CSE491/596 Lecture Mon. Oct. 12: Kinds and Examples of Reductions

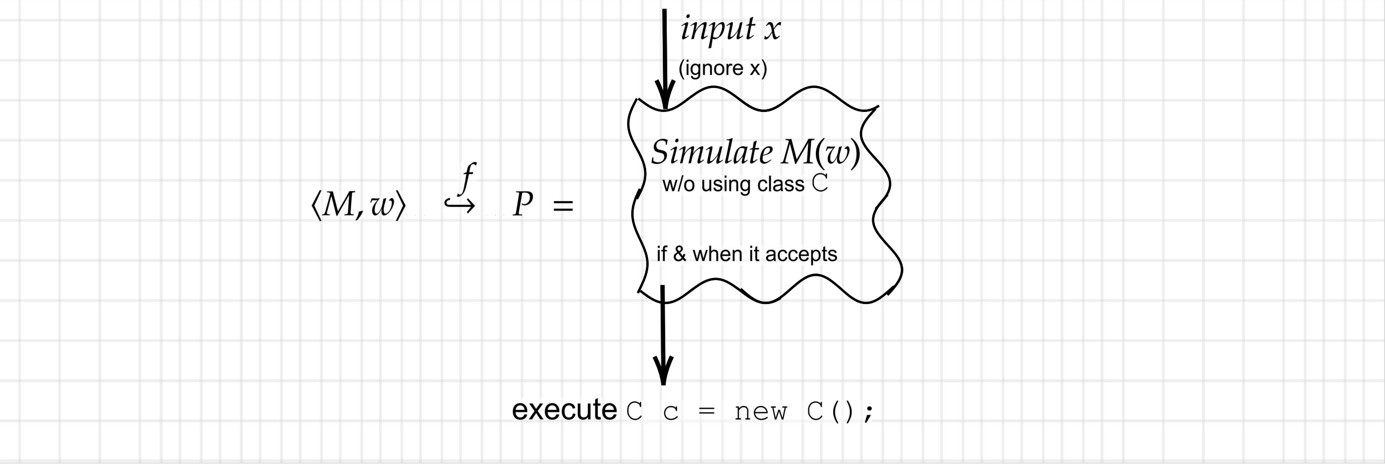
The first example is relevant to trying to cut down "code bloat" by removing unused classes from object-oriented code.

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 importa 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.

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 $\mathtt{C}$.

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 $\langle P, \mathtt{C}\rangle$
for $t = 1, 2, 3, 4, \ldots$ :
  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 $\langle P\rangle$.
```

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

Many "Waiting for Godot" kind reductions have a target problem that does involve a particular input x. My illustrating one where x is *quantified* rather than *given* sets up the second main kind of reduction, which I call the "All-Or-Nothing Switch." The switch, when combined with embellishments we'll see later, can apply to multiple problems at once. Here are three:

NE_{TM}

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

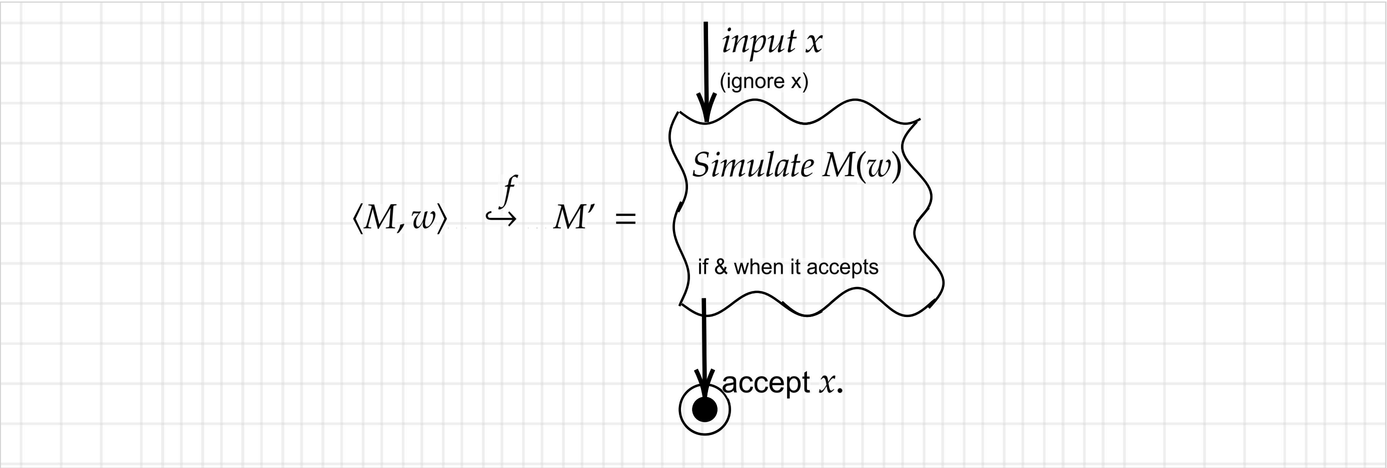
ALL_{TM}

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. 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 $f(\langle M, w\rangle) \notin Eps_{TM}$.

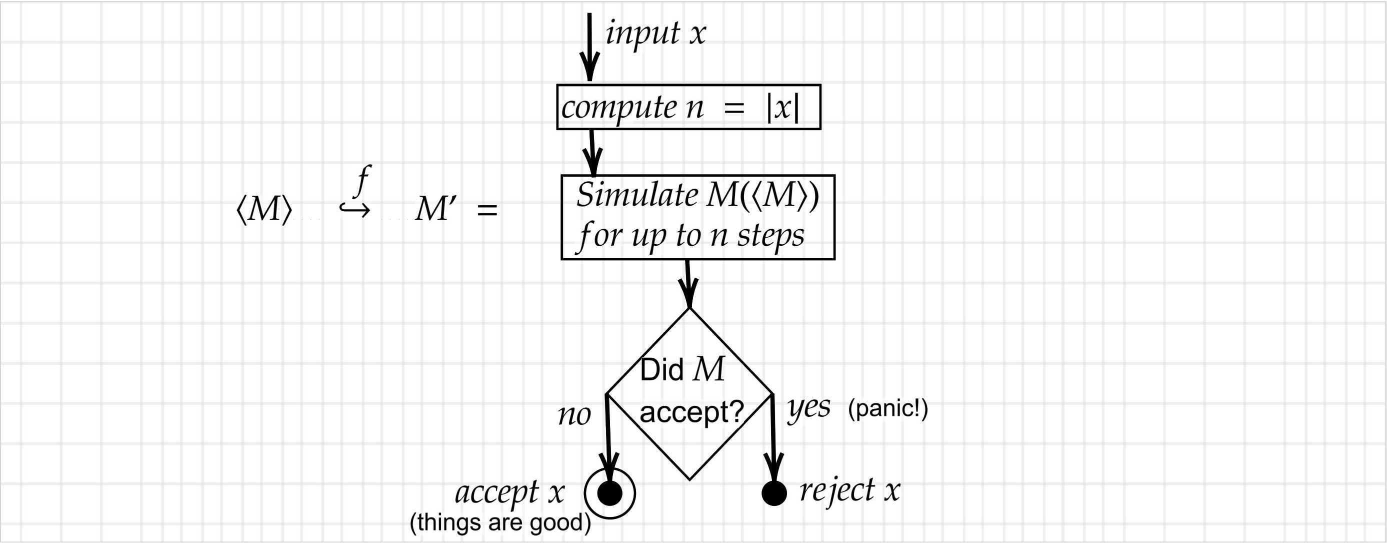
So we have simultaneously shown $A_{TM} \leq_m NE_{TM}$, $A_{TM} \leq_m ALL_{TM}$, and $A_{TM} \leq_m Eps_{TM}$.
Thus
all three of these problems and their languages are undecidable.

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 $t = 1, 2, 3, \dots$: 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....

[The transition to Wednesday's lecture will probably be here.]

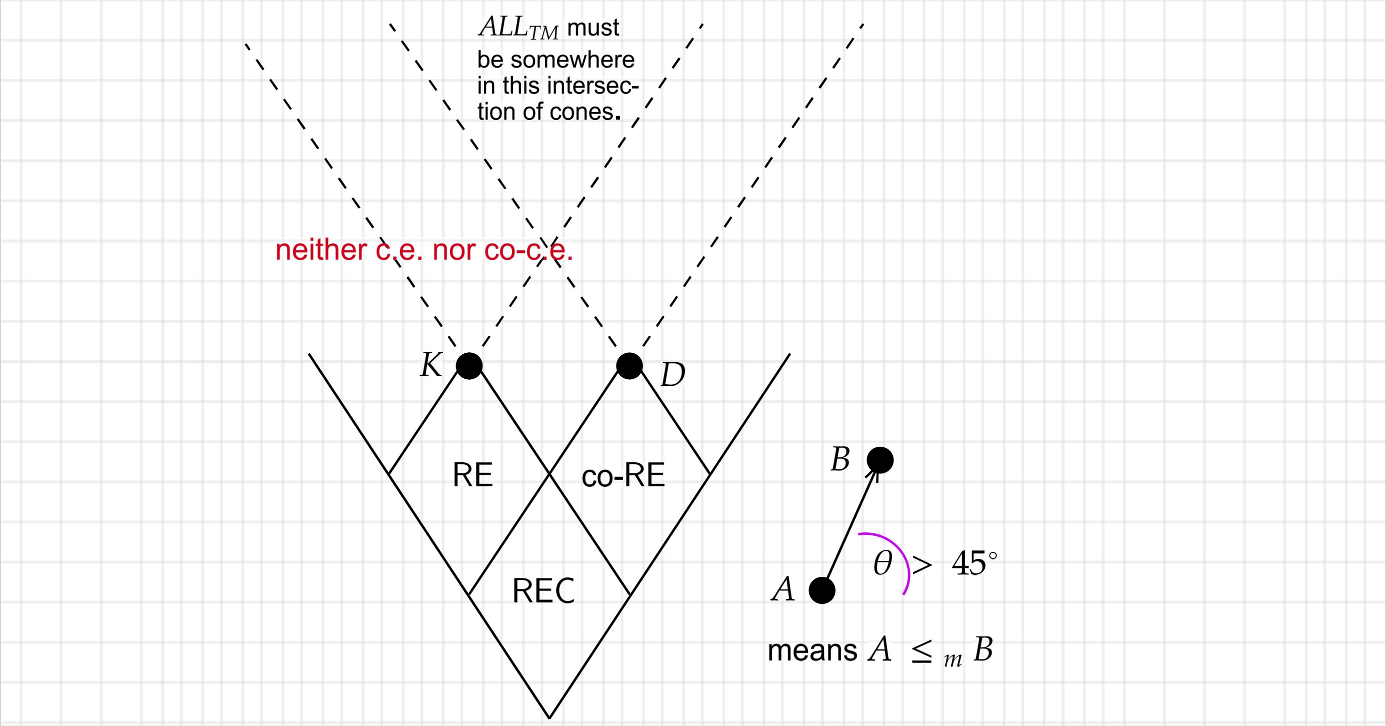
The third useful reduction technique is something I call the "Delay Switch." The intuition and attitude are the opposite of "Waiting for Godot" and the all-or-nothing switch. This time you picture your target machine M' or target program P as monitoring a condition that you hope *doesn't* happen, such 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:*



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 $A_{TM} \leq_m ALL_{TM}$, hence by transitivity, $K_{TM} \leq_m ALL_{TM}$. 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.

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.

[where the material goes from here: mapping-equivalence $\equiv_m$; hardness and completeness; more examples of neither-c.e.-nor-co-c.e.; Rice's theorem; Gödel's theorems as a presentation option; then on to computational complexity theory by the end of next week.]