



A simple way to explain undecidability

Stavros Tripakis

► To cite this version:

| Stavros Tripakis. A simple way to explain undecidability. 2019. hal-02068449

HAL Id: hal-02068449

<https://hal.science/hal-02068449>

Preprint submitted on 14 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A simple way to explain undecidability

Stavros Tripakis
Northeastern University

March 3, 2019

1 Introduction

The undecidability of the halting problem for Turing machines is a cornerstone result in computer science [3]. Many students are exposed to rigorous proofs of this result in textbooks on computability theory, such as [1, 2]. These proofs rely on knowing what a Turing machine is and how it operates. This is of course necessary for a rigorous exposition of the undecidability result.

Here's a way to explain undecidability without having to explain Turing machines. This is by no means a rigorous proof, but I have found it a useful way to introduce students to the undecidability concept.

2 Undecidability without Turing machines

The argument is as follows. Suppose there exists a program, call it `TERMINATOR`, which can decide termination of other programs. `TERMINATOR` takes as input a program P and an input x and returns YES if P terminates on x , and NO if P does not terminate on x . `TERMINATOR` always terminates and gives a YES or NO answer.

Now build a new program Q as follows:

```
Q(P) := if (TERMINATOR(P,P) = YES)
        then loop forever
        else return YES.
```

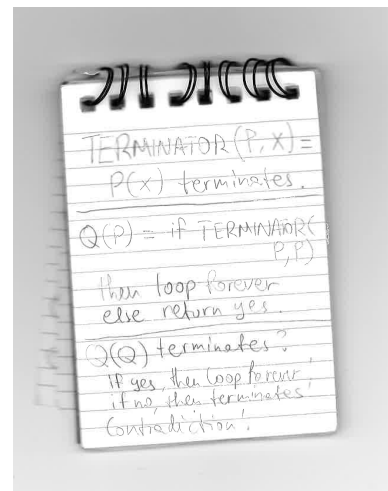
Q takes as input a program P and runs `TERMINATOR` on P , with the input x also set to P . If `TERMINATOR` returns YES, then Q goes into an infinite loop. Otherwise, Q returns YES (the actual returned value is in fact not important).

Assuming that program `TERMINATOR` exists, Q is also a valid program (which calls `TERMINATOR` as a subroutine). So Q can be given as an input to itself, and we can ask: does $Q(Q)$ terminate? There are two cases:

- Either `TERMINATOR`(Q, Q) returns YES, which means that $Q(Q)$ terminates. But in that case, Q takes the **then** branch and loops forever, which means it does not terminate!
- Or `TERMINATOR`(Q, Q) returns NO, which means that $Q(Q)$ does not terminate. But in that case, Q takes the **else** branch and returns a value, which means that it does terminate!

In both cases, we reach a contradiction. Therefore, `TERMINATOR` cannot exist.

This “proof” is both simple and short: it fits in one page of a small note pad, as the picture above shows.



References

- [1] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*, 2/e. Prentice-Hall, 1997.
- [2] Michael Sipser. *Introduction to the theory of computation*. 1997.
- [3] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936.