ECS 120 Lesson 19 – Reducibility

Oliver Kreylos

Monday, May 14th, 2001

Last time, we proved that the acceptance problem for Turing Machines, A_{TM}, is undecidable. We showed that by using a rather involved proof based on the diagonalization method proposed by Georg Cantor and later refined by Kurt Gödel. Proofs of this type are usually difficult to understand, and usually even more difficult to come up with in the first place. Today we will discuss another technique of proving certain problems undecidable that is easier to apply in practice.

1 Problem Reductions

Let us start by having a look at a typical software engineering task – designing an algorithm for a given problem. A typical approach to this is top-down development, where the stated problem is broken up into smaller problems, which are then solved independently, and later their results are combined to form a solution to the original problem. More generally, if there is a way to convert a solution for one problem R into a solution to another problem P, then solving problem P can be *reduced* to solving problem R, in symbols $P \longrightarrow R$.

As an example, let us consider the problem of finding the median of a set of numbers X. By definition, the median is the number that would end up in the middle of the set if the set were sorted. More formally, m is the median of a set of numbers X, if and only if

- $m \in X$ (the median must be an element of the set),
- $\left|\left\{ x \in X \mid x \leq m \right\}\right| \geq \lceil \frac{|X|}{2} \rceil$, and

- $\left|\left\{\, x \in X \mid x \geq m \,\right\}\right| \geq \lceil \frac{|X|}{2} \rceil$.

In other words, both the number of elements smaller than or equal to m and the number of elements larger than or equal to m must be at least half as large as the total number of elements in X divided by two, rounded up to the nearest integer.

The problem of finding the median seems difficult at first, but it can easily be reduced to the problem of sorting set X by the following algorithm:

Algorithm MEDIAN $\longrightarrow$ SORT Given a set of numbers X,

1. Sort the set X using some sorting algorithm.
2. Return the element at position $\lceil |X|/2 \rceil$ in the sorted set.

So to solve the median finding problem, we now only have to solve the sorting problem. In the terminology defined above, the problem of finding a median reduces to the problem of sorting a set.

Reducing an unknown problem to a known one is a good strategy to develop an algorithm, but it cannot be applied to proving that an unknown problem is undecidable. But let us use another approach: If P is a problem known to be undecidable, and P can be reduced to another problem R, i. e., $P \longrightarrow R$, then R must be undecidable as well. If there would be an algorithm for R, there would also be an algorithm for P: One would just solve R first, and then apply the reduction algorithm to solve P. But this is a contradiction to the assumption that P is undecidable; therefore, R must be undecidable as well.

It is important to note the "reverse" direction of the reduction: A known undecidable problem must be reduced to the unknown problem, not the other way around. This is a common mistake in undecidability proofs using reductions.

2 The Halting Problem – Final Revision

As a first example of using reductions, let us finally prove the undecidability of the halting problem for Turing Machines: $\mathrm{H}_{\mathrm{TM}} = \left\{\, \langle M, w \rangle \mid M \text{ halts on } w \,\right\}$. We will be using the acceptance problem for Turing Machines, which we proved undecidable last time, as a problem to reduce from (reduce *from*, not reduce *to*).

The task at hand now is to design an algorithm that can convert an algorithm to solve the halting problem into an algorithm to solve the acceptance problem. The idea is to improve the "flawed" algorithm to decide A_{TM} by simulating the computation of M on w that we have seen in our discussion of Universal Turing Machines. As we said, a UTM can almost solve the acceptance problem: the only flaw is that a UTM will not terminate if the simulated machine M loops on input word w. But using our hypothetical algorithm to solve the halting problem, we can check for this case before starting simulation. Thus, our reduction algorithm works as follows:

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

1. Run the Turing Machine solving the halting problem on input $\langle M, w \rangle$. If it rejects, reject.

2. Simulate the computation of M on input w by using a UTM. If M accepts, accept; if M rejects, reject.

This is a valid algorithm, because we assumed that the halting problem is decidable. Thus the machine solving it is a decider and will always terminate after a finite number of steps. If the decider for H_{TM} rejects, we know that M will loop on w; thus, the machine for A_{TM} must reject. Otherwise machine M is guaranteed to terminate on word w, so all left to do is find out whether it will accept or reject. This can be done by simulating M's computation. This simulation will always terminate (M is guaranteed to terminate), and therefore the complete algorithm can decide the acceptance problem.

But we proved the acceptance problem to be undecidable last time; therefore the assumption that the halting problem can be decided must have been wrong, and the halting problem is undecidable.

3 Other Problems About Turing Machines – Rice's Theorem

The same strategy can be applied to other questions about Turing Machines and their languages as well, using different reduction algorithms and different known undecidable problems.

3.1 The Emptiness Problem for Turing Machines

An interesting question about a Turing Machine is whether it accepts any word at all, i. e., whether its language is empty or not. For finite state machines, for example, this could easily be decided by searching for a valid path from the start state to a final state in the automaton's transition diagram. For a Turing Machine, however, this question cannot be decided. We prove this claim by reducing the acceptance problem to the emptiness problem using the following reduction algorithm:

Algorithm $\mathrm{A_{TM}} \longrightarrow \mathrm{E_{TM}}$ 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 that works the following way: On input x,
   (a) Check whether $x \neq w$. If so, reject.
   (b) Run M on input x.
2. Run the Turing Machine deciding the emptiness problem on input $\langle M_1 \rangle$. If it accepts, reject; otherwise, accept.

The idea behind this algorithm is to construct a "helper Turing Machine" M_1 that accepts exactly the intersection of the languages $L(M)$ and $\{w\}$. If this language is empty, the language of M does not contain w, i. e., M does not accept w. Otherwise, M does accept w. The helper Turing Machine M_1 works in the following way: It first checks whether the input word is identical to w. If not, the word is not in the intersection $L(M) \cap \{w\}$ and is rejected. Otherwise, M_1 computes exactly as M. Obviously, the language of M_1 is exactly the intersection $L(M_1) = L(M) \cap \{w\}$. By feeding the description of M_1 into the decider for the emptiness problem, we can find out whether $L(M_1)$ is empty, which is equivalent to $w \notin L(M)$, or non-empty, which is equivalent to $w \in L(M)$.

It is important to note that M_1 is not a deciding Turing Machine for the acceptance problem. In fact, it is not a deciding Turing Machine at all: If M loops on w, so will M_1 on input w. Machine M_1 is only a helper construct that is fed into the decider for the emptiness problem, which in turn has some "magical" way of finding out whether M_1's language is empty. Machine M_1

will never be executed by the above algorithm, so the fact that it can loop does not matter.

Altogether, since the algorithm is a valid reduction from $\mathrm{A_{TM}} \longrightarrow \mathrm{E_{TM}}$, we can conclude that $\mathrm{E_{TM}}$ must be undecidable.

3.2 Rice's Theorem

We already saw that at least three interesting problems about Turing Machines and their languages are undecidable. We could also ask some other questions:

- Are two Turing Machines M_1 and M_2 equivalent, i. e., are their languages $L(M_1)$ and $L(M_2)$ identical?
- Is the language of a Turing Machine M regular, i. e., is there a DFA D such that $L(M) = L(D)$?
- Is the language of a Turing Machine M context-free, i. e., is there a PDA P such that $L(M) = L(P)$?

As it turns out, all of these questions are undecidable. This leads to a fundamental theorem first stated by Rice:

Theorem 1 (Rice's Theorem) *Every non-trivial problem about the languages of Turing Machines is undecidable.*

To understand when exactly this theorem applies, we have to exactly define the two terms "non-trivial" and "about the languages..."

1. A non-trivial problem is one that has at least one instance and one non-instance. In other words, problem P is non-trivial if and only if there exists a $w_1 \in \Sigma^*$ such that $w_1 \in P$, and there exists a $w_2 \in \Sigma^*$ such that $w_2 \notin P$. A problem that violates the first constraint does not have any instances; it can be decided by a machine that always rejects. If a problem violates the second constraint, every possible word is an instance. Such a problem can be decided by a machine that always accepts. Both kinds of problems are not interesting.
2. A problem P is about the languages of Turing Machines if its answer does not depend on the specific structure of the Turing Machine, but

only on its language. In other words, the answer of problem P on two machines M_1 and M_2 must be identical if the two machines have identical languages: $L(M_1) = L(M_2) \Longrightarrow P(\langle M_1 \rangle) = P(\langle M_2 \rangle)$.

As we will see in a later homework assignment, some questions about the structure of Turing Machines *are* decidable – this is not a violation of Rice's theorem if those questions do not satisfy the condition of point 2 from above. For example, the question whether a Turing Machine has less than 50 states is definitely decidable; it does not contradict Rice's theorem, because there can be two different machines M_1 and M_2, where M_1 has 49 states and M_2 has 51 states, but both accept the same language (one way to construct M_2 is to just add two unused states to M_1).

4 Linear Bounded Automata

Before we move on to show another common reduction technique, let us finally close the gap between pushdown automata and Turing Machines. As was said in class, there is an equivalent machine type for each level in the Chomsky Hierarchy, but so far we have only seen PDAs accepting CH–2 languages and Turing Machines accepting CH–0 languages. To create a machine type for context-sensitive (CH–1) languages, we restrict Turing Machines by not allowing them to write past the end of their input. In other words, we create a new machine type called *linear bounded automaton (LBA)* that only has a finite amount of tape: Exactly enough to write down the input word. Apart from this restriction, LBAs work exactly like general Turing Machines.

We are not going to prove that LBAs accept context-sensitive languages, but we will show that the acceptance problem for LBAs is decidable, a tool we can later use to construct more complex reductions. This proof will also show that the class CH–1 is a subset of the class of decidable languages, as was stated earlier in class.

4.1 The Acceptance Problem for LBAs

To show that the acceptance problem for LBAs is decidable, we have to construct an algorithm that can decide it. To design such an algorithm, we reconsider the "flawed" approach of simulating the execution of a Turing Machine that we unsuccessfully tried to apply to the acceptance problem for Turing Machines. In that case the approach broke down, because Turing

Machines can loop in ways that can not be detected by another Turing Machine. For LBAs, this is not the case. To understand why not, let us recall the definition of computation for Turing Machines.

We defined an instantaneous description (ID) of a Turing Machine as a "snapshot of computation" that allows us to unambiguously examine the configuration a Turing Machine is in. An ID was a triple encoding the

- current tape contents as a string $\gamma \in \Gamma^*$,
- the current state as an element $q \in Q$, and
- the current head position by splitting up the tape string γ into a left part u and a right part v, such that the current character is the first character of v.

For a Turing Machine, the number of different configurations was infinite, which made it impossible to detect a loop in the computation, which in turn made the acceptance problem undecidable.

For an LBA, however, the number of possible IDs is finite: The tape string is always of length n (if n is the length of the input word w, $n = |w|$), and since there is a finite number of characters in Γ, there can be at most $|\Gamma|^n$ different tape strings. The number of states, $|Q|$, is also finite, as is the number n of possible head positions. Therefore, the total number of possible IDs is $N = |Q| \cdot n \cdot |\Gamma|^n$. This number may be very big, but it is always finite.

We now recall that computation of a Turing Machine was defined as a chain of IDs $\mathrm{ID}_0 \vdash \mathrm{ID}_1 \vdash \cdots \vdash \mathrm{ID}_k$, where ID_0 is an initial configuration, and the last configuration in the chain is a halting configuration. If a Turing Machine loops, the chain is infinite and does not have a final halting configuration.

We now observe that, if any ID appears twice in a computation chain, the Turing Machine must loop. The reason is that computation is completely and deterministically specified by the current ID; if a chain goes from one ID a to another, and then comes back to ID a, the computation must repeat this loop forever. A Turing Machine is not halted externally by the end of the input word, as a finite state machine is; it can only halt by itself. That means if there is a loop in the computation chain, the computation must loop itself. We can now apply the Pigeonhole Principle, as we did when deriving the Pumping Lemma for regular languages: If a computation of an LBA on an input word of length n continues for at least N steps without halting,

there must be a double occurrence of some ID in the chain of $N + 1$ IDs. Therefore, the machine must loop.

With this observation, we can construct a decider for the acceptance problem for LBAs:

Algorithm $\mathrm{A}_{\mathrm{LBA}}$ On input $\langle M, w \rangle$, where M is an LBA and w is an input word,

1. Compute the total number N of IDs of M when given input w.
2. Simulate machine M for at most N steps of computation.
3. If M accepted, accept. If M rejected, reject. Otherwise, M must be in a loop; reject.

This algorithm is correct; it exactly accepts if M accepts, and it rejects if M either rejects or loops. It also terminates after at most N steps; this means the acceptance problem for LBAs is decidable.