

CS 273, Lecture 20

More decidable problems, and simulating TM and “real” computers

3 April 2008

This lecture presents more example of languages that are *Turing decidable*, from Sipser section 4.1.

1 Review: decidability facts for regular languages

A language is *decidable* if there is a TM that is a decider (i.e., a TM that always stops) that accepts this language.

Decidable DFA problems. If M is a DFA, the string encoding of M is written $\langle M \rangle$. The encoding of a pair M and w is written $\langle M, w \rangle$

Remember our notation for languages, e.g.:

E_{DFA} (language is empty)

EQ_{DFA} (two languages are the same)

DFA accepts a word is TM decidable. Consider the language

$$A_{\text{DFA}} = \left\{ \langle M, w \rangle \mid M \text{ is a DFA, and } w \text{ is a word} \right\}.$$

Here we need to encode a DFA, but that’s not much different than specifying a graph with labels on the edges (given details). As such, this language is well defined.

DFA accepts no word is TM decidable. Consider the language

$$E_{\text{DFA}} = \left\{ \langle M \rangle \mid M \text{ is a DFA and } L(M) = \emptyset \right\}.$$

This is just (s, t) -reachability, and as such this language is decidable.

Two DFA’s accepts the same language TM decidable. Consider the language

$$EQ_{\text{DFA}} = \left\{ \langle M, N \rangle \mid M, N \text{ are DFA's and } L(M) = L(N) \right\}.$$

This is just (s, t) -reachability, and as such this language is decidable.

Regular expression generation is TM decidable.

$$A_{\text{regex}} = \left\{ \langle R, w \rangle \mid R \text{ is regular expression generating } w \right\}.$$

To decide this language, the TM can convert R into a DFA, and then verify whether or not the DFA accepts w .

NFA generation is TM decidable.

$$A_{\text{NFA}} = \left\{ \langle M, w \rangle \mid M \text{ is a NFA generating } w \right\}.$$

The TM can convert M into a DFA, and then verify whether or not the DFA accepts w .

In general, problems involving regular languages are decidable.

2 Decidable problems involving context-free languages

The situation with context-free languages is more complicated, because some problems are Turing decidable and some are not.

2.1 Context-free languages are TM decidable

Given a PDA P , we are interested in the question of whether we can build a TM decider that accepts $L(P)$. Observe, that we can turn P into an equivalent CFG, and this CFG can be turned into an equivalent CNF grammar G . With G it is now easy to decide if an input word w is in $L(G)$.

Specifically, observe that since G is in CNF, we know that the derivation of a word w takes exactly $2|w| - 1$ steps.¹ (We showed this in lecture 15.) So, to tell if G generates w , we can generate all parse trees with $2|w| - 1$ variable nodes. (Or, if you prefer, $3|w| - 1$ node of all types in the tree.)

There's a couple ways to generate these trees. We could start at the root (start symbol) and search through all ways to expand each node using the rules of the grammar G . Then see whether any of these trees yields w (has w on its leaves).

Or we could generate all binary trees with $2|w| - 1$ nodes. Generate all possible assignments of variable labels to these nodes. Add a layer of leaves at the bottom containing w . Then check whether each tree is a valid parse tree for G .

Either way, this algorithm of searching over all possible such trees is not very efficient, but it will always terminate. If the word is derivable by G , we will discover the tree generating it, and the TM would accept w , otherwise, after exhausting all possible trees, the TM would reject the input. We conclude that given a PDA, we can always generate a TM accepting the same language, and furthermore, the TM halts for all inputs.

Of course, we described the algorithm in very high level (i.e., the algorithm generating all possible parsing trees and checking if one of them generates w legally). Hopefully, the reader can see how one can implement this algorithm using regular programming language

¹Except if w is the empty string, which requires one step. But that special case isn't a big issue.

(say, Java). The key observation that we can also implement this algorithm as a TM. The details would be overwhelming without a compiler, but it could be done.

Lemma 2.1 *Given a PDA P , there is a TM M which is a decider, and $L(P) = L(M)$. Namely, for every PDA there exists an equivalent TM.*

2.2 Is a word in a CFG?

Consider the language

$$A_{\text{CFG}} = \left\{ \langle G, w \rangle \mid G \text{ is a CFG and } G \text{ generates } w \right\}.$$

Is it decidable?

Indeed, it is, and the TM is similar to the one in Section 2.1. However, instead of the grammar G being hard-coded into our TM, we need to dynamically accept any context-free grammar as input. How does this affect the design of the TM?

First, for this problem, the grammar will come in on the TM's input tape. So when it first starts up, The TM for A_{CFG} will copy bits of the grammar (e.g. the start symbol, the rules) onto appropriate working tapes. A Turing machine handling only a single context-free language would store these details in its state diagram². When it powers up, its first action (before reading any of the input tape) would be to run through a fixed sequence of states that write the grammar information out onto the working tapes.

If you have trouble imaging this for a whole context-free grammar, think about storing a short string like UIUC in the TM state diagram, to be written out on (say) tape 2. The first state transition in the TM would write U onto tape 2, the next three transitions would write I, then U, then C. Finally, it would move the tape 2 head back to the beginning and transition into the first state that does the actual computation.

Second, when we build a TM M that handles only one fixed grammar, the conversion to CNF is done before we create the code for M . A TM deciding A_{CFG} must do the CNF conversion on the fly. This is ok, because CNF conversion is a straightforward algorithm for which we could write TM code.

We could also hard-code the string w into our TM but leave the grammar as a variable input:

$$A_{\text{CFG},w} = \left\{ \langle G \rangle \mid G \text{ is a CFG and } G \text{ generates } w \right\}.$$

Clearly this is also decidable, for any string w .

2.3 Is a CFG empty?

Consider the language

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

²Using a *lot* of states!

Is it decidable?

To this end, the TM mark all the variables that can generate (in one step) a string of terminals (or ϵ of course). We will refer to such a variable as being useful. Now, the TM iterates repeatedly over the rules of G . For a rule

$$A \rightarrow w,$$

where w is a string of terminals and variables, the variable A is useful, if all the variables of w are useful, and in such a case we will mark A as useful. The loop halts when the TM has made a full pass through the rules of G without marking anything new as useful.

This TM accepts the input grammar if the initial variable of G is useful, and otherwise it rejects.

At every iteration over all the rules of G the TM must designate at least one new variable as new to repeat this process again. So it follows that the number of outer iterations performed by this algorithm is bounded by the number of variables in the grammar G , implying that this algorithm always terminates.

Lemma 2.2 *The language E_{CFG} is decidable.*

3 Undecidable problems for CFGs

Unfortunately, not all CFG problems are decidable.

Consider the language

$$EQ_{CFG} = \left\{ \langle G, G' \rangle \mid G, G' \text{ are context-free grammars and } L(G) = L(G') \right\}.$$

Is this language TM decidable? Namely, can we decide if two CFG are equivalent. To remind you, this question was solvable for DFAs (and thus also for regular expression). Somewhat surprisingly, this language is *not* TM decidable. We will see a proof later in the course.

3.1 Is a CFG full?

Consider the language

$$ALL_{CFG} = \left\{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^* \right\}.$$

Is it decidable?

This language is also not decidable. Again, we'll see a proof later in the term.

4 Simulating a real computer with a Turing machine

We would like to argue that we can simulate a “real” world computer on a Turing machine. Here are some key program features that we would like to simulate on a TM.

- Numbers & arithmetic:

We already saw in previous lecture how some basic integer operations can be handled. It's not too hard to extend these to negative integers and perform all required numerical operations if we allow a TM with multiple tapes. As such, we can assume that we can implement any standard numerical operation.

Of course, can also do floating point operations on a TM. The details are overwhelming but they are quite doable. In fact, until 15 years ago, many computers implemented floating point operations using integer arithmetic. If you haven't already seen how floating point arithmetic works, you will (e.g. in CS 232, 257).

- Stored constant strings:

The program we are trying to translate into a TM might have strings and constants in it. For example, it might check if the input contains the (all important) string UIUC. As we saw above, we can encode such strings in the states. Initially, on power-up, the TM starts by writing out such strings, onto a special tape that we use for this purpose.

- Random-access memory:

We will use an associative memory. Here, consider the memory as having a unique label to identify it (i.e., its address), and content. Thus, if cell 17 contains the value `abc`, we will consider it as storing the pair `(17, abc)`. We can store the memory on a tape as a list of such pairs. Thus, the tape might look like:

`(17, abc)$ (1, samuel)$ (85, no clue)$... (11, stamp)$ _____`

Here, address 17 stores the string `abc`, address 1 stores the string `samuel`, and so on.

Reading the value of address x from the tape is easy. Suppose x is written on \textcircled{i} , and we would like to find the value associated with x on the memory tape and write it onto \textcircled{j} . To do this, the TM scans \textcircled{mem} the memory tape (i.e., the tape we use to simulate the associative memory) from the beginning, till the TM encounter a pair in \textcircled{mem} having x as its first argument. It then copies the second part of the pair to the output tape \textcircled{j} .

Storing new value (x, y) in memory is almost as easy. If a pair having x as first element exists you delete it out (by writing a special crossout character over it), and then you write the new pair (x, y) in the end of the tape \textcircled{mem} .

If you wanted to use memory more efficiently, the new value could be written into the original location, whenever the original location had enough room. You could also write new pairs into crossed-out regions, if they have enough room. Implementations of C `malloc/free` and Java garbage collection use slightly more sophisticated versions of these ideas. However, TM designers rarely care about efficiency.

- Subroutine calls:

To simulate a real program, we need to be able to do calls (and recursive calls). The standard way to implement such things is by having a stack. It's clear how to implement a stack on its own TM tape.

We need to store three pieces of information for each procedure call.

- Private working space
- The return value
- The name of the state to go to after the call is finished

The private working space needs to be implemented with a stack, because a set of nested procedure calls might be active all at once, including several recursive calls to the same procedure.

The return value can be handled by just putting it onto a designated register tape, say ₂₄.

Right before we give control over to a procedure, we need to store the name of the state it should return to when it's done. This allows us to call a single fixed piece of code from several different places in our TM. Again, these return points need to be put on a stack, to handle nested procedure calls.

After it returns from a procedure, the TM reads the state name to return to. A special set of TM states handle reading a state name and transitioning to the corresponding TM state.

These are just the most essential features for a very simple general-purpose. In CS 421, you'll see how to implement fancier program features (e.g. garbage collection, objects) on top of this simple model.

5 Turing machine simulating a Turing machine

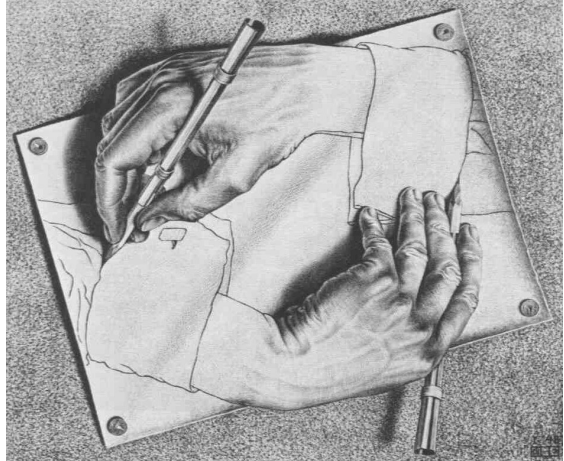
5.1 Motivation

We already seen that a TM can simulate a DFA. We think about TMs as being just regular computer programs. So think about an interpreter. What is it? It's a program that reads in another program (for example, think about Java virtual machine) and runs it.³

³Things of course are way more complicated in practice, since Java virtual machines nowadays usually compile portions of the code being run frequently to achieve faster performance, but still, you can safely think about a JVM as an interpreter.

So, what would be the equivalent of an interpreter in the language of Turing machines? Well, it's a TM that reads in a description of a TM M , and an input w for it, and simulates running M on w .

Initially this construct looks very weird - inherently circular in nature. But it is useful for the same reason interpreters are useful: It enables us to manipulate TMs (i.e., programs) directly and modify them without knowing in advance what they are. In particular, we can start talking computationally about ways of manipulating TMs (i.e., programs).



For example, in a perfect world (which we are not living in, naturally), we would like to give a formal specification of a program (say, a TM that decides if a prime number is prime), and have another program that would swallow this description and spit out the program performing this computation (i.e., have a computer that writes our programs for us).

A more realistic example is a compiler which translates (say) Java code into assembler. It takes code as input and produces code in a different language as output. We could also build an optimizer that reads Java code and produces new code, also in Java but more efficient. Or a cheating helper program that reads Java code and writes out a new version with different variable names and modified comments.

5.2 The universal Turing machine

We would like to build the *universal Turing machine* U_{TM} that recognizes the language

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

It would be important to emphasize here that U_{TM} is **not** a decider. Namely, it stops only if M accepts w , but it might run forever if M does not accept w .

So, the input for U_{TM} is an encoding $\langle M, w \rangle$. As a first step, the U_{TM} would verify that the input is in the right format (such a reasonable encoding for a TM was given as an exercise in the homework). The U_{TM} would copy different components of the input into different tapes:

$$\text{Ⓜ}_1 : \delta.$$

It's going to be a sequence (separated by \$) of transitions. A transition $(q, c) \rightarrow (q', t, L)$ would be encoded as a string of the form:

$$(\#q, c) - (\#q', t, L)$$

where $\#q$ is the index which is the index of the state q (in M) and $\#q'$ is the index of q' .

$$\text{Ⓜ}_2 : \#q_0.$$

$\odot_3 : \#q_{acc}$.

$\odot_4 : \#q_{rej}$.

$\odot_5 : \$w$.

Once done copying the input, the U_{TM} would move the head of \odot_5 to the beginning of the tape. It then performs the following loop:

(I) Loop:

(i) Scan \odot_1 to find transition matching state on \odot_2 and the character under the head of \odot_5 .

(ii) Update state on \odot_2 .

(iii) Update character and head position on \odot_5 .

We repeat this till the state in \odot_2 is equal to the state written on either \odot_3 (q_{acc}) or \odot_4 (q_{rej}).

Naturally, U_{TM} accepts if $\odot_2 = \odot_3$ and rejects if $\odot_2 = \odot_4$.