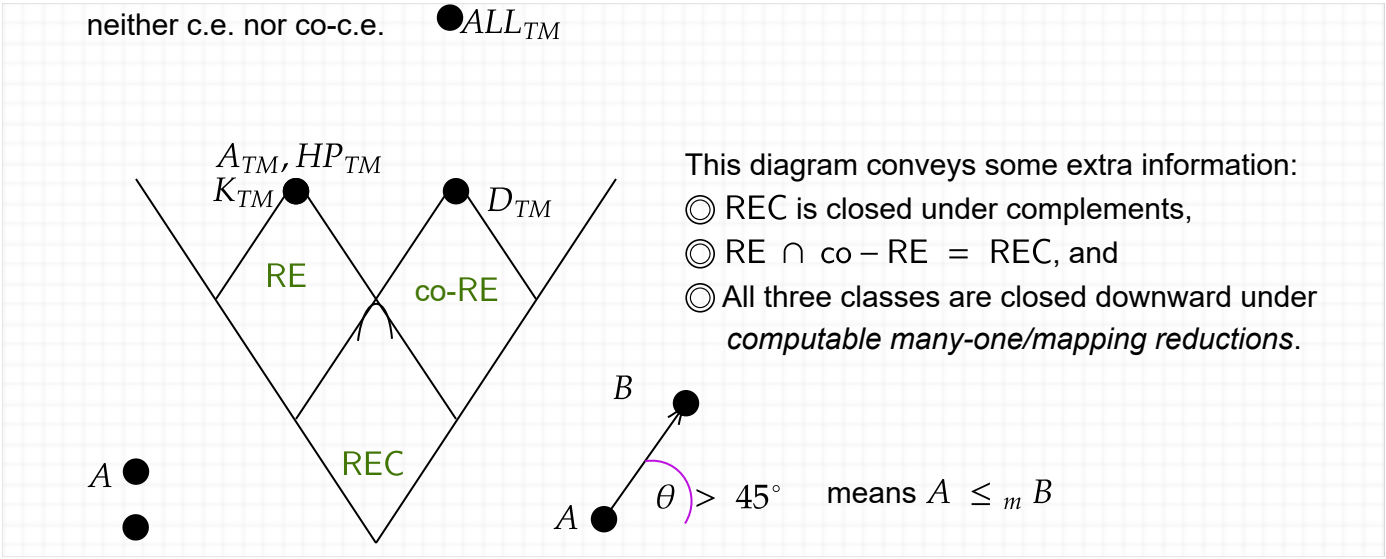
CSE396 Spr26 Lecture Thu. Apr. 23: Completeness and More Mapping Reductions

The last lecture showed that A_{TM} and HP_{TM} mapping-reduce to each other, and likewise A_{TM} reduces back to K_{TM}, besides $K_{TM} \leq_M A_{TM}$ which we showed first.

Definition: If $A \leq_m B$ and $B \leq_m A$ then we write $A \equiv_m B$ and say that the languages A and B are **mapping-equivalent**.

Thus $A_{TM} \equiv_m HP_{TM} \equiv_m K_{TM}$. Mapping equivalence is reflexive, symmetric, and (because reductions are transitive) transitive, so it is indeed an equivalence relation. By the logic of the "cone diagrams," mapping-equivalent languages occupy the same spot:



Why put A_{TM} at 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 C = RE) **under** a reducibility relation $\leq_r$ (such as computable mapping reductibility $\leq_m$) if:

1. $B \in$ C, and
2. for all languages $A \in$ 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 $B \in$ 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. 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. $\boxtimes$

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."

We've explained why the "yes-cases" languages of the A_{TM} and HP_{TM} and K_{TM} languages **are** computably enumerable. But what about the NE_{TM} language? How can we build a program P that will always accept (the code $\langle M \rangle$ of) a given Turing machine M when $L(M) \neq \emptyset$? And only when $L(M) \neq \emptyset$. We can envision P incorporating the *Turing Kit* simulator and first trying whether a given M, without loss of generality having $\Sigma = \{0, 1\}$, accepts the empty string, then whether it accepts the string "0", next "1" and "00", and so on. The hitch is that if M on one string never halts, you never get to try the next string... We get the same issue with the "USEFUL CLASS" problem in the previous lecture.

The fix is to organize all the runs on inputs to work in parallel in a staggered manner that is commonly called "dovetailing." I like to picture dovetailing as occurring inside an enclosing arbitrary time-allowance loop. In this case of "USEFUL CLASS", note that we are trying to analyze just a program P and the object class C in question---not any particular inputs to P. So here is the dovetailing loop:

input $\langle P, \texttt{C} \rangle$
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** $\langle P, \texttt{C} \rangle$.

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

One can do similar for the machine M given as instance to the NE_{TM} problem. It is helpful to reflect that the "type" of this problem is "just a machine M". The type of the "USEFUL CLASS" problem is "Just a program P with the specified class C", 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:

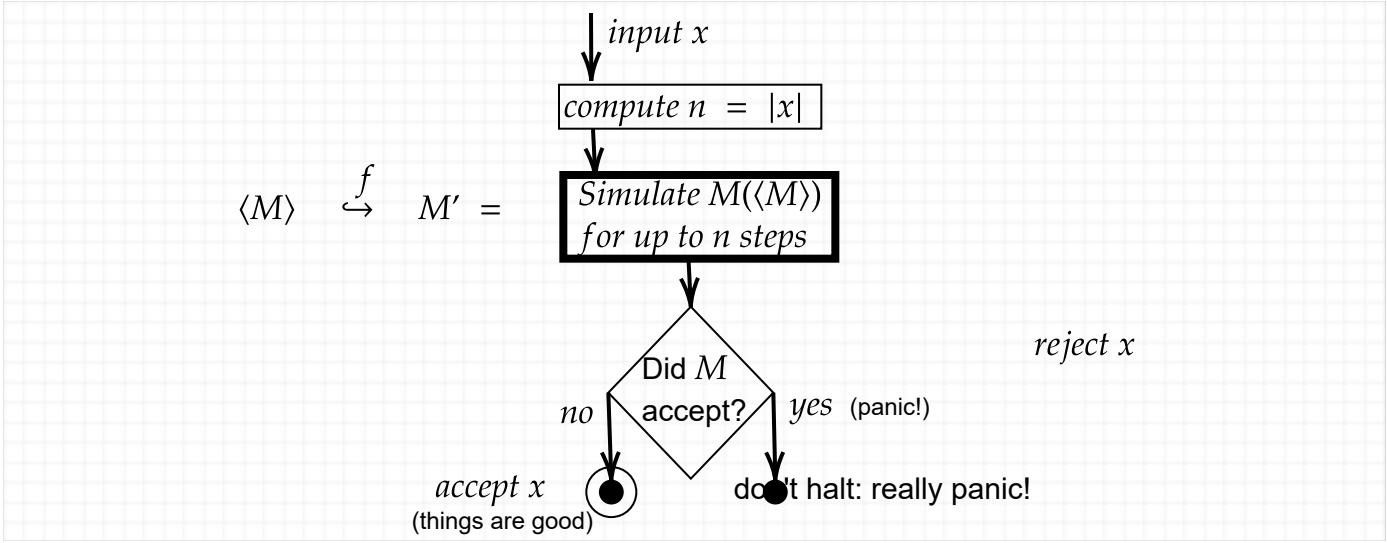
- there exists an x such that when P is run on x, ...
- ...there exists a step at which P creates an object of the class C.

A language defined *solely* by *existential* quantifiers in this way, down to "bedrock" predicates like creating a class object that are *decidable*, is generally ***c.e.***

Now we continue with a third "reductions toolkit" technique that is good for working with languages that are *not* c.e.

III The "Delay Switch"

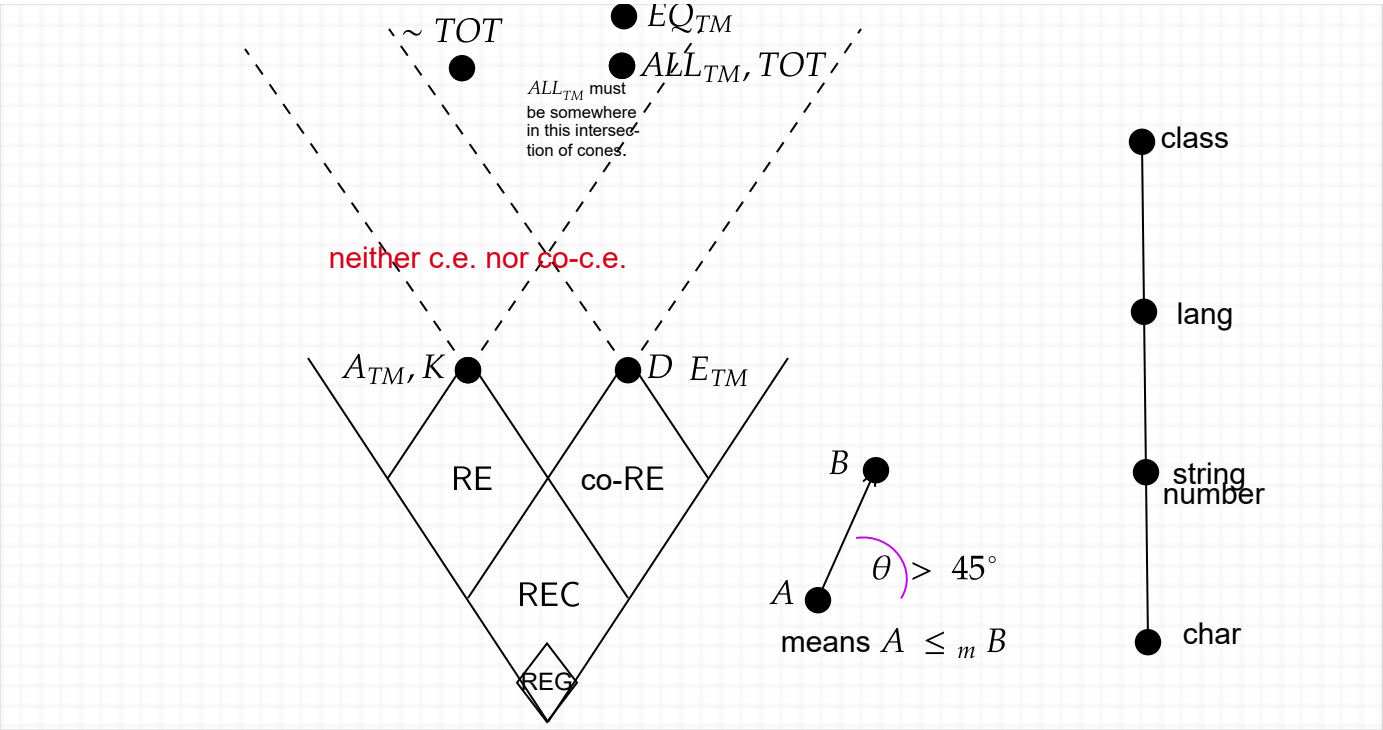
The third useful reduction technique is something I call the "Delay Switch." The intuition and attitude are the opposite of "Waiting for Godot" and the all-or-nothing switch. This time you picture your target machine M' or target program P as monitoring a condition that you hope *doesn't* happen, such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



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

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

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."

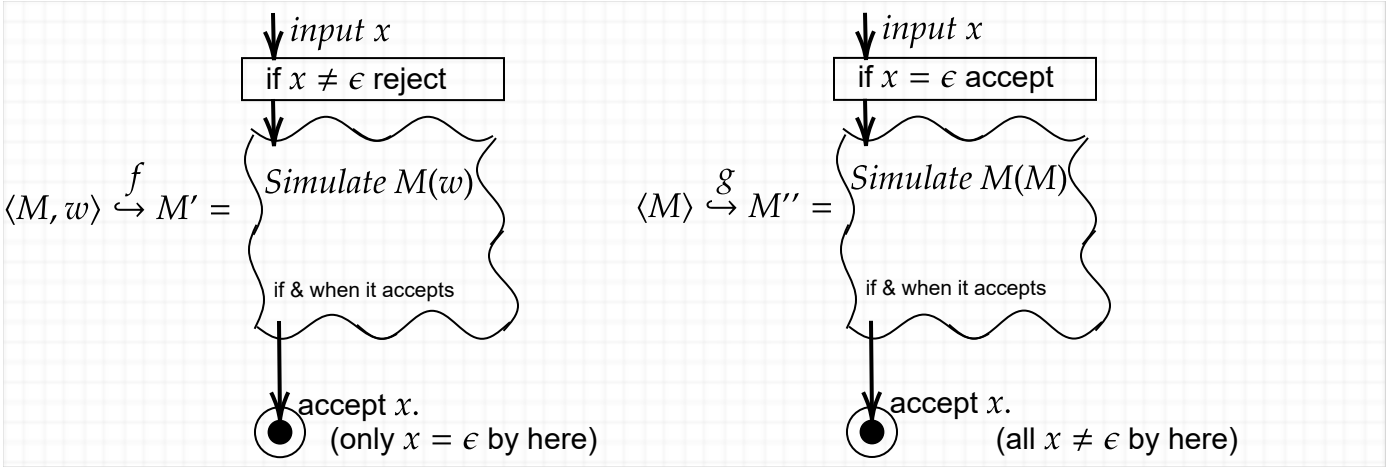
Example: $EQ_{TM} = \{\langle M_1, M_2\rangle : L(M_1) = L(M_2)\}$. Prove this is neither c.e. nor co-c.e.

Make a special case the target: the case where M_2, say, has $L(M_2) = \Sigma^*$. Call that M_{all}. Then $\langle M_1, M_{all}\rangle \in EQ_{TM} \iff \langle M_1\rangle \in ALL_{TM}$. 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.

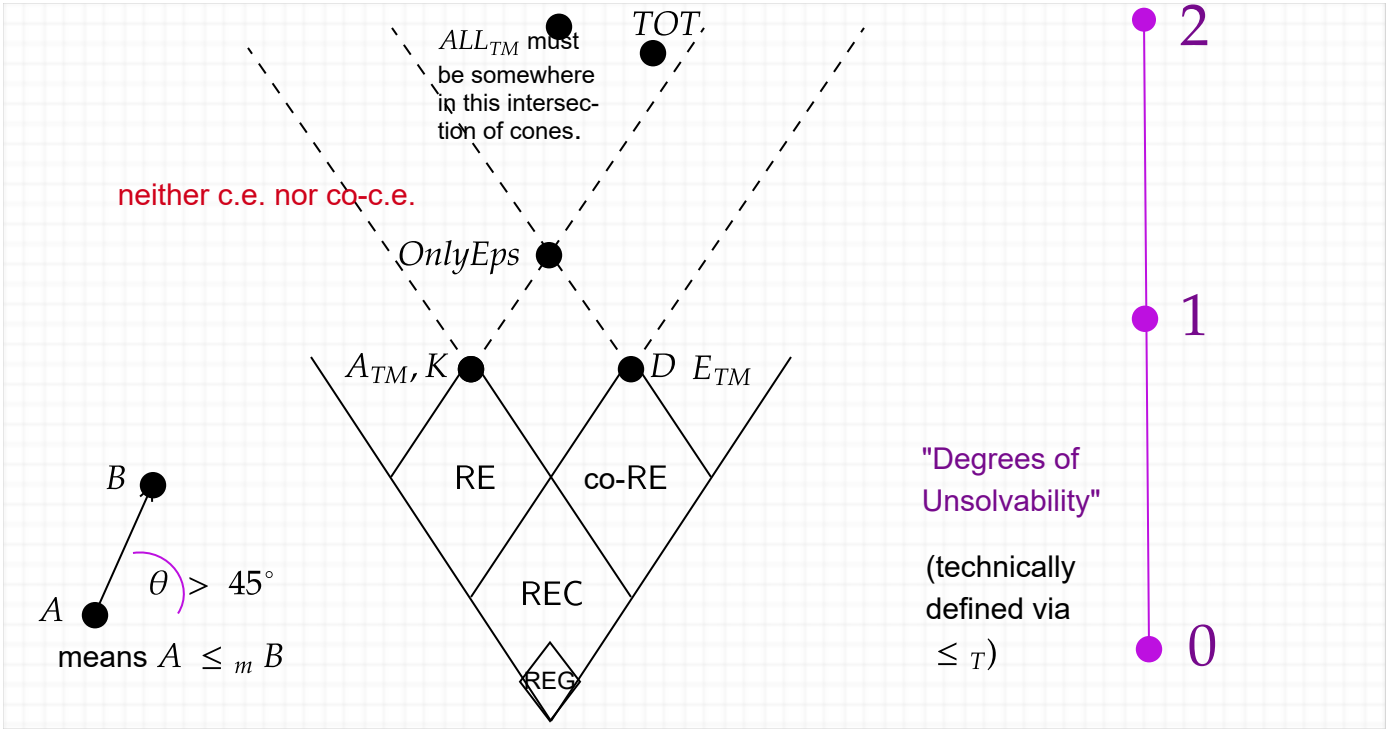
Here is a trickier problem with a trickier name:

$OnlyEps_{TM}$
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 $D_{TM} \leq_m OnlyEps_{TM}$.



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.



Some footnotes:

- 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 $\exists$ versus $\forall$ in logic) down to a single stick, as at right below. So I prefer to avoid them at this point.
- 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.