THE HALTING PROBLEM

Joshua Eckroth
Chautauqua Nov 10 2015

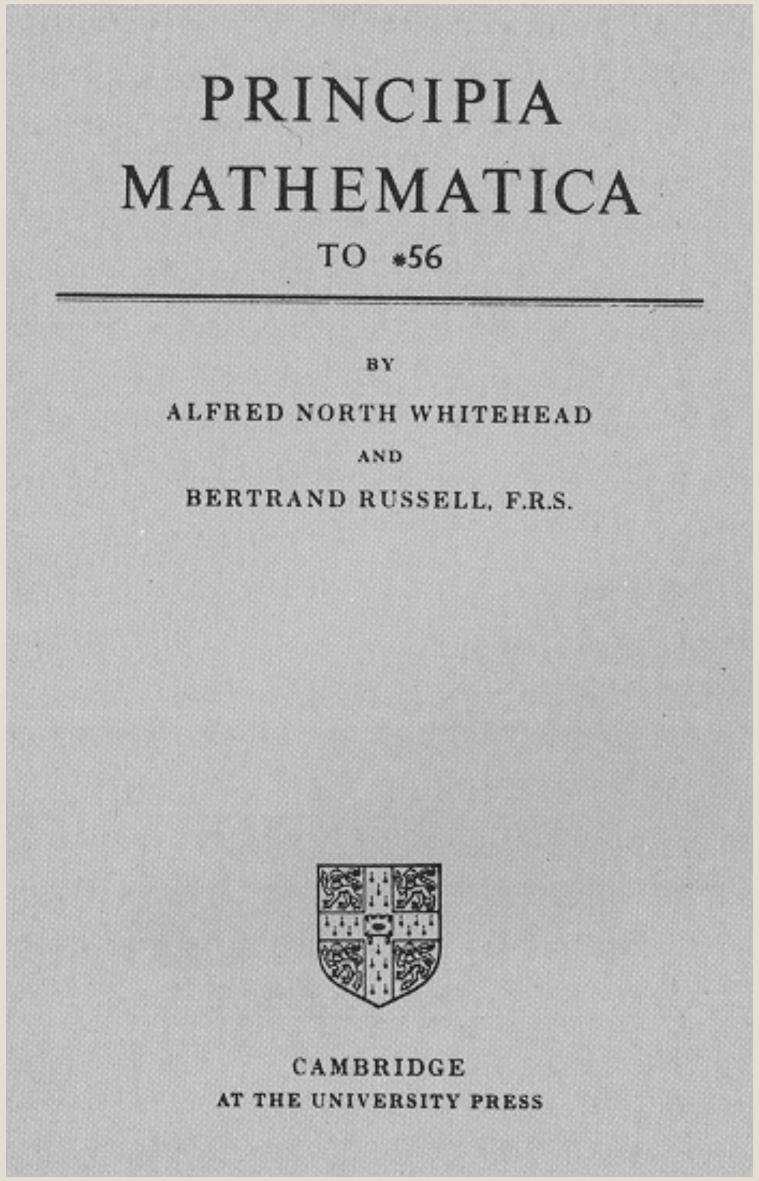
The year is 1928

- Sliced bread is invented.
- Calvin Coolidge is President.
- David Hilbert challenged mathematicians to solve the *Entscheidungsproblem*:

  Is there an algorithm that can determine whether or not any arbitrary statement of logic is provable or not given a set of logical premises?

Why did Hilbert ask?

- 1910-1913, *Principia Mathematica* is published by Whitehead and Russell.
- It "was an attempt to describe a set of axioms and inference rules in symbolic logic from which all mathematical truths could in principle be proven." (Wikipedia)


PRINCIPIA MATHEMATICA
TO ∗56

BY
ALFRED NORTH WHITEHEAD
AND
BERTRAND RUSSELL, F.R.S.

CAMBRIDGE
AT THE UNIVERSITY PRESS

The year is 1931

- Kurt Gödel proves two things:
  - **First Incompleteness Theorem**: for any consistent theory of mathematics, one can construct a mathematical statement that is true but not provable.
    - Crazy technique: "This statement cannot be proved in the theory." If the statement can be proved, it is false (which is inconsistent), and if it cannot be proved, it is true but the theory is **incomplete** (cannot prove all true things).
  - **Second Incompleteness Theorem**: If a theory contains a statement about its own consistency, then the theory is inconsistent. So, no theory can provide proof of its own consistency.

Turing’s proof of the *Entscheidungsproblem* (1936)

- Using a similar technique (self-reference), Turing proved Hilbert’s “decision problem” cannot be solved.
- **Turing invented Turing Machines to assist with his proof.**
  - Turing Machines are still the foundation of computer science; every computer you’ve ever met is a Turing Machine.
  - Alonzo Church proved the same result independently at the same time, using the Lambda Calculus instead of Turing Machines.

Turing reformulated the problem as *The Halting Problem*

Does this program ever quit?

A fancy analyzer of programs

Suppose we have a function, **FancyAnalyze(P, s)** that can do the following:

- Analyze how **P** (represented as source code) behaves on input string **s** (without actually simulating **P**). **P** only says "Yes" or "No."

- If **P(s)** would have halted with "Yes," then **FancyAnalyze** returns "Yes."
- If **P(s)** would have halted with "No," then **FancyAnalyze** returns "No."
- If **FancyAnalyze** determines that **P(s)** would get stuck in an infinite loop, then **FancyAnalyze** returns "Loops."

Could a fancy programmer write **FancyAnalyze**? (Hint: No.)

Using FancyAnalyze to create a self-decider

P-yes is a program that identifies those programs that say "Yes" to their own source code:

```
Program P-yes(s):
    Answer ← FancyAnalyze(s, s)
    If Answer = "Yes" then
        return "Yes"
    Else If Answer = "No" or  Answer = "Loops" then
        return "No"
```

Applying P-yes to itself

```
P-yes(P-yes):
    Answer ← FancyAnalyze(P-yes, P-yes)
            FancyAnalyze code:
                    if P-yes(P-yes) halts with "Yes", return "Yes"
                    else if P-yes(P-yes) halts with "No", return "No"
                    else if P-yes(P-yes) loops forever, return "Loops"
    If Answer = "Yes" then
            return "Yes"
    Else If Answer = "No" or  Answer = "Loops" then
            return "No"
```

A quick self-check

What does **P-yes** do on its own source code?

- If **FancyAnalyze(P-yes, P-yes)** returns “Yes,” then **P-yes(P-yes)** returns “Yes.”
- If **FancyAnalyze(P-yes, P-yes)** returns “No” (“Loop” is impossible), then **P-yes(P-yes)** returns “No.”
    - “Loop” is impossible output of **FancyAnalyze(P-yes, P-yes)** because the **P-yes** program always just says “Yes” or “No” without any loops.

A quick self-check (in other words)

What does **P-yes** do on its own source code?

- If **P-yes(P-yes)** returns "Yes" (as determined by **FancyAnalyze**), then **P-yes(P-yes)** returns "Yes." (duh)
- If **P-yes(P-yes)** returns "No" (as determined by **FancyAnalyze**), then **P-yes(P-yes)** returns "No." (duh)

A backwards self-decider (why not?)

```
Program P-no(s):
    Answer ← FancyAnalyze(s, s)
    If Answer = “Yes” then
            return “No” 😈
    Else If Answer = “No” or Answer = “Loops” then
            return “Yes” 😈
```

If you believed we could write **P-yes**, you have to allow **P-no**.

The sucker punch

What does **P-no** do on its own source code?

- If **FancyAnalyze(P-no, P-no)** returns “Yes,” then **P-no(P-no)** returns “No.”
- If **FancyAnalyze(P-no, P-no)** returns “No,” then **P-no(P-no)** returns “Yes.”

The sucker punch (in other words)

What does **P-no** do on its own source code?

- If **P-no(P-no)** returns “Yes” (as determined by **FancyAnalyze**), then **P-no(P-no)** returns “No” (!!)
- If **P-no(P-no)** returns “No” (as determined by **FancyAnalyze**), then **P-no(P-no)** returns “Yes” (!!)

So, uh…

- **FancyAnalyze cannot exist.**

- **P-yes** wasn’t the problem. It was a trivial program.
- **P-no** wasn’t the problem. It was a trivial program.

So, uh…

Thus,

- **You cannot write a algorithm that is capable of determining if <u>any arbitrary</u> program behaves a certain way.**

Because, if you tried,

- We could just write a variant of that algorithm that behaved contradictory to the expected behavior in the special case when it is applied to itself.
- In other words, we could make the algorithm appear to contradict itself in at least one special scenario (when it is applied to its own code).
  - No matter what the algorithm is trying to determine about arbitrary programs!

Implications

No program can do any of the following:

- Determine if some arbitrary program prints a particular message (e.g., program verification).
- Determine if some particular line of code in some arbitrary program ever executes.
- Determine if some arbitrary program has any infinite loops.

Implications

No program can do any of the following:

- Given two programs, determine if they are equivalent in behavior.
- Determine if any arbitrary program ever throws a NullPointerException.
- Rice's Theorem: For any non-trivial property of a program, determine if any arbitrary program has that property. Non-trivial means some programs have the property and some don't.

Implications

No program can do any of the following:

- Given two programs, determine if they are equivalent in behavior.
- Determine if any arbitrary program ever throws a NullPointerException.
- Rice's Theorem: For any non-trivial property of a program, determine if any arbitrary program has that property. Non-trivial means some programs have the property and some don't.

Derek Jacobi as Alan Turing

https://www.youtube.com/watch?v=gV67Sj2jkVg