The Limits of Algorithmic Computation

5DV037 — Fundamentals of Computer Science
Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Background: das Entscheidungsproblem

- In 1928, the eminent German mathematician David Hilbert (with Wilhelm Ackermann) posed *das Entscheidungsproblem* (the decision problem).
- The goal was to have an algorithm which would solve all mathematical problems. (A universal theorem prover.)
- In 1931, the Austrian mathematician Kurt Gödel showed that this is impossible via the *incompleteness theorem for arithmetic of the natural numbers*.
- In 1936, the British mathematician Alan Turing used a simple computer model to show that there are well-defined language problems which cannot be solved by computer.
- In 1937, the US mathematician Alonzo Church independently showed similar result for first-order logic.

Why Should You Care?

- Real systems in AI use *theorem provers* to make decisions on what to do.
- The result shows that theorem provers for first-order predicate logic (a very common modelling tool) cannot always decide on the truth value of an assertion.
  - It might run forever (but it is not possible to tell whether it will.)
- Proving that a program is "correct" (that it satisfies certain conditions) is also very important in software engineering of critical systems.
- The result shows that this is not possible in the general case.

Why Should You Care? — 2

Example: Here is a practical example.

- Suppose that you are an assistant in an introductory programming course.
- You must grade 300 programs which are supposed to sort a list of numbers.
- You decide instead that you will write a program which will take as input the program of each student and decide whether or not it is correct.
- Unfortunately, the theory shows that this is not possible.
- The problem is that your program might run forever, but you cannot tell that it will.
- You could still write a program which would work in certain cases (*e.g.*, the student program must sort a list of 1000 element in less than a second), but the theory shows that you cannot write a general solution.

The Number of Strings over Σ

Context: A finite nonempty alphabet Σ.

- The number of strings in Σ^* is *countable*.
- This means that they may be put into *bijective (one-to-one) correspondence* with the natural numbers $\mathbb{N}$.
- List the strings in order of increasing length.

Example: $\Sigma = \{a, b\}$

  - All strings of length 0: $\{\lambda\}$.
  - All strings of length 1: $\{a, b\}$.
  - All strings of length 2: $\{aa, ab, ba, bb\}$.
  - All strings of length 3: $\{aaa, aab, aba, abb, baa, bab, bba, bbb\}$.
  - $\vdots$

- Just list the shorter strings before the longer ones in lexicographic order.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	$\cdots$
α	λ	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb	$\cdots$

- Such a list is called an *enumeration* of the strings, and the set is called *enumerable*.

Enumeration Procedures

Context: A finite nonempty alphabet Σ and a language $L \subseteq \Sigma^*$.

- A DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ is an *enumerator* for L if there is a distinguished state $q_s \in Q$ with the property that if the machine is started in configuration $\mathcal{I}\langle M, \lambda\rangle = \langle q_0, \lambda, \sqcup, \lambda\rangle$ then it will execute a computation sequence

$$\mathcal{I}\langle M, \lambda\rangle \ \vdash_M^* \ D_1 \ \vdash_M^* \ D_2 \ \vdash_M^* \ \ldots \ \vdash_M^* \ D_i \ldots \ \vdash_M^* \ \ldots$$

  in which:
  - Each D_i is an output configuration in state q_s for some $\alpha_i \in L$;
  - Every string in L is one of the α_i's.
  - Every configuration of the computation which is in state q_S is one of the D_i's;
- If L is infinite, this computation must also be infinite.
- The computation is called a *(recursive) enumeration* of L,
- and L is said to be *(recursively) enumerable*.

Selection from an Enumeration

- Given an enumerator $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ for a language L, it is easy to build a machine $M' = (Q', \Sigma, \Gamma', \delta', q_0', \sqcup, F')$ which takes as input $i \in \mathbb{N}$ and computes the i^{th} element in the enumeration.
- Just run the enumerator, and keep a counter of how many strings have been found.
- It is also possible to eliminate duplicates, by keeping a list of those strings which have already been found (on a second tape or some other region of a single tape).
- Invoke the Church-Turing thesis!

The Number of Languages over Σ

Context: A finite nonempty alphabet Σ.

Fact: The number of languages over Σ is not countable.

Proof outline: Suppose, to the contrary, that $L_0, L_1, L_2, \ldots L_i, \ldots, \ldots$ is such an enumeration.

- Let $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_i \ldots$ be an enumeration of Σ^*.
- Define L to be the language which includes α_i iff $\alpha_i \notin L_i$ (for each $i \in \mathbb{N}$).
- Then L cannot be L_i for any $i \in \mathbb{N}$, because $\alpha_i \in L$ iff $\alpha_i \notin L_i$. $\square$
- This is a *diagonalization argument*.

Encoding the DTMs over Σ as Strings in $\{0,1\}^*$

Context: A DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, ❑, F)$ over Σ with $\{0,1\} \subseteq \Sigma$.

- Without loss of generality, assume that M has exactly one final state.
- For n states, represent them $1, \ldots, n$, with 1 the start state and n the final state.
- Represent Γ as $\{1, 2, \ldots, m\}$, with 1 representing ❑.
- Encode states and tape symbols in *unary* using this convention.
  - $1 \rightsquigarrow 1$, $2 \rightsquigarrow 11$, $3 \rightsquigarrow 111$, *etc.*.
- Represent L, R, and S as 1, 11, and 111, respectively.
- Represent the transition $\delta(q, a) = (q', a', d)$ as

  $$\mathrm{Code}(q)0\mathrm{Code}(a)0\mathrm{Code}(q')0\mathrm{Code}(a')0\mathrm{Code}(d)$$

  in which $\mathrm{Code}(x)$ is the code of x in unary, as described above.
- Represent the DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, ❑, F)$ as a string $\langle T_1, T_2, \ldots, T_k \rangle$ in which
  - each T_i describes one entry of δ as indicated above;
  - each comma is represented by a 0;
  - n and m may be recovered from the T_i's.

The Number of DTMs over Σ

Context: A finite nonempty alphabet Σ containing $\{0, 1\}$.

A useful naming convention: Let DTM_Σ denote the set of all DTMs over Σ, using the encoding described on the previous slide.

- Thus, DTM_Σ is a language over Σ.
- It encodes canonical representations of DTMs, up to a renaming of states.

Observation: DTM_Σ is a recursively enumerable language.

Proof: Use the representation given on the previous slide as the basis for an enumeration.

- Generate machines with smaller n and m before machines with larger values. $\square$

- There are (many) more languages than there are DTMs.
- Most languages are not Turing acceptable. $\square$

The Universal DTM for Σ

Context: Fix:

- A finite alphabet Σ with $\{0, 1\} \subseteq \Sigma$ (rename if necessary).
- A (recursive) enumeration $M_0, M_1, \ldots, M_i, \ldots$ of the DTMs with input alphabet Σ (as just described).
- A (recursive) enumeration $\alpha_0, \alpha_1, \ldots, \alpha_i, \ldots,$ of the strings in Σ^*.

- A *universal DTM* (or *universal Turing machine*) for alphabet Σ takes two arguments:
  - An index i identifying the DTM M_i; and
  - An index j identifying the string α_j;
- and:
  - halts in an accepting state if $\alpha_j \in \mathcal{L}(M_i)$;
  - halts in a rejecting state if $\alpha_j \notin \mathcal{L}(M_i)$ and M_i halts on input α_i;
  - does not halt if M_i does not halt on input α_j.
- Thus, a universal Turing machine is essentially an interpreter for DTMs.
- But, for simplicity, the definition deals with acceptance only, and not with computation of functions.

Building a Universal DTM for Σ

- It is straightforward to build such a machine.
- It has three main steps to compute $M_i(\alpha_j)$, defined by subroutines:
    - Compute the representation of M_i (by running an enumerator).
    - Compute the representation of α_j (by running an enumerator).
    - Run a "DTM interpreter" on (M_i, α_j).
- The easiest way to argue that this can be done is to appeal to the Church-Turing thesis.
    - You can write a program in C to do this, can't you?
    - Tedious, but certainly possible.
- Build the machine explicitly as a three-tape DTM, and then appeal to the equivalence to a one-tape machine.

Notation: Let UDTM_Σ denote a universal DTM over Σ.

- Write $\text{UDTM}_\Sigma\langle i, j\rangle$ for the result of running UDTM_Σ on input $\langle i, j\rangle$.
    - *i.e.*, simulate M_i on input α_j.

The Halting Problem

Definition: The *halting problem* for DTMs over Σ is, given arbitrary $i, j \in \mathbb{N}$, determine whether UDTM_Σ halts on input $\langle i, j \rangle$.

- In other words, determine whether M_i halts when run from initial configuration $\mathcal{I}\langle M_i, \alpha_j \rangle$.
- The goal is to show that there is no DTM which can compute the answer to this question.
- To show this, begin by defining a modified universal DTM which only cares about halting.

Notation: Let HUDTM_Σ denote the DTM which takes two inputs and computes

$$\mathsf{HUDTM}_\Sigma\langle i, j \rangle = \begin{cases} 1 & \text{if } \mathsf{UDTM}_\Sigma \text{ halts on input } \langle i, j \rangle \\ \text{undefined} & \text{if } \mathsf{UDTM}_\Sigma \text{ does not halt on input } \langle i, j \rangle \end{cases}$$

- It is trivial to build HUDTM_Σ from UDTM_Σ.

The Halting Problem — 2

Notation: Let HUDTM_Σ denote the DTM which takes two inputs and computes

$$\mathrm{HUDTM}_\Sigma\langle i,j\rangle = \begin{cases} 1 & \text{if } \mathrm{UDTM}_\Sigma \text{ halts on input } \langle i,j\rangle \\ \text{undefined} & \text{if } \mathrm{UDTM}_\Sigma \text{ does not halt on input } \langle i,j\rangle \end{cases}$$

- Now, conjecture that a machine which computes the function obtained by replacing "undefined" by 0 in the definition of HUDTM_Σ could be built:

$$\mathrm{Halt}_\Sigma\langle i,j\rangle = \begin{cases} 1 & \text{if } \mathrm{UDTM}_\Sigma \text{ halts on input } \langle i,j\rangle \\ 0 & \text{if } \mathrm{UDTM}_\Sigma \text{ does not halt on input } \langle i,j\rangle \end{cases}$$

- Such a machine would solve the halting problem.

Diagonalization and the Halting Problem

$$\mathrm{Halt}_{\Sigma}\langle i, j\rangle = \begin{cases} 1 & \text{if } \mathrm{UDTM}_{\Sigma} \text{ halts on input } \langle i, j\rangle \\ 0 & \text{if } \mathrm{UDTM}_{\Sigma} \text{ does not halt on input } \langle i, j\rangle \end{cases}$$

- The values computed by Halt_{Σ} may be viewed as entries in a matrix.
- Row i describes the *halting pattern* of M_i.
- Of special interest is the diagonal.
- Call the function so defined $\Delta\text{-}\mathrm{Halt}_{\Sigma}$: $\quad \Delta\text{-}\mathrm{Halt}_{\Sigma}\langle i\rangle = \mathrm{Halt}_{\Sigma}\langle i, i\rangle$.

$$\begin{array}{c|cccccccc}
 & \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_i & \cdots & \alpha_j & \cdots \\
\hline
M_0 & & & & & & & & \\
M_1 & & & & & & & & \\
M_2 & & & & & & & & \\
\vdots & & & & & & & & \\
M_i & & & & & & & \mathrm{Halt}_{\Sigma}\langle i, j\rangle & \cdots \\
\vdots & & & & & & \ddots & \vdots &
\end{array}$$

Diagonalization and the Halting Problem

$$\mathrm{Halt}_\Sigma\langle i, j\rangle = \begin{cases} 1 & \text{if } \mathrm{UDTM}_\Sigma \text{ halts on input } \langle i, j\rangle \\ 0 & \text{if } \mathrm{UDTM}_\Sigma \text{ does not halt on input } \langle i, j\rangle \end{cases}$$

- The values computed by Halt_Σ may be viewed as entries in a matrix.
- Row i describes the *halting pattern* of M_i.
- Of special interest is the diagonal.
- Call the function so defined $\Delta\text{-}\mathrm{Halt}_\Sigma$: $\quad \Delta\text{-}\mathrm{Halt}_\Sigma\langle i\rangle = \mathrm{Halt}_\Sigma\langle i, i\rangle$.

$$\begin{array}{c|cccccccc}
 & \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_i & \cdots & \alpha_j & \cdots \\
\hline
M_0 & \mathrm{Halt}_\Sigma\langle 0, 0\rangle & & & & & & & \\
M_1 & & \mathrm{Halt}_\Sigma\langle 1, 1\rangle & & & & & & \\
M_2 & & & \mathrm{Halt}_\Sigma\langle 2, 2\rangle & & & & & \\
\vdots & & & & \ddots & & & & \\
M_i & & & & & \mathrm{Halt}_\Sigma\langle i, i\rangle & & \mathrm{Halt}_\Sigma\langle i, j\rangle & \cdots \\
\vdots & & & & & & \ddots & \vdots &
\end{array}$$

Diagonalization and the Halting Problem

$$\mathrm{Halt}_\Sigma\langle i,j\rangle = \begin{cases} 1 & \text{if UDTM}_\Sigma \text{ halts on input } \langle i,j\rangle \\ 0 & \text{if UDTM}_\Sigma \text{ does not halt on input } \langle i,j\rangle \end{cases}$$

- The values computed by Halt_Σ may be viewed as entries in a matrix.
- Row i describes the *halting pattern* of M_i.
- Of special interest is the diagonal.
- Call the function so defined Δ-Halt_Σ: $\Delta\text{-}\mathrm{Halt}_\Sigma\langle i\rangle = \mathrm{Halt}_\Sigma\langle i,i\rangle$.

$$\begin{array}{c|cccccccc}
 & \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_i & \cdots & \alpha_j & \cdots \\
\hline
M_0 & \Delta\text{-}\mathrm{Halt}_\Sigma\langle 0\rangle & & & & & & & \\
M_1 & & \Delta\text{-}\mathrm{Halt}_\Sigma\langle 1\rangle & & & & & & \\
M_2 & & & \Delta\text{-}\mathrm{Halt}_\Sigma\langle 2\rangle & & & & & \\
\vdots & & & & \ddots & & & & \\
M_i & & & & & \Delta\text{-}\mathrm{Halt}_\Sigma\langle i\rangle & & \mathrm{Halt}_\Sigma\langle i,j\rangle & \cdots \\
\vdots & & & & & & \ddots & \vdots &
\end{array}$$

Diagonalization and the Halting Problem — 2

- Now consider the function $\overline{\Delta}\text{-Halt}_\Sigma$: $\overline{\Delta}\text{-Halt}_\Sigma\langle i\rangle = 1 - \Delta\text{-Halt}_\Sigma\langle i\rangle$.
- This function cannot describe the halting pattern of any of the M_i.
- $\overline{\Delta}\text{-Halt}_\Sigma\langle\alpha_i\rangle \neq \text{Halt}_\Sigma\langle M_i, \alpha_i\rangle$.
- But it describes the halting pattern of $\Delta'\text{-Halt}_\Sigma$:

$$\Delta'\text{-Halt}_\Sigma\langle i\rangle = \begin{cases} \text{undefined} & \text{if } \text{UDTM}_\Sigma \text{ halts on input } \langle i, i\rangle \\ 0 & \text{if } \text{UDTM}_\Sigma \text{ does not halt on input } \langle i, i\rangle \end{cases}$$

- Hence $\Delta'\text{-Halt}_\Sigma$ cannot be computed by any DTM.

	α_0	α_1	α_2	$\cdots$	α_i	$\cdots$	α_j	$\cdots$
M_0	$\Delta\text{-Halt}_\Sigma\langle 0\rangle$							
M_1		$\Delta\text{-Halt}_\Sigma\langle 1\rangle$						
M_2			$\Delta\text{-Halt}_\Sigma\langle 2\rangle$					
$\vdots$				$\ddots$				
M_i					$\Delta\text{-Halt}_\Sigma\langle i\rangle$		$\text{Halt}_\Sigma\langle i, j\rangle$	$\cdots$
$\vdots$						$\ddots$	$\vdots$	

Diagonalization and the Halting Problem — 2

- Now consider the function $\overline{\Delta}\text{-Halt}_\Sigma$: $\overline{\Delta}\text{-Halt}_\Sigma\langle i\rangle = 1 - \Delta\text{-Halt}_\Sigma\langle i\rangle$.
- This function cannot describe the halting pattern of any of the M_i.
- $\overline{\Delta}\text{-Halt}_\Sigma\langle \alpha_i\rangle \neq \text{Halt}_\Sigma\langle M_i, \alpha_i\rangle$.
- But it describes the halting pattern of $\Delta'\text{-Halt}_\Sigma$:

$$\Delta'\text{-Halt}_\Sigma\langle i\rangle = \begin{cases} \text{undefined} & \text{if UDTM}_\Sigma \text{ halts on input } \langle i, i\rangle \\ 0 & \text{if UDTM}_\Sigma \text{ does not halt on input } \langle i, i\rangle \end{cases}$$

- Hence $\Delta'\text{-Halt}_\Sigma$ cannot be computed by any DTM.

	α_0	α_1	α_2	$\cdots$	α_i	$\cdots$	α_j	$\cdots$
M_0	$1 - \Delta\text{-Halt}_\Sigma\langle 0\rangle$							
M_1		$1 - \Delta\text{-Halt}_\Sigma\langle 1\rangle$						
M_2			$1 - \Delta\text{-Halt}_\Sigma\langle 2\rangle$					
$\vdots$				$\ddots$				
M_i					$1 - \Delta\text{-Halt}_\Sigma\langle i\rangle$		$\text{Halt}_\Sigma\langle i, j\rangle$	$\cdots$
$\vdots$						$\ddots$	$\vdots$	

Diagonalization and the Halting Problem — 2

Theorem (The halting problem is unsolvable): The function Halt_Σ which determines whether an arbitrary DTM M_i halts on an arbitrary input α_j is not computable by any DTM.

Proof:
- Δ'-Halt_Σ is not computable by any DTM.
- But Δ'-Halt_Σ is trivially obtainable from Δ-Halt_Σ, so the latter cannot be computable either.
- Since Δ-Halt_Σ is just Halt_Σ restricted to the diagonal, so if Δ-Halt_Σ is not computable, neither can be Halt_Σ. □

Corollary: There exists a language over Σ which is semidecidable (Turing acceptable) but not decidable.

Proof: Just use the language $L = \{\langle i, j\rangle \in \mathbb{N} \times \mathbb{N} \mid \text{HUDTM}_\Sigma\langle i, j\rangle = 1\}$. Encode the numbers in binary with 00 separating them. □

Note: The proof given works for any Σ with at least two elements (regarded as 0 and 1).

- It is possible to establish an undecidability result for Σ containing only one element (will be done shortly).

Proving that Other Problems are Undecidable

- Equipped with the knowledge that the halting problem is undecidable, it is not difficult to establish that many other problems are undecidable as well.

- The most common technique is *reduction*, whose idea is as follows:
  - Let L be a language which defines the problem to be shown undecidable.
  - Assume, to the contrary, that there is a decider M for L.
  - Use M as a component in the construction of a machine which solves the halting problem, a contradiction.

An Example of Reduction

Problem: Show that there is no decider which determines whether or not a given DTM M computes the total *successor* function $n \mapsto n + 1$.

- Let M be any DTM which computes this function.
- Construct the following machine with single input $i \in \mathbb{N}$:

  begin

  Determine M_i using an enumerator;

  Determine α_i using an enumerator;

  Run M_i on α_i; /* Only halting matters */

  Run M on input i; /* Only reached if M_i halts on α_i */

  end
- This machine computes the function which is $i + 1$ if M_i halts on α_i and undefined otherwise.
- Feed a description of this DTM to a decider for the successor function to compute Δ-Halt_Σ.
- So, no such decider can exist.

Black-Box Properties of Computations

- The main idea of the example on the previous slide is not tied to the particular function $n \mapsto n + 1$.
- With minor modifications, it applies to a very wide class of problems.

Definition: A *black-box property* of a DTM M is any statement which concerns solely:

(a) the language which M accepts; and/or

(b) the functions which M computes (of any number of variables).

- A black-box property may not depend upon how M computes.

Examples: Y = black-box property; N = not black-box property.

- M halts on all inputs. (Y)
- $\mathcal{L}(M) = L$ for a given language L. (Y)
- M computes a given partial function f. (Y)
- M returns to its starting state during some computation. (N)
- M uses at most 1000 tape squares during any computation. (N)

Rice's Theorem for Recursive Languages

- An black-box property is called *nontrivial* is some DTMs have that property while others do not.

Theorem (H. Gordon Rice 1953): Let P be a nontrivial black-box property of DTMs. Then the question of whether a given DTM M has that property is undecidable.

- In layman's words, this theorem says that almost nothing about the behavior of DTMs is decidable.

Proof sketch: The general idea follows the reduction example for the successor function $n \mapsto n + 1$.

- Use a decider M for a nontrivial black-box property P do build a decider for the halting problem.
- The resulting contradiction establishes that the decider for P cannot exist.
- There are a few more details to consider; they are sketched briefly on the following slide. $\square$

Proof Idea for Rice's Theorem

- Let P a nontrivial black-box property of DTMs.
- This property partitions the DTMs into:
  - $S_1 =$ all DTMs with property P.
  - $S_2 =$ all DTMs without property P.
- The DTM which never halts on any input must be in one of these classes.
- Assume, without loss of generality, that it is in S_2.
- Let M be any machine in S_1.
- Construct the following machine which takes input $i \in \mathbb{N}$:

begin
  Determine M_i using an enumerator;
  Determine α_i using an enumerator;
  Run M_i on α_i; /* Only halting matters */
  Run M on a suitable input obtained from i; /* Only reached if M_i halts on α_i */
end

- This machine is in S_1 if M_i halts on input α_i and in S_2 if not.
- Thus, a decider for P may be used to solve the halting problem.

A Practical Application of Rice's Theorem

- Recall the following example situation, posed earlier.
- Suppose that you are an assistant in an introductory programming course.
- You must grade 300 programs which are supposed to sort a list of numbers.
- You decide instead that you will write a program which will take as input the program of each student and decide whether or not it is correct.
- An application of Rice's Theorem establishes that it is not possible to write such a program.
- It defines a nontrivial black-box property of machines (programs).

The Application of Rice's Theorem to Functions

- The following questions about a DTM M are undecidable:
  - Is the function f_M which M computes total?
  - Is $f_M(i)$ defined for a given fixed i?
  - Is $f_M(i)$ defined for some $i \in \mathbb{N}$?
  - Is $f_M(i)$ defined for only finitely many $i \in \mathbb{N}$?
  - Is $f_M = g$ for some given function g?
- Note that the last element in the list above is a special case of the "grading program" problem identified earlier.
- It is not possible build a decider which takes as input another program and decides whether or not it computes a specified function.

Total vs. Partial Correctness of Programs

- In a property of the form

  $$f_M = g \text{ for some given total function } g$$

  g may be thought of as a program specification which M must satisfy.

- In program verification, there are two notions of satisfaction of a specification.

  Total correctness: f_M agrees with g everywhere (*i.e.*, $f_M = g$).

  Partial correctness: f_M agrees with g whenever M halts.

- Think of this in terms of a concrete example of a total function.

  Example: The successor function $\quad \text{succ} : n \mapsto n + 1$.

- Even the machine which never halts agrees with succ whenever it halts, so it is a partially correct realization of that function.

- Although partial correctness is "weaker" than total correctness, both are undecidable in the general case, in view of Rice's Theorem.

The Application of Rice's Theorem to Languages

- The following questions about a DTM M are undecidable:
  - Is $\mathcal{L}(M) = L$ for a given fixed L?
    - Is $\mathcal{L}(M) = \emptyset$?
    - Is $\mathcal{L}(M) = \Sigma^*$?
  - Is $\mathcal{L}(M) \subseteq L$ for a given fixed $L \neq \Sigma^*$?
  - Is $L \subseteq \mathcal{L}(M)$ for a given fixed $L \neq \emptyset$?
  - Is $\mathcal{L}(M)$ a regular language?
  - Is $\mathcal{L}(M)$ a context-free language?
  - Is $\mathcal{L}(M)$ the intersection of two CFLs?
  - Is $\mathcal{L}(M)$ the complement of a CFL?
  - Is $\mathcal{L}(M)$ a deterministic CFL?
  - Is $\mathcal{L}(M)$ an inherently ambiguous CFL?
  - Is $\mathcal{L}(M)$ a recursive language?
  - Is $\mathcal{L}(M) = \mathcal{L}(M)^R$?
- and many more...

More Complex Applications of Rice's Theorem

- Consider the question Q:

  Given two DTMs M and M', is $\mathcal{L}(M) = \mathcal{L}(M')$.

- Such questions can often be answered in the negative by showing that a subproblem is not decidable.
- For example, from the previous slide it is known that the following question is undecidable:

  For a given DTM M, is $\mathcal{L}(M) = \emptyset$?

- Thus, fixing M' to be any DTM for which $\mathcal{L}(M') = \emptyset$, a special case of the question Q is obtained which is known to be undecidable.
- If it is not possible to decide $\mathcal{L}(M) = \emptyset$, then it is certainly not possible to decide $\mathcal{L}(M) = \mathcal{L}(M')$ for arbitrary M'.

Problems for Which Rice's Theorem is not Applicable

- Rice's theorem is not directly applicable to questions which ask how rather than just what.

Example: Does an arbitrary DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ return to its initial state q_0 during the computation for input string $\alpha \in \Sigma^*$?

- Such problems may often be solved by choosing an appropriate reduction.
- Let $M' = (Q', \Sigma, \Gamma, \delta', q'_0, \sqcup, F)$ be the DTM with
  - $Q' = Q \cup \{q'_0\}$ ($q'_0 \notin Q$),
  - $\delta' =$ everything in δ plus:
    - $\delta(q'_0, a) = (q_0, a, S)$ for each $a \in \Gamma$.
    - $\delta'(q, a) = (q'_0, a, S)$ whenever $\delta(q, a)$ is undefined.
- M' returns to its initial state q'_0 precisely from the configurations for which M halts.
- Thus if the question of returning to the initial state were decidable, so too would be the halting problem.
- Thus, this question is undecidable.

Showing Semidecidability

- It is often possible to show semidecidability directly by describing how an accepter would work.

Example: Consider $\{M \in \mathrm{DTM}_\Sigma \mid f_M(i) \text{ is defined for } \underline{\text{some}}\ i > 10\}$.

- Build a machine which searches for an $i > 10$ with f_M defined:

  Run M on $i = 10$ for 10 steps.
  Run M on $i = 10, 11$ for 11 steps.
  Run M on $i = 10, 11, 12$ for 12 steps.
  $\vdots$
  Run M on $i = 10, 11, 12, \ldots, i$ for i steps.
  $\vdots$

- Now consider $\{M \in \mathrm{DTM}_\Sigma \mid f_M(i) \text{ is defined for } \underline{\text{all}}\ i > 10\}$.
- This technique does not work!
- This language is not semidecidable.

Languages Which are Not Semidecidable

- Contrast the following two questions about a arbitrary DTM M, relative to a fixed total function g:
  - Q1: Is $f_M(i) = g(i)$ for all $i \in \{0, 1, \ldots, 9\}$?
  - Q2: Is $f_M(i) = g(i)$ for all $i \in \mathbb{N}$?
- Both problems are undecidable, in view of Rice's theorem.
- However, Q2 is "more undecidable" than Q1.
- Q1 is *semidecidable*; if the answer is "yes", that fact can be uncovered by a computation.
  - Run a machine which simulates M on the inputs in $\{0, 1, \ldots, 9\}$, time sharing equitably. If f_M is defined on all ten inputs, this will eventually be determined.
- Neither Q2 nor its complement are semidecidable; any attempt to answer either "yes" or "no" many not halt.
  - It is not possible to timeshare equitably amongst an infinite set of possibilities.
  - This is not a formal argument!

Completely Undecidable Languages

- Call a language $L \subseteq \Sigma^*$ *completely undecidable* if neither L nor its complement $\overline{L} = \Sigma^* \setminus L$ is semidecidable (Turing enumerable).
- To extend this idea to properties of functions requires a little care.
- Recall that DTM_Σ denotes the encodings of all DTMs over Σ.
- Let P be a property of functions, and let $\mathrm{DTM}_\Sigma\langle P\rangle$ denote
$$\{M \in \mathrm{DTM}_\Sigma \mid f_M \text{ has property } P\}.$$
- As a language, the complement of $\mathrm{DTM}_\Sigma\langle P\rangle$ may be divided into two parts.
    - $\mathrm{DTM}_\Sigma\langle\overline{P}\rangle = \{M \in \mathrm{DTM}_\Sigma \mid f_M \text{ does not have property } P\}$.
    - $\{\alpha \in \Sigma^* \mid \alpha \notin \mathrm{DTM}_\Sigma\}$ (*i.e.*, α does not encode a DTM.)
- The second set is always decidable, and almost always uninteresting.
- Thus, it is more direct to call a property P *completely undecidable* if neither $\mathrm{DTM}_\Sigma\langle P\rangle$ nor $\mathrm{DTM}_\Sigma\langle\overline{P}\rangle$ is semidecidable.
- This idea extends naturally to multi-argument functions and other properties of DTMs, but the details are not elaborated here.

Determining Complete Undecidability

- There are tools for establishing that languages and properties are completely undecidable.
  - A second *Rice's theorem (for recursively enumerable languages)*.
  - This theorem is beyond the scope of this course.
- An informal approach is to consider both the language and its complement, and argue that neither can be recursively enumerable.
- As noted on the previous slide, a "practical" example of a problem which fall into this category is the question of whether $f_M = g$ for a fixed function g.
- This is essentially the problem of determining whether a program (M) meets a total specification g.
- That it is totally undecidable says that not only that:
  - it is not possible to determine that a program meets a given specification g, but also
  - it is not possible to determine that a program does not meet a given specification g.

Decision Problems Which Require Other Techniques

Example: Given two CFGs G_1 and G_2, is $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

- It turns out that it is an undecidable question, but....
- Rice's Theorem, and the other reduction techniques which have been presented, cannot address this problem.
- It is a question about a more restricted class of languages.
- Compare it to:

Example: Given two *regular* grammars G_1 and G_2, is $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

- This question is decidable, as was shown earlier in the course.
- The corresponding question for deterministic CFGs was recently shown to be decidable as well [Géraud Sénizergues 1997].
- Techniques for addressing such problems will not be covered in this course.

Decidable Questions about DTMs

- There are some questions about DTMs which are decidable.

Example: For fixed $n \in \mathbb{N}$ and $\alpha \in \Sigma^*$, does the DTM M visit more than n tape squares during the computation with initial configuration $\mathcal{I}\langle M, \alpha\rangle$?

- The number of configurations which the machine can reach is bounded by these conditions.
- Hence, if it runs long enough, it must return to a previous configuration.
- At that point, it is known that the machine will loop forever and hence cannot reach any new configurations.
- Thus, it cannot visit any new tape squares either.

Grammars and Semidecidable Languages

- Recall that a language $L \subseteq \Sigma^*$ is:
  - $\mathcal{L}(M)$ for some NFA M iff it is $\mathcal{L}(G)$ for some regular grammar G;
  - $\mathcal{L}(M)$ for some NPDA M iff it is $\mathcal{L}(G)$ for some CFG G.

Question: Is there a corresponding characterization for DTMs?

- Recall that an *(unrestricted) phrase-structure grammar* (*PSG*) $G = (V, \Sigma, S, P)$ has productions of the form $\alpha \to \beta$ for $\alpha \in (V \cup \Sigma)^* \setminus \{\lambda\}$ and $\beta \in (V \cup \Sigma)^*$.

Theorem: The language $L \subseteq \Sigma^*$ is accepted by some DTM M iff it is generated by some phrase-structure grammar G.

- More formally, $L = \mathcal{L}(M)$ for some DTM M iff $L = \mathcal{L}(G)$ for some phrase-structure grammar G. $\square$

Decidability for Languages over a Single Letter

- The ideas which have been developed surrounding undecidability are based upon an alphabet Σ with at least two letters.
- However, the two letters are needed only to encode DTMs.
- The results themselves apply to single-letter alphabets (*e.g.*, $\Sigma = \{a\}$).
- The argument is simple and is illustrated by example.

Example: Let $L = \{\alpha \in \{a\}^* \mid \text{Length}(\alpha) \geq 3\}$.

- To show that this language is not Turing decidable, let $L' = L$, but with L' regarded as a subset of $\{0, 1, a, b\}^*$.
- If L were decidable, the following scheme would yield a decider for L'.

  begin

  Run a preprocessor which discards all strings containing b, 0, or 1;

  If the input makes it past this preprocessor, run a decider M for L on it;

  end

Enumerators and Semidecidable Languages

- Recall that M is a *(recursive) enumerator* for the language $L \subseteq \Sigma^*$ if M produces the strings of L, one after the other, in a systematic way.

- In this case, the language L is said to be *recursively enumerable*.

Theorem: The language L is recursively enumerable iff it is semidecidable (*i.e.*, Turing enumerable). $\square$

Summary of equivalent properties: Let $L \subseteq \Sigma^*$. The following are equivalent:

(a) $L = \mathcal{L}(M)$ for some DTM M (Turing acceptable, semidecidable).

(b) L is recursively enumerable (by some DTM M).

(c) $L = \mathcal{L}(G)$ for some phrase-structure grammar G. $\square$

Rice's Theorem in Perspective

- Rice's theorem says that nothing nontrivial about the "black-box" behavior of DTMs (and hence programs in a general-purpose language) is decidable.
- This does not mean that nothing is decidable.
- Every algorithm defines a general form of decider.
- Computer scientists develop and implement algorithms for a living.

Principle: Keep in mind, Rice's theorem says that if the inputs to a process are to be *all* programs or *all* machines, then no black-box property can be decided.

- By restricting the scope of the objects being evaluated, many properties are decidable.