Lecture 16

Undecidability and the Halting problem

In this lecture, we will prove that certain languages are undecidable. There are two main techniques for doing so: the first is a technique called *diagonalization*, and the second is called *reductions.*

16.1 A simple undecidable language

We will start with the following language:

$$\textsc{Diag} = \{\langle M\rangle : M \text{ is a TM and } \langle M\rangle \notin L(M)\}.$$

This is the language of all encodings of Turing machines M which have the property that the encoding of M is not accepted by M. Informally, you can think of $\textsc{Diag}$ as the set of all strings which look like the code of a self-non-accepting computer program, where we define a program to be self-non-accepting if it does not return "accept" when run on its own source code.

We will show that $\textsc{Diag}$ is undecidable. To do this, suppose by contradiction that there was a Turing machine M that decides $\textsc{Diag}$. Now, what happens if we run M on the input $\langle M\rangle$ (that is, if we feed M its own source code as input)? Since M decides a language, it never loops, so it either accepts $\langle M\rangle$ or rejects it. So there are two cases: (1) if M accepts $\langle M\rangle$, then by definition $\langle M\rangle \in L(M)$, which means $\langle M\rangle \notin \textsc{Diag}$ by the definition of $\textsc{Diag}$. However, since M accepted $\langle M\rangle$, this contradicts the assumption that M decides $\textsc{Diag}$. (2) If M rejects $\langle M\rangle$, then by definition $\langle M\rangle \notin L(M)$, which means $\langle M\rangle \in \textsc{Diag}$ by the definition of $\textsc{Diag}$. However, since M rejected $\langle M\rangle$, this again contradicts the assumption that M decides $\textsc{Diag}$.

In both cases, we get a contradiction. This means a Turing machine M deciding $\textsc{Diag}$ could not have existed in the first place, so $\textsc{Diag}$ is undecidable.

This proof style, of feeding a program its own source code and getting a paradox, is called *diagonalization.* Proofs of this style also come up when proving that certain infinite sets are uncountable, as we saw in the first lecture.

Note that we can also conclude that $\textsc{Diag}$ is not recognizable. To see this, first note that $\textsc{Diag}$ is co-recognizable (that is, its complement is recognizable). This is because we can create a Turing machine M' which accepts any badly-formatted input, and for any input of the form $\langle M\rangle$ for a Turing machine M, the machine M' simulates M on input $\langle M\rangle$, and returns what $M(\langle M\rangle)$ returns (so $M'(\langle M\rangle)$ accepts if and only if $M(\langle M\rangle)$ accepts). Then it is not hard to check that M' accepts exactly the strings not in $\textsc{Diag}$, meaning that $L(M') = \overline{\textsc{Diag}}$ and that $\textsc{Diag}$ is co-recognizable.

Now, recall that if a language is both recognizable and co-recognizable, then it is decidable. Since we know that $\textsc{Diag}$ is not decidable, and since it is co-recognizable, we conclude is must not be recognizable.

16.2 The halting problem

Next, we will consider a somewhat more natural language called the halting problem. It is the language

$$H_{TM} = \{\langle\langle M\rangle, w\rangle : M \text{ is a TM and } M \text{ halts on } w\}.$$

Recall that a Turing machine M halts on w if it either accepts w or rejects w (in other words, M halts on w if and only if M does not loop on w).

We will show that the halting problem H_{TM} is undecidable. This was first proved by Alan Turing in 1936. The proof, which also uses diagonalization, is so elegant that it inspired Geoffrey K. Pullum turned it into a poem in the style of Dr. Seuss. Without further ado, here is his Seuss-style poem proving that the Halting problem is undecidable, titled *Scooping the Loop Snooper*. [1]

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports 'Bad!' – which means you're in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use – and it's simple to do.
I'll define a procedure, which I will call Q,
that will use P's predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what's the right thing to say
of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,

[1]Copyright © 2008, 2012 by Geoffrey K. Pullum. From the website: "Permission is hereby granted to reproduce or distribute this work for non-commercial, educational purposes relating to the teaching of computer science, mathematics, or logic, provided this attribution is included."

and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on *itself.*
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be –
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A *reductio*: There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!

16.3 Understanding the proof

Let's go through this poem-proof more slowly. The goal is to show that H_{TM}, the language of all (TM,input) pairs in which the TM eventually halts in the input, is undecidable. To do this, we assume by contradiction that there is some Turing machine P which decides this language; that is, P accepts inputs of the form $\langle\langle M\rangle, w\rangle$ where $M(w)$ halts, and rejects badly-formatted inputs as well as inputs $\langle\langle M\rangle, w\rangle$ where $M(w)$ loops.

The next step of the proof is to construct another Turing machine Q. Q takes as input a string s, which it interprets as a Turing machine $\langle A\rangle$; if the input s does not have this format, Q rejects. Q then makes two copies of the input string to form the string $\langle\langle A\rangle, \langle A\rangle\rangle$, and then simulates P on this input. Since we know P is a Turing machine, we know that Q can simulate it on any input of its choice – effectively this is the same as running P as a subroutine of Q, or making a function call to P in a normal programming language.

Finally, if $P(\langle\langle A\rangle, \langle A\rangle\rangle)$ accepts, the program Q proceeds to deliberately enter an infinite loop. On the other hand, if $P(\langle\langle A\rangle, \langle A\rangle\rangle)$ rejects, Q halts (either accepts or rejects, it doesn't matter).

We have now described the program Q. Next, as the poem says, "this program called Q wouldn't stay on the shelf; I would ask it to forecast its run on *itself.* When it reads its own source code, just what will it do? What's the looping behavior of Q run on Q?"

In our notation, we run $Q(\langle Q\rangle)$. What happens? First, note that this is a correctly formatted input. This means the first action of Q on this input is to create the string $\langle\langle Q\rangle, \langle Q\rangle\rangle$ and run P on it. Observe that from P's perspective, this is also a correctly-formatted input: it looks like a pair, the first element of which is an encoding of a Turing machine Q, and the second of which is a string $w = \langle Q\rangle$.

This means that (by definition) P will accept if $\langle\langle Q\rangle, \langle Q\rangle\rangle \in H_{TM}$, or in other words, if $Q(\langle Q\rangle)$ halts, and it will reject if $Q(\langle Q\rangle)$ loops. However, if P accepts – meaning that it thinks $Q(\langle Q\rangle)$ halts – then Q, which called it as a subroutine, deliberately starts to loop. On the other hand, if P rejects – meaning that it thinks $Q(\langle Q\rangle)$ loops – then Q, which called it as a subroutine, immediately halts. In both cases, we get a contradiction: it now seems impossible for $Q(\langle Q\rangle)$ to halt, and also impossible for $Q(\langle Q\rangle)$ to loop.

This contradiction completes the proof, ruling out the possibility of a Turing machine P which decides H_{TM}. Hence H_{TM} is undecidable.

Note that it is not too hard to see that H_{TM} is recognizable: to recognize this language, we just need a TM that accepts if the input TM halts, and we can do this simply by simulating the inputted TM and by accepting if it ever halts. This is enough to tell us that H_{TM} is not co-recognizable: that is, $\overline{H_{TM}}$ is not recognizable (since otherwise, if both H_{TM} and its complement were recognizable, then it would be decidable).

16.4 Proving undecidability using reductions

Apart from diagonalization, the other main way to prove that a language A is not decidable is to show that if A were decidable, then some other language B would also be decidable, where B is a language we know is undecidable. For example, consider the language A_{TM} from the previous lecture:

$$A_{TM} = \{\langle\langle M\rangle, w\rangle : M \text{ is a TM and } w \in L(M)\}.$$

We will now show that this language is undecidable by showing that if it was decidable, then H_{TM} would also be decidable. That is, we start with a hypothetical Turing machine M_A which decides A_{TM}, and we construct a Turing machine M_H which decides H_{TM}.

The TM M_H will work as follows. First, it will check that the input has the form $\langle\langle M\rangle, w\rangle$, rejecting otherwise. Next, M_H will run M_A on the input $\langle\langle M\rangle, w\rangle$, and accept if M_A accepts this input. Finally, M_H will modify the input Turing machine $\langle M\rangle$ by exchanging its accept and reject states, yielding the Turing machine $\langle M'\rangle$. It will then run M_A on the input $\langle\langle M'\rangle, w\rangle$, and accept if M_A accepts this input, rejecting otherwise.

To see why M_H decides the halting problem, first note that since M_A decides A_{TM}, it always halts; since M_H just calls M_A as a subroutine twice, it must also always halt.

Next, consider an input $\langle\langle M\rangle, w\rangle$ which is in H_{TM}. This means that $M(w)$ halts, so it either accepts or rejects. If it accepts, then $M_A(\langle\langle M\rangle, w\rangle)$ accepts, so M_H accepts the input $\langle\langle M\rangle, w\rangle$. On the other hand, if $M(w)$ rejects, then $M'(w)$ accepts (since M' is the same as M with accept and reject states flipped), and hence $M_A(\langle\langle M'\rangle, w\rangle)$ accepts, which means that M_H still accepts its input $\langle\langle M\rangle, w\rangle$.

Conversely, consider an input not in H_{TM}. If it is badly formatted, M_H immediately rejects it. Otherwise, if the input is $\langle\langle M\rangle, w\rangle$, then $M(w)$ loops. In this case, $M'(w)$ loops as well, since M never reaches its accept or reject state when run on w. Hence $M_A(\langle\langle M\rangle, w\rangle)$ and $M_A(\langle\langle M'\rangle, w\rangle)$ both reject, which means that $M_H(\langle\langle M\rangle, w\rangle)$ correctly rejects.

We conclude that M_H decides the language H_{TM}. Since this is impossible, the Turing machine M_A must not exist, meaning that A_{TM} is undecidable. Note that since A_{TM} is recognizable, we also conclude that it must not be co-recognizable (or else A_{TM} would be decidable).

16.4.1 A formal notion of reductions

In general, the word "reduction" just refers to showing that if you have an algorithm for one problem A, you can use it to construct an algorithm for a different problem B. We will now formalize one particular type of reduction, called a *mapping reduction.*

Before we can define a mapping reduction, we first need to talk about Turing machines that can *output* strings rather than just accepting or rejecting. We write the following definition.

Definition 16.1. *Let Σ and Γ be alphabets (nonempty finite sets), and let $f\colon \Sigma^* \to \Gamma^*$ be a function. We say that f is* computable *if there is a Turing machine M with input alphabet Σ and tape alphabet containing Γ that has the following property: for each $w \in \Sigma^*$, we have*

$$(q_0, \llcorner\!\lrcorner)w \vdash_M^* (q_{\text{acc}}, \llcorner\!\lrcorner)f(w),$$

where q_0 is the start state of M.

This definition is saying that f is computable if there is some Turing machine M that computes it. What we mean by a Turing machine M computing f is that M must accept every input string, and moreover, for every input string $w \in \Sigma^*$, $M(w)$ must write exactly the string $f(w)$ on its tape before accepting (and move the tape head to the left of the string $f(w)$ written on the tape). This is just one way to define what it means for a Turing machine to output a string; the details of this definition don't matter too much, because other ways of defining the output of a Turing machine will lead to equivalent definitions of computability.

Now that we've defined computable functions, we can define mapping reductions between languages.

Definition 16.2. *Let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be two languages. We say that A* (mapping) reduces *to B, denoted $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \to \Gamma^*$ such that for all $w \in \Sigma^*$, $w \in A \Leftrightarrow f(w) \in B$.*

In other words, a mapping reduction is a computable function which converts inputs to A (that is, strings in Σ^*) to inputs to B (that is, strings in Γ^*) such that yes-inputs for A map to yes-inputs for B, and no-inputs for A map to no-inputs for B. Remember that such a function must be *computable* in order to count as a mapping reduction.

If A reduces to B, then intuitively, A is easier to decide or recognize than B, which is why we write $A \leq_m B$. In fact, we have the following theorem.

Theorem 16.3. *Let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be two languages, and suppose that $A \leq_m B$. Then*

1. *If B is recognizable, then A is recognizable.*
2. *If B is co-recognizable, then A is co-recognizable.*
3. *If B is decidable, then A is decidable.*

Proof. Suppose B was recognizable. This means there is a Turing machine M_B which recognizes it. Moreover, since $A \leq_m B$, there is a mapping reduction f from A to B, which is computed by some Turing machine M_f. We now design a Turing machine M_A recognizing A. On input $w \in \Sigma^*$, the machine M_A will start by simulating M_f on w until it halts, obtaining $f(w)$ on the tape (recall that

M_f always halts by definition, since it computes a function f). Next, the machine M_A will simulate M_B on $f(w)$ and will accept if $M_B(f(w))$ ever accepts and reject if $M_B(f(w))$ ever rejects.

Note that if $w \in A$, then since f is a mapping reduction, $f(w) \in B$, which means that $M_B(f(w))$ must accept and hence $M_A(w)$ accepts. Conversely, if $w \notin A$, then $f(w) \notin B$, which means that $M_B(f(w))$ does not accept (it either rejects or loops) and hence $M_A(w)$ does not accept. We conclude that $L(M_A) = A$, so A is recognizable.

Next, observe that if $A \leq_m B$, then the mapping reduction f from A to B is also a mapping reduction from $\overline{A}$ to $\overline{B}$. That's because f is a function from Σ^* to Γ^* which has the property that $w \in A$ if and only if $f(w) \in B$, which also means $w \notin A$ if and only if $f(w) \notin B$, so f is automatically also a mapping reduction $\overline{A}$ to $\overline{B}$. Hence if B is co-recognizable, then $\overline{B}$ is recognizable, and since $\overline{A} \leq_m \overline{B}$, this also means that $\overline{A}$ is reconizable, so A is co-recognizable.

Finally, suppose B is decidable. In this case, B is both recognizable and co-recognizable. Since $A \leq_m B$, we've seen that A is also both recognizable and co-recognizable, which means that A is decidable. □

Notice that as part of this proof, we saw that

$$A \leq_m B \Rightarrow \overline{A} \leq_m \overline{B}.$$

Since the complement of the complement of A is A (and similarly for B), we also have $\overline{A} \leq_m \overline{B} \Rightarrow A \leq_m B$, so

$$A \leq_m B \Leftrightarrow \overline{A} \leq_m \overline{B}.$$

It is also the case that $A \leq_m A$ for every language A, since we can take the mapping reduction function f to be the identity function $f(w) = w$. A further useful property is transitivity: if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$. To see this, consider the mapping reduction f from A to B, and consider the mapping reduction g from B to C. Let h be the composition of f with g, so $h(w) = g(f(w))$ for all w. It is not hard to see that h is computable (to compute $h(w)$, first compute $f(w)$ and then compute g of the result). In addition, if $w \in A$, then $f(w) \in B$, so $g(f(w)) \in C$, and conversely, if $w \notin A$, then $f(w) \notin B$, and hence $g(f(w)) \notin C$. Hence h is a mapping reduction from A to C.

Mapping reductions can be used to show that languages are not recognizable or decidable. To show that A is not decidable, all we have to show is that $C \leq_m A$ for some language C that we already know is not decidable (for example, C might be H_{TM} or A_{TM}). This just means we need to find a function f from Σ^* (where Σ is the alphabet of C) to Γ^* (where Γ is the alphabet of A) such that:

1. f is computable,
2. if $w \in C$, then $f(w) \in A$,
3. if $w \notin C$, then $f(w) \notin A$.

When showing mapping reductions, remember to show all three of these steps! Also, make sure to give a reduction in the right direction, showing $C \leq_m A$ where C is a language you know is hard (e.g. A_{TM} or H_{TM}) and A is a language you want to prove is hard. Giving a reduction the wrong way – that is, showing $A \leq_m C$ instead of $C \leq_m A$ – is one of the most common mistakes students make.

16.4.2 A proof of undecidability via a mapping reduction

To give an example of mapping reductions, consider the following language:

$$\text{AE} = \{\langle M \rangle : M \text{ is a TM which accepts } \epsilon\}.$$

We will show that AE is not decidable. We will do this via a mapping reduction from A_{TM}.

To do this, we need a computable function f which maps inputs in A_{TM} to inputs in AE and inputs not in A_{TM} to inputs not in AE. We describe how to compute such a function f via a Turing machine M_f.

On input x, the machine M_f will first check whether x is formatted like $\langle\langle M\rangle, w\rangle$ where M is a Turing machine and w is an input string. If the input x to M_f does not have this form, then M_f will output some string not in AE (it doesn't matter which string; we can pick $\langle N\rangle$ where N is a Turing machine which always immediately rejects, for example. Then $\langle N\rangle \notin$ AE because $N(\epsilon)$ rejects).

Next, if the input to M_f has the form $\langle\langle M\rangle, w\rangle$, then M_f will create a new Turing machine $\langle M_w\rangle$ which first writes w on the tape, and then runs M on w. This can be done by "hard coding" w into the description of the Turing machine M_w; more explicitly, the machine M_w will have one state for each symbol of w, and from the start state, it will transition through all of these $|w|$ states and write to the tape a symbol of w each time. Once w is written on the tape, M_w will run M on the tape contents. This ensures that $M_w(\epsilon)$ behaves the same as $M(w)$.

We've now described the Turing machine M_f. Note that M_f always halts; it doesn't need to actually run the input machine $\langle M\rangle$ on anything, it just modifies its code to get $\langle M_w\rangle$ and outputs this new code. Hence M_f computes some function f. We claim that f is a mapping reduction from A_{TM} to AE.

To see this, consider a string $x \in A_{TM}$. Then $x = \langle\langle M\rangle, w\rangle$ for some TM M and some input string w. In this case, $f(x)$ is the string $\langle M_w\rangle$. Since $x \in A_{TM}$, we know that $M(w)$ accepts. Since $M_w(\epsilon)$ behaves the same as $M(w)$, we conclude taht $M_w(\epsilon)$ accepts, so $f(x) = \langle M_w\rangle \in$ AE.

Conversely, suppose that $x \notin A_{TM}$. Then if x is badly formatted, we know that $f(x) \notin$ AE, by design. If $x = \langle\langle M\rangle, w\rangle$, then $f(x) = \langle M_w\rangle$. In this case, we know that $M(w)$ does not accept (since $\langle\langle M\rangle, w\rangle \notin A_{TM}$), which means that $M_w(\epsilon)$ does not accept. Hence $f(x) \notin$ AE. This completes the proof, showing that $A_{TM} \leq_m$ AE. Since we know that A_{TM} is not decidable, we conclude that AE is also not decidable.

16.5 Next steps

A major theme of the rest of the course will be these two lines from Pullum's poem:

> You can never find general mechanical means
> for predicting the acts of computing machines.

Actually, we should amend this statement slightly: there are no general mechanical means for predicting the acts of computing machines *better than just simulating the machine.*

This theme will come up repeatedly in the study of computability: roughly speaking, every question of the from 'will the input machine do X" will be undecidable, unless you can figure it out just by simulating the input machine. This theme will come up again when we study time-bounded computation as well, in the last unit of the course.