Reductions in Computability Theory

Prakash Panangaden

21st May 2020

The concept of reduction is central to computability and complexity theory. The phrase "P reduces to Q" is often used in a confusing way. In this note I will clarify how it is used and explain the difference between two different notions of reduction: many-one reduction and Turing reduction.

Intuitively "P reduces to Q" means that if you can solve problem Q then you can solve problem P. A reduction is an explicit proof of this implication. The notation is $P \leq Q$. The phrasing is, in my opinion confusing, whereas the notation conveys a clear message. If $P \leq Q$ then Q is *more difficult than* P, this is what the notation suggests. If Q is more difficult than P then if we show $P \leq Q$ and P is something that we know is undecidable then we know that Q must be undecidable. This is usually how we use reductions.

We start with $HP \leq Q$ where HP stands for the halting problem and thus show Q is undecidable. Thus we expand out stock of undecidable problems. When we know that Q is undecidable we can do another reduction $Q \leq R$ and now we know that the problem R is undecidable. Thus the relation $\leq$ is *transitive*: we can chain reductions together.

It is time for some precise definitions. We assume that we are working with some fixed alphabet Σ and we are considering strings in Σ^*. We assume that all the problems that we consider are *language membership* problems. Recall that a language is just a subset of Σ^*. This is not a restriction: all computational problems can be recast easily as language membership problems. Now we can talk about a language A being reduced to B: this means that if you can solve the problem "is the word w in B?" you can solve the problem "is the word x in A?"

Definition 1. A language A is said to be **mapping reducible to**[1] B, written $A \leq_m B$, if there is a *total computable* function $f : \Sigma^* \longrightarrow \Sigma^*$ such

[1]Many, in fact, most books call this "many-one" reducibility.

that $x \in A$ if and only if $f(x) \in B$. The function f is called the **reduction**.

Even though there is an "if and only if" in the definition it is **not** a symmetric concept. If we have such a reduction and we have an algorithm to decide membership in B we can clearly use it to decide membership in A. It is crucial that we have the "if and only if" for this to work correctly. Note, however, that if we had an algorithm to decide membership in A it cannot be used to decide membership in B. Why not? Because there is no reason to assume that f is a surjection. So given a word w for which we wish to answer the question "is w in B?" we would need to find an x such that $f(x) = w$: there is no reason to think that we can find such an x.

The following theorem is obvious from the discussion above.
Theorem 2. If $A \leq_m B$ and B is decidable then A is decidable. Conversely, if A is undecidable then B must be undecidable.

In fact mapping reduction is a very strong relation and can be used to establish more.
Theorem 3. If $A \leq_m B$ and B is CE (or Turing recognizable) then so is A. If A is not CE then neither is B.

Proof . Suppose that B is CE so there is an algorithm isinB which on an input x will halt and say "true" if x is in B and may loop forever if x is not in B. Now we define an algorithm isinA as follows:

```
1.   Read input x
2.   Apply f to x to produce y
3.   Run isinB on y
4.   Output whatever the line 3 produces
```

Line 2 must terminate because f is *total.* If x is in A and the reduction is correct then $f(x) = y \in B$. Since we have an algorithm isinB that is guaranteed to terminate and say "true" if its input is in B we know that isinA terminates and says "true" if $x \in A$. If x is not in A, then $f(x)$ will not be in B and line 3 may run forever. Thus A is CE. If A is not CE, then B cannot be CE by contraposing what we just proved. ∎

Recall that a set is called **co-CE** if its complement is CE. Thus the non-halting set

$$\overline{H_{TM}} \overset{\text{def}}{=} \{\langle M, x\rangle | M(x) \uparrow\}$$

is a typical example of a co-CE set. We can use mapping reductions to show that some sets are not CE by showing reductions from this set. First

an important observation. It says that mapping reduction respects complementation. It is obvious from the definition of $\leq_m$.
Observation 4. If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$.

Now we can state the following corollary.
Corollary 5. If $A \leq_m B$ and A is CE but not decidable then B is not co-CE.

Proof . If B were co-CE then $\overline{B}$ would be CE and the mapping reduction could be rewritten as $\overline{A} \leq_m \overline{B}$ so we would have that $\overline{A}$ is CE. But if A is CE and $\overline{A}$ is also CE then A is decidable. This contradicts the assumption that A is not decidable. ∎

Now for an example. We define the language of (encodings of) Turing machines that accept nothing:

$$EMP_{TM} \overset{\text{def}}{=} \{\langle M \rangle | L(M) = \emptyset\}.$$

Intuitively, this should be clearly undecidable. We will prove this by showing $A_{TM} \leq_m \overline{EMP_{TM}}$. We need to define a reduction function $f : \Sigma^* \longrightarrow \Sigma^*$ which is total and computable. We assume that Σ consists of all the characters that one has in a typical programming language. The problem we are trying to solve is: given $\langle M, w \rangle$, the encoding of a Turing machine M and a word w, accept if and only if M accepts w. Our reduction function has to work with any string, not just strings that are encodings of acceptance problems. We settle this by mapping any string that is not of the right form to the string "fubar." This string is certainly not part of EMP_{TM} so we are fine. Now, we define a new TM M':

```
Input x
Simulate M on w
If M accepts w accept x
```

Now clearly if M accepts w, M' will accept every input and if M does not accept w then M' will not accept anything. Thus

$$w \in L(M) \Leftrightarrow L(M') \neq \emptyset.$$

We do not ever run M'. We write *the code of* M' and feed $\langle M' \rangle$ to the supposed decision algorithm for $\overline{EMP_{TM}}$. Thus if we could decide $\overline{EMP_{TM}}$ we could decide A_{TM} which we know is impossible. Our function f reads its input, if it is in the right format it will output the code of M', if it is in the wrong format it will output "fubar"; this is a total computable string to string function. In subsequent discussions I will not mention what to do if the strings are not in the right format to constitute an instance of the decision problem of interest.

Of course, this also shows that EMP_{TM} is undecidable. However, it is impossible to find a mapping reduction from A_{TM} to EMP_{TM}. If we could do that we would have also shown that $\overline{A_{TM}} \leq_m \overline{EMP_{TM}}$. But we can easily see by dovetailing that $\overline{EMP_{TM}}$ is CE. Then we would conclude that $\overline{A_{TM}}$ is CE, hence A_{TM} is co-CE; but we known that A_{TM} is CE. These two facts together imply that A_{TM} is decidable, a contradiction.

An easy exercise at this point is to show $\overline{EMP_{TM}} \leq_m A_{TM}$; this also shows that $\overline{EMP_{TM}}$ is CE and hence that EMP_{TM} is co-CE. A slight modification of your solution to this exercise will show that $EMP_{TM} \leq_m \overline{A_{TM}}$.

Here is the solution to $EMP_{TM} \leq_m \overline{A_{TM}}$. We take the input $\langle M \rangle$ given as an instance of the problem $\langle M \rangle \in EMP_{TM}$. We construct the *code of* the following Turing machine M':

```
Input x (ignored)
Dovetail M on all words
If it accepts any word accept x
```

Now clearly M' does **not** accept the empty word if and only $L(M) = \emptyset$.

What we definitely do not have is $EMP_{TM} \leq_m A_{TM}$. I will now describe a more liberal notion of reduction called **Turing reduction**.
Definition 6. Let P be a language (or decision problem). We say that M is a Turing machine with **oracle** P, written M^P if M is an ordinary Turing machine which can at any time obtain, in one step, a definite answer to the question $w \in P$ for any word w.

An example is a Turing machine which can consult an "oracle" and find the answer to the halting problem for any instance of HP. Now suppose that I have a TM equipped with an oracle for A_{TM}. With such a TM I can solve the emptiness problem. Here is how: given $\langle M \rangle$, we define a new machine called N which works as follows:

```
Input x (ignored)
Dovetail M on all words
If M accepts any word accept x
```

We then ask our oracle if $\langle N, \varepsilon \rangle \in A_{TM}$. If the oracle says "no" accept $\langle M \rangle$ else reject. If $L(M) = \emptyset$ then N does not accept ε. Thus by flipping the oracle's answer we can answer the emptiness problem for the original machine. This shows that $EMP_{TM} \in TM^{A_{TM}}$.

Definition 7. We say that A Turing reduces to B, written $A \leq_T B$, if a Turing machine with an oracle for B can answer membership queries for A.

What we have shown above is $EMP_{TM} \leq_T A_{TM}$. This shows that Turing reduction is more liberal than mapping reduction. Every mapping reduction is a valid Turing reduction but not vice-versa. Turing reduction is not sensitive to complementation. Thus, for example, one can easily[2] show that $A_{TM} \leq_T \overline{A_{TM}}$ which is of course not possible with mapping reduction.

Here is the basic result about Turing reduction.

Theorem 8. If $A \leq_T B$ and if A is undecidable then B is also undecidable. If B is decidable then A is decidable.

Thus, if we want to show undecidability Turing reduction is enough. Mapping reductions are harder to construct but they give more information.

It is sometimes hard for beginners to see the real difference between mapping reduction and Turing reduction in specific instances. If we look closely at the Turing reductions above we asked the oracle a question and we did some post-processing on the oracle's answer. In mapping reduction we packaged a question to the oracle (this is what f does) but we do not get to do anything with the answer other than return it; we cannot even negate it. More generally, with Turing reduction we can ask the oracle many questions but with mapping reduction we get just one question and no post-processing of the answer.

Turing reduction is deep and interesting but we will not explore it in any depth. We will only use it to establish undecidability.

[2]Indeed this is trivial, make sure you understand this.