Lecture 22: Reductions

16 April 2009

1 What is a reduction?

Last lecture we proved that A_{TM} is undecidable. Now that we have one example of an undecidable language, we can use it to prove other problems to be undecidable.

Meta definition: Problem A ***reduces*** to problem B, if given a solution to B, then it implies a solution for A. Namely, we can solve B then we can solve A. We will done this by A $\implies$ B.

An ***oracle*** ORAC for a language L is a function that receives as a word w, and it returns true if and only if $w \in L$. An oracle can be thought of as a black box that can solve membership in a language without requiring us to consider the question of whether L is computable or not. Alternatively, you can think about an oracle as a provided library function that computes whatever it requires to do, and it always return (i.e., it never goes into an infinite loop).

Intuitively, a TM decider for a language L is the ultimate oracle. Not only it can decide if a word is in L, but furthermore, it can be implemented as a TM that always stops.

In the context of showing languages are undecidable, the following more specific definition would be useful.

Definition 1.1 A language X ***reduces*** to a language Y, if one can construct a TM decider for X using a given oracle ORAC_Y for Y.

We will denote this fact by $X \implies Y$.

In particular, if X reduces to Y then given a decider for the language Y (i.e., an oracle for Y), then there is a program that can decide X. So Y must be at least as "hard" as X. In particular, if X is undecidable, then it must be that Y is also undecidable.

Warning. It is easy to get confused about which of the two problems "reduces" to the other. Do not get hung up on this. Instead, concentrate on getting the right outline for your proofs (proving them in the right direction, of course).

Reduction proof technique. Formally, consider a problem B that we would like to prove is undecidable. We will prove this via reduction, that is a proof by contradiction, similar in outline to the ones we have seen for regular and context-free languages. You assume that your new language L (i.e., the language of B) is decided by some TM M. Then you use M as a component to create a decider for some language known to be undecidable (typically A_{TM}). This is would imply that we have a decider for A (i.e., A_{TM}). But this is a contradiction

since A (i.e., A_{TM}) is not decidable. As such, we must have been wrong in assuming that L was decidable.

We will concentrate on using reductions to show that problems are undecidable. However, the technique is actually very general. Similar methods can be used to show problems to be not TM recognizable. We have used similar proofs to show languages to be not regular or not context-free. And reductions will be used in CS 473 to show that certain problems are "NP complete", i.e. these problems (probably) require exponential time to solve.

1.1 Formal argument

Lemma 1.2 *Let X and Y be two languages, and assume that $X \implies Y$. If Y is TM decidable then X is TM decidable.*

Proof: Let $\mathtt{T}$ be the TM decider for Y. Since X reduces to Y, it follows that there is a procedure $\mathtt{T}_{X|Y}$ (i.e., TM decider) for X that uses an oracle for Y as a subroutine. We replace the calls to this oracle in $\mathtt{T}_{X|Y}$ by calls to $\mathtt{T}$. The resulting TM $\mathtt{T}_X$ is a TM decider and its language is X. Thus X is TM decidable. ■

The counter-positive of this lemma, is what we will use.

Lemma 1.3 *Let X and Y be two languages, and assume that $X \implies Y$. If X is TM undecidable then Y is TM undecidable.*

2 Halting

We remind the reader that A_{TM} is the language

$$\mathrm{A}_{\mathsf{TM}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a TM and } M \text{ accepts } w \right\}.$$

This is the problem that we showed (last class) to be undecidable (via diagonalization). Right now, it is the only problem we officially know to be undecidable.

Consider the following slight modification, which is all the pairs $\langle M, w \rangle$ such that M ***halts*** on w. Formally,

$$A_{\mathrm{Halt}} = \left\{ \langle M, w \rangle \;\middle|\; M \text{ is a TM and } M \text{ stops on } w \right\}.$$

Intuitively, this is very similar to A_{TM}. The big obstacle to building a decider for A_{TM} was deciding whether a simulation would ever halt or not.

To show formally that A_{Halt} is undecidable, we show that we can use a oracle for A_{Halt} to build a decider for A_{TM}. This construction looks like the following.

Lemma 2.1 *The language A_{TM} reduces to A_{Halt}. Namely, given an oracle for A_{Halt} one can build a decider (that uses this oracle) for A_{TM}.*

Proof: Let ORAC_{Halt} be the given oracle for A_{Halt}. We build the following decider for A_{TM}.

Decider-A_{TM} $\Big(\langle M, w\rangle\Big)$

$res \leftarrow \mathsf{ORAC}_{Halt}\Big(\langle M, w\rangle\Big)$
`// if` M `does not halt on` w `then reject.`
if $res =$ reject **then**
  halt and reject.

`//` M `halts on` w `since` $res =$accept.
`// Thus, simulating` M `on` w `would terminate in finite time.`
$res_2 \leftarrow$Simulate M on w (using U_{TM}).

return res_2.

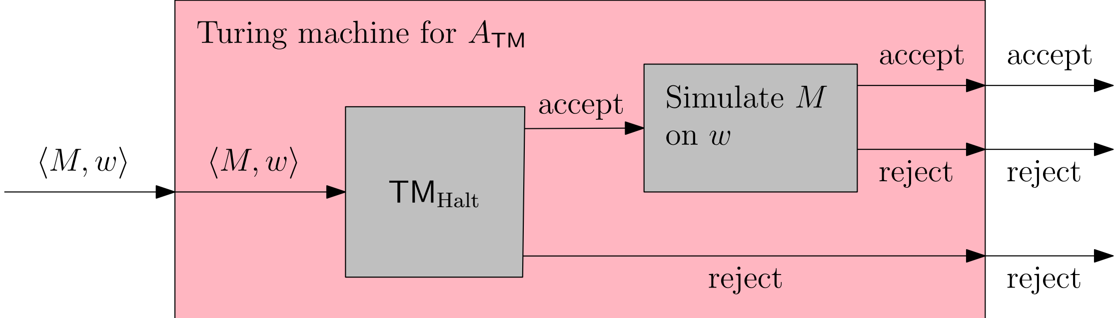
Clearly, this procedure always return and as such its a decider for A_{TM}. ■

Theorem 2.2 *The language A_{Halt} is not decidable.*

Proof: Assume, for the sake of contradiction, that A_{Halt} is decidable. As such, there is a TM, denoted by $\mathsf{TM}_{\mathrm{Halt}}$, that is a decider for A_{Halt}. We can use $\mathsf{TM}_{\mathrm{Halt}}$ as an implementation of an oracle for A_{Halt}, which would imply by Lemma 2.1 that one can build a decider for A_{TM}. However, A_{TM} is undecidable. A contradiction. It must be that A_{Halt} is undecidable. ■

We will be usually less formal in our presentation. We will just show that given a TM decider for A_{Halt} implies that we can build a decider for A_{TM}. This would imply that A_{TM} is undecidable.

Thus, given a black box (i.e., decider) $\mathsf{TM}_{\mathrm{Halt}}$ that can decide membership in A_{Halt}, we build a decider for A_{TM} is follows.



This would imply that if A_{Halt} is decidable, then we can decide A_{TM}, which is of course impossible.

3 Emptiness

Now, consider the language

$$E_{\mathsf{TM}} = \Big\{ \langle M\rangle \;\Big|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) = \emptyset \Big\}.$$

Again, we assume that we have a decider for E_{TM}. Let us call it $\mathsf{TM}_{E_{\mathsf{TM}}}$. We need to use the component $\mathsf{TM}_{E_{\mathsf{TM}}}$ to build a decider for A_{TM}.

A decider for A_{TM} is given M and w and must decide whether M accepts w. We need to restructure this question into a question about some Turing machine having an empty language. Notice that the decider for E_{TM} takes only one input: a Turing machine. So we have to somehow make the second input (w) disappear.

The key trick here is to hard-code w into M, creating a TM M_w which runs M on the fixed string w. Specifically the code for M_w might look like:

TM M_w:

1. Input $= x$ (which will be ignored)
2. Simulate M on w.
3. If the simulation accepts, accept. If the simulation rejects, reject.

Its important to understand what is going on. The input is $\langle M \rangle$ and w. Namely, a string encoding M and a the string w. The above shows that we can write a procedure (i.e., TM) that accepts this two strings as input, and outputs the string $\langle M_w \rangle$ which encodes M_w. We will refer to this procedure as **EmbedString**. The algorithm **EmbedString**($\langle M, w \rangle$) as such, is a procedure reading its input, which is just two strings, and outputting a string that encodes the TM $\langle M_w \rangle$.

It is natural to ask, what is the language of the machine encoded by the string $\langle M_w \rangle$; that is, what is $L(M_w)$?

Because we are ignoring the input x, the language of M_w is either Σ^* or $\emptyset$. It is Σ^* if M accepts w, and it is $\emptyset$ if M does not accept w.

We are now ready to prove the following theorem.

Theorem 3.1 *The language E_{TM} is undecidable.*

Proof: We assume, for the sake of contradiction, that E_{TM} is decidable, and let $\mathsf{TM}_{E_{\mathsf{TM}}}$ be its decider. Next, we build our decider **AnotherDecider-A_{TM}** for A_{TM}, using the **EmbedString** procedure described above.

```
AnotherDecider-A_TM(⟨M, w⟩)
        ⟨M_w⟩ ← EmbedString (⟨M, w⟩)
        r ← TM_{E_TM}(⟨M_w⟩).
        if r = accept then
                reject.

        // TM_{E_TM}(⟨M_w⟩) rejected its input

        return accept
```

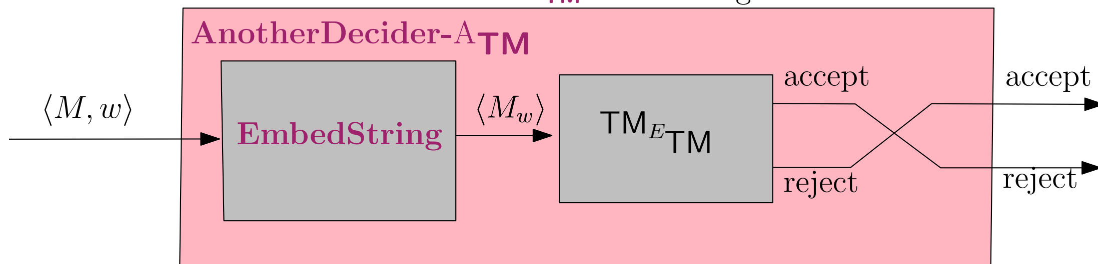
Consider the possible behavior of **AnotherDecider-A_{TM}** on the input $\langle M, w \rangle$.

- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is empty. This implies that M does not accept w. As such, **AnotherDecider-A_{TM}** rejects its input $\langle M, w \rangle$.
- If $\mathsf{TM}_{E_{\mathsf{TM}}}$ accepts $\langle M_w \rangle$, then $L(M_w)$ is not empty. This implies that M accepts w. So **AnotherDecider-A_{TM}** accepts $\langle M, w \rangle$.

Namely, **AnotherDecider-A_{TM}** is indeed a decider for A_{TM}, (its a decider since it always stops on its input). But we know that A_{TM} is undecidable, and as such it must be that our assumption that E_{TM} is decidable is false. ■

In the above proof, note that **AnotherDecider-A_{TM}** is indeed a decider, so it always halts, either accepting or rejecting. By contrast, M_w might not always halt. So, when we do our analysis, we need to think about what happens if M_w never halts. In this example, if M never halts on w, then w will be treated just like the explicit rejection cases and this is what we want.

Here is the code for **AnotherDecider-A_{TM}** in flow diagram form.



Observe, that **AnotherDecider-A_{TM}** never actually runs the code for M_w. It hands the code to a function $\mathsf{TM}_{E_{\mathsf{TM}}}$ which analyzes what the code would do if we ever did choose to run it. But we never run it. So it does not matter that M_w might go into an infinite loop.

Also notice that we have two input strings floating around our code: w (one input to the decider for A_{TM}) and x (input to M_w). Be careful to keep track of which strings are input to which functions. Also be careful about how many inputs, and what types of inputs, each function expects.

4 Equality

An easy corollary of the undecidability of E_{TM} is the undecidability of the language

$$EQ_{\mathsf{TM}} = \left\{ \langle M, N \rangle \;\middle|\; M \text{ and } N \text{ are } \mathsf{TM}\text{'s and } L(M) = L(N) \right\}.$$

Lemma 4.1 *The language EQ_{TM} is undecidable.*

Proof: Suppose that we had a decider **DeciderEqual** for EQ_{TM}. Then we can build a decider for E_{TM} as follows:

TM R:

1. Input $= \langle M \rangle$

2. Include the (constant) code for a TM T that rejects all its input. We denote the string encoding T by $\langle T \rangle$.
3. Run **DeciderEqual** on $\langle M, T \rangle$.
4. If **DeciderEqual** accepts, then accept.
5. If **DeciderEqual** rejects, then reject.

■

Since the decider for E_{TM} (i.e., $\mathsf{TM}_{E_{\mathsf{TM}}}$) takes one input but the decider for EQ_{TM} (i.e. **DeciderEqual**) requires two inputs, we are tying one of **DeciderEqual**'s input to a constant value (i.e., T).

There are many Turing machines that reject all their input and could be used as T. Building code for R just requires writing code for one such TM.

5 Regularity

It turns out that almost any property defining a TM language induces a language which is undecidable, and the proofs all have the same basic pattern. Let us do a slightly more complex example and study the outline in more detail.

Let

$$\text{Regular}_{\mathsf{TM}} = \left\{ \langle M \rangle \;\middle|\; M \text{ is a } \mathsf{TM} \text{ and } L(M) \text{ is regular} \right\}.$$

Suppose that we have a TM **DeciderRegL** that decides $\text{Regular}_{\mathsf{TM}}$. In this case, doing the reduction from halting, would require to turn a problem about deciding whether a TM M accepts w (i.e., is $w \in \mathrm{A}_{\mathsf{TM}}$) into a problem about whether some TM accepts a regular set of strings.

Given M and w, consider the following TM M'_w:

TM M'_w:

(i) Input $= x$

(ii) If x has the form $\mathtt{a}^n\mathtt{b}^n$, halt and accept.

(iii) Otherwise, simulate M on w.

(iv) If the simulation accepts, then accept.

(v) If the simulation rejects, then reject.

Again, we are ***not*** going to execute M'_w directly ourself. Rather, we will feed its description $\langle M'_w \rangle$ (which is just a string) into **DeciderRegL**. Let **EmbedRegularString**denote this algorithm, which accepts as input $\langle M \rangle$ and w, and outputs $\langle M'_w \rangle$, which is the encoding of the machine M'_w.

If M accepts w, then every input x will eventually be accepted by the machine M'_w. Some are accepted right away and some are accepted in step (i). So if M accepts w then the language of M'_w is Σ^*.

If M does not accept w, then some strings x (that are of the form $\mathtt{a}^n\mathtt{b}^n$) will be accepted in step (ii) of M'_w. However, after that, either step (iii) will never halt or step (iv) will reject. So the rest of the strings (that are in the set $\Sigma^* \setminus \left\{ \mathtt{a}^n\mathtt{b}^n \;\middle|\; n \geq 0 \right\}$) will not be accepted. So the language of M'_w is $\mathtt{a}^n\mathtt{b}^n$ in this case.

Since $\mathtt{a}^n\mathtt{b}^n$ is not regular, we can use our decider **DeciderRegL** on M'_w to distinguish these two cases.

Notice that the test in step (ii) was cooked up specifically to match the capabilities of our given decider **DeciderRegL**. If **DeciderRegL** had been testing whether our language contained the string "uiuc", step (ii) would be comparing x to see if it was equal to "uiuc". This test can be anything that a TM can compute without the danger of going into an infinite loop.

Specifically, we can build a decider for A_{TM} as follows.

YetAnotherDecider-$A_{\mathsf{TM}}(\langle M, w\rangle)$
    $\langle M'_w \rangle \leftarrow$ **EmbedRegularString** $(\langle M, w\rangle)$
    $r \leftarrow$ **DeciderRegL**$(\langle M'_w\rangle)$.
    return r

The reason why **YetAnotherDecider-**A_{TM} does the right thing is that:

— If **DeciderRegL** accepts, then $L(M'_w)$ is regular. So it must be Σ^*. This implies that M accepts w. So **YetAnotherDecider-**A_{TM} should accept $\langle M, w\rangle$.

— If **DeciderRegL** rejects, then $L(M'_w)$ is not regular. So it must be $\mathtt{a}^n\mathtt{b}^n$. This implies that M does not accept w. So **YetAnotherDecider-**A_{TM} should reject $\langle M, w\rangle$.

6 Windup

Notice that the code in Section 5 is almost exactly the same as the code for the E_{TM} example in Section 3. The details of M_w and M'_w were different. And one example passed on the return values from **YetAnotherDecider-**A_{TM} directly, whereas the other example negated them. This similarity is not accidental, as many examples can be done with very similar proofs.

Next class, we will see Rice's Theorem, which uses this common proof template to show a very general result. Namely, almost any nontrivial property of a TM's language is undecidable.