CSE396 Spr26 Lecture Tue. Apr. 14: RE, Co-RE, and Mapping Reductions (Ch. 5)

By way of review, we can also define D and other problems with regard to other progarmming formalisms besides Turing machines, for instance:

$D_{Java} = \{p : p \text{ compiles in Java to a program } P \text{ such that } P(p) \text{ does not execute } \texttt{System.exit(0)}\}$.
$D'_{Java} = \{p : p \in D_{Java} \text{ or } p \text{ does not compile in Java at all}\}$.
$K_{Java} = \{p : p \text{ compiles in Java to a program } P \text{ such that } P(p) \text{ does execute System. exit(0)}\}$
Then K_{Java} is literally the complement of the language D'_{Java}.

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 $\langle M \rangle$?

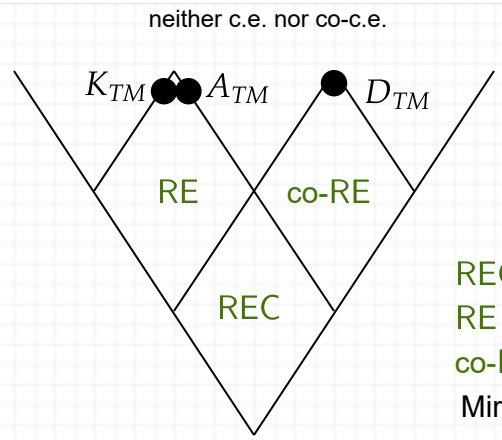
The language, also called K_{TM} or often just "**K**", is $\{\langle M \rangle : \langle M \rangle \in L(M)\}$. This language is not literally the complement of D_{TM} because of strings p' that are not valid codes $\langle M \rangle$ of Turing machines. But K_{TM} is the literal complement of the augmented 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

Some Theorems

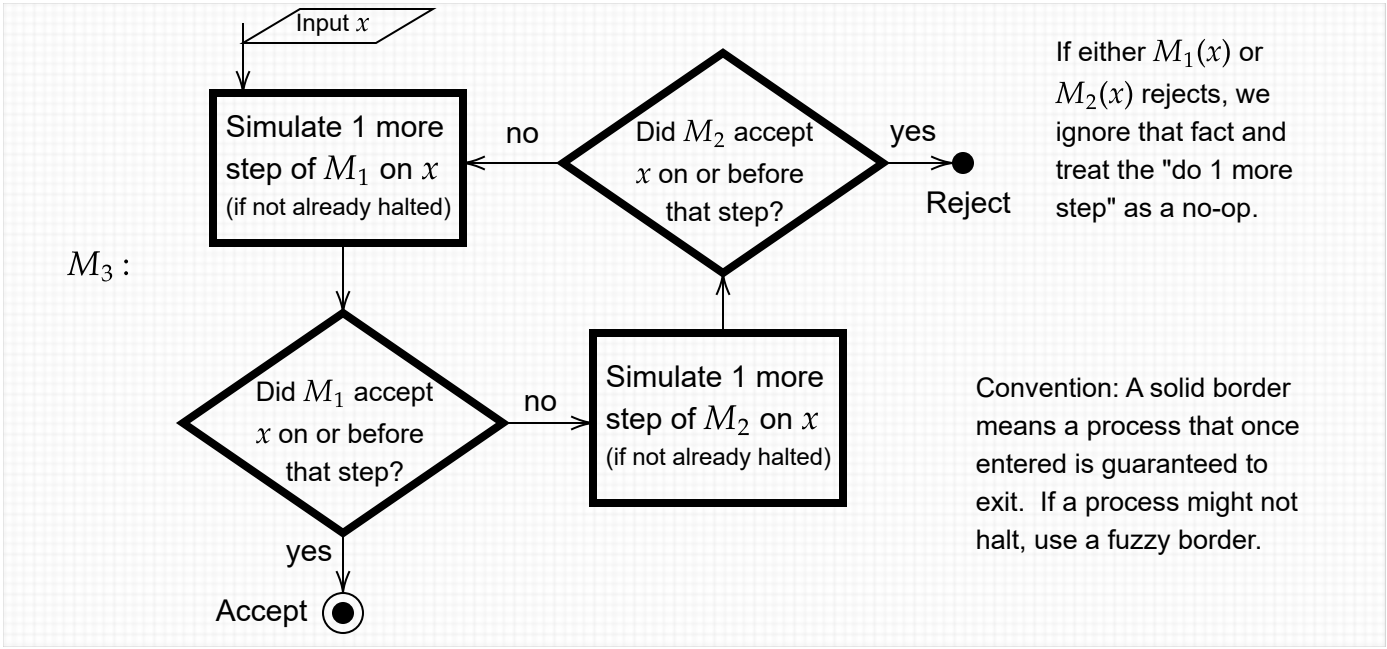
The fact of the complement of a decidable language being decidable can now be stated as a closure property: the class REC (like the class REG of regular languages and the class DCFL of DCFLs, but not like the class CFL of CFLs) is closed under complements. But we can prove something more:

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

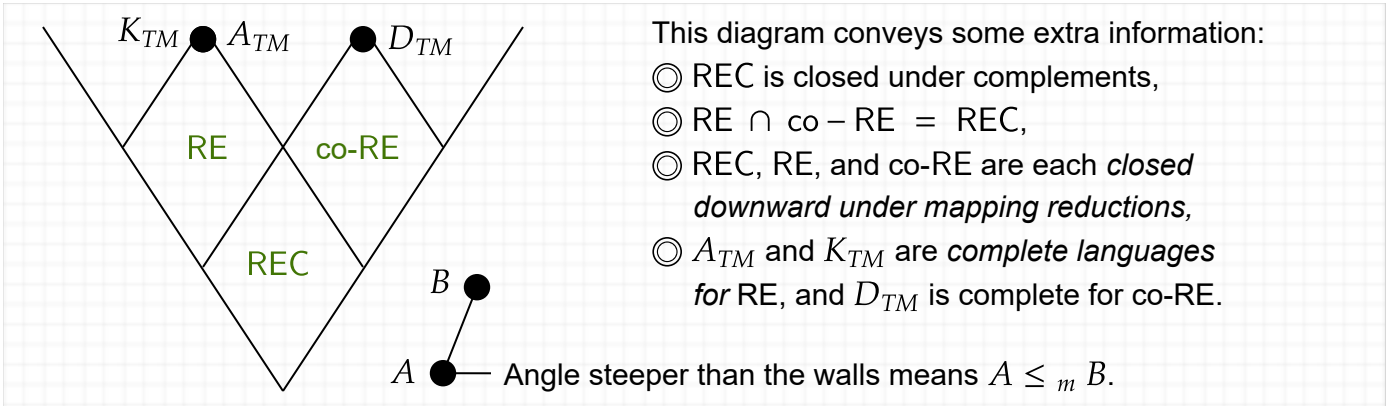
Put another way, L is decidable iff L is both c.e. and co-c.e. This can be summarized in a Venn diagram manner as REC = RE ∩ co - RE.

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:

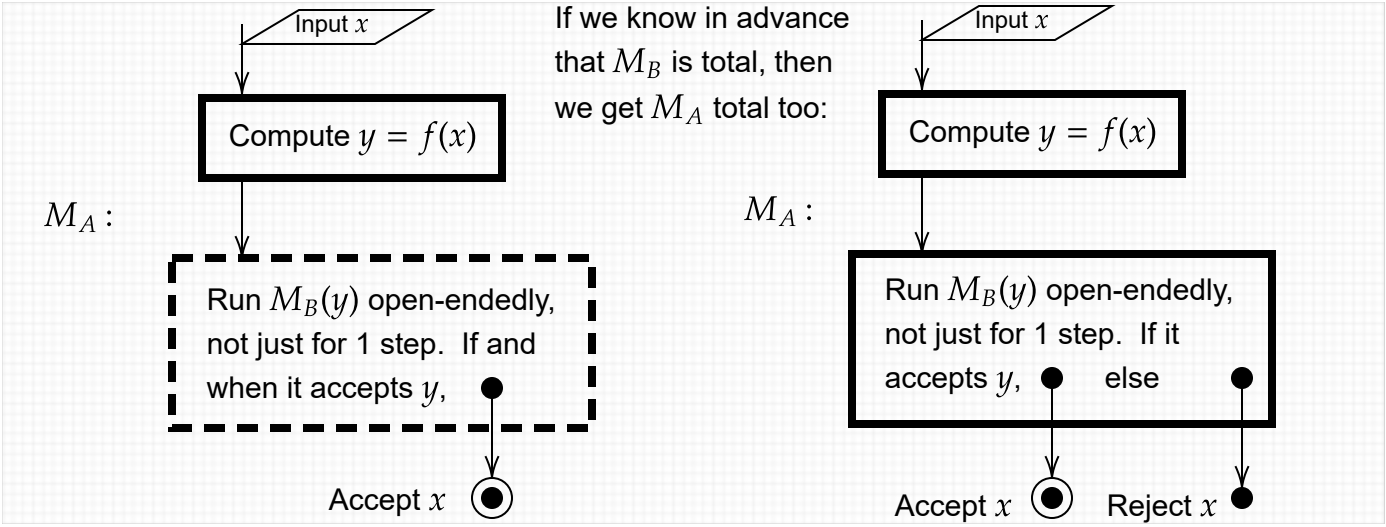


Besides the text's pseudocode, flowchart diagrams are another acceptable way to specify Turing machines. That is, they specify programs, and programs are already equivalent to TMs. This TM M_3 is total because for any x, it will exit either when M_1 accepts x or when M_2 accepts x---and one of them does. 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. ☒



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:



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: 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. $\boxtimes$

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. ☒

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 $L(M_1) = L(M_2) \iff L(M_3) = \varnothing$.

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.

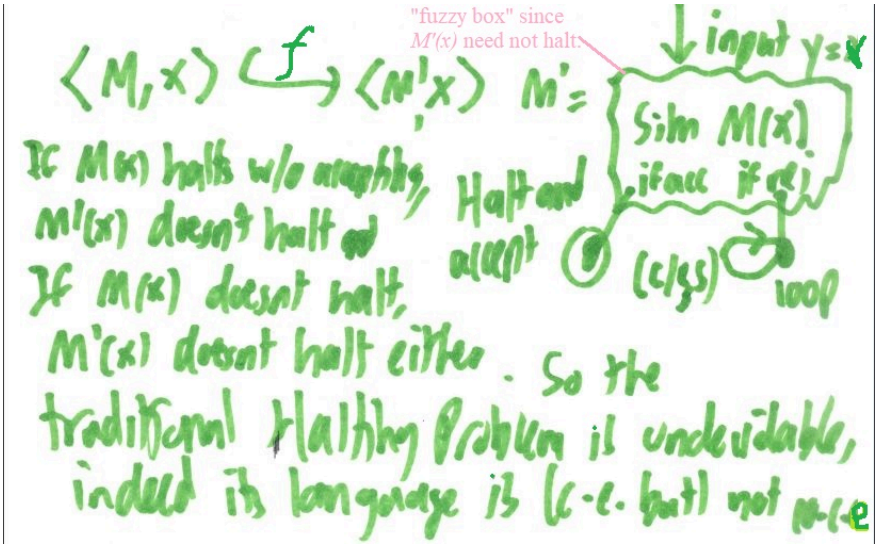
4. The mapping $f_4(\langle M \rangle) = \langle M, M \rangle$ reduces K_{TM} to A_{TM}. Thus:
   (a) By Theorem 1(b), because the A_{TM} language is c.e., we got that K is c.e.
   (b) But by Theorem 2(a), because K_{TM} is undecidable, we got that A_{TM} is undecidable.

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"):



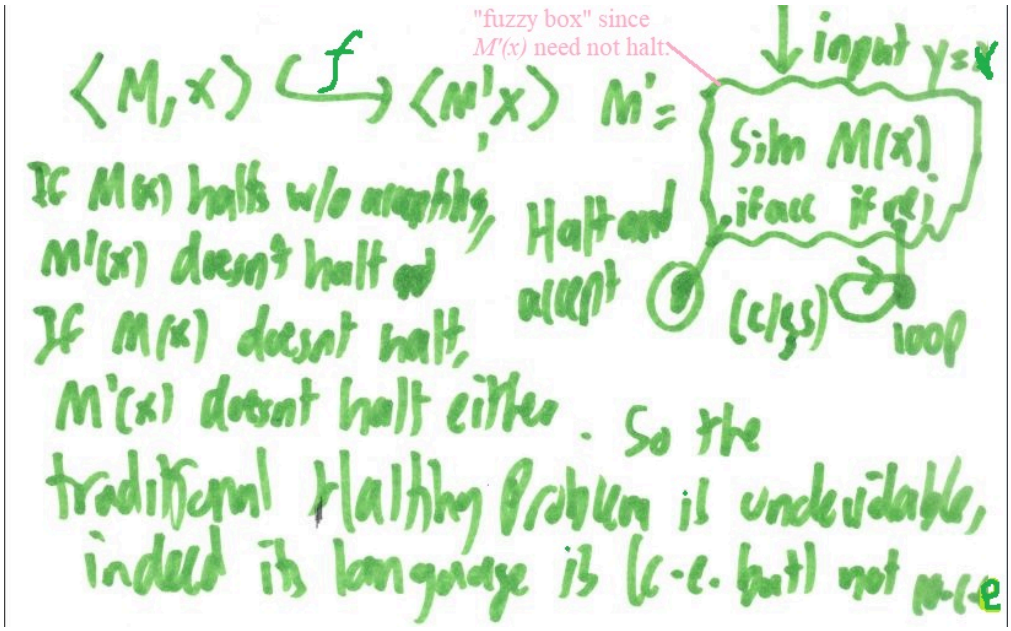
I use the acronym "CCC" for the three things one needs to say:

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 $\langle M, w\rangle \in A_{TM} \iff \langle M', w\rangle \in HP_{TM}$. $\boxtimes$



This finally shows that the classic Halting Problem is undecidable.

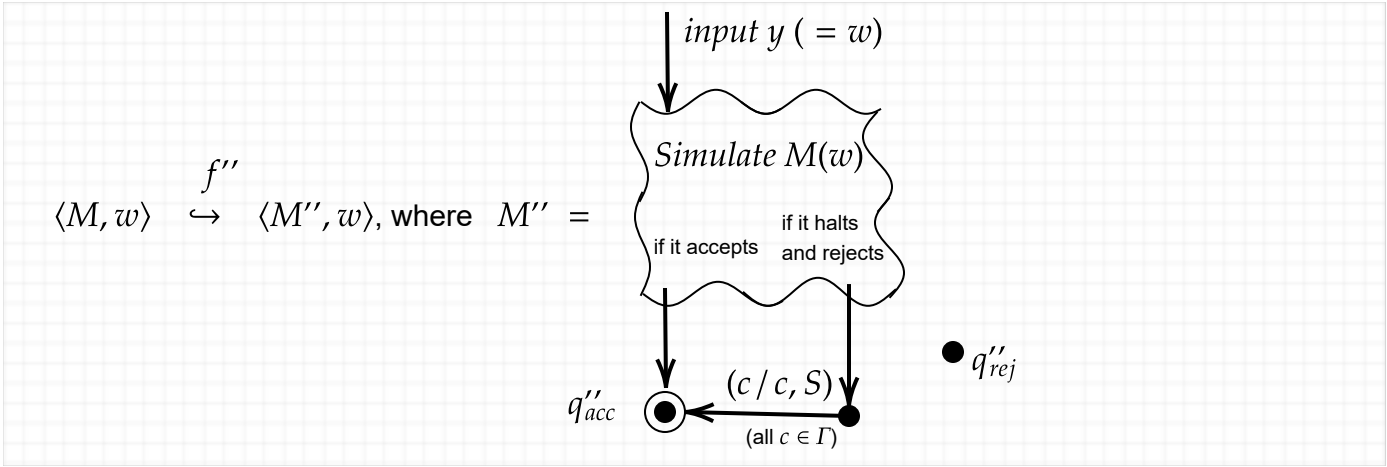
[As a footnote, for the second part of correctness, we could prefer using the *converse* rather than the *inverse* of the first part:

$M'(w)\downarrow \implies M'$ 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:

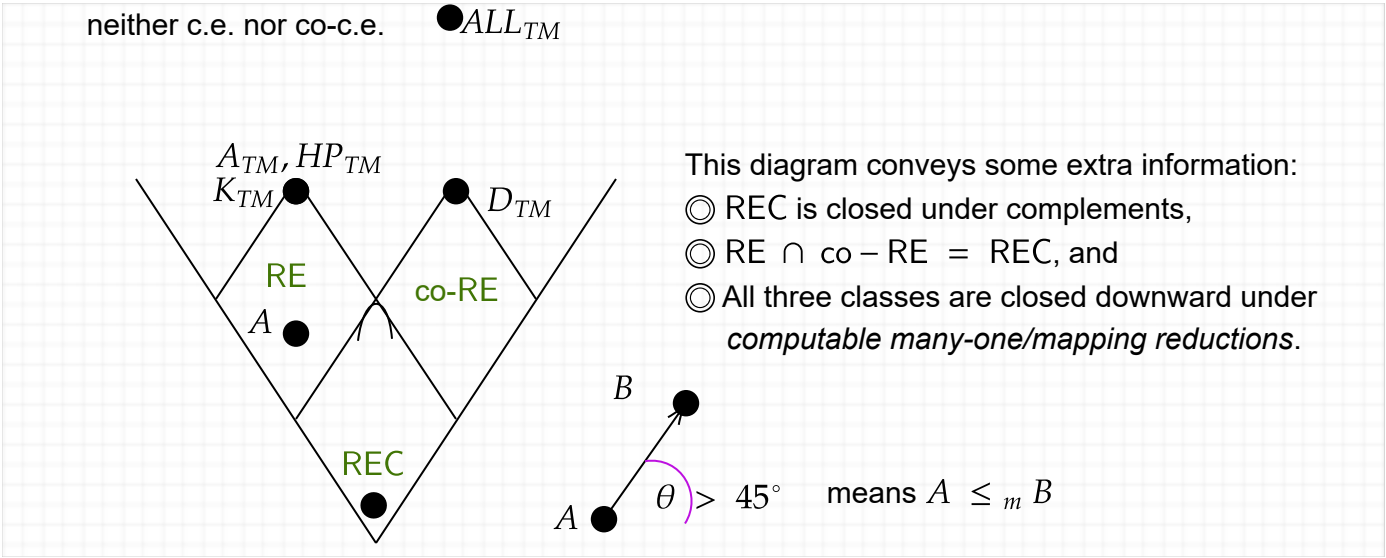
$M(w)\downarrow \implies M(w)$ goes to q_{acc} or to $q_{rej} \implies M''(w)$ goes to q''_{acc} either way $\implies M''$ accepts w.
$M(w)\uparrow \implies M''(w)$ does not halt either, so M'' does not accept w.

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:



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 C of languages (such as $\mathrm{C} = \mathrm{RE}$) **under** a reducibility relation $\leq_r$ (such as computable mapping reductibility $\leq_m$) if:

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

If only the latter holds, we say that B is **hard** for C under the reducibility. In the case where C is RE, we also say that B is RE**-complete** (or RE**-hard** if we don't have $B \in \mathrm{RE}$), 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.