ITCS 532:
4. Undecidable Problems

Rob Egrot

Undecidable Problems

- In the previous class we saw that there are undecidable problems.
- This was based on a cardinality argument.
- The set of formal languages for a finite alphabet is uncountable.
- But the set of Turing machines for this language is countable.
- Since every recursively enumerable language requires a TM to semidecide it, and no TM semidecides more than one language, there must be languages that are not r.e.
- So there are undecidable problems.
- But can we find an actual undecidable problem?

The Halting Problem

- Given a Turing machine T and an input I, either $T(I)$ halts or it runs forever.
- One must happen, and both cannot be true at the same time.
- So the question of whether a Turing machine T halts on input I is a *decision problem* whose instances are pairs (T, I).
- We know that we can encode pairs (T, I) of Turing machines and inputs over the alphabet $\{0, 1\}$.
- So, the question is, can we create a Turing machine that decides this problem?
- Is there a TM that takes as input $\mathbf{code}(T, I)$ and accepts if $T(I)$ halts, and rejects if it does not (or if its input is not the code for a TM and an input)?

If the Halting Problem were decidable...

- If a Turing machine H exists that can decide the halting problem then, for one thing, it would imply that r.e. languages are recursive.
- Why?

Why is the Halting Problem undecidable

- How can we prove the Halting Problem is undecidable?
- We will look for a contradiction.
- So suppose H exists. I.e., suppose for any input J we have $H(J)$ accepts if $J = \mathbf{code}(T, I)$ for some Turing machine T and input I such that $T(I)$ halts, and rejects otherwise.
- What are the consequences of this?

Self Reference

- There are many logical paradoxes arising from 'self reference'.
- E.g. 'the set of all sets that don't contain themselves'.
- Can we use this here?
- Consider the machine H' such that $H'(\textbf{code}(T))$ accepts iff $T(\textbf{code}(T))$ halts, and rejects otherwise.
- Note that if H exists, then H' also exists, as we can run $H(\textbf{code}(T, \textbf{code}(T)))$.
- What happens if we run H' on $\textbf{code}(H')$?
- H' always halts, so $H'(\textbf{code}(H'))$ accepts. No problem here.

A Variation

- Define a new machine H''.
- This machine is based on H' but it halts if $T(\mathbf{code}(T))$ *doesn't* halt, and it loops forever if $T(\mathbf{code}(T))$ halts.
- This modification is easy to make; it's like the one used to prove that recursive languages are recursively enumerable.

$$H''(\mathbf{code}(T)) = \begin{cases} \textit{Halt} \text{ if } T(\mathbf{code}(T)) \text{ doesn't halt} \\ \textit{Loop forever} \text{ if } T(\mathbf{code}(T)) \text{ halts} \end{cases}$$

In addition, $H''(I)$ halts if I is not the coded form of a TM.

Self Reference Again

- What happens when we run H'' on its own code.
- I.e. what is $H''(\mathbf{code}(H''))$?
- According to the definition, $H''(\mathbf{code}(H''))$ halts if and only if $H''(\mathbf{code}(H''))$ doesn't halt.

$$H''(\mathbf{code}(H'')) = \begin{cases} \textit{Halt} \text{ if } H''(\mathbf{code}(H'')) \text{ doesn't halt} \\ \textit{Loop forever} \text{ if } H''(\mathbf{code}(H'')) \text{ halts} \end{cases}$$

- This is a clear contradiction, so we must conclude that H'' cannot exist. But H'' is a simple modification of H, so H cannot exist either. This gives us:

Theorem 1
The halting problem is undecidable.

The Halting Problem *is* Semidecidable

- The Halting Problem is semidecidable.
- Why?
- As a consequence, the set of recursive languages is strictly contained in the set of $r.e.$ languages.
- Because the formal language containing strings $\mathbf{code}(T, I)$ (in some alphabet) such that $T(I)$ halts is r.e. but not recursive.

Languages That Are Not R.E.

- The Halting problem is semidecidable but not decidable, so its associated language is r.e. but not recursive.
- We know from the countable vs uncountable argument that there are languages that are not r.e.
- Can we find an example?
- It turns out the answer is yes.
- First some facts about recursive and r.e. languages.

Theorem 2

If L is a recursive formal language then the complement language $\bar{L}$ is also recursive.

Proof.

- If L is recursive then there's a Turing machine T that accepts if $I \in L$ and rejects if $I \notin L$.
- To get a machine that decides $\bar{L}$ we just need to swap the 'accept' and 'reject' states of T.

$\square$

Theorem 3

Let L be a formal language over a finite alphabet. If L is r.e. and the complement language $\bar{L}$ is also r.e. then L is recursive.

Proof.

- If there are machines T_1 and T_2 that semidecide L and $\bar{L}$ respectively then we can construct a machine T to decide L as follows.
- Given input I we simulate $T_1(I)$ and $T_2(I)$.
- Either $I \in L$ or $I \in \bar{L}$ so one of $T_1(I)$ and $T_2(I)$ will halt.
- If $T_1(I)$ halts T accepts, if $T_2(I)$ halts then T rejects.

□

Theorem 4

Let L_1 and L_2 be formal languages over a finite alphabet.

1. *If L_1 and L_2 are recursive then $L_1 \cup L_2$ and $L_1 \cap L_2$ are recursive.*
2. *If L_1 and L_2 are r.e. then $L_1 \cup L_2$ and $L_1 \cap L_2$ are r.e.*

Proof.

- Let T_1 and T_2 be machines that decide L_1 and L_2 respectively.
- Simulating T_1 and T_2 on any input I.
- If either $T_1(I)$ or $T_2(I)$ accepts then $I \in L_1 \cup L_2$. On the other hand, if both reject then $I \notin L_1 \cup L_2$.
- If both accept then $I \in L_1 \cap L_2$, but if one rejects then $I \notin L_1 \cap L_2$.
- Similarly if T_1 and T_2 are machines that semidecide L_1 and L_2 then $I \in L_1 \cup L_2$ if and only if either $T_1(I)$ or $T_2(I)$ halts, and $I \in L_1 \cap L_2$ if and only if $T_1(I)$ and $T_2(I)$ both halt.

$\square$

Three languages

Definition 5 (SA - 'self accepting')

SA is the language over $\{0, 1\}$ that contains the codes of all Turing machines T such that $T(\mathbf{code}(T))$ halts.

Definition 6 (NSA - 'not self accepting')

NSA is the language over $\{0, 1\}$ that contains the codes of all Turing machines T such that $T(\mathbf{code}(T))$ does not halt.

Definition 7 (NSA')

NSA' is NSA together with all the strings that are not codes for Turing machines. So $NSA' = \overline{SA}$.

The Halting Problem and SA

- SA contains all the yes instances of the modified Halting Problem from earlier.
- We showed there is no TM that decides this problem.
- So SA is not recursive (but is r.e. as we can design a machine to semidecide it).
- So...

Theorem 8
NSA' is not r.e.

Proof.
Since SA is r.e., if NSA' is also r.e. then SA is recursive by theorem 3 (as $NSA' = \overline{SA}$). But SA is not recursive, so NSA' cannot be r.e. □

Application to NSA

Corollary 9

NSA is not r.e.

Proof.

- By the definition of coding there is an algorithm that says whether a string x is or is not the coded form of a Turing machine.
- So if NSA was r.e. then we could combine this with the algorithm semideciding NSA:
    1. Check if input string x is the code of a Turing machine. If no then halt, if yes go to step 2.
    2. Run algorithm semideciding NSA. If this halts then halt, otherwise just keep going.
- This algorithm semidecides NSA', which is a contradiction because we know NSA' is not r.e.

□

A Direct Argument for *NSA*

We could also prove that *NSA* is not r.e. directly, using similar logic to what we used to show the halting problem is not decidable.

Theorem 10
NSA is not r.e. (without using the fact that the modified halting problem is undecidable).

Proof.

- Suppose there is a machine M that semidecides *NSA*.
- So $M(\textbf{code}(T))$ halts if $T(\textbf{code}(T))$ does not halt, and runs forever if $T(\textbf{code}(T))$ halts.
- So by definition $M(\textbf{code}(M))$ halts if and only if $M(\textbf{code}(M))$ does not halt, which is a contradiction.

$\square$

Another Argument for NSA'

Corollary 11

NSA' is not r.e.

Proof.

If NSA' were r.e. then we could use the following algorithm on input string x:

1. Run the algorithm that semidecides NSA'. If this halts then go to the next step.
2.
   - Run the algorithm that decides if x is the code of a Turing machine.
   - If x is the code of a Turing machine then it must not be self-accepting, as it's in NSA'.
   - This means it's in NSA, so we should halt.
   - If x is not the code of a Turing machine then we should instead go into an infinite loop.

This algorithm semidecides NSA, which would contradict the fact that NSA is not r.e. □

Another Argument for SA

We can also use previously proved results to show that SA is not recursive without referencing the halting problem.

Theorem 12

SA is not recursive.

Proof.

If SA is recursive then $\overline{SA} = NSA'$ is recursive by theorem 2. This would contradict corollary 11. $\square$

Back to the Halting Problem

- We have found two conceptually straightforward languages, one of which is recursive but not r.e. (SA), and the other which is not even r.e. (NSA).
- The arguments are perhaps easier to understand than the somewhat complicated definition of H'' used in the proof that the Halting problem is not recursive.
- In particular the proof that NSA is not r.e. from theorem 10 uses self-reference to get a contradiction in a much more direct way than the proof that the Halting problem is undecidable does.
- However, the Halting problem is independently interesting, and historically important, which is why we discuss it in some detail.

The Empty Tape Halting Problem

- We know that the halting problem is undecidable.
- That is, there is no Turing machine H that can act on $\mathbf{code}(T, I)$ for another Turing machine T and accept if $T(I)$ halts and reject if it does not.
- Consider the following decision problem.

Definition 13 (empty tape halting problem)

Is there a Turing machine E that given $\mathbf{code}(T)$ for another Turing machine T will accept if T halts on the empty string and will reject otherwise?

Solving the Empty Tape Halting Problem

- Suppose this machine E exists.
- Consider a Turing machine T and an input I.
- Using T and I we design a new Turing machine T_I that does the following on input J.
  1. T_I erases input J from the tape.
  2. T_I simulates T on I.
- What happens if we run E on $\textbf{code}(T_I)$?

$$E(\textbf{code}(T_I)) \text{ accepts } \iff T_I \text{ halts on empty input } \iff T(I) \text{ halts.}$$

$$E(\textbf{code}(T_I)) \text{ rejects } \iff T_I \text{ does not halt on empty input } \iff T(I) \text{ does not halt.}$$

- This looks a lot like solving the halting problem, which we know is impossible.
- This can't happen so we will conclude that E should not exist.

Formal Version

- We want to formalize the argument we just made to show E can't exist.
- First, assume E solving the ETHP exists.
- Construct a TM H as follows:

1. Given input $\mathbf{code}(T, I)$ the first thing H does is construct $\mathbf{code}(T_I)$. This is complicated but it can be done.
2. H then simulates $E(\mathbf{code}(T_I))$.
3. $H(\mathbf{code}(T, I))$ accepts $\iff E(\mathbf{code}(T_I))$ accepts $\iff T(I)$ halts, and rejects otherwise.

- H is a well defined Turing machine, and H solves the halting problem.
- The only questionable step in the construction of H is the assumption that E exists.
- Since H cannot exist we conclude that E cannot exist either, and so the empty tape halting problem is also undecidable.

Reduction

This is an example of a general technique called *reduction*. The general strategy is as follows.

1. Start with a decision problem D whose decidability is not known, and a decision problem U that is known to be undecidable (e.g. the halting problem).
2. Show that instances I_U of U can be converted by an algorithm to instances I_D of D so that
   - I_D is a yes instance of $D \iff I_U$ is a yes instance of U.
   - I_D is a no instance of $D \iff I_U$ is a no instance of U.
3. If there is an algorithm for solving D then we could combine this with our instance converting algorithm to find an algorithm for solving U.
4. Since U is undecidable no algorithm for solving U exists, and we conclude that an algorithm for solving D cannot exist either.

Ordering by Hardness

Definition 14 (reducible)

A is reducible to B if there is an algorithm that converts instances I_A of A to instances I_B of B so that I_B is a yes instance of B if and only if I_A is a yes instance of A, and I_B is a no instance of B if and only if I_A is a no instance of A.

Definition 15 ($A \leq B$)

Given decision problems A and B we write $A \leq B$ if A is reducible to B.

- Informally we think of this as meaning that B is at least as hard as A.
- I.e. if we had a solution for B we could use it to solve A, but we wouldn't necessarily be able to use a solution for A to solve B.
- With U and D from the previous slide we would write $U \leq D$.

A Hierarchy of Decision Problems

- This produces a kind of hierarchy of problems (and formal languages), ordered by their relative decidability.
- It turns out that all the decidable problems are grouped together at the bottom of this hierarchy.

Theorem 16

Let D be a decidable problem, and let A be any other decision problem with at least one yes instance and at least one no instance. Then $D \leq A$.

Proof.

- Since D is decidable, given an instance I_D of D we can find out whether it is 'yes' or 'no' by running the decision algorithm for D.
- If I_D is a 'yes' we convert it to a 'yes' of A, and if not we convert it to a 'no' of A.
- So $D \leq A$.

□

Basic Properties of $\leq$

- The $\leq$ relation for decision problems is reflexive (i.e. $A \leq A$), and transitive (i.e. $A \leq B$ and $B \leq C \implies A \leq C$).
- Why?
- $\leq$ is not symmetric.
- Why?

Halts for all Inputs (HAI)

- Suppose we have the code of a Turing machine T.
- Is there an algorithm that we can run on $\textbf{code}(T)$ that tells us whether or not $T(I)$ halts for all inputs I?
- The answer is no, and one of the exercises for this section is to prove this by reducing the Halting problem to this problem.
- I.e. to show that $HP \leq HAI$.

Equivalence of Turing machines (ETM)

- Suppose we have the code for two Turing machines.
- Is there an algorithm we can use that will tell us if they do the same thing for all inputs (i.e. if $T_1(I) = T_2(I)$ for all I)?
- It turns out the answer is no, and we can prove this using the reduction technique.
- We will show that $HAI \leq ETM$.

$HAI \leq ETM$

- An instance of HAI is just a Turing machine T, and T is a yes instance if and only if $T(I)$ halts for all I.
- We want to use T to construct T_1 and T_2 so that $T(I)$ halts for all I if and only if $T_1(I) = T_2(I)$ for all I.
- We construct T_1 so that it operates as follows:
  1. $T_1(I)$ simulates $T(I)$.
  2. If $T(I)$ halts then $T_1(I)$ erases the tape then writes a single '1' before halting.
- We construct T_2 so that $T_2(I)$ just writes a single '1' on the tape before halting for all I.
- T is a yes instance of HAI if and only if $T(I)$ halts for all I.
- This happens if and only if $T_1(I)$ writes a single '1' for all I.
- But $T_1(I) = 1$ for all I if and only if $T_1(I) = T_2(I)$ for all I.
- So T is a 'yes' instance of HAI if and only if (T_1, T_2) is a yes instance of ETM.
- So $HAI \leq ETM$ and so ETM is undecidable.