CS245 — Logic and Computation

Undecidability and the Halting Problem

Jonathan Buss

with thanks to

Borzoo Bonakdarpour, Daniela Maftuleac and Tyrel Russell
What is Computability?

From an informal, intuitive perspective, what does it mean for something to be computable?

One natural interpretation is that something is *computable if it can be calculated by a systematic procedure — a way to automatically arrive at a result.*

*One might think that, given enough resources and a sufficiently sophisticated program, a computer could solve any problem.*

*However, some problems cannot be automated.*
What is “Systematic”?

An instruction such as “guess the correct answer” does not seem to be systematic.

An instruction such as “try all possible answers” is less clear cut: it depends on whether the possible answers are finite or infinite in number.

Henceforth we will imagine working with a specific programming language, and using this as the definition of reasonable instructions (think of Java, C, or whatever you like).

In practice, we will describe algorithms in informal pseudo-code, but with enough detail that their programmability should be clear.
A decision problem is a problem which calls for an answer of either yes or no, on each possible input.

In other words, an input is a yes/no question; the problem specifies what the answer should be, for each input.

Examples of decision problems:

1. Given a formula of propositional logic, is it satisfiable?
2. Given a positive integer, is it prime?
3. Given two vertices of a graph, is there a path between them?
4. Given a multivariate polynomial, does it have any integer roots?
5. Given a program and input, will the program terminate on the input?
A Curious Problem

Given: a positive integer $n$.

Question: Does the following terminate, starting from that value of $n$?

$$
\text{while ( n > 1 )} \{
\quad \text{if ( n is even )} \{ \n\quad \quad n = \frac{n}{2} ; 
\quad \}
\quad \text{else} \{ \n\quad \quad n = 3*n + 1 ; 
\quad \}
\}
$$

Nobody knows whether the answer is always "yes"!

Nevertheless, the problem is a decision problem. For a given $n$, the answer is either "yes" or "no".
A Curious Problem

Given: a positive integer \( n \).

Question: Does the following terminate, starting from that value of \( n \)?

```java
while ( n > 1 ) {
    if ( n is even ) {
        n = n/2 ;
    } else {
        n = 3*n + 1 ;
    }
}
```

Nobody knows whether the answer is always “yes”!

Nevertheless, the problem is a decision problem. For a given \( n \), the answer is either “yes” or “no”. 
Decidable and Undecidable Problems

A decision problem is *decidable* if there is an algorithm that, given an input to the problem,

- outputs “yes” (or “true”) if the input has answer “yes” and
- outputs “no” (or “false”) if the input has answer “no”.

A problem is *undecidable* if it is not decidable.

Proving decidability and undecidability:

- To prove that $D$ is decidable, one can give an algorithm and prove it works.
- To prove that $D$ is undecidable, one must show that every algorithm fails to decide $D$. 
To prove undecidability, we need a precise definition of “algorithm”. That is, we need to know the domain of the quantifier “every algorithm ...”.

Many possibilities exist.

- A programming language (precisely defined).
- A description of computer hardware (precisely defined).
- Other definition of “computable function”.

We shall do none of those here. Instead, we shall rely on our intuitive notion of algorithm, as given by a programming language. Some properties are required, however, ....
We make the following assumptions about our programming language.

1. Any reasonable “algorithm” can be implemented in the language.
2. In particular, there is an “executor” or “interpreter” that, given any program, can execute that program on any desired input. (E.g., Dr. Racket.)
3. When the interpreter executes a program, it can isolate the program, so that the program cannot modify the interpreter itself, nor any of the interpreter’s own private data.
One of the best-known undecidable problems is the *Halting Problem*. Given a program $P$ and an input $I$, will $P$ terminate if run on input $I$?

An example that does not terminate:

```java
while ( true ) {} ;
```

Can we write a program that takes as an input any program $P$ and input $I$ for $P$ and returns true, if the program terminates (halts) on $I$ and returns false, otherwise?
Testing Whether a Program Halts

Somewhat more formally, can the following program exist?

```c
bool halts ( program P, input I )

// When given a program P and input I, the function should
//  * Always halt with an answer.
//  * If a call "P ( I )" would halt [terminate],
//    - then halts ( P, I ) returns true.
//  * If a call "P ( I )" would run forever,
//    - then halts ( P, I ) returns false.

{ ...
```
Testing Whether a Program Halts

**Theorem.**
No computer program can perform the task required of “halts”, correctly for all programs and inputs.

That is, the Halting Problem is undecidable.
Proof: By contradiction.

Assume that there exists some function such that “\texttt{halts ( \( P, I \) )}” returns \texttt{true} if the program \( P \) halts on input \( I \) and returns \texttt{false} if \( P \) does not halt on input \( I \).

Two significant points to note:

• A program \( \texttt{halts} \) can be called by some other program.
• The second argument to “\texttt{halts ( \( P, I \) )}” may be anything — which includes \( P \) itself (or a description of it).
A Program Calling Itself

Let’s consider the following function, which uses `halts`.

```c
bool self_halt ( P ) {
    return halts ( P, P );
}
```

This should determine whether $P$ terminates when given itself as input.

What happens if we call “`self_halt ( self_halt )`”? 

self\_halt ( self\_halt )

⇒ halts ( self\_halt , self\_halt )

⇒ ...

⇒ \[
\begin{cases} 
\text{true,} & \text{if } self\_halt ( self\_halt ) \text{ halts} \\
\text{false,} & \text{if } self\_halt ( self\_halt ) \text{ does not halt.}
\end{cases}
\]

Since evaluation of \text{halts} always terminates, the evaluation of self\_halt ( self\_halt ) also terminates.

Since \text{halts} gives the correct answer, the final result must be “true”.
OK, now let’s consider the following function.

```cpp
bool halt_if_loop ( P ) {
    if ( halts ( P, P ) { 
        while ( true ) { } 
    } 
    return true ; 
}
```

What happens if we invoke `halt_if_loop` with itself as its argument?
```
halt_if_loop ( halt_if_loop )
  ⇒ if ( halts ( halt_if_loop , halt_if_loop ) ) { ...
  ⇒ ...
  ⇒ \[
    \begin{cases} 
      \text{while ( true )} \{ \}, \\
      \quad \text{if } \text{halt}_\text{if}_\text{loop} ( \text{halt}_\text{if}_\text{loop} ) \text{ halts}, \\
      \quad \text{return } \text{true}, \\
      \quad \text{if } \text{halt}_\text{if}_\text{loop} ( \text{halt}_\text{if}_\text{loop} ) \text{ does not halt.}
    \end{cases}
  \]
```

The call "\text{halt}_\text{if}_\text{loop} ( \text{halt}_\text{if}_\text{loop} )" terminates iff the call "\text{halt}_\text{if}_\text{loop} ( \text{halt}_\text{if}_\text{loop} )" does not terminate.

Program \text{halt}_\text{if}_\text{loop} cannot exist!
Thus also program \text{halts} cannot exist.
The Technique: “Diagonalization”

The technique used in the proof of the undecidability of the halting problem is called **diagonalization**.

It was originally devised by Georg Cantor (in 1873) for a different purpose.

Cantor was concerned with the problem of measuring the sizes of infinite sets. Are some infinite sets larger than others?

**Example.** The set of even natural numbers is the same size (!!) as the set of all natural numbers.

**Example.** The set of all infinite sequences over \{0, 1\} is larger than the set of natural numbers.

(Idea of proof: (A) Assume that each sequence has a number. (B) find a sequence that is different from every numbered sequence.)
A set $S$ is **countable** if there is a one-to-one correspondence between $S$ and the set of natural numbers ($\mathbb{N}$).

How do we prove that a set is **uncountable**?
Cantor’s diagonal argument

Consider an infinite sequence \( S = (s_1, s_2, ...) \), where each element \( s_i \) is an infinite sequence of 1s or 0s (\( s_i \) is a binary string of infinite length).

\[
\begin{align*}
    s_1 &= (0, 0, 0, 0, 0, 0, ...) \\
    s_2 &= (1, 1, 1, 1, 1, 1, ...) \\
    s_3 &= (0, 1, 0, 1, 0, 1, ...) \\
    s_4 &= (1, 0, 1, 0, 1, 0, ...) \\
    s_5 &= (1, 1, 0, 0, 1, 1, ...) \\
    s_6 &= (0, 0, 1, 1, 0, 0, ...) \\
    \ldots
\end{align*}
\]
Cantor’s diagonal argument

There is a sequence \( \bar{s} \) such that if \( s_{n,n} = 1 \), then \( \bar{s}_n = 0 \), and otherwise \( \bar{s}_n = 1 \).

\[
\begin{align*}
s_1 &= (0, 0, 0, 0, 0, 0, \ldots) \\
s_2 &= (1, 1, 1, 1, 1, 1, \ldots) \\
s_3 &= (0, 1, 0, 1, 0, 1, \ldots) \\
s_4 &= (1, 0, 1, 0, 1, 0, \ldots) \\
s_5 &= (1, 1, 0, 0, 1, 1, \ldots) \\
s_6 &= (0, 0, 1, 1, 0, 0, \ldots) \\
&\vdots
\end{align*}
\]

Define \( \bar{s} = (1, 0, 1, 1, 0, 1, \ldots) \).

Then \( \bar{s} \) is not any of the sequences \( s_i \) on the list \( S \).
Cantor’s diagonal argument

By construction, \( \overline{s} \) is not contained in the countable sequence \( S \).

Let \( T \) be a set consisting of all infinite sequences of 0s and 1s. By definition, \( T \) must contain \( S \) and \( \overline{s} \).

Since \( \overline{s} \) is not in \( S \), the set \( T \) cannot coincide with \( S \).

Therefore, \( T \) is uncountable; it cannot be placed in one-to-one correspondence with \( \mathbb{N} \).
Finding Other Undecidable Problems

The “diagonalization” we did for the halting problem is rather tricky. In order to prove some other problem undecidable, it helps to have another method.

We shall describe the method known as *reduction* between problems. (An informal description, only — like our “definition” of algorithm.)

First, we note that you already know the method, even though you may not know its name....
Reduction Between Problems

A reduction from problem A to problem B is an algorithm (or program) to solve problem A that relies on an algorithm (or program) to solve B.

Example:

Problem A: given an array, find its median element. (That is, the element s.t. half of the elements are smaller and half are larger.)

Problem B: given an array, sort it.

An algorithm for problem A:

• Sort the array (i.e., solve problem B).
• Look at position $n/2$, where $n$ is the length of the array.

The above is a reduction from A to B.
The Significance of a Reduction

Suppose we have a reduction from A to B.

• If we can solve B, then we can solve A.
  Even if we didn’t write the code for B — maybe it’s in a “library” — we can still use it.
  This is a very common and useful technique.

• Conversely, if A is undecidable, then B is undecidable.
  No one can write code for A, since it’s undecidable.
  Thus no one can write code for B, either.
  It would provide code for A, but such code doesn’t exist.
Undecidability Via Reduction

Let problem A be the Halting Problem:
Given P and I, does P halt on input I?

Let problem B be the “Looping Problem”:
Given P and I, does P run forever on input I?

Suppose we had an algorithm loops for the Looping Problem. Then we could write the following:

```cpp
bool halts ( P, I ) {
    if ( loops ( P, I ) ) {
        return false ;
    } else {
        return true ;
    }
}
```
We have the program

```cpp
bool halts ( P, I ){
    if ( loops ( P, I ) ) {
        return false ;
    } else {
        return true ;
    }
}
```

If the function `loops` solves the Looping Problem, then the function `halts` solves the Halting Problem. Since the latter is impossible, the former is also impossible.

That is, the Looping Problem is undecidable.
Another Undecidable Problem

Provability of a formula in FOL:

Given a formula $\varphi$, does $\varphi$ have a proof?
(Or, equivalently, is $\varphi$ valid?)

**Theorem.** Provability is undecidable.

**Lemma.** Given a program $P$ and an input $I$, one can compute a FOL formula $\varphi_{P,I}$ that expresses,

“The computation of $P$ on $I$ halts.”

I.e., there exists a sequence of states that is a correct, halting computation.
Outline of proof: Suppose that some algorithm $Q$ decides provability.

An algorithm:

1. Given $P$ and $I$, produce the formula $\varphi_{P,I}$ (from the lemma).
   
   If $P$ halts, then if $\varphi_{P,I}$ is provable — give the computation.
   
   If $P$ does not halt, then $\varphi_{P,I}$ is not provable (due to soundness).

2. Run algorithm $Q$ on input $\varphi_{P,I}$, and give the same answer (Y/N).

This algorithm decides Halting.

But no algorithm decides the Halting problem.

Thus $Q$ cannot exist — no algorithm decides provability.
Another Undecidable Problem

The Post Correspondence Problem (devised by Emil Post)

Given a finite sequence of pairs \((s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\) such that all \(s_i\) and \(t_i\) are binary strings of positive length, is there a sequence of indices \(i_1, i_2, \ldots, i_n\) with \(n \geq 1\) such that the concatenation of the strings \(s_{i_1} s_{i_2} \ldots s_{i_n}\) equals \(t_{i_1} t_{i_2} \ldots t_{i_n}\)?
An Instance of the Post Correspondence Problem

Suppose we have the following pairs: (1,101), (10,00), (011,11). Can we find a solution for this input?

Yes. Indices (1,3,2,3) work, since $s_1s_3s_2s_3 = 101110011 = t_1t_3t_2t_3$.

What about the pairs (001,0), (01,011), (01,101), (10,001)? In this case, there is no sequence of indices. Remember that an index can be used arbitrarily many times. This gives us some indication that the problem might be unsolvable in general, as the search space is infinite.
An Instance of the Post Correspondence Problem

Suppose we have the following pairs: \((1,101), (10,00), (011,11)\). Can we find a solution for this input?

Yes. Indices \((1,3,2,3)\) work, since \(s_1s_3s_2s_3 = 10110011 = t_1t_3t_2t_3:\)

<table>
<thead>
<tr>
<th></th>
<th>(s_1)</th>
<th>(s_3)</th>
<th>(s_2)</th>
<th>(s_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(t_1)</td>
<td>(t_3)</td>
<td>(t_2)</td>
<td>(t_3)</td>
</tr>
</tbody>
</table>

What about the pairs \((001,0), (01,011), (01,101), (10,001)\)?

In this case, there is no sequence of indices.

Remember that an index can be used arbitrarily many times. This gives us some indication that the problem might be unsolvable in general, as the search space is infinite.
The problem

Given a multivariate polynomial with integer co-efficients, does it have any roots that are (tuples of) integers?

is undecidable.

The proof is similar in principle: show that deciding whether an equation has integral solutions allows deciding whether a program halts.

The details, however, get quite complicated.

The issue was publicly raised in 1900 (by Hilbert) and not solved until 1971 (by Matjasevič and Robinson).