the Halting Program

Eric C.R. Hehner

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

The Halting Problem is this: write a Pascal function to determine whether the execution of any Pascal procedure terminates. The choice of programming language was arbitrary; it could have been any other language. Without loss of generality and without changing the character of the problem, I consider halting for procedures with no parameters; input to a procedure can always be replaced by a definition of a sequence of values, call it *input* , within the procedure. The function to determine halting, let's call it *halts* , has one text parameter to say what procedure we are applying it to. We could pass the whole procedure as text, but it is simpler to pass just the procedure name as text, and to provide a dictionary of function and procedure definitions that is accessible to *halts* , so that the call *halts* ('*p*') allows *halts* to look up '*p*' in the dictionary, and retrieve its text for analysis. The dictionary also allows *halts* to look up the texts of all functions and procedures that *p* calls, and so on, transitively.

Here is the function header of *halts* , but its body is missing; in place of the body there is a comment to specify the desired computation. Then there are three procedures, *stop* , *go* , and *twist* , for *halts* to apply to. And finally there is a main block that applies *halts* to the *twist* procedure.

```
function halts (p: string): boolean;
{ return true if p represents a parameterless Pascal procedure whose execution terminates; }
{ return false otherwise }

procedure stop; begin end;

procedure go; begin go end;

procedure twist; begin if halts ('twist') then go else stop end;

begin if halts ('twist') then twist end.
```

The execution of procedure *stop* terminates immediately, so according to the specification of *halts* , the result of *halts* ('*stop*') should be *true* . The execution of procedure *go* never terminates, so according to the specification of *halts* , the result of *halts* ('*go*') should be *false* . If the result of *halts* ('*twist*') is *true* , then *twist* calls *go* , whose execution does not terminate, so the result of *halts* ('*twist*') should be *false* . If the result of *halts* ('*twist*') is *false* , then *twist* calls *stop* , whose execution terminates, so the result of *halts* ('*twist*') should be *true* . This is an inconsistency in the specification of *halts* . Therefore *halts* cannot be programmed according to its specification.

The preceding argument is one of the standard arguments that halting is incomputable. Elsewhere [0] I have argued that halting for all Pascal procedures might be computed in some other language that is not callable from Pascal. In this paper, I show how halting for a lower-level language might be computed in that same language.

There are infinitely many Pascal procedures and functions, so the dictionary of procedure and function definitions is infinite. But let me suppose that we want *halts* to apply to only the three procedures defined above, so the above definitions (including *halts* , when it is written) are a sufficient amount of the dictionary. This is obviously a limitation, but at least it includes an example of termination, an example of nontermination, and the *twist* example that is used to prove the impossibility of programming *halts* .

Here is a start at programming *halts* .

```
function halts (p: string): boolean;
begin if p='stop' then halts:= true
        else if p='go' then halts:= false
              else if p='twist' then if (I am called from outside  twist ) then halts:= true
                                     else halts:= false end
```

Or, more succinctly and less perspicuously,

```
function halts (p: string): boolean;
begin halts:= (p='stop') or ((p='twist') and (I am called from outside  twist )) end
```

All that's missing is to replace (I am called from outside *twist*) by some programming that determines whether the call to *halts* came from outside or inside *twist* . Function *halts* is supposed to apply to all procedures, not just these three, so the *halts* programmer cannot analyze all of them and simply list the answers, as I have done for these three procedures. So the *halts* program must do the analysis, and I leave the hard work of programming that analysis to someone else. My purpose is to show how *halts* ('*twist*') can overcome the apparent inconsistency in its specification.

Before programming that missing piece, let's see how *halts* will work. The main block calls *halts* ('*twist*') . In *halts* , *p*='*twist*' , and then (I am called from outside *twist*) is somehow determined to be *true* , and so the result of *halts* ('*twist*') is *true* . Returning to the main block, *twist* is executed. The execution of *twist* calls *halts* ('*twist*') . Once again in *halts* , *p*='*twist*' , and this time (I am called from outside *twist*) is somehow determined to be *false* , so the result of *halts* ('*twist*') is *false* . Returning to *twist* , *stop* is executed, and the execution of *twist* terminates, and the execution of the main block also terminates.

In the main block, *halts* said that the execution of *twist* terminates, and then, true to its word, the execution of *twist* terminated. If I may indulge in anthropomorphism, *twist* tried to trick *halts* , but *halts* tricked *twist* , and thus preserved its own integrity. Or maybe *halts* sacrificed its integrity when tricking *twist* : it is supposed to tell the truth all the time, but it lied to *twist* . In this paper, I require *halts* to tell the truth only when asked from the main block, outside all procedures.

I do not know how to determine where a call came from in Pascal; perhaps it is impossible. Pascal must be compiled (translated) to machine language for execution, and it is easy to determine where a call came from in machine language. I will take a step in the direction of machine language, but remain in Pascal: I remove functions and procedures and calls from Pascal. The remaining language is still Turing Machine equivalent; there is no call instruction in the Turing Machine language. Where a modern programming language uses a call, Turing used an interpreter, named the Universal Machine. Machine instructions for call and return require a stack for return addresses; I have put the stack size at ∞ , which is not in Pascal, because Turing Machines have an infinite memory; in this example program, stack size 2 is sufficient. If *halts* were recursive, then the parameter values and result values would also need a stack, but in our example, *halts* is not recursive, so a parameter variable and result variable are sufficient. Some of the calls are last action calls; a good compiler will compile them as simple branch instructions; I replace them with **goto**s. Execution begins at *main* , which is label 0, and ends at the end of *main* , which is label 7.

```
var return: array [0..∞] of integer;  {return address stack}
var top: integer; {the number of filled items and first free index in  return }
var p: string;  {in place of the parameter for  halts }
var result: boolean;  {in place of the result returned by  halts }

{halts} 1: begin result:= (p='stop') or ((p='twist') and (return[top–1] = 6));
                 top:= top–1;  goto return[top] end;

{stop} 2: begin top:= top–1;  goto return[top] end;

{go} 3: begin goto 3 end;

{twist} 4: begin p:= 'twist';  return[top]:= 5;  top:= top+1;  goto 1;
               5: if result then goto 3 else goto 2 end;

{main} 0: begin return[0]:= 7;  top:= 1;  {return address stack initialization}
                p:= 'twist';  return[top]:= 6;  top:= top+1;  goto 1;
                6: if result then goto 4;  7: end.
```

Obviously, this *halts* function is a long way from fulfilling the original specification. Its one merit is to show that in a low-level language, such as Pascal without functions and procedures and call, or any assembly language, or any machine language, including Turing Machine language, we can "compute halting" for that language. By "compute halting", I mean compute, within the *main* block of code, whether execution of any other block of code terminates. But we cannot compute, within a block of code, whether execution of that same block of code terminates. To fulfill the original specification, *halts* should compute halting for any block of code, and that is impossible.

Acknowledgement

I acknowledge that Bill Stoddart published this same idea, and a lot more, in a brilliant paper [1] in 2017. I also acknowledge that in 2018 Song Zhou had a closely related idea.

Reference

[0] E.C.R.Hehner: Epimenides, Gödel, Turing: an Eternal Gölden Twist
[1] W.Stoddart: Halting Misconceived. EuroForth 2017, http://www.complang.tuwien.ac.at/anton/euroforth/ef17/papers/stoddart.pdf

other papers on halting