Undecidability

Andreas Klappenecker

[based on slides by Prof. Welch]
Sources


Understanding Limits of Computing

- So far, we have studied how efficiently various problems can be solved.
- There has been no question as to whether it is possible to solve the problem.
- If we want to explore the boundary between what can and what cannot be computed, we need a model of computation.
Models of Computation

• Need a way to clearly and unambiguously specify how computation takes place
• Many different mathematical models have been proposed:
  • Turing Machines
  • Random Access Machines
  • ...
• They have all been found to be equivalent!
Church-Turing Thesis

- Conjecture: Anything we reasonably think of as an algorithm can be computed by a Turing Machine (specific formal model).
- So we might as well think in our favorite programming language, or in pseudocode.
- Frees us from the tedium of having to provide boring details
  - in principle, pseudocode descriptions can be converted into some appropriate formal model
Short Review of some Basic Set Theory Concepts
Some Notation

If $A$ and $B$ are sets, then the set of all functions from $A$ to $B$ is denoted by $B^A$.

If $A$ is a set, then $P(A)$ denotes the power set, i.e., $P(A)$ is the set of all subsets of $A$. 
Cardinality

Two sets $A$ and $B$ are said to have the same cardinality if and only if there exists a bijective function from $A$ onto $B$.

[A function is bijective if it is one-to-one and onto]

We write $|A| = |B|$ if $A$ and $B$ have the same cardinality.

[Note that $|A| = |B|$ says that $A$ and $B$ have the same number of elements, even if we do not yet know about numbers!]
Set theorists count

• \(0 = \emptyset\) // the empty set exists by axiom
  This set contains no elements

• \(1 = \{0\} = \{\emptyset\}\) // form the set containing \(\emptyset\)
  This set contains one element

• \(2 = \{0,1\} = \{\emptyset, \{\emptyset\}\}\)
  This set contains two elements

• Keep including all previously created sets as elements of the next set.
Example

Theorem: \(|P(X)| = |2^X|\)

Proof: The bijection from \(P(X)\) onto \(2^X\) is given by the characteristic function. q.e.d.

Example: \(X = \{a,b\}\)

\(\emptyset\) corresponds to \(f(a)=0, f(b)=0\)

\(\{a\}\) corresponds to \(f(a)=1, f(b)=0\)

\(\{b\}\) corresponds to \(f(a)=0, f(b)=1\)

\(\{a,b\}\) corresponds to \(f(a)=1, f(b)=1\)
More About Cardinality

Let $A$ and $B$ be sets.

We write $|A| \leq |B|$ if and only if there exists an injective function from $A$ to $B$.

We write $|A| < |B|$ if and only if there exist an injective function from $A$ to $B$, but no bijection exists from $A$ to $B$.
Cardinality

Cantor’s Theorem: Let $S$ be any set. Then $|S| < |P(S)|$.

Proof: Since the function $i$ from $S$ to $P(S)$ given by $i(s) = \{s\}$ is injective, we have $|S| \leq |P(S)|$.

Claim: There does not exist any function $f$ from $S$ to $P(S)$ that is surjective.

Indeed, $T = \{ s \in S : s \notin f(s) \}$ is not contained in $f(S)$. An element $s$ in $S$ is either contained in $T$ or not.

- If $s \in T$, then $s \notin f(s)$ by definition of $T$. Thus, $T \neq f(s)$.
- If $s \notin T$, then $s \in f(s)$ by definition of $T$. Thus, $T \neq f(s)$.

Therefore, $f$ is not surjective. This proves the claim.
An Uncountable Set

Theorem: The set $\mathbb{N}^\mathbb{N} = \{ f \mid f: \mathbb{N} \rightarrow \mathbb{N} \}$ is not countable.

Proof: We have $|\mathbb{N}| < |P(\mathbb{N})|$ by Cantor’s theorem. Since $|P(\mathbb{N})| = |2^\mathbb{N}|$ and $2^\mathbb{N}$ is a subset of $\mathbb{N}^\mathbb{N}$ we can conclude that

$$|\mathbb{N}| < |P(\mathbb{N})| = |2^\mathbb{N}| \leq |\mathbb{N}^\mathbb{N}|.$$ q.e.d.
Alternate Proof: 
The Set $\mathbb{N}^\mathbb{N}$ is Uncountable

Seeking a contradiction, we assume that the set of functions from $\mathbb{N}$ to $\mathbb{N}$ is countable. Let the functions in the set be $f_0, f_1, f_2, \ldots$ We will obtain our contradiction by defining a function $f^d$ (using "diagonalization") that should be in the set but is not equal to any of the $f_i$'s.
Diagonalization

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>4</td>
<td>14</td>
<td>34</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>55</td>
<td>32</td>
<td>777</td>
<td>3</td>
<td>21</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>90</td>
<td>2</td>
<td>5</td>
<td>21</td>
<td>66</td>
<td>901</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>44</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>f</td>
<td>80</td>
<td>56</td>
<td>32</td>
<td>12</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>f</td>
<td>43</td>
<td>345</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>
Diagonalization

- Define the function: \( f^d(n) = f_n(n) + 1 \)
- In the example:
  - \( f^d(0) = 4 + 1 = 5 \), so \( f^d \neq f_0 \)
  - \( f^d(1) = 32 + 1 = 33 \), so \( f^d \neq f_1 \)
  - \( f^d(2) = 5 + 1 = 6 \), so \( f^d \neq f_2 \)
  - \( f^d(3) = 7 + 1 = 8 \), so \( f^d \neq f_3 \)
  - \( f^d(4) = 3 + 1 = 4 \), so \( f^d \neq f_4 \)
  - etc.
Uncomputable Functions Exist!

Consider all programs (in our favorite model) that compute functions in $\mathbb{N}^\mathbb{N}$.

The set $\mathbb{N}^\mathbb{N}$ is uncountable, hence cannot be enumerated.

However, the set of all programs can be enumerated (i.e., is countable).

Thus there must exist some functions in $\mathbb{N}^\mathbb{N}$ that cannot be computed by a program.
Set of All Programs is Countable

- Fix your computational model (e.g., programming language).
- Every program is finite in length.
- For every integer $n$, there is a finite number of programs of length $n$.
- Enumerate programs of length 1, then programs of length 2, then programs of length 3, etc.
Uncomputable Functions

• Previous proof just showed there must exist uncomputable functions
• Did not exhibit any particular uncomputable function
• Maybe the functions that are uncomputable are uninteresting...
• But actually there are some VERY interesting functions (problems) that are uncomputable
The Halting Problem
The Function Halt

• Consider this function, called Halt:
  • input: code for a program P and an input X for P
  • output: 1 if P terminates (halts) when executed on input X, and 0 if P doesn't terminate (goes into an infinite loop) when executed on input X

• By the way, a compiler is a program that takes as input the code for another program

• Note that the input X to P could be (the code for) P itself
  • in the compiler example, a compiler can be run on its own code
The Function Halt

- We can view Halt as a function from N to N:
  - P and X can be represented in ASCII, which is a string of bits.
  - This string of bits can also be interpreted as a natural number.
- The function Halt would be a useful diagnostic tool in debugging programs.
Halt is Uncomputable

• Suppose in contradiction there is a program $P_{\text{halt}}$ that computes Halt.

• Use $P_{\text{halt}}$ as a subroutine in another program, $P_{\text{self}}$.

• Description of $P_{\text{self}}$:
  • input: code for any program $P$
  • constructs pair $(P,P)$ and calls $P_{\text{halt}}$ on $(P,P)$
  • returns same answer as $P_{\text{halt}}$
The diagram illustrates the concept of a self-referential program $P_{self}$. When the program takes an input $P$, it examines its own output on input $P$. If $P$ halts on input $P$, it outputs 1; otherwise, it outputs 0.
Halt is Uncomputable

• Now use $P_{\text{self}}$ as a subroutine inside another program $P_{\text{diag}}$.

• Description of $P_{\text{diag}}$:
  • input: code for any program $P$
  • call $P_{\text{self}}$ on input $P$
  • if $P_{\text{self}}$ returns 1 then go into an infinite loop
  • if $P_{\text{self}}$ returns 0 then output 1

• $P_{\text{diag}}$ on input $P$ does the opposite of what program $P$ does on input $P$
The diagram illustrates the process described by the Turing machine $P_{\text{diag}}$. It shows how $P_{\text{diag}}$ operates on input $P$, where $P_{\text{self}}$ is the part of the machine that checks if $P$ halts on input $P$. The state $P_{\text{halt}}$ outputs 1 if $P$ halts on input $P$ and 0 if $P$ doesn't halt on input $P$. The process is cyclic, indicating the self-referential nature of the algorithm.
Halt is Uncomputable

- Review behavior of $P_{\text{diag}}$ on input $P$:
  - If $P$ halts when executed on input $P$, then $P_{\text{diag}}$ goes into an infinite loop
  - If $P$ does not halt when executed on input $P$, then $P_{\text{diag}}$ halts (and outputs 1)

- What happens if $P_{\text{diag}}$ is given its own code as input? It either halts or doesn't.
  - If $P_{\text{diag}}$ halts when executed on input $P_{\text{diag}}$, then $P_{\text{diag}}$ goes into an infinite loop
  - If $P_{\text{diag}}$ doesn't halt when executed on input $P_{\text{diag}}$, then $P_{\text{diag}}$ halts

Contradiction
Halt is Uncomputable

• What went wrong?
• Our assumption that there is an algorithm to compute Halt was incorrect.
• So there is no algorithm that can correctly determine if an arbitrary program halts on an arbitrary input.
Undecidability
Undecidability

• The analog of an uncomputable function is an **undecidable set**.

• The theory of what can and cannot be computed focuses on identifying sets of strings:
  - an algorithm is required to "decide" if a given input string is in the set of interest
  - similar to deciding if the input to some NP-complete problem is a YES or NO instance
Undecidability

• Recall that a (formal) language is a set of strings, assuming some encoding.

• Analogous to the function Halt is the set $H$ of all strings that encode a program $P$ and an input $X$ such that $P$ halts when executed on $X$.

• There is no algorithm that can correctly identify for every string whether it belongs to $H$ or not.
More Reductions

• For NP-completeness, we were concerned with (time) complexity of problems:
  • reduction from P1 to P2 had to be fast (polynomial time)

• Now we are concerned with computability of problems:
  • reduction from P1 to P2 just needs to be computable, don't care how slow it is
Many-One Reduction

\[ f: \text{all strings over } L_1\text{'s alphabet} \rightarrow \text{all strings over } L_2\text{'s alphabet} \]
Many-One Reduction

- YES instances map to YES instances
- NO instances map to NO instances
- computable (doesn't matter how slow)
- Notation: \( L_1 \leq_m L_2 \)
- Think: \( L_2 \) is at least as hard to compute as \( L_1 \)
Many-One Reduction Theorem

Theorem: If $L_1 \leq_m L_2$ and $L_2$ is computable, then $L_1$ is computable.

Proof: Let $f$ be the many-one reduction from $L_1$ to $L_2$. Let $A_2$ be an algorithm for $L_2$. Here is an algorithm $A_1$ for $L_1$.

- input: $x$
- compute $f(x)$
- run $A_2$ on input $f(x)$
Implication

- If there is no algorithm for $L_1$, then there is no algorithm for $L_2$.
- In other words, if $L_1$ is undecidable, then $L_2$ is also undecidable.
- Pay attention to the direction!
Example of a Reduction

- Consider the language $L_{NE}$ consisting of all strings that encode a program that halts (does not go into an infinite loop) on at least one input.
- Use a reduction to show that $L_{NE}$ is not decidable:
  - Show some known undecidable language $\leq_m L_{NE}$.
  - Our only choice for the known undecidable language is $H$ (the language corresponding to the halting problem)
  - So show $H \leq_m L_{NE}$.
Example of a Reduction

- Given an arbitrary H input (encoding of a program P and an input X for P), compute an $L_{\text{NE}}$ input (encoding of a program $P'$)
  - such that P halts on input X if and only if $P'$ halts on at least one input.
- Construction consists of writing code to describe $P'$.
- What should $P'$ do? It's allowed to use P and X
Example of a Reduction

• The code for $P'$ does this:
  • input $X'$:
  • ignore $X'$
  • call program $P$ on input $X$
  • if $P$ halts on input $X$ then return whatever $P$ returns

• How does $P'$ behave?
  • If $P$ halts on $X$, then $P'$ halts on every input
  • If $P$ does not halt on $X$, then $P'$ does not halt on any input
Example of a Reduction

• Thus if \((P,X)\) is a YES input for \(H\) (meaning \(P\) halts on input \(X\)), then \(P'\) is a YES input for \(L_{NE}\) (meaning \(P'\) halts on at least one input).

• Similarly, if \((P,X)\) is \(NO\) input for \(H\) (meaning \(P\) does not halt on input \(X\)), then \(P'\) is a \(NO\) input for \(L_{NE}\) (meaning \(P'\) does not halt on even one input).

• Since \(H\) is undecidable, and we showed \(H \leq_m L_{NE}\), \(L_{NE}\) is also undecidable.
Generalizing Such Reductions

• There is a way to generalize the reduction we just did, to show that lots of other languages that describe properties of programs are also undecidable.

• Focus just on programs that accept languages (sets of strings):
  • I.e., programs that say YES or NO about their inputs
  • Ex: a compiler tells you YES or NO whether its input is syntactically correct
Properties About Programs

- Define a property about programs to be a set of strings that encode some programs.
  - The "property" corresponds to whatever it is that all the programs have in common
- Example:
  - Program terminates in 10 steps on input y
  - Program never goes into an infinite loop
  - Program accepts a finite number of strings
  - Program contains 15 variables
  - Program accepts 0 or more inputs
Functional Properties

- A property about programs is called **functional** if it just refers to the language accepted by the program and not about the specific code of the program
  - Program terminates in 10 steps on input y (n.f.)
  - Program never goes into an infinite loop (f.)
  - Program accepts a finite number of strings (f.)
  - Program contains 15 variables (n.f.)
Nontrivial Properties

- A functional property about programs is **nontrivial** if some programs have the property and some do not.

- Example of nontrivial programs:
  - Program never goes into an infinite loop
  - Program accepts a finite number of strings

- Example of a trivial program:
  - Program accepts 0 or more inputs
Rice's Theorem

• Every nontrivial (functional) property about programs is undecidable.
• The proof is a generalization of the reduction shown earlier.
• Very powerful and useful theorem:
  • To show that some property is undecidable, only need to show that is nontrivial and functional, then appeal to Rice's Theorem
Applying Rice's Theorem

• Consider the property "program accepts a finite number of strings".

• This property is functional:
  • it is about the language accepted by the program and not the details of the code of the program

• This property is nontrivial:
  • Some programs accept a finite number of strings (for instance, the program that accepts no input)
  • some accept an infinite number (for instance, the program that accepts every input)

• By Rice's theorem, the property is undecidable.
Implications of Undecidable Program Property

• It is not possible to design an algorithm (write a program) that can analyze any input program and decide whether the input program satisfies the property!
• Essentially all you can do is simulate the input program and see how it behaves
  • but this leaves you vulnerable to an infinite loop