CISC 5200: Computer Language Theory

Chapter 4
Decidability

Arthur G. Werschulz

Fordham University
Department of Computer and Information Sciences

Spring, 2020

Limitations of algorithmic solvability

- In this chapter, we investigate the problem-solving power of algorithms.
  Some problems can be solved algorithmically; some *cannot*.
- Why study unsolvability?
  - If a particular problem is unsolvable:
    - Searching for an algorithm is futile.
    - Maybe a weaker version of the problem *is* solvable.
  - Gain a perspective on the limits of computation.
  - First step in studying *complexity* (inherent computational difficulty) of a problem:
    1. Is it algorithmically solvable?
    2. Is it *efficiently* algorithmically solvable?

Section 4.1: Decidable Languages

Decidable languages

- We start with problems that are *decidable*.
- We look at some problems involving
  1. regular languages, and
  2. context-free languages.

Decidable problems for regular languages

- We give algorithms for testing ...
  - whether a FA accepts a string (A_{DFA}),
  - whether an NFA accepts a string (A_{NFA}),
  - whether a regular expression accepts a string (A_{REX}),
  - whether the language of a FA is empty (E_{DFA}),
  - whether two FA are equivalent (EQ_{DFA}).
- Represent these problems by languages, rather than FA.
  - Example: $A_{DFA} = \{\langle B, w\rangle : B$ is a DFA that accepts string $w\}$.
  - The problem of testing whether a DFA B accepts an input w is the same as testing whether $\langle B, w\rangle \in A_{DFA}$.
  - Showing that the language A_{DFA} is decidable is the same thing as showing that the computational problem is decidable.

Does a DFA accept a given string?

Theorem

The language

$$A_{DFA} = \{\langle B, w\rangle : B \textit{ is a DFA that accepts string } w\}$$

is decidable.

Proof idea.

Present a TM M that decides A_{DFA}.

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, then ACCEPT, else REJECT."

Does a DFA accept a given string? (cont'd)

Proof idea (cont'd).

Background for actual proof:

- Book's description too simple—leads to wrong understanding.
- The TM M cannot "be" the DFA B.
  If it could, then things would be simple. Both would have essentially the same transition functions (TM just needs to move right over w as each symbol is read.)
- For DFA B, must take string $\langle B\rangle$ (encoding B's five components) as input, and then simulate B on string w.
  - This means the algorithm for simulating any DFA must be embodied in the TM 's state transitions.
  - Think about this. Given a current state and input symbol, scan the tape for the encoded transition function, and then use that info to determine new state.

Does a DFA accept a given string? (cont'd)

Proof idea (cont'd).

- Note that the TM must be able to simulate *any* such DFA, and not just this particular DFA.
- Keep track of current state and position in w by writing on the tape.
- Must describe how a TM simulates a DFA.
- Initially, current state is q_0 and current position is leftmost symbol in w.
- The states and position are updated using the transition function δ.
  The TM M's δ is *not* the same as the DFA B's δ.
- When M finishes processing, ACCEPT if in an accept state and REJECT otherwise.
- From the details of the implementation, it is clear that the processing always finishes in a finite number of steps. □

Does an NFA accept a given string?

Theorem

The language

$$A_{\mathrm{NFA}} = \{\langle B, w\rangle : B \text{ is a NFA that accepts string } w\}$$

is decidable.

Proof idea.

Since we have proven that DFAs are decidable, only need to convert the NFA to a DFA.

N = "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the subset construction of Theorem 1.39.
2. Run TM M on input $\langle C, w\rangle$, using the theorem we proved earlier.
3. If M accepts, then ACCEPT, else REJECT." □

Does a regular expression generate a given string?

Theorem

The language

$$A_{\mathrm{REX}} = \{\langle R, w\rangle : R \text{ is a regular expression generating string } w\}$$

is decidable.

Proof.

Let P = "On input $\langle R, w\rangle$, where R is a reg exp and w a string:

1. Convert R to an equivalent NFA A as in Chapter 1.
2. Run TM N (from proof in previous slide) on the input $\langle A, w\rangle$.
3. If N accepts, then ACCEPT, otherwise REJECT. " □

Does a DFA accepts *any* string at all?

Theorem

The language

$$E_{\mathrm{DFA}} = \{\langle A\rangle : A \text{ is a DFA and } L(A) = \emptyset\}$$

is decidable.

Proof.

- **Proof idea:** A DFA accepts some string iff it is possible to reach the accept state from the start state. How can we check this?
- We can use a marking algorithm similar to the one used in Chapter 3.

□

Does a DFA accepts *any* string at all? (cont'd)

Theorem

The language E_{DFA} *is decidable.*

Proof.

- Marking algorithm:
- T = "On input $\langle A\rangle$, where A is a DFA:
  1. Mark the start state of A.
  2. Repeat until no new states get marked:
     Mark any state that has a transition coming into it from any previously-marked state.
  3. If no accept state is marked, then ACCEPT; otherwise, REJECT." □

This proof is clearer than most of the previous ones, since its pseudocode is detailed enough to easily allow implementation.

Are two DFAs equivalent?

Theorem

The language

$$\mathrm{EQ}_{\mathrm{DFA}} = \{\langle A, B\rangle : A, B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable.

Proof.

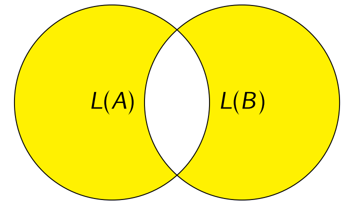
Construct a new DFA C from A and B, where C accepts only those strings accepted by either A or B, but not both.
If A and B accept the same language, then C will accept nothing; use previous construction to check the latter.

F = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C (more on this in a minute . . .).
2. Run TM T (from proof on last side) on input $\langle C\rangle$.
3. If T accepts, then ACCEPT; if T rejects, then REJECT." □

Are two DFAs equivalent? (cont'd)

$$L(C) = L(A) \triangle L(B) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(L(B) \cap \overline{L(A)}\right).$$



- We used proofs by construction to show that regular languages are closed under union, intersection, and complement.
- We can use those constructions to construct a FA that accepts $L(C)$.

Does a given CFG generate a given string?

Theorem

The language

$$A_{\mathrm{CFG}} = \{\langle G, w\rangle : G \text{ is a CFG and } w \in L(G)\}$$

is decidable.

Proof.

For a CFG G and a string w, we want to determine whether G generates w.

- One idea: Use G to go through all derivations.
- This won't work: it yields a TM that's a recognizer, not a decider. That is, it could infinite loop on rules such as $A \to xA$.
- Helpful fact: If G is in Chomsky normal form, then a string of length n will have a derivation using $2n - 1$ steps.

Interlude: Chomsky normal form implies compact derivations

Notation: $X \overset{*}{\Rightarrow} w$ means that $w \in \Sigma^*$ has a leftmost derivation from $X \in V$.

Lemma

Let $G = (V, \Sigma, R, S)$ be a CFG in Chomsky normal form. Let $w \in \Sigma^*$ be nonempty and let $X \in V$. Then any leftmost derivation $X \overset{*}{\Rightarrow} w$ has exactly $2|w| - 1$ steps.

Proof.

By induction on $|w|$.
Basis step: Let $|w| = 1$. Then $w = c$ for some $c \in \Sigma$. Since G is in Chomsky normal form, the only possible derivation $X \overset{*}{\Rightarrow} c$ must be $X \to c$, which has 1 step.

Interlude: Chomsky normal form implies compact derivations (cont'd)

Proof (cont'd).

Induction step: Let $k > 1$ and suppose true for all w with $|w| < k$; we must show true when $|w| = k$. Let $|w| = k$. First step in derivation $X \overset{*}{\Rightarrow} w$ must be $X \to AB$ for some $A, B \in V$. There exist nonempty $x, y \in \Sigma^*$ such that $A \overset{*}{\Rightarrow} x$ and $B \overset{*}{\Rightarrow} y$ within the derivation $X \overset{*}{\Rightarrow} w$. Note that $w = xy$, and so $|x| + |y| = k$. Since $|x| > 0$ and $|y| > 0$, we have $0 < |x| < |x| + |y| = k$ and so $A \overset{*}{\Rightarrow} x$ in $2|x| - 1$ steps. Similarly, $B \overset{*}{\Rightarrow} y$ in $2|y| - 1$ steps. Since $X \overset{*}{\Rightarrow} w$ involves the step $X \to AB$ along with the derivations $A \overset{*}{\Rightarrow} x$ and $B \overset{*}{\Rightarrow} y$, we do $X \overset{*}{\Rightarrow} w$ in

$$1 + (2|x| - 1) + 2(|y| - 1) = 2(|x| + |y|) + 1 - 2 = 2k - 1$$

steps. □

Does a given CFG generate a given string? (cont'd)

Theorem

The language $\mathrm{A}_{\mathrm{CFG}}$ *is decidable.*

Proof (cont'd).

- For a CFG G and a string w, we want to determine whether G generates w.
  1. Convert G to Chomsky normal form.
  2. Let $n = |w|$.
  3. List all derivations using $2n - 1$ steps. If any generates w, then ACCEPT, else REJECT.
- Very inefficient!
  - The number of k-step derivations is exponential in k.
  - So this algorithm runs in time exponential in k. □

Can we do better? Yes! Using dynamic programming, can run in time $O(n^3)$, see Chapter 7.

Does a CFG generate any string at all?

Theorem

The language

$$\mathrm{E}_{\mathrm{CFG}} = \{\langle G \rangle : G \textit{ is a CFG and } L(G) = \emptyset\}$$

is decidable.

How can you do this?

- Brute force approach? Try all possible strings.
  - Won't work; the number of strings unbounded.
  - This would yield a TM that's a *recognizer*, rather than a *decider*.
- Instead, think of this as a graph problem, where you want to know whether you can reach a string of terminals from the start state.
  - Do you think it is easier to work forwards or backwards?
  - Backwards:
    Want to know whether start variable can generate a string of terminals.

Does a CFG generate any string at all? (cont'd)

Theorem

The language $\mathrm{E}_{\mathrm{CFG}}$ *is decidable.*

Proof.

- Mark all the terminal symbols.
- Keep working backwards so that if the RHS of any rule has only marked items, then mark the LHS.
  For example, if $X \to YZ$ with Y and Z being marked, then mark X.
- Do previous step until nothing new is marked.
- If you have marked S, then done and REJECT; otherwise, ACCEPT. □

$\mathrm{EQ}_{\mathrm{CFG}}$ is not a decidable language

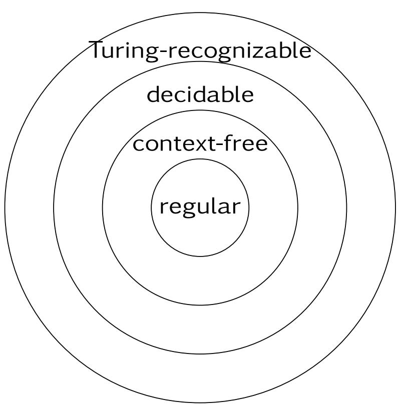
- Earlier, we showed that $\mathrm{EQ}_{\mathrm{DFA}}$ is decidable.
- This proof used the fact that DFAs are closed under union, intersection, and complementation.
- We can't adapt this proof to show that $\mathrm{EQ}_{\mathrm{CFG}}$ is decidable, since CFLs aren't closed under complements and intersection.
- As it turns out, $\mathrm{EQ}_{\mathrm{CFG}}$ is *not* decidable.
- Undecidability proofs are covered in Chapter 5.

Every CFL is decidable

- A few slides back, we showed that any CFG is decidable. We haven't done this yet for a CFL.
- But since we already know that CFGs define CFLs, this is not an issue. Thus we can ignore PDAs, since it's most likely easier to prove with a CFG than with a PDA.
- **Proof:** Let G be a CFG for a CFL A. We design a TM M_G that decides A. It uses the fact that $\mathrm{A}_{\mathrm{CFG}}$ is decidable; we'll let S denote the deciding TM for $\mathrm{A}_{\mathrm{CFG}}$.

  M_G ="On input w:
  1. Run TM S on input $\langle G, w\rangle$.
  2. If S accepts, then ACCEPT; if S rejects, then REJECT."
- This leads us to the following picture of the language hierarchy.

Hierarchy of classes of languages



- We can convert an FA into a CFG, and so regular⇒context-free.
- We just proved that every CFL language is decidable.
- By definition (Chapter 3), decidable⇒Turing-recognizable. Reverse implication?

Section 4.2: The Halting Problem

The Halting Problem

- One of the most philosophically important theorems in the theory of computation.
- "There is a specific problem that is algorithmically unsolvable."
- In fact, ordinary/practical problems may be unsolvable.
- **Example:** Software verification. Would like to come up with an algorithm that solves the following problem:
  - Given a computer program and a precise specification of what the program is supposed to do, the algorithm is to determine whether the program works as specified.
  - This cannot be done!
- Our first undecidable problem: does an arbitrary TM accept a given input string?

The Halting Problem (cont'd)

- $A_{TM} = \{\langle M, w\rangle : M$ is a TM and M accepts $w\}$.
- A_{TM} is undecidable.
- The culprit? M can loop on w.
- If we could determine that it M loops forever, then could reject. Hence A_{TM} is often called the *halting problem*. As we will show, it is impossible to determine whether an arbitrary TM will always halt (i.e., on every possible input).
- Note that this problem is Turing recognizable: simply simulate M on input w; if M accepts, then we ACCEPT; if it ever rejects, then we REJECT.
- We start by describing our proof technique: the *diagonalization method*.

Diagonalization method

- In 1873, mathematician Georg Cantor was concerned with the problem of measuring the sizes of infinite sets.
- How can we tell whether one infinite set is bigger than the other, or whether they have the same size? What does this even mean?
  - We cannot use the counting method that we'd use for finite sets, because we'd never finish. (Example: How many even integers are there?)
  - Which is larger: the set of all even integers or the set of all finite strings over $\{0,1\}$? (Note: the latter is equivalent to the set of all integers.)
- Cantor observed that two finite sets have the same size if there's a one-to-one correspondence between the two sets.
- Cantor's idea: extend this idea to infinite sets.

Review of function properties

- From basic discrete math (e.g., CS1100).
- Let $f\colon A \to B$.
  - f is *injective* or *one-to-one* if f never maps two distinct elements in A to the same element in B, i.e., if

    $$x, y \in A \land x \neq y \Rightarrow f(x) \neq f(y)$$

    or

    $$x, y \in A \land f(x) = f(y) \Rightarrow x = y.$$

    Examples: The "add-two" function $g\colon \mathcal{Z} \to \mathcal{Z}$ defined by

    $$g(x) = x + 2 \qquad \forall\, x \in \mathcal{Z}$$

    is injective, but the absolute value function $h\colon \mathcal{Z} \to \mathcal{Z}$ defined by

    $$h(x) = |x| \qquad \forall\, x \in \mathcal{Z}$$

    is not injective.

Review of function properties (cont'd)

- Recalling that $f\colon A \to B \ldots$
  - The f is *surjective* if every item in B can be expressed as f applied to some element in A, i.e., i.e., if

    $$b \in B \Rightarrow b = f(a) \text{ for some } a \in A.$$

    (We also say f maps A *onto* B.)
    Examples: The add-two function given above is surjective.
    But the modified add-two function $\tilde{g}\colon \mathcal{N} \to \mathcal{N}$

    $$\tilde{g}(x) = x + 2 \qquad \forall\, x \in \mathcal{N}$$

    is not surjective. (There's no $x \in \mathcal{N}$ such that $\tilde{g}(x) = 1$.)
  - If f is both injective and surjective, it is said to be *bijective* or a (*one-to-one*) *correspondence*.
    (In other words, it's a pairing of the two sets A and B.)

An example of pairing set items

- Let $\mathcal{N} = \{1, 2, 3, \ldots\}$ and $\mathcal{E} = \{2, 4, 6, \ldots\}$ be the natural numbers and the even natural numbers.
- Then $|\mathcal{N}| = |\mathcal{E}|$.
- Why? The function $f\colon \mathcal{N} \to \mathcal{E}$, defined as

  $$f(x) = 2x \qquad \forall\, x \in \mathcal{N},$$

  is a bijection (one-to-one correspondence, pairing) of the two sets $\mathcal{N}$ and $\mathcal{E}$.
- Somewhat counterintuitive, since $\mathcal{E} \subsetneq \mathcal{N}$.
- **Definition:** A set is *countable* if it is finite *or* if it has the same size as $\mathcal{N}$. (Latter case is often said to be *countably infinite* or *enumerable*.)
- So, $\mathcal{E}$ is countable (actually, countably infinite).

Example: Rational numbers

- Let

  $$\mathcal{Q} = \left\{ \frac{m}{n} : m, n \in \mathcal{N} \right\},$$

  the set of positive rational numbers.
- $\mathcal{Q}$ seems much larger than $\mathcal{N}$.
- As we shall see, there is a one-to-one correspondence between $\mathcal{N}$ and $\mathcal{Q}$. Hence, $|\mathcal{Q}| = |\mathcal{N}|$.
- This correspondence must
  - list all the elements of $\mathcal{Q}$,
  - label them (the first with 1, then second with 2, etc.), and
  - make sure that each element in $\mathcal{Q}$ is counted exactly once.

Correspondence between $\mathcal{N}$ and $\mathcal{Q}$

- To get our listing, we make an infinite matrix containing all the positive rational numbers.

$$\begin{array}{cccccc}
\frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\
\frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \cdots \\
\frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \cdots \\
\frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \frac{4}{5} & \cdots \\
\frac{5}{1} & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

- Bad way to list: row by row. Since first row is infinite, would never get to second row!
- Instead, access via the diagonals, but not listing rationals that are equivalent to previously-listed rationals.
- So, the order is $\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{3}{1}, \frac{1}{3}, \ldots$ (note how we skipped $\frac{2}{2}$, since $\frac{2}{2} = \frac{1}{1}$).

Correspondence between $\mathcal{N}$ and $\mathcal{Q}$ (cont'd)

- This yields a correspondence between $\mathcal{N}$ and $\mathcal{Q}$. This correspondence function $f : \mathcal{N} \rightarrow \mathcal{Q}$ is given by the table

n	1	2	3	4	5	…
$f(n)$	$\frac{1}{1}$	$\frac{2}{1}$	$\frac{1}{2}$	$\frac{3}{1}$	$\frac{1}{3}$	…

and may be visualized via the picture

$$
\begin{array}{cccccc}
\frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\
\frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \cdots \\
\frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \cdots \\
\frac{4}{1} & \frac{4}{2} & \frac{4}{3} & \frac{4}{4} & \frac{4}{5} & \cdots \\
\frac{5}{1} & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots &
\end{array}
$$

Theorem: $\mathcal{R}$ is uncountable

- $\mathcal{R}$ is the set of *real numbers*, i.e., numbers with a decimal representation, possibly having an infinite number of digits after the decimal point. For example:
  - $2 = 2.0$
  - $2\frac{3}{5} = 2.6$
  - $\frac{1}{7} = 0.142857142857142857142857\ldots$
  - $\sqrt{2} = 1.4142135623730950488016887\ldots$
  - $\pi = 3.1415926535897932384626433\ldots$
  - $e = 2.7182818284590452353602874\ldots$
- Will show that $\mathcal{R}$ is *uncountable*, i.e., there can be no pairing of elements between $\mathcal{R}$ and $\mathcal{N}$.
- Proof by contradiction: Given any proposed pairing, we can always find some $x \in \mathcal{R}$ that does not appear in the pairing.

Finding a new value x

- Consider the example mapping:

n	$f(n)$
1	3.14159…
2	55.5555…
3	0.12345…
4	0.50000…
⋮	⋮

- Assume that it is complete.
- We now describe a method that is guaranteed to generate a value x not in the (supposedly complete) infinite list of $\mathcal{R}$.

Finding a new value x (cont'd)

- Generate $x \in [0, 1]$ as follows:
  - To guarantee that $x \neq f(1)$, pick a digit not equal to the first digit after the decimal point. Any value other than 1 will work; let's choose 4. Number so far is 0.4.
  - To guarantee that $x \neq f(2)$, pick a digit not equal to the second digit after the decimal point. Any value other than 5 will work; let's choose 6. Number so far is 0.46.
  - To guarantee that $x \neq f(3)$, pick a digit not equal to the third digit after the decimal point. Any value other than 3 will work; let's choose 4. Number so far is 0.464.
  - Continue, choosing values along the "diagonal" of digits. That is, the nth digit of x is chosen as something different than the nth digit of $f(n)$.
    Note: We take care to never choose 0 or 9.
- When done, we are guaranteed to have a value x not already in the list, since it differs in at least one position with every other number in the list.

Proof that $\mathcal{R}$ is uncountable

- In fact, the unit interval $[0,1] = \{x \in \mathcal{R} : 0 \leq x \leq 1\}$ is uncountable.
- Proof by contradiction: Suppose that $[0,1]$ is countable.
- Let $r_1, r_2, r_3, \ldots$ be an enumeration of $[0,1]$.
- Write the decimal expansions of these numbers as

$$r_n = 0.r_{n,1}r_{n2,}r_{n,3}\ldots \qquad \forall\, n \in \mathcal{N}$$

- Now define $x = 0.x_1x_2x_3\ldots$, where

$$x_n = \begin{cases} r_{n,n}+1 & \text{if } r_{n,n} \neq 8 \text{ and } r_{n,n} \neq 9, \\ 5 & \text{if } r_{n,n} = 8 \text{ or } r_{n,n} = 9 \end{cases}$$

- Then x cannot be any of the r_n, since the nth digits of x and r_n differ.
- Contradiction!

Implications

- We have just proved that $\mathcal{R}$ is uncountable.
- Important application in the theory of computation.
- It implies that non-decidable (in fact, non-Turing-recognizable) languages exist. The reason? There are uncountably many languages, yet only countably many Turing Machines. (This needs proof.)
- Since each TM can only recognize a single language and there are more languages that TMs, some languages are not recognized by any TM.
- **Corollary:** Some (actually, "most") languages are not Turing-recognizable.

Proof that some languages are not Turing-recognizable

- The set Σ^* of strings over Σ is countable.
  Why? Form a listing of Σ^* by listing all strings of length 0, followed by all strings of length 1, followed by all strings of length 2, etc.
- The set of all Turing Machines is countable. Why?
  - Each TM M has an encoding $\langle M \rangle$ into a string.
  - If we simply omit all strings that do not represent valid TMs, we get a list of all TMs.

Proof that some languages are not Turing-recognizable (cont'd)

- The set $\mathcal{L}$ of all languages over Σ is uncountable. Why?
  - The set $\mathcal{B}$ of all binary sequences is uncountable. (Use the same diagonalization proof as for $\mathcal{R}$.)
  - $\mathcal{L}$ is uncountable because it has a correspondence with $\mathcal{B}$.
    - Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$.
    - If $A \in \mathcal{L}$, its *characteristic sequence* $\chi_A = (x_1, x_2, x_3, \ldots)$ is defined by taking

$$x_i = \begin{cases} 0 & \text{if } s_i \notin A \\ 1 & \text{if } s_i \in A \end{cases} \qquad \forall \in \mathcal{N}$$

    - The function $f\colon \mathcal{L} \to \mathcal{B}$ given by

$$f(A) = \chi_A \qquad \forall\, A \in \mathcal{L}$$

      is a bijection.
    - Since $\mathcal{B}$ is uncountable and there is a bijection between $\mathcal{B}$ and $\mathcal{L}$, we see that $\mathcal{L}$ is uncountable.
- Since $\mathcal{L}$ is uncountable and the set of TMs is countable, there exists some language to which no TM corresponds.

The Halting Problem is undecidable

- We shall prove that the Halting Problem is undecidable.
  - We started this a while ago . . . .
  - Let $A_{TM} = \{\langle M, w\rangle : M \text{ is a TM and accepts } w\}$.
- Proof technique:
  - Assume A_{TM} is decidable and obtain a contradiction.
  - A diagonalization proof.

Proof: The Halting Problem is undecidable

- Assume A_{TM} is decidable.
- Let H be a decider for A_{TM}.
  - Input $\langle M, w\rangle$, where M is a TM and w is a string.
  - If M accepts w, then H halts and accepts; otherwise, H halts and rejects.
- Construct a TM D using H as a a subroutine.
  - D calls H to determine what M does when the input string is its own description $\langle M\rangle$.
  - D then outputs the *opposite* of H's answer.
  - In summary: $D(\langle M\rangle)$ accepts if M does not accept $\langle M\rangle$, and rejects if M accepts $\langle M\rangle$.
- Now run D on its own description:
  - $D(\langle D\rangle)$ accepts if D does not accept $\langle D\rangle$.
  - $D(\langle D\rangle)$ rejects if D accepts $\langle D\rangle$.
  - A contradiction!! So H cannot be a decider for A_{TM}.

The diagonalization proof

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\dots$	$\langle D\rangle$	$\dots$
M_1	<u>ACC</u>	REJ	ACC	REJ	$\dots$	ACC	$\dots$
M_2	ACC	<u>ACC</u>	ACC	ACC	$\dots$	ACC	$\dots$
M_3	REJ	REJ	<u>REJ</u>	REJ	$\dots$	REJ	$\dots$
M_4	ACC	ACC	REJ	<u>REJ</u>	$\dots$	ACC	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
D	REJ	REJ	ACC	ACC	$\dots$	?	$\dots$

Slightly more concrete version

- Suppose that one can write a C++ function

  `bool halts(Function p, InputSet x);`

  that takes as parameters:
  - any C++ function p, and
  - any input x for p.

  and whose return value is
  - `true` if p halts on x, and
  - `false` if p does not halt on x.
- Consider the C++ function

  `void foo(Function x) { while halts(x, x);}`
- Does `foo(foo)` halt?
  - `foo(foo)` halts $\Longrightarrow$ `foo(foo)` does not halt.
  - `foo(foo)` does not halt $\Longrightarrow$ `foo(foo)` halts.
  - `foo(foo)` halts $\Longleftrightarrow$ `foo(foo)` does not halt . . . Contradiction!
- Thus we have proven that you cannot write a program to determine if an arbitrary program will or will not halt after a finite number of steps.

What does this mean?

Recall what was said earlier:

- The halting problem is not some contrived problem.
- The halting problem asks whether we can tell if some TM M will accept an input string.
- We are asking whether the language

$$A_{TM} = \{\langle M, w\rangle : M \text{ is a TM and } M \text{ accepts } w\}$$

  is decidable.
- The language A_{TM} is *not* decidable!
  - Both M and w are input variables.
  - Halting of some algorithms (e.g., sorting algorithms) *is* decidable.
- A_{TM} is Turing-recognizable (we covered this earlier): simulate the TM M on w; if it accepts/rejects, then we accept/reject.
- The halting problem is special: it gets at the heart of the matter.

Co-Turing recognizable

A language is *co-Turing recognizable* if it is the complement of a Turing-recognizable language.

Theorem

A language is decidable iff it is both Turing-recognizable and co-Turing recognizable.

- Why? To be Turing-recognizable, we must accept in finite time. If we don't accept, we may reject or loop (in which case it is not decidable.)
- We can invert any "question" by taking the complement. This flips the ACCEPT and REJECT answers. Thus if we invert the question for a Turing-recognizable language, then we would get the answer to the original REJECT question in finite time.

Theorem

A language is decidable iff it is both Turing-recognizable and co-Turing recognizable.

Proof.

Forward direction is easy: If A is decidable, then both A and $\overline{A}$ are Turing-recognizable. Backward direction?

- Assume A and $\overline{A}$ are Turing recognizable, with M_1 and M_2 recognizing A and $\overline{A}$.
- Define M ="On input w:
  1. Run both M_1 and M_2 on input w in parallel.
  2. If M_1 accepts, then ACCEPT; if M_2 accepts, then REJECT."
- Every string is in either A or $\overline{A}$, so every string w must be accepted by either M_1 or M_2. Because M halts whenever M_1 or M_2 accepts, M always halts and so is a decider.
- Furthermore, M accepts A and rejects $\overline{A}$. So M is a decider for A, and hence A is decidable. $\square$

Implications

- **Corollary:** For any undecidable language A, either A or its complement $\overline{A}$ is not Turing-recognizable.
- **Corollary:** $\overline{A_{TM}}$ is not Turing-recognizable.
- **Proof:** We know that A_{TM} is Turing-recognizable, but not decidable.
- If $\overline{A_{TM}}$ were also Turing-recognizable, then A_{TM} would be Turing-decidable, which it is not.
- Thus $\overline{A_{TM}}$ is not Turing-recognizable.