ECS 120 Lesson 21 – Mapping Reducibility

Oliver Kreylos

Friday, May 18th, 2001

In the last two lectures, we have successfully used reduction techniques to prove several problems undecidable. Today we are going to refine our understanding of reducibility, to be able to apply these techniques to a broader class of problems.

1 Computable Functions

Until now, we only looked at Turing Machines as acceptors of languages: When given an input word $w \in \Sigma^*$, a machine M either accepts the word or rejects (or loops). In other words, we viewed Turing Machines as computing functions $f_M: \Sigma^* \to \{\text{accept}, \text{reject}\}$. In doing so, we ignored the fact that Turing Machines leave some string on their work tape when they finish. This becomes obvious from our definition of computation: A Turing Machine M accepts a word w, if and only if the initial configuration (ϵ, q_0, w) is related to an accepting configuration $(u, q_{\text{accept}}, v)$, where $u, v \in \Gamma^*$ are arbitrary strings. By shifting our focus from the acceptance to the string left behind on the work tape, we can view Turing Machines as *transducers* ("computers") by defining their output as follows: If a Turing Machine M halts (either accepting or rejecting), and the current tape content is a string $x\gamma_1\gamma\sqcup\sqcup\ldots$, where $x \in \Sigma^*$, $\gamma \in \Gamma^*$, and $\gamma_1 \in \Gamma \setminus \Sigma$ is a character not in the input alphabet, then the *output* of M is the string x. In other words, we consider as output the longest prefix of the current tape contents on halting that consists only of input characters. This definition of output is similar to the definition of input of a Turing Machine: The input word $w \in \Sigma^*$ consists only of input characters, and is delimited by the first blank character, which is not in Σ. Using this definition, the tape content of M on halting can directly be fed

into another Turing Machine, and the second machine's input would exactly be the first machine's output.

Since a (deterministic) Turing Machine M always produces the same output when given the same input, the input and output strings are related by a function $f_M: \Sigma^* \to \Sigma^*$: On input w, machine M produces $f_M(w)$ as output. This leads to the following definition:

Definition 1 (Computable Functions) *A function $f: \Sigma^* \to \Sigma^*$ is called* computable*, if and only if there exists a Turing Machine M that halts on any input word $w \in \Sigma^*$ and produces $f(w) \in \Sigma^*$ as output.*

We have already encountered transducing Turing Machines and computable functions several times, without actually pointing them out as such. For example, most reduction proofs seen so far consisted of an algorithm that created a description of a machine, wrote it to tape, and ran another Turing Machine on that output. Also, we have seen algorithms to convert between language representations: NFA to DFA, regular expression to NFA, context-free grammar to PDA, etc. All these algorithms can be viewed as transducing Turing Machines: Given an encoding of a context-free grammar, a Turing Machine could write the encoding of an equivalent PDA to its tape and halt.

2 Mapping Reducibility

We can use the notion of transducing Turing Machines and computable functions to refine our understanding of reducibility, in order to apply it to a broader class of problems. Let A and B be two languages. If there is a computable function f, such that any element in A is mapped by f to an element in B, and every element not in A is mapped to an element not in B, see Figure 1, then the membership of a word w in A can be decided by computing $f(w)$ and deciding its membership in B: $w \in A \iff f(w) \in B$. In this way, deciding problem A can be reduced to deciding problem B. This leads to the following definition:

Definition 2 *Mapping Reducibility Let $A, B \subset \Sigma^*$ be two problems. We say that A is* mapping-reducible *to B, if and only if there exists a computable function $f: \Sigma^* \to \Sigma^*$ such that $w \in A \iff f(w) \in B$. We write this as $A \leq_M B$.*

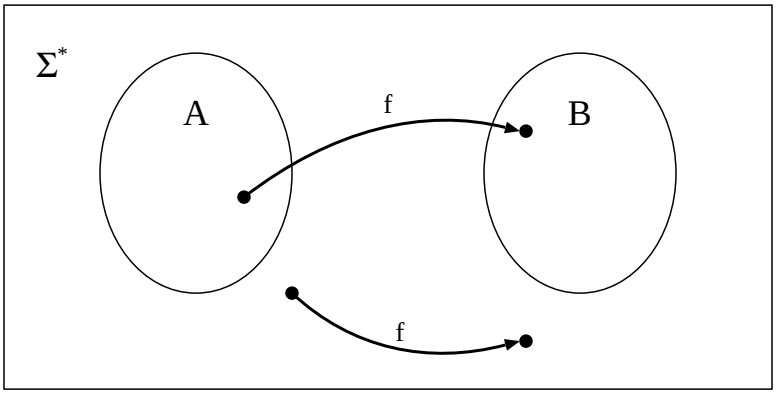

Figure 1: A mapping from a language A to a language B.

In order to prove that a problem A is mapping-reducible to another problem B, we have to show that there exists a computable function f as required by the definition. This function is usually described in the form of an algorithm. For example, to mapping-reduce the acceptance problem $\mathrm{A_{TM}}$ to the halting problem $\mathrm{HALT_{TM}}$, we construct the following transducing Turing Machine:

Algorithm $\mathrm{A_{TM}} \leq_M \mathrm{HALT_{TM}}$ On input $\langle M, w\rangle$, where M is a Turing Machine and w is an input word,

1. Construct a new TM M' in the following way: On input x,
   (a) Run machine M on x.
   (b) If M accepts, accept.
   (c) If M rejects, loop.
2. Write $\langle M', w\rangle$ onto the tape and halt.

This algorithm always terminates, hence it defines a computable function. Also, if machine M accepts w, then the constructed machine M' will accept w and halt. If, on the other hand, M does not accept w, the machine M' will loop – either implicitly, by M looping, or explicitly, in step (c). This means, the input word $\langle M, w\rangle$ is an element of $\mathrm{A_{TM}}$ exactly if the output word $\langle M', w\rangle$ is an element of $\mathrm{HALT_{TM}}$. Then, by definition, this algorithm mapping-reduces $\mathrm{A_{TM}}$ to $\mathrm{HALT_{TM}}$.

We can compare this mapping-reduction algorithm to the "usual" reduction algorithm seen before. The initial steps are identical, but the old reduction algorithm would explicitly run a hypothetical machine deciding $\mathrm{HALT}_{\mathrm{TM}}$ on the constructed machine M' and the input word w and then report its result. Here, we only create the input for the halting problem and leave the actual decision to someone else. In software engineering terms, "usual" reduction calls a decider for the reduced problem as a subroutine, whereas mapping-reduction is merely an *adaptor* for the reduced problem. This difference is illustrated in Figure 2.

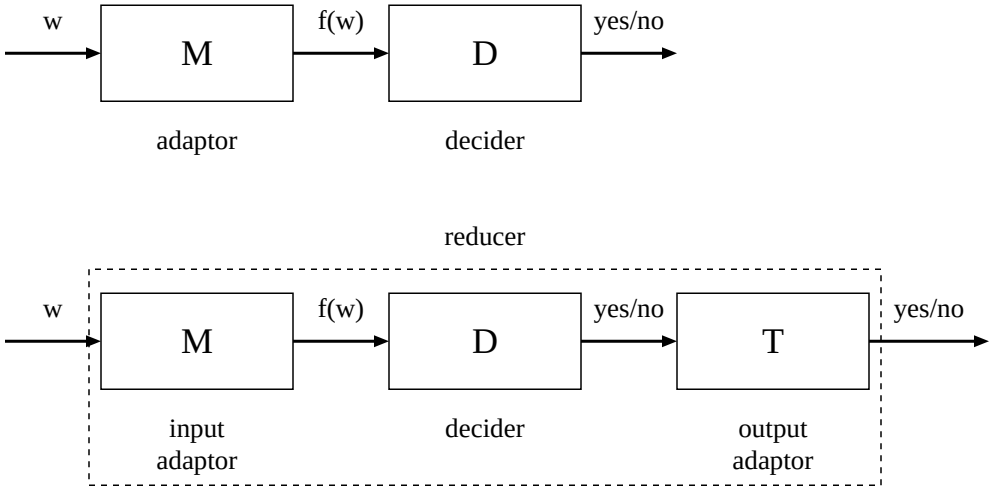

Figure 2: Mapping-reduction as an adaptor vs. reduction using a subroutine.

In this way, mapping-reducibility is a special case of reducibility as we used it before: In the latter, a problem is reduced by first transforming the input, then solving the reduced problem, and then transforming the reduced problem's result. In mapping-reducibility, this last step is omitted: The answer to the reduced problem is exactly the answer to the original problem.

Mapping-reducibility is more refined: Reduction in the usual way relies on the assumption that there is a decider for the reduced problem, because it is used as a subroutine. Generally, it is our goal to show that such a decider *does not* exist. Thus, reductions only serve theoretical purposes. Mapping-reducibility, on the other hand, only describes how to convert an entity of the original into an entity of the reduced problem, and is always valid, even if the reduced problem is undecidable. Therefore, it can also be used to reason about undecidable languages, as we will see shortly.

2.1 Properties of Mapping-reducibility

From the definition of mapping-reducibility, we can derive several properties:

- For every problem A, $A \leq_M A$. Every problem can be mapping-reduced to itself, by the identity function represented by a Turing Machine that immediately halts and leaves its input unchanged.
- If $A \leq_M B$, and $B \leq_M C$, then $A \leq_M C$. Mapping-reducibility is a transitive relation.
- If $A \leq_M B$, then $\overline{A} \leq_M \overline{B}$. If $w \in A \iff f(w) \in B$, then also $w \notin A \iff f(w) \notin B$, or $w \in \overline{A} \iff f(w) \in \overline{B}$.
- If $A \leq_M B$, and B is decidable, then A is decidable. Since mapping-reducibility is a special case of reducibility, this rule still applies.
- If $A \leq_M B$, and A is undecidable, then B is undecidable. This is the contraposition of the above property.
- If $A \leq_M B$, and B is recognizable, then A is recognizable. Since $w \in A \iff f(w) \in B$, if there is a recognizer for B, then A is recognized by running the recognizer for B on input $f(w)$.
- If $A \leq_M B$, and A is unrecognizable, then B is unrecognizable. This is the contraposition of the above property.

The last two rules show that mapping-reducibility can also be used to show the recognizability/unrecognizability of languages.

3 Classifying Problems by "Difficulty"

From what we have seen so far, we can (very informally) classify problems L (and their complements $\overline{L}$) by increasing level of "relative difficulty:"

1. $L \in \text{DL} \implies \overline{L} \in \text{DL}$. If a language is decidable, so is its complement. By definition, there exists a Turing Machine M such that M accepts every word $w \in L$, and rejects every word $w \in \overline{L}$.

2. $L \in \text{RE} \setminus \text{DL} \implies \overline{L} \notin \text{RE}$. If a language is recognizable, but not decidable, then its complement cannot be recognizable. By definition, there exists a Turing Machine M such that M accepts every word $w \in L$, but it might either reject or loop on any word $w \in \overline{L}$.

3. $L \notin \text{RE}$ and $\overline{L} \in \text{RE}$. If a language is not recognizable, but its complement is, we call it *co-recognizable*. There exists a Turing Machine that rejects every word $w \in \overline{L}$, but it might either accept or loop on any word $w \in L$.

4. $L \notin \text{RE}$ and $\overline{L} \notin \text{RE}$. There exists neither a Turing Machine to recognize L, nor a Turing Machine to recognize $\overline{L}$.

4 A Non-recognizable, Non-co-recognizable Language

So far, we have seen languages in the first three difficulty classes. For example,

- A_{LBA}, the acceptance problem for linear bounded automata, is decidable, and so is $\overline{\text{A}_{\text{LBA}}}$.

- A_{TM}, the acceptance problem for Turing Machines, is undecidable but recognizable; this means that $\overline{\text{A}_{\text{TM}}}$ is not recognizable but co-recognizable.

To show the existence of a problem in the fourth class, we consider the problem EQ_{TM}, the equivalence problem for Turing Machines, and use mapping-reductions from the non-recognizable language $\overline{\text{A}_{\text{TM}}}$ to show that neither EQ_{TM} nor $\overline{\text{EQ}_{\text{TM}}}$ are recognizable.

Algorithm $\overline{\text{A}_{\text{TM}}} \leq_M \text{EQ}_{\text{TM}}$ On input $\langle M, w \rangle$, where M is a Turing Machine and w is an input word,

1. Construct a Turing Machine M_1: On input x,

   (a) Reject.

2. Construct a Turing Machine M_2: On input x,

   (a) Run M on w (ignore the actual input word x).

(b) If M accepts, accept. If M rejects, reject.

3. Write $\langle M_1, M_2 \rangle$ onto the tape.

The first constructed machine M_1 has an empty language. The second constructed machine has an empty language exactly if M does not accept w. Thus, the languages of M_1 and M_2 are identical if and only if M does *not* accept w. Therefore, this algorithm is a mapping-reduction from $\overline{\mathrm{A}_{\mathrm{TM}}}$ to $\mathrm{EQ}_{\mathrm{TM}}$, which implies that $\mathrm{EQ}_{\mathrm{TM}}$ is not recognizable.

Algorithm $\overline{\mathrm{A}_{\mathrm{TM}}} \leq_M \overline{\mathrm{EQ}_{\mathrm{TM}}}$ On input $\langle M, w \rangle$, where M is a Turing Machine and w is an input word,

1. Construct a Turing Machine M_1: On input x,

   (a) Accept.

2. Construct a Turing Machine M_2: On input x,

   (a) Run M on w (ignore the actual input word x).

   (b) If M accepts, accept. If M rejects, reject.

3. Write $\langle M_1, M_2 \rangle$ onto the tape.

The first constructed machine M_1 has Σ^* as its language. The second constructed machine has Σ^* as its language exactly if M accepts w. Thus, the languages of M_1 and M_2 are *not* identical if and only if M does *not* accept w. Therefore, this algorithm is a mapping-reduction from $\overline{\mathrm{A}_{\mathrm{TM}}}$ to $\overline{\mathrm{EQ}_{\mathrm{TM}}}$, which implies that $\overline{\mathrm{EQ}_{\mathrm{TM}}}$ is not recognizable.