Chapter 4 Lecture Notes (Section 4.2: The “Halting” Problem)

David Martin
dm@cs.uml.edu

With modifications by Prof. Karen Daniels, Fall 2010

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.
Back to $\Sigma_1$

- So the fact that $\Sigma_1$ is not closed under complement means that there exists some language $L$ that is not recognizable by any TM.

- By Church-Turing thesis this means that no imaginable finite computer, even with infinite memory, could recognize this language $L$!
Non-recognizable languages

- We proceed to prove that non-Turing recognizable languages exist, in two ways:
  - A nonconstructive proof using Georg Cantor’s famous 1873 diagonalization technique, and then
  - An explicit construction of such a language.
A nonconstructive proof

Let $L \subseteq \{0,1\}^*$ be defined by:

$$L = \begin{cases} 
0^* \text{ if Hillary Clinton is president on February 1, 2013} \\
1^* \text{ otherwise}
\end{cases}$$

Is $L$ decidable?

No; there exists a machine $M$ that recognizes the appropriate language; we just don’t know what machine it is right now.
Learning how to count

- **Definition**: Let A and B be sets. Then we write $A \approx B$ and say that A is **equinumerous** to B if there exists a one-to-one, onto function (a “correspondence”) $f: A \rightarrow B$

- Note that this is a purely mathematical definition: the function $f$ does not have to be expressible by a Turing machine or anything like that.

- **Example**: $\{1, 3, 2\} \approx \{\text{six, seven, BBCCD}\}$

- **Example**: $\mathbb{N} \approx \mathbb{Q}$ (textbook example 4.15)
  - See next slide...
Learning how to count (continued)

Example: \( N \approx Q \) (textbook example 4.15)

\[ \begin{array}{cccccc}
1 & 1 \\
2 & 2 \\
3 & 3 \\
4 & 4 \\
5 & 5 \\
\end{array} \]

Source: Sipser textbook
Countability

- **Definition** A set $S$ is **countable** if $S$ is finite or $S \approx \mathbb{N}$.

- Saying that $S$ is countable means that you can line up all of its elements, one after another, and cover them all.

- Note that $\mathbb{R}$ is *not* countable (Theorem 4.17), basically because choosing a single real number requires making infinitely many choices of what each digit in it is (see next slide).
Countability (continued)

- **Theorem 4.17**: \( \mathbb{R} \) is not countable.

- **Proof Sketch**: By way of contradiction, suppose \( \mathbb{R} \approx \mathbb{N} \) using correspondence \( f \).
  
  Construct \( x \in \mathbb{R} \) such that \( x \) is not paired with anything in \( \mathbb{N} \), providing a contradiction.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( f(n) )</th>
<th>( x \in (0,1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14159...</td>
<td>( x ) is not ( f(n) ) for any ( n ) because it differs from ( f(n) ) in ( n )th fractional digit.</td>
</tr>
<tr>
<td>2</td>
<td>5.55555...</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.12345...</td>
<td>( x = 0.4641... )</td>
</tr>
<tr>
<td>4</td>
<td>0.50000...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Caveat: How to circumvent \( 0.1999... = 0.2000... \) problem?

Source: Sipser textbook
A non-$\Sigma_1$ language

Each point is a language in this Venn diagram.
Strategy

- We’ll show that there are more (a lot more) languages in ALL than there are in $\Sigma_1$
  - Namely, that $\Sigma_1$ is countable but ALL isn’t countable
  - Which implies that $\Sigma_1 \neq$ ALL
  - Which implies that there exists some $L$ that is not in $\Sigma_1$

- For simplicity and concreteness, we’ll work in the universe of strings over the alphabet $\{0,1\}$. 
Countability of $\Sigma_1$

- **Theorem** $\Sigma_1$ is countable
- **Proof** The strategy is simple. $\Sigma_1$ is the class of all languages that are Turing-recognizable. So each one has (at least) one TM that recognizes it. We’ll concentrate on listing those TMs.
Countability of TM

- Let $\text{TM} = \{ <M> \mid M \text{ is a Turing Machine with } \Sigma = \{0,1\} \}$
  - Notation: $<M>$ means the string encoding of the object $M$
  - Previously, we thought of our TMs as abstract mathematical things: drawings on the board, or 7-tuples: $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$
  - But just as we can encode every C++ program as an ASCII string, surely we can also encode every TM as a string
  - It’s not hard to specify precisely how to do it—but it doesn’t help us much either, so we won’t bother
  - Just note that in our full specification of a TM $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, each element in the list is finite by definition
  - So writing down the sequence of 7 things can be done in a finite amount of text
- In other words, each $<M>$ is a string
Countability of TM

- Now we make a list of all possible strings in lexicographical order,
- Cross out the ones that are not valid encodings of Turing Machines,
- And we have a mapping \( f : \mathbb{N} \to \text{TM} \)
  - \( f(1) = \) first (smallest) TM encoding on list
  - \( f(2) = \) second TM encoding on list
  - ... 
- This is part of textbook’s proof of Corollary 4.18 (Some languages are not Turing-recognizable).
Back to countability of $\Sigma_1$

- Now consider the list $L(f(1)), L(f(2)), \ldots$
  - Turns each TM enumerated by $f$ into a language
  - So we can define a function $g : \mathbb{N} \rightarrow \Sigma_1$ by $g(i) = L(f(i))$, where $f(i)$ returns the $i^{\text{th}}$ Turing machine
  - Now: is this a correspondence? Namely,
    - Is it onto?
    - Is it one-to-one?
Fixing \( g : \mathbb{N} \rightarrow \Sigma_1 \)

- Go ahead and make the list \( g(1), g(2), \ldots \)
- But cross out each element that is a repeat, removing it from the list
- Then let \( h : \mathbb{N} \rightarrow \Sigma_1 \) be defined by
  \[ h(i) = \text{the } i^{\text{th}} \text{ element on the reduced list} \]
- Then \( h \) is both one-to-one and onto
- **Thus \( \Sigma_1 \) is countable**
What about ALL?

- **Theorem** (Cantor, 1873) For every set \( A \), \( A \not\subseteq \mathcal{P}(A) \)
  - See next several slides for proof.
  - See textbook for a different way to show ALL is uncountable using characteristic sequence associated with (uncountable) set of all infinite binary sequences.

- Remember ALL = \( \mathcal{P}(\{0,1\}^*) \)
  - set of all (languages) = set of all (subsets of \( \{0,1\}^* \))

- Note that \( \{0,1\}^* \) is countable
  - Just list all of the strings in lexicographical order

- **Corollary to Theorem** ALL = \( \mathcal{P}(\{0,1\}^*) \) is uncountable
  - So \( \Sigma_1 \) is countable but ALL isn’t
  - So they're not equal
Cantor’s Theorem

Theorem For every set A,  \( A \nsubseteq \mathcal{P}(A) \)

Proof We’ll show by contradiction that no function \( f: A \rightarrow \mathcal{P}(A) \) is onto. So suppose \( f: A \rightarrow \mathcal{P}(A) \) is onto. We define a set \( K \subseteq A \) in terms of it:

\[
K = \{ x \in A \mid x \notin f(x) \}
\]

Since \( K \subseteq A \), \( K \in \mathcal{P}(A) \) as well (by definition of \( \mathcal{P} \)). Since \( f \) is onto, there exists some \( z \in A \) such that \( f(z) = K \). Looking closer,

Case 1: If \( z \in K \) then \( z \notin f(z) \) then \( z \notin K \)

by definition of \( K \)

by definition of \( z \)

so \( z \in K \) certainly can’t be true...
Cantor’s Theorem

\[
K = \{ x \in A \mid x \notin f(x) \}
\]

unchanged

\[
\begin{align*}
K & \in \mathcal{P}(A) \\
z & \in A \text{ and } f(z) = K
\end{align*}
\]

On the other hand,

Case 2: If \( z \notin K \) \( \Rightarrow \) \( z \in f(z) \) \( \Rightarrow \) \( z \in K \)

by definition of \( K \)

by definition of \( z \)

so \( z \notin K \) can’t be true either! \quad \text{QED}
Cantor’s Theorem: Example

- For every proposed $f : A \to \mathcal{P}(A)$, the theorem constructs a set $K \in \mathcal{P}(A)$ that is not $f(x)$ for any $x$

- Let $A = \{1, 2, 3\}$
  $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$

- Propose $f : A \to \mathcal{P}(A)$, show $K$
Diagonalization

- All we’re really doing is identifying the squares on the diagonal and making them different than what’s in our set K
- So that we’re guaranteed K ≠ f(1), K ≠ f(2), …
- The construction works for infinite sets too

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ _ , _ , _ }</td>
</tr>
<tr>
<td>2</td>
<td>{ _ , _ , □, _ }</td>
</tr>
<tr>
<td>3</td>
<td>{ _ , _ , _ , □ }</td>
</tr>
</tbody>
</table>
Non-recognizable languages

- So we conclude that there exists some $L \in \text{ALL} - \Sigma_1$ (many such languages)
- But we don’t know what any $L$ looks like exactly
- Turing constructed such an $L$ also using diagonalization (but not the relation)
- We now turn our attention to it
Programs that process programs

- In §4.1, we considered languages such as
  \[ A_{\text{CFG}} = \{ <G,w> \mid G \text{ is a CFG and } w \in L(G) \} \]

- Each element of \( A_{\text{CFG}} \) is a *coded pair*
  - Meaning that the grammar \( G \) is encoded as a string *and*
  - \( w \) is an arbitrary string *and*
  - \( <G,w> \) contains both pieces, in order, in such a way that the two pieces can be easily extracted

- The question “does grammar \( G_1 \) generate the string 00010?” can then be phrased equivalently as:
  - Is \( <G_1,00010> \in A_{\text{CFG}}? \)
Programs that process programs

- \( A_{\text{CFG}} = \{ <G,w> \mid G \text{ is a CFG and } w \in L(G) \} \)

- The *language* \( A_{\text{CFG}} \) somehow represents the question “does *this* grammar accept *that* string?”

- Additionally we can ask: is \( A_{\text{CFG}} \) itself a regular language? context free? decidable? recognizable?
  - We showed previously that \( A_{\text{CFG}} \) is decidable (as is almost everything similar in §4.1)
$A_{TM}$ and the Universal TM

- $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in L(M) \}$
- We will show that $A_{TM} \in \Sigma_1 - \Sigma_0$
  - (It’s recognizable but not decidable)
- **Theorem** $A_{TM}$ is Turing-recognized by a fixed TM called $U$ (the Universal TM)
  - This is not stated as a theorem in the textbook (it does appear as part of proof of **Theorem 4.11: $A_{TM}$ is undecidable**), but should be: it’s really important
$A_{TM} = L(U)$

$A_{TM} = \{ <M,w> \mid M \text{ is a TM and } w \in L(M) \}$

U is a 3-tape TM that keeps data like this:

1. $<M>$  
   never changes
2. $q$  
   a state name
3. $c_1 c_2 c_3 \ldots$  
   tape contents & head pos

On startup, U receives input $<M,w>$ and writes $<M>$ onto tape 1 and $w$ onto tape 3. (If the input is not of the form $<M,w>$, then U rejects it.) From $<M>$, U can extract the encoded pieces $(Q,\Sigma,\Gamma,\delta,q_0,q_{acc},q_{rej})$ at will. It continues by extracting and writing $q_0$ onto tape 2.
\[ A_{TM} = L \left( U \right) \]

\[ A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in L(M) \} \]

1. \(< M >\)  \(\text{never changes}\)
2. \(q\)  \(\text{a state name}\)
3. \(c_1 \ c_2 \ c_3 \ldots\)  \(\text{tape contents \& head pos}\)

To simulate a single computation step, \(U\) fetches the current character \(c\) from tape 3, the current state \(q\) on tape 2, and looks up the value of \(\delta(q,c)\) on tape 1, obtaining a new state name, a new character to write, and a direction to move. \(U\) writes these on tapes 2 and 3 respectively.

If the new state is \(q_{acc}\) or \(q_{rej}\) then \(U\) accepts or rejects, respectively. Otherwise it continues with the next computation step.
The Universal TM U

- This U is **hugely important**: it’s the theoretical basis for *programmable* computers.
- It says that there is a *fixed* machine U that can take computer programs as *input* and behave just like each of those programs
  - Note that U is **not** a decider
  - See VMware
- Since $A_{TM} = L(U)$, we have shown that $A_{TM}$ is Turing-recognizable ($\Sigma_1$)
The “Halting” Problem

- $A_{TM} = \{<M,w> | M \text{ is a TM and } w \in L(M)\}$

- This appears in our textbook as:
  - $A_{TM} = \{<M,w> | M \text{ is a TM and } M \text{ accepts } w\}$
  - This emphasizes the fact that $U$ might loop (i.e. might not halt) on input $<M,w>$.
  - $A_{TM}$ is therefore sometimes called the halting problem.
  - We use "" here due to Chapter 5’s discussion...

- $A_{TM}$ is called the acceptance problem in Chapter 5
- The “real” halting problem is defined there as:
  - $HALT_{TM} = \{<M,w> | M \text{ is a TM and } M \text{ halts on input } w\}$
$A_{TM}$ is undecidable

**Theorem 4.11** (Turing) $A_{TM} \notin \Sigma_0$

**Proof** Suppose that $A_{TM} = L(H)$ where $H$ is a decider. We’ll show that this leads to a contradiction.

Let $D$ be a TM that behaves as follows:

1. Input $x$
2. If $x$ is not of the form $<M>$ for some TM $M$, then $D$ rejects
3. Simulate $H$ on input $<M, <M>>$
   - If $H$ accepts $<M, <M>>$, then $D$ rejects
   - If $H$ rejects $<M, <M>>$, then $D$ accepts

$$H(<M, w>) = \begin{cases} 
\text{accept} & \text{if } M \text{ accepts } w \\
\text{reject} & \text{if } M \text{ does not accept } w
\end{cases}$$
“Simulate H”

- Steps 1 and 2 are not so hard to imagine
- How does D “simulate H on (some other input)”?
  - If someone creates an H, we follow this outline to build D — which has the entire H program built in as a subroutine
  - Note we run H on a different input than the one that D is given
- Also, we didn’t say what D does if H goes into an infinite loop
  - It’s OK because H does not do that, by the assumption that H is a decider
Language accepted by $D$

(Repeat) $D$ behaves as follows:

1. $D$: input $x$
2. if $x$ is not of the form $<M>$ for some TM $M$, then $D$ rejects
3. simulate $H$ on input $<M, <M> >$
   - If $H$ accepts $<M, <M> >$, then $D$ rejects
   - If $H$ rejects $<M, <M> >$, then $D$ accepts

So $L(D)=\{ <M> | H$ rejects $<M, <M> > \}$

Now $H$ is a recognizer (even a decider) for $A_{TM}$, so if $H$ rejects $<M, <M> >$ then it means that the machine $M$ does not accept $<M>$.

So $L(D)=\{ <M> | <M> \notin L(M) \}$
Impossible machine

- So $L(D) = \{ <M> \mid <M> \notin L(M) \}$
- What if we give a copy of $D$’s own description $<D>$ to itself as input? As in Cantor’s theorem, we have trouble:
  - $<D> \in L(D) \Rightarrow <D> \notin L(D)$
  - $<D> \notin L(D) \Rightarrow <D> \in L(D)$
- So this $D$ can’t exist. But it was defined as a fairly straightforward wrapper around $H$: so $H$ must not exist either. That is, there is no decider for $A_{TM}$. QED
To summarize...

H accepts \(<M,w>\) exactly when M accepts w.

D rejects \(<M>\) exactly when M accepts \(<M>\).

D rejects \(<D>\) exactly when D accepts \(<D>\).

contradiction!
Diagonalization in this proof?

Mi is a TM.

Blank entry implies either loop or reject.

Now consider H, which is a decider.
Diagonalization in this proof? (cont.)

D computes the opposite of each diagonal entry because its behavior is opposite H’s behavior on input \(<M_i, <M_i>>\).

Cannot compute opposite of this entry itself!

Source: Sipser textbook
Current landscape

Each point is a language in this Venn diagram.

\[ A_{\text{TM}} \in \Sigma_1 - \Sigma_0 \]
Decidability versus recognizability

**Theorem 4.22** For every language $L$, $L \in \Sigma_0 \iff (L \in \Sigma_1 \text{ and } L^c \in \Sigma_1)$

*Recall that complement of a language is the language consisting of all strings that are not in that language.*

**Proof** The $\Rightarrow$ direction is easy, because $\Sigma_0 \subseteq \Sigma_1$ and $\Sigma_0$ is closed under complement.

For the $\Leftarrow$ direction, suppose that $L \in \Sigma_1$ and $L^c \in \Sigma_1$. Then there exist TMs so that $L(M_1) = L$ and $L(M_2) = L^c$. To show that $L \in \Sigma_0$, we need to produce a *decider* $M_3$ such that $L = L(M_3)$. 
Theorem 4.22 continued

L(M_1)=L, L(M_2)=L^c, and we want a *decider* M_3 such that L=L(M_3)

Strategy: given an input x, we know that either $x \in L$ or $x \in L^c$. So $M_3$ does this:

1. $M_3$: input x
2. set up tape #1 to simulate $M_1$ on input x
   and tape #2 to simulate $M_2$ on input x
3. compute one transition step of $M_1$ on tape 1
   and one transition step of $M_2$ on tape 2
   - if $M_1$ accepts, then $M_3$ accepts
   - if $M_2$ accepts, then $M_3$ rejects
   - else goto 3

This is like running both $M_1$ and $M_2$ *in parallel*. 
Theorem 4.22 conclusion

- For each string $x$, either $M_1$ accepts $x$ or $M_2$ accepts $x$, but never both
  - So the machine $M_3$ will always halt eventually in step 3
  - Therefore, $M_3$ is a decider

- $M_3$ accepts those strings in $L$ and rejects those strings in $L^c$
  - So $L(M_3) = L

QED
Getting a non-recognizable language from $A_{TM}$

- $L \in \Sigma_0 \iff (L \in \Sigma_1 \text{ and } L^c \in \Sigma_1)$
- $L \not\in \Sigma_0 \iff (L \not\in \Sigma_1 \text{ or } L^c \not\in \Sigma_1)$

Now since we know that $A_{TM} \not\in \Sigma_0$, and we know that $A_{TM} \in \Sigma_1$, it must be true that $A_{TM}^c \not\in \Sigma_1$.

- $A_{TM} = \{ <M,w> \mid M \text{ is a TM and } w \in L(M) \}$
- $A_{TM}^c = \{ x \mid x \text{ is not of the form } <M,w> \text{ or } (x = <M,w> \text{ and } w \not\in L(M)) \}$

If we narrow this down to strings of the form $<M,w>$, then the language is still unrecognizable:

- $NA_{TM} = \{ <M,w> \mid M \text{ is a TM and } w \not\in L(M) \}$
Unrecognizability

- $\text{NA}_{\text{TM}} = \{ <M, w> \mid M \text{ is a TM and } w \notin L(M) \}$

- What does it mean that $\text{NA}_{\text{TM}}$ is unrecognizable?
  - Every TM recognizes a language that’s different than $\text{NA}_{\text{TM}}$
  - Either it accepts strings that are not in $\text{NA}_{\text{TM}}$, or it fails to accept some strings that actually are in $\text{NA}_{\text{TM}}$

- Analogy to C programs:
  - Write a C program that takes another C program as input and prints out “loop” if the other C program goes into an infinite loop.
Each point is a language in this Venn diagram.

\[ \Delta_{\text{TM}} \in \Sigma_1 - \Sigma_0 \]

\[ \text{NA}_{\text{TM}} \in \text{ALL} - \Sigma_1 \]