CSE396 Lecture Thu. 4/29: Degrees of Undecidability

This lecture will conclude the coverage of Chapter 5, mainly skipping section 5.2. To repeat some of Tuesday before we pick up in the middle of the first "reduction design pattern":

Three Design Patterns For Reductions

The motivation is similar to that in general code: the ideas of reductions are often reusable.

I. "Wait For It"

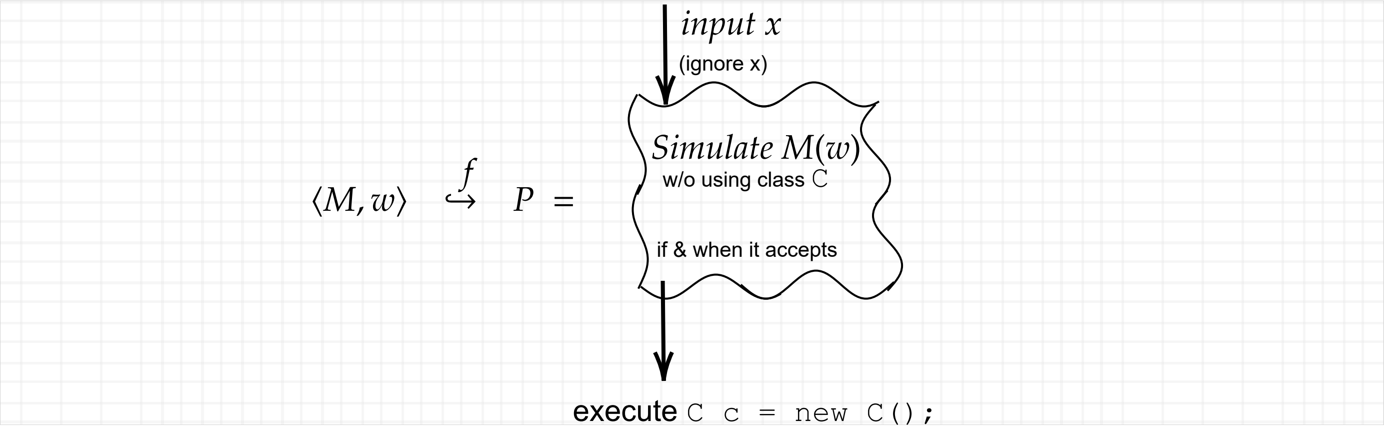
Long ago, certainly before *Hamilton*, I used to call the first one "Waiting For Godot" after the Samuel Beckett play in which (spoiler alert---wait, giving a spoiler alert for that play is an ultimate existential absurdity) ... When we first had the *Turing Kit* and Java was new and intimations of the "Internet of Things" started to buzz, I called this the generic reduction to the "Brew Coffee" problem: if you switch on your Java-enabled coffee maker M', will it brew coffee? You see, M' might ask Alexa to invoke the *Turing Kit* on a given $\langle M, w\rangle$, and brew your coffee only if and when M accepts w. This year, with the Turing $£50$ note, I considered joking about the "ATM Problem": if you put your card in and try to withdraw $£50$, will it give you a Turing or a background check that never halts? But let's do it with a problem that is actually highly relevant and attempted in practice when trying to cut down "code bloat" by removing unused classes from object-oriented code.

UsefulClass
Instance: A Java program P and a class `C` defined in the code of P.
Question: Is there an input x such that $P(x)$ creates an object of class `C`?

We mapping-reduce A_{TM} to the language of this decision problem. We need to compute $f(\langle M, w\rangle) = P$ such that:

- M accepts $w \implies$ for some x, $P(x)$ executes an instruction like `C c = new C();`
- M does not accept $w \implies$ for all x, $P(x)$ never executes any statement involving `C`.

I like to picture f as dropping M and w into a flowchart for P:



A key fine point in the correctness logic is that the class `C` does not appear anywhere else in the code of P. The main body of P can be entirely a call to the *Turing Kit* program with M and w pre-packaged. This body does not use any classes besides those in the *Turing Kit* itself. Even if $M(w) \uparrow$, whereupon P never halts either, it remains true that the class `C` is never used---so that removing it would not change the behavior of P, not on any input x.

Building the program P is straightforward given any M and w: just fix M and w to be the arguments in the call to the *Turing Kit*'s main simulation routine and append the statement shown in the diagram after the place in the *Turing Kit*'s own java code where it shows the `String accepted` dialog box. Thus the code mapping f itself is computable, indeed, easily linear-time computable.

The conclusion is that the problem of detecting (never-)used classes is undecidable. It may seem that programs $P = P_{M.w}$ are irrelevant ones by which to demonstrate this because they are so artificial and stupidly impractical. However:

1. the reduction to these programs shows that there is "no silver bullet" for deciding the useful-code problem in *all* cases; and
2. the programs $P_{M.w}$ are "tip of an iceberg" of cases that have so solidly resisted solution that most people don't try---exceptions such as the Microsoft Terminator Project are rare.

This kind of reduction is one I call "Waiting for Godot" after a play by Samuel Beckett in which two people spend the whole time waiting for the title character but he never appears. The real import is that there are a lot of "waiting for..." type problems about programs P that one would like to tell in advance by examining the code of P. The moral is that most of these problems, by dint of being undecidable in their general theoretical formulations, are practically hard to solve. The practical problem of eliminating code bloat by removing never-used classes is one of them. Without strict version control, whether blocks of code have become truly "orphaned" and no longer executed can become hard to tell.

[The transition to Thursday's lecture came out here.]

For a side note, the "type" of the target problem is "Just a progarm P", not "a program and an input string" as with A_{TM} itself. We did not map $\langle M, w\rangle$ to $\langle P, w\rangle$; w is not the input to P. Instead, x is quantified existentially in the statement of the problem. This makes sense: the code is useful so long as *some* input uses it. The language of the problem combines two existential conditions:

- there exists an x such that when P is run on x, ...
- ...there exists a step at which P creates an object of the class `C`.

A language defined by **existential** quantifiers in this way, down to "bedrock" predicates like creating a class object that are *decidable*, is generally ***c.e.*** The kind of algorithmic technique used to show this is commonly called "dovetailing." I like to picture dovetailing as occurring inside an enclosing arbitrary time-allowance loop. In this case, noting that we are trying to analyze P:

```
input $\langle P, \texttt{C} \rangle$
for $t = 1, 2, 3, 4, \dots$ :
  for each input $x$ up to $t$ (or you can say: of length up to $t$):
    run $P(x)$ for $t$ steps.  If $P(x)$ builds an object of class C during those steps, accept $\langle P, \texttt{C} \rangle$.
```

This is a program R such that $L(R) = \{\langle P, \texttt{C} \rangle : (\exists x, t)[P(x) \textit{ builds a } \texttt{C} \textit{ object within } t \textit{ steps}]\}$, which is the language of the USEFULCLASS problem. So this language is c.e. but undecidable.

II The "All-or-Nothing Switch"

This actually builds on the "wait-for-it" kind of reductions. Note that HP_{TM} had an instance type that specified both "an M and an input x" but UsefulClass had the instance type "just a program P" where the x part was *quantified* as "Does there exist an x such that $P(x)$...?" When there is flexibility in how the "x" part is treated, we can often hit a whole bunch of problems with a reduction at once. Here are three (and K_{TM} will make a fourth):

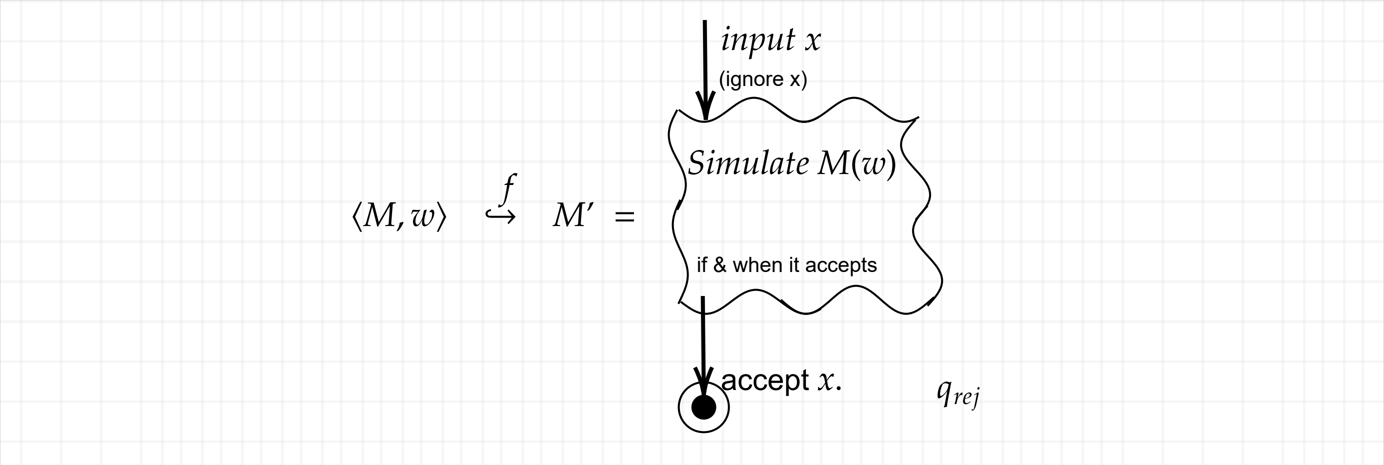
NE_{TM}
Instance: A TM M'.
Question: Is $L(M') \neq \emptyset$?

ALL_{TM}
Instance: A TM M'.
Question: Is $L(M') = \Sigma^*$?

Eps_{TM}
Instance: A TM M'.
Question: Does M' accept ϵ?

In the first problem, it might seem more natural to phrase the question as "is $L(M) = \emptyset$?" but that would make the *language* of the problem become $\{\langle M \rangle : L(M) = \emptyset\}$, which is called E_{TM}. The reason we need to use $NE_{TM} = \{\langle M \rangle : L(M) \neq \emptyset\}$ is that when doing mapping reductions, we need to make "yes" cases of the *source* problem line up with "yes" answers to the *target* problem. We will see that usually it is impossible to do it the other way, switching "yes" and "no". The language NE_{TM} as-defined is c.e. Here is the reduction:

Here M' is a Turing machine, but we could get it by using the same call to the *Turing Kit* and then converting the resulting Java code to a Turing machine as proved in the Friday 10/2 lecture. Or we can just build M' by having M' (which depends on M and w) first write the fixed string $\langle M, w \rangle$ on its tape next to x (or even in place of x in this case) and then go to the start state of a universal TM U which is made to run on $\langle M, w \rangle$. Either way, f is computable---since U is fixed and the initial "write $\langle M, w \rangle$" step takes time proportional to the length of $\langle M, w \rangle$ to code, the latter more clearly makes f linear-time computable. So this Construction is Computable.



Here is the one-shot Correctness analysis for all three target problems:

$M\ accepts\ w \implies$ the "fuzzy box" main body of M' always exits, regardless of the input x;
$\implies$ for all inputs x, M' accepts x;
$\implies L(M') = \Sigma^*$, which also implies that $L(M') \neq \emptyset$ and M' accepts ϵ.

Thus,
$\langle M, w\rangle \in A_{TM} \implies f(\langle M, w\rangle) = \langle M'\rangle$ is in all of ALL_{TM}, NE_{TM}, and Eps_{TM}. Whereas,

$M\ doesn't\ accept\ w \implies$ the main body of M' rejects or never finishes; either way, it never accepts;
$\implies$ for all inputs x, M' does not accept x;
$\implies L(M') = \emptyset$, which also implies that $L(M') \neq \Sigma^*$ and $\epsilon \notin L(M')$.

Thus,
$\langle M, w\rangle \notin A_{TM} \implies f(\langle M, w\rangle) \notin E_{TM}$, $f(\langle M, w\rangle) \notin ALL_{TM}$, and $f(\langle M, w\rangle) \notin Eps_{TM}$.

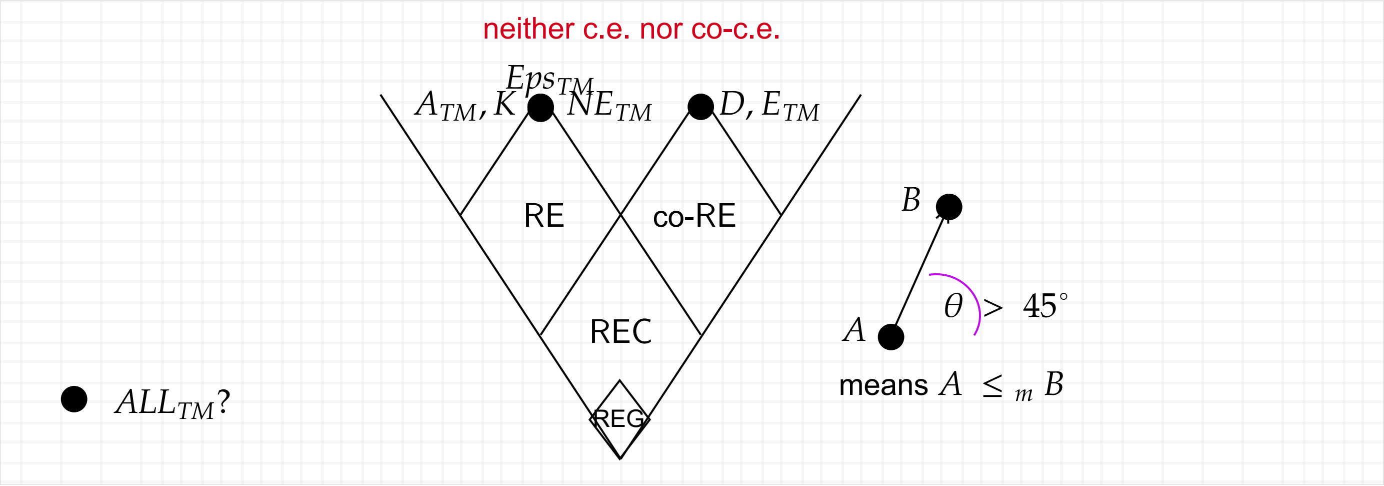
We also get $\langle M, w\rangle \in A_{TM} \iff f(\langle M, w\rangle) = \langle M'\rangle \in K_{TM}$. So this shows $A_{TM} \leq_m K_{TM}$ too.

So we have simultaneously shown $A_{TM} \leq_m NE_{TM}$, $A_{TM} \leq_m ALL_{TM}$, and $A_{TM} \leq_m Eps_{TM}$. Thus all three of these problems and their languages are undecidable.

We also have $K_{TM} \leq_m NE_{TM}$ by transitivity (or you can show it directly). By the complements rule for reductions, this means that $D_{TM} \leq_m E_{TM}$. By the Sipser naming scheme, again we have

E_{TM}
Instance: A TM M.
Question: Is $L(M) = \emptyset$?

and the corresponding *language*, also called E_{TM}, is $\{\langle M\rangle : L(M) = \emptyset\}$. Thus E_{TM} is hard for co-RE, and since it belongs to co-RE (i.e., it is co-c.e., since NE_{TM} is c.e.), it is complete for co-RE. The picture now is as follows (it's OK to lose the "TM" subscripts on K and D especially):

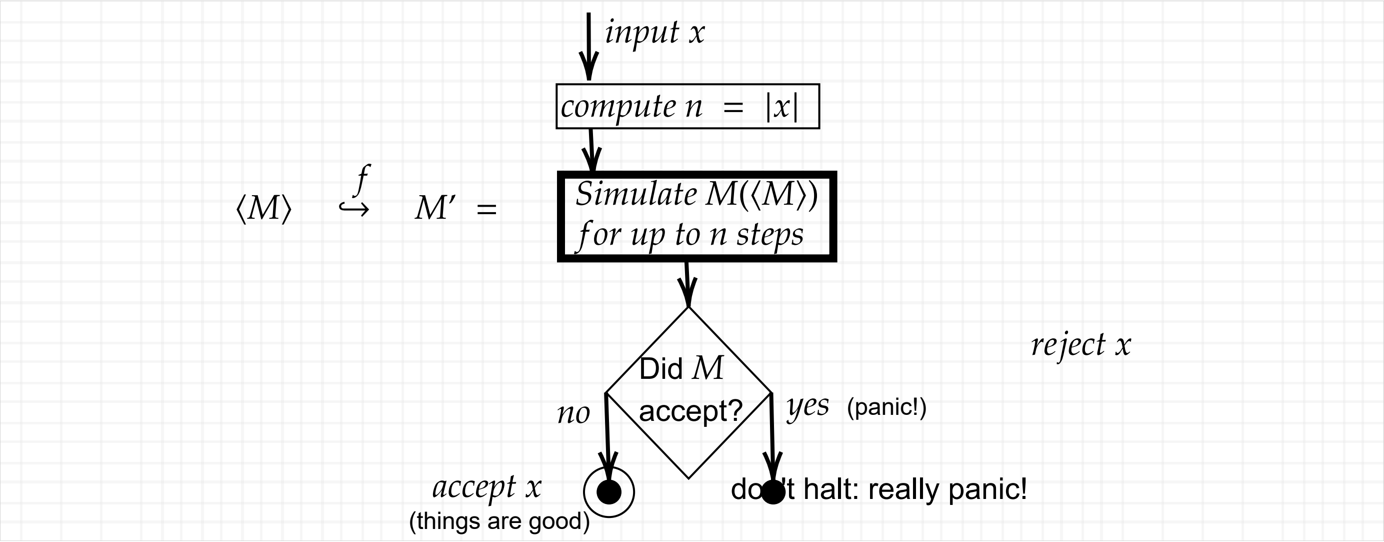


In passing, here's a self-study question: How would you go about showing $A_{TM} \leq_m K_{TM}$? Showing $K_{TM} \leq_m A_{TM}$ was easy, but now we have to package an arbitrary pair $\langle M, w\rangle$ into a single machine M' that accepts its own code if and only if M accepts w. If you think about this task intensionally, it may seem daunting: how can we vary the code of M' for all the various w strings so that M' does or does not accept its own code depending on whether w gets accepted by M. How on earth can we pack two things into one? But if you think extensionally in terms of the correctness logic of a reduction, the answer might "jump off the page" at you...

By showing $A_{TM} \leq_m NE_{TM}$, we have not only shown that the NE_{TM} language is undecidable, we have shown it is not co-c.e. But since the A_{TM} language is c.e., NE_{TM} could be c.e.---and indeed it is, by dovetailing: Given any TM M, for $t = 1, 2, 3, \ldots$: try M on all inputs $x < t$ for up to t steps. If M' is found to accept any of them within t steps, accept $\langle M'\rangle$, else continue. That the language (of) Eps_{TM} is c.e. is simpler to see: given M', just run $M'(\epsilon)$ and accept $\langle M'\rangle$ if and when M' accepts ϵ. But how about the language of ALL_{TM}? Hmmm....

III The "Delay Switch"

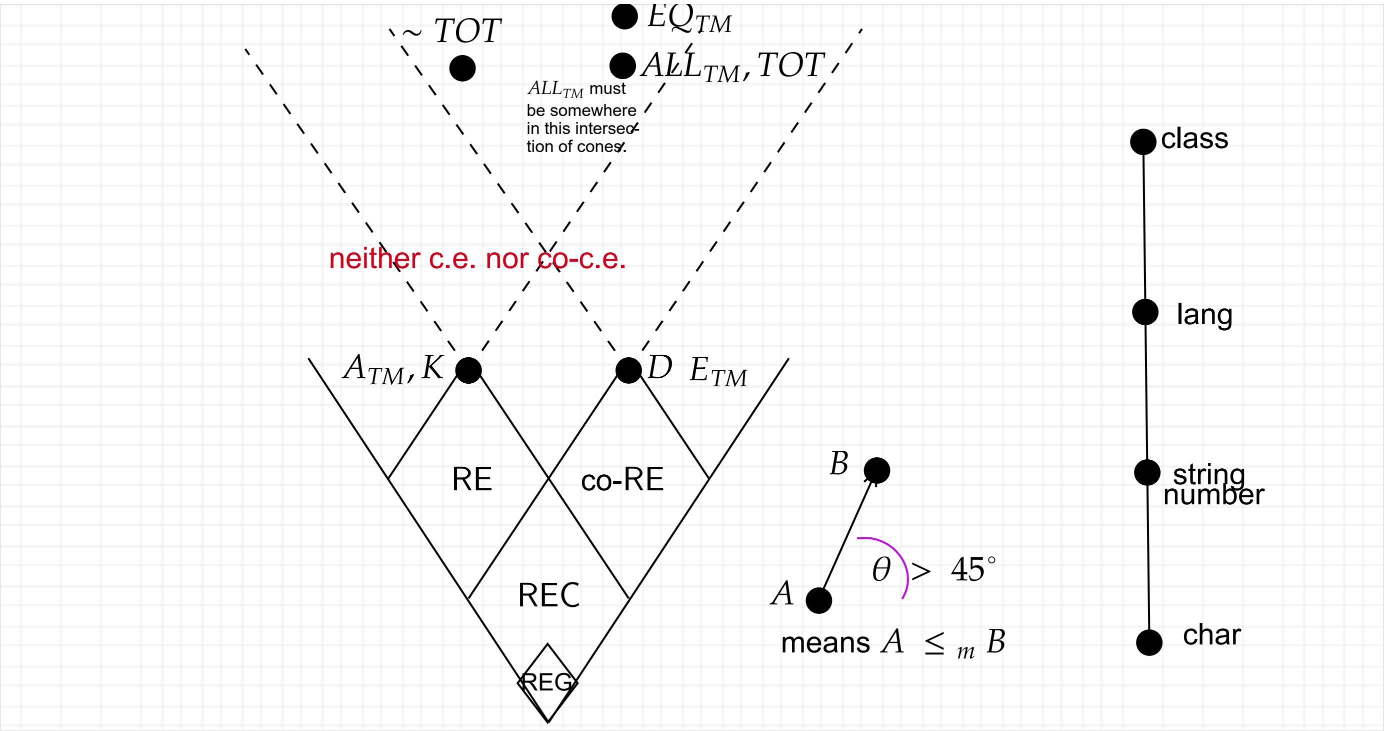
The third useful reduction technique is something I call the "Delay Switch." The intuition and attitude are the opposite of "Waiting for Godot" and the all-or-nothing switch. This time you picture your target machine M' or target program P as monitoring a condition that you hope *doesn't* happen, such as when doing security for a building. The input x to the target machine is first read as giving a length t_0 of time that you have to monitor the condition for. Usually we just take $t_0 = |x|$, the length of the input string x (you may always call this length n too). If the condition doesn't happen over that time---that is, if no "alarm" goes off---then you stay in a good status. But if the alarm goes off within t_0 steps, then you "panic" and make M' (or P) do something else. Because this is a general tool, let's show an example of the construction even before we decide what problems we're reducing *to* and *from:*


input x
compute $n = |x|$
$\langle M \rangle \overset{f}{\hookrightarrow} M' =$
Simulate $M(\langle M \rangle)$ for up to n steps
Did M accept?
no
yes (panic!)
reject x
accept x
(things are good)
don't halt: really panic!

This flowchart is a little more complicated, but it is just as easily computed given the code of M. We've given $\langle M \rangle$ not $\langle M, w \rangle$ in order to help tell this apart from the other reductions and because of the source problem we get. A key second difference is that all the components of M' are solid boxes: $M'(x)$ always halts for any x. The logical analysis now says:

- If M never accepts its own code, then the diamond always takes the *no* branch. So every input x gets accepted, and so $L(M') = \Sigma^*$.
- If M does accept its own code, then there is a number t of steps at which the acceptance occurs. Thus for any input x of length $n \geq t$, the simulation of $M(\langle M \rangle)$ in the main body sees the acceptance. So the *yes* branch of the diamond is taken, and the "post-alarm" action in this case is to circle the wagons and reject x. This means that all but the finitely many x having $|x| < t$ get rejected, so not only is $L(M') \neq \Sigma^*$, it isn't even infinite.

What this amounts to is: $\langle M \rangle \in D_{TM} \iff L(M') = \Sigma^* \iff f(\langle M \rangle) \in ALL_{TM}$. So we have shown $D_{TM} \leq_m ALL_{TM}$, whereas before we showed $A_{TM} \leq_m ALL_{TM}$ (and it follows that $K_{TM} \leq_m ALL_{TM}$). Since D_{TM} is not c.e., this means we have shown that ALL_{TM} is not c.e. either. Hence ALL_{TM} is *neither c.e. nor co-c.e.* To convey this consequence pictorially:



There is an intuition which we will later turn into a theorem while proving its version for NP and co-NP at the same time. The language D_{TM} has a purely negative feel: the set of M such that M does *not* accept its own code. When we boil this down to immediately verifiable statements, we introduce a universal quantifier:

For all time steps t, M does not accept its own code in that step.

The watchword is that the D_{TM} language is definable by *purely universal quantification over decidable predicates.* So is the E_{TM} language:

For all inputs x and **all** time steps t, M does not accept x within t steps.

We could combine this into just one "for all" quantifier by saying: **for all** pairs $\langle x, t\rangle$... In any event, much like having a purely existential definition is the hallmark of being c.e., haveing a purely universal definition makes a language co-c.e. This is to be expected, because a negated definition of the form

$$\neg(\exists t)R(i,t) \quad \text{flips around to become} \quad (\forall t)\neg R(i,t).$$

If the language $\{\langle i, t\rangle : R(i,t)\ holds\}$ is decidable, then so is its complement, which (ignoring the issue of strings that are not valid codes of pairs) is the language of $\neg R(i,t)$. So we get the same bedrock of decidable conditions in either case.

With ALL_{TM}, however, we have to combine both kinds of quantifier into one statement to define it. The simplest definition of "$L(M) = \Sigma^*$" is:

For all inputs x, **there exists** a timestep t such that **[M accepts x at step t]**.

The square brackets are there to suggest that the predicate they enclose is a "solid box" meaning decidable. Believe-it-or-else, this predicate is also named for Stephen Kleene...in a slightly different form which we will cover once we hit complexity theory. For now, let us state:

Theorem (the proof will come when we hit NP and co-NP next week):

- A language L is c.e. if and only if it can be defined using only one or more initial existential ($\exists$) quantifiers in front of a decidable predicate.
- A language L is co-c.e. if and only if it can be defined using only one or more initial universal ($\forall$) quantifiers in front of a decidable predicate.

Now you might wonder: is there a more clever way to define the notion of "$L(M) = \Sigma^*$" using just one kind of quantifier? The fact that ALL_{TM} is neither c.e. nor co-c.e. says a definite **no** to this possibility.

As for what it means in practice, you can use the "logical feel" of a problem to pre-judge whether it is c.e. or co-c.e. (in which case, if asked to show the problem undecidable, the choice of problem to reduce *from* is mostly forced), or neither---in which case, it's "*carte blanche"*---before proving exactly how it is classified. For example, consider

$$TOT = \{M : M \textit{ is total, i.e.,} \ \forall x, \ M(x) \downarrow \}.$$

It has a "for all" feel to it. So the first intuition says it is not c.e. That is correct, and we can prove it by showing $D_{TM} \leq_m TOT$ via the delay switch. Then we can ask whether it is not co-c.e. either. In fact, TOT is highly similar to ALL_{TM} and the same ideas as for $A_{TM} \equiv_m HP_{TM}$ work to show $ALL_{TM} \equiv_m TOT$. A similar case is EQ_{TM}, which we can reduce ***from*** ALL_{TM} "by restriction."

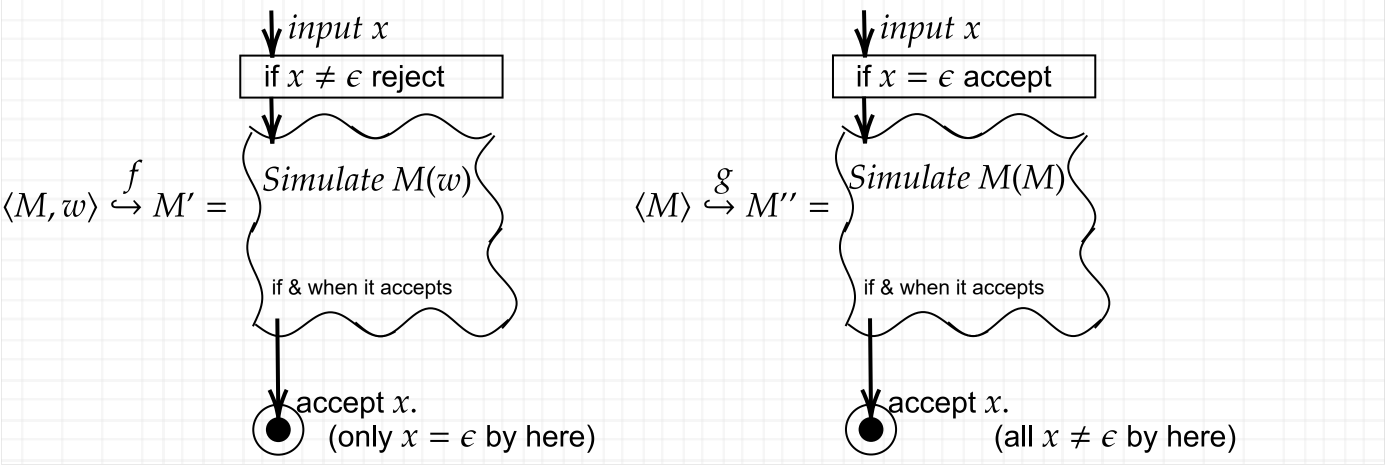
Example: $EQ_{TM} = \{\langle M_1, M_2 \rangle : L(M_1) = L(M_2)\}$. Prove this is neither c.e. nor co-c.e.

Make a special case the target: the case where M_2, say, has $L(M_2) = \Sigma^*$. Call that M_{all}. Then $\langle M_1, M_{all} \rangle \in EQ_{TM} \iff \langle M_1 \rangle \in ALL_{TM}$. So $ALL_{TM} \leq_m EQ_{TM}$ by the simple reduction $f(M) = (M, M_{all})$. Because we showed ALL_{TM} is neither c.e. nor co-c.e., the same "45° cone logic" says that EQ_{TM} is neither c.e. nor co-c.e.

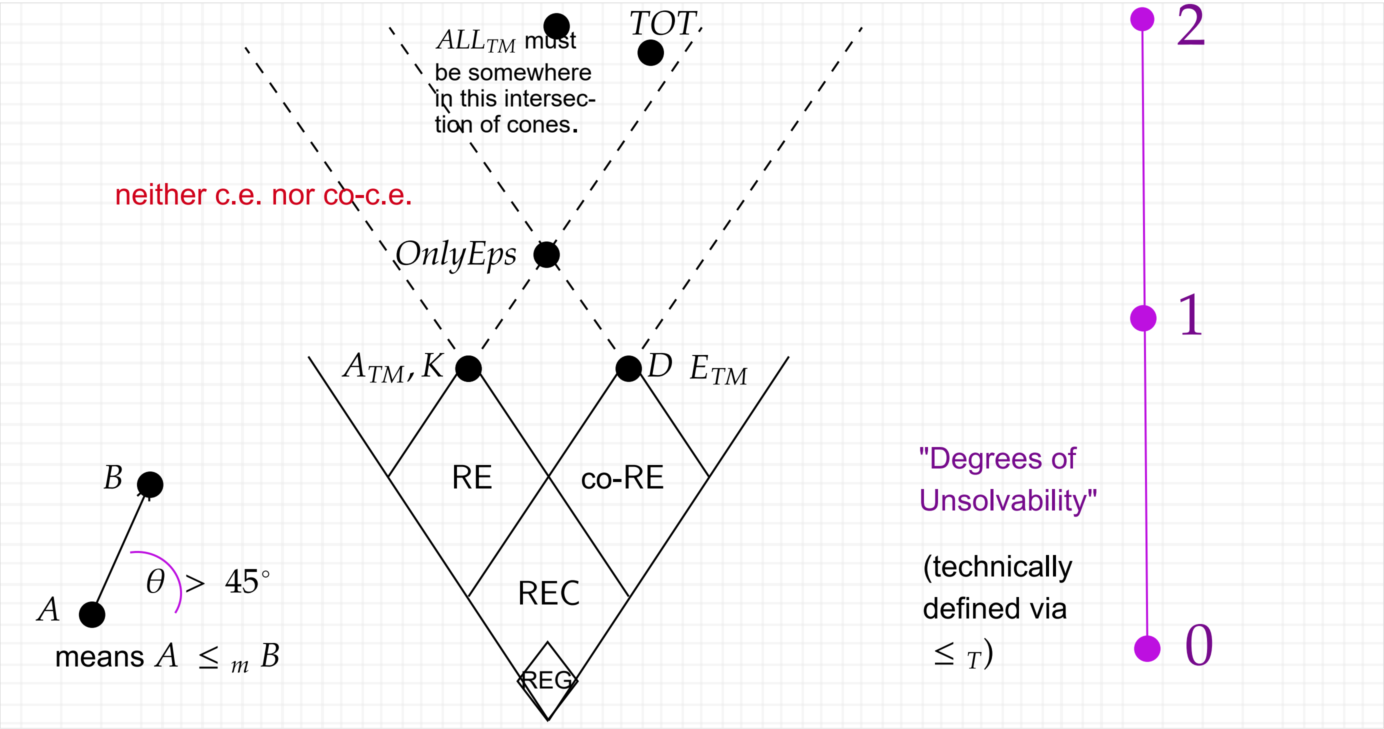
Here is a trickier problem with a trickier name:

$OnlyEps_{TM}$
INST: A Turing machine M'.
QUES: Is $L(M') = \{\epsilon\}$? That is, does M' accept ϵ but no other string?

Here are diagrams of reductions showing $A_{TM} \leq_m OnlyEps_{TM}$ and then $D_{TM} \leq_m OnlyEps_{TM}$.


input x
if $x \neq \epsilon$ reject
$\langle M, w\rangle \overset{f}{\hookrightarrow} M' =$
Simulate $M(w)$
if & when it accepts
accept x.
(only $x = \epsilon$ by here)
input x
if $x = \epsilon$ accept
$\langle M\rangle \overset{g}{\hookrightarrow} M'' =$
Simulate $M(M)$
if & when it accepts
accept x.
(all $x \neq \epsilon$ by here)

For self-study, do the correctness logic on these reductions. Also make the second one work with the "delay switch" idea. It turns out that the $OnlyEps$ language is in the least $\equiv_m$ equivalence class of languages that reduce from both K and D. In particular, it is lower than ALL_{TM} and TOT. [Technically, $OnlyEps$ and K and D are all in the same equivalence class under Alan Turing's original reducibility notion, called **Turing reductions** and written $\leq_T$. But Turing reductions would collapse the left-right dimension (which corresponds to ∃ versus ∀ in logic) down to a single stick, as at right below. So I prefer to avoid them at this point.]


ALL_{TM} must be somewhere in this intersection of cones.
TOT
neither c.e. nor co-c.e.
$OnlyEps$
A_{TM}, K
D E_{TM}
RE
co-RE
REC
REG
B
A
θ > 45°
means $A \leq_m B$
"Degrees of Unsolvability"
(technically defined via $\leq_T$)
2
1
0

[We can drop the "TM" subscripts not only when the context is clear but because using Java or any other high-level programming language would give exactly the same classification of the analogously-defined languages, e.g. A_{Java}, D_{Java}, K_{Java}, $OnlyEps_{Java}$, etc. But now we will see machines between Turing machines and DFAs for which the classifications do change and the distinction between "decidable" and "undecidable" is almost on a knife-edge.]

Reductions Via Computation Traces

[This parallels the second half of section 5.1 of Sipser, but emphasizes the problems that pertain to context-free grammars and PDAs first, rather than discuss Linear Bounded Automata (LBAs) first. IMHO, LBAs go more naturally as a connection to space complexity in chapter 7.]

We recall (from the April 8 lecture) the definition of **instantaneous descriptions** (**IDs**, also called **configurations**), which give the current state, current tape contents aside from blanks, and current head position(s) at any point in a computation by a Turing machine. The starting ID on an input $x \in \Sigma^*$ is denoted by $I_0(x)$. For a single-tape Turing machine M with start state s this can have the simple form $I_0(x) = sx$ where the state is treated as a character. If we make TMs do "good housekeeping" when they are about to produce an output y by blanking out everything on their tape(s) except for y, then the computations can end in a unique final ID $I_f = q_{acc}y$. If the TM is a decider, we can suppose it outputs 1 for "accept" and 0 for "reject". Then it has a unique accepting ID $I_f = q_{acc}1$. We also defined the relation $I \vdash_M J$ to mean the ID I can go to the ID J in a single step by M. Thus a **valid accepting computation** (**trace**) has the form

$$I_0(x) \vdash_M I_1 \vdash_M I_2 \vdash_M I_3 \vdash_M \cdots \vdash_M I_{t-2} \vdash_M I_{t-1} \vdash_M I_f ,$$

and a valid computation that halts and rejects can be defined analogously with $I_{rej} = q_{rej}0$ as the last ID I_t. Then t is the **number of steps**---that is, the **time** taken by the computation---and we generally suppose this is at least $n+1$ where n is the length of x. The computation trace itself can be encoded as a string

$$\vec{c} = \langle I_0(x), I_1, I_2, I_3, \ldots, I_{t-2}, I_{t-1}, I_t \rangle .$$

of length $O(t^2)$, since the IDs can expand by at most one char in each step. There is some flexibility in representing traces, of which the funkiest is to write every odd-numbered ID in reverse, i.e., $\vec{c'} = \langle I_0(x), I_1^R, I_2, I_3^R, \ldots, I_{t-2}, I_{t-1}^R, I_t \rangle$ (if t is even). The key question is:

> What kinds of machines---or combinations Z of machines or other formal objects---can tell whether strings of this kind really represent valid computations?

That is, given any Turing machine M, what does it take to recognize the language V_M of its valid computation traces? Let's write this as a definition and observe a key set of facts:

Definition: For any Turing machine M (wlog. a single-tape deterministic TM), V_M is the language of its valid (accepting) computation traces, and V'_M stands for the form where every odd ID is written reversed.

Theorem: $L(M) = \emptyset \iff V_M = \emptyset \iff V'_M = \emptyset$. ☒

Note that even if M is not total---indeed even if $L(M)$ is c.e. but undecidable so that M cannot be total---the language V_M can be decidable. This is because you are not just given x but an entire string $w = \langle I_0(x), I_1, I_2, I_3, \dots, I_{t-2}, I_{t-1}, I_t \rangle$, and you just need to determine by looking entirely within the bounds of w itself whether it is valid. This means checking that

$$I_{k-1} \vdash_M I_k$$

for all k, $1 \leq k \leq t$. This relation is decidable by checking that the action of some instruction $(q, c/d, D, r)$ in the code of M that is applicable in I_{k-1} (for instance on a single-tape TM, the ID could be $uqcv$ for some strings $u, v \in \Gamma^*$ and $c \in \Gamma$) produces the ID I_k. This is analogous to saying that we can "deduce" I_k from I_{k-1}. This is the machine analogue of checking a formal logical proof (the way you may have done in CSE191), but in some ways it's easier:

1. A line k in a formal mathematical proof might depend on multiple lines $i, j < k$ that could be anywhere in the preceding steps. Whereas, in a computation $\vec{c}$, the dependence is only on the previous step.
2. Or line k in the proof could be an axiom standing by itself. In $\vec{c}$ this only happens at the start.
3. Although technically the only logical rule needed other than introducing an axiom is *modus ponens*, the details for *instantiating* an infinite axiom *schema* can be pretty hairy---indeed, the induction rule is often "hidden" that way. With computations $\vec{c}$, no such shenanigans!

So we should be interested in, what are the simplest kinds of machines that can check computation traces, or formalisms that can represent valid traces (or invalid ones)? The action we need to check in any pair $I_{k-1} \vdash I_k$ (?) is local to just a few chars in one part of I_{k-1} and I_k. Most of the task is checking that the rest of the IDs is identical, which is much like deciding the "Double Word" language $\left\{ ww : w \in \Sigma^* \right\}$.

Indeed, the task over the whole trace is like an iterated form of the "Double Word" language---except that in the case where odd-numbered IDs are reversed it is like iterated palindrome checking. Let's visualize the latter in a case where $x = 011001$ and the first three instructions executed by a single-tape TM M are $(s, 0/0, R, p)$, $(p, 1/0, R, q)$, and $(q, 1/1, L, r)$. Then

$$\vec{c} = \langle s011001, 0p11001, 00q1001, 0r01001, \dots \rangle$$

and

$$\vec{c'} = \langle s011001, 10011p0, 00q1001, 10010r0, \dots \rangle$$

Suppose we tried to make a DPDA P_1 verify that $\vec{c'}$ is valid. P_1 can push the first ID $s011001$ onto its stack and check it against the second ID conveniently because it is in reversed form as $I_1^R = 10011p0$. The comma between them acts as a marker enabling the palindrome check to be started deterministically. All the instructions of M are hard-coded into the states of P_1 in a way that it can tell that the local change from $s0$ to $0p$ (or rather, $p0$ in reverse) is legal. But by the end of I_1^R its stack is empty, so it cannot right away check I_1^R against I_2. It can, however, start pushing I_2 in order to check against I_3^R. The upshot is that we need a second DPDA P_2 to skip over $I_0(x)$ and check I_1^R against I_2, I_3^R against I_4, and so on. To make a long story of details short:

Lemma: For any Turing machine M, V'_M equals the intersection of two DCFLs. Moreover, there is a computable mapping h such that $h(\langle M \rangle) = \langle P_1, P_2 \rangle$ giving DPDAs P_1 and P_2 such that $V'_M = L(P_1) \cap L(P_2)$.

What does the mapping h do? It sets up $L(M) = \emptyset \iff V'_M = \emptyset \iff L(P_1) \cap L(P_2) = \emptyset$. Thus it mapping-reduces E_{TM} to the problem

$E \cap_{DPDA}$:
INST: Two DPDAs P_1, P_2.
QUES: Is $L(P_1) \cap L(P_2) = \emptyset$?

Theorem: $E \cap_{DPDA}$ and the analogously-defined problems $E \cap_{DCFL}$ and $E \cap_{NPDA}$ and $E \cap_{CFL}$, are all undecidable---indeed their languages are not c.e.---because E_{TM} mapping-reduces to each of them. In particular, this means that although the emptiness problem for one CFL is decidable, whether the intersection of two CFLs is empty is undecidable.

How about the complement of V'_M, or even the complement of V_M without reversing IDs? Here, basically, a string $w = \langle I_0(x), I_1, I_2, I_3, \dots, I_{t-2}, I_{t-1}, I_t \rangle$ belongs to $\widetilde{V}_M$ if and only if either:

- it doesn't have the correct form as a sequence of IDs, or
- there is a screwup $I_{k-1} \nvdash I_k$ for some k: no legal instruction can execute the change, or some other character mismatch.

The first fault can always be detected on the fly---that's another reason we can often ignore the issue of "invalid codes" and assume a given string w has the right "angle-bracket" format. The main point is that if the second happens, it is enough that it jappens for *just one k*. Hence a nondeterministic PDA N can guess which k and then *verify* that there is a screwup. (If a branch of N guesses the wrong j, some other branch will guess the right k and accept; or if there is no screwup or other fault, all branches will correctly reject.) The ability of a PDA to detect a mismatch is related to the reason the *complement* of the double-word language *is* a CFL. Thus we conclude:

[The "jappens" above is an on-purpose typo, to illustrate a one-character screwup in lecture.]

Lemma: For any Turing machine M, $\widetilde{V}_M$ and $\widetilde{V'}_M$ are both CFLs. Moreover, there is a computable mapping h such that $h(\langle M \rangle) = \langle G \rangle$ giving a CFG G such that $L(G) = \widetilde{V}_M$, and likewise a mapping

$h'(\langle M\rangle) \ = \ \langle G'\rangle$ such that $L(G') = \widetilde{V'_M}$. ☒

This finally brings us to the proof of a long-promised fact:

Theorem: The ALL_{CFG} problem is undecidable.

Proof: $\langle M\rangle \in E_{TM} \ \equiv L(M) = \emptyset \iff V_M = \emptyset \iff \widetilde{L(G)} = \emptyset \iff L(G) = \Sigma^*$, where G is given by the computable mapping $h(\langle M\rangle)$. ☒

In fact, this is part of a "Meta-Theorem":

General Theorem: For any type of machine or "machine combo" Z that can verify computation traces, the E_Z problem ("emptiness problem for Z-machines") is undecidable. If the combo represents "broken traces" instead, then ALL_Z is undecidable.

The main instance of Z defined and proved in the text is a kind of machine called a **Linear Bounded Automaton** (**LBA**). This is defined as a Turing machine (nondeterministic or deterministic, which we can specify as **NLBA** or **DLBA**) that on any input x uses only the cells initially occupied by x (plus optionally the blanks to the left and right of x, or we can initialize with endmarkers $\wedge x\$$ or $\langle x\rangle$ instead). We will talk about LBAs next time anyway.

[actually, "next time" will be the continuation of this lecture, *attacca* as they say in music...]