Reductions

Vassos Hadzilacos

The notion of reducing a problem P to a problem Q is a natural one that we employ often: To find a solution to P we use a known (or assumed) solution to Q. For example, knowing that there is a road from Toronto to Kapuskasing, I can reduce the problem P of driving from Toronto to Ogoki to the problem Q of driving from Kapuskasing to Ogoki: If a solution to Q exists (i.e., I can drive from Kapuskasing to Ogoki), then a solution to P exists as well: I can drive from Toronto to Kapuskasing, which I know can be done, and use the solution to Q to drive from there to Ogoki. If, on the other hand, I somehow determine that no solution to P exists (i.e., there is no way to drive from Toronto to Ogoki), then I can also deduce that there is no solution to Q (i.e., there is no way to drive from Kapuskasing to Ogoki).

Notice the two ways of using a reduction: A "positive" (using a solution to Q to solve P), and a "negative" (using the fact that no solution exists to P to conclude that no solution exists to Q). Also notice the direction of the reduction, and what drives the deduction in each case: Solution to Q implies solution to P; no-solution to P implies no-solution to Q. To indicate that problem P reduces to problem Q we use some variant of the notation $P \leq Q$ (with the $\leq$ sometimes decorated with subscripts or superscripts). This notation is intended to suggest that, in some sense, P is "no harder" than Q: knowing how to deal with Q, ensures that we can also deal with P. This notation also suggests the possibility that P may be much easier than Q: One way of solving P is via Q; but perhaps there are other, easier ways to solve P that don't involve Q at all.

The notion of reduction plays an important role in computer science, in both of the uses mentioned above. Those of you who have taken the algorithms course (CSCC73) will recognize the positive use of reductions. There are certain central problems with well-known efficient solutions (shortest paths, maximum flow, linear programming) to which we can reduce many other problems, and thereby obtain efficient solutions for them. In this course we will make extensive use of the *negative* use of reductions to show that certain problems are, in some sense, hard because other problems, already known (or suspected) to be hard, reduce to them.

Recall that a language L (i.e., a set of finite strings) can be viewed as a decision problem or, equivalently, as a binary-valued function: namely, the problem of determining whether a given string x is in L (a yes-instance of the decision problem) or not (a no-instance); or, equivalently, the function that outputs 1 if $x \in L$ and 0 if $x \notin L$. Thus we can talk about reductions between *languages* (or sets, since a language is just a set).

Mapping reductions

The conceptually simplest type of reduction, and the one that we will mostly use in this course, is the ***mapping reduction*** of one language to another.

Definition 1 *Language $L \in \Sigma^*$ **mapping reduces** to language $L' \in \Sigma'^*$, denoted $L \leq_m L'$, if there is a* computable *function $f : \Sigma^* \to \Sigma'^*$ such that $x \in L$ if and only if $f(x) \in L'$.*

The situation is illustrated in Figure 1 below.

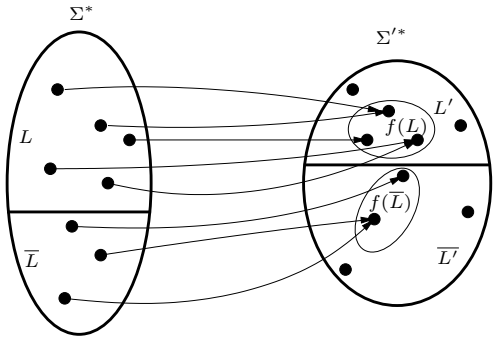

Figure 1: A mapping reduction

The set Σ^* is partitioned into L and $\overline{L}$, and Σ'^* is partitioned into L' and $\overline{L'}$. The function f maps L into a subset of L', and $\overline{L}$ into a subset of $\overline{L'}$. Note that f is not required to be one-to-one: it is possible (and indeed typically the case) that multiple strings in L map to the same string in L'. For this reason, mapping reductions are also known as ***many-one reductions***. Furthermore, f is not required to be onto: it is possible (and indeed typically the case) that some elements of L' are not mapped onto by any element of L and some elements of $\overline{L'}$ are not mapped onto by any element of $\overline{L}$. On the other hand, as we will see shortly, it is a crucial requirement of the definition that f be a *computable* function.

Intuitively, the function f translates the question "is x in L?" to the question "is $f(x)$ in L'?" so that the answer to both questions is the same. Therefore, if we have an algorithm A' to answer the question "is x' in L'?", we automatically get an algorithm A to answer the question "is x in L?": Algorithm A simply takes its input string x, computes $x' = f(x)$ (this can be done precisely because f is computable), and then uses the algorithm A' to determine whether $x' \in L'$, in which case it returns 1 (meaning that $x \in L$), or $x' \notin L'$, in which case it returns 0 (meaning that $x \notin L$).

In lecture we proved the following simple but important facts about mapping reductions.

Theorem 4.3 *If $L \leq_m L'$ and L' is decidable then L is decidable; equivalently, if $L \leq_m L'$ and L is undecidable then L' is undecidable.*

Theorem 4.4 *If $L \leq_m L'$ and L' is recognizable then L is recognizable; equivalently, if $L \leq_m L'$ and L is unrecognizable then L' is unrecognizable.*

Theorem 4.5 *If $L \leq_m L'$ then $\overline{L} \leq_m \overline{L'}$.*

Theorem 4.6 *The mapping-reduces relation is transitive: If $L \leq_m L'$ and $L' \leq_m L''$ then $L \leq_m L''$.*

Turing reductions

In a mapping reduction f of language L to language L' we obtain the L-decider that uses an L'-decider in a particularly simple manner (see Figure 2 below): All we do, is transform the input x into the input $f(x)$, we use the L'-decider to determine if $f(x) \in L'$, and we use the answer to that question directly as our answer for whether $x \in L$. That is, we use the L'-decider *exactly once* (on $f(x)$) and adopt the answer of L' for $f(x)$ *unaltered* as L's answer for x. Note that we are not even allowed to complement the answer of L', i.e., to output "no" if the decider for L' outputs "yes" and vice versa.

One can imagine more sophisticated ways of using an assumed L'-decider to help us construct an L-decider: To determine if $x \in L$ we might want to ask not only whether one string $f(x)$ is in L', but many of them. Which ones may depend on the answers we get to previous questions; for example, if $f(x)$ is not in L' we might want to then know if $g(x)$ is in L' for a different computable function g. And our answer as to whether $x \in L$ may depend on the answers we got to all the questions we asked of the L'-decider.

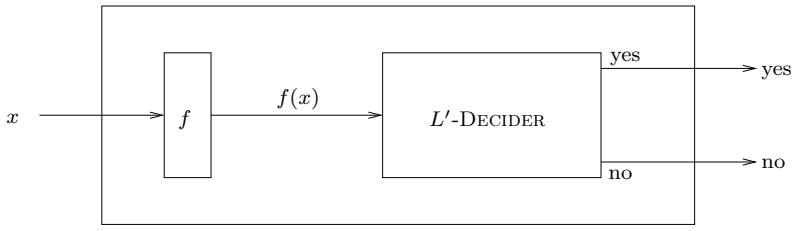

Figure 2: An L-decider that uses an L'-decider, via mapping reduction f

This more general type of reduction is called a Turing reduction. Intuitively, we say that L ***Turing-reduces*** to L', denoted $L \leq_T L'$, if there is an algorithm A that decides L given a subroutine that decides L', and A uses that subroutine as a "black box" — i.e., A is not allowed to "look inside" the given L'-decider to see how it works. A may use the L'-decider several times, on several different inputs, and use all the information it has garnered from these multiple uses of the L'-decider to determine whether its input string x is in L.

It is important to note here that a Turing reduction is an algorithm that *decides* (not merely *recognizes*) a language: It always halts on all inputs, provided the given L'-decider returns correct values, according to its specification. Also note that with this view of reduction, *the L'-decider can be assumed to solve arbitrary problems, including problems that are undecidable or even unrecognizable*: it is given to us as a black box; we do not ask how it works.

The idea is certainly familiar to us as programmers, because we often use subroutines or functions or libraries that someone else has written and which we can access only as black boxes: We give them some input and we get an output that we know (or hope) satisfies certain properties; we have no idea *how* the black box computes its result, only *what* it computes. Thus we can only argue the correctness of our programs relative to the correctness of the black boxes they use; that is, we argue that our programs are correct *provided* the black boxes they use in fact work as advertised: they terminate and produce an output according to their specification; if the black boxes misbehave and as a result our program bombs, either by looping forever or by producing incorrect outputs, it is not our fault.

What follows is an account of how to *formalize* Turing reductions. To do so, we need the notion of an "oracle Turing machine". A ***Turing machine with an oracle for language*** L is a 2-tape Turing machine that has two additional states: a ***query*** state $q_?$ and a ***response*** state $q_!$. Tape 1 is referred to as the "work tape" and is like a regular Turing machine tape, initially containing the oracle Turing machine's input; tape 2 is referred to as the "oracle tape" and initially all its cells contain blanks. The machine operates according to the usual rules for a 2-tape Turing machine, except that when it enters the query state $q_?$, the following actions are performed, all in one step:

(a) the string x that is written on the oracle tape, from the leftmost cell up to (but not including) the first blank symbol, is erased (replaced by blanks),
(b) the symbol 1 (respectively, 0) is written on the leftmost cell if $x \in L$ (respectively, $x \notin L$);
(c) the oracle tape head is moved to the leftmost cell; and
(d) the machine enters the response state $q_!$.

Thus, entering the query state is like making a call to a subroutine L'-Decider(x) that returns 1 if $x \in L$ and 0 if $x \notin L$, and entering the response state is like returning from this call. We denote a TM with an oracle for language L as M^L. As noted earlier, the oracle L is not required to be a decidable language.

Similarly, we can define a ***Turing machine with an oracle for function*** f, denoted M^f. The only difference from the preceding definition is that in action (b) of a step, what is written on the oracle tape is $f(x)$, i.e., the string output by the function f applied to the string written on the oracle tape when the oracle Turing machine is in state $q_?$. The oracle function f is not required to be computable. Note that

a Turing machine with a language oracle is just a Turing machine with a binary-valued function oracle. This is consistent with our view of languages as decision problems, i.e., binary-valued functions.

We can now define the notion of Turing reduction precisely.

Definition 2 *A language L **Turing-reduces** to language L', written $L \leq_T L'$, if there is an oracle Turing machine $M^{L'}$ that* decides *L. More generally, a function f reduces to function f' if there is an oracle Turing machine $M^{f'}$ that computes f.*

Note that in Definition 2 we require $M^{L'}$ to be a decider: it must halt on every input x and determine whether $x \in L$ or $x \notin L$. Similarly, $M^{f'}$ must halt on every input with the correct output $f(x)$, so f is a total (not partial) function.

Exercise: Consider Theorems 4.3-4.6 and determine whether each of them holds for Turing reductions, instead of mapping reductions. If so, prove the corresponding theorem; otherwise, provide a counterexample, i.e., a pair of languages L and L' for which the statement is not true. In your proofs it suffices to rely on the informal definition of Turing reduction $L \leq_T L'$, in terms of an algorithm for L that uses a subroutine for L' as a black box.