Decidable and Undecidable Problems

Lecture 15

Andrew Black
Andrew Tolmach
Recall: Recognizable vs. Decidable

• A language L is **Turing recognizable** if some Turing machine recognizes it.
  ▶ Some strings not in L may cause the TM to loop
  ▶ Turing recognizable = *recursively enumerable* (RE)

• A language L is **Turing decidable** if some Turing machine decides it
  ▶ To decide is to return a definitive answer; the TM must halt on all inputs
  ▶ Turing decidable = decidable = *recursive*
Problems about Languages

• Consider some decision problems about languages, machines, and grammars:
  ▶ Ex.: Is there an algorithm that given any DFA $M$ and any string $w$, tells whether $M$ accepts $w$?
  ▶ Ex.: Is there an algorithm that given any two CFG’s $G_1$ and $G_2$ tells whether $L(G_1) = L(G_2)$?
  ▶ Ex. Is there an algorithm that given any TM $M$ tells whether $L(M) = \emptyset$?

• By Church-Turing thesis: “is there an algorithm?” = “is there a TM?”
Machine encodings

- We can encode machine or grammar descriptions (and inputs) as strings over a finite alphabet.

  - Example: Let’s encode the DFA $M = (Q,\Sigma,\delta,q_1,F)$ using the alphabet $\{0,1\}$
    - First, assign a unique integer $\geq 1$ to each $q \in Q$ and $x \in \Sigma$
    - Code each transition $\delta(q_i,x_j) = q_k$ as $0^i10^j10^k$
    - Code $F = \{q_p,...,q_r\}$ as $0^p1...10^r$
    - Code $M$ by concatenating codes for all transitions and $F$, separated by 11

  - We write $\langle M \rangle$ for the encoding of $M$ and $\langle M,w \rangle$ for the encoding of $M$ followed by input $w$
Problems on encodings

• We can specify problems as languages over the encoding strings.
  ▶ Ex.: \( A_{DFA} = \{ \langle M, w \rangle \mid M \text{ is a DFA that accepts } w \} \)
  ▶ Ex.: \( EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFG’s and } L(G) = L(H) \} \)
  ▶ Ex.: \( E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \} \)

• Now we can ask “is there a TM that decides this language?” (i.e., is there an algorithm that solves this problem?)
A decidable language

• To show that a language is decidable, we have to describe an algorithm that decides it
  ▶ We’ll allow informal descriptions as long as we are confident they can in principle be turned into TMs

• Consider $A_{\text{DFA}} = \{ \langle M, w \rangle \mid M \text{ is a DFA that accepts } w \}$

• Algorithm: Check that $M$ is a valid encoding; if not reject. Simulate behavior of $M$ on $w$. If $M$ halts in an accepting state, accept; if $M$ halts in a rejecting state, reject.
  ▶ We coded essentially this algorithm in DFA.c, although machine encoding was not read from input
Another decidable language

• Consider \( A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \} \)

• First attempt: build a TM that enumerates all possible derivations in \( G \). If it finds \( w \), it accepts. If it doesn’t find \( w \), it rejects.

• Problem: there may be an infinite number of derivations! So TM may never be able to reject.

• This TM **recognizes** \( A_{\text{CFG}} \), but doesn’t **decide** it.
Another try

• Consider $A_{ChCFG} = \{ \langle G, w \rangle \mid G$ is a CFG in Chomsky normal form that generates $w \}$

• We know that any derivation of $w$ in $G$ requires $2|w| - 1$ steps.

• So a TM that enumerates all derivations of this length can \textbf{decide} $A_{ChCFG}$.

• We also know an algorithm to convert an arbitrary CFG into CNF.

• Combining these two algorithms into a single TM gives a machine that \textbf{decides} $A_{CFG}$. 
Reduction

• We solved the decision problem for $A_{CFG}$ by algorithmically transforming the input into the form needed by another problem for which we could find a deciding TM.

• This strategy of reducing one problem $P$ to another (known) problem $Q$ is very common.
  ▶ If $P$ reduces to $Q$, and $Q$ is decidable, then $P$ is decidable.

• Must be certain that reduction process can be described by a TM!
Reductions (Hopcroft §9.3.1)

- Reductions must turn +ve instances of $P_1$ into +ve instances of $P_2$, -ve instances into -ve.
- It's common that only a small part of $P_2$ be the target of the reduction.
- Reduction is a TM that translates an instance of $P_1$ into an instance of $P_2$. 

\[ P_1 \quad \text{yes} \quad \text{no} \quad P_2 \]

\[ \text{yes} \quad \text{no} \]

Monday, 24 May 2010
The Value of Reductions

If there is a reduction from $P_1$ to $P_2$, then:

1. If $P_1$ is undecidable, so is $P_2$
2. If $P_1$ is non-RE, then so is $P_2$
The Value of Reductions

If there is a reduction from $P_1$ to $P_2$, then:

1. If $P_1$ is undecidable, so is $P_2$
2. If $P_1$ is non-RE, then so is $P_2$

Proof by contradiction:

Suppose that $P_2$ is decidable …

then we can use $P_2$ to decide $P_1$
The Value of Reductions

If there is a reduction from $P_1$ to $P_2$, then:

1. If $P_1$ is undecidable, so is $P_2$
2. If $P_1$ is non-RE, then so is $P_2$
The Value of Reductions

If there is a reduction from $P_1$ to $P_2$, then:

1. If $P_1$ is undecidable, so is $P_2$
2. If $P_1$ is non-RE, then so is $P_2$

Proof by contradiction:

Suppose that $P_2$ is recognizable …
then we can use $P_2$ to recognize $P_1$
Some other decidable problems

- $A_{NFA} = \{\langle M, w \rangle | M \text{ is an NFA that accepts } w \}$
  - By direct simulation, or by reduction to $A_{DFA}$.

- $A_{REX} = \{\langle R, w \rangle | R \text{ is a regular expression that generates } w \}$
  - By reduction to $A_{NFA}$.

- $E_{DFA} = \{\langle M \rangle | M \text{ is a DFA and } L(D) = \emptyset \}$
  - By inspecting the DFA’s transitions to see if there is any path to a final state.

- $EQ_{DFA} = \{\langle M_1, M_2 \rangle | M_1, M_2 \text{ are DFA’s and } L(M_1) = L(M_2) \}$
  - By reduction to $E_{DFA}$.

- $E_{CFG} = \{\langle G \rangle | G \text{ is a CFG and } L(G) = \emptyset \}$
  - By analysis of the CFG productions.
So far, we’ve fed descriptions of simple machines to TM’s. But nothing stops us from feeding descriptions of TM’s to TM’s!

In fact, this is really what we’ve been leading up to.

**A universal** TM U behaves as follows:

- U checks input has form \(\langle M, w \rangle\) where M is an (encoded) TM and w is a string
- U simulates behavior of M on input w.
- If M ever enters an accept state, U accepts
- If M ever rejects, U rejects
Role of Universal TM

• U models a (real-world) stored program computer.
  ▶ Capable of doing many different tasks, depending on program you feed it

• Existence of U shows that the language \( A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} \) is Turing-recognizable

• But it doesn’t show that \( A_{TM} \) is Turing-decidable
  ▶ If M runs forever on some w, U does too (rather than rejecting)
A\textsuperscript{TM} is undecidable

• Proof is by contradiction.

• Suppose A\textsuperscript{TM} is decidable. Then some TM H decides it.
  
  ‣ That is, for any TM M and input w, if we run H on \langle M, w \rangle then H accepts if M accepts w and rejects if M does not accept w.

• Now use H to build a machine D, which
  
  ‣ when started on input \langle M \rangle, runs H on \langle M, \langle M \rangle \rangle

  ‣ does the opposite of H: if H rejects, D accepts and if H accepts, D rejects.
H cannot exist

• We have

\[ D(\langle M \rangle) = \begin{cases} 
accept & \text{if } M \text{ does not accept } \langle M \rangle \\
reject & \text{if } M \text{ accepts } \langle M \rangle. 
\end{cases} \]

• But now if we run D with its own description as input, we get

\[ D(\langle D \rangle) = \begin{cases} 
accept & \text{if } D \text{ does not accept } \langle D \rangle \\
reject & \text{if } D \text{ accepts } \langle D \rangle. 
\end{cases} \]

• This is paradoxical! So D cannot exist. Therefore H cannot exist either. So A\text{TM} is not decidable.
An unrecognizable language

• A language $L$ is decidable $\iff$ both $L$ and $\overline{L}$

  are Turing-recognizable.

  ▶ Proof: $\Rightarrow$ is obvious. For $\Leftarrow$, we have TM’s $M_1$ and $M_2$ that recognize $L$, $\overline{L}$ respectively. Use them to build a TM $M$ that runs $M_1$ and $M_2$ in parallel until one of them accepts (which must happen). If $M_1$ accepts $M$ accepts too; if $M_2$ accepts, $M$ rejects.

• $A_{TM}$ is not Turing-recognizable.

  ▶ Proof by contradiction. Suppose it is. Then, since $A_{TM}$ is recognizable, $A_{TM}$ is decidable. But it isn’t!
HALT_{TM} is undecidable

• HALT_{TM} = \{⟨M,w⟩ | M is a TM and M halts on input w\}

• Proof is by reduction from A_{TM}.

• If problem P reduces to problem Q, and P is undecidable, then Q is undecidable!
  ▷ Otherwise, we could use Q to decide P.

• So must show how a TM that decides HALT_{TM} can be used to decide A_{TM}. 
Acceptance reduces to Halting

• Assume TM R decides HALT\textsuperscript{TM}.

• Then the following TM S decides A\textsuperscript{TM}:
  ▶ First, S runs R on \langle M,w \rangle.
  ▶ If R rejects, we know that M does not halt on w. So M certainly does not accept w. So S rejects.
  ▶ If R accepts, S simulates M on w until it halts (which it will!)
    ◦ If M is in an accept state, S accepts; if M is in a reject state, S rejects.

• Since S cannot exist, neither can R.
Another undecidable problem

- \( E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \} \) is undecidable.

- Proof is again by reduction from \( A_{TM} \): we suppose TM \( R \) decides \( E_{TM} \) and use it to define a TM that decides \( A_{TM} \) as follows:
  - Check that input has form \( \langle M, w \rangle \); if not, reject.
  - Construct a machine description \( \langle M_1 \rangle \) such that \( L(M_1) = L(M) \cap \{w\} \). (How?)
  - Run \( R \) on \( \langle M_1 \rangle \). If it accepts, \( L(M) \cap \{w\} = \emptyset \), so \( w \notin L(M) \), so reject. If it rejects, \( L(M) \cap \{w\} \neq \emptyset \), so \( w \in L(M) \), so accept.
Rice’s Theorem

• In fact, the approach of this last result can be generalized to prove Rice’s Theorem:

• Let $P$ be any **non-trivial** property of Turing-recognizable languages

  ▶ Non-trivial means $P$ is true of some but not all

• Then $\{\langle M \rangle | P \text{ is true of } L(M) \}$ is undecidable

• Examples of undecidable properties of $L(M)$:

  ▶ $L(M)$ is empty, non-empty, finite, regular, CF, ...
Other Undecidable Problems

• Problems about CFGs $G, G_1, G_2$
  ▶ Is $G$ ambiguous? Is $L(G_1) \subseteq L(G_2)$? Is $L(G)$ context-free?

• Post’s Correspondence Problem

• Hilbert’s 10th Problem
  ▶ Does a polynomial equation $p(x_1, x_2, \ldots, x_n) = 0$ with integer coefficients have a solution consisting of integers?

• Equivalence Problem
  ▶ Do two arbitrary Turing-computable functions have the same output on all arguments?
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1} s_{i_2} \ldots s_{i_k} = t_{i_1} t_{i_2} \ldots t_{i_k}\) ?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - abbabaab
Post's Correspondence Problem

Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1} s_{i_2} \ldots s_{i_k} = t_{i_1} t_{i_2} \ldots t_{i_k}\) ?

Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)

- The sequence 1, 2, 1, 3, 4 gives us
  - abbabaab
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1,t_1), (s_2,t_2),..., (s_n,t_n)\), is there a sequence of indices \(i_1,i_2,...,i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}...s_{i_k} = t_{i_1}t_{i_2}...t_{i_k}\)?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence \(1, 2, 1, 3, 4\) gives us
    - \text{abbabaab}
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1,t_1), (s_2,t_2), \ldots, (s_n,t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\)?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - \text{abbabaab}
Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), ..., (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, ..., i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}...s_{i_k} = t_{i_1}t_{i_2}...t_{i_k}\)?

Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
- The sequence 1, 2, 1, 3, 4 gives us - abbabaab
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1} s_{i_2} \ldots s_{i_k} = t_{i_1} t_{i_2} \ldots t_{i_k}\) ?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - abbabaab
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1,t_1), (s_2,t_2), \ldots, (s_n,t_n)\), is there a sequence of indices \(i_1,i_2,\ldots,i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\)?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence \(1, 2, 1, 3, 4\) gives us
    - \(abbabaab\)
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\) ?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - abbabaab
Post's Correspondence Problem

Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), ..., (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, ..., i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}...s_{i_k} = t_{i_1}t_{i_2}...t_{i_k}\) ?

Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
- The sequence \(1, 2, 1, 3, 4\) gives us \(abbabaab\)
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1,t_1), (s_2,t_2), \ldots, (s_n,t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\)?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - \(abbabaab\)
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1,t_1), (s_2,t_2), \ldots, (s_n,t_n)\), is there a sequence of indices \(i_1,i_2,\ldots,i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\) ?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
    - abbabaab
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1} s_{i_2} \ldots s_{i_k} = t_{i_1} t_{i_2} \ldots t_{i_k}\)?

- Example: \((ab, a), (b, bb), (aa, b), (b, aab)\)
  - The sequence 1, 2, 1, 3, 4 gives us
  - abbabaab
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, \ldots, i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}\ldots s_{i_k} = t_{i_1}t_{i_2}\ldots t_{i_k}\)?

- Example: \((ab, a), (b, ab)\)
  - has no solution
  - Why?
Post's Correspondence Problem

- Given a finite sequence of pairs of strings \((s_1, t_1), (s_2, t_2),..., (s_n, t_n)\), is there a sequence of indices \(i_1, i_2, ..., i_k\) (duplications allowed) such that \(s_{i_1}s_{i_2}...s_{i_k} = t_{i_1}t_{i_2}...t_{i_k}\) ?

- There is no algorithm that can decide, for an arbitrary instance of Post's Correspondence problem, whether there is a solution.
The Halting Problem, and other things uncomputable: An approach by counting
Computability

• Anything computable can be computed by a Turing machine …
  ▶ or one of the equivalent models, such as a partial recursive function or a $\lambda$-calculus expression

• But: not everything is computable

• Basic argument:
  ▶ There are a countably-infinite number of Turing machines (partial recursive function, $\lambda$-calculus expressions…)
  ▶ There are an uncountable number of functions $\mathbb{N} \rightarrow \mathbb{N}$
Countability of Turing Machines

• To prove that a set is countably infinite, we need only exhibit a bijection between its elements and $\mathbb{N}$
  ▶ an injection suffices to show that it is countable

• That’s called an “Effective Enumeration”
  ▶ you have a way of “counting off” the Turing Machines

• Basic idea: you can encode anything (e.g., a description of a Turing Machine) in binary
  ▶ but any string of binary digits can be interpreted as a (large) integer
Hein’s enumeration

• Take a (large) integer $n$
  ▶ Write it in base-128 notation
  ▶ regard each base-128 digit as an ASCII character
  ▶ ask: is the resulting ascii string a description of a Turing machine?

• If so, that’s the $n^{th}$ Turing machine

• If not, arbitrarily say that the $n^{th}$ Turing machine is “(0, a, a, S, Halt)”

• If we do this for all $n \in \mathbb{N}$, we will eventually get all the TMs
And for $\lambda$-calculus?

- All the expressions can also be effectively enumerated…
  - and also the primitive recursive functions,
  - and the Markov algorithms…

- The details are unimportant, so long as you agree that it makes sense to talk about the Turing machine (or $\lambda$-expression …) corresponding to a certain number.
Functions over the Natural Numbers

• There are an uncountable number of functions in $\mathbb{N} \to \mathbb{N}$

• We prove this by a diagonalization argument
  ▶ the same kind of argument that you used to prove that there were more real numbers than integers.

• Assume that there are a countable number of functions

• establish a contradiction
  ▶ This is in chapter 2.4 of Hein (p.121) if you need to refresh your memory!
• Assume that $\mathbb{N} \to \mathbb{N}$ is countably infinite.

• Then there is a enumeration $f_0, f_1, f_2, f_3, \ldots$ of all of the functions in $\mathbb{N} \to \mathbb{N}$

• Now consider the function $g: \mathbb{N} \to \mathbb{N}$ defined as follows:

$$g(n) = \text{if } f_n(n) = 1 \text{ then } 2 \text{ else } 1$$

• Then $g$ is not one of the $f_i$

  • it differs from $f_0$ at 0, from $f_1$ at 1, …

• This contradicts the assumption that $\mathbb{N} \to \mathbb{N}$ is countably infinite.
• There are *lots* of uncomputable functions
  ▶ in fact: an uncountable number of them!
One Uncomputable Function

• Assume that the following function $H(x)$ is computable
  ▷ $H(x) = \text{if } f_x \text{ halts on input } X \text{ then loop forever else 0}$

• Then $H$ must be in our enumeration of computable functions, say $H = f_k$
  ▷ So: $f_k(x) = \text{if } f_x \text{ halts on input } x \text{ then loop forever else 0}$

• Now apply $f_k$ to its own index:
  ▷ $f_k(k) = \text{if } f_k \text{ halts on input } k \text{ then loop forever else 0}$
  ▷ Thus: if $f_k(k)$ halts, then $f_k(k)$ loops forever, but if $f_k(k)$ loops forever, then $f_k(k) = 0$

• We have a contradiction
The Halting Problem

• Is there a Turing Machine that can decide whether the execution of an arbitrary TM halts on an arbitrary input?

• Is there a λ-calculus expression that can decide whether the application of an arbitrary λ-term to a second λ-term will reach a normal form?

• Is there a simple program that can decide whether an arbitrary simple program will halt when given arbitrary initial values for its variables?
The Halting Problem

• Is there a Turing Machine that can decide whether the execution of an arbitrary TM halts on an arbitrary input? **No**

• Is there a $\lambda$-calculus expression that can decide whether the application of an arbitrary $\lambda$-term to a second $\lambda$-term will reach a normal form?

• Is there a simple program that can decide whether an arbitrary simple program will halt when given arbitrary initial values for its variables?
The Halting Problem

- Is there a Turing Machine that can decide whether the execution of an arbitrary TM halts on an arbitrary input? **No**

- Is there a $\lambda$-calculus expression that can decide whether the application of an arbitrary $\lambda$-term to a second $\lambda$-term will reach a normal form? **No**

- Is there a simple program that can decide whether an arbitrary simple program will halt when given arbitrary initial values for its variables?
The Halting Problem

• Is there a Turing Machine that can decide whether the execution of an arbitrary TM halts on an arbitrary input?  No

• Is there a λ-calculus expression that can decide whether the application of an arbitrary λ-term to a second λ-term will reach a normal form?  No

• Is there a simple program that can decide whether an arbitrary simple program will halt when given arbitrary initial values for its variables?  No
What this *doesn’t* mean

- Nothing about these results says that for *some* TM, or *some* simple program, or for *some* λ-expression, applied to *some* input, we can’t decide whether it will halt.

- The unsolvability of the Halting problem just says that we can’t *always* do it
Decidability

- A decision problem is a question with a yes or no answer.
- The problem is decidable if there is an algorithm/function/TM that can input the problem and always halt with the correct answer.
- The problem is semi-decidable (aka partially decidable, aka partially solvable) if there is an algorithm that halts and answers yes when the correct answer is yes, but may run forever if the answer is no.
Examples

• Is there an algorithm to decide if the following *simple* programs halt on arbitrary initial state:
Examples

• Is there an algorithm to decide if the following \textit{simple} programs halt on arbitrary initial state:

\[ X := 0 \]
Examples

• Is there an algorithm to decide if the following *simple* programs halt on arbitrary initial state:

```
X := 0
while X ≠ 0
do Y := succ(X) od
```
Examples

• Is there an algorithm to decide if the following *simple* programs halt on arbitrary initial state:

\[
\begin{align*}
X &:= 0 \\
\text{while } X \neq 0 \\
& \quad \text{do } Y := \text{succ}(X) \od 
\end{align*}
\]

• Is there an algorithm to decide if an arbitrary *simple* program halts on arbitrary initial state?
Examples

• Is there an algorithm to decide if the following simple programs halt on arbitrary initial state:

\[
\begin{align*}
X &:= 0 \\
\text{while } X \neq 0 \\
&\, \text{do } Y := \text{succ}(X) \text{ od}
\end{align*}
\]

• Is there an algorithm to decide if an arbitrary simple program halts on arbitrary initial state?

• What about Java programs? ML programs?
More Undecidable Problems

- Is there a Turing Machine that can recognize any Regular Language?
- Is there a Turing Machine that can recognize any Context-free language?
- Are all languages Turing-Recognizable?
What’s a “Language”? 

- A language over an alphabet A is a set of strings from A*
  - In other words: each subset of A* is a language
    - A language is a member of \( P(A^*) \)
  - A* is countably infinite (for any finite A)
  - So \( P(A^*) \) is uncountable

- There are an uncountable number of languages
Why is $\mathcal{P}(A^*)$ Uncountable?

- The set $\mathcal{B}$ of infinite binary sequences is uncountable
- $A^*$ can be enumerated, say, in lexicographic order
- Any particular language, $L$, over $A$ can be represented as a bit-mask, that is, as an element of $\mathcal{B}$
Proof
Proof

A = \{a, b\}

L_1 = \{ w \in A^* \mid w \text{ starts with } a \}
Proof

\[ A = \{a, b\} \]
\[ L_1 = \{ w \in A^* \mid w \text{ starts with } a \} \]
\[ A^* = \{ \Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots \} \]
\[ L_1 = \{ a, aa, ab, aaa, aab, \ldots \} \]
\[ \chi(L_1) = \{ 0, 1, 0, 1, 1, 0, 0, 1, 1, \ldots \} \]
Proof

\[ A = \{a, b\} \]
\[ L_1 = \{ w \in A^* \mid w \text{ starts with } a \} \]

\[ A^* = \{ \Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots \} \]
\[ L_1 = \{ a, aa, ab, aaa, aab, \ldots \} \]

\[ \chi(L_1) = \{ 0, 1, 0, 1, 1, 0, 0, 1, 1, \ldots \} \]

✓ \( \chi(L_1) \) is called the characteristic sequence of \( L_1 \)
Proof

\[ A = \{a, b\} \]
\[ L_1 = \{ w \in A^* \mid w \text{ starts with } a \} \]
\[ A^* = \{ \Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots \} \]
\[ L_1 = \{ a, aa, ab, aab, \ldots \} \]
\[ \chi(L_1) = \{ 0, 1, 0, 1, 1, 0, 0, 1, 1, \ldots \} \]

✓ \( \chi(L_1) \) is called the characteristic sequence of \( L_1 \)

✓ \( \chi(L_1) \) is a member of \( B \)
Proof

\[ A = \{ a, b \} \]
\[ L_1 = \{ w \in A^* \mid w \text{ starts with } a \} \]
\[ A^* = \{ \Lambda, a, b, aa, ab, ba, bb, aaa, aab, \ldots \} \]
\[ L_1 = \{ a, aa, ab, \ldots \} \]
\[ \chi(L_1) = \{ 0, 1, 0, 1, 1, 0, 0, 1, 1, \ldots \} \]

✓ \( \chi(L_1) \) is called the characteristic sequence of \( L_1 \)

✓ \( \chi(L_1) \) is a member of \( \mathcal{B} \)

✓ We have just displayed a bijection between \( \mathcal{B} \) and the languages over \( A \)
Some languages are not recognizable

- There are an uncountable number of languages
- There are a countable number of Turing machines
- Each Turing machine recognizes exactly one language
A Little History

• At the start of the 20th Century, it was thought that all mathematical problems were decidable
  • if you could formulate the problem precisely, and if you were smart enough, you could always come up with an algorithm to solve it.
  • 1931: Kurt Gödel showed that this was impossible
Gödel’s Incompleteness Theorems

1. There are first-order statements about the natural numbers that can neither be proved nor disproved from Peano’s axioms

2. It’s impossible to prove from Peano’s axioms that Peano’s axioms are consistent.
• Two key ideas behind Gödel’s proof

1. Gödel Numbering

Each formula (or sequence of formulae) can be encoded as an integer; each integer represents a formula or a sequence of formulae.

So: \( \omega(x, y) \) asserts that \( y \) is (the Gödel number of) a proof of \( x \).

\[ \forall y . \neg \omega(x, y) \]

asserts that \( x \) is unprovable.

2. Self reference (diagonalization)

If \( p \) is the Gödel number of \( \forall y . \neg \omega(x, y) \), then

\( \zeta = \forall y . \neg \omega(p, y) \)

asserts that \( \zeta \) is unprovable.
Turing: applied Gödel to Computability

• The same two ideas:
  • Encoding: any “computing machine”, or program, can be represented as data (which the machine can take as input).
  • Self-reference: a machine (or program) operating on a description of itself as input
Halting Problem for Programmers

• Student claims that they have a program
  
  ```
  fun halts(program, input): boolean = …
  ```

• Note that `halts` takes an encoding of a program as its first argument.

• But look:

  ```
  paradox(program) = if halts(program, program)
  then loopForever
  else true
  ```

  `paradox(paradox)` answers *what*?
paradox(paradox) = if halts(paradox, paradox) then loopForever else true

- If paradox halts when run on itself as input, then …
- If paradox does not halt when run on itself as input …
- Either way, we have a contradiction
- Therefore, you can’t write the program halts