all. Usually feeding a Turing Machine its own description does not make much sense, the important part is that it is *possible*.

Using this idea, the behaviour of D can be summarized as follows:

$$D(\langle M \rangle) := H(\langle M, \langle M \rangle \rangle) = \begin{cases} \text{accept}, & \text{if } M \text{ accepts } \langle M \rangle \\ \text{reject}, & \text{otherwise} \end{cases}$$

If H is a deciding Turing Machine, so is D: It will copy its input word $\langle M \rangle$ to form the new input word $\langle M, \langle M \rangle \rangle$, run H on that input, and after H either accepts or rejects do the same. Based on D, we now build a "negator" machine N, that reads the same input as D and just negates D's answer:

$$N(\langle M \rangle) := \neg D(\langle M \rangle) = \neg H(\langle M, \langle M \rangle \rangle) = \begin{cases} \text{reject}, & \text{if } M \text{ accepts } \langle M \rangle \\ \text{accept}, & \text{otherwise} \end{cases}$$

Again, if D is a deciding Turing Machine, so is N. The question now is: Is N a valid Turing Machine? As a decider, N has to stop on each possible input word w; so what happens if the machine N is fed its own description $\langle N \rangle$? In that case, the answer would be

$$N(\langle N \rangle) = \begin{cases} \text{reject}, & \text{if } N \text{ accepts } \langle N \rangle \\ \text{accept}, & \text{otherwise} \end{cases}$$

If N would ever accept its own description, this would mean that N accepts $\langle N \rangle \iff N$ does not accept $\langle N \rangle$ – a direct contradiction. If, on the other hand, N would ever reject its own description, this would mean that N rejects $\langle N \rangle \iff N$ accepts $\langle N \rangle$ – another contradiction. Thus, N can neither accept nor reject its own description; its only possible reaction is to loop. But by our assumption, N is a decider, so it can never loop. This means our assumption must have been wrong, which in turn means that the acceptance problem is not decidable.

To "visualize" this proof in a manner similar to Table 1, we can build a table listing all Turing Machines down the rows, and their reaction to another Turing Machine's description across the columns. Since the set of Turing Machines is countable, this table can be created. If we number all Turing Machines in some way, the table could look like Table 2. Machine N is constructed by negating all diagonal entries in that table; since N must be an element of the table itself, the contradiction occurs when the diagonal entry of the row containing N must be constructed: It must be the negation of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\langle M_5 \rangle$	...	$\langle N \rangle$	...
M_1	<u>accept</u>	reject	loop	accept	loop	...		
M_2	reject	<u>loop</u>	reject	accept	accept	...		
M_3	accept	reject	<u>reject</u>	reject	accept	...		
M_4	loop	accept	reject	<u>accept</u>	reject	...		
M_5	reject	loop	accept	accept	<u>loop</u>	...		
$\vdots$				$\vdots$				
N	reject	accept	accept	reject	accept	...	"clash"	...
$\vdots$				$\vdots$				

Table 2: A table of Turing Machines and their reactions to the encodings of other Turing Machines. The entries for machine N are constructed by negating diagonal entries.