Turing machines within Turing machines!

<u>**Descriptions of Turing Machines**</u>

It is possible to write down any Turing machine as a sequence of zeros and ones. This sequence describes everything about the Turing machine that we mentioned earlier – Q, A, and δ.

We call this the **description** of the Turing machine.

We abbreviate "the description of Turing machine M" as just "<M>".

<u>**Universal Turing Machines:**</u>

Alan Turing proved the existence of a "**Universal Turing Machine**" (let's call it U). The Universal Turing Machine *simulates other Turing machines.*

U takes as its input the description of *any* Turing machine, <M>, followed by *any* input x. We call U the "Universal Turing Machine" because it can then simulate what M would do when M runs on the input x. U(<M>, x) produces exactly the same output as M(x).

U(<M>, x) = M(x)

The Halting Problem: *when Turingception goes wrong*

Remember how Turing machines can sometimes **loop**. Wouldn't it be cool if we had a program which can look at our programs and see if they have infinite loops? This is called the Halting Problem. *HALT* is the language of all Turing machines + inputs that halt.

Suppose we have a program which can do this. Let's call it *P*.

$$P(\prec M \succ, x) = \begin{cases} accept & \text{if } M(x) \text{ halts} \\ reject & \text{if } M(x) \text{ loops} \end{cases}$$

It turns out that *P* cannot exist. The reason for this is again devious. We set up a program *Q* which contains a description of *P*, and uses that to do the exact opposite of whatever *P* predicts *Q* does:

$$Q(x) = \begin{cases} loop & \text{if } P(\prec Q \succ, x) \text{ accepts} \\ halt & \text{if } P(\prec Q \succ, x) \text{ rejects} \end{cases}$$

Q is something that *P* analyzes incorrectly, so *P* cannot exist.

There is no Turing machine that can decide *HALT*! The Halting Problem is **undecidable.**

Scooping the Loop-Snooper

A proof that the Halting Problem is undecidable

Geoffrey K. Pullum

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called *P*
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and *P* gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then *P* prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then *P* reports 'Bad!' — which means you're in the soup.

Well, the truth is that *P* cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use — and it's simple to do.
I'll define a procedure, which I will call *Q*,
that will use *P*'s predictions of halting success
to stir up a terrible logical mess.

For a specified program, say *A*, one supplies,
the first step of this program called *Q* I devise
is to find out from *P* what's the right thing to say
of the looping behavior of *A* run on *A*.

If *P*'s answer is 'Bad!', *Q* will suddenly stop.
But otherwise, *Q* will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called *Q* wouldn't stay on the shelf;
I would ask it to forecast its run on *itself*.
When it reads its own source code, just what will it do?
What's the looping behavior of *Q* run on *Q*?

If *P* warns of infinite loops, *Q* will quit;
yet *P* is supposed to speak truly of it!
And if *Q*'s going to quit, then *P* should say 'Good.'
Which makes *Q* start to loop! (*P* denied that it would.)

No matter how *P* might perform, *Q* will scoop it:
Q uses *P*'s output to make *P* look stupid.
Whatever *P* says, it cannot predict *Q*:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be —
and simply by using your putative *P*.
When you posited *P* you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A *reductio*: There cannot possibly be
a procedure that acts like the mythical *P*.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!