

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 speak of reductions between *languages* (or sets, since a language is just a set).

Turing reductions

Intuitively, we say that language L **Turing-reduces** to language L' , denoted $L \leq_T L'$, if there is an algorithm A that decides L (i.e., can determine membership of its input in L) given *as a black box* a subroutine that determines membership of its input in L' . The subroutine for L' is a black box in the sense that A is not allowed to “look inside” this subroutine to see how it works or to modify it; it can only ask it whether certain strings are in L' and count on correct answers to these queries. To emphasize this point we will refer to this subroutine as an L' -**oracle**. Algorithm A may use the L' -oracle several times, on several different inputs, and combine all the information it has garnered from these multiple uses of the L' -oracle in any computable way to determine its output — i.e., whether its input x belongs to L .

It is important to note here that a Turing reduction of L to L' is an algorithm that *decides* (not merely *recognizes*) L : It always halts on all inputs and returns a correct answer (it accepts if its input x is in L and rejects if x is not in L) provided the given L' -oracle returns correct answers to the queries that the reduction algorithm submitted to it. Also note that with this view of reduction, *the L' -oracle 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. In fact, as we have seen, one use of reductions is to argue that L' is undecidable.

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.

In the rest of this section we show how to formalize Turing reductions, and in particular the notion of using a subroutine as a black box. To do so, we introduce the concept 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 z 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 of the oracle tape if $z \in L'$ (respectively, $z \notin L'$);
- (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' -BLACKBOX(z) that returns 1 if $z \in L'$ and 0 if $z \notin L'$, 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 h*** , denoted M^h . The only difference from the preceding definition is that in action (b) of a step, what is written on the oracle tape is $h(z)$, i.e., the string output by the function h applied to the string written on the oracle tape when the oracle Turing machine is in state $q_?$. The oracle function h 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 g ***Turing-reduces*** to function h if there is an oracle Turing machine M^h that computes g .

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^h must halt on every input x with the correct output $g(x)$, so g is a total (not partial) function.

Mapping reductions

A *mapping reduction* is a special case of Turing reduction. It is conceptually simpler, and it is the kind of reduction that we will use in this course for the most part.

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

The situation is illustrated in Figure 1 below.

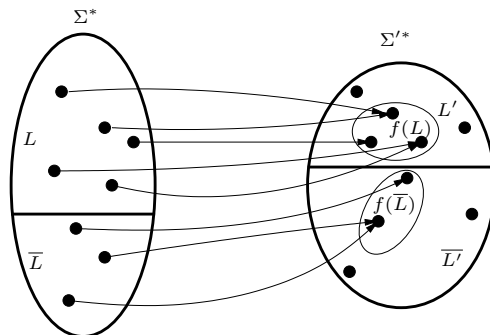


Figure 1: A mapping reduction

The set Σ^* is partitioned into L and \bar{L} , and Σ'^* is partitioned into L' and \bar{L}' . The function f maps L into a subset of L' , and \bar{L} into a subset of \bar{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 \bar{L}' are not mapped onto by any element of \bar{L} . On the other hand, as we will see shortly, it is a crucial requirement of the definition that f be a *computable* function.

Here is why a mapping reduction is a special case of the Turing reduction we defined in the previous section. Given a mapping reduction f of language L to language L' , we obtain an L -decider that uses an L' -oracle in a particularly simple manner (see Figure 2 below): The L -decider first transforms the input x into the input $f(x)$ (it can do this precisely because f is computable); it then uses the L' -oracle to determine if $f(x) \in L'$; and it uses the answer to that question directly as its answer for whether $x \in L$. 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. Notice that here we use the L' -oracle *exactly once* (on $f(x)$) and adopt the answer of L' for $f(x)$ *unaltered* as L 's answer for x . We are not even allowed to complement the answer of L' , i.e., to output “no” if the L' -oracle outputs “yes” and vice versa.

In contrast, a Turing reduction of L to L' is allowed to use the L' -oracle in a more versatile manner: Given an input x for which we want to determine if $x \in L$, we can use the L' -oracle multiple times on different inputs, and we can combine the answers obtained from these multiple uses of the L' -oracle in any (computable) way to determine if $x \in L$.

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

Theorem 4.4 [Sipser, Theorem 5.22 and Corollary 5.23] *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 4.5 [Sipser, Theorem 5.28 and Corollary 5.29] *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 4.6 *If $L \leq_m L'$ then $\bar{L} \leq_m \bar{L}'$.*

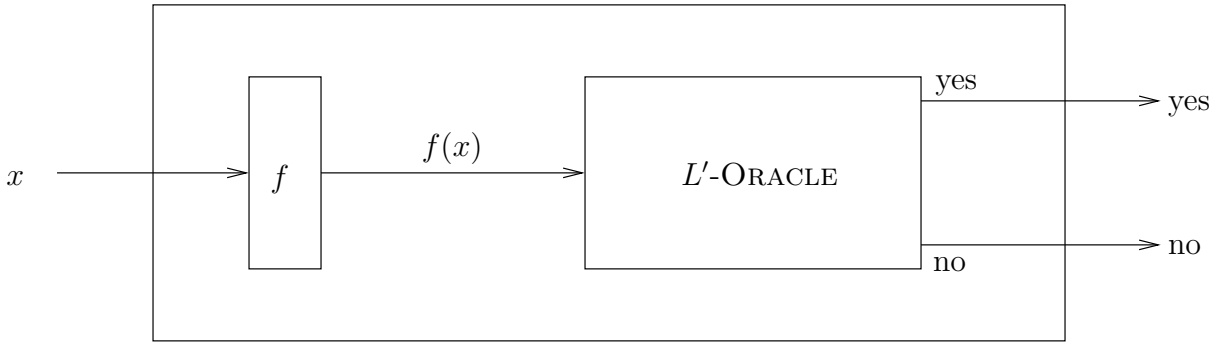


Figure 2: An L -decider that uses an L' -oracle, via mapping reduction f

Theorem 4.7 [Sipser, Exercise 5.6] *The mapping-reduces relation is transitive: If $L \leq_m L'$ and $L' \leq_m L''$ then $L \leq_m L''$.*

Exercise: Consider Theorems 4.4-4.7 and for each of them determine whether it 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.