the Halting Bot

Eric C.R. Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

The famous halting problem imagines that there is a program to compute halting. I'll use the Python programming language, but it doesn't matter which programming language we use. The definition of a halting function in Python begins like this:

def *halts* (*p*, *x*):

The function is named *halts*. It has two parameters *p* and *x*. Parameter *p* is the name of a Python function that has one parameter *x*. There is also a dictionary of definitions so that *halts* can look up any function name in the dictionary and retrieve its definition for analysis. We won't (can't) write the body of *halts*, but we imagine that it analyses *p* and any function called by *p* to determine whether execution of *p* (*x*) terminates. If *halts* determines that execution of *p* (*x*) terminates, it returns "yes", and if *halts* determines that execution of *p* (*x*) does not terminate, it returns "no".

Here is a function to try *halts* on.

def *twist* (*x*): **if** *halts* ("*twist*", "*twist*") == "yes": *twist* (*x*)

The function is named *twist*. It has one parameter *x*. If *halts* ("*twist*", "*twist*") says "yes", meaning that execution of *twist* ("*twist*") terminates, then *twist* calls itself recursively with the same argument *x* as the original call, creating a nonterminating loop; therefore the result of *halts* ("*twist*", "*twist*") was wrong. If *halts* ("*twist*", "*twist*") says "no", meaning that execution of *twist* ("*twist*") does not terminate, then execution of *twist* terminates; again, the result of *halts* ("*twist*", "*twist*") was wrong. The conclusion is that it is impossible to write the body of *halts* to always give correct answers.

The argument in the preceding paragraphs was unnecessarily complicated because parameter *x* played no role. Let's get rid of it. We redefine *halts* with this definition header:

def *halts* (*p*):

It applies to Python functions that have no parameters. And we apply it to *twist*, defined as

def *twist* ( ): **if** *halts* ("*twist*") == "yes": *twist* ( )

If *halts* ("*twist*") says "yes", then *twist* calls itself recursively, and execution does not terminate. If *halts* ("*twist*") says "no", then execution of *twist* terminates. Either way, *halts* gives the wrong answer. It is impossible to write the body of *halts* to always give correct answers.

The argument is still unnecessarily complicated because we don't need *halts* to tell us about all Python functions. We just need it to tell us about *twist*. Let's get rid of *p*. We redefine *halts* as

def *halts* ( ): **return** "yes"

if execution of *twist* ( ) terminates, and

def *halts* ( ): **return** "no"

if it doesn't. But according to the definition of *twist*,

def *twist* ( ): **if** *halts* ( ) == "yes": *twist* ( )

if *halts* ( ) returns "yes" then execution of *twist* is nonterminating, and if *halts* ( ) returns "no" then execution of *twist* is terminating. Therefore it is impossible to define *halts* with the intended meaning. We are supposed to conclude that the constant *halts* is incomputable. The word “incomputable” suggests that there is a correct answer, but we cannot compute it. In fact,

there isn't a correct answer, which means the definition of *halts* is inconsistent. The same was true for the earlier versions in which *halts* had a parameter or two, but the inconsistency was obscured by the parameters.

The argument can be further simplified by eliminating the call from *twist* as follows:

def *twist* ( ): **if** (execution of *twist* terminates): *twist* ( )

Function *twist* is not fully written, just as *halts* was not fully written. To complete the definition, we just need to replace (execution of *twist* terminates) by either **True** or **False** , whichever one is correct. But neither is correct.

Here is an imagined conversation between Alan, who channels Alan Turing but in modern language, and an AI bot named Botty.

Alan: Can you solve the halting problem?
Botty: Yes.
Alan: Do you work by computation?
Botty: Yes.
Alan: But Turing proved that halting is incomputable, and all computer scientists accept that proof.
Botty: Almost all. There are one or two who don't accept it.
Alan: I challenge you to try the following Python procedure.

def *twist* ( ): **if** *BottySays* ( ) == "yes": *twist* ( )

Does its execution halt? Please answer “yes” or “no”.
Botty: Your program is incomplete. The function named *twist* calls another function named *BottySays* that you have not yet defined. Please define it so that I can answer your question.
Alan: I will define it to be your answer to my question.
Botty: Then we are at an impasse (or deadlock). I am waiting for you to define it, and you are waiting for me to answer.

Which position is more reasonable?
Botty: I will say whether execution terminates after you show me the program.
Alan: I will show you the program after you say whether its execution terminates.

In Turing's original halting problem proof, it is supposed that all programs already exist, including *halts* and *twist* . There is no impasse about which program is written first. We conclude that *halts* cannot tell us whether execution of *twist* terminates. Or, if you prefer, we conclude that the supposition that *halts* exists and tells us whether execution of *twist* terminates was wrong. This is existence in the mathematical sense: something exists if it is not inconsistent to talk about it. Mathematical existence is timeless.

Software engineers do not consider that all programs already exist. Each day they write new programs that did not exist the day before. That is existence in the physical sense: a thing exists if you can sense it. Physical existence depends on time; a thing can exist at one time, but not at another time. By telling a story about Alan and Botty, I made existence a physical question. If *halts* does not exist, a software engineer can write it, so that it now exists, and tells whether the execution of all previously existing programs terminates. But then, the next day, someone can write *twist* , which calls *halts* , so that *halts* does not tell whether the execution of *twist* terminates. And the day after that, *halts2* can be written that works for all previous programs

including *twist* . And then *twist2* can be written for which *halts2* does not work, and so on, in leapfrog fashion.

Earlier, we supposed *halts* and *twist* are both written in Python. It didn't matter that the language was Python, but it did matter that *halts* and *twist* are in the same language, so *twist* can call *halts* . In Turing's proof, he numbered programs, and he could do that because they were assumed to be written in a specific language, the Turing Machine language. Then he imagined a *twist*-like program, which has a number because it is in the Turing Machine language, and then a *halts* program, which also has a number because it is in the same language. When Alan was talking to Botty, he wrote *twist* in Python, and he waited for Botty to answer so he could translate that answer into Python. The inconsistency mentioned earlier, from which we concluded that the *halts* program cannot be written, was a result of insisting that *halts* is written in the same language as the programs to which it applies.

That gives us a way out of both loops (the logical loop of mathematical existence, and the temporal loop of physical existence): just write *halts* in a different language from the programs to which it applies. Then *twist* cannot be written because it cannot call *halts* . If you are a software engineer, you may say there is a way for programs in one language to call programs in another language. I have two answers to that. One answer is to say that if programs in language *L0* can call programs in language *L1* , then effectively *L0* and *L1* together are one language. My other, perhaps better, answer is to say that *halts* , written in *L1* , applies to all those programs in *L0* that do not call programs in other languages. Those *L0* programs are a Turing-complete set of programs, so we can say that halting is computable.

There is one more objection to my suggestion that we can compute halting for all programs in Turing-Machine-Equivalent language *L0* by writing *halts* in a different language *L1* . If we could write *halts* in *L1* , we could just translate it into *L0* . But we know there is no *L0* program to compute halting for all *L0* programs. So there can't be an *L1* program to do so either. This objection is false because program translation does not necessarily preserve meaning. Here is a simple example of natural language translation that does not preserve meaning. The sentence “This sentence is in English.” is true. Its translation into French, “Cette phrase est en anglais.”, is false. Similarly there can be a *halts* program in *L1* which gives correct answers for every *L0* program. We translate it into *L0* . The *L0 twist* program calls the *L0 halts* . Perhaps the *L1 halts* correctly says that execution of *L0 twist* halts. Its translation, *L0 halts* , incorrectly says that *L0 twist* does not halt. And that is why execution of *L0 twist* halts, making *L1 halts* correct.

Botty, the AI bot, is a program. But it is not just a function of its immediate input. For example, you might ask Botty “How many times have you answered this question?”. Perhaps the answer is 3 , and that's what Botty says. Then you ask the exact same question again, and get the answer 4 . If the *Botty* program is written in Python, then Alan can write *twist* to call *Botty* and find out its answer. Even if it's not written in Python, perhaps there is a way for *twist* to call it and find out its answer. What will it say about whether execution of *twist* terminates? Since Botty is intelligent, I expect it to say that there is no correct answer. This does not fulfill the requirements of the halting problem, which demands a yes or no answer. Neither does it fulfill my requirements, which demand that it is not callable from the programs to which it applies. I see no reason why we cannot write a program in some language other than Python, not callable from Python, that computes halting for all Python programs.