THE HALTING PROBLEM: PROOFS AND DEMOS

David Wyde

September 29, 2020

ABSTRACT

This paper examines proofs that certain algorithms cannot exist. We review Turing's 1936 case for undecidability, modern analysis of the halting problem, and a separate argument that Turing gave in 1954. We provide simplified Python implementations of the programs in question and explore a gap in Turing's latter proof.

1 Introduction

Hilbert's decision problem asks whether logical statements can be proven or not. Turing studied whether a machine can tell if an algorithm produces an infinite amount of useful output. The modern halting problem checks if a computer program ever terminates[1].

This paper critiques three proofs that no general decision algorithm can exist: two arguments by Turing and one given in modern computer science textbooks.

2 Turing, 1936

2.1 Background

In 1936, Turing introduced a model of computation now called a Turing machine. In the same paper he proved that this form of computer cannot solve certain problems[2].

In Turing's original formulation, a Turing machine never stops running. It starts with a tape (a one-dimensional array of values, analogous to a modern computer's memory), a particular entry of the tape that it is examining, and a state that determines what the next step should be.

At each point in a computation, a machine can erase or overwrite the current symbol and possibly move left or right. It then enters a new state. For the Turing machines that we are discussing, these transitions are deterministic: there is always one possible action for the machine to take.

A Turing machine outputs 0's and 1's. Any other symbols that it writes to its tape are for bookkeeping. Turing called a machine that writes an infinite number of 0's or 1's "circle-free"; a machine that gets stuck in a non-printing state is "circular". Every Turing machine runs forever.

Turing's undecidable problem is to build a machine that, given the specification of another machine, determines if the supplied machine is circle-free: does the input machine provide an infinite number of valuable output bits, even if they are just 0's, or does it get stuck in a silent loop?

2.2 Turing's 1936 Proof

To prove that the above decision problem is unsolvable, Turing assumed that a decision algorithm exists and showed that this assumption leads to a contradiction. Since the presence of a decision algorithm causes an impossible result, Turing concluded that no decision algorithm can exist.

Turing called the hypothetical decision algorithm $\mathcal{D}$. His undecidable algorithm, $\mathcal{H}$, uses $\mathcal{D}$ as a step in its computation. Turing's plan was to show that, while $\mathcal{H}$ should be circle-free, it results in an endless sequence of $\mathcal{H}$ calling itself: circular behavior.

Each Turing machine has an integer identifier, called a description number, that encodes the machine's initial configuration and rules. $\mathcal{H}$ loops through all possible description numbers and analyzes each of the corresponding programs. Its goal is to build a sequence of characters called β': the first output bit from the first circle-free machine, the second output bit from the second circle-free machine, and so on.

$\mathcal{H}$ runs $\mathcal{D}$ on each Turing machine to see if the provided program is circle-free. If so, $\mathcal{H}$ gets the relevant output bit from that machine and appends it to β'. If β' has three entries when $\mathcal{H}$ finds a circle-free machine, the fourth bit of that machine's output becomes the fourth bit of β'.

The key step in Turing's proof is when $\mathcal{H}$ analyzes itself. $\mathcal{H}$ is a Turing machine, so $\mathcal{H}$ must eventually come upon its own description number. In order to get the next bit of β', $\mathcal{H}$ needs to check if $\mathcal{H}$ itself is circle-free and, if so, find the appropriate bit of its own output.

Turing argued that each of $\mathcal{H}$'s steps is deterministic and finite. $\mathcal{D}$ detects whether the current description number corresponds to a circle-free machine, then $\mathcal{H}$ calculates the output bits of any such machine until it gets the next bit of β'. Thus, $\mathcal{H}$ should be circle-free.

On the other hand, Turing noted that $\mathcal{H}$ must be circular. When $\mathcal{H}$ evaluates itself, the main $\mathcal{H}$ must compute the beginning of the output bits from calling $\mathcal{H}$. Now $\mathcal{H}$ enters an infinite loop of calling itself: it always computes the first part of β' but never outputs the bit that corresponds to $\mathcal{H}$'s description number.

Turing argued that $\mathcal{H}$ must be both circular and circle-free, which is a contradiction. He concluded that $\mathcal{D}$, the decision algorithm that $\mathcal{H}$ used, cannot exist.

2.3 Is $\mathcal{H}$ Circular?

$\mathcal{D}$, by construction, can detect all circular behavior. This should include the infinite recursion that occurs when $\mathcal{H}$ simulates $\mathcal{H}$. Can $\mathcal{D}$ recognize that $\mathcal{H}$ is not circle-free, skip it, and move on to the next description number?

To test the limits of $\mathcal{D}$ and $\mathcal{H}$, we can approximate them in a modern programming language. We used Python 3.6.9 to implement a simplified $\mathcal{D}$ that can detect some kinds of infinite recursion. In particular, it can detect the circular behavior in $\mathcal{H}$. Turing's proof assumed the existence of a perfect $\mathcal{D}$.

We will analyze the modern form of the halting problem: can an algorithm determine whether a provided program loops forever or halts? We do not claim to have solved the halting problem, but instead ask whether Turing's example is undecidable.

We provide the source code for $\mathcal{D}$, called `halts`, in Appendix A. Given an input function `func`, `halts(func)` should return `False` if `func` leads to an infinite loop and `True` otherwise.

2.4 A Python Implementation of Turing's 1936 Proof

The following function halts:

```
def does_halt():
    """ A program that halts. """
    return True
```

On the other hand, a call to the following function leads to infinite recursion:

```
def infinite_recursion():
    """ A non-halting program via infinite recursion. """
    infinite_recursion()
```

The function `infinite_recursion` calls itself over and over again. This is non-halting behavior, though modern software might crash with an error.

Our implementation of $\mathcal{D}$, `halts`, reports these results:

```
>>> halts(does_halt)
True

>>> halts(infinite_recursion)
False
```

Turing's $\mathcal{H}$ can be approximated as

```
def turing():
    """ Turing's undecidable task: get one output character from every program. """
    programs = [infinite_recursion, turing, does_halt]
    b_prime = []
    for prog in programs:
        if halts(prog):
            result = prog()
            r = len(b_prime)
            diagonal = str(result)[r:r + 1]
            b_prime.append(diagonal)
    return b_prime
```

The above function contains infinite recursion, so we expect it not to halt:

```
>>> halts(turing)
False
```

On the other hand, our $\mathcal{H}$ implementation works when we run it directly:

```
>>> turing()
['T']
```

β' is equal to the first character of `does_halt`'s `True` return value.

In this implementation, `halts(turing)` returns `False`, but `turing()` does halt. This may seem counterintuitive, but there is an explanation.

2.5 Analysis of the Demo

A key question is whether `halts(func)` should return `True` or `False` when non-halting behavior occurs while checking if `func` halts. It is open to interpretation how `halts(func)` should return when calls to `halts(func)` lead to non-halting behavior outside the body of `func`. This can happen when `func` enters into an infinite loop if and only if `halts(func)` returns `True`, as in Turing's proof.

If `halts` returns `True` in that case, there will be times when running `func()` results in an infinite loop but `halts(func)` returns `True`. This will be non-halting behavior that `halts` cannot detect. On the other hand, `halts(func)` could return `False` if it encounters an infinite loop that occurs outside of `func`. This approach results in cases when `func()` halts but `halts(func)` returns `False`.

An implementation of `halts` should pick one of these two interpretations of $\mathcal{D}$; a third path is for `halts` to throw an error in this situation. This paper prefers the second interpretation, that `halts(func)` returns `False` if `halts` causes infinite recursion. Thus, `halts(func)` asks whether non-halting behavior occurs in either `func` or `halts(func)`.

This is the behavior in the demo. In short, the outer call to `halts(turing)` returns `False` when calling `halts(turing)` at the top level, while the inner call to `halts(turing)` returns `False` when the top-level call is `turing()`.

When calling `halts(turing)` at the top level, `halts` runs `turing()` to check it for infinite recursion. This leads to a nested call to `halts(turing)` when the loop reaches `turing` in its sequence of all programs. This inner call to `halts(turing)` causes the original call to `halts(turing)` to recognize infinite recursion and return `False`.

In contrast, in the top-level call to `turing()`, the inner call to `halts(turing)` returns `False` during the loop through all programs. The rest of the loop body does not run on that iteration, so there are no further recursive calls to `turing`. The loop proceeds to the next program, `does_halt`.

This suggests that Turing's $\mathcal{H}$ can exist: $\mathcal{D}$ can detect that the inner call to $\mathcal{H}$ is circular, so the main $\mathcal{H}$ should not get stuck in infinite recursion.

3 Modern Proofs

Some contemporary sources give a different proof that the halting problem is undecidable. This section focuses on the version in a text by Dasgupta, Papadimitriou, and Vazirani[3]. Sipser's introductory book gives a proof via Turing machines that accept a language, rather than using infinite loops in the halting problem[4].

The steps are similar in both proofs:

1. Assume that a decision checker exists.
2. Create a function that does the opposite of the decision checker.
3. Pass the new function to itself.

The idea is that the new program will only halt if it does not halt, which is a contradiction. The authors conclude that no halt-checking program can exist.

Again, we can use Python to test this situation. The program in question is

```
def modern(f):
    """ A modern proof for the halting problem: do the opposite of `halts`. """
    if halts(f, f):
        modern(f)
```

If the input program f halts when passed itself as input, the contradictory program modern starts over. This example calls itself again with the same input; the textbook version uses a goto statement.

Using the same definition of halts as in the prior section, we see similar results to Turing's $\mathcal{H}$:

```
>>> halts(modern, modern)
False

>>> modern(modern)
>>>
```

A direct call to modern finishes, but halts reports a failure to terminate. As with Turing's proof, this is due to a non-halting branch that the program only reaches if a call to halts returns True. Our implementation of halts returns False if non-halting behavior occurs outside the body of func, so these are the expected results.

4 Limitations

The given implementation of halts has several limitations.

One caveat is that halts calls the function that it checks. This seems reasonable, but maybe there is a different interpretation of the halting problem that requires static analysis.

Our halts can only detect infinite recursion that involves a cycle of identical calls. It notices halting behavior in functions like

```
def cycle(n):
    """ A recursive program that cycles infinitely: 0 -> 1 -> 2 -> 0. """
    cycle((n + 1) % 3)
```

On the other hand, it cannot handle cases with external state that changes between function calls. It also misses non-cyclical infinite recursion like

```
def rising(n):
    """ Infinite recursion with a different `n` each time. """
    return rising(n + 1)
```

It seems possible that a true halts could model how n changes and recognize that the above code leads to infinite recursion. This issue is also relevant to procedural infinite loops:

```
n = 1
while n > 0:
    n += 1
```

Certain patterns are hard to predict, but that may be due to inadequacies in our current tools rather than inherent undecidability.

5 Turing, 1954

Turing gave a separate proof of undecidability in a 1954 article[5]. Instead of the halting problem, he explored a game involving binary sequences. Turing used *B* and *W* for black and white.

The goal of this game is to transform one sequence of characters into another. A sample task is to turn *WWB* into *WWW*. Each puzzle features rules that describe how to replace strings of letters. Two example rules allow a player to replace *WW* with *B* and *BB* with *WWW*. In this case, *WWB* becomes *BB* by the first rule. Then, *BB* becomes *WWW* by the second rule. This puzzle can be solved in two steps.

Some puzzles are easy to complete, some require a longer series of expansions and contractions, and others are impossible to solve. One unsolvable puzzle starts with *B*, has a goal of *W*, and has the rule that *B* can become *BB*.

As in the halting problem, it may not be obvious ahead of time whether a certain starting string and set of rules can ever turn into the given target. The naive approach, applying rules until the game reaches its desired state, gets stuck in infinite loops on unsolvable puzzles.

Turing used a restricted version of the game to prove that no algorithm can solve every puzzle. As in 1936, his idea was to show that no general decision process can exist.

The simplified game requires that at most one rule can be applied in each situation: the moves are unambiguous. This version of the game ends with success when a single *W* or *B* remains and there are no possible moves. Turing said that these puzzles "come out".

Turing labeled puzzles that end up with just a *W* as class I, and any other puzzles (puzzles that end with *B*, unsuccessful puzzles, or setups with invalid rules) as class II.

This proof started by assuming that a puzzle-solving machine can exist. Turing's goal was to demonstrate that this leads to a contradiction, which would prove that there is no general solver for this game.

Turing encoded this hypothetical solver within a puzzle called *K*. Given a second puzzle, with its setup encoded as a series of rules that *K* can process, *K* outputs *B* or *W* to indicate whether the provided puzzle is in class I or class II.

Turing explained the format of *K*'s output:

> The puzzle *K* might be made to announce its results in a variety of ways, and we may be permitted to suppose that the end result is *B* for rules *R* of class I, and *W* for rules of class II. The opposite choice would be equally possible, and would hold for a slightly different set of rules K', which however we do not choose to favour with our attention.

The key step in this undecidability proof happens when *K* analyzes itself. This seems analogous to Turing's 1936 proof, but the argument is different. Turing called *K*'s test of itself *P(K,K)*:

> The puzzle *P(K,K)* is bound to come out, but the properties of *K* tell us that we must get end result *B* if *K* is in class I and *W* if it is in class II, whereas the definitions of the classes tell us that the end results must be the other way round. The assumption that there was a systematic procedure for telling whether puzzles come out has thus been reduced to an absurdity.

This contradiction occurs because Turing chose *B* for class I and *W* for class II in the previous quote. The opposite assignment, *W* for class I and *B* for class II, would have worked.

Given two ways to format *K*'s output, Turing showed the straightforward case but skipped the other one. This is a hole in the proof.

6 Summary

For a function `func` and a hypothetical halt-checker `halts`, there are at least three cases:

#1. `func()` halts and `halts(func)` returns `True`.

#2. `func()` does not halt and `halts(func)` returns `False`.

#3. `func()` halts if and only if `halts(func)` returns `False`.

The specification for `halts` is ambiguous: should #3 programs return `True`, `False`, or an error? The halting problem requires `halts` to categorize three cases and output a Boolean. Turing's 1936 proof and modern textbooks say that #3 programs are contradictions.

In this paper we group #2 and #3 together. With this approach, the halting problem asks if `func()` terminates and `halts(func)` returns `True`. If so, `halts(func)` returns `True`; in all other cases, `halts(func)` returns `False`. It would also be possible to group #1 and #3 together, a choice that is a matter of interpretation.

Turing's 1954 paper takes an approach similar to treating #2 and #3 as the same result. That paper categorizes puzzles as either class I (puzzles that terminate with a single *W*) or class II (all other cases, including unsolvable puzzles and cases that end with a single *B*).

The use of `if-else` in Turing 1954 (class I vs. everything else) is different than `if-elif` in Turing 1936 and the modern proofs (halting vs. non-halting). A black-and-white puzzle is always either class I or class II; a do-opposite program (case #3) is in an awkward space between halting and non-halting.

7 Conclusion

Several common proofs of undecidability are flawed.

Turing's 1936 proof and its modern adaptations state that specific programs are impossible to implement. We provide simplified Python versions of both algorithms.

Turing's 1954 proof of an undecidable problem sets up a situation with two cases but only handles one of them. This is a gap in the argument.

We hope that future work can repair these proofs, come up with new arguments, or better understand which decision problems are unsolvable.

References

[1] C. Petzold, *The Annotated Turing*. Indianapolis: Wiley Publishing, Inc., 2008, pp. 47, 76, 179.

[2] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, 1937 (written 1936).

[3] S. Dasgupta, C. Papadimitriou, and U. Vazirani, *Algorithms*. New York: McGraw-Hill, 2008, pp. 263–264.

[4] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Boston: Cengage Learning, 2013, pp. 207–210.

[5] A. M. Turing. "Solvable and Unsolvable Problems," *Science News*, no. 31, pp. 7–23, 1954. Reprinted in B. J. Copeland, The Essential Turing. Oxford: Oxford University Press, 2004, pp. 582–595.

Appendix A Demo for the Halting Problem

```
class DoesNotHalt(Exception):
    """ An exception to indicate an infinite loop (non-halting behavior). """

# Wrap functions to run code before and after function calls.
wrapped = {}

# Track nested function calls in a stack.
previous = []

def halts(f, *args):
    """ Test whether a function halts on a particular input. """
    f_wrapped = wrap(f)
    try:
        f_wrapped(*args)
    except DoesNotHalt:
        # An infinite loop occurred.
        # Clean up the call stack, starting from the right.
        # This should also handle `while True` loops.
        halts_entry = ('halts', format_args(f, *args))
        for i, call_entry in enumerate(reversed(previous)):
            if call_entry == halts_entry:
                del previous[len(previous) - 1 - i:]
                break
        return False
    return True

def format_args(*values):
    """ Identify objects by their type and name (if present) or value. """
    return tuple((type(val), getattr(val, '__name__', val)) for val in values)

def wrap(f):
    """ Track calls to functions, to detect some types of infinite recursion. """
    name = f.__name__
    if name not in wrapped:
        def inner(*args):
            call = (name, format_args(*args))
            # Check for an identical call in the call stack.
            # This indicates infinite recursion (assuming no external state).
            if call in previous:
                raise DoesNotHalt

            # Call the original function and look for repeated calls.
            previous.append(call)
            result = f(*args)
            if call in previous:
                previous.remove(call)
            return result
        # Wrap the original function and update its namespace (for recursion).
        inner.__name__ = name
        wrapped[name] = f
        f.__globals__[name] = inner
        return inner
    else:
        return f

# Preemptively wrap the `halts` function.
halts = wrap(halts)
```