Lecture 16

Universal stack machines and a non-semidecidable language

In this lecture we will describe a universal stack machine. This is a stack machine that, when given the encoding of an arbitrary stack machine, can simulate that machine on a given input. To describe such a machine, we must naturally consider encodings of stack machines, and this will be the first order of business for the lecture.

Once we are done discussing universal stack machines, we will encounter our first example of a language that is not semidecidable (and is therefore not decidable). By using the non-semidecidability of this language, many other languages can be shown to be either undecidable or non-semidecidable, as we will see in the lecture following this one.

16.1 An encoding scheme for DSMs

In the previous lecture we discussed in detail an encoding scheme for DFAs, and we observed that this scheme is easily adapted to obtain an encoding scheme for NFAs. While we did not discuss specific encoding schemes for regular expressions and context-free grammars, we made use of the fact that one can devise encoding schemes for these models without difficulty.

We could follow a similar route for DSMs, as there are no new conceptual difficulties that arise for this model in comparison to the other models just mentioned. However, given the high degree of importance that languages involving encodings of DSMs will have in the remainder of the course, it is fitting to take a few moments to be careful and precise about this notion. As is the case for just about every encoding scheme we consider, there are many alternatives to the encoding scheme for DSMs we will define—our focus on the specifics of this encoding scheme is done
in the interest of clarity and precision, and not because the specifics themselves are essential to the study of computability.

Throughout the discussion that follows, we will assume that
\[ M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \] (16.1)
is a given DSM whose encoding is to be described. We will assume that \( M \) has \( r \) stacks, for \( r \) being an arbitrary positive integer (which our encoding scheme must specify). Along similar lines to the discussion of string encodings, as well as the encodings of other models discussed in the previous lecture, we will make the assumption that the state set \( Q \) of \( M \) takes the form
\[ Q = \{q_0, \ldots, q_{m-1}\} \] (16.2)
for some positive integer \( m \), and that the input and stack alphabets of \( M \) take the form
\[ \Sigma = \{0, \ldots, k-1\} \quad \text{and} \quad \Gamma = \{0, \ldots, n-1\} \] (16.3)
for positive integers \( k \) and \( n \) satisfying \( k < n \). It is necessarily the case that \( k < n \), as the bottom-of-the-stack marker \( \diamond \) is contained in \( \Gamma \) but not \( \Sigma \), and hereafter we will identify the bottom-of-the-stack marker \( \diamond \) with the last symbol \( n-1 \) of \( \Gamma \). The encoding scheme we will define will encode \( M \) as a binary string \( \langle M \rangle \in \{0, 1\}^* \).

We will first describe how the possible actions that \( M \) makes on each individual non-halting state will be encoded. Once this is done, an encoding of the transition function will be obtained by simply encoding an ordered list of the encodings that describe the actions that \( M \) makes on its individual non-halting states. Throughout this process, let us agree that each state \( q \in Q \) is to be encoded as the binary string \( \langle q \rangle \) that is obtained by encoding that state’s index in binary notation (so that \( \langle q_0 \rangle = 0, \langle q_1 \rangle = 1, \langle q_2 \rangle = 10, \text{etc.} \)), and that stack symbols are encoded in a similar way (so that \( \langle 0 \rangle = 0, \langle 1 \rangle = 1, \langle 2 \rangle = 10, \text{etc.} \)).

There are two possibilities for a given non-halting state \( q \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\} \):

1. The state \( q \) is a push state. This means that there is a single transition that originates from state \( q \); this transition must be labeled by a stack symbol \( a \in \Gamma \) and must lead to some state \( p \in Q \).

2. The state \( q \) is a pop state. This means that there is one transition originating from the state \( q \) for each stack symbol \( a \in \Gamma \). We will write \( p_a \) to denote the state to which \( M \) transitions when it pops \( a \) off of the stack. (Note that this is a context-dependent notation that only makes sense when we have in mind a particular choice of \( q \).)
Figure 16.1: An example of a DSM whose encoding will be calculated. This DSM appeared in Lecture 12; it erases the stack \( X \) and accepts. Here, however, the reject state has been explicitly included in the diagram, and the states are clearly labeled by their names \( q_0 \), \( q_1 \), \( q_2 \), and \( q_3 \). For the sake of this example, we will assume that there is just one stack, so that \( X \) refers to stack number 0.

In both cases, one of the stacks, indexed by \( s \in \{0, \ldots, r - 1\} \), is associated with the state, and once again we will encode this stack index as a string \( \langle s \rangle \) using binary notation. For the first case, in which \( q \) is a push state, we will encode the information summarized above as follows:

\[
\langle \langle q \rangle, \langle s \rangle, 0, \langle a \rangle, \langle p \rangle \rangle.
\]  \hspace{1cm} (16.4)

For the second case, in which \( q \) is a pop state, we will encode the information summarized above as follows:

\[
\langle \langle q \rangle, \langle s \rangle, 1, \langle p_0 \rangle, \ldots, \langle p_{n-1} \rangle \rangle.
\]  \hspace{1cm} (16.5)

In the first case, the 0 in the third position indicates that the state \( q \) is a push state, while the 1 in the third position indicates that the state \( q \) is a pop state in the second case.

For example, consider the DSM whose state diagram is pictured in Figure 16.1. The state \( q_0 \) is a pop state, and by following the prescription above, we encode the actions corresponding to this state by the binary string

\[
\langle 0, 0, 1, 0, 0, 1 \rangle = 00100101100100101.
\]  \hspace{1cm} (16.6)

The state \( q_1 \), on the other hand, is a push state, and the actions associated with this state are encoded as

\[
\langle 1, 0, 0, 10, 10 \rangle = 0110010010100100100.
\]  \hspace{1cm} (16.7)
These are the only two non-halting states of $M$, and therefore the transition function of this DSM is encoded as follows:

$$\langle \delta \rangle = \langle \langle 0, 0, 1, 0, 1 \rangle, \langle 1, 0, 10, 10 \rangle \rangle.$$  \hfill (16.8)

(As a binary string, this string’s length is about the text-width of this page; there’s not much point in writing it out explicitly.)

Aside from the transition function, we just need to specify these things (which are all represented by nonnegative integers) to complete the specification of a DSM $M$:

1. The number of stacks $r$.
2. The number of states $m$.
3. The number of input symbols $k$.
4. The number of stack symbols $n$.
5. Which state is the accept state.
6. Which state is the reject state.

The specific ordering we choose doesn’t really matter as long as we pick an ordering and stick to it, so let us decide that the encoding of the entire DSM $M$ is as follows:

$$\langle M \rangle = \langle \langle r \rangle, \langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \langle q_{\text{acc}} \rangle, \langle q_{\text{rej}} \rangle \rangle.$$  \hfill (16.9)

For example, the complete DSM $M$ illustrated in Figure 16.1 is encoded as

$$\langle M \rangle = \langle 1, 100, 10, 11, \langle \delta \rangle, 10, 11 \rangle,$$  \hfill (16.10)

where $\langle \delta \rangle$ is as in (16.8).

### 16.2 A universal stack machine

Now that we have defined an encoding scheme for DSMs, we can consider the computational task of simulating a given DSM on a given input. A universal stack machine is a stack machine that can perform such a simulation—it is universal in the sense that it is one single DSM that is capable of simulating all other DSMs.

Recall from Lecture 13 that a configuration of an $r$-DSM

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$  \hfill (16.11)

is a tuple

$$c = (q, x_0, \ldots, x_{r-1}) \in Q \times (\Gamma^*)^r.$$  \hfill (16.12)
Such a configuration indicates that the current state of $M$ is $q$ and that the contents of the $r$ stacks of $M$ are described by the strings $x_0, \ldots, x_{r-1}$. Assuming that $Q = \{q_0, \ldots, q_{m-1}\}$ and $\Gamma = \{0, \ldots, n-1\}$, there is an obvious way that such a configuration can be encoded into a binary string: simply take this string to be
\[ \langle c \rangle = \langle \langle q \rangle, \langle x_0 \rangle, \ldots, \langle x_{r-1} \rangle \rangle. \] (16.13)

As before, $\langle q \rangle$ is the encoding of the nonnegative integer index of $q$ using binary notation and $\langle x_0 \rangle, \ldots, \langle x_{r-1} \rangle$ are binary strings encoding the strings $x_0, \ldots, x_{r-1}$ using the method we have been discussing for the last two lectures.

Now, if we wish to simulate the computation of a given DSM $M$ on some input string $w$, a natural approach is to keep track of the configurations of $M$. Specifically, we will begin with the initial configuration of $M$ on input $w$, which is
\[ (q_0, w\diamond, \diamond, \ldots, \diamond), \] (16.14)
and then repeatedly compute the next configuration of $M$, over and over until perhaps we eventually reach a configuration whose state is $q_{\text{acc}}$ or $q_{\text{rej}}$, at which point we can stop. Of course, we might never reach such a configuration—if $M$ runs forever on input $w$, our simulation will also run forever. As it turns out (and as we will see later), there is no way to know whether or not the simulation will eventually stop, but this is OK—we’re just looking for a simulation that directly mimics $M$ on input $w$, including the possibility that the simulation runs forever when the same is true of $M$ on input $w$.

With this approach in mind, let us focus on the task of simply determining the next configuration, meaning the one that results from one computational step, for a given DSM $M$ and a given configuration of $M$. That is, we can focus on the function $f$ having the form
\[ f : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}^* \] (16.15)
that is defined as follows. For every DSM
\[ M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \] (16.16)
and every configuration
\[ c = (q, x_0, \ldots, x_{r-1}) \] (16.17)
of $M$, the function $f$ is defined so that
\[ f(\langle M \rangle, \langle c \rangle) = \langle d \rangle, \] (16.18)
where $d$ is the configuration obtained by running $M$ for one step starting from the configuration $c$ (as specified by Definition 13.2).
The function $f$ is actually not all that difficult to compute. For example, suppose that $c = (p, x_0, \ldots, x_{r-1})$, where $p$ happens to be a push state of $M$ that pushes $a$ onto stack number 0 and transitions to the state $q$. The function $f$ must then satisfy

$$f(M, c) = q, x_0, x_1, \ldots, x_{r-1}. \quad (16.19)$$

If instead it were the case that $p$ was a pop state of $M$ that, when $a$ is popped off of stack number 1, transitions to the state $q$, then we would have

$$f(M, c) = q, x_0, x_1, \ldots, x_{r-1}. \quad (16.20)$$

We also need to worry about the special case in which an empty stack is popped, and for any input to $f$ that does not take the form $(M, c)$ for $M$ being a DSM and $c$ a valid configuration of $M$, we could simply define the output of $f$ to be $\varepsilon$ (which is an arbitrary choice that doesn’t really matter for the purposes of the simulation). It is also convenient to define

$$f(M, c) = c \quad (16.21)$$

whenever $c$ is a halting configuration of $M$.

The difficulty in computing the function $f$ is, naturally, that one needs to examine the encoding $\langle M \rangle$ in order to determine how the encoding $\langle c \rangle$ of each configuration is to be transformed into the encoding $\langle d \rangle$ of the configuration that results from running $M$ for one step. It would be a time-consuming process to explicitly describe a DSM that computes $f$, but at a conceptual level it would not be unreasonable to describe this task as being fairly straightforward. If we were to do this carefully, perhaps we would start by defining a subroutine that searches through the encoding $\langle \delta \rangle$ of the transition function of $M$ to find the instruction corresponding to a given state, as well as a subroutine that applies a given instruction to a given configuration. Here is a rather high-level description how the required computation might be performed:

1. Test to see that $x = \langle M \rangle$ and $y = \langle c \rangle$ for some choice of a DSM $M$ and a valid configuration $c$ of $M$. Output $\varepsilon$ and halt if this is not the case.
2. Check if $c$ is a halting configuration of $M$. Output $\langle c \rangle$ and halt if this is the case.
3. Supposing that $c = (q, x_0, \ldots, x_{r-1})$ for $q$ being a non-halting state of $M$, process the encoding $\langle \delta \rangle$ of the transition function of $M$ to obtain the instructions corresponding to the state $q$ of $M$. This will be a string of the form

$$\langle q, s, 0, a, \langle p \rangle \rangle \quad (16.22)$$
Lecture 16

The DSM $U$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle \langle M \rangle, \langle w \rangle \rangle$ for $M$ being a DSM and $w$ being a string over the alphabet of $M$, then reject.

2. Set $Y \leftarrow \langle M \rangle$ and $Z \leftarrow \langle c \rangle$, for $c$ being the initial configuration of $M$ on input $w$.

3. Accept if $Z$ stores an accepting configuration of $M$ and reject if $Z$ stores a rejecting configuration of $M$. (If $Z$ stores a non-halting configuration of $M$, then continue to the next step.)

4. Compute $Z \leftarrow f(Y, Z)$, where $f$ is the next-configuration function described previously, and goto step 3.

Figure 16.2: A high-level description of a DSM $U$ that recognizes the language $A_{DSM}$.

if $q$ is a push state, or a string of the form

$$\langle \langle q \rangle, \langle s \rangle, 1, \langle p_0 \rangle, \ldots, \langle p_{n-1} \rangle \rangle$$  \hspace{1cm} (16.23)

if $q$ is a pop state.

4. Modify $\langle c \rangle$ according to the instructions corresponding to the state $q$ obtained in the previous step to obtain the encoding $\langle d \rangle$ that results from running $M$ for one step on configuration $c$. Output $\langle d \rangle$ and halt.

With the function $f$ in hand, one can simulate the computation of a given DSM $M$ on a given input $w$ in the manner suggested above, by starting with the initial configuration of $M$ on $w$ and repeatedly applying $f$.

Now consider the following language, which is the natural DSM analogue of the languages $A_{DFA}$, $A_{NFA}$, $A_{REG}$, and $A_{CFG}$ discussed in the previous lecture:

$$A_{DSM} = \{ \langle \langle M \rangle, \langle w \rangle \rangle : M \text{ is a DSM and } w \in L(M) \}.$$  \hspace{1cm} (16.24)

We conclude that $A_{DSM}$ is semidecidable: the DSM $U$ described in Figure 16.2 is such that $L(U) = A_{DSM}$. This DSM has been named $U$ to reflect the fact that it is a universal DSM.

**Proposition 16.1.** The language $A_{DSM}$ is semidecidable.
16.3 A non-semidecidable language

It is natural at this point to ask whether or not $A_{DSM}$ is decidable, given that it is semidecidable. It is not decidable, as we will soon prove. Before doing this, however, we will consider a different language and prove that this language is not even semidecidable. Here is the language:

$$\text{DIAG} = \{ \langle M \rangle : M \text{ is a DSM and } \langle M \rangle \not\in L(M) \}.$$  \hspace{1cm} (16.25)

That is, the language DIAG contains all binary strings $\langle M \rangle$ that, with respect to the encoding scheme we discussed at the start of the lecture, encode a DSM $M$ that does not accept this encoding of itself. (Note that if it so happens that the string $\langle M \rangle$ encodes a DSM whose input alphabet has just one symbol, then it will indeed be the case that $\langle M \rangle \not\in L(M)$.)

**Theorem 16.2.** The language DIAG is not semidecidable.

**Proof.** Assume toward contradiction that the language DIAG is semidecidable. There must therefore exist a DSM $M$ such that $L(M) = \text{DIAG}$.

Now, consider the encoding $\langle M \rangle$ of $M$. By the definition of the language DIAG one has

$$\langle M \rangle \in \text{DIAG} \iff \langle M \rangle \not\in L(M).$$  \hspace{1cm} (16.26)

On the other hand, because $M$ recognizes DIAG, it is the case that

$$\langle M \rangle \in \text{DIAG} \iff \langle M \rangle \in L(M).$$  \hspace{1cm} (16.27)

Consequently,

$$\langle M \rangle \not\in L(M) \iff \langle M \rangle \in L(M),$$  \hspace{1cm} (16.28)

which is a contradiction. We conclude that DIAG is not semidecidable. \qed

**Remark 16.3.** Note that this proof is very similar to the proof that $P(\mathbb{N})$ is not countable from the very first lecture of the course. It is remarkable how simple this proof of the non-semidecidability of DIAG is; it has used essentially none of the specifics of the DSM model or the encoding scheme we defined.

Now that we know DIAG is not semidecidable, we prove that $A_{DSM}$ is not decidable.

**Proposition 16.4.** The language $A_{DSM}$ is undecidable.
The DSM $K$ operates as follows on input $x \in \{0, 1\}^*$:

1. If it is not the case that $x = \langle M \rangle$ for $M$ being a DSM, then reject.
2. Run $T$ on input $\langle \langle M \rangle, \langle M \rangle \rangle$. If $T$ accepts, then reject, otherwise accept.

**Proof.** Assume toward contradiction that $A_{DSM}$ is decidable. There must therefore exist a DSM $T$ that decides $A_{DSM}$. Define a new DSM $K$ as described in Figure 16.3.

For a given DSM $M$, we may now ask ourselves what $K$ does on the input $\langle M \rangle$. If it is the case that $\langle M \rangle \in \text{DIAG}$, then by the definition of DIAG it is the case that $\langle M \rangle \not\in L(M)$, and therefore $\langle \langle M \rangle, \langle M \rangle \rangle \not\in A_{DSM}$ (because $M$ does not accept $\langle M \rangle$). This implies that $T$ rejects the input $\langle \langle M \rangle, \langle M \rangle \rangle$, and so $K$ must accept the input $\langle M \rangle$. If, on the other hand, it is the case that $\langle M \rangle \not\in \text{DIAG}$, then $\langle M \rangle \in L(M)$, and therefore $\langle \langle M \rangle, \langle M \rangle \rangle \in A_{DSM}$. This implies that $T$ accepts the input $\langle \langle M \rangle, \langle M \rangle \rangle$, and so $K$ must reject the input $\langle M \rangle$. One final possibility is that $K$ is run on an input string that does not encode a DSM at all, and in this case it rejects.

Considering these possibilities, we find that $K$ decides DIAG. This, however, is in contradiction with the fact that DIAG is non-semidecidable (and is therefore undecidable). Having obtained a contradiction, we conclude that $A_{DSM}$ is undecidable, as required.  

\[\Box\]