Lecture 20: More decidable problems, and simulating TM and "real" computers

9 April 2009

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

Decidable DFA problems. The following languages are all decidable.

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

This is the language of all DFAs with an empty language.

(B) EQ_{DFA}: the language of all pairs of DFAs that have the same language.

(C) $A_{DFA} = \left\{ \langle M, w \rangle \,\middle|\, M \text{ is a DFA, } w \text{ is a word, and } M \text{ accepts } w \right\}$.

Here $\langle M, w \rangle$ is in the language if and only if the DFA M accepts w.

(D) $A_{NFA} = \left\{ \langle M, w \rangle \,\middle|\, M \text{ is a NFA generating } w \right\}$.

(E) $EQ_{DFA} = \left\{ \langle M, D \rangle \,\middle|\, M, D \text{ are DFA's and } L(M) = L(D) \right\}$.

(F) $A_{regex} = \left\{ \langle R, w \rangle \,\middle|\, R \text{ is a regular expression generating } w \right\}$.

To decide this language, the TM can convert R into a DFA M, and then check if $\langle M, w \rangle \in A_{DFA}$.

2 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 RA P, we are interested in the question of whether we can build a TM decider that accepts L(M). Observe, that we can turn P into an equivalent CFG, and this CFG can be turned into an equivalent CNF grammar $\mathcal{G}$. With $\mathcal{G}$ it is now easy to decide if an input word w is in $\mathsf{L}(\mathcal{G})$. Indeed, we can either using the CYK algorithm to decide if a word is in the grammar, or alternatively, enumerate all possible parse trees for the given CNF that generates the given word w. That is, if $n = |w|$, then we need to generate all possible parse trees with $2n - 1$ internal nodes (since this is the size of a parse tree deriving such a word in CNF), and see if any of them generates w. In either case, we have the following.

Lemma 2.1 *Given a RA* P*, there is a TM* T *which is a decider, and* $\mathsf{L}(\mathtt{P}) = \mathsf{L}(\mathtt{T})$. *Namely, for every RA there exists an equivalent TM.*

2.2 Is a word in a CFG?

The following construction of a TM is somewhat similar to the one in Section 2.1.

Lemma 2.2 *The language* $\mathrm{A}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G}, w \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathcal{G} \text{ generates } w \right\}$. *is decidable.*

Proof: We build a TM $\mathtt{T}_{\mathsf{CFG}}$ for $\mathrm{A}_{\mathsf{CFG}}$. The input for it is the pair $\langle \mathcal{G}, w \rangle$. AS a first step, we convert $\mathcal{G}$ to be in CNF (we saw the algorithm of how to do this in detail in class). Let $\mathcal{G}'$ denote the resulting grammar. Next, we use CYK to decide if $w \in \mathsf{L}(\mathcal{G}')$. If it is, the TM $\mathtt{T}_{\mathsf{CFG}}$ accepts, otherwise it rejects. ■

Given a TM decider $\mathtt{T}_{\mathsf{CFG}}$ for $\mathrm{A}_{\mathsf{CFG}}$, building a TM decider that has language equal to a specific given $\mathcal{G}$ is easy. Specifically, given $\mathcal{G}$, we would like to build a TM decider $\mathtt{T}'$ such that $\mathsf{L}(\mathtt{T}') = \mathsf{L}(\mathcal{G})$.

So, modify the given TM to encode $\mathcal{G}$. As a first step, the new TM $\mathtt{T}'$ would write $\mathcal{G}$ on the input tape (next to the input word w). Next, it would run the TM $\mathtt{T}_{\mathsf{CFG}}$ on this given input to decide if $\langle \mathcal{G}, w \rangle \in \mathrm{A}_{\mathsf{CFG}}$.

Remark 2.3 (Encoding instances inside a TM.) The above demonstrates that given a more general algorithm we can use it to solve the problem for specific instances. This is done by encoding the given specific instance in the constructed TM.

If you have trouble imaging this encoding the whole CFG grammar into the TM, as done above, 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.

Note, that doing this encoding of a specific instance inside the TM, does not necessarily yields the most efficient TM for the problem. For example, in the above, we could first convert the given instance into CNF before encoding it into the TM.

We could also hard-code the string w into our TM but leave the grammar as a variable input. We omit the proof of the following easy lemma.

Lemma 2.4 *Let w be a specific string. The language $A_{\mathsf{CFG},w} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathcal{G} \text{ generates } w \right\}$ is decidable.*

2.3 Is a CFG empty?

Lemma 2.5 *The language $\mathsf{E}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathsf{L}(\mathcal{G}) = \emptyset \right\}$ is decidable.*

Proof: We already saw that in the conversion algorithm of a CFG into CNF (this was one of the initial steps of this conversion). We shortly re-sketch the algorithm.

To this end, the TM mark all the variables in $\mathcal{G}$ 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 $\mathcal{G}$. For a rule $\mathsf{X} \rightarrow w$, where w is a string of terminals and variables, the variable X is useful, if all the variables of w are useful, and in such a case we will mark X as useful. The loop halts when the TM has made a full pass through the rules of $\mathcal{G}$ without marking anything new as useful.

This TM accepts the input grammar if the initial variable of $\mathcal{G}$ is useful, and otherwise it rejects.

At every iteration over all the rules of $\mathcal{G}$ 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 $\mathcal{G}$‘, implying that this algorithm always terminates. ∎

2.4 Undecidable problems for CFGs

We quickly mention a few problems that are not TM decidable. We will prove this fact later in the course. The following languages are *not* TM decidable.

(i) $\mathsf{EQ}_{\mathsf{CFG}} = \left\{ \langle \mathcal{G}, \mathcal{G}' \rangle \;\middle|\; \mathcal{G}, \mathcal{G}' \text{ are CFG and } \mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathcal{G}') \right\}$.

To see why this is surprising, we remind the reader that this language was solvable for DFAs.

(ii) $ALL_{\mathsf{CFG}} = \left\{ \langle \mathcal{G} \rangle \;\middle|\; \mathcal{G} \text{ is a CFG and } \mathsf{L}(\mathcal{G}) = \Sigma^* \right\}$.

3 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 is 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 20 years[1] ago, many computers implemented

[1] This number keep changing. Very irritating.

floating point operations using integer arithmetic. Hardware implementation of floating point-operations became mainstream, when Intel introduced the i486 in 1989 that had FPU (floating-point unit). You would probably will see/seen how floating point arithmetic works in computer architecture courses.

- **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 $(17, \texttt{abc})$. We can store the memory on a tape as a list of such pairs. Thus, the tape might look like:

  $$(17, \texttt{abc})\$(1, \texttt{samuel})\$(85, \texttt{noclue})\$ \ldots (11, \texttt{stamp})\$\texttt{_____} \ldots$$

  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 ✇$_i$, and we would like to find the value associated with x on the memory tape and write it onto ✇$_j$. To do this, the TM scans ✇$_{mem}$ the memory tape (i.e., the tape we use to simulate the associative memory) from the beginning, till the TM encounter a pair in ✇$_{mem}$ having x as its first argument. It then copies the second part of the pair to the output tape ✇$_j$.

  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 cross-out character over it), and then you write the new pair (x, y) in the end of the tape ✇$_{mem}$.

  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 is clear how to implement a stack on its own TM tape.

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

  (i) private working space,

  (ii) the return value,

  (iii) and the name of the state to return to after the call is done.

  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 is 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.

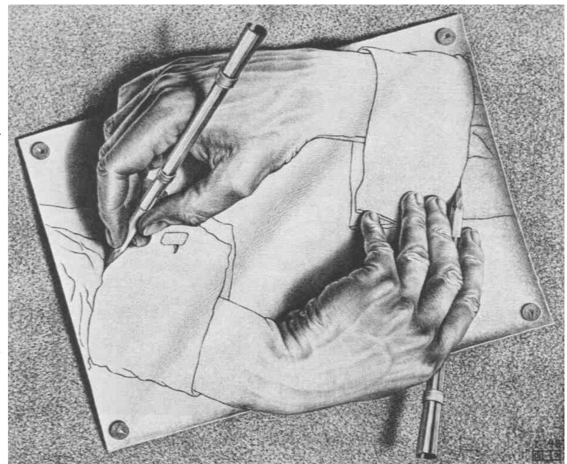
After it returns from a procedure, the TM reads the state name to return to. A special set of TM states handle reading a state name and transitioning to the corresponding TM state.

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

4 Turing machine simulating a Turing machine

4.1 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 is a program that reads in another program (for example, think about Java virtual machine) and runs it.[2]



So, what would be the equivalent of an interpreter in the language of Turing machines? Well, its 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 enable 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 spits 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 assembly code. It takes code as input and procedures code in a different language as output. We could

[2]Things 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 (i.e., just in time compilation [JIT]), but still, you can safely think about a JVM as an interpreter.

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.

4.2 The universal Turing machine

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

$$\mathsf{A}_{\mathsf{TM}} = \left\{ \langle \mathtt{T}, w \rangle \;\middle|\; \mathtt{T} \text{ is a } \mathsf{TM} \text{ and } \mathtt{T} \text{ accepts } w \right\}.$$

We emphasize that U_{TM} is **<u>not</u>** a decider. Namely, its stops only if $\mathtt{T}$ accepts w, but it might run forever if $\mathtt{T}$ does not accept w.

To simplify our discussion, we assume that $\mathtt{T}$ is a single tape machine with some fixed alphabet (say $\Sigma_{\mathtt{T}} = \{\mathtt{0}, \mathtt{1}\}$ and the tape alphabet is $\Gamma_{\mathtt{T}} = \{\mathtt{0}, \mathtt{1}, \llcorner\!\lrcorner\}$. To simplify the discussion, the TM for A_{TM} is going to be a multi-tape machine. Naturally, one can convert this TM into a single tape TM.

So, the input for U_{TM} is an encoding $\langle \mathtt{T}, w \rangle$. 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:

✇$_1$: Transition function δ of $\mathtt{T}$.

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

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

where $\#q$ is the index which is the index of the state q (in $\mathtt{T}$) and $\#q'$ is the index of q'.

More specificially, you an think about the states of $\mathtt{T}$ being numbered between 1 and m, and $\#q$ is just the binary representation of the index of the state q.

✇$_2$: $\#q_0$ – the initial state of $\mathtt{T}$.

✇$_3$: $\#q_{\text{acc}}$ – the accept state of $\mathtt{T}$.

✇$_4$: $\#q_{\text{rej}}$ – the reject state of $\mathtt{T}$.

✇$_5$: $\$w$ – the input tape to be handled.

Once done copying the input, the U_{TM} would move the head of ✇$_5$ to the beginning of the tape. It then performs the following loop:

(I) Loop:

   (i) Scan ✇$_1$ to find transition matching state on ✇$_2$ and the character under the head of ✇$_5$.

   (ii) Update state on ✇$_2$.

(iii) Update character and head position on ☢$_5$.

We repeat this till the state in ☢$_2$ is equal to the state written on either ☢$_3$ (q_{acc}) or ☢$_4$ (q_{rej}).

Naturally, U_{TM} accepts if ☢$_2$ = ☢$_3$ and rejects if ☢$_2$ = ☢$_4$ at any point during the simulation.