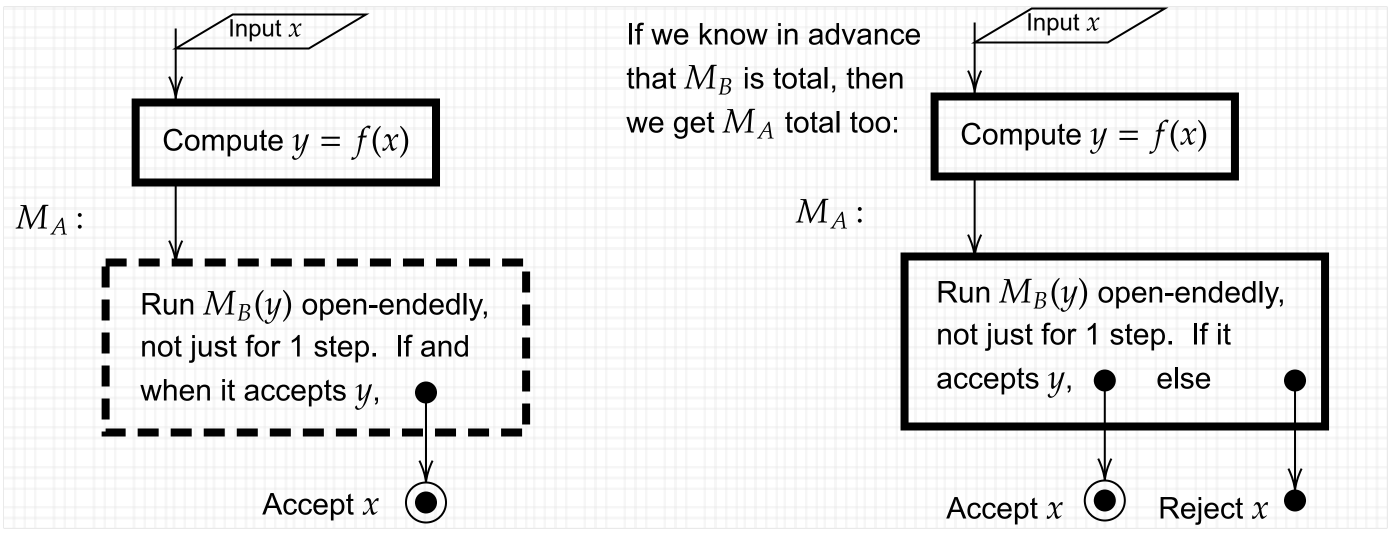
CSE396 Lecture Tue. 4/27: Mapping Reductions

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:



Input x
Compute $y = f(x)$
M_A :
Run $M_B(y)$ open-endedly, not just for 1 step. If and when it accepts y,
Accept x
If we know in advance that M_B is total, then we get M_A total too:
Input x
Compute $y = f(x)$
M_A :
Run $M_B(y)$ open-endedly, not just for 1 step. If it accepts y, else
Accept x
Reject x

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 $x \in \Sigma^*$, $x \in A \iff f(x) \in B$.

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 $x \in \Sigma^*$, $x \in A \iff f(x) \in B$. This is written $A \leq_m B$.

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 1: 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 $x \in A \iff f(x) \in B$ is logically equivalent to $x \in \widetilde{A} \iff f(x) \in \widetilde{B}$. 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 2: 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. $\boxtimes$

Examples of Mapping Reductions

We have already seen numerous examples of mapping reductions---just not yet labeled as such:

1. The mapping $f_1(\langle M \rangle) = \langle M' \rangle$, where M is a DFA and M' is obtained by interchanging its accepting and rejecting states, reduced the ALL_{DFA} problem to the E_{DFA} problem. This was a "positive use": because E_{DFA} has a decider, we got a decider for ALL_{DFA}.
    (a) We further solved E_{DFA} by appeal to what I called NE_{DFA} but this was not by a mapping of instances; it was by re-interpeting what "yes" and "no" meant.
    (b) Super-technically, we must define $f_1(w)$ for all $w \in \Sigma^*$. If w is not a valid code of a DFA, then we can recognize that fact and map $f_1(w) = \langle M_0 \rangle$, where M_0 is a fixed DFA for which the answer to the **target problem** is "no." Henceforth, we allow assuming that the input is a valid code. This is **not** the same as assuming it is a case for which the answer to the **source problem** is "yes."

2. The mapping $f_2(\langle M_1, M_2 \rangle) = \langle M_3 \rangle$, where M_1 and M_2 are DFAs and M_3 is their Cartesian product with XOR as the operation, reduced the EQ_{DFA} problem to the E_{DFA} problem. The reduction f_2 was **correct** because $L(M_1) = L(M_2) \iff L(M_3) = \emptyset$.

3. The problem EQ_{NFA} takes two NFAs N_1, N_2 and asks whether $L(N_1) = L(N_2)$. It can be reduced to EQ_{DFA} by the mapping $f_3(\langle N_1, N_2 \rangle) = \langle M_1, M_2 \rangle$, where M_1 and M_2 are the conversion of N_1 and N_2 into DFAs. The function f_3 can take a long time to compute in many NFA cases, but it is *computable* and reduces EQ_{NFA} to EQ_{DFA}. Because reductions are transitive, the composition $f_2 \circ f_3$ is a computable function that reduces EQ_{NFA} all the way to E_{DFA}, and that gives us a decider for EQ_{NFA}. But because of the use of NFA-to-DFA, neither f_2 nor the decider we get is "polynomial-time" efficient.

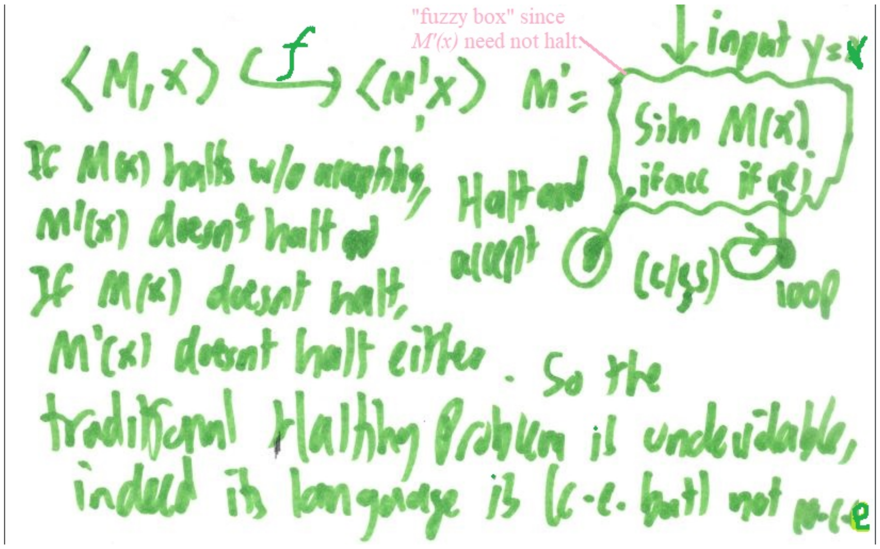
4. The mapping $f_4(\langle M \rangle) = \langle M, M \rangle$ reduces K_{TM} to A_{TM}. Thus:
    (a) By Theorem 1(b), because the A_{TM} language is c.e., we got that K is c.e.
    (b) But by Theorem 2(a), because K_{TM} is undecidable, we got that A_{TM} is undecidable.

The mapping f_4 is especially simple: it basically just doubles the given string. The f_4 example shows how reductions can be used both "positively" (for ***upper bounds*** like "is c.e.") and "negatively" (for ***lower bounds*** like "is not decidable"). Here is another example of the latter:

Example: $A_{TM} \leq_m HP_{TM}$ via $f(\langle M, w\rangle) = \langle M', w\rangle$, where M' is transformed from M as follows:

- We may presume M is in the text's normal form with q_{acc} and q_{rej} as its only halting states.
- Make M' by adding a loop $(q_{rej}, c/c, S, q_{rej})$ for every char c in the work alphabet Γ of M.
- [Super-technically, we can bolt on a new rejecting state q'_{rej} that is never reached in order to "restore" the test's normal form for cosmetic purposes.]

Here is a little picture that shows just about everything we need to say (never mind that it says "x" instead of "w"):


$\langle M, x\rangle \xrightarrow{f} \langle M', x\rangle$ $M' =$
"fuzzy box" since $M'(x)$ need not halt.
input $y = x$
Sim $M(x)$
if acc
if rej
Halt and accept
$(c/c, S)$
loop
If M(x) halts w/o accepting, M'(x) doesn't halt and If M(x) doesn't halt, M'(x) doesn't halt either. So the traditional Halting Problem is undecidable, indeed its language is (c.e. but) not co-c.e.

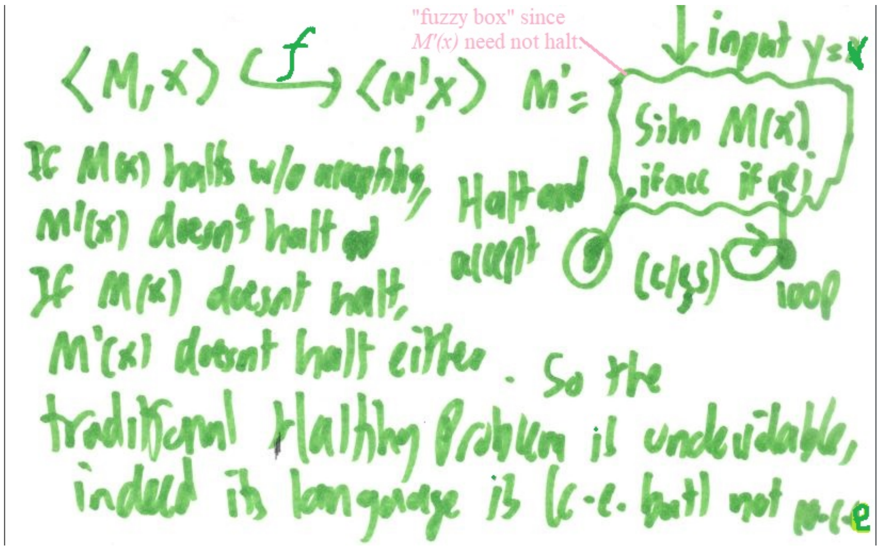
I use the acronym "CCC" for the three things one needs to say:

1. **C**onstruction: how M' is built from M, so as to define $f(\langle M, w\rangle) = \langle M', w\rangle$.
2. **C**omputability: Often we could say this is "obvious", but it helps to give knowledge of the **c**omplexity of the reduction too. E.g., the mappings f_1 and f_4 above are super-simple, while the Cartesian-product f_2 is not so simple---but its quadratic time counts as "polynomial time." The mapping f_3 involves converting any given NFAs into DFAs, so it is exponential time, but still counts as computable. This mapping f is super-simple.
3. **C**orrectness: the "$x \in A \iff f(x) \in B$" part, where here the "x" is $\langle M, w\rangle$. It often helps to break the " $\iff$ " into two implications going from the source problem to the target problem. So to show that M accepts $w \iff M'$ on input w halts, we verify:.

Here, "A" is the (language of the) A_{TM} problem, "x" is is $\langle M, w\rangle$ as an instance of the A_{TM} problem, "B" is the HP_{TM} problem, "$f(x)$" is a similarly-structured instance $\langle M', w\rangle$ of the HP_{TM} problem (where the w part happens to be the same), and the "$x \in A \iff f(x) \in B$" is the statement $\langle M, w\rangle \in A_{TM}$ (i.e., $\langle M, w\rangle$ is in the A_{TM} *language*) if and only if $\langle M', w\rangle \in HP_{TM}$ language. To verify this,

M accepts $w \implies M(w)$ goes to $q_{acc} \implies M'(w)$ goes to its own q'_{acc} as well $\implies M'(w)\downarrow$.
M does not accept $w \implies$ either $M(w)\uparrow$ or $M(w)$ goes to $q_{rej} \implies M'(w)\uparrow$ either way.

Put together, we have that $\langle M, w\rangle \in A_{TM} \iff \langle M', w\rangle \in HP_{TM}$. ☒


"fuzzy box" since $M'(x)$ need not halt
$\langle M,x\rangle \xrightarrow{f} \langle M',x\rangle$ $M'=$
input $y=x$
Sim $M(x)$
if acc
if rej
Halt and accept
$(c/c,S)$
loop

If M(x) halts w/o accepting, M'(x) doesn't halt and If M(x) doesn't halt, M'(x) doesn't halt either. So the traditional Halting Problem is undecidable, indeed its language is (c.e. but) not co-c.e

This finally shows that the classic Halting Problem is undecidable.

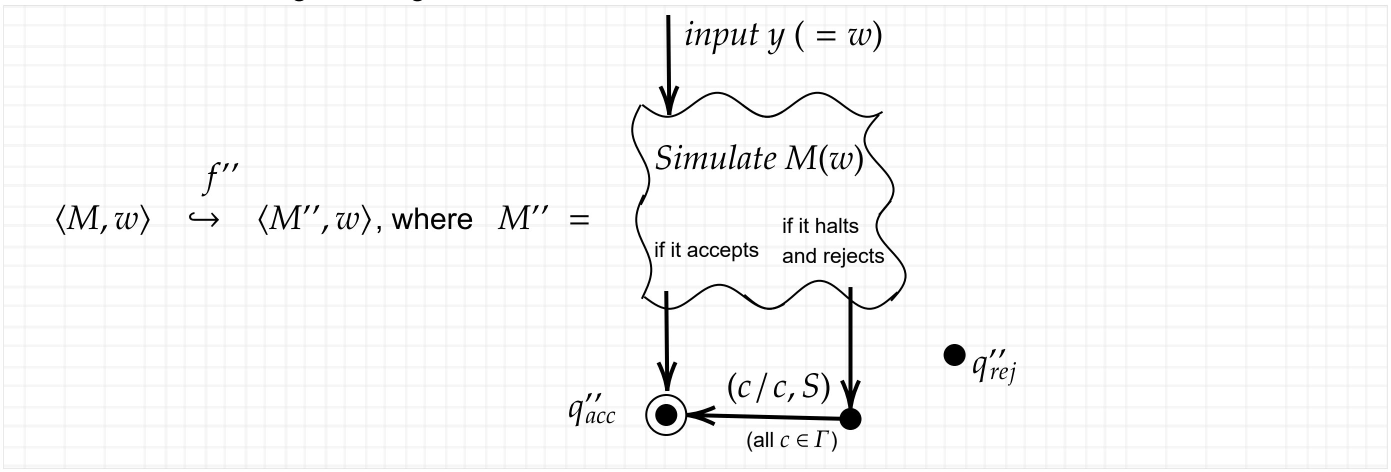
[As a footnote, for the second part of correctness, we could prefer using the *converse* rather than the *inverse* of the first part:

$M'(w)\downarrow \implies M'$ accepts w (because q'_{acc} is the only place M' can halt $\implies M$ accepts w too. Thus $w \in L(M) \iff M'(w)\downarrow$ so the reduction is correct. ☒

But I prefer always keeping the flow going from the source problem to the target problem. Getting the "from-to" logic backwards is one of the most common mistakes with reductions.]

An important self-study question: **Does the same f also reduce HP_{TM} back to A_{TM}?**

That would need us to say that $M(w)\downarrow \iff M'$ **accepts** w. That is not what happens in the code constructed by the above f mapping. But we can make a new mapping f'' with a different "code modification" that brings this logic about:



This mapping f'' is equally super-simple to compute: it adds arcs from the old q_{rej} to the accepting state rather than loops at q_{rej}. The correctness logic is:

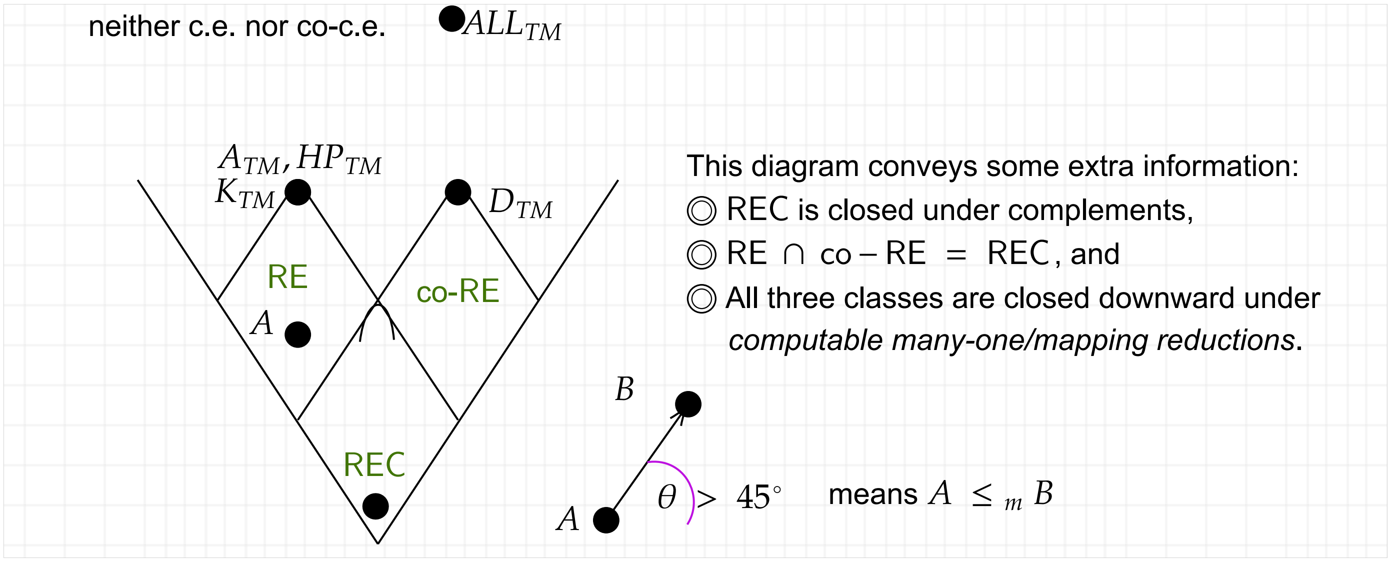
$M(w)\downarrow \implies M(w)$ goes to q_{acc} or to $q_{rej} \implies M''(w)$ goes to q''_{acc} either way $\implies M''$ accepts w.
$M(w)\uparrow \implies M''(w)$ does not halt either, so M'' does not accept w.

This entitles us to say $\langle M, w\rangle \in HP_{TM} \iff \langle M'', w\rangle \in AP_{TM}$. ☒

Thus, in fact, the Acceptance and Halting Problems are **mapping equivalent**, for which we write

$$A_{TM} \equiv_m HP_{TM}.$$

This underscores why, historically, "accepting" and "halting" were considered the same thing, and why accepting states are called "final" states. We can show mapping equivalence graphically by putting the "dots" for each language in the same place in our diagrams:



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*.

Actually, K_{TM} is mapping-equivalent to A_{TM} as well. This may seem surprising because K_{TM} has "less stuff": its **instance type** is "just an M" rather than "an M and a w". The converse reduction $A_{TM} \leq_m K_{TM}$ will be an incidental benefit of the "All-Or-Nothing Switch" *reduction design pattern* below. Thus we can move its dot into the same location at the very top of RE. Why the very top? It's because every c.e. language A accepted by a fixed machine M_A mapping reduces to A_{TM} via the "super-simple" mapping

$$f(x) = \langle M_A, x\rangle,$$

which is correct because $x \in A \iff M_A$ accepts $x \iff \langle M_A, x\rangle \in A_{TM}$. This state of affairs is summarized by the following key definition.

Definition: A language B is **complete** for a class $\mathbb{C}$ of languages (such as $\mathbb{C} = \text{RE}$) **under** a reducibility
relation $\leq_r$ (such as computable mapping reductibility $\leq_m$) if:

1. $B \in \mathbb{C}$, and
2. for all languages $A \in \mathbb{C}$, $A \leq_r B$.

If only the latter holds, we say that B is **hard** for $\mathbb{C}$ under the reducibility. In the case where $\mathbb{C}$ is RE, we also say that B is RE**-complete** (or RE**-hard** if we don't have $B \in \text{RE}$), and the synonyms **r.e.-complete**, **c.e.-complete** or just "complete" come into play (but not "recognizably complete").

Thus A_{TM}, HP_{TM}, and K_{TM} are all complete for RE. Moreover, D_{TM} is complete for co-RE. In point of fact, they are all complete under "super-simple" reductions---as we will shortly see for K_{TM} while doing hardness for NE_{TM}, K_{TM}, and ALL_{TM} in one swoop. We will also see that ALL_{TM} is not in RE, so it is RE-hard without being RE-complete. The class REC should actually "collapse to a single point" under $\leq_m$ because of the following trivial theorem:

Theorem 3: All decidable languages are $\equiv_m$-equivalent (technically except for $\emptyset$ and Σ^*).

Proof: Suppose A and B are decidable, and B is neither is $\emptyset$ or Σ^*. Then there is a "yes string" $y_0 \in B$ and a "no string" $z_0 \notin B$. By A being decidable, we can take a total TM M_A such that $L(M_A) = A$. Then define the mapping f as follows, for all $x \in \Sigma^*$:

$$f(x) = \begin{cases} y_0 & \text{if } M_A \text{ accepts } x \\ z_0 & \text{if } M_A \text{ rejects } x \end{cases}.$$

Because M_A is total, we can compute $f(x)$ in all cases, and clearly $x \in A \iff f(x) \in B$ by the choice of the two strings. Since the exception of $\emptyset$ and Σ^* technically reducing only *from* themselves is often ignored, we can say all decidable sets are trivially complete for REC. ☒

But under simpler reductions than $\leq_m$, such as **polynomial-time mapping reducibility** $\leq_m^p$, the equivalence no longer holds globally---e.g., if M_A does not run in polynomial time. The classes REC, RE, and co-RE all "keep their shape" under $\leq_m^p$ (and in fact, basically every reduction seen in this course except ones like f_3 needing NFA-to-DFA will be computable in quadratic time at worst). Indeed, A_{TM}, HP_{TM}, K_{TM}, etc. are all complete under $\leq_m^p$, though next wee what we will care about is completeness for the class NP under $\leq_m^p$. The one place where the diagram misleads is that REC does *not* have complete sets under $\leq_m^p$, which we try to signify by putting a little round arc under its "peaked top."

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.

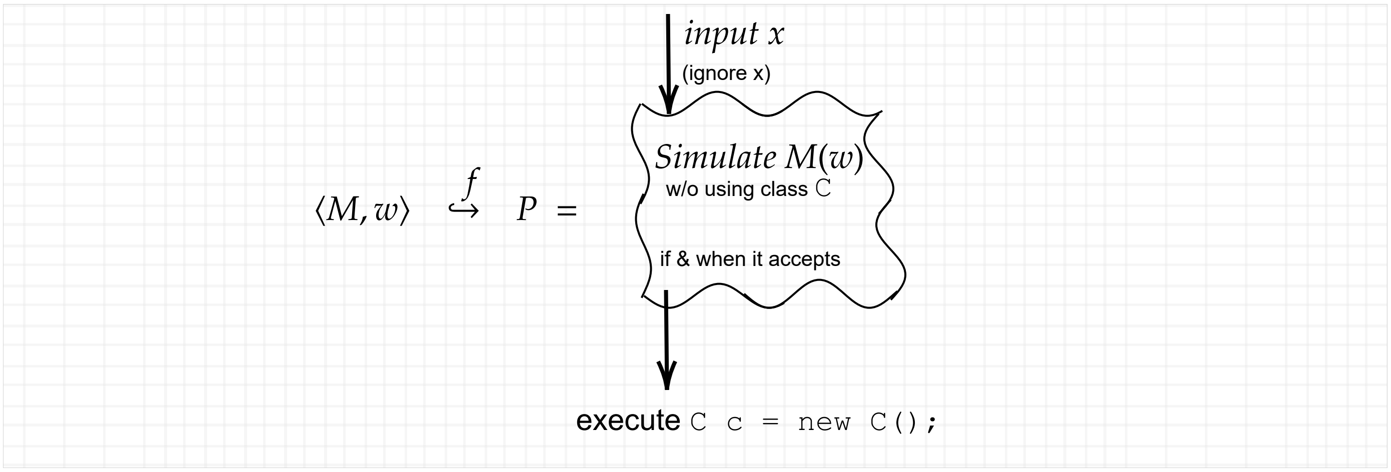
USEFULCLASS

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 $\langle P, \mathtt{C}\rangle$
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** $\langle P, \mathtt{C}\rangle$.
```

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.

II The "All-or-Nothing Switch"

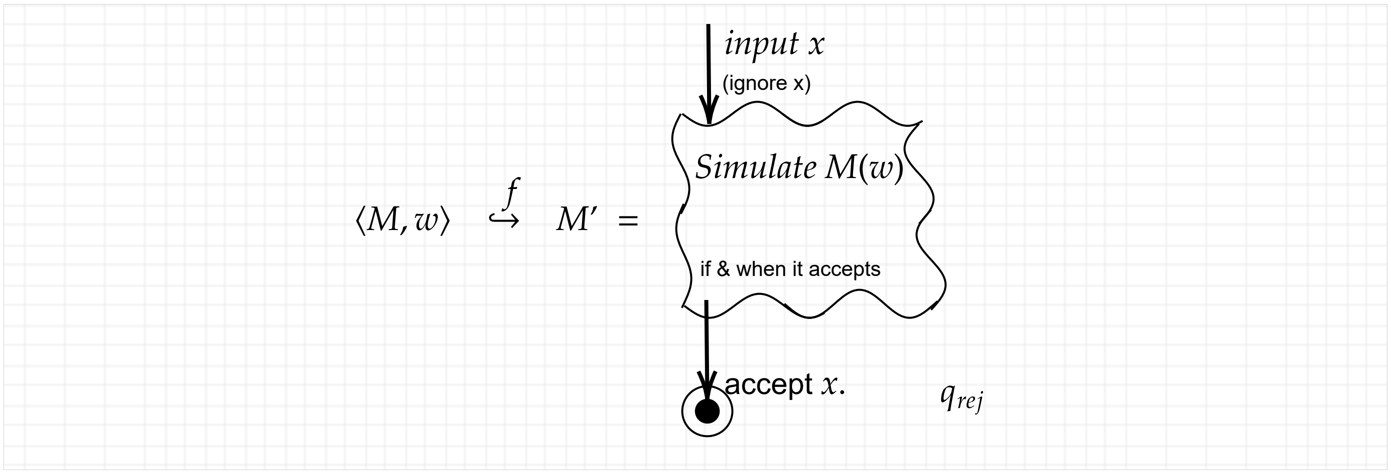
This actually builds on the "wait-for-it" kind of reductions. Note that HP_{TM} had an instance type that specified both "an M and an input x" but UsefulClass had the instance type "just a program P" where the x part was *quantified* as "Does there exist an x such that $P(x)$...?" When there is flexibility 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. 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 $f(\langle M, w\rangle) \notin Eps_{TM}$.

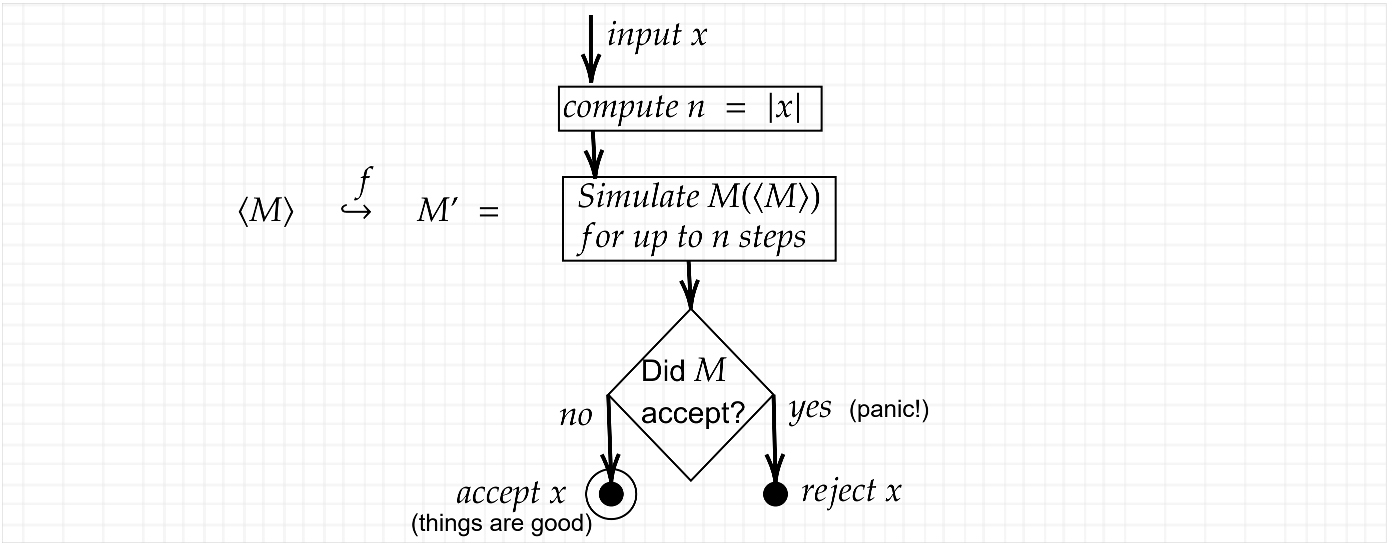
So we have simultaneously shown $A_{TM} \leq_m NE_{TM}$, $A_{TM} \leq_m ALL_{TM}$, and $A_{TM} \leq_m Eps_{TM}$.
Thus
all three of these problems and their languages are undecidable.

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 wn 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:*

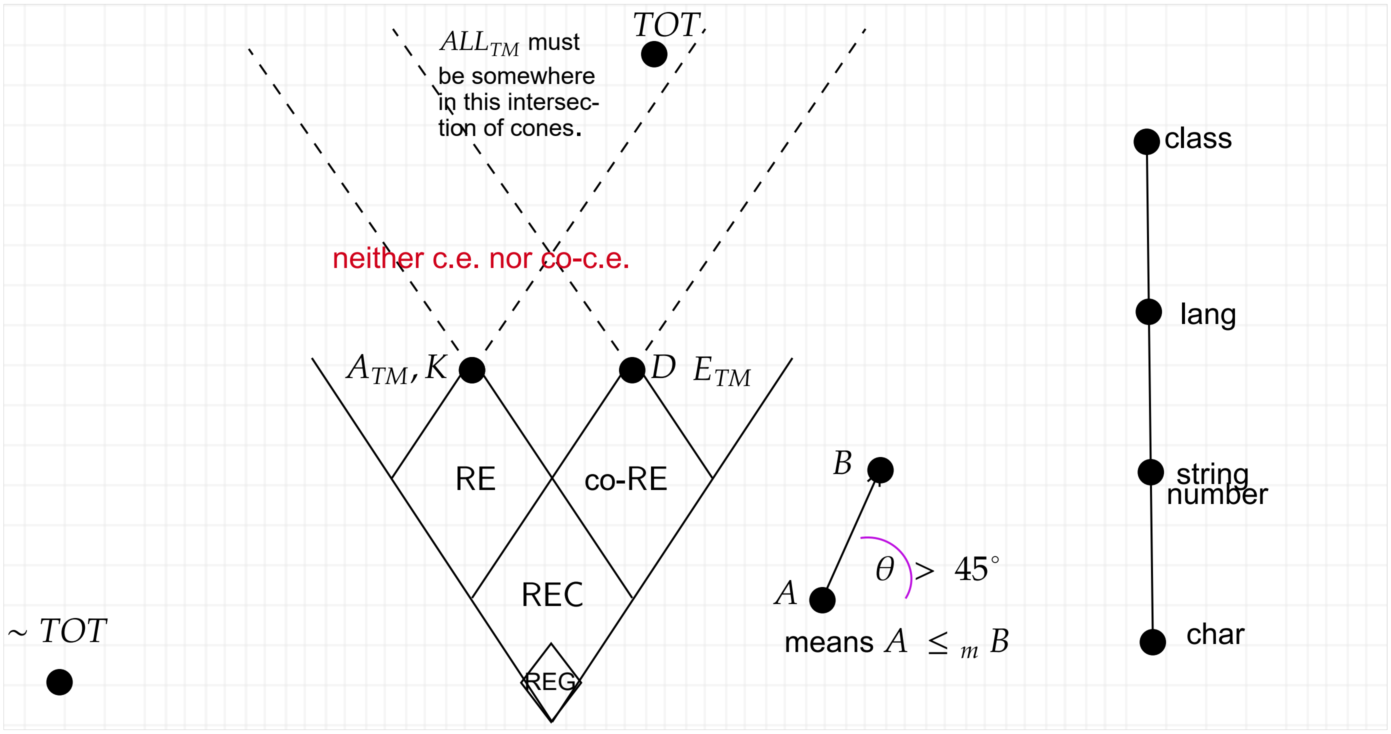


input x
compute $n = |x|$
$\langle M \rangle \overset{f}{\hookrightarrow} M' =$
Simulate $M(\langle M \rangle)$ for up to n steps
Did M accept?
no
yes (panic!)
accept x
(things are good)
reject x

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. For example, consider

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

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 $\langle M_1, M_{all}\rangle \in EQ_{TM}$ *if and only if* $\langle M_1\rangle \in ALL_{TM}$. And $ALL_{TM} \leq_m EQ_{TM}$ by the simple 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.

[where the material goes from here: more examples of neither-c.e.-nor-co-c.e.; then undecidability via TM computation histories in the latter part of section 5.1, finally saying why ALL_{CFG} and (equivalently) ALL_{PDA} are undecidable.]