# Proof of $P \neq NP$ for a special case of computer programs with unprotected source code

## 1. Discussion of the first incompleteness theorem.

Before starting, let's discuss Godel's first incompleteness theorem. There, logical statements in a formal langage are assigned natural numbers, and statements with free variables are modified by substitution, in which a free variable is replaced by the number which may represent another statement with a free variable. Such a convention is well-accepted in other branches of Maths, where we might replace a variable in a polynomial expression with a whole polynomial, thereby describing a composite of two functions.

If we remove the formalism of numbering, Godel's statement of the Richard paradox is

> The statment 'The statment $x$ would be unprovable if the free variable in that statment were replaced by the entire unsubstituted statement' would be unprovable if the free variable in that statement were replaced by the entire unsubstituted statement.

If we label this statement $L$ and perform just a single substitution on the inner statement it becomes

> The statement "The statment 'The statement $x$ would be unprovable if the free variable in that statement were replaced by the entire unsubstituted statement' would become unprovable if the free variable in that statement were replaced by the entire unsubstituted statement" would become unprovable if the free variable in that statement were replaced by the entire unsubstituted statement.

This is the statment

> The statement $L$ would become unprovable if the free variable in that statement were replaced by the entire unsubstituted statement.

The trick is that the statement above is in fact also exactly the same as the result of replacing the free variable in $L$ by the unsubstituted statement, so the statement asserts its own unprovability.

The unprovability is not surprising if we consider things from a standpoint of something like lambda calculus, as the substitutions which were done did not really clear all the free variables. Let's start with a $\lambda$ calculus term with a free variable, so $\lambda y.f(y)$ where $f(y)$ signifies a term in which $y$ occurs only as a free variable if at all. If we call this 'statement $x$,' then the assertion that statement $x$ would be unprovable if the free variable were replaced by the entire unsubstituted statment, is the assertion $n(f(f(y)))$, and more rigorously since $y$ is a free variable this is $\lambda y.n(f(f(y)))$. The assertion that this is unprovable once the whole expression replaces the free variable is $\lambda y.n(f(f(n(f(f(y))))))$ and the point is, it still has a free variable, even while we interpret it as asserting its own unprovability.

## 2. A party game

Let's base a party game on Godel's statement, so we imagine a party where attendees include Tom, Jerry, and Butch (the dog). However we're imagining them being English, so they would enjoy a party game more than just chasing each other.

We will call 'that silly statement' about Jerry, the endless statement

> Jerry can't prove in less than 100 thoughts that Jerry can't prove in less than 100 thoughts that Jerry can't prove in less than 100 thoughts....

And there is a similar silly statement about each other character, such as 'that silly statement' about Tom or Butch.

The game consists of a guest challenging another to prove the silly statement. So that Tom might challenge Butch,

> "Can you prove that silly statement about Jerry?"

To win, all that Butch has to do is answer, dismissively and contemptuously as he can ,

> 'Obviously.'

This suffices, because if it were false, it would be the assertion that it is not only provable, but provable in less than 100 of Jerry's thoughts.

The game would only become fun if Tom asked Jerry if *he* can prove it.

If Jerry answered 'obviously,' the others could shout, "Then it's false!"

Here is where Jerry has a fillibuster strategy. He might count $1, 2, 3, ..., 100$ and then shout 'Obviously.'

The others would have the retort, it is not so obvious now. And it would be to Jerry to try to explain, he dutifully waited and counted out 100 seconds before answering, so the earlier objection no longer holds.

But the others would counter, ‘That could at most prove that you *didn’t* prove it in less than 100 thoughts, you have still failed to show that you *couldn’t* prove it in less than 100 thoughts.

Jerry’s answer might be, to try to pretend that he is a simple-minded creature with no free will, and if the game were ever played again, everyone could be assured that his answer would still be the same.

Then there is the objection, this only shows that you couldn’t prove it within 100 *seconds,* not 100 *thoughts.*

And then as is typical in such cartoons, Jerry would need to produce an eeg machine or a mind reading machine, to show the sequence of mental images which he was imagining during the time. And the counting is then not necessary, and his answer could have been the same as the others, just ‘Obviously,’ but to win the game he would have to prove that the sequence of mental images has occurred, and would occur invariantly.

## 3. Names and protection of source code

It would be completely easy to formalize the incompleteness theorem and instantiate the character of Jerry in a variable Turing machine, to give a proof of $P \neq NP$ except for one missing detail: the character's name.

We can pose the problem, within a formal language, of algorithmically finding proofs of provable statements. Since proofs can be verified in polynomial time, the $P = NP$ question for degree $m$, for each $m = 1, 2, 3, \ldots$ is equivalent to the existence of a Turing machine which, when presented with a statement of length $a$ which has a proof of length $b$, of finding one within $(a+b)^m$ steps.

We can formulate 'that silly statement,' replacing the number 100 with a variable $n$, encoded alphabetically so that the length of 'that silly statement' grows logarithmically with $n$, and replacing the name 'Jerry' with the full source code of a Turing machine.

So we are describing a logical proposition $S(s,n)$ where $s$ is the source code of a Turing machine and $n$ a natural number, which asserts that $s$ does not find a proof of $S(s,n)$ within $n$ steps.

Just as in Godel's incompleteness theorem, there is no difficulty encoding such a statement as a finite sequence of characters.

We might wish next to set up the langauge so that a short sequence of characters (not unlike the word 'obviously' in the party game) would also provide a proof. This cannot be possible if the Turing machine which will find the proof happens to have source code identical to the input string $s$, however, for the same reason that in the party game Jerry cannot just answer 'obviously.' However, it is a consistent solution to the problem of the Turing machine with source code $s$ finding a proof of $S(s,n)$ if it for example merely runs the code of $s$ as a subroutine, waits $n$ steps, and then observes that there was not yet any proof. That observation constitutes a proof. In fact, since there couldn't have been such a proof, logically, it suffices merely for the Turing machine to 'fillibuster' for $n$ steps before announcing 'Obviously.'

Designers of such a Turing machine would have no difficulty writing such a source code $s$, yet the difficulty is that the Turing machine does not know, as it were, its own name. It must act even when receiving as input the task of proving $S(t, n)$ for other source codes $t$, as though it may be the case that $t$ is its own source code. And, crucially, the verifier also must verify the length of the fillibuster, hence one sees that the verifier fails to run in polynomial time!

That is to say, in the case of Turing machines which cannot figure out which source code is their own, the problem is not a problem with a polynomial time verifier.

In real cases, in practise, computer programs can, if they have permission, examine the region in memory or on a hard drive where the program or installer resides. Or to know a 'hash' of the source or object code.

We can set up a problem now with a polynomial time verifier, which poses the problem of finding a proof of $S(s, n)$ for natural numbers $n$ and source codes $s$. The Turing machines to which we'll pose the problem will be enhanced, with access function code(), returning its own source code. Also, when it is to verify the work of a Turing machine, we will *give the verifier access to the source code of the Turing machine.*

The function code() is available to most real-life computer programs. Each Turing machine with source when given the input $S(s, n)$ can find its own source code, letting $t = code()$ and output 'obviously' if $s \neq t$ But if $t = s$ the Turing machine must fillibuster for at least $n$ steps and then (implicitly) point out that the source code includes the fillibuster. The only reason verifier needs to be given access to the source code of the turing machine being tested is to verify that when the Turing machine claims that $t$ is not its own code, it is not just lying. Only when $t = s$ does it need to be able to verify the purported proof which claims that the source code includes the fillibuster.

For this case, we have flexibility in setting up the formal language, to restrict what types of proofs are acceptable. We are in a position not unlike when formulating government regulations; we can insist that it is a requirment for the Turing machine not only to wait $n$ steps, but to 'be seen to have waited $n$ steps' by the requirement to insert a particular 'government approved' sequence of code that is easily verifiable proof of having waited.

## Conclusion.

I sense that if this were set up and tested, what one would see is something like Turing machines going into loops and trying to parse 'that silly statement,' which is in fact endless, and then giving up after a certain count. That is, although the way we as humans, or at least me, can understand this currently is along the lines of a party game, I actually do think that it illustrates that purported polynomial time proof-finding algorithms can always be 'hacked.'

It is interesting that we needed to give the algorithms actually extra power, more power than a Turing machine has, before we could hack them. The fact that we can't quite hack a pure Turing machine, being weaker, obviously does not imply that there is a polynomial time Turing proof finder. Only that it is hard to hack Turing machines to demonstrate them failing.

If we remove the condition that the program can recognize its own source code, the 'government' requirement about proofs becomes impossible to satsify, since it must be applied to source codes which the designer of the solving program cannot modify.

John Atwell Moody
19 May, 2016