DECIDABILITY

21 February 2024 02:12 PM

expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

THEOREM 4.1

A_{DFA} is a decidable language.

PROOF IDEA We simply need to present a TM M that decides A_{DFA}.

```
M = "On input ⟨B, w⟩, 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, accept. If it ends in a
       nonaccepting state, reject."
```

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w\rangle$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components, Q, Σ, δ, q_0, and F. When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects.

Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

We can prove a similar theorem for nondeterministic finite automata. Let

$$A_{\mathsf{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}.$$

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

```
N = "On input ⟨B, w⟩ where B is an NFA, and w is a string:
    1. Convert NFA B to an equivalent DFA C, using the procedure for
       this conversion given in Theorem 1.39.
    2. Run TM M from Theorem 4.1 on input ⟨C, w⟩.
    3. If M accepts, accept; otherwise, reject."
```

Similarly, we can determine whether a regular expression generates a given string. Let $A_{\mathsf{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$.

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

```
P = "On input ⟨R, w⟩ where R is a regular expression and w is a string:
    1. Convert regular expression R to an equivalent NFA A by using
       the procedure for this conversion given in Theorem 1.54.
    2. Run TM N on input ⟨A, w⟩.
    3. If N accepts, accept; if N rejects, reject."
```

Theorems 4.1, 4.2, and 4.3 illustrate that, for decidability purposes, presenting the Turing machine with a DFA, NFA, or regular expression are all equivalent because the machine is able to convert one form of encoding to another.

$$E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}.$$

THEOREM 4.4

E_{DFA} is a decidable language.

```
T = "On input ⟨A⟩ where A is a DFA:
    1. Mark the start state of A.
    2. Repeat until no new states get marked:
    3.     Mark any state that has a transition coming into it from any
           state that is already marked.
    4. If no accept state is marked, accept; otherwise, reject."
```

$EQ_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$.

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem we use Theorem 4.4. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

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

This expression is sometimes called the ***symmetric difference*** of $L(A)$ and $L(B)$ and is illustrated in the following figure. Here $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines. Once we have constructed C we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

```
F = "On input ⟨A, B⟩, where A and B are DFAs:
    1. Construct DFA C as described.
    2. Run TM T from Theorem 4.4 on input ⟨C⟩.
    3. If T accepts, accept. If T rejects, reject."
```

$A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$.

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.

To make this Turing machine into a decider we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 130) we showed that, if G were in Chomsky normal form, any derivation of w has $2n-1$ steps, where n is the length of w. In that case checking only derivations with $2n-1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

```
S = "On input ⟨G, w⟩, where G is a CFG and w is a string:
    1. Convert G to an equivalent grammar in Chomsky normal form.
    2. List all derivations with $2n - 1$ steps, where n is the length of
       w, except if $n = 0$, then instead list all derivations with 1 step.
    3. If any of these derivations generate w, accept; if not, reject."
```

$E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$.

THEOREM 4.8

E_{CFG} is a decidable language.

In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines *for each variable* whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables. The TM R implements this algorithm.

PROOF

```
R = "On input ⟨G⟩, where G is a CFG:
    1. Mark all terminal symbols in G.
    2. Repeat until no new variables get marked:
    3.     Mark any variable A where G has a rule $A \to U_1U_2\cdots U_k$ and
           each symbol $U_1, \ldots, U_k$ has already been marked.
    4. If the start variable is not marked, accept; otherwise, reject."
```

$EQ_{\mathsf{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$.

a similar strategy to prove that EQ_{CFG} is decidable. But something is wrong with this idea! The class of context-free languages is *not* closed under complementation or intersection, as you proved in Exercise 2.2. In fact, EQ_{CFG} is not decidable. The technique for proving so is presented in Chapter 5.

Now we show that every context-free language is decidable by a Turing ma-

THEOREM 4.9

Every context-free language is decidable.

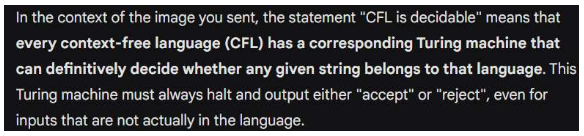
PROOF IDEA Let A be a CFL. Our objective is to show that A is decidable.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

In the context of the image you sent, the statement "CFL is decidable" means that

Every context-free language is decidable.

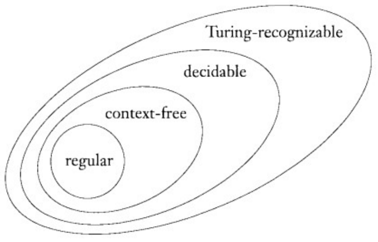

In the context of the image you sent, the statement "CFL is decidable" means that **every context-free language (CFL) has a corresponding Turing machine that can definitively decide whether any given string belongs to that language.** This Turing machine must always halt and output either "accept" or "reject", even for inputs that are not actually in the language.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

```
M_G = "On input w:
    1. Run TM S on input ⟨G, w⟩
    2. If this machine accepts, accept; if it rejects, reject."
```

TM S is chomsky normal form

classes of languages that we have described so far: regular, context free, decidable, and Turing-recognizable. The following figure depicts this relationship.



HALTING PROBLEM

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

THEOREM 4.11

A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus this theorem shows that recognizers *are* more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM}.

```
U = "On input ⟨M, w⟩, where M is a TM and w is a string:
    1. Simulate M on input w.
    2. If M ever enters its accept state, accept; if M ever enters its
       reject state, reject."
```

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could *reject*. Hence A_{TM} is sometimes called the ***halting problem***. As we demonstrate, an algorithm has no way to make this determination.

The Turing machine U is interesting in its own right. It is an example of the ***universal Turing machine*** first proposed by Turing. This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine. The universal Turing machine played an important early role in stimulating the development of stored-program computers.

DIAGONALISATION

DEFINITION 4.12

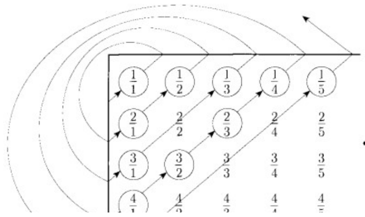
Assume that we have sets A and B and a function f from A to B. Say that f is ***one-to-one*** if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is ***onto*** if it hits every element of B—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the ***same size*** if there is a one-to-one, onto function $f\colon A \longrightarrow B$. A function that is both one-to-one and onto is called a ***correspondence***. In a correspondence every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

DEFINITION 4.14

A set A is ***countable*** if either it is finite or it has the same size as $\mathcal{N}$.

THEOREM 4.17

$\mathcal{R}$ is uncountable.



We construct the desired x by giving its decimal representation. It is a number between 0 and 1, so all its significant digits are fractional digits following the decimal point. Our objective is to ensure that $x \neq f(n)$ for any n. To ensure that $x \neq f(1)$ we let the first digit of x be anything different from the first fractional digit 1 of $f(1) = 3.\underline{1}4159\ldots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$ we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.5\underline{5}5555\ldots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12\underline{3}45\ldots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f, we obtain all the digits of x, as

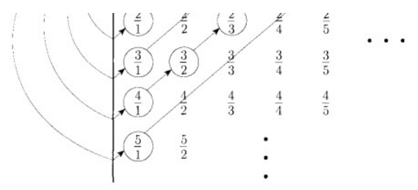


FIGURE 4.16
A correspondence of $\mathcal{N}$ and $\mathcal{Q}$

digit 1 of $f(1) = 3.\underline{1}4159\ldots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$ we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.5\underline{5}5555\ldots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12\underline{3}45\ldots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f, we obtain all the digits of x, as shown in the following table. We know that x is not $f(n)$ for any n because it differs from $f(n)$ in the nth fractional digit. (A slight problem arises because certain numbers, such as $0.1999\ldots$ and $0.2000\ldots$, are equal even though their decimal representations are different. We avoid this problem by never selecting the digits 0 or 9 when we construct x.)

The preceding theorem has an important application to the theory of computation. It shows that some languages are not decidable or even Turing-recognizable, for the reason that there are uncountably many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine. Such languages are not Turing-recognizable, as we state in the following corollary.

COROLLARY 4.18

Some languages are not Turing-recognizable.

PROOF To show that the set of all Turing machines is countable we first observe that the set of all strings Σ^* is countable, for any alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable we first observe that the set of all infinite binary sequences is uncountable. An ***infinite binary sequence*** is an unending sequence of 0s and 1s. Let $\mathcal{B}$ be the set of all infinite binary sequences. We can show that $\mathcal{B}$ is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that $\mathcal{R}$ is uncountable.

Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with $\mathcal{B}$, thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$. The ith bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the ***characteristic sequence*** of A. For example, if A were the language of all strings starting with a 0 over the alphabet $\{0,1\}$, its characteristic sequence χ_A would be

$\Sigma^* = \{$	ε ,	0 ,	1 ,	00 ,	01 ,	10 ,	11 ,	000 ,	001 ,	$\cdots$	$\}$;
$A = \{$		0 ,		00 ,	01 ,			000 ,	001 ,	$\cdots$	$\}$;
$\chi_A =$	0	1	0	1	1	0	0	1	1	$\cdots$	.

The function $f\colon \mathcal{L} \longrightarrow \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto and hence a correspondence. Therefore, as $\mathcal{B}$ is uncountable, $\mathcal{L}$ is uncountable as well.

THE HALTING PROBLEM IS UNDECIDABLE

Now we are ready to prove Theorem 4.11, the undecidability of the language

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

$\longrightarrow$ undecidable Recognizable

PROOF We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

$\longrightarrow$ Now since H decides for every machine But D cannot be decided. ∴ contraⁿ H & D both cannot exists.

Now since H (the decider) cannot exist, therefore not every input <M,w> can be decided. Therefore Lang. is undecidable.

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M\rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D.

```
D = "On input ⟨M⟩, where M is a TM:
    1. Run H on input ⟨M, ⟨M⟩⟩.
    2. Output the opposite of what H outputs; that is, if H accepts,
       reject and if H rejects, accept."
```

Don't be confused by the idea of running a machine on its own description! That is similar to running a program with itself as input, something that does occasionally occur in practice. For example, a compiler is a program that translates other programs. A compiler for the language Pascal may itself be written in Pascal, so running that program on itself would make sense. In summary,

$$D(\langle M\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M\rangle. \end{cases}$$

What happens when we run D with its own description $\langle D\rangle$ as input? In that case we get

$$D(\langle D\rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

$\longrightarrow$ if it REJECTS then it means D accepts <D> which recurrsively means it rejects. This goes on & on so it is undecisive

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus neither TM D nor TM H can exist.

4 on so it is undecisive

A TURING-UNRECOGNIZABLE LANGUAGE

a language and its complement are Turing-recognizable, the language is decidable. Hence, for any undecidable language, either it or its complement is not Turing-recognizable. Recall that the complement of a language is the language consisting of all strings that are not in the language. We say that a language is ***co-Turing-recognizable*** if it is the complement of a Turing-recognizable language.

THEOREM 4.22

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

PROOF We have two directions to prove. First, if A is decidable, we can easily see that both A and its complement $\overline{A}$ are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For the other direction, if both A and $\overline{A}$ are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for $\overline{A}$. The following Turing machine M is a decider for A.

```
M = "On input w:
    1. Run both M_1 and M_2 on input w in parallel.
    2. If M_1 accepts, accept; if M_2 accepts, reject."
```

Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2. In this case M takes turns simulating one step of each machine, which continues until one of them accepts.

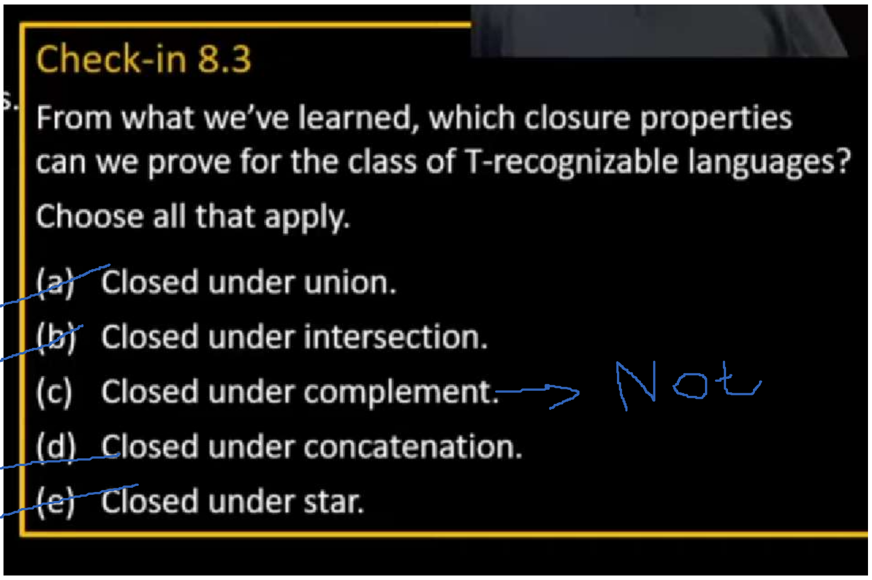
Now we show that M decides A. Every string w is either in A or $\overline{A}$. Therefore either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A. So M is a decider for A, and thus A is decidable.

COROLLARY 4.23

$\overline{A_{\mathsf{TM}}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{\mathsf{TM}}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{\mathsf{TM}}}$ must not be Turing-recognizable.

on a Turing machine and gave one example of a problem, A_{TM}, that is computationally unsolvable. In this chapter we examine several additional unsolvable



Check-in 8.3

From what we've learned, which closure properties can we prove for the class of T-recognizable languages?

Choose all that apply.

[✓ (a)] Closed under union.

[✓ (b)] Closed under intersection.

(c) Closed under complement. $\longrightarrow$ Not

[✓ (d)] Closed under concatenation.

[✓ (e)] Closed under star.