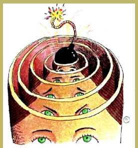


Module Home Page

Title Page

Page 1 of 12

Back

Full Screen

Close

Quit

Lecture 35:
The Halting Problem

Aims:

- To introduce the notions of computability and non-computability; and
- To prove that the Halting Problem is non-computable.

35.1. Introduction

- Some problems are unsolvable.
- We're talking about
  - important and interesting problems,
  - precisely-defined problems (not vague problems such as 'compose a poem on topic T')

  for which
  - no matter how clever we are,
  - no matter how much execution time, memory, etc. we allow

  there is provably no algorithm to solve the problem.
- Our terminology: **computable** and **non-computable**.
- Here are some examples of non-computable problems that we brushed up against earlier in the module:
  - Tiling Problems for the integer grid (lecture 1);
  - Take in a program P; return YES if P will ever perform a division by zero, else return NO (lecture 10);
  - Take in a program P; return an adequate set of test data for program P (lecture 10);
  - Take in a problem specification PS and a program P; return a proof that P solves PS (if it does solve it), else return **fail** (lectures 10 & 22).
- But we're going to look first at the most famous of all non-computable problems, the *Halting Problem*.

35.2. The Halting Problem

35.2.1. What is the Halting Problem?

- As we have discussed previously, compilers detect errors in programs (prior to translating the source program into a target program). In particular, they detect errors in the syntax and the static semantics. They cannot, in general, detect errors in the dynamic semantics. There is a good reason for this: it's impossible. The Halting Problem is one of the problems we might like compilers to solve, but which no compiler can ever solve.

- It is traditional to explain this problem (and computability in general) using something called a Turing machine. However, we will explain this topic using MOCCA programs, which is just as good. (In fact, we could use Java programs, or C programs, or Pascal programs, or…, and these would be just as good also.) We'll come back to Turing machines in a few lectures' time, and that's when we can show that using MOCCA (or Java, etc.) is just as good.

- The following problem is non-computable:

Problem 35.1. *The Halting Problem*

Parameters:	*A MOCCA program P, and a potential input x to P.*
Returns:	*YES if P would have terminated had we run it on input x; NO otherwise.*

(We are assuming that P expects just one input x. Again nothing hangs on this. Indeed, if P requires more than one input, we can think of it as taking in a single input: the concatenation of the individual inputs.)

- E.g. A solution to the Halting Problem should return YES when given the following program and input $x = 11$:

```
while x ≠ 1
{    x := x − 2;
}
```

- E.g. A solution to the Halting Problem should return NO when given the previous program and input $x = 12$.
- **Question:** Why can't we simply run P on x and see what happens?

35.2.2. Preliminaries

- We will prove that the Halting Problem is non-computable.
- We want to prove:

> There is no MOCCA program which, upon accepting any pair $\langle P, x\rangle$ consisting of the text of a legal MOCCA program P and a string of symbols x, terminates after some finite amount of time, and outputs YES if P halts when run on input x and NO if P does not halt when run on input x.

- A key observation is that such a program, if it exists, must work for every pair $\langle P, x\rangle$. And, of course, such a program, if it exists, is itself a legal MOCCA program, so it has to work on itself.
- We shall prove the nonexistence of such a program by contradiction.

- Proof by contradiction: assume the negation of what we want to prove, and then derive a contradiction.

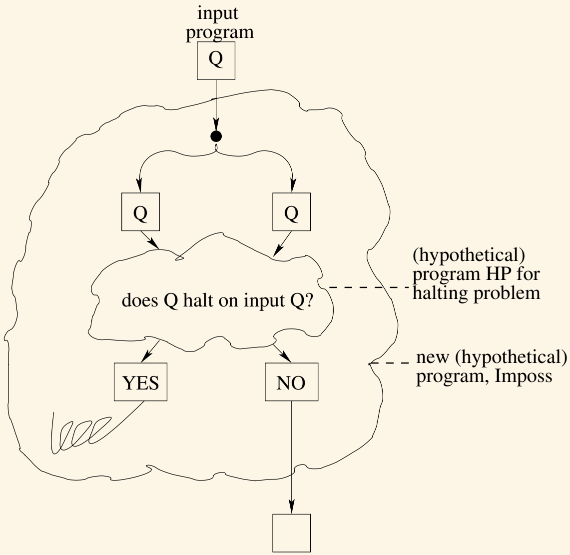
$$\frac{\frac{W_1}{W_2 \wedge \neg W_2}}{\neg W_1}$$

- Sketch of our proof:
  - We will assume such a program does exist. Call it HP.
  - We will construct another program, *Imposs*, that uses perfectly reasonable operations, but it also uses HP as a procedure.
  - We will show that there's something wrong with *Imposs*: there is a particular input on which it cannot terminate and also on which it cannot not terminate! This is impossible (contradiction)!
  - Since *Imposs* will have been constructed in a perfectly reasonable way, the only part that can be responsible for the contradiction is the procedure HP.
  - Therefore, HP cannot exist.

35.2.3. The Proof

- Assume a MOCCA program for solving the Halting Problem does exist. Call it HP.
- We construct a new MOCCA program, *Imposs*, as follows:

Introduction
The Halting Problem
Module Home Page
Title Page
Page 6 of 12
Back
Full Screen
Close
Quit



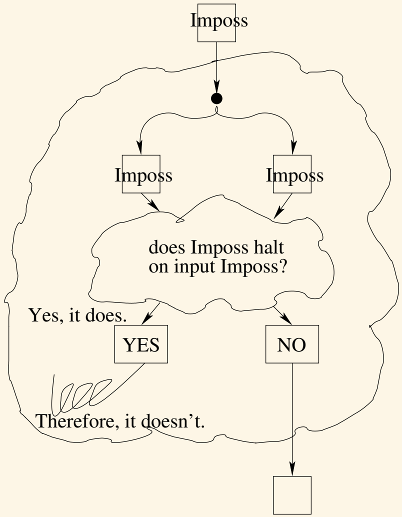
- Here's *Imposs* again in a more textual form:

Algorithm: $\mathit{Imposs}(Q)$

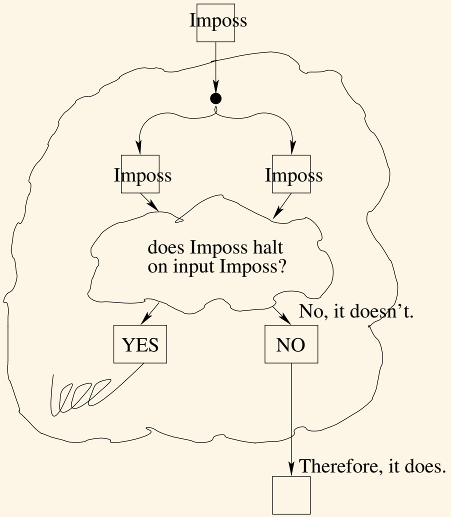
$\mathit{ans} := \mathrm{HP}(Q, Q)$;
if $\mathit{ans} = \mathrm{YES}$
{ **while true**
  {
  }
}
else
{ **return** whatever ;
}

- And here's an explanation of program *Imposs* in words.
  - *Imposs* is a MOCCA program which takes in a single input, Q, which itself should be the text of a MOCCA program.
  - *Imposs* begins by making a copy of Q.
  - *Imposs* then calls HP. Recall that HP expects two inputs, so when *Imposs* activates HP, it supplies the two copies of Q.
  - HP must eventually return either YES (i.e. program Q does terminate for input Q) or NO (i.e. program Q does not terminate for input Q).
  - *Imposs* reacts as follows:
    - If HP returned YES, *Imposs* enters an infinite loop.
    - If HP returned NO, *Imposs* terminates (the output being unimportant).

- We will now show that *Imposs* is a logical impossibility.
- Specifically, we will show that there is a certain input on which *Imposs* cannot terminate, but it also cannot *not* terminate!
- The input that causes the problem is *Imposs* itself!
- We'll spell out two cases
  - We'll suppose *Imposs* applied to itself does terminate.
  - Then we'll suppose *Imposs* applied to itself does not terminate.
- Suppose *Imposs*, when given itself as input, does terminate.
- What happens? Two copies of *Imposs* are made. These are fed to HP, which returns an answer. It tells us, in this case, whether *Imposs* terminates on *Imposs*. We are assuming that it does. So HP would return YES. However, at this point, we enter the infinite loop, so *Imposs* never terminates.
- But this means that, when we assume that *Imposs* does terminate on *Imposs*, then *Imposs* does not terminate on *Imposs*!



- Suppose instead that *Imposs*, when given itself an input, does *not* terminate.
- What happens? We copy *Imposs*. We feed the copies into HP. It returns NO. However, at this point, *Imposs* terminates.
- But this means that, when we assume that *Imposs* does not terminate on *Imposs*, then *Imposs* does terminate on *Imposs*!



- From these two cases, we have found that
  - *Imposs* cannot terminate when run on itself, and
  - *Imposs* cannot not terminate when run on itself!

  Something is very wrong with *Imposs*.

- But *Imposs* was constructed quite legally. So, the only part of *Imposs* that can be held responsible is HP.
- We conclude that a program HP, solving the Halting Problem, simply cannot exist.
- (How much did this depend on MOCCA? Not at all.)

Acknowledgements

This treatment is mostly based on [Har92]. I acknowledge also the influence of Achim Jung's *Models of Computation* course notes [Jun].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

Module Home Page
Title Page
Back
Full Screen
Close
Quit

References

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.

[Jun] A. Jung. Models of Computation (Course Notes).