CSE491/596 Lecture Wed. Oct. 7. Undecidable Languages

Define $D_{TM} = \{\langle M\rangle : M \textit{ does not accept } \langle M\rangle\}$. 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. Another is to use the recursive enumeration $M_0, M_1, M_2, \dots$ of DTMs, that is, to treat their codes as "Gödel Numbers." Then the definition looks like:

$$D_{TM} = \{i : i \notin L(M_i)\}.$$

The proof then goes: if Q existed, it would equal M_q for some number q. But then Q accepts $q \iff$ ... as above.

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 had such an A and f. Then we would have the subset

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

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

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

The contradiction rolls back to the beginning, so there cannot be such an A and f. ☒

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:

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

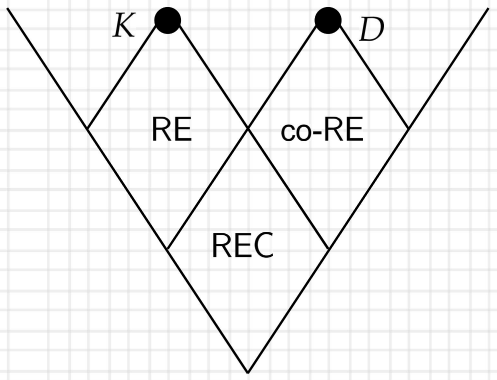
If D_{Java} were c.e. then by the equivalence of Java and TMs, there would be a Java program Q such that $L(Q) = D_{Java}$ (where acceptance means exiting normally). Then Q would have a valid code q that compiles to Q and ... the logic is the same as before.

One nice aspect of Gödel Numbers is that you don't have to worry about strings that are not valid codes. So if we define

$$D = D_{TM} = \{i : i \notin L(M_i)\}$$
$$K = K_{TM} = \{i : i \in L(M_i)\} = \{i : \langle M_i, i\rangle \in A_{TM}\}$$

then K_{TM} is literally the complement of D_{TM}.

neither c.e. nor co-c.e.



This diagram conveys some extra information:

◎ REC is closed under complements,

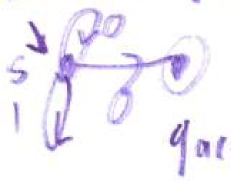
◎ RE $\cap$ co $-$ RE $=$ REC, and

◎ All three classes are closed downward under *computable many-one/mapping reductions*.

We will prove these after we establish the equivalence between Turing machines and high-level programming languages.

The trick of switching q_{acc} and q_{rej} does not affect running time or space, so it also shows that EXP, PSPACE, P, L, are closed under ~.

- Does not work for NTMs: Same as with an NFA:



$L = \{x : x \text{ ends in } 0\}$. but swapping $F = \{s\}$ does not $\tilde{L} = \{x : x = \varepsilon \text{ or } x \text{ ends in } 1\}$ do $\tilde{L}$.

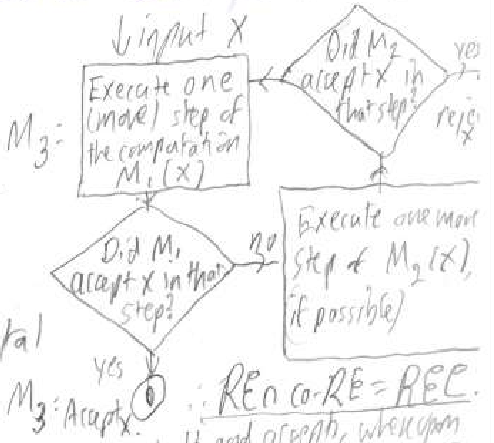
So does not show that NP = co-NP. Also does not show <u>NL = co-NL</u>, but this is a famous theorem (1988)

- Does not work if M does not halt for all inputs

But we <u>can</u> show

<u>Theorem</u>: if A and $\tilde{A}$ are both c.e., then A is decidable.

Proof: Take DTMs M_1 and M_2 such that $L(M_1) = A$ and $L(M_2) = \tilde{A}$. Build M_3 "via flowchart" as follows.



M_3:
input x
Execute one (more) step of the computation $M_1(x)$
Did M_1 accept x in that step?
no
Execute one more step of $M_2(x)$, (if possible)
Did M_2 accept x in that step?
yes
reject x
yes
M_3: Accept x.

Then M_3 is total because for all x, exactly one of $M_1(x)$ and $M_2(x)$ halts and accepts, whereupon $M_3(x)$ halts and decides. So A is decidable. ☒

$\therefore$ <u>RE $\cap$ co-RE = REC.</u>

Footnotes:
I showed that the diagonalization proof looks like visually if you don't write q in place of $\langle Q \rangle$:

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

The reason I wrote q is that doing so made the later variant proofs closer in form. There was also a question in the chat:

Q: if a is in A, then a should be in $\mathcal{P}(A)$, right?

A: Then $\{a\}$ is in $\mathcal{P}(A)$. Consider $\epsilon :$ *string* versus $\emptyset, \{\epsilon\} :$ *language*.

For a followup, Bertrand Russell came up with the most vicious and viscous diagonal set of all:

$$D = \{x : x \notin x\}.$$

The original freewheeling approach to set theory allowed you to state the possibility of a set being a member of itself. If you expect that no set could ever be a member of itself, then D would become "the set of all sets"---but then D would be a member of itself. Indeed, we immediately get the logical equivalence $D \in D \iff D \notin D$. The resolution is not that "$x \notin x$" is always true, but rather that "$x \in x$" and "$x \notin x$" do not compile --- because the $\in$ relation and its negation are allowed to be used only between objects of type T and `set<`T`>` for some type T. This led to a hierarchy of types, which were called "ramified", and a book *Principia Mathematica* that can be analogized to reading the object code of a compiler, both painful...