CSE396 Lecture Thu. 4/22: Undecidability

Picking up from before, we defined the Acceptance Problem for Turing machines:

A_{TM} :
INST: A deterministic Turing machine M and an input w to M.
QUES: Does M accept w?

Sometimes regarded same as the Halting Problem, but we prefer to define that separately:

HP_{TM} :
INST: A deterministic Turing machine M and an input w to M.
QUES: Does $M(w) \downarrow$?

The A_{TM} language is $\{\langle M, w\rangle : w \in L(M)\}$. There is indeed a Turing machine that accepts (exactly) it: $A_{TM} = L(U)$ where U is any universal Turing machine, such as the one in Tuesday's lecture. But recall that U is not total: whenever $M(w) \uparrow$, $U(\langle M, w\rangle) \uparrow$ too. We will see that there is no decider for A_{TM}. The text states and proves this directly, but we will get there by a parallel road: the following defines the "diagonal language" in place of the text's "diagonal machine D."

D_{TM} :
INST: A deterministic Turing machine M.
QUES: Does M **not** accept its own code $\langle M\rangle$?

The language is $D_{TM} = \{\langle M\rangle : M \textit{ does not accept } \langle M\rangle\}$. Note that the case $M(\langle M\rangle) \uparrow$, that is, M not halting on its own code, counts as $\langle M\rangle$ being **in** the language D_{TM} even though you can't immediately "register" that condition.

Theorem: The language D_{TM} is not **c.e.**---that is, there does not exist a TM Q such that $L(Q) = D_{TM}$.

I am using the letter Q in a new way, to refer to a whole machine rather than its set of states, in order to reinforce the point that this machine *does not actually exist* although the proof involves talking about it as if it did. We can say Q is *quixotic*, after Don Quixote.

Proof. Suppose such a Q existed. Then it would have a string code $q = \langle Q\rangle$. Then we could run Q on input q. The logical analysis of that run, on hypothesis $L(Q) = D_{TM}$, is:

$$\begin{aligned} Q \textit{ accepts } q &\iff q \textit{ is in } D_{TM} && \text{by } L(Q) = D_{TM} \\ &\iff Q \textit{ does not accept } q && \text{by definition of } q \in D_{TM}. \end{aligned}$$

The analysis makes a statement equivalent to its negation, which is a "logical rollback" condition. The rollback goes all the way to the first sentence of the proof. So such a Q cannot exist. ☒

It is worth reworking this proof in several ways. One is to follow the chain of implications in both directions like a cat chasing its tail:

> "If Q accepts its own code q, then $q \in L(Q)$. But $L(Q) = D_{TM}$, which by definition is the language of codes of machines that do *not* accept their own code. So Q must not accept its own code q. But then q meets the definiton for being in D_{TM}. Since we have $D_{TM} = L(Q)$ to begin with, this means Q accepts its own code q. But if Q accepts its own code q, then..."

Another help is to compare with an abstract proof about sets. Consider functions f whose arguments are elements of a set A and whose outputs are subsets of A. The $\underline{\delta}(p, c)$ function from an NFA becomes such a function when you fix the char c. Thus we write $f : A \to \mathcal{P}(A)$ where $\mathcal{P}$ denotes the power set. Then f being *onto* would mean that every subset of A is a value of f on some argument(s). But we have:

Theorem: No function $f : A \to \mathcal{P}(A)$ can ever be onto $\mathcal{P}(A)$.

Proof: Suppose we have a function $f : A \to \mathcal{P}(A)$. Then we *do* have the subset

$$D = \{a \in A : a \text{ is not in the set } f(a)\}.$$

By f being onto, there would exist $d \in A$ such that $f(d) = D$. But then:

$$\begin{aligned} d \in D &\iff d \text{ is in the set } f(d) && \text{by } f(d) = D \\ &\iff d \text{ is not in the set } f(d) && \text{by definition of } d \in D. \end{aligned}$$

The contradiction rolls back to the beginning, so f cannot be onto the power set $\mathcal{P}(A)$. ☒

When A is a finite set, this is obvious just by counting. Suppose $A = \{1, 2, 3, 4, 5\}$. Then there are $2^5 = 32$ subsets but only 5 elements of A to go around. As the size of A increases this becomes "more and more obvious." The historical kicker is that the proof works even when A is infinite. Georg Cantor gave ironclad criteria by which it follows that $\mathcal{P}(A)$ always has higher **cardinality** than A. In the case where $A = \mathbb{N}$ or $A = \Sigma^*$ this tells us that the set of all languages has higher cardinality than A, i.e., is not countably infinite. Because we have only countably many (string codes or Gödel numbers of) Turing machines, this is an "existence proof" that many languages don't have machines. The function $f(\langle M \rangle) = L(M)$ cannot be onto $\mathcal{P}(\Sigma^*)$.

Many sources give the illustration where the real numbers $\mathbb{R}$ are used in place of $\mathcal{P}(\Sigma^*)$. There is a nagging technical issue that two different decimal or binary expansions like $0.01111...$ and $0.1000...$ can denote the same number (0.5 in this case) but in decimal one can avoid it. The real number that is "not counted" is pictured by going down the main diagonal of an infinite square grid, hence the name *diagonalization* for the whole idea. But I like to do without it.

Yet another variation is to define D with regard to other progarmming formalisms besides Turing machines, for instance:

$D_{Java} = \{p : p \text{ compiles in Java to a program } P \text{ such that } P(p) \text{ does not execute } \texttt{System.exit}(0)\}.$

If D_{Java} were c.e. then by the equivalence of Java and TMs, there would be a Java program Q such that $L(Q) = D_{Java}$ (where acceptance means exiting normally). Then Q would have a valid code q that compiles to Q and ... the logic is the same as before.

For a "technote", if we add to D_{Java} the strings p' that do **not** compile in Java, it does not change the logic. The resulting "augmented" language D'_{Java} still does not have a program Q that accepts it, because the above reasoning was all about valid codes---if Q existed then it would have a valid code q, and that's enough to drive the contradiction.

From Undecidability of D_{TM} to A_{TM}

The following is the complementary problem to D_{TM}. The name with "K" is not used here in Sipser's text but is a natural add-on to Sipser's naming scheme:

K_{TM} :
INST: A deterministic Turing machine M.
QUES: Does M **yes** accept its own code $\langle M \rangle$?

The language, also called K_{TM} or often just "**K**", is $\{\langle M \rangle : \langle M \rangle \in L(M)\}$. This language is not literally the complement of D_{TM} because of strings p' that are not valid codes $\langle M \rangle$ of Turing machines. But K_{TM} is the literal complement of the augmented language D'_{TM} we would get if we threw all those invalid codes p' into D_{TM}. Now here is the reasoning of why K_{TM} is undecidable:

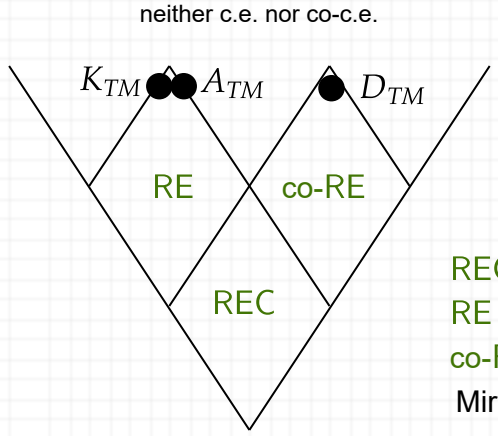
1. If K_{TM} were **decidable**, then there would be a **total** Turing machine V_K such that $L(V_K) = K_{TM}$.
2. Because V_K is total, if we interchange its q_{acc} and q_{rej}, then we would get a total machine V' such that $L(V') = \widetilde{L(V_K)}$.
3. But $\widetilde{K}_{TM} = D'_{TM}$, which is the diagonal language plus all strings that are not valid TM codes.
4. So V' would be a TM Q such that $L(Q) = D'_{TM}$---but we just showed that such a Q cannot exist.
5. Thus V_K cannot exist *as a total machine*, so K_{TM} is undecidable.

Now there does exist a machine U_K that accepts K_{TM}: On input $\langle M \rangle$, double it up to become $\langle M, M \rangle$ and run our universal TM U on that. If M accepts its own code, then U will accept $\langle M, M \rangle$, and vice-versa. The machine U_K is not total, though, because whenever M doesn't even halt on its own code, the run of $U(\langle M, M \rangle)$ won't halt either. Finally, a similar chain of reasoning tells us why the A_{TM}

problem is undecidable:

1. If A_{TM} were decidable, then U could be replaced by a total machine V such that $L(V) = L(U) = A_{TM}$.
2. But then using V in place of U above would make a **total** machine V_K such that $L(V_K) = K_{TM}$.
3. We just showed that such a V_K cannot exist. So V cannot exist, so A_{TM} is undecidable.

The text combines these elements into one chain to prove that A_{TM} is undecidable. There are advantages to that, but one plus point of our breakdown is that we can map out more languages. Here is our first example of what I call a *landscape diagram* (or "cone diagram"):



This diagram conveys some extra information:
◎ REC is closed under complements,
◎ RE ∩ co − RE = REC,
◎ and more to come in Ch. 5 involving *mapping reductions* and *complete languages*.

REC is the class of decidable languages
RE is the class of computably enumerable languages
co-RE is the class of complements of c.e. languages.
Mirror-reflection shows complementation

A Turbid Historical Melange of Confusions and Synonyms

Alan Turing used the word **computable** in English---134 times in his famous 1936 paper---but applied it to *numbers* in a way that got used for single outputs as well as whole languages or functions. The substring "ecidable" appears **0** times in his paper. The usage "computable language" never took hold the way "computable function" does today---and nobody says "decidable function." Alonzo Church had already done equivalent work via *systems of recursion*, and the word **recursive** became applied to both languages and functions, meaning **decidable** for languages and **total computable** for functions f. (The latter term enforces that $Dom(f) = \Sigma^*$, whereas **partial computable** allows $f(x)$ to be undefined for some strings x.) Turing traveled from Cambridge University to Princeton to become Church's PhD student in 1936--38.

If M is a Turing machine that is not total, we can still **recursively enumerate** the strings it accepts by running an infinite loop `for t = 1,2,3,4,...` whose body does the following for each iteration t: try $M(x)$ for all strings x of length up to t for t steps, and print out any x that gets accepted. This defines what the text calls an **enumerator**. It's IMHO confusing because an enumerator has no input and never halts, but the usage stuck, and languages accepted by Turing machines became called **recursively enumerable**, abbreviated **r.e.** (I will not mention enumerators otherwise.)

Recursive and **r.e.** were the standard terms 40+ years ago. Church visited UB in 1990 so my using them when I arrived in 1989 was not out of place. But others were already feeling that those terms sounded too remote. **Decidable** took over the former usage, but what to do with the latter? The term **computably enumerable** with abbreviation **c.e.** gained a following and was adopted by newer texts. Sipser ducked the issue by preferring **recognizable**, which begins with the letters "re"---yet has its own confusion because others have used it to mean decidable. **But**: the **classes** of languages retain their old names: REC for recursive/decidable and RE for r.e./c.e. I've seen "DEC" once or twice, but "CE" never. So to sum up:

- The terms **decidable** and **recursive** are synonyms, and the class of such languages is called REC.
- The terms **computably enumerable** (abbreviated **c.e.**) and **recursively enumerable** (**r.e.**) are synonyms, as are the terms (Turing-)**acceptable** and (Turing-)**recognizable**. The latter is used in our text, but I personally shy away from it---partly for the next reason too. The class of such languages is (only) called RE.
- The complement of a recognizable language is called **co-c.e.** or **co-r.e.** The usages "co-recognizable" or "co-computably enumerable" would be too painful for words... The class of such languages is only called co-RE.
- A computable function f is called **computable** (or **recursive**), but you still have to clarify whether its domain is all of Σ^* (or equivalently for numerical functions, is all of the natural numbers $\mathbb{N}$) or allows f to be a **partial function**.

Some Theorems

The fact of the complement of a decidable language being decidable can now be stated as a closure property: the class REC (like the class REG of regular languages and the class DCFL of DCFLs, but not like the class CFL of CFLs) is closed under complements. But we can prove something more:

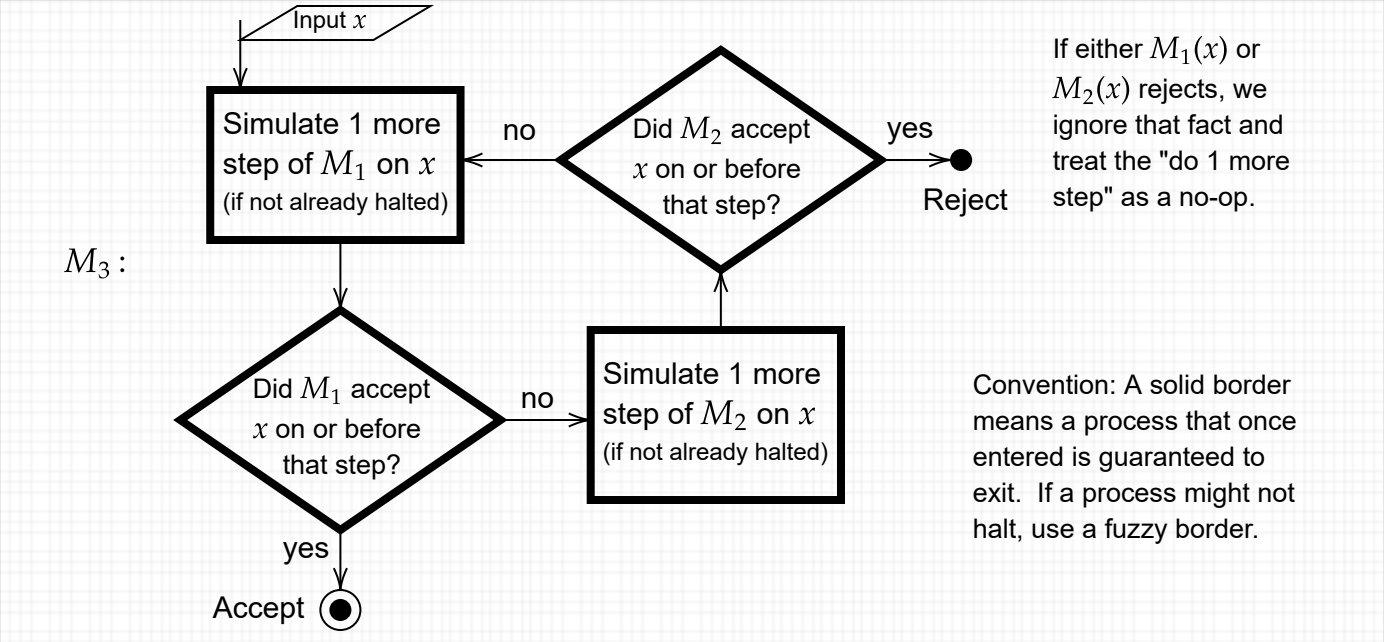
Theorem: A language L is decidable if and only if L and its complement $\widetilde{L}$ are both c.e.

Put another way, L is decidable iff L is both c.e. and co-c.e. This can be summarized in a Venn diagram manner as $\text{REC} = \text{RE} \cap \text{co-RE}$.

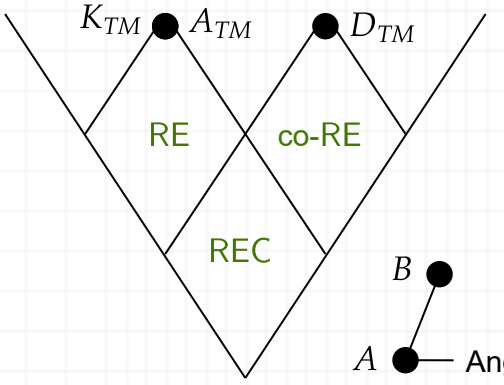
Proof: If L is decidable then it is automatically c.e. And its complement $\widetilde{L}$ is also decidable, which makes it c.e. too. So the "only if" part is immediate. The "if" part is trickier:

Suppose L and its complement $\widetilde{L}$ are both c.e. That means there are Turing machines M_1 and M_2 such that $L(M_1) = L$ and $L(M_2) = \widetilde{L}$. Now neither M_1 nor M_2 is guaranteed to halt on a given input x. But each must halt on the inputs x that it accepts---and by $L \cup \widetilde{L} = \Sigma^*$, every string x gets

accepted by M_1 or by M_2. This allows us to combine M_1 and M_2 into a single machine, for which I will introduce an old-fashioned simple form of **flowchart** notation:



Besides the text's pseudocode, flowchart diagrams are another acceptable way to specify Turing machines. That is, they specify programs, and programs are already equivalent to TMs. This TM M_3 is total because for any x, it will exit either when M_1 accepts x or when M_2 accepts x---and one of them does. And it accepts x if and only if M_1 did the accepting, so $L(M_3) = L(M_1) = L$. Thus we have built a total machine that recognizes L, so L is decidable. ☒



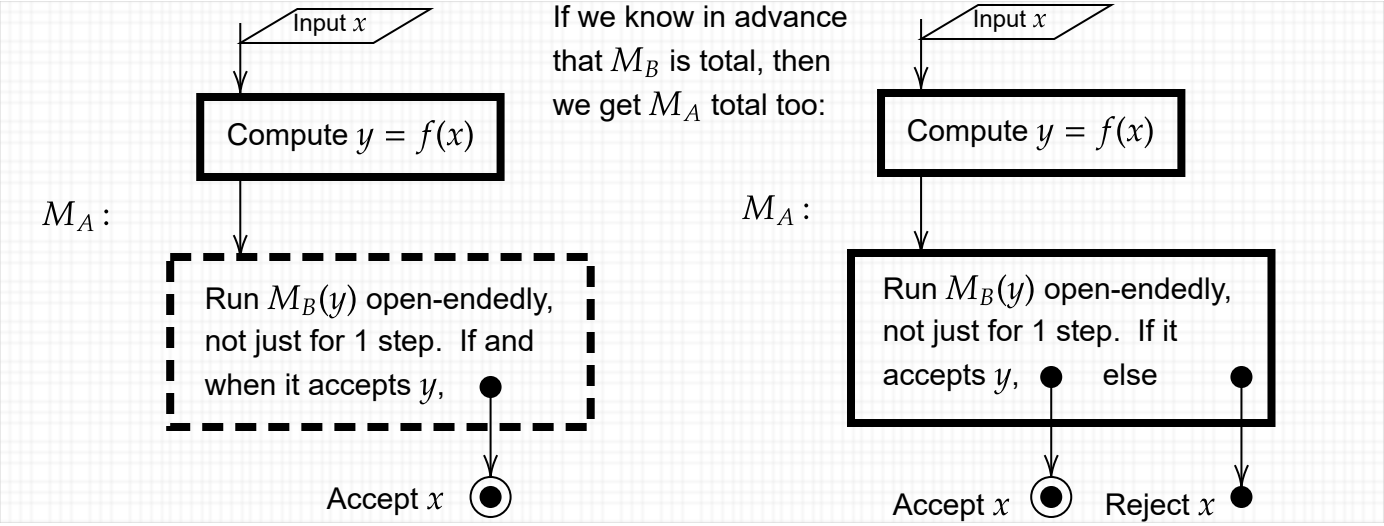
This diagram conveys some extra information:

- REC is closed under complements,
- RE ∩ co − RE = REC,
- REC, RE, and co-RE are each *closed downward under mapping reductions,*
- A_{TM} and K_{TM} are *complete languages for* RE, and D_{TM} is complete for co-RE.

Angle steeper than the walls means $A \leq_m B$.

Mappings

If we have a total computable function $f : \Sigma^* \to \Sigma^*$, then we can put it, too, inside a solid box. Suppose we have a TM M_B that recognizes a language B, and we design a TM M_A like so:



In either case, we have $L(M_A) = \{x : f(x) \in L(M_B)\}$. Putting $A = L(M_A)$ as well as $B = L(M_B)$, what we have is that for all $x \in \Sigma^*$, $\boldsymbol{x \in A \iff f(x) \in B}$.

Chapter 5's title topic "Mapping Reducibility" doesn't come until section 5.3, but we put it up-front:

Definition: A language A **mapping-reduces** to a language B if there is a total computable function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$, $\boldsymbol{x \in A \iff f(x) \in B}$. This is written $\boldsymbol{A \leq_m B}$.

We also say $A \leq_m B$ **via** f and call f a **mapping reduction**. The historical term is to call f a **many-one reduction** to say that f need not be a 1-to-1 correspondence. The above flowchart diagrams already prove the first two of the following main implications about mapping reductions:

Theorem: Suppose A and B are any languages such that $A \leq_m B$. Then:

(a) If B is decidable, then A is decidable.
(b) If B is c.e., then A is c.e.
(c) If B is co-c.e., then A is co-c.e.

Proof: Only part (c) is left to prove, and it needs only the fact that $\boldsymbol{x \in A \iff f(x) \in B}$ is logically equivalent to $\boldsymbol{x \in \widetilde{A} \iff f(x) \in \widetilde{B}}$. If B is co-c.e., then $\widetilde{B}$ is c.e., and we have $\widetilde{A} \leq_m \widetilde{B}$. By part (b), this makes $\widetilde{A}$ c.e., which means that A is co-c.e. ☒

We will use this to prove more problems to be undecidable---and more languages to be not c.e. or even neither c.e. nor co-c.e.---by applying the *contrapositive* form:

Theorem': Suppose A and B are any languages such that $A \leq_m B$. Then:

(a) If A is undecidable, then B is undecidable.
(b) If A is not c.e., then B is not c.e.
(c) If A is not co-c.e., then B is not co-c.e. ☒