CMPU 240 · Spring 2026

Reductions and Undecidability

Writing proofs by reduction

Reduction proofs are quite short, and the proofs you'll write for homework (and exam) problems will be very similar to the examples in the lecture slides. However, the form of abstraction reductions involve will be unfamiliar to most students, so you may find these problems challenging. This handout provides some additional help and more examples to consider.

Is my language decidable or not?

Some problems ask you to determine whether some language L is decidable or not (and perhaps prove that your answer is correct).

Rice's theorem[1] states that any non-trivial property of Turing machines *that depends only on the language that the Turing machine recognizes* is undecidable. For example, it's undecidable whether a Turing machine has a language that's non-empty, contains 17 elements, contains the string `Vassar`, is infinite, and so forth.

On the other hand, facts about the Turing machine's *structure* are decidable. For example, you can tell if the Turing machine has more than 15 states, has no transitions into its accept state, or the like. Such properties just require parsing the encoding of the Turing machine and doing some straightforward check (e.g., counting the number of states listed).

Facts about a Turing machine's *behavior* may or may not be decidable, and it's not always easy to tell which. For example, it is decidable whether a Turing machine, run on blank input, ever moves left, but it is not decidable whether the same Turing machine ever moves left three times in a row.[2]

[1] Sipser solved exercise 5.28

[2] Why? Ask yourself whether each property is essential for a Turing machine to have its full computational ability or not.

Basic template

To prove that a language L is undecidable, you will normally want to use a reduction from a language already proven to be undecidable (e.g., A_{TM} or $HALT_{\mathrm{TM}}$). A reduction proof will look something like:

By contradiction; assume $L \in \mathbf{R}$. Then there's a decider D for L. We will now construct a Turing machine S that decides A_{TM}:

S = "On input $\langle M, w\rangle$, where M is the code for a Turing machine and w is a string,

> [Pseudocode explaining how S decides whether M accepts w, using D as a subroutine.]

[Explain why S decides A_{TM}.] We have reached a contradiction because we know that A_{TM} is undecidable. Thus our assumption was wrong and L is undecidable. ■

Sipser uses R rather than D, but this makes me think "recognizer" when it should be a decider.

This is called "reducing A_{TM} to L".

Sometimes you can write a shorter or simpler proof by replacing A_{TM} with some other problem that's known to be undecidable. If so, remember that you may need to change the inputs to S. For instance, elements of the language EQ_{TM} are pairs of Turing machines, so a reduction from EQ_{TM} requires constructing a Turing machine S whose inputs are pairs $\langle M_1, M_2\rangle$ of two Turing machines.[3]

If you aren't sure what problem to reduce to L, use A_{TM}.

3 For simple examples of the basic idea, see Sipser's proofs of the undecidability of $HALT_{\mathrm{TM}}$ (p. 217) and EQ_{TM} (p. 220).

Turing machine language problems

In the simple examples above, S fed its inputs M and w more or less directly into the subroutine D. Many other problems require S to rewrite the code for M before feeding it to D.

It is often easier to imagine your Turing machines as computer programs written, e.g., in Python or C. That is, S reads the source code for program M, and it writes to disk a new file of source code for a program M'.

Often, you can imagine S doing the rewrite by simple copy-and-paste, e.g., copying the input code for M and then adding a new `main()` function and perhaps a few additional declarations.

The new Turing machine typically has the input w "hardcoded" into it. If you were writing the new machine's code in Python or C, you would write a local variable declaration and copy the input value of w into that declaration.

E.g., if the input value were the string `Quokkas␣are␣cute`, the code for the new machine would contain a line like this:

```
w = "Quokkas are cute"
```

When we hardcode the input w, we will often use M_w as the name for the new Turing machine to make it easy to remember.

The Turing machine M_w is typically designed so it accepts one set of strings (call the set X) if M accepts w, and it accepts a totally different set of strings (call the set Y) if M doesn't accept w. One of these sets should have the property that D tests for, and the other should not.[4] In other words, M_w functions as an *adapter*: it converts the property we want to detect – whether M accepts w – into the property that D knows how to test for. Building this adapter is the substantive step in any such reduction; once M_w has been constructed, running D on $\langle M_w \rangle$ and returning its answer is mechanical.

The code for M_w typically involves simulating M on w, plus some other tests on x that can clearly be computed in a finite amount of time.

For some reductions, we have M_w test x first, e.g.,

1 Input is a string x.

2 If x has some easily tested property φ, *accept*.

3 Otherwise, simulate M on w. If M accepts, *accept*; if M rejects, *reject*.

For either of these two steps, you could also take the opposite action, e.g., reject all strings x that have property φ.[5]

For other reductions, we have M_w simulate M on w first, before we test x, e.g.,[6]

1 Simulate M on w.

2 If M rejects, *reject*.

3 Otherwise, *accept* exactly when x has some easily tested property φ.

4 E.g., see the reductions for L_{VASSAR} and $HALTEMPTY_{\mathrm{TM}}$ in the additional examples section below, where M_w accepts all input strings x if M accepts w, and it accepts no input string x if M doesn't accept w.

5 For examples of this pattern, see Sipser's reductions for E_{TM} (p. 217) and $REGULAR_{\mathrm{TM}}$ (p. 219).

6 Examples of this pattern include L_3 (in the additional examples section), and the proof of Rice's theorem (see Sipser's solved exercise 5.28 on p. 213).

Common problems writing reductions

A common problem is to get confused about which values are input to which Turing machines. In the above proofs, the string w is an input to the machine S, i.e., the machine that decides A_{TM}. The string x is an input to the Turing machine M_w. Look back through the proofs and check carefully where x occurs and where w occurs.

The input to the Turing machine D is usually the code for the machine M_w. You can imagine D as a Python or C program, which reads a plain-text file of source code for some other program. Note that w and x are *not* inputs to D. The value of w is hardcoded into the code for M_w. The value of x is not specified anywhere: it is just the placeholder name that the description of M_w uses for "whatever input M_w might one day be run on".

Since the value of x is not specified, D has to consider all possible values for the input x and figure out what M_w would do on each of these values. If D is testing a non-trivial property of M_w, you can imagine that it might be hard to figure out what it will do on all possible input strings x. Remember that D is only a hypothetical Turing machine whose existence will be contradicted at the end of the proof. That is, D can't really exist, because it's trying to do a task that's too hard.

Proving that a behavior is undecidable

Undecidable behavior properties normally restrict the Turing machine's behavior in cosmetic ways but still allow it to complete the standard range of Turing machine tasks. Such a problem involves a language L which is a set of all Turing machines with a specific behavior B. For example,

$$L = \{\langle M\rangle \mid M \text{ prints three 1s in a row on blank input}\}$$

or

$$L = \{\langle M\rangle \mid M \text{ has a state which is never entered on any input string}\}.$$

For this class of problems, reductions typically require modifying the input Turing machine code $\langle M\rangle$ to produce a new Turing machine M_w which does pretty much the same thing as M on w, except that

1. When M would have halted, M_w does behavior B then halts, and
2. M_w never accidentally does behavior B while it's computing.

Typical examples of this type of reduction are the proofs for L_{111} and L_x in the additional examples section. In the case of L_{111}, we ensure that M_w can never accidentally print three `1`s in a row by replacing every instance of `1` in its code by a new character `1′`.

Proving that a behavior is decidable

Decidable behavior properties normally restrict the Turing machine's behavior in a way that significantly restricts its functionality. For an example of a decidable Turing machine behavior, consider the language

$$L = \{\langle M\rangle \mid M \text{ never moves left on input } \texttt{VASSAR}\}.$$

L is decidable because Turing machines that never move left are so constrained that they either halt very quickly, or not at all.

More precisely, if a Turing machine M never moves left, it reads through the whole input, then starts looking at blank tape cells. Once it is on the blank part of the tape, it can cycle through its set of states, but after $|Q|$ moves, it has run out of distinct configurations and must be in a loop. So, if you watch M for six moves (the length of the string `VASSAR`) and then for an additional $|Q| + 1$ moves, it has either halted or it is in an infinite loop.

Therefore, to decide L, you run the input Turing machine for $|Q| + 7$ moves. After that many moves, it has either

- moved left (in which case you *reject*)
- halted (in which case you *accept*), or
- gone into an infinite loop without ever moving left (in which case you *accept*).

This algorithm is a decider (not just a recognizer) for L, because it definitely halts on any input Turing machine M.

Example: L_{VASSAR}

THEOREM The language

$$L_{VASSAR} = \{\langle M \rangle \mid L(M) \text{ contains the string } \texttt{VASSAR}\}$$

is undecidable.

PROOF We can prove this language is undecidable by reduction from A_{TM}. Suppose that L_{VASSAR} were decidable and let D be a Turing machine deciding it. We use D to construct a Turing machine S that decides A_{TM}:

> S = "On input $\langle M, w \rangle$, where M is a Turing machine and w is a string,

- Construct $\langle M_w \rangle$, where M_w = 'On input x,
  - Simulate M on w.
  - If M accepts, *accept*; if M rejects, *reject*.'
- Run D on $\langle M_w \rangle$.
- If D accepts, *accept*; if D rejects, *reject*."

If M accepts w, the language of M_w contains all strings and, thus, the string VASSAR. If M doesn't accept w, the language of M_w is the empty set and, thus, doesn't contain the string VASSAR. So $D(\langle M_w \rangle)$ accepts exactly when M accepts w. Thus, S decides A_{TM}.

But we know that A_{TM} is undecidable, so S can't exist. Therefore we have a contradiction, and L_{VASSAR} must have been undecidable.
■

Example: $HALTEMPTY_{\mathrm{TM}}$

THEOREM The language

$$HALTEMPTY_{\mathrm{TM}} = \{\langle M \rangle \mid M \text{ halts on } \varepsilon\}$$

is undecidable.

PROOF We can prove this language is undecidable by reduction from $HALT_{\mathrm{TM}}$. Suppose that $HALTEMPTY_{\mathrm{TM}}$ were decidable and let D be a Turing machine deciding it. We use D to construct a Turing machine S deciding $HALT_{\mathrm{TM}}$:

> S = "On input $\langle M, w \rangle$, where M is a Turing machine and w is a string,
>
> - Construct $\langle M_w \rangle$, where M_w is a Turing machine that writes w on the empty tape and then runs M.
> - Run D on $\langle M_w \rangle$.
> - If D accepts, *accept*. If D rejects, *reject*."

D accepts $\langle M_w \rangle$ if and only if M_w halts on ε, and M_w halts on ε if and only if M halts on w. Thus, S decides $HALT_{\mathrm{TM}}$.

But we know that $HALT_{\mathrm{TM}}$ is undecidable, so S can't exist. Therefore we have a contradiction, and $HALTEMPTY_{\mathrm{TM}}$ must have been undecidable. ■

Example: L_{111}

THEOREM The language

$$L_{111} = \{\langle M\rangle \mid M \text{ prints three } \texttt{1}\text{s in a row on blank input}\}$$

is undecidable.

PROOF We can prove this language is undecidable by reduction from A_{TM}. Suppose that L_{111} were decidable. Let D be a Turing machine deciding L_{111}. We will now construct a Turing machine S that decides A_{TM}:

S = "On input $\langle M, w\rangle$, where M is a Turing machine and w is a string,

- Construct $\langle M_w\rangle$, where M_w = 'On input x:
  - Ignore x.
  - Simulate M on w, but substituting some previously unused symbol $\texttt{1}'$ for $\texttt{1}$, everywhere that $\texttt{1}$ occurs in w or the code for M. (We can also choose the simulator's own working alphabet so that it never writes $\texttt{1}$ as bookkeeping.)
  - If M rejects w, *reject*.
  - If M accepts w, print $\texttt{111}$ on the tape and then *accept*.'
- Run D on $\langle M_w\rangle$.
- If D accepts, *accept*. If D rejects, *reject*."

If M accepts w, then M_w will print $\texttt{111}$ on any input (and thus on a blank input). If M does not accept w, then M_w is guaranteed never to print $\texttt{111}$ accidentally, since every $\texttt{1}$ that appeared in M or w has been replaced by $\texttt{1}'$. So D will accept $\langle M_w\rangle$ exactly when M accepts w. Therefore, S decides A_{TM}.

But we know that A_{TM} is undecidable, so S can't exist. Therefore we have a contradiction, and L_{111} must have been undecidable. ■

Example: L_x

THEOREM The language

$$L_x = \{\langle M\rangle \mid M \text{ writes an } \mathtt{x} \text{ at some point, when started on blank input}\}$$

is undecidable.

PROOF Suppose that L_x were decidable. Let D be a Turing machine deciding L_x. We will now construct a Turing machine S that decides A_{TM}:

> S = "On input $\langle M, w\rangle$, where M is a Turing machine and w is a string,

- Construct $\langle M_w\rangle$, where M_w = 'On input y:
  - Ignore y.
  - Simulate M on w, but substituting some previously unused symbol ψ for x, everywhere that x occurs in w or the code for M. (We can also choose the simulator's own working alphabet so that it never writes x as bookkeeping.)
  - If M rejects w, *reject*.
  - If M accepts w, print x on the tape and then *accept*.'
- Run D on $\langle M_w\rangle$. If D accepts, *accept*. If D rejects, *reject*."

If M accepts w, then M_w will print x on any input (and thus on a blank input). If M rejects w or loops on w, then M_w is guaranteed never to print x accidentally. So D will accept $\langle M_w\rangle$ exactly when M accepts w. Therefore, S decides A_{TM}.

But we know that A_{TM} is undecidable, so S can't exist. Therefore we have a contradiction, and L_x must have been undecidable. ■

Example: L_3

THEOREM The language

$$L_3 = \{\langle M \rangle \mid |L(M)| = 3\},$$

that is, the language that contains all Turing machines whose languages contain exactly three strings, is undecidable.

PROOF Proof by reduction from A_{TM}. Suppose that L_3 were decidable and let D be a Turing machine deciding it. We use D to construct a Turing machine S deciding A_{TM}:

S = "On input $\langle M, w \rangle$, where M is a Turing machine and w is a string,

- Construct $\langle M_w \rangle$, where M_w = 'On input x,
  - Simulate M on w.
  - If M rejects w, *reject*.
  - Otherwise, *accept* x exactly when x is one of the three strings `Vassar`, `College`, or `Poughkeepsie`.'
- Run D on $\langle M_w \rangle$.
- If D accepts, *accept*. If D rejects, *reject*."

If M accepts w, the language of M_w contains exactly three strings. If M doesn't accept w, the language of M_w is the empty set. So $D(\langle M_w \rangle)$ accepts exactly when M accepts w. Thus, S decides A_{TM}.

But we know that A_{TM} is undecidable, so S can't exist. Therefore we have a contradiction, and L_3 must have been undecidable. ■

Acknowledgments

This handout is based on material written by Nancy Ide.