Reductions

Mahesh Viswanathan

April 14, 2016

A *reduction* is a way of converting one problem into another problem such that a solution to the second problem can be used to solve the first problem. We say the first problem *reduces* to the second problem. Reducing problem A to a problem B shows that an algorithmic solution to problem B implies an algorithmic solution to problem A. Thus reductions provide a mechanism to compare the computational difficulty of two problems — if A reduces to B then A is (computationally) *no more difficult* than B, or (contrapositively) B is (computationally) *at least* as difficult as A. We will make precise these notions and give many examples of reductions.

But first, what are reductions? What does it mean to say that problem A reduces to B?

Definition 1. *A* reduction *(a.k.a. mapping reduction/many-one reduction) from a language $A \subseteq \Sigma^*$ to a language $B \subseteq \Sigma^*$ is a* computable function $f : \Sigma^* \to \Sigma^*$ *such that*

$$w \in A \textit{ if and only if } f(w) \in B$$

In this case, we say A is reducible *to B, and we denote it by $A \leq_m B$.*

A reduction defined by a function f needs to be *computable.* What that means is that there is a Turing machine M such that on any input w, M halts with $f(w)$ on its tape. The requirement that M halts is important. Given the Church-Turing thesis, we could also say that a reduction f is computable if we can write a Java function `computef` of type `String` $\to$ `String` such that `computef`(w) always halts and returns $f(w)$.

Intuitively, a reduction is a transformation f of inputs of A to inputs of B. We have two requirements on this transformation. First f needs to be computable. Second, solving problem A on input w should yield the same answer as solving problem B on the transformed input $f(w)$; this is captured by the fact that $w \in A$ iff $f(w) \in B$ [1].

Mapping/many-one reductions (Definition 1) are only one form of reductions. In this course, they are the only type of reductions we will consider, and so we will often drop "mapping/many-one" and just refer to them as reductions.

Example 1. Let us a look at the first formal example of a reduction. Recall the following two problems

$$\begin{aligned} \textsc{SelfReject} &= \{\langle M\rangle \mid M \text{ does not accept } \langle M\rangle\} \\ \textsc{Accept} &= \{\langle M, w\rangle \mid M \text{ accepts } w\} \end{aligned}$$

Recall that we have shown that $\textsc{SelfReject}$ is not recursively enumerable (and hence also undecidable), while $\textsc{Accept}$ is recursively enumerable because the universal Turing machine accepts/recognizes/solves $\textsc{Accept}$. Let us consider the complement of $\textsc{SelfReject}$, which formally is

$$\overline{\textsc{SelfReject}} = \{\langle M\rangle \mid M \text{ accepts } \langle M\rangle\}$$

We will show that $\overline{\textsc{SelfReject}} \leq_m \textsc{Accept}$. What this requires is for us to come up with a function that takes input for $\overline{\textsc{SelfReject}}$ (i.e., source code of TM/program) and produces an input for $\textsc{Accept}$ (i.e., a

[1] Recall that a language A defines a decision problem: Given input w, determine if $w \in A$.

pair of program + input). Let us define f as follows: $f(\langle M\rangle) = \langle M, \langle M\rangle\rangle$. In other words, given a program M, f returns a pair that is $\langle M, \langle M\rangle\rangle$.

To prove that f is a reduction, we need to show two things. First that f is computable, i.e., we need to come up with a program M_f that computes f. The program M_f simply "copies" the input string $\langle M\rangle$ (source code of a program/TM) twice to generate the string $\langle M, \langle M\rangle\rangle$. This program M_f will clearly halt no matter what M is because all it is doing is coping the source code.

The second thing we need to argue is that $\langle M\rangle \in \overline{\text{SelfReject}}$ iff $f(\langle M\rangle) \in \text{Accept}$. Observe that $\langle M\rangle \in \overline{\text{SelfReject}}$ iff M accepts $\langle M\rangle$ (by definition of the language $\overline{\text{SelfReject}}$) iff $\langle M, \langle M\rangle\rangle \in \text{Accept}$ (by definition of the language Accept) iff $f(\langle M\rangle) \in \text{Accept}$, since $f(\langle M\rangle) = \langle M, \langle M\rangle\rangle$

Many of the examples in these notes involve languages/problems where the input is the source code of a program/TM. In addition, there is also the separate program that computes the reduction f itself. When carrying out reductions it is important to not get confused between all of these. For example, in Example , there is the program M which is input to $\overline{\text{SelfReject}}$, and there is program (which also happens to M) that the reduction produces as input to Accept. There is a third program, namely M_f, that computes the function f, by simply copying its input string twice to produce an output. It is best to clearly separate all these different programs both while thinking and when writing solutions, by giving them names (like M, M_f) instead of refering to them as "the program" or "that program" or "it".

1 Properties of Reductions

The primary power of reductions comes from the fact that reductions allow us compare the computational difficulty of problems. This is also captured in the way we denote reductions as $\leq_m$. When a problem A is reduced to B, we write it as $A \leq_m B$ to suggest that "A is no more difficult than B". This is formally proved in the next result.

Theorem 1. *Suppose $A \leq_m B$. Then the following are true.*

1. *If B is recursively enumerable then A is recursively enumerable.*
2. *If B is decidable then A is decidable.*

Proof. Suppose f is a reduction from A to B and f is computed by Turing machine M_f. Suppose M_B is a TM such that $\mathbf{L}(M_B) = B$ (i.e., M_B accepts/recognizes/solves B). So we have M_B accepts u iff $u \in B$. Consider the following program M_A

$M_A(w)$
    $u = M_f(w)$
    `return` $M_B(u)$

Informally, on input w, M_A calls M_f to compute $u = f(w)$, then calls M_B on u, and returns "accept" if M_B accepts u and rejects otherwise. Since M_f computes f, it means that M_f halts on all inputs (from the definition of what it means for a function to be computable) and the step assigning u terminates. Thus, M_A will halt w if and only if M_B halts on u. Moreover, since f is a reduction, we have $w \in A$ iff $f(w) \in B$. This gives us the following line of reasoning: M_A accepts w iff M_B accepts $u = f(w)$ iff $u = f(w) \in B$ iff $w \in A$. Thus, M_A solves the right problem, i.e., $\mathbf{L}(M_A) = A$, and A is recursively enumerable. Further, since M_A halts whenever M_B halts, if we know that M_B decides B (i.e., halts on all inputs and $\mathbf{L}(M_B) = B$) then M_A decides A, thus completing the proof of both statements.

It is important to note how critical our assumption that M_f (the program computing f) halts on *all* inputs is. Without that assumption, the step computing u may not halt, and then M_A may not halt on inputs $w \in A$, simply because the step computing u failed to halt. $\square$

The contrapositive of Theorem 1, which is often the way reductions are used, says that if $A \leq_m B$ then B is at least as difficult as A. Thus, if A is a computationally hard problem, then so is B. This is captured by the following corollary.

Corollary 2. *Suppose $A \leq_m B$. Then the following are true.*

1. *If A is not recursively enumerable then B is not recursively enumerable.*
2. *If A is undecidable then B is undecidable.*

Proof. These statements are just contrapositives of Theorem 1. $\square$

Given the reduction in Example , it implies that Accept is undecidable.

Proposition 3. Accept *is undecidable.*

Proof. Observe that since SelfReject is not recursively enumerable, it is also not decidable. Moreover since decidable languages are closed under complementation (Discussion Lab 21), $\overline{\text{SelfReject}}$ is also undecidable. From Example , we have $\overline{\text{SelfReject}} \leq_m$ Accept. Finally, using Corollary 2, we have Accept is undecidable because $\overline{\text{SelfReject}}$ is undecidable. $\square$

Proposition 3 is often the way reductions are used — we prove a problem to be "difficult" (undecidable or not recursively enumerable) by showing that it is at least as difficult as some other problem that is known to be difficult. Here we conclude that Accept is difficult (undecidable) because it is at least as difficult as $\overline{\text{SelfReject}}$ that is known to be undecidable.

We conclude this section with a couple of other important properties about reductions.

Theorem 4. *If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$.*

Proof. Let f be a reduction from A to B computed by M_f. We claim that f is also a reduction from $\overline{A}$ to $\overline{B}$. Clearly, we know that f is computed by M_f. And we have,

$$w \in \overline{A} \text{ iff } w \notin A \text{ iff } f(w) \notin B \text{ (since } f \text{ is a reduction from } A \text{ to } B) \text{ iff } f(w) \in \overline{B}$$

$\square$

Theorem 5. *If $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.*

Proof. Suppose f is a reduction from A to B, computed by program/TM M_f, and g is a reduction from B to C, computed by program/TM M_g. To prove $A \leq_m C$ we need to define a reduction h from A to C. Take $h = g \circ f$, i.e., for every w, $h(w) = g(f(w))$. To prove that h is reduction, we need to show that h is computable, and that h satisfies the properties of a reduction. We do this in order. Observe that

$M_h(w)$
  $u = M_f(w)$
  $v = M_g(u)$
  `return` v

always halts (because M_f and M_g always halt) and computes the function $h = g \circ f$. Next, we have $w \in A$ iff $f(w) \in B$ (since f is a reduction from A to B) iff $g(f(w)) \in C$ (since g is a reduction from B to C) iff $h(w) \in C$ (since $h(w) = g(f(w))$). Thus, h is a reduction from A to C. $\square$

2 Examples

We now give more examples of reductions and their use in proving problems to be difficult.

Proposition 6. *The language* Halt $= \{\langle M, w\rangle \mid M$ *halts on* $w\}$ *is undecidable.*

Proof. We will show that Accept $\leq_m$ Halt. To do this we need to come up with a function f that takes inputs for problem Accept (i.e., pairs $\langle M, w\rangle$ of source code + input) and produces inputs for problem Halt (i.e., pairs $\langle M', w'\rangle$). As a first step, let us describe a program $g(M)$, where M is a TM.

```
g(M)(x)
    result = M(x)
    if (result = accept)
        return accept
    else if (result = reject)
        while true do
```

The program $g(M)$ does the following: On input string x, it calls M. Now M may or may not halt on x. If M halts and returns `accept` then $g(M)$ also stops and returns `accept`. On the other hand, if M halts and returns `reject` then $g(M)$ enters an infinite loop and does not halt.

We are now ready to define the reduction from Accept to Halt: $f(\langle M, w\rangle) = \langle g(M), w\rangle$. To complete the proof, we need to argue that f is computable and f satisfies properties of a reduction.

Computing f: The program M_f computing f does the following. Given the source code M and input w, M_f will generate the source code $g(M)$ given above, and copy the string w to the output. Notice, to generate the source code for $g(M)$ only involves in producing the text above. It does not involving running the program M; when $g(M)$ is executed it will call M. Thus, the program M_f is a very simple program that involves producing a fixed text (namely the source code of $g(M)$ above) and coping the string w. Hence, it always halts, no matter what string $\langle M, w\rangle$ it is called on.

f is a reduction: Suppose $\langle M, w\rangle \in$ Accept. Then by definition, M accepts w (and halts). When $g(M)$ is run with input w, $g(M)$ will call M on w which will halt and return `accept`, and $g(M)$ will then go the then-branch of the conditional and accept (and halt). Thus, $f(\langle M, w\rangle) = \langle g(M), w\rangle \in$ Halt. On the other hand, suppose $\langle M, w\rangle \notin$ Accept. Then M does not accept w. This could be because either M does not halt on w or M halts on w but returns `reject`. Now notice that when $g(M)$ is run with input w, if M does not halt on w, neither does $g(M)$. And if M halts on w and rejects then $g(M)$ will go to the else-branch and enter an infinite loop. Thus in either case, we have $g(M)$ does not halt on w. Putting this together we have, $\langle M, w\rangle \notin$ Accept then $f(\langle M, w\rangle) = \langle g(M), w\rangle \notin$ Halt. So

$$\langle M, w\rangle \in \textsc{Accept} \text{ iff } f(\langle M, w\rangle) \in \textsc{Halt}$$

Finally, since Accept $\leq_m$ Halt and Accept is undecidable (Proposition 3), we can conclude, using Corollary 2, that Halt is undecidable.

Before we conclude, it is useful to observe that if we had taken the reduction to be $h(\langle M, w\rangle) = \langle M, w\rangle$ then it does not work. h is clearly computable, but we don't have $\langle M, w\rangle \notin$ Accept implies $\langle M, w\rangle \notin$ Halt. The reason is if $\langle M, w\rangle \notin$ Accept, then it is possible that M halts and rejects, in which case $h(\langle M, w\rangle) = \langle M, w\rangle \in$ Halt. □

It is sometimes confusing to know which direction to reduce problems. It is useful to remember the following mnemonic. We always denote reductions by $\leq_m$. The first problem (the one being "reduced") is written to the left of $\leq_m$ and the second problem (the one to which we are reducing) is written to the right of $\leq_m$, and we always transform inputs from left-to-right (first problem input changed to second problem inputs). This mnemonic also helps determine what needs to be done in a certain situation. Suppose we want to prove a problem to be "difficult" then we need to *lower bound* it, i.e., write it to the right of the reduction symbol. So we need to find a well known difficult problem A and show $A \leq_m L$. On the other hand, we want to show that L is "easy" then we need to *upper bound* it, i.e., write it to left of the reduction symbol. So we need to find a well known easy problem B and show $L \leq_m B$. In Proposition 6, we we wanted to show that Halt is difficult (undecidable) we lower bounded it by showing Accept $\leq_m$ Halt.

Proposition 7. *The language* $E_{\mathrm{TM}} = \{\langle M\rangle \mid \mathbf{L}(M) = \emptyset\}$ *is not recursively enumerable.*

Proof. We want to prove that E_{TM} is diffcult (not r.e.) and so we want to lower bound it by a language that is not r.e. We know one example of such language SelfReject, so we will show SelfReject $\leq_m E_{\mathrm{TM}}$. This requires us to transform an input to SelfReject (source code of program/TM $\langle M\rangle$) to an input for E_{TM} (another source code $\langle N\rangle$). For a program M, let us define $f(\langle M\rangle)$ to be the following program

```
f(⟨M⟩)(x)
    result = M(⟨M⟩)
    if result = accept then
        return accept
    else if result = reject then
        return reject
```

Informally, $f(\langle M\rangle)$ does the following: It ignores its input x, runs M on $\langle M\rangle$. If M halts and accepts then $f(M)$ accepts x, and if M halts and rejects then $f(M)$ rejects x.

Observe that the function f is computable. The program M_f computing f will simply output the above program, when given $\langle M\rangle$ as input. Again, it is useful to remember that M_f does not execute the code $f(M)$; it simply produces it, and so M_f always halts.

Next, we need to argue that $\langle M\rangle \in \textsc{SelfReject}$ iff $f(\langle M\rangle) \in E_{\text{TM}}$. Suppose $\langle M\rangle \in \textsc{SelfReject}$ then (by definition of SelfReject) that means that M does not accept $\langle M\rangle$. There are two possible reasons for this. If M does not halt on $\langle M\rangle$ then $f(\langle M\rangle)$ also does not halt on any input x and so $\mathbf{L}(f(\langle M\rangle)) = \emptyset$. If M halts and rejects $\langle M\rangle$ then $f(\langle M\rangle)$ will enter the else branch and reject input x (no matter what x is). Thus, $\mathbf{L}(f(\langle M\rangle)) = \emptyset$ again, and so $f(\langle M\rangle) \in E_{\text{TM}}$. On the other hand, if $\langle M\rangle \notin \textsc{SelfReject}$ then it means that M (halts and) accepts $\langle M\rangle$. In this case, $f(\langle M\rangle)$ will go to the then-branch and accept x. In this case $\mathbf{L}(f(\langle M\rangle)) = \Sigma^* \neq \emptyset$. Putting all these observations together we have $\langle M\rangle \in \textsc{SelfReject}$ iff $f(\langle M\rangle) \in E_{\text{TM}}$.

Finally, since SelfReject is not recursively enumerable, and $\textsc{SelfReject} \leq_m E_{\text{TM}}$, from Corollary 2, we can conclude that E_{TM} is not recursively enumerable. □

Proposition 8. *The language* $\textsc{Regular} = \{\langle M\rangle \mid \mathbf{L}(M)$ *is regular*$\}$ *is undecidable.*

Proof. Our proof will rely on showing that $\textsc{Accept} \leq_m \textsc{Regular}$. We need to transform inputs to Accept (pairs of program+input) into inputs to Regular (program). For a pair $\langle M, w\rangle$ define $f(\langle M, w\rangle)$ to be the program

```
f(⟨M,w⟩)(x)
    if x is of the form 0^n 1^n for some n
        return accept
    else
        result = M(w)
        if result = accept
            return accept
        else
            return reject
```

The program $f(\langle M, w\rangle)$ does the following when executed. If x is a string of 0's followed by 1s where the number of 0s is equal to the number of 1s (i.e., x is of form 0^n1^n) then x is accepted. Otherwise, we run M on w, and accept x only if M accepts w.

It is straightforward to see that there is a program M_f that halts on all inputs and produces the source code for $f(\langle M, w\rangle)$ on input $\langle M, w\rangle$. Next, observe that if M does not accept w, then the only strings which $f(\langle M, w\rangle)$ accepts are those of the form 0^n1^n, and so $\mathbf{L}(f(\langle M, w\rangle)) = \{0^n1^n \mid n \geq 0\}$. In this case, $\mathbf{L}(f(\langle M, w\rangle))$ is non-regular and so $f(\langle M, w\rangle) \notin \textsc{Regular}$. On the other hand, if M accepts w then $f(\langle M, w\rangle)$ accepts every string; so $\mathbf{L}(f(\langle M, w\rangle)) = \Sigma^*$ which is regular. Putting it together we have

$$\langle M, w\rangle \in \textsc{Accept} \text{ iff } f(\langle M, w\rangle) \in \textsc{Regular}$$

Finally, since $\textsc{Accept} \leq_m \textsc{Regular}$ and Accept is undecidable, Regular is undecidable.

Is Regular recursively enumerable? It turns out we can prove an even stronger result that shows that Regular is not recursively enumerable. We can do this by showing $\textsc{SelfReject} \leq_m \textsc{Regular}$. For a program M, define the program $g(M)$ to be

$g(M)(x)$

```
result = M(⟨M⟩)
if result = accept
    if x is of the form 0^n 1^n
        return accept
return reject
```

$g(M)$ runs M on $\langle M \rangle$. If M accepts then $g(M)$ accepts x if it is of the form $0^n 1^n$. Otherwise, $g(M)$ does not accept; either it does not halt if M does not halt, or it rejects.

Clearly the function g is computable. Moreover, we have if $\langle M \rangle \in \textsc{SelfReject}$ then M does not accept $\langle M \rangle$, which means that $g(M)$ does not accept any string. So $\mathbf{L}(g(M)) = \emptyset$ is regular. On the other hand, if $\langle M \rangle \notin \textsc{SelfReject}$ then M accept $\langle M \rangle$ and so $g(M)$ accepts exactly the strings of the form $0^n 1^n$. In this case we will have $\mathbf{L}(g(M)) = \{0^n 1^n \mid n \geq 0\} \notin \textsc{Regular}$. Thus, $\langle M \rangle \in \textsc{SelfReject}$ iff $g(M) \in \textsc{Regular}$. Finally, since $\textsc{SelfReject}$ is not recursively enumerable, we have $\textsc{Regular}$ is not recursively enumerable. □

Proposition 9. *$EQ_{\mathrm{TM}} = \{\langle M_1, M_2 \rangle \mid \mathbf{L}(M_1) = \mathbf{L}(M_2)\}$ is not r.e.*

Proof. Recall, from Proposition 7, that E_{TM} is not recursively enumerable. We will prove this proposition by showing $E_{\mathrm{TM}} \leq_m \mathrm{EQ}_{\mathrm{TM}}$. Consider the following program

$M_\emptyset(x)$

```
return reject
```

No matter what the input is, $M_\emptyset$ rejects it. Thus, $\mathbf{L}(M_\emptyset) = \emptyset$. Define the reduction from E_{TM} to $\mathrm{EQ}_{\mathrm{TM}}$ as follows: $f(\langle M \rangle) = \langle M, M_\emptyset \rangle$. Again it is easy to see that f is computable: the program M_f computing f, just copies its input $\langle M \rangle$ onto its output and also writes down the code for $M_\emptyset$. Further, $\langle M \rangle \in E_{\mathrm{TM}}$ iff $\mathbf{L}(M) = \emptyset = \mathbf{L}(M_\emptyset)$ iff $\langle M, M_\emptyset \rangle \in \mathrm{EQ}_{\mathrm{TM}}$. □