7. REDUCTIONS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

October 2024

We have proven that there exist undecidable problems and then exhibited one particular example: *the halting problem*.

But what if we wanted to systematically classify newly encountered problems based on decidability? We are given a new problem f and we want some method by which we can tell whether it is decidable or undecidable.

At this point, we could prove decidability by designing a machine M_f that decides f.

For undecidability, we could attempt a proof by contradiction in the style of creating machine D from the previous lecture, which helped us show that a machine that decides `HALT` cannot exist.

But we will now introduce a handy "tool" that allows us to use *known* problems in order to establish the (un)decidability of *new* problems: **reductions**.

We will start with an informal description of reductions, but it should be pointed out that you are already familiar with and have used reductions before in the context of programming!

Consider this programming task:

"Write a program that checks if a string is a palindrome."

You could start writing a solution "from scratch", keeping track of two indices (which start from the beginning and end of the input string, respectively, and work their way towards the middle) and checking if the characters at those indices match. Or, you might come up with a very short solution:

```
1 int isPalindrome(const char *s) {
2     return (strcmp(s, reverse(s)) == 0);
3 }
```

That is, you've *reduced* the problem of determining whether a string is a palindrome, to the problem of finding its reverse. In programming, we write these sort of solutions for a variety of reasons: to keep code readable and maintainable, to modularize it so that we can use bits of it in other places etc.

In the context of Computability Theory, we shall use reductions for the purpose of proving *(un)solvability*[†]: if `reverse` is *solvable*, then so is `isPalindrome`.

(Actually, we also make use of the `strcmp` function, so we should say that you've reduced our problem to the one of determining the reverse *and* comparing it with the original.)

1 Intuition for Turing Machines

1.1 Prefix strings

Consider the following decision problem, `PREFIX`:

"Given two binary strings s and p, does s start with p?"

For $s = 0011010010$ and $p = 001101$, the answer is "yes", while for $s = 01010110$ and $p = 0110$, the answer is "no".

Is the problem decidable?

To prove that it is, we can design a Turing Machine that decides it.

We will call this machine `isPrefix` and we'll adopt the following conventions:

[†]By which we mean any of: "computability", "acceptability", "decidability".

- the input of the machine will be a string over the alphabet $\Sigma = \{0, 1, \&\}$: it will be of the form $s\&p$ (e.g. 0011010010&001101)
- the machine will have two tapes

The machine will first go right, "moving" s from the first tape onto the second, by deleting each symbol on the first tape and writing it on the second.

Once it reaches the delimiter symbol &, it will reset (move left) the head of the second tape to the first symbol of s, then move the first head onto the first symbol of p.

The machine then moves both heads in tandem to the right:

- if there is a symbol mismatch and the head on the first tape is scanning a 0 or a 1, reject.
- if the head of the first tape reaches a blank symbol, accept.

Note 7.1. In the `resources` directory associated with this lecture, you will find the full description of this machine, written in the syntax for the online simulator.

1.2 Substrings

Let's consider a different, but similar, decision problem `SUBSTRING`:

"Given two binary strings s and p, does s *contain* p?"

For $s = 0011010010$ and $p = 1010$, the answer is "yes"[‡], while for $s = 01010110$ and $p = 0100$, the answer is "no".

Is the problem decidable?

To prove that it is, we can design a Turing Machine that decides it... Or, we could show that a solution to the prefix problem entails a solution to this one.

Consider the following proposition, which asserts that a string p is a substring of s iff p is the prefix of some suffix of s:

$$\texttt{SUBSTRING}(enc((s,p))) = TRUE \Leftrightarrow \exists i, 1 \leq i \leq n, \text{ s.t. }, \texttt{PREFIX}(enc((s_i...s_n, p))) = TRUE$$

Where $s_1, s_2, ..., s_n$ are the symbols that make up s.

`PREFIX` being decidable, we know that there is a machine that decides it (we even presented such a machine, `isPrefix`, in detail in the previous section). Starting from this machine, we can construct another, `isSubstring`, that decides `SUBSTRING`. Below, we give a pseudocode description of the machine:

`isSubstring`$[s\&p]$:

1. **for** $1 \leq i \leq |s|$:
2. **compute** $s' = s_i...s_n$
3. **if** $|s'| < |p|$ transition to N
4. **simulate** `isPrefix`$[s'\&p]$
5. **if** `isPrefix`$[s'\&p] \rightarrow TRUE$
6. **transition** to Y

There are many techniques of reducing a problem to another; in this course, we will only address what are known as *"mapping reductions"*, in which we take an input for a problem and transform it into the input for another problem.

The reduction above, from `PREFIX` to `SUBSTRING` is **not** a mapping reduction.

[‡]The string 1010 can be found starting from the fourth symbol of string 0011010010.

2 Mapping reductions

In this section, let $f : \Sigma^* \to \{FALSE, TRUE\}$ and $g : \Sigma^* \to \{FALSE, TRUE\}$ be two decision problems over some alphabet Σ.

Definition 7.1.

$$f \leq_m g \stackrel{\text{def}}{=} \exists t : \Sigma^* \to \Sigma^*, \text{ s.t. } \begin{cases} t \text{ is computable} \\ \forall w \in \Sigma^\star, f(w) = g(t(w)) \end{cases}$$

Read the left-hand-side as "*f is mapping reducible to g*".

You can think of reducibility as a tool to compare the "hardness" of problems; if $f \leq_m g$, than solving f is "at most as hard as solving g".

The computability of t is important, because it means that if we have a machine that decides M_g, we can construct a machine that decides f: on any input w, first compute $t(w)$, then simulate $M_g[t(w)]$.

Theorem 7.1. (g is decidable) $\wedge$ ($f \leq_m g$) $\Rightarrow$ f is decidable

Proof. Remember what it means for a problem to be *decidable*: there exists some machine that decides it. So this theorem tells us that if there is a machine M_g that decides g and f is reducible to g, there is a machine M_f that decides f.

f being reducible to g means that there is some computable function t such that for any input w, $f(w) = g(t(w))$. t being computable, means there is some machine M_t that *computes* it.

Based on M_g and M_t, we will construct a machine M_f; the proof is similar to the composition exercise from the lab on computing. M_f will contain both the states+tape symbols of M_t and M_g, and almost the same transitions, with the following exception: when M_t would transition to H, M_f will transition to some auxiliary state that places the read/write head over the first non-blank symbol, then transitions to M_g's initial state.

M_f's initial state is the same as M_t's initial state.

Expressed as pseudocode:

$M_f[w]$:

1. **simulate** $M_t[w]$ to compute $t(w)$
2. **simulate** $M_g[t(w)]$
3. **if** $M_g[t(w)] \to TRUE$
4. **transition** to Y
5. **else**
6. **transition** to N

□

Theorem 7.2. (g is acceptable) $\wedge$ ($f \leq_m g$) $\Rightarrow$ f is acceptable

Proof. We use the exact same construction as in the proof above. If M_g accepts g, then M_f will accept f. The only difference is that there might be some outputs of t on which $M_g[t(w))]$ doesn't halt; $M_f[w]$ won't halt either, but it's ok because $g(t(w)) = FALSE \Rightarrow f(w) = FALSE$.

□

Theorem 7.3. (f is undecidable) $\wedge$ ($f \leq_m g$) $\Rightarrow$ g is undecidable

This is a direct consequence of Theorem 7.1 and it will be our main tool for proving a new problem is undecidable.

Similarly, we get:

Theorem 7.4. (f is unacceptable) $\wedge$ ($f \leq_m g$) $\Rightarrow$ g is unacceptable

3 More undecidable problems

To see how an undecidability proof using reductions looks like, we introduce two new interesting decision problems: ALL and ANY.

We need a known undecidable problem to reduce to each of the new ones, to show that they, in turn, are undecidable.

For now, the only such problem we know is HALT.

3.1 ALL

Definition 7.2.

$$\texttt{ALL}(enc(M)) = \begin{cases} TRUE & \forall w \in \Sigma^*, M[w] \text{ halts} \\ FALSE & otherwise \end{cases}$$

ALL is similar to HALT. Each of them takes as input a machine M; but whereas HALT also takes some specific input w to tell whether $M[w]$ halts, ALL tells if M halts on *"all"* inputs.

Theorem 7.5. $\texttt{ALL} \notin R$

Proof. We will prove that $\texttt{HALT} \leq_m \texttt{ALL}$. This result, combined with that of Theorem 7.3 shows that $\texttt{ALL} \notin R$.

Our transformation t takes as input the encoding of a (machine, string) tuple (M, w) and maps it to the encoding of a single machine M'. We need to specify the behavior of M' in such a way that: $\texttt{HALT}(enc((M, w))) = TRUE \Leftrightarrow \texttt{ALL}(enc(M')) = TRUE$.

$M'[v]$:

1. **erase** v
2. **write** w
3. **simulate** $M[w]$

Is M' well-defined?

The erasing of the input v should seem straightforward: you've implemented machines that erase the contents of their tape at the lab.

But what does it mean for M' to refer to w or to M? The answer is that the contents of w and the encoding of M are *hardcoded* into M'.

You can imagine that M' has a series of states: $q_{w_1}, q_{w_2}, ..., q_{w_n}$ – one for each symbol w_i of w. For each of these states and the blank symbol, there is a transition that writes the symbol, goes to the state corresponding to the next symbol and moves the head right: $\delta_{M'}(q_{w_i}, \square) = (q_{w_{i+1}}, w_i, \rightarrow)$.

Similarly, all the states, tape symbols and transitions of M are hardcoded into M'; after writing the last symbol of w on the tape, M' resets its head onto the first symbol, w_1, and transitions to the initial state of M.

But is the transformation good?

If $\texttt{HALT}(enc((M, w))) = TRUE$, then $M[w]$ halts, so on any input v, M', which has the same behavior as $M[w]$ will also halt, thus $\texttt{ALL}(enc(M')) = TRUE$.

Similarly, if $\texttt{ALL}(enc(M')) = TRUE$, then M' halts on all inputs, which means $M[w]$ halts, so $\texttt{HALT}(enc((M, w))) = TRUE$.

But what about the computability of the transformation? At first, you might wonder what happens if $M[w]$ doesn't halt; but that is of no concern to the machine M_t which computes t. M_t simply has to read the encoding of (M, w) and write out the encoding of M', a machine which at some point will simulate $M[w]$. But it is not M_t which runs $M[w]$! Thus M_t can exist, t is computable.

□

Note 7.2. ALL and HALT are decision problems, so their domain is the entire set of strings: Σ^*.

However, the encoding scheme we described for Turing Machines makes it so that not every string is a valid encoding of a machine. So what do we mean when we say $\texttt{HALT}(enc((M,w)))$ and what is the value of $\texttt{HALT}(w)$ when w is not a valid encoding of a (M,w) tuple?

The answer is that $\texttt{HALT}(w)$ is then $FALSE$. Whenever we define a decision problem and write, for the input variable, an encoding of some object, we implicitly mean that for any string that is not a valid encoding of such an object, the answer is $FALSE$.

This is then relevant for reductions $f \leq_m g$; whenever f takes as input an encoded object, the transformation needs to map it to an input for which g is $FALSE$.

But this is always easy to do; for $\texttt{HALT} \leq_m \texttt{ALL}$, any input that is not a valid encoding of a (M,w) tuple can be mapped to ε, or any other string that is not a valid encoding of a Turing Machine.

From now on, such subtlety will be kept implicit.

3.2 ANY

Definition 7.3.

$$\texttt{ANY}(enc(M)) = \begin{cases} TRUE & \exists w \in \Sigma^*, M[w] \text{ halts} \\ FALSE & otherwise \end{cases}$$

Theorem 7.6. $\texttt{ANY} \notin R$

Proof. As before, we will reduce HALT to our new problem, i.e. we will prove that $\texttt{HALT} \leq_m \texttt{ANY}$. This result, combined with that of Theorem 7.1 show that $\texttt{ANY} \notin R$.

Our transformation t takes as input the encoding of a (machine, string) tuple (M,w) and maps it to the encoding of a single machine M'. We need to specify the behavior of M' in such a way that: $\texttt{HALT}(enc((M,w))) = TRUE \Leftrightarrow \texttt{ANY}(enc(M')) = TRUE$.

$M'[v]$:

1. **if** $v = 1011$:
2. **erase** v
3. **write** w
4. **simulate** $M[w]$
5. **else**
6. **loop**

M' only simulates $M[w]$ for a particular input, arbitrarily chosen by the transformation. We could've actually used the same description for M' as in the ALL reduction above, but we choose to write it this way in order to emphasize the differences between ALL and ANY.

If $\texttt{HALT}(enc((M,w))) = TRUE$, then $M[w]$ halts, so on input 1011, M' will have the same behavior as $M[w]$, so it will also halt, thus $\texttt{ANY}(enc(M')) = TRUE$.

Similarly, if $\texttt{ANY}(enc(M')) = TRUE$, then there is an input on which M' halts; because the "else" branch always makes it loop, we can only look at the "then" branch of the "if": here M' will behave as $M[w]$, so it means $M[w]$ halts and $\texttt{HALT}(enc((M,w))) = TRUE$.

□

Theorem 7.7. $\texttt{ANY} \in RE$

The first idea that might come up for a machine that accepts ANY is to start from the first string[§] and run the input machine on it; if it doesn't halt, move on to the next string and so on. But this idea doesn't work; if the input machine doesn't halt on the first string, then our simulation will never halt, so we cannot move on to the second string.

[§]Remember that $|\Sigma^*| = |\mathbb{N}|$, so there is some ordering of strings.

What we need to do is to run the input machine *concurrently* on multiple strings. But on how many strings? We cannot simply run one step for each string, then move on to running the second step for each string – we would never finish the strings for which we need to run one step!

We must, at every step, partially run just one more input.

Proof. We prove $\texttt{ANY} \in RE$ by constructing a machine that accepts $\texttt{ANY}$.

$A_{\texttt{ANY}}[enc(M)]$:

1. $n := 0$
2. $strings := \emptyset$
3. **forever**:
4. $\quad strings := strings \cup \{n^{th} \text{ string}\}$
5. $\quad$ **for each** w **in** $strings$:
6. $\quad\quad$ **simulate** $M[w]$ for one more step (or the first step, for the string just added)
7. $\quad\quad$ **if** $M[w]$ halted, transition to Y
8. $\quad n := n + 1$

If there is any string w for which $M[w]$ halts, it will be discovered by $A_{\texttt{ANY}}$, so $A_{\texttt{ANY}}$ accepts $\texttt{ANY}$.

□

Notice that, in the previous section, we did not present the construction of a machine that accepts ALL. The reason for that is that such machine doesn't exist!

To prove that ALL is not acceptable, we could reduce a known unacceptable problem to it. But, at the moment, we don't know any such problem, so we need to take a slight detour, studying the *complement* of a decision problem, to arrive at an example: $\overline{\texttt{HALT}}$.

4 Complement of a decision problem

Let $f : \Sigma^* \to \{FALSE, TRUE\}$ be a decision problem.

Definition 7.4. The complement of f is a decision problem $\overline{f}$, such that

$$\overline{f}(w) = \begin{cases} TRUE & f(w) = FALSE \\ FALSE & f(w) = TRUE \end{cases}$$

In other words, the complement of a decision problem is the problem that maps each string to the opposite $\{FALSE, TRUE\}$ answer.

When it comes to the relationship between a problem and its complement with regards to decidability and acceptability, we can obtain two interesting theorems.

Theorem 7.8. $f \in R \Leftrightarrow \overline{f} \in R$

This tells us that, if a problem is decidable, so is its complement. We can say that the set of decidable problems, R, is *"closed under complementation"*.

Proof. We will prove the right implication: $f \in R \Rightarrow \overline{f} \in R$.

Because the complement of a problem's complement is the original problem ($\overline{\overline{f}} = f$), the proof of the left implication is nearly-identical.

$f \in R$ means that there exists a machine D_f that decides f. Starting from it, we construct a machine $D_{\overline{f}}$ that decides $\overline{f}$.

$D_{\overline{f}}$ will have all the same elements of D_f, except that all transitions to Y will be replaced by transitions to N and vice-versa.

I.e., any transition $\delta_{D_f}(q, g) = (Y, h, d)$ will become $\delta_{D_{\overline{f}}}(q, g) = (N, h, d)$ and any transition $\delta_{D_f}(q, g) = (N, h, d)$ will become $\delta_{D_{\overline{f}}}(q, g) = (Y, h, d)$, for some $q \in Q;\ g, h \in \Gamma;\ d \in \{\leftarrow, -, \rightarrow\}$.

In effect, any input accepted by D_f is rejected by $D_{\overline{f}}$ and any input rejected by D_f is accepted by $D_{\overline{f}}$.

So $D_{\overline{f}}$ decides $\overline{f}$, thus $\overline{f} \in R$.

□

Theorem 7.9. $(f \in RE) \wedge (\overline{f} \in RE) \Leftrightarrow f \in R$

If a problem is acceptable and its complement is acceptable, then the problem is *decidable*. Combined with the previous theorem, we obtain that its complement is also decidable.

Proof. We will prove the right implication: $(f \in RE) \wedge (\overline{f} \in RE) \Rightarrow f \in R$.

The left implication can be obtained from the fact that $R \subseteq RE$ and the previous theorem.

$f \in RE$ means that there exists a machine A_f that accepts f. Similarly, $\overline{f} \in RE$ means that there exists a machine $A_{\overline{f}}$ that accepts $\overline{f}$.

Starting from these, we construct a machine D_f that decides f.

It's important to note that, for any string w, either $f(w) = TRUE$ or $\overline{f}(w) = TRUE$. Which means that either $A_f[w]$ will halt in the Y state, or $A_{\overline{f}}$ will halt in the Y state.

So we will run these two machines concurrently and notice which one halts. Note that if $A_{\overline{f}}$ halts, we need to reverse its output. We also said that machines that accept a problem don't *necessarily* loop, but may also halt in final state N, whenever the problem's answer is $FALSE$, so we need to take this into account.

$D_f[w]$:

1. **forever:**
2. **simulate** one step of $A_f[w]$
3. **if** $A_f[w]$ halted in state Y:
4. **transition** to Y
5. **elseif** $A_f[w]$ halted in state N:
6. **transition** to N
7. **simulate** one step of $A_{\overline{f}}[w]$
8. **if** $A_{\overline{f}}[w]$ halted in state Y:
9. **transition** to N
10. **elseif** $A_{\overline{f}}[w]$ halted in state N:
11. **transition** to Y

□

5 Unacceptable problems

Our first example of a problem outside RE is the complement of the halting problem.

Definition 7.5.

$$\overline{\texttt{HALT}}(enc((M,w))) = \begin{cases} TRUE & M[w] \rightarrow \perp \\ FALSE & otherwise \end{cases}$$

Note 7.3. We now reiterate what was said in Note 7.2. Taken as it is, for any string that is not a valid encoding of a (M, w) tuple, both HALT and $\overline{\texttt{HALT}}$ should be $FALSE$; but this contradicts our definition of a problem's complement.

So we convene that whenever a problem is presented as *"the original"*, then it will be $FALSE$ on invalid strings, while the problem presented as *"its complement"* will be $TRUE$.

This is merely a convention and does not affect any of our results; we could just as well say that HALT is $TRUE$ on invalid strings and $\overline{\texttt{HALT}}$ is $FALSE$.

Why is $\overline{\texttt{HALT}}$ not in RE?

Remember that, in the last lecture, we proved that $\texttt{HALT} \in RE$. Earlier, in this lecture, we proved Theorem 7.9. So, if $\overline{\texttt{HALT}} \in RE$, then $\texttt{HALT} \in R$. But in the last lecture, we also proved that $\texttt{HALT} \notin R$, which leads to a contradiction. Thus our assumption must be false and $\overline{\texttt{HALT}} \notin RE$.

However, there is still some "solvability" to $\overline{\texttt{HALT}}$: we can write a machine that, whenever $\overline{\texttt{HALT}}(enc((M,w)) = FALSE$, halts in the final state N, giving us the correct answer. Whenever $\overline{\texttt{HALT}}(enc((M,w)) = TRUE$, our machine might halt in Y, or run forever. This is obvious due to the symmetry to HALT: we just take the machine that accepts HALT and swap Y for N.

We might call $\overline{\texttt{HALT}}$ a "rejectable problem"; in Computability Theory, such problems are simply known as "problems whose complement is acceptable", which leads us to the following set:

Definition 7.6. $coRE \stackrel{\text{def}}{=} \{f \in \mathbb{D} \mid \overline{f} \in RE\}$

Note that $R = RE \cap coRE$.

These three sets capture all decision problems that are, in some sense, "solvable": either we can always produce the correct answer (R), or produce the correct answer whenever it's $TRUE$ (RE), or produce the correct answer whenever it's $FALSE$ ($coRE$).

We shall now see that ALL belongs to neither of these sets.

6 Beyond $RE \cup coRE$

Theorem 7.10. $\texttt{ALL} \notin RE$

Having discovered an unacceptable problem, $\overline{\texttt{HALT}}$, we can now prove this using reductions. We will show that $\overline{\texttt{HALT}} \leq_m \texttt{ALL}$.

Proof. We cannot have the same reduction as in the case of $\texttt{HALT} \leq_m \texttt{ALL}$; we cannot check if a machine runs forever and halt if it does. We have to resort to a clever trick:

$M'[v]$:

1. **erase** v
2. **write** w
3. **simulate** $M[w]$ for $|v|$ steps
4. **if** $M[w]$ halted
5. **loop**
6. **else**
7. **halt**

We're not checking whether $M[w]$ runs forever, but rather we're checking if the machine is still running after i steps, for some natural number i which depends on the input. If $M[w]$ runs forever, then it will not halt in i steps, for any i.

Note that i doesn't have to be the *length* of the input; we chose this for ease of writing; it could be the result from some other function from strings to naturals. But that function has to be **strictly increasing** (thus having no maximum value). Otherwise, it would be possible for $M[w]$ to halt in a number of steps just above that maximum value.

$\overline{\texttt{HALT}}(enc((M,w)) = TRUE$ means that $M[w]$ never halts; thus it will not halt after any number of steps, so M' will always take the "else" branch and halt for every input; $\texttt{ALL}(enc(M')) = TRUE$.

$\overline{\texttt{HALT}}(enc((M,w)) = FALSE$ means that $M[w]$ halts and it will do so in some number of steps n; thus for any input v, such that $|v| \geq n$, the simulation of $M[w]$ will have halted, so M will take the "then" branch, looping forever; so M' doesn't halt on every input and $\texttt{ALL}(enc(M')) = FALSE$.

Together with Theorem 7.4 we obtain: $\texttt{ALL} \notin RE$.

□

We now go one step further:

Theorem 7.11. $\texttt{ALL} \notin coRE$

To prove this, we need to show that $\overline{\texttt{ALL}} \notin RE$.

Proof. We will do this via a reduction from $\overline{\texttt{HALT}}$, proving $\overline{\texttt{HALT}} \leq_m \overline{\texttt{ALL}}$.

But we have already proven this! Remember the definition of a mapping reduction and that of a problem's complement. The transformation required by a mapping reduction maps all inputs for which $\overline{\texttt{HALT}}$ is $TRUE$ onto strings for which $\overline{\texttt{ALL}}$ is $TRUE$; similarly for $FALSE$.

This means that *the same transformation* used to show $\texttt{HALT} \leq_m \texttt{ALL}$, can be employed between their complements. In general, $f \leq_m g \Leftrightarrow \overline{f} \leq_m \overline{g}$.

We have shown that $\texttt{HALT} \leq_m \texttt{ALL}$ in subsection 3.1, thus $\overline{\texttt{HALT}} \leq_m \overline{\texttt{ALL}}$ and so $\overline{\texttt{ALL}} \notin RE$ ($\texttt{ALL} \notin coRE$).

□

In conclusion, $\texttt{ALL} \notin RE \cup coRE$, so we can't solve it in any way.