CS 4510 : Automata and Complexity : About Reductions

Subrahmanyam Kalyanasundaram

March 3, 2010

Reductions are extremely interesting and useful, as we have seen in class already. We only proved one undecidability result directly; that of A_{TM}. For all of the other undecidable problems, we used reductions – from A_{TM} or other languages that we have proved to be undecidable already – to prove the undecidability. Let us define the notion formally.

Definition 1 (Computable Function). *A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a* computable function *if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.*

Definition 2 (Reducibility). *Language A is* reducible *or* mapping reducible *to language B, denoted $A \leq_m B$ if and only if there is a computable function f, such that for every w, we have*

$$w \in A \Longleftrightarrow f(w) \in B$$

The function f is called the reduction *of A to B.*

What does a reduction really mean? What does it mean to reduce a problem A to problem B? Why the word "reduce"? The idea is that if A is reducible to B, we are converting an instance of a problem A to one of problem B. So if we know to how test membership in B, we can use that, with the reduction function, to test membership in A. How?

To check if $w \in A$,

1. We use the reduction machine to compute to compute $f(w)$
2. Using the algorithm for B, check if $f(w) \in B$
3. If $f(w) \in B$, we have $w \in A$
4. If $f(w) \notin B$, we have $w \notin A$

Let us get to the important points here :

- **The machine used for reduction should have reasonably bounded computation power.** In the Turing machine case, we need the reductions to be Turing computable. Note that it does not make sense to have a reduction which uses more power than the machines which solve B. Eventually we are looking for a Turing machine algorithm to solve A, so if the reduction uses a more powerful machine, then we don't have a Turing machine algorithm to solve A. Step 1 of the above algorithm would not be doable by a Turing machine. We will see more of this in the complexity theory part, when we talk about polynomial time reductions.

- **The reduction need not be both ways.** Just because $A \leq_m B$, we need not have $B \leq_m A$. Nor is it required to have $B \leq_m A$. There might be languages that are reducible to each other (we will see lots of examples in the complexity part), but in general reductions are only in one direction.

- **We need f to satisfy both arrows of $w \in A \iff f(w) \in B$.** This is easy to be confused with the above bullet. But both directions of the correspondence are needed. Why? When we check if $f(w) \in B$ we should be able to conclude if $w \in A$.

  - Suppose $f(w) \in B$. Then we are using the "to the left" direction of the arrow. That is, we are using that $w \in A \Longleftarrow f(w) \in B$ to conclude that $w \in A$.
  - Suppose $f(w) \notin B$. Then we are using the "to the right" direction of the arrow. That is, we are using that $w \in A \Longrightarrow f(w) \in B$. By contrapositive of this, we have $f(w) \notin B \Longrightarrow w \notin A$ to conclude that $w \notin A$.

  So both directions of the arrow are crucial to our reduction. Like the silly example we saw in class: If we didn't require the left direction of the arrow, all we are looking for is $w \in A \Longrightarrow f(w) \in B$. We could blindly set $f(w)$ to be an arbitrary string in B, which does not depend on w at all. When $w \in A$, we have the right direction fine, we have $f(w) \in B$. But the problem is that even when $w \notin A$, we have $f(w) \in B$. So when we get an answer that $f(w) \in B$, we cannot conclude anything about the membership of $w \in A$. The string w might be or might not be in A. Similarly for the other direction of the arrow.