Week 14: The Limitative Results II: The Halting Problem

Introduction to the Philosophy of Mathematics
Dr. Neil Barton
neil.barton@uni-konstanz.de

5. February 2020

Recap

Last time we examined Gödel's two limitative results, and some of their philosophical implications.

In particular we saw that:

(1.) There is a sentence G that, assuming $\mathbf{PRA}$ is ω-consistent[1] such that $\mathbf{PRA} \nvdash G$ and $\mathbf{PRA} \nvdash \neg G$.[2]

(2.) There is a sentence $Con(\mathbf{PRA}) =_{df} \neg\exists x Prf(x, \ulcorner 0 = 1 \urcorner)$ such that if $\mathbf{PRA}$ is ω-consistent then $\mathbf{PRA} \nvdash Con(\mathbf{PRA})$.

This essentially put to bed Hilbert's Programme in its unrestricted form.

However, these days, we look at a modified version of Hilbert's Programme on which we look at *relative consistency*; if I *assume* some theory $\mathbf{T}$ is consistent, what other theories $\mathbf{T}'$ can I prove consistent using $\mathbf{T}$ (over a weak theory like $\mathbf{PRA}$)?

This time we'll examine the idea of a *computation*, and see some *positive* results, but also a key *limitative* result.

This isn't so connected to examining the idea of *infinity* per se, but its important for a few reasons:

1. The notion of infinity (and finite) is important for specifying what a *computation* is, and this latter idea is in turn of serious importance.
2. There are close links between properties of formal theories and the idea of computation.
3. As we'll see, the particular limitative result we'll consider (the Halting Problem) has real practical implications, and started from the philosophical problem of how to analyse computation.

Several scholars contributed to the modern theory of computing. Early ideas appear in Leibniz and Babbage,

[1] A theory $\mathbf{T}$ is ω-inconsistent if there is a formula $\phi(x)$ such that $\mathbf{PRA} \vdash \phi(n)$ for every standard natural number n, but $\mathbf{PRA} \vdash \exists x \neg\phi(x)$. It is ω-consistent iff it is not ω-inconsistent.

[2] Remember $\mathbf{PRA} \vdash G \leftrightarrow \neg\exists x Prf(x, \ulcorner G \urcorner)$.

and the possibilities of the power of computing start to appear in the work of Ada Lovelace.

We will focus the work of Alan Turing (1912–1915), who was both brilliant and suffered terribly at the hands of the British government. He made significant contributions to cryptography and computing, and was then convicted for his homosexuality (which was illegal at the time), underwent chemical castration, and (probably[3]) committed suicide at the age of 41.

1 Turing Computability

Over and over again we've seen the notion of *computability* occur.

e.g. The BHK-interpretation of the intuitionistic connectives and quantifiers, Hilbert's desire to have primitive recursive ways of checking problems.

But what is a *computer* and what is a *computation*?

The notion of a *computer* before the invention of *mechanical computers* largely involved *people*.

> The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. (Turing, 'Computing machinery and intelligence')

So the rough idea of a computation was the following:

(i) A *mechanical procedure* for writing new lines from old.

(ii) This should be *checkable*, we can examine whether a particular production of some lines conforms to the rules of production.

But this doesn't seem to be any *advance*, I've defined the idea of *computation* in terms of *mechanical procedure*, and this is also a philosophically problematic notion.

What Turing did was to provide (with help from others) a *formal model* for the notion of *mechanical procedure* and *computation*.

Definition. A *Turing machine* consists of:

(i) An infinite *tape* consisting of infinitely many *cells*.

(ii) Cells can contain either a 1, or are blank (we'll represent this by 0).

(iii) A machine that contains:

  (a) A list of *instructions* (also known as *states*) and *rules* for moving between the states.

  (b) A *head* that can first read whatever is in a cell, and then either (a) erase whatever is in a cell, (b) print a 1, (c) move left, (d) move right, or (e) halt (depending what the rules say).

[3] This is contested. Other theories include accidental cyanide poisoning and assassination by the secret service.

Turing used this to provide a formal model for what a *computation* was.

We say that a function f is *Turing-computable* iff there is a Turing-machine that computes f.

Example. Let Bin^1 be the set of all binary strings consisting solely of 1s. Then the function $f : Bin^1 \times Bin^1 \rightarrow Bin^1$ that takes an n-length sequence of 1s and an m-length sequence of 1s and returns the $(m + n)$-length sequence of 1s is Turing-computable.

State 1
If 1 move right and go to state 1.
If 0 print 1 and go to state 2.

State 2
If 1 move right and go to state 2.
If 0 move left and go to state 3.

State 3
If 1 erase and go to state 3.
If 0 move left and go to state 4.

State 4
If 1 go left and go to state 4.
If 0 move right and go to state 5.

State 5
If 0 halt.
If 1 halt.

Turing computability seems incredibly weak, but in fact is a very strong notion of computability.

Turing showed that a huge variety of functions are Turing computable (every Primitive Recursive function, for a start).

2 The Church-Turing Thesis

In fact, stronger than this, it's not just that lots of *functions* turn out to be Turing-computable, but it can be shown that almost any way you might come up with different notion of computability turns out to be equivalent to Turing-computability.

What you do to show this is, given another notion of computation, show how the moves of that kind of computation can be simulated using a Turing-machine.

Add more tapes? Equivalent to Turing computability.

Expand the alphabet? Also equivalent to Turing computability.

Work in a grid? Equivalent.

Rough Definition. An *abacus machine* is a machine that has *registers* that it can *scan* and contain a number of units, and can either *add* a unit to a register, *subtract* a unit from a register, *sum* two registers, or halt (contingent upon whether or not there is a unit in a register).

Pre-theoretically, abacus-computability looks way stronger than Turing computability.

Fact. Abacus computability is equivalent to Turing computability.

The key point here is that Turing machines can run as long as you like.

That's an awful lot of time in which to simulate other machines.

The blowup (amount of extra time required) to simulate a machine using a Turing machine can be enormous here (e.g. exponential).

Nonetheless, every kind of computability that we have come up with has turned out to be equivalent to Turing computability.

This has lead to the following informal thesis:

Church-Turing Thesis. Any effective computation can be computed by a Turing machine.

3 Universal Turing Machines

Many mechanical procedures require many different kinds of machines.

e.g. In my kitchen, I can use an *oven* to heat things up (to some temperature), a *garlic press* to mash up garlic, a *potato masher* to make mashed potatoes, a *knife* to cut things, a *juice press* to make juice.

I need various kinds of *machine* to carry out various kinds of *mechanical procedure*.

Modern day computers aren't like this though, we can instruct them to carry out lots of *different* functions.

Very abstractly and roughly, here's how that works.

What does a computation by a Turing machine consist of? Well, its a *finite* list of instructions (the machine), combined with a *finite* list of how to transfer from one state to another, combined with a particular *finite* input.

So exactly, as we did with Gödel-numbering, we can *code* a Turing machine by a particular number, and the input by some number.

This allows us to treat pairs (or n-tuples, depending on the machine) of Turing machines and inputs as inputs into other Turing machines.

There is then the possibility of having a *single* Turing machine tell us how *any* Turing machine behaves on a certain input.

Rough Definition.[4] A *Universal Turing Machine* is a Turing machine U that can compute the output of *any* other Turing machine M on the input x (by taking $\ulcorner M, x \urcorner$ as its input).

Theorem. (Turing) There is a Universal Turing Machine U.

This (very abstractly) is why you can have a smartphone rather than a million different devices for different purposes. You have a *Universal Turing Machine* (well, something roughly equivalent to one for small inputs) and people can write in a *programming language* that corresponds (with the help of a compiler) to what's going on in the machine-code, and you can instruct the smartphone to execute some particular code (and put

[4] See [Boolos et al., 2007] for the details.

bunny-ears on a selfie or whatever).

However, notice that we're running machines on the codes of other machines.

You're diagonalisation spidey-senses may be tingling...

4 The Halting Problem

If we have a machine, something we'd like to know is if computations work given particular inputs.

What do we need for a computation to work?

We need the machine to act on the input(s), write the result of applying the function to the tape, and then *halt*.

If the machine hasn't yet halted, we don't yet know if it will halt.

Perhaps it just spins off forever and never finishes its computation...

...but maybe it is just a few steps away from finishing the computation and halting.

What we would like is a general way of telling whether a machine M halts given input x.

Since we can feed the codes of machines and inputs to other machines, what we'd like is a machine H that, when fed $\ulcorner M \urcorner$ and $\ulcorner x \urcorner$ outputs 1 if M halts on x, and 0 if M does not halt on x.

The Halting Problem. Is there such an H?

Answer. There *cannot* be such an H.

Proof Sketch.[5] Suppose there was such an H, such that $H(M, x)$ outputs 1 if M halts on x and 0 otherwise. Using H, we then construct the following program:

> **The Loop Snooper Scooper.**
> If $H(x, x) = 1$ then Loop Forever[a]
> Else Halt.
>
> [a] It's easy to define a subroutine that loops forever.

But now we run The Loop Snooper Scooper on *its own code*. Does this computation halt? If it halts, H returns 1, and then it loops forever (by definition of the Loop Snooper Scooper). Contradiction! If it loops forever, then H returns 0, but then it halts. Contradiction! Hence there is no such H.

5 Some Implications

Hilbert's Entscheidungsproblem. Is there an effective procedure or algorithm to decide for every statement in first-order logic whether or not it is derivable in that logic?

Answer. No. *Rough* reason (among many). You can show that Turing machines correspond to first-order formulas, and a solution to the Entscheidungsproblem would imply a solution to the Halting Problem.

Relationship between Turing Machines and Consistency Sentences.

[5] This is somewhat fiddly to define in detail. You need to really get into how to build the machine.

You can think of a consistency sentence as saying that some Turing machine cannot exist (specifically one that churns out a code of $0 = 1$ from the codes of some axioms). So if certain consistency sentences are *true*, then we get *empirical* predictions about how computers will behave.

Moreover, if some axioms about some infinite sets are *true* then (unless we're hardcore dialethists) they're *consistent*, and so the truth of claims about infinite sets has direct empirical consequences.

Applications to Computers. There's a host of applications arising from the philosophical project of analysing computation formally and then showing the Halting Problem cannot be computed.

A **debugger** is a program that runs on the code of other programs to check them for errors. The Halting Problem implies that there is no *perfect debugger* (i.e. a debugger that works on any program).

If there were such a program, then you could compute the Halting Problem, since failure to halt is just one such kind of error.

There's a lot of other applications here. For example Joey Eremondi writes[6]:

> You want a compiler that finds the fastest possible machine code for a given program? Actually the halting problem.

[6]See his answer to 'Why is the Halting Problem so Important?' on the CS-Stackexchange.

> You have JavaScript, with some variables at a high security levels, and some at a low security level. You want to make sure that an attacker can't get at the high security information. This is also just the halting problem.
>
> You have a parser for your programming language. You change it, but you want to make sure it still parses all the programs it used to. Actually the halting problem.
>
> You have an anti-virus program, and you want to see if it ever executes a malicious instruction. Actually just the halting problem.

6 Questions and Discussion

Great questions again this week! I hope we covered some already.

Question. How should we interpret the Church-Turing thesis? Do you think it's true?

Question. What is the relationship between Turing machines and the potential infinite?

Question. Does the statement of first-order theory in the Entscheidungsproblem (as I've formulated it) correspond to 'mathematics'?

Question/Optional Exercise. (Difficult.) Suppose Emmy is a psychic genius who can easily see (by intuition or whatever) whether a machine M (as described above) will halt on input x. Let's allow a Turing machine

to call on Emmy at any point in a computation (so there's an option for states where given that the current tape contents codes a Turing machine M and input x, the Turing machine can send the input off to $\mathcal{O}$ and either receive a 1 if M halts on x, and 0 otherwise, suppose this gets printed on a separate tape.) Does this solve the Halting Problem?

Hint 1. The question is a trick. It's very similar to asking "Does adding $Con(\mathbf{PRA})$ to $\mathbf{PRA}$ solve the 'consistency problem'?"

Hint 2. There is an expanded notion of computation, where we allow a machine at any point to ask Emmy (or $\mathcal{O}$) the halting question at any time during a computation. So this expanded notion of computation can solve the 'Halting Problem' by just asking Emmy for standard machines. But is the problem we started with the only halting problem we can define?

References

[Boolos et al., 2007] Boolos, G., Burgess, J., P., R., and Jeffrey, C. (2007). *Computability and Logic*. Cambridge University Press.