Incomputability

[A very short introduction.]

Impossibility

1905 Albert Einstein: *It is impossible to go faster than the speed of light.*

1925 Werner Heisenberg: *It is impossible to know both position and velocity of a particle.*

1931 Kurt Gödel: *There are 'theorems' of mathematics that are true (have no counterexamples) but not provable.*

1935/36 Alan Turing & Alonzo Church: *There are algorithmic problems that have no algorithmic solution.*

Hilbert's Entscheidungsproblem

In 1900, David Hilbert asked for an algorithm to solve this problem

Problem: Entscheidungsproblem
Input: A mathematical statement f
Output: Whether or not there is a proof of f.

Given an algorithm for the Entscheidungsproblem, we could input the Riemann Hypothesis, $\mathbf{P} = \mathbf{NP}$, or Goldbach's conjecture and, eventually, get an answer as to whether there is a proof. (Furthermore we would get a proof or a disproof.)

In different ways, Church and Turing showed that no such algorithm could exist.

- As a side effect of their efforts, they both gave formal

definitions (later proved equivalent) of 'computable'.

- Turing, almost as an aside, introduced the concept of the 'general purpose computer'.

The Halting Problem

As an example, we will prove that there is no algorithm for the following problem

> Problem: The halting problem
> Input: A description p of an algorithm that takes a string as an input; and a string s.
> Output: true if the p halts when fed s as an input; false if p does not halt when fed s as an input.

The halting problem has an important direct application in program verification: Suppose we have an algorithm for the following problem

> Problem: The verification problem.
> Input: A procedure, a precondition, and a postcondition.
> Output: True if for every input that satisfies the precondition, the procedure halts in a state that satisfies the postcondition. Otherwise false.

If there were an algorithm for this problem, there would be an algorithm for the halting problem. Therefore there is no algorithm to solve the verification problem.

Furthermore, many other problems (including the entscheidungsproblem) can be shown to be no more computable than the halting problem. (If we had an algorithm for the entscheidungsproblem, we could construct an algorithm for the halting problem.)

To be definite, we will take p to be a program (hence a string) defining one or more methods written in an 'idealized' Java-like language. The first method defined will be considered the 'main' method.

- I say 'idealized' because we will assume that the program is run on a machine that does not run out of memory.

Define function H so that $H(p, s)$ is true iff the algorithm described by string p halts when it is executed with string s as input: $H(p, s) = (p \text{ fed } s \text{ halts})$

$$H(p, s) = (p \text{ fed } s \text{ halts})$$

Example. If $p0$ is

"void a(String s) { while(true) { } }"

then $H(p0, s) = \text{false}$ (for all s)

Example. if $p1$ is

"void a(String s) { while(s.length==0) { } }"

then the value of $H(p1, s)$ depends on the length of s.

(If p is not a program —e.g. has a syntax error— we will assume $H(p, s) = \text{false}$.)[1]

[1] A full proof would mathematically define "p fed s halts". Instead, I am relying on your knowledge of programming languages. A proper definition might begin by defining a 'computation' as a sequence of states and end by defining "p fed s halts" to mean: every computation of p that starts with s has finite length.

We say that an algorithm **computes** a function f iff it

- never crashes (e.g. no array bounds violations),
- halts for every input, and
- outputs $f(x)$, for every input x in the domain of f.

We say that a function is **computable** if there is an algorithm that computes it.

We will take it as a fact that any algorithm that computes a function from strings to booleans can be encoded in our Java-like language.

Recap: For all strings p and s, $H(p, s)$ is true iff the algorithm described by string p halts when it is executed with string s as input

$$H(p, s) = (p \text{ fed } s \text{ halts}) \tag{1}$$

Theorem. Function H is not computable.

Proof (by contradiction)

1. Assume H is computable.
2. That means there is an always-terminating algorithm that computes function H.
3. We can express this algorithm as a program in our Java-like language. Let h be such a program
$$h = \text{“boolean h(String p, String s)}\{\cdots\}\text{”}$$
4. Suppose the procedure name “d” is not used in h.[2] Let d be the string
“void d(String s){ if(h(s,s)) { while(true) { } } } ” ^ h
5. From the steps 2,3, and 4, for all strings s,
$$(d \text{ fed } s \text{ halts}) = \neg H(s, s)$$
6. Now consider feeding string d to itself. (Cannibalism!)
7. From step 5 $(d \text{ fed } d \text{ halts}) = \neg H(d, d)$
8. By definition (1) $H(d, d) = (d \text{ fed } d \text{ halts})$
9. From steps 7 and 8, $H(d, d) = \neg H(d, d)$.
10. This is a contradiction. Our assumption at step 1 must be false.
11. Thus H is not computable.

[2] If there is a procedure named “d” in program h, we can pick another name.

Remarks

- Crucial to this proof is the idea that a program is also data.
- This proof hinges on the assumption that our Java-like language is able to compute any computable function.
  - As evidence for this we can show that this language can express the same algorithms as many other languages and formalisms, including Turing Machines and Church's Lambda calculus.
  - In a sense, all (idealized) programming languages are equivalent to each other and to both Turing's model of computation (Turing machines) and Church's model of computation (Lambda calculus).
  - The assumption that all computable functions can be expressed in these models of computation is called the Church-Turing thesis.
- An alternative point of view questions whether the H function was well-defined to begin with. Thus at step 10 the correct conclusion is that either H is not computable or H was not well-defined. This point of view is elaborated in E.C.R. Hehner 'Problems with the Halting Problem'.
- As mentioned, Turing used Turing machines[3] rather than a "Java-like language". First off, high-level programming languages did not exist at the time. More importantly, to make the proof complete he

[3] He didn't call them "Turing machines". He called them "A-machines"; "A" is for "automatic".

needed to define exactly what it means to ‘feed a string to a program’; the semantics of Turing machines can be described in a few lines of mathematics; high-level languages are much more complex.

- It should not be concluded that, since the halting problem is insolvable, we therefore can not write programs to verify other programs. What should be concluded is that such a program will sometimes report that it can neither verify nor prove incorrect its input.

Historical Notes

It may seem surprising that the first application of the formal definition of 'algorithm' was to show that something could not be done by an algorithm. However, think about it this way: As long as people were showing that more and more things can be done by algorithms, the concept was allowed to expand; there was no need to circumscribe it. Once they needed to show that something could not be done by an algorithm, it was necessary to really pin-down the concept, so that it could be argued that it would not grow any more.

Church and Turing made these discoveries independently in 1935/36. Later Turing was Church's PhD student. While doing his PhD, he also experimented with building digital circuits with relays. He had a major role in WWII designing both code breaking algorithms and electromechanical code-breaking machines. ("But, it was probably a good thing that the security people didn't know [that Turing was homosexual], because he might then have been fired and we might have lost the war" — I. J. Good.) After WWII, Turing designed one of the first general-purpose electronic computers. Thus Turing both invented the concept of the general purpose computer (as a mathematical tool) and made important contributions to its eventual electronic realization.

Church was an important logician and supervisor of many other important logicians. His lambda-calculus became

the foundation for today's functional programming languages.

Recommended Reading/Watching

David Harel

- *Algorithmics: The spirit of computing*
  - A layperson's introduction to computing theory.

Martin Davis

- *The Engines of Logic*
  - A highly readable history, from Leibniz to Turing, of the interplay between logic and computation. Although a history, the math is very well explained.

BBC Horizon

- *The Strange Life and Death of Dr. Turing*
  - http://www.youtube.com/watch?v=gyusnGbBSHE
  - http://www.youtube.com/watch?v=5LHFzNMgWzw