Decidable and Undecidable Languages

The Halting Problem and
The Return of Diagonalization

Friday, November 11 and Tuesday, November 15, 2011
Reading: Sipser 4; Kozen 31; Stoughton 5.2 & 5.3

CS235 Languages and Automata
Department of Computer Science
Wellesley College

Decidability and Semi-Decidability

A Turing Machine decides a language if it rejects every string it doesn’t accept — i.e., it never loops.

The recursive languages = the set of all languages that are decided by some Turing Machine = all languages described by a non-looping TM.

These are also called the Turing-decidable or decidable languages.

We will use Dec to name this set.

We’ll soon see examples of languages that are in RE but not in Dec. We call these languages semi-decidable+ (Lyn’s nonstandard terminology)

Every TM for a semi-decidable+ language halts in the accept state for strings in the language but loops for some strings not in the language.

Any language outside Dec is undecidable. All semi-decidable+ languages are undecidable, but we’ll see there are undecidable languages that aren’t semi-decidable+

Recursively Enumerable Languages

L(M) = {w | w is accepted by the Turing Machine M}

The recursively enumerable (r.e.) languages = the set of all languages that are the language of some Turing Machine.

These are also called Turing-acceptable and Turing-recognizable languages.

We will use RE to name this set.

There are many languages in RE that are not in CFL.

Dec vs. RE

For every language L in Dec, there is a deciding machine M that for an input string w is guaranteed to deliver a ball to either the accept pipe or reject pipe.

For every language L in RE, there is an accepting machine M that for an input string w is guaranteed to deliver a ball to the accept pipe if w ∈ L. However, if w ∉ L, a ball might not be delivered to the reject pipe (M might loop).
Warm-Up: Some Decidable Languages

Show that the following languages are decidable by describing (at a high level) an algorithm that decides them (see more in Sipser 4.1)

- \( \text{ACCEPT}_{\text{DFA}} = \{ <\text{DFA}>, w \mid w \in L(\text{DFA}) \} \)
- \( \text{EMPTY}_{\text{DFA}} = \{ \text{DFA} \mid L(\text{DFA}) = \emptyset \} \)
- \( \text{ALL}_{\text{DFA}} = \{ \text{DFA} \mid L(\text{DFA}) = \Sigma_{\text{DFA}}^* \} \)
- \( \text{EQ}_{\text{DFA}} = \{ (<\text{DFA}_1>, <\text{DFA}_2>) \mid L(\text{DFA}_1) = L(\text{DFA}_2) \} \)
- \( \text{ACCEPT}_{\text{CFG}} = \{ <\text{CFG}>, w \mid w \in L(\text{CFG}) \} \)
- \( \text{EMPTY}_{\text{CFG}} = \{ <\text{CFG} \mid L(\text{CFG}) = \emptyset \} \)

(Warning: \( \text{EQ}_{\text{CFG}} \) and \( \text{ALL}_{\text{CFG}} \) are not decidable!)

What about \( \text{ACCEPT}_{\text{TM}}? \)

\( \text{ACCEPT}_{\text{TM}} = \{ (<M>, w) \mid w \in L(M) \} \) (i.e. \( M \) accepts \( w \))

A Turing Machine \( M_{\text{UTM}} \) can accept \( \text{ACCEPT}_{\text{TM}} \) as follows:

1. Check if \( <M> \) is a well-formed Turing Machine description. If not, \( M_{\text{UTM}} \) rejects \( (<M>, w) \)
2. If \( <M> \) is well-formed, \( M_{\text{UTM}} \) simulates the running of \( M \) on \( w \), e.g., via a 4-tape machine:
   - Tape 1 holds \( <M> \).
   - Tape 2 is the tape \( M \) works on (initially \( w \)).
   - Tape 3 holds the head position for tape 2.
   - Tape 4 holds the state of \( M \).

   In this case, \( M_{\text{UTM}} \) will:
   - accept \( (<M>, w) \) iff \( M \) accepts \( w \).
   - reject \( (<M>, w) \) iff \( M \) rejects \( w \).
   - loop iff \( M \) loops on \( w \).

So \( \text{ACCEPT}_{\text{TM}} \) is in \( \text{RE} \).

Big question: is \( \text{ACCEPT}_{\text{TM}} \) in \( \text{Dec} \)? i.e., is there another Turing Machine that decides \( \text{ACCEPT}_{\text{TM}} \)? (We'll see the answer is no.)
\( M_{UTM} \) is a Universal Turing Machine!

\( M_{UTM} \) is a universal Turing Machine

\( = \) a Turing Machine interpreter written as a Turing Machine.

There’s nothing strange about this:

- We can write an SML interpreter in any language, including SML.
- We can write a Java compiler in any language, including Java.
- Why not write a Turing Machine interpreter as a Turing Machine?

The tricky bit is bootstrapping (take CS251 for more details):

- Our first SML interpreter can’t be written in SML;
- Our first Java compiler can’t be written in Java.

\section*{Self-Reference is not a Problem}

Consider the following:

- A decommenting program can decomment any text file, including the decommenting program itself.
- A Java compiler can compile any Java program, including one that specifies a Java compiler.
- An SML interpreter can evaluate any SML program, including one that specifies an SML interpreter.

\textit{Moral:} There is nothing inherently problematic about a program being called on “itself”. The program supplied as argument is just data, and the running program \( P \) doesn’t “know” that this data describes \( P \)!

You can’t eat yourself, but you can eat a description of yourself!
What does \( M_{UTM} \) do on the input \( (< M_{UTM}>, (< M >, w) ) \) ?

\section*{The Halting Problem}

\( \text{HALT}_{TM} = \{ (\langle M \rangle, w) | M \text{ halts on } w \text{ (i.e. } M \text{ decides } w, \text{ cannot loop)} \} \)

Is \( \text{HALT}_{TM} \) in \( \text{RE} \) (Turing-acceptable)?

Yes: Simulate \( M \) running on \( w \). Accept if \( M \) accepts or rejects \( w \).
Loop if \( M \) loops on \( w \).

Is \( \text{HALT}_{TM} \) in \( \text{Dec} \) (decidable)?

No! We’ll show this by diagonalization. Intuitively, the problem is that no TM for \( \text{HALT}_{TM} \) can always reject \( \langle M \rangle, w \) when \( M \) loops on \( w \).

So \( \text{HALT}_{TM} \) is semi-decidable+; our first example of such a language.

In the next lecture, we’ll see that we can use \( \text{HALT}_{TM} \) to show that other languages are semi-decidable+, including \( \text{ACCEPT}_{TM} \).

\section*{Behavior of Turing Machines}

The behavior of all Turing machines can be summarized by an infinite 2D table whose rows are Turing machines and whose columns are input strings. A table entry is A (accept), R (reject), or L (loop).

Because TM descriptions are countable, TMs are enumerable in a sequence \( M_1, M_2, M_3, \ldots \)

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
input strings & % & a & z & aa & zz \\
\hline
\text{Turing Machines} & M_1 & A & R & ... & A & L & ... & R & ... \\
& M_2 & R & L & ... & R & R & ... & R & ... \\
& M_3 & A & A & ... & R & L & ... & A & ... \\
& M_4 & L & L & ... & A & A & ... & R & ... \\
& M_5 & R & A & ... & R & A & ... & L & ... \\
& ... & ... & ... & ... & ... & ... & ... & ... & ...
\hline
\end{tabular}
\end{center}
Towards Diagonalization

With diagonalization in mind, we focus on the subtable that results from keeping only those columns in whose input strings are valid TM descriptions.

<table>
<thead>
<tr>
<th>Turing Machines</th>
<th>inputs that are TM descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁</td>
<td>&lt;M₁&gt;</td>
</tr>
<tr>
<td>M₂</td>
<td>&lt;M₂&gt;</td>
</tr>
<tr>
<td>M₃</td>
<td>&lt;M₃&gt;</td>
</tr>
<tr>
<td>M₄</td>
<td>&lt;M₄&gt;</td>
</tr>
<tr>
<td>M₅</td>
<td>&lt;M₅&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>M_DIAG</td>
<td>&lt;M_DIAG&gt;</td>
</tr>
</tbody>
</table>

Suppose \( \text{HALT}_{TM} \) Is Decidable

If \( \text{HALT}_{TM} \) is decidable, then there is a Turing Machine \( M_{\text{HALT}} \) that, for all inputs \( (<M>,w) \), decides if \( M \) halts on \( w \).

<table>
<thead>
<tr>
<th>Turing Machines</th>
<th>inputs that are TM descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁</td>
<td>&lt;M₁&gt;</td>
</tr>
<tr>
<td>M₂</td>
<td>&lt;M₂&gt;</td>
</tr>
<tr>
<td>M₃</td>
<td>&lt;M₃&gt;</td>
</tr>
<tr>
<td>M₄</td>
<td>&lt;M₄&gt;</td>
</tr>
<tr>
<td>M₅</td>
<td>&lt;M₅&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>M_DIAG</td>
<td>&lt;M_DIAG&gt;</td>
</tr>
</tbody>
</table>

Alternative Diagonalization for \( \text{HALT}_{TM} \)

We can instead perform the diagonalization on the behavior table for \( M_{\text{HALT}} \). If \( M_{\text{HALT}} \) exists, then there is a machine \( M_{\text{DIAG}} \) defined as:
\[ M_{\text{DIAG}}(<M>) = \begin{cases} \text{HALT} & \text{if } M_{\text{HALT}}((<M>,<M>)) \text{ returns } \text{HALT} \\ \text{accept} & \text{otherwise} \end{cases} \]

<table>
<thead>
<tr>
<th>Turing Machines</th>
<th>inputs that are TM descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁</td>
<td>&lt;M₁&gt;</td>
</tr>
<tr>
<td>M₂</td>
<td>&lt;M₂&gt;</td>
</tr>
<tr>
<td>M₃</td>
<td>&lt;M₃&gt;</td>
</tr>
<tr>
<td>M₄</td>
<td>&lt;M₄&gt;</td>
</tr>
<tr>
<td>M₅</td>
<td>&lt;M₅&gt;</td>
</tr>
<tr>
<td>&lt;M_DIAG&gt;</td>
<td>&lt;M_DIAG&gt;</td>
</tr>
</tbody>
</table>
The Halting Problem in Scheme*

Suppose we could write a function `halts?` that determines if an input function `f` halts when applied to its argument `f`:

```
(define (halts? f x) ...)
```

Then we could write the following:

```
(define (loop) (loop))
```

```
(define (diag f)  (if (halts? f f) (loop) #t))
```

Suppose `(diag diag)` halts. Then it should loop!

Suppose `(diag diag)` loops. Then it should halt with `#t`!

This accurately captures the diagonalization dilemma, but is a bit hokey since a Scheme `halts?` function can't examine the structure of the input function in the way that a Turing Machine `M_{HALT}` can examine the structure of a TM description.

* It's tougher to show this in SML because its type system prohibits self-application.

---

So What? Does the Halting Problem Matter?

Yes! Compilers and other programs that analyze programs often want to perform termination analysis to determine whether or not the evaluation of a particular subexpression will terminate.

E.g., it's often helpful to evaluate a subexpression, but only safe to do so if evaluation will terminate (otherwise, the compiler might not terminate!)

Sadly, the halting problem says there is no iron-clad way to determine in advance whether or not subexpression evaluation will terminate.

We'll see later that various heuristics can be used, but they can only conservatively approximate exact termination analysis.

We'll also see that most forms of program analysis suffer from the same problems as termination analysis.

---

With Great Power Comes Great Uncomputability

Turing machines and equivalent models of computation (lambda calculus, Java, SML, etc.) are far more powerful than finite automata and pushdown automata.

But the power is gained via features that can cause programs to loop infinitely. If we want the power, we must live with the looping.

---

Programs that loop vs. taking a long time

How do we distinguish programs that run a long time from ones that loop?

E.g. 3x+1 problem:

```
f(x) = \begin{cases} 
3x + 1, & \text{if } x \text{ is odd} \\
\frac{x}{2}, & \text{if } x \text{ is even}
\end{cases}
```

Problem: for all `n`, is there some `i` s.t. `f^n(n) = 1`? I.e., is it the case that iterating `f` at a starting point never loops?

No one knows! This is an open problem!
Are there non-RE Languages?

Our next big question: are there any languages that are not RE?

We'll see the answer is yes. In fact, way yes (lots of them!)

- Is there any language out here?

RE = Recursively Enumerable (Turing-Recognizable/Acceptable) Languages
HALT_{TM} * semi-decidable*
Dec = Recursive (Turing-Decidable) Languages
a^b^c^ a^b^n w^* * decidable
CFL = Context-Free Languages
a^b^n w^a^n
Reg = Regular Languages
a^b^* (a+b)^* b^*b(a+b)^*

RE is NOT Closed Under Complement

Suppose \( M \) accepts \( L \) and \( \overline{M} \) accepts \( \overline{L} \).
Then \( L \) is decidable (by \( M' \) below!)

Important detail: \( M \) and \( \overline{M} \) must be run in parallel, not sequentially!
See next slide.

So if \( L \) is semi-decidable* (\( L \) in RE - Dec), then \( L \) can't be in RE!

Dec is Closed Under Complement

Suppose \( M \) is a Turing Machine that decides \( L \).
We can construct a machine \( \overline{M} \) that decides \( \overline{L} \):

Importance: We often want to run two accepting machines \( M_1 \) and \( M_2 \) on the same or different inputs.
The machines should not be run sequentially (say \( M_1 \) before \( M_2 \)) because if \( M_1 \) loops, \( M_2 \) will never run.
Instead, the machines are run in parallel by performing one step of \( M_1 \) followed by one step of \( M_2 \), alternating between the two machines.
After each step, we can check whether either machine is in its accept state or reject state. So it's possible to run \( M_1 \) and \( M_2 \) in parallel until one accepts, both accept, one rejects, or both reject.
Complementary, My Dear Watson

co-RE is the set of languages whose complements are RE.

If \( L \) is semi-decidable+ (in \( RE \rightarrow Dec \)), then \( L \) is semi-decidable- (in co-RE \( \rightarrow Dec \)).

We'll say \( L \) is semi-decidable if it's semi-decidable+ or semi-decidable-.

Recursive (Turing-Decidable) Languages

- \( L_{semi+} \)
- \( L_{semi-} \)
- \( HALT_{TM} \)
- \( Dec \)
- \( RE \)

Decision (Turing-Decidable) Languages

- \( L_{semi+} \)
- \( L_{semi-} \)
- \( HALT_{TM} \)
- \( Dec \)
- \( RE \)

Recursive with RE Complements

- semi-decidable-

We can enumerate all possible Turing Machine descriptions, so \( RE \) and co-RE must be countable.

But we know that \( Lan = P(\Sigma^*) \) is an uncountable set.

So \( Lan - (RE \cup co-RE) \) must contain many languages. We'll call these languages not-even-semidecidable. We'll see concrete examples of these next lecture.

For an not-even-semidecidable language, every Turing Machine must loop for some strings in the language and some not in the language.

Decidable and Undecidable Languages

What We're Aiming For

- \( EQ_{TM} \)
- \( HALT_{TM} \)
- \( ACCEPT_{TM} \)
- \( RE \)
- \( Dec \)

Note: undecidable = semi-decidable+ U semi-decidable- U not-even-semidecidable

Closure Properties of Dec and RE

Dec is closed under:
- union
- intersection
- concatenation
- Kleene star
- complement

RE is closed under:
- union
- intersection
- concatenation
- Kleene star

We've already seen the complement story:
- \( L \) in Dec implies \( \overline{L} \) in Dec.
- \( L \) and \( \overline{L} \) are in RE implies \( L \) and \( \overline{L} \) in Dec.
- \( L \) is semi-decidable+ (in \( RE \rightarrow Dec \)) and \( \overline{L} \) semi-decidable- (in co-RE \( \rightarrow Dec \))

Now we'll study the other properties.
**RE is Closed Under Union**

Suppose $L_1, L_2$ are accepted by machines $M_1, M_2$, respectively. $L_1 \cup L_2$ is accepted by the following machine $M_\cup$, which runs $M_1$ and $M_2$ in parallel until one accepts or both reject.

It is essential to run $M_1$ and $M_2$ in parallel. Why?

A similar diagram shows Dec is closed under union, but in that case $M_1$ and $M_2$ can be run sequentially. Why?

Similar arguments (PS10) show RE, Dec are closed under intersection.

---

**RE is Closed Under Concatenation**

Suppose $L_1, L_2$ are accepted by machines $M_1, M_2$, respectively. $L_1 @ L_2$ is accepted by the following machine $M_\oplus$, which runs copies of $M_1$ and $M_2$ in parallel on all possible decompositions of $w$ into pairs of substrings until one pair accepts or all reject.

This diagram can be implemented by a Turing Machine program that loops over all possible decompositions, interleaving the steps for each.

Similar ideas show that Dec is closed under concatenation and that both RE and Dec are closed under Kleene star.