**St. Francis Xavier University**
**Department of Computer Science**

**CSCI 356: Theory of Computing**
**Lecture 5: Reducibility**
**Fall 2021**

# 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 $A$ to an equivalent instance of some other problem $B$. 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.

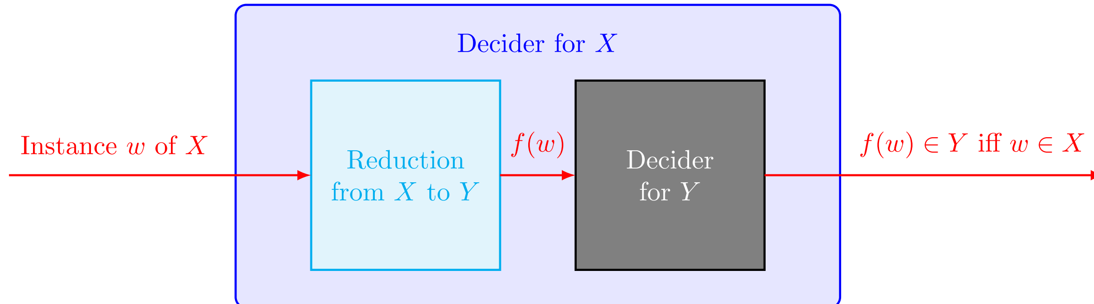
With the notion of a computable function, we can now formally define a mapping reduction.

**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. □

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.*

## 1.1 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.

### 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**
  $r \leftarrow \textsc{CheckSensor}(val)$

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 $\mathcal{M}_{HTM}$ 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. $\square$

Here, we will prove the undecidability of $HALT_{\mathsf{TM}}$ in two ways. First, we will use a less formal "reduction-style" argument similar to the one we used to prove the undecidability of $A_{\mathsf{TM}}$ and other decision problems. Then, we will give a formal mapping reduction from $A_{\mathsf{TM}}$ to $HALT_{\mathsf{TM}}$.

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

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

We construct a new Turing machine $\mathcal{M}_{ATM}$ that decides the membership problem $A_{\mathsf{TM}}$. The machine $\mathcal{M}_{ATM}$ 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. Run $\mathcal{M}_{HD}$ on input $\langle \mathcal{M}, w \rangle$.
2. (a) If $\mathcal{M}_{HD}$ accepts, then simulate $\mathcal{M}$ on $w$ until $\mathcal{M}$ halts.
      i. If $\mathcal{M}$ accepts, then accept.
      ii. If $\mathcal{M}$ rejects, then reject.
   (b) If $\mathcal{M}_{HD}$ rejects, then reject.

Therefore, if such a machine $\mathcal{M}_{HD}$ 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, $\mathcal{M}_{HD}$ must not exist, and so $HALT_{\mathsf{TM}}$ must be undecidable. □

Now, to prove that $HALT_{\mathsf{TM}}$ is undecidable using a mapping reduction from $A_{\mathsf{TM}}$, we must give a computable function $f$ that takes as input $\langle \mathcal{M}, w \rangle$ (that is, an input to $A_{\mathsf{TM}}$) and produces an output $\langle \mathcal{M}', w \rangle$, where $\langle \mathcal{M}', w \rangle \in HALT_{\mathsf{TM}}$ if and only if $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$.

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

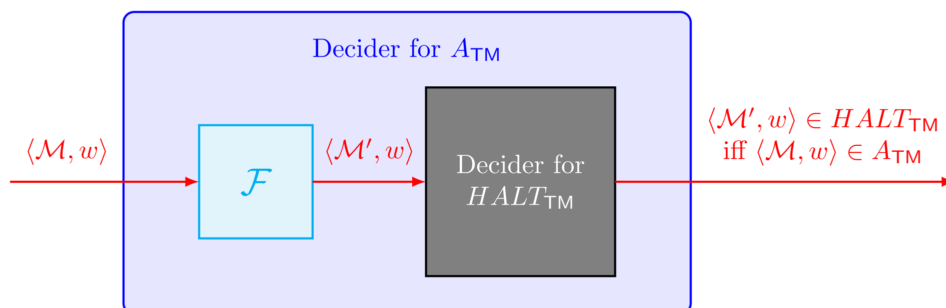
*Proof.* To prove undecidability by way of reduction, we construct a Turing machine $\mathcal{F}$ that takes as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and $w$ is an input word, and computes the following function $f$ that reduces instances of $A_{\mathsf{TM}}$ to instances of $HALT_{\mathsf{TM}}$:

1. 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. Output $\langle \mathcal{M}', w \rangle$.

If $\mathcal{M}$ accepts $w$ (i.e., $\langle \mathcal{M}, w \rangle \in A_{\mathsf{TM}}$), then $\mathcal{M}'$ halts (i.e., $\langle \mathcal{M}', w \rangle \in HALT_{\mathsf{TM}}$). On the other hand, if $\mathcal{M}$ does not accept $w$, then $\mathcal{M}'$ does not halt.

Since we know that $A_{\mathsf{TM}}$ is undecidable, and since $A_{\mathsf{TM}} \leq_m HALT_{\mathsf{TM}}$, we conclude that $HALT_{\mathsf{TM}}$ is also undecidable as a consequence of Corollary 5. □

As we did before, we can present the idea of this proof diagrammatically:



In the end, however, both proofs communicate the same fact: $HALT_{\mathsf{TM}}$ is undecidable.

Observe that our two undecidability proofs are quite similar in that they both construct Turing machines, but the purpose of each Turing machine in each proof sets them apart. In the first proof, we constructed $\mathcal{M}_{ATM}$ to ostensibly decide $A_{\mathsf{TM}}$, which led us to a contradiction. In the second proof, we constructed $\mathcal{F}$ to compute a reduction, where the reduction itself constructed a "sub-Turing machine" $\mathcal{M}'$ in the process of formatting the input to $A_{\mathsf{TM}}$ in terms of an input for $HALT_{\mathsf{TM}}$.

**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 accepted by the Turing machine. But the inverse statement is not necessarily true: if the Turing machine's language is *nonempty*, then we can't conclude that the given word *is* accepted by the machine. The nonempty language could contain words that are not the given word.

Instead of the usual approach, then, we will do the following. We still receive as input $\langle \mathcal{M}, w \rangle$, where $\mathcal{M}$ is a Turing machine and $w$ is an input word, but we will modify the description of $\mathcal{M}$ so that the only word it accepts is $w$. In doing so, we "bake in" $w$ to the description of the Turing machine. We will then give the description of the modified machine $\mathcal{M}'$ to our machine that decides $E_{\mathsf{TM}}$. In this way, testing the emptiness of $L(\mathcal{M}')$ is equivalent to testing whether $\mathcal{M}$ accepts $w$: $L(\mathcal{M}')$ is nonempty if and only if $w \in L(\mathcal{M})$.

**Theorem 10.** *$E_{\mathsf{TM}}$ is undecidable.*

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

We construct a new Turing machine $\mathcal{M}_{ATM}$ that decides the membership problem $A_{\mathsf{TM}}$. The machine $\mathcal{M}_{ATM}$ 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. If $x = w$, then simulate $\mathcal{M}$ on $w$.

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

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

   $\mathcal{M}'$2. If $x \neq w$, then reject.

2. Run $\mathcal{M}_{ETM}$ on input $\langle \mathcal{M}' \rangle$.
3. (a) If $\mathcal{M}_{ETM}$ accepts, then reject.

   (b) If $\mathcal{M}_{ETM}$ rejects, then accept.

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

Now that we know $E_{\mathsf{TM}}$ is undecidable, is the problem at least semidecidable? Surprisingly, no! Remember that $E_{\mathsf{TM}}$ asks whether a given Turing machine $\mathcal{M}$ accepts *no* input words. In order to positively semidecide this property (i.e., get a "yes" answer), we would need to check that every possible input word over the alphabet $\Sigma$ is *not* accepted by $\mathcal{M}$. Since there are infinitely many words over $\Sigma$, we quickly end up in an infinite loop.

On the other hand, the complementary problem $\overline{E_{\mathsf{TM}}}$ is semidecidable, since every Turing machine with a nonempty language must accept at least one input word. Indeed, we can reduce from $A_{\mathsf{TM}}$ to $\overline{E_{\mathsf{TM}}}$, so the procedure for semideciding $\overline{E_{\mathsf{TM}}}$ is similar to that for semideciding $A_{\mathsf{TM}}$. As a consequence, we get the following result.

**Theorem 11.** *$E_{\mathsf{TM}}$ is co-semidecidable.*

*Proof Sketch.* The non-emptiness problem for Turing machines, $\overline{E_{\mathsf{TM}}}$, is semidecidable. As a result, $E_{\mathsf{TM}}$ is co-semidecidable. □

**Universality Problem**

Moving on to the universality problem for Turing machines, $U_{\mathsf{TM}}$, we obtain the same outcome as we had for $E_{\mathsf{TM}}$. Indeed, the proof of undecidability for the universality problem is almost identical to that for the emptiness problem; we just need to swap accepting and rejecting outcomes in the last step of the computation of $\mathcal{M}_{ATM}$, since the universality problem is, in a sense, the "opposite" of the emptiness problem.

**Theorem 12.** *$U_{\mathsf{TM}}$ is undecidable.*

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

We construct a new Turing machine $\mathcal{M}_{ATM}$ that decides the membership problem $A_{\mathsf{TM}}$. The machine $\mathcal{M}_{ATM}$ 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. If $x = w$, then simulate $\mathcal{M}$ on $w$.

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

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

   $\mathcal{M}'$2. If $x \neq w$, then reject.

2. Run $\mathcal{M}_{UTM}$ on input $\langle \mathcal{M}' \rangle$.

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

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

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

Unlike $E_{\mathsf{TM}}$, however, we cannot prove that $U_{\mathsf{TM}}$ is co-semidecidable. In fact, $U_{\mathsf{TM}}$ is neither semidecidable nor co-semidecidable; it lies entirely outside of our language hierarchy! The idea behind the proof involves a reduction from another decision problem about *total machines*, or Turing machines that halt on all inputs:

$$T_{\mathsf{TM}} = \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ is a Turing machine that halts on all input words}\}.$$

By analogy, if $HALT_{\mathsf{TM}}$ is the "halting" version of $A_{\mathsf{TM}}$, then $T_{\mathsf{TM}}$ is the "halting" version of $U_{\mathsf{TM}}$. Observe that, since total machines halt on all inputs, such machines decide their associated languages.

While we won't go through the complete proofs here, we can construct two reductions: $HALT_{\mathsf{TM}} \leq_m T_{\mathsf{TM}}$ and $HALT_{\mathsf{TM}} \leq_m \overline{T_{\mathsf{TM}}}$. As a consequence of the first reduction, and by the fact that mapping reductions are closed under complement, we know that $\overline{HALT_{\mathsf{TM}}} \leq_m \overline{T_{\mathsf{TM}}}$, which implies that $\overline{T_{\mathsf{TM}}}$ is not semidecidable. At the same time, by the second reduction and the same fact, we know that $\overline{HALT_{\mathsf{TM}}} \leq_m T_{\mathsf{TM}}$, which similarly implies that $T_{\mathsf{TM}}$ is not semidecidable.

Using a reduction from the decision problem $T_{\mathsf{TM}}$, we obtain our negative semidecidability results for $U_{\mathsf{TM}}$.

**Theorem 13.** *$U_{\mathsf{TM}}$ is neither semidecidable nor co-semidecidable.*

*Proof Sketch.* We can construct a reduction $T_{\mathsf{TM}} \leq_m U_{\mathsf{TM}}$. By the fact that mapping reductions are closed under complement, we know that $\overline{T_{\mathsf{TM}}} \leq_m \overline{U_{\mathsf{TM}}}$.

Since neither $T_{\mathsf{TM}}$ nor $\overline{T_{\mathsf{TM}}}$ are semidecidable, we have that neither $U_{\mathsf{TM}}$ nor $\overline{U_{\mathsf{TM}}}$ are semidecidable. Saying that $\overline{U_{\mathsf{TM}}}$ is not semidecidable is equivalent to saying that $U_{\mathsf{TM}}$ is not co-semidecidable. □

**Equivalence Problem**

Recall that the equivalence problem for Turing machines asks whether the languages of two Turing machines are equivalent; that is, no word belongs to one language but not the other.

In each of our previous undecidability proofs, we reduced from $A_{\mathsf{TM}}$ to the given problem. We did this mostly because of the fact that $A_{\mathsf{TM}}$ was our "first" undecidable problem, and because it was easy for us to connect the membership problem to other familiar decision problems. For the equivalence problem, on the other hand, we don't need to restrict ourselves to $A_{\mathsf{TM}}$; we can reduce from a different decision problem.

Think back to the definition of the emptiness problem for Turing machines: if $\langle \mathcal{M} \rangle \in E_{\mathsf{TM}}$, then $L(\mathcal{M}) = \emptyset$. The emptiness problem is just the equivalence problem in disguise, where one of the languages is the empty language! Therefore, if we fix one of the Turing machines to accept no words, we can reduce the problem of testing emptiness to the problem of testing equivalence, and we can in turn obtain our undecidability result.

**Theorem 14.** *$EQ_{\mathsf{TM}}$ is undecidable.*

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

We construct a new Turing machine $\mathcal{M}_{ETM}$ that decides the emptiness problem $E_{\mathsf{TM}}$. The machine $\mathcal{M}_{ETM}$ takes as input $\langle \mathcal{M} \rangle$, where $\mathcal{M}$ is a Turing machine, and performs the following steps:

1. Run $\mathcal{M}_{EQTM}$ on input $\langle \mathcal{M}, \mathcal{M}_{\emptyset} \rangle$, where $\mathcal{M}_{\emptyset}$ is a Turing machine that accepts no input words.
2. (a) If $\mathcal{M}_{EQTM}$ accepts, then accept.
   (b) If $\mathcal{M}_{EQTM}$ rejects, then reject.

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

Similar to the universality problem, the equivalence problem for Turing machines is neither semidecidable nor co-semidecidable, meaning yet another decision problem lies entirely outside of our language hierarchy. To prove this result, we reduce from $U_{\mathsf{TM}}$ instead of $E_{\mathsf{TM}}$ as we did in our previous proof. We are able to construct this reduction since the universality problem asks whether $L(\mathcal{M}) = \Sigma^*$ for some Turing machine $\mathcal{M}$, meaning that it too is just the equivalence problem in disguise. Since we know that $U_{\mathsf{TM}}$ is neither semidecidable nor co-semidecidable, the same must be true for $EQ_{\mathsf{TM}}$.

**Theorem 15.** *$EQ_{\mathsf{TM}}$ is neither semidecidable nor co-semidecidable.*

*Proof Sketch.* We can construct a reduction $U_{\mathsf{TM}} \leq_m EQ_{\mathsf{TM}}$. Since $U_{\mathsf{TM}}$ is neither semidecidable nor co-semidecidable by Theorem 13, we get the same result for $EQ_{\mathsf{TM}}$. □

## 1.2 Undecidable Problems for Context-Free Languages (Redux)

Recall that, previously, we noted both $U_{\textsf{CFG}}$ and $EQ_{\textsf{CFG}}$ were undecidable. However, we didn't prove either of those claims, mainly because we didn't have the tools to do so back then. Here, let's wrap up our work by presenting both of these proofs. Before we do so, however, we require one notion relating to the computation of a Turing machine.

Recall that the *configuration* of a Turing machine is a representation of the current state, tape contents, and input head position of the Turing machine at some point in the computation. In essence, a configuration is a "snapshot" of the Turing machine mid-computation. Depending on the current state of the machine, a configuration may be a *start configuration* (if the current state is $q_0$), an *accepting configuration* (if the current state is $q_{\text{accept}}$), or a *rejecting configuration* (if the current state is $q_{\text{reject}}$).

If we consider the entire sequence of configurations of a Turing machine from start to finish—that is, from the start configuration to either an accepting or rejecting configuration—we get a complete picture of the Turing machine's computation. This sequence of configurations is known as a *computation history*.

**Definition 16** (Computation history)**.** Given a Turing machine $\mathcal{M}$ and an input word $w$, a computation history for $\mathcal{M}$ on $w$ is a sequence of configurations $C_1, C_2, \ldots, C_m$, where $C_1$ is the start configuration of $\mathcal{M}$ on $w$, $C_m$ is either an accepting configuration or a rejecting configuration, and each configuration $C_i$ yields the following configuration $C_{i+1}$.

- If $C_m$ is an accepting configuration, then the sequence forms an accepting computation history.
- If $C_m$ is a rejecting configuration, then the sequence forms a rejecting computation history.

Note that a computation history for a Turing machine $\mathcal{M}$ on an input word $w$ only exists when $\mathcal{M}$ halts on $w$. As a result, the sequence of configurations $C_1, C_2, \ldots, C_m$ always has a finite number of elements. Deterministic computations always have exactly one computation history per input word, while nondeterministic computations may have multiple computation histories per input word.

Why do we require this notion of computation histories in order to prove that our remaining context-free problems are undecidable? As it turns out, we can apply the idea of reductions to the computation histories of Turing machines in order to establish undecidability results. We will use these *reductions via computation histories* to prove the undecidability of one of our context-free decision problems.

### Universality Problem

To prove the undecidability of $U_{\textsf{CFG}}$, we will construct a reduction from $A_{\textsf{TM}}$ to $U_{\textsf{CFG}}$ that makes use of computation histories. Our reduction will require us to make two slight changes: first, as part of the reduction, we will be constructing a context-free model of computation rather than a Turing machine; and second, we must format the computation history in a way that allows us to process it correctly. We will see how both of these changes are handled as we work through the proof.

**Theorem 17.** *$U_{\textsf{CFG}}$ is undecidable.*

*Proof.* Assume by way of contradiction that $U_{\textsf{CFG}}$ is decidable. We will show how to use the decision algorithm for $U_{\textsf{CFG}}$ to decide $A_{\textsf{TM}}$.

Given a Turing machine $\mathcal{M}$ and an input word $w$, we construct a context-free grammar $G$ that generates all words if and only if $\mathcal{M}$ does not accept its input word $w$. Specifically, we construct $G$ in such a way that the words it generates correspond to non-accepting computation histories of $\mathcal{M}$ on $w$. In this way, $\mathcal{M}$ accepts $w$ if and only if $G$ does not generate the accepting computation history of $\mathcal{M}$ on $w$, meaning that the language of $G$ is not universal.

We will assume that the computation history of $\mathcal{M}$ is written in the form $\texttt{\#}C_1\texttt{\#}C_2^{\text{R}}\texttt{\#}C_3\texttt{\#}C_4^{\text{R}}\texttt{\#}\ldots\texttt{\#}C_m\texttt{\#}$, where $C_i$ is the $i$th configuration of $\mathcal{M}$ and # is a special boundary marker. Note that every even-numbered configuration is written in reverse; this is the aforementioned "format" change we require in order to process the computation history correctly.

Working from Definition 16, we can deduce that a computation history is non-accepting if one of the following conditions is met:

1. The computation history does not start with the start configuration as $C_1$;
2. The computation history does not end with an accepting configuration as $C_m$; or
3. Some configuration $C_i$ does not yield the following configuration $C_{i+1}$ according to the transition function of $\mathcal{M}$.

To construct the context-free grammar $G$, we will construct a pushdown automaton $\mathcal{A}$ and use our conversion process to turn it into a grammar. The pushdown automaton $\mathcal{A}$ checks each of the three non-accepting conditions, and it does so by nondeterministically guessing which condition it checks.

- In one nondeterministic branch, $\mathcal{A}$ checks the first condition by reading the beginning of its input word and accepting if the segment between the first two boundary markers, $C_1$, is not the start configuration.
- In another nondeterministic branch, $\mathcal{A}$ checks the second condition by reading the end of its input word and accepting if the segment between the last two boundary markers, $C_m$, is not an accepting configuration.
- In the last nondeterministic branch, $\mathcal{A}$ checks the third condition by scanning the input word until it nondeterministically selects a configuration $C_i$. $\mathcal{A}$ then pushes the symbols of $C_i$ to its stack until it reads a boundary symbol, and pops each symbol of $C_i$ as it reads the corresponding symbol of $C_{i+1}$ from the input word. (We can match corresponding symbols in the correct order as a result of the "format" change from earlier.) If there is any difference between symbols that was not produced by the transition function of $\mathcal{M}$, $\mathcal{A}$ accepts.

Clearly, every word accepted by the pushdown automaton $\mathcal{A}$ corresponds to a non-accepting computation history of $\mathcal{M}$, and so the language of the context-free grammar $G$ consists of the same non-accepting computation histories.

Therefore, if it were possible to construct such a grammar $G$ to decide $U_{\mathsf{CFG}}$, then we could decide $A_{\mathsf{TM}}$ as well. However, we know that $A_{\mathsf{TM}}$ is undecidable. Thus, $G$ must not exist, and so $U_{\mathsf{CFG}}$ must be undecidable. □

### Equivalence Problem

Finally, we come to the equivalence problem for context-free grammars, $EQ_{\mathsf{CFG}}$. As we did with $EQ_{\mathsf{TM}}$, we will use the observation that another undecidable problem for context-free grammars—namely, $U_{\mathsf{CFG}}$—is really the equivalence problem in disguise, and we will reduce from that problem to $EQ_{\mathsf{CFG}}$.

**Theorem 18.** *$EQ_{\mathsf{CFG}}$ is undecidable.*

*Proof.* Assume by way of contradiction that $EQ_{\mathsf{CFG}}$ is decidable, and suppose that $\mathcal{M}_{EQCFG}$ is a Turing machine that decides $EQ_{\mathsf{CFG}}$.

We construct a new Turing machine $\mathcal{M}_{UCFG}$ that decides the universality problem $U_{\mathsf{CFG}}$. The machine $\mathcal{M}_{UCFG}$ takes as input $\langle G \rangle$, where $G$ is a context-free grammar, and performs the following steps:

1. Run $\mathcal{M}_{EQCFG}$ on input $\langle G, H \rangle$, where $H$ is a context-free grammar such that $L(H) = \Sigma^*$.
2. (a) If $\mathcal{M}_{EQCFG}$ accepts, then accept.
   (b) If $\mathcal{M}_{EQCFG}$ rejects, then reject.

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