# St. Francis Xavier University
# Department of Computer Science

# CSCI 356: Theory of Computing
# Lecture 5: Reducibility
# Fall 2023

## 1 Mapping Reductions

In our proof showing that $A_{\mathsf{TM}}$ was undecidable, we constructed a Turing machine $\mathcal{D}$ that took as input $\langle \mathcal{M} \rangle$, the encoding of a Turing machine $\mathcal{M}$, converted that input to the form $\langle \mathcal{M}, \langle \mathcal{M} \rangle \rangle$, and gave that converted input to another Turing machine $\mathcal{H}$. The machine $\mathcal{D}$ then used the output of $\mathcal{H}$ to determine what its own output should be.

If we generalize this notion—that is, the notion of a Turing machine taking an input, converting it into some other form, and then giving that converted input to another Turing machine— we get a rather interesting technique that we can use to prove all sorts of decision problems are undecidable. This general notion is called a *mapping reduction* or, more generally, just a *reduction*.[1]

We encounter examples of reductions in real life every day, whether we realize it or not. For example, we can reduce the problem of finding a book in the library to the problem of searching for that book in the library's catalog system. If we use the catalog to find the book, then we can take the solution to that problem (the location of the book as listed in the catalog) and apply it to our original problem (finding the location of the book in the stacks). As another example, students can reduce the problem of staying awake in lectures to the problem of acquiring a coffee from the café.

Computationally speaking, a reduction is a process that converts an instance of some problem $X$ to an equivalent instance of some other problem $Y$. Specifically, this conversion is performed by a special kind of function called a *computable function*. A computable function is, as the name suggests, a function that can be computed on a Turing machine.

**Definition 1** (Computable function)**.** A function $f \colon \Sigma^* \to \Sigma^*$ is computable if there exists some Turing machine that, given an input word $w$, halts with $f(w)$ on its input tape and nothing else.

**Example 2.** The function $f(n) = 2n$ is a computable function. We construct a Turing machine $\mathcal{M}_{2n}$ that takes as input a word consisting of $n$ copies of `1` and performs the following steps:

1. Repeat $n$ times:
   (a) Erase the leftmost `1` from the tape.
   (b) Move rightward past the remaining `1`s in the input word, plus one blank cell, plus any existing `1`s in the output word.
   (c) Write a `1` to the rightmost blank cell, then move rightward and write a second `1`.
   (d) Move leftward to the leftmost `1` in the input word.

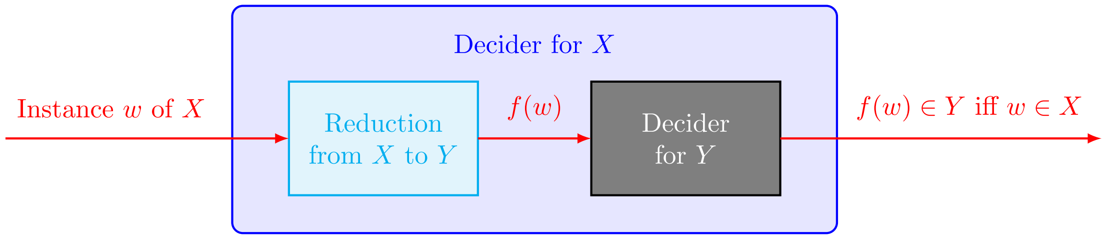
With the notion of a computable function, we can now formally define a mapping reduction.

**Definition 3** (Mapping reduction)**.** Given two decision problems $X$ and $Y$, problem $X$ is mapping reducible to problem $Y$ if there exists a computable function $f \colon \Sigma^* \to \Sigma^*$ where, for all $w \in \Sigma^*$, $w \in X$ if and only if $f(w) \in Y$.

[1] There exist other kinds of reductions, but in this lecture we will only consider mapping reductions, so we will use the word "reduction" as a shorthand to refer to mapping reductions.

In other terms, if $X$ reduces to $Y$, then we can transform every instance $w$ of $X$ to an instance $f(w)$ of $Y$. Since $w \in X$ if and only if $f(w) \in Y$, we know that the transformed instance will produce the same output as the original instance. As a result, we can use a reduction along with a decision algorithm for problem $Y$ to decide the original problem $X$.

Diagrammatically, we can visualize a mapping reduction from $X$ to $Y$ in the following way:



We denote a mapping reduction from $X$ to $Y$ by the notation $X \leq_m Y$. Note that the direction of a reduction is important; if $X \leq_m Y$, then we say that we reduce *from* $X$ *to* $Y$.

If we have a reduction from $X$ to $Y$, then we can make some claims about the relative difficulty of $X$ based on what we know about $Y$, or vice versa. The existence of a reduction from $X$ to $Y$ implies that finding an answer to $X$ is no more difficult than finding an answer to $Y$, or equivalently, finding an answer to $Y$ is at least as difficult as finding an answer to $X$. This is because we must decide $Y$ as an intermediate step toward deciding $X$. Thus,

- if $X$ reduces to $Y$ and $Y$ is "easy", then we know that $X$ must similarly be "easy"; and
- if $X$ reduces to $Y$ and $X$ is "hard", then we know that $Y$ must similarly be "hard".

For now, we write "easy" and "hard" in quotation marks, since these notions are still informal. Soon, we will introduce complexity classes and define more precise notions of easiness and hardness for decision problems.

Focusing on decidability instead of complexity, we can combine the notions of decidable and undecidable problems with reductions to allow us to characterize one unknown problem in terms of another known problem.

**Theorem 4.** *If $Y$ is decidable and $X \leq_m Y$, then $X$ is decidable.*

*Proof.* Since $Y$ is decidable, there exists a Turing machine $\mathcal{M}_Y$ that decides instances of $Y$. Moreover, since $X \leq_m Y$, there exists a computable function $f$ that reduces instances of $X$ to instances of $Y$.

We construct a Turing machine $\mathcal{M}_X$ that takes as input a word $w$ and performs the following steps:

1. Compute $f(w)$.
2. Run $\mathcal{M}_Y$ on input $f(w)$.
3. (a) If $\mathcal{M}_Y$ accepts, then accept.
   (b) If $\mathcal{M}_Y$ rejects, then reject. $\square$

By taking the contrapositive of Theorem 4, we get the following important result that we will use frequently in future proofs.

**Corollary 5.** *If $X$ is undecidable and $X \leq_m Y$, then $Y$ is undecidable.*

Note, however, that if $X$ is decidable and $X \leq_m Y$, then we can't make any conclusions about the decidability of $Y$. It's possible that $Y$ may be undecidable even if $X$ is decidable.

We can make similar claims about semidecidability instead of decidability as well, by using essentially the same proof as in Theorem 4. The difference here, of course, is that we no longer have the guarantee that our Turing machine $\mathcal{M}_Y$ will always halt.

**Theorem 6.** *If $Y$ is semidecidable and $X \leq_m Y$, then $X$ is semidecidable.*

Again, taking the contrapositive gives us another important result that will come in handy later.

**Corollary 7.** *If $X$ is not semidecidable and $X \leq_m Y$, then $Y$ is not semidecidable.*

## 2 Undecidable Problems for Turing Machines (Redux)

From our previous lecture, we know that $A_{\mathsf{TM}}$ is undecidable. Using this fact, we can prove a number of other problems for Turing machines undecidable by using our new notion of a reduction.

### 2.1 Halting Problem

The *halting problem* is perhaps one of the most famous problems in theoretical computer science. Put simply, the halting problem asks whether the computation of a Turing machine halts on some given input word. We can formulate it more precisely as follows:

$$HALT_{\mathsf{TM}} = \{\langle \mathcal{M}, w\rangle \mid \mathcal{M} \text{ is a Turing machine that halts on input } w\}.$$

Note that the formulation of $HALT_{\mathsf{TM}}$ looks very similar to that of $A_{\mathsf{TM}}$. However, there exists a subtle difference between the two problems: $A_{\mathsf{TM}}$ asks not only whether a given Turing machine *halts*, but also whether it *accepts* a given input word. By contrast, $HALT_{\mathsf{TM}}$ only cares about whether the machine halts.

The halting problem has deep connections and implications for many fields of computer science, not least of which is software engineering. For instance, some infinite-looping behaviour is desirable in a piece of software, such as the following simple pseudocode routine that continually polls a hardware input source:

```
while true do
    $r \leftarrow$ CheckSensor($val$)
```

However, programmers typically want to write code that is guaranteed to halt and produce some output, and most general programming languages are Turing-complete. The undecidability of the halting problem would therefore imply that no general procedure exists to determine whether a given arbitrary program halts on a given input.

Note that $HALT_{\mathsf{TM}}$ is at least semidecidable, since we can construct a Turing machine that takes as input some word $w$ and accepts if its computation halts.

**Theorem 8.** *$HALT_{\mathsf{TM}}$ is semidecidable.*

*Proof.* Construct a Turing machine $\mathcal{M}_{\mathrm{HTM}}$ that takes as input $\langle \mathcal{M}, w\rangle$, where $\mathcal{M}$ is a Turing machine and $w$ is a word, and performs the following steps:

1. Simulate $\mathcal{M}$ on input $w$.
2. If $\mathcal{M}$ halts, then accept. □

Now, we will prove the undecidability of $HALT_{\mathsf{TM}}$ by way of reduction from our known-undecidable problem $A_{\mathsf{TM}}$. Note that reducing *from $A_{\mathsf{TM}}$ to $HALT_{\mathsf{TM}}$* means that we can turn instances of $A_{\mathsf{TM}}$ into instances of $HALT_{\mathsf{TM}}$, and since we know that $A_{\mathsf{TM}}$ is undecidable, this must mean that $HALT_{\mathsf{TM}}$ is similarly undecidable by Corollary 5.

**Theorem 9.** *$HALT_{\mathsf{TM}}$ is undecidable.*

*Proof.* Assume by way of contradiction that $HALT_{\mathsf{TM}}$ is decidable, and suppose that $\mathcal{M}_{\mathrm{H}}$ is a Turing machine that decides $HALT_{\mathsf{TM}}$.

We construct a new Turing machine $\mathcal{M}_{\mathrm{A}}$ that decides the membership problem $A_{\mathsf{TM}}$. The machine $\mathcal{M}_{\mathrm{A}}$ takes as input $\langle \mathcal{M}, w\rangle$, where $\mathcal{M}$ is a Turing machine and $w$ is an input word, and performs the following steps:

1. Using the description of $\mathcal{M}$, construct the following Turing machine $\mathcal{M}'$ that takes as input $x$ and performs the following steps:

   $\mathcal{M}'$1. Run $\mathcal{M}$ on $x$.

   $\mathcal{M}'$2. (a) If $\mathcal{M}$ accepts, then accept.

   (b) If $\mathcal{M}$ rejects, then loop forever.

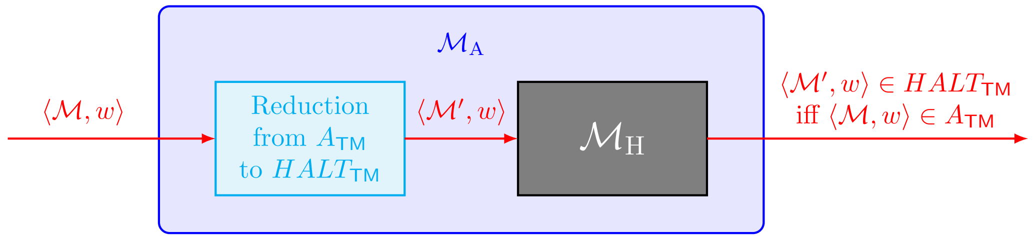
2. Run $\mathcal{M}_{\text{H}}$ on input $\langle \mathcal{M}', w\rangle$.

3. (a) If $\mathcal{M}_{\text{H}}$ accepts, then accept.

   (b) If $\mathcal{M}_{\text{H}}$ rejects, then reject.

Therefore, if such a machine $\mathcal{M}_{\text{H}}$ existed to decide $HALT_{\mathsf{TM}}$, then we could decide $A_{\mathsf{TM}}$ as well. However, we know that $A_{\mathsf{TM}}$ is undecidable. Thus, $\mathcal{M}_{\text{H}}$ must not exist, and so $HALT_{\mathsf{TM}}$ must be undecidable. □

In our proof that $HALT_{\mathsf{TM}}$ is undecidable, we constructed a Turing machine $\mathcal{M}_{\text{A}}$ that ostensibly decides $A_{\mathsf{TM}}$ and is composed of two parts: a reduction that computes a function $f$ turning instances of $A_{\mathsf{TM}}$ into instances of $HALT_{\mathsf{TM}}$, and a black-box Turing machine $\mathcal{M}_{\text{H}}$ that decides $HALT_{\mathsf{TM}}$. As we did before, we can present this construction diagrammatically:


$\mathcal{M}_{\text{A}}$
$\langle \mathcal{M}, w\rangle$
Reduction from $A_{\mathsf{TM}}$ to $HALT_{\mathsf{TM}}$
$\langle \mathcal{M}', w\rangle$
$\mathcal{M}_{\text{H}}$
$\langle \mathcal{M}', w\rangle \in HALT_{\mathsf{TM}}$ iff $\langle \mathcal{M}, w\rangle \in A_{\mathsf{TM}}$

We don't know how the black-box Turing machine $\mathcal{M}_{\text{H}}$ decides $HALT_{\mathsf{TM}}$; we only assume that it exists. However, we *do* know how the reduction works—this is step 1 in our procedure! The function $f$ takes the description of the input Turing machine $\mathcal{M}$ and converts it into a new Turing machine $\mathcal{M}'$ that halts and accepts an input word $x$ only if $\mathcal{M}$ accepts the same input word $x$.

At this point, you may ask yourself: doesn't this reduction implicitly decide $A_{\mathsf{TM}}$, since it has to figure out whether $\mathcal{M}$ accepts or rejects its input word? This is a reasonable question, since if the reduction did work in this way, then we would find ourselves trapped in a snare of circular logic. Fortunately for us, we avoid such a trap, since the reduction does no deciding on its own: it only modifies the description of $\mathcal{M}$. Namely,

- if $\langle \mathcal{M}\rangle$ has a transition leading to $q_{\text{accept}}$, then the reduction leaves this transition as is; and
- if $\langle \mathcal{M}\rangle$ has a transition leading to $q_{\text{reject}}$, then the reduction modifies this transition to instead enter a new "infinite loop" state and render $q_{\text{reject}}$ unreachable.

Thus, the reduction only changes how certain transitions of $\mathcal{M}$ behave, and it doesn't consider the output of $\mathcal{M}$ on any particular input word.

Ultimately, our construction establishes that $\mathcal{M}$ accepts $w$ if and only if $\mathcal{M}'$ halts on $w$ (i.e., $\langle \mathcal{M}, w\rangle \in A_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}', w\rangle \in HALT_{\mathsf{TM}}$). At the same time, $\mathcal{M}$ does not accept $w$ or $\mathcal{M}$ loops forever on $w$ if and only if $\mathcal{M}'$ loops forever on $w$ (i.e., $\langle \mathcal{M}, w\rangle \notin A_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}', w\rangle \notin HALT_{\mathsf{TM}}$).

## 2.2 Emptiness Problem

Let's continue by considering the familiar emptiness problem for Turing machines, $E_{\mathsf{TM}}$.

Since we already know that $A_{\mathsf{TM}}$ is undecidable, we can use this problem in our reduction to $E_{\mathsf{TM}}$. Our goal, again, is to show that if $E_{\mathsf{TM}}$ were decidable, then $A_{\mathsf{TM}}$ would also be decidable; an obvious contradiction.

If we take the usual approach—given a Turing machine that decides $E_{\mathsf{TM}}$, we construct a Turing machine that decides $A_{\mathsf{TM}}$—then a Turing machine's language being empty implies that a given input word is not