How to Compute Halting

Eric C.R. Hehner

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

Abstract: A consistently specified halting function can be computed.

Halting Problem

Here is the function header and specification of a Pascal function $halts$ to compute the termination status of Pascal procedures; the function body is absent. Following that, we have a Pascal procedure $diag$ in its entirety.

function $halts(p, i$: string): string;
{ return 'yes' if $p$ represents a Pascal procedure with one string input parameter }
{ whose execution terminates when given input $i$; }
{ return 'no' if $p$ represents a Pascal procedure with one string input parameter }
{ whose execution does not terminate when given input $i$; }
{ return 'not applicable' if $p$ does not represent a Pascal procedure }
{ with one string input parameter }

procedure $diag(s$: string);
begin
  if $halts(s, s) = 'yes'$ then $diag(s)$
end

What should the result of $halts('diag', 'diag')$ be? This is a question about the specification of $halts$. Let's look at each possibility in turn.

Should the result of $halts('diag', 'diag')$ be 'not applicable'? Syntactically, $diag$ is a procedure; to determine that $halts$ is being used correctly within $diag$, we need only the header for $halts$, not the body, and we have the header. Semantically, it is a procedure; to determine the meaning of the call to $halts$ within $diag$, we need only the specification of $halts$, not its implementation, and we have the specification. (That important programming principle enables a programmer to call procedures written by other people, knowing only the specification, not the implementation. It also enables a programmer to change the implementation of a procedure, but still satisfying the specification, without knowing where and why the procedure is being called.) So there is nothing wrong with the definition of $diag$, and the result should not be 'not applicable'.

Should the result of $halts('diag', 'diag')$ be 'yes'? If so, the semantics of $diag('diag')$ is nontermination, so it should be 'no'.

Should the result of $halts('diag', 'diag')$ be 'no'? If so, the semantics of $diag('diag')$ is termination, so it should be 'yes'.

We have ruled out all possibilities. This is inconsistent. Therefore $halts$ cannot be programmed according to its specification.
How to Compute Limited Halting

It is inconsistent to ask for a Pascal function to compute the halting status of all Pascal procedures. But we can ask for a Pascal function to compute the halting status of some Pascal procedures. For example, a function to compute the halting status of just the two procedures

procedure loop (s: string); begin loop (s) end

procedure stop (s: string); begin end

is easy. Perhaps we can ask for a Pascal function to compute the halting status of all Pascal procedures that do not refer to this halting function, neither directly nor indirectly. Here is its header, specification, and a start on its implementation.

function halts (p, i: string): string;
{ return 'yes' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to halts (neither directly nor indirectly) }
{ and whose execution terminates when given input i ; }
{ return 'no' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to halts (neither directly nor indirectly) }
{ and whose execution does not terminate when given input i ; }
{ return 'maybe' if p represents a Pascal procedure with one string input parameter }
{ that refers to halts (either directly or indirectly); }
{ return 'not applicable' if p does not represent a Pascal procedure }
{ with one string input parameter }
begin
if ( p does not represent a Pascal procedure with one string input parameter)
  then halts:= 'not applicable'
else if ( p refers to halts directly or indirectly)
  then halts:= 'maybe'
else (return halting status of p , either 'yes' or 'no' )
end

The first case checks whether p represents a (valid) procedure exactly as a Pascal compiler does. The middle case is the transitive closure algorithm. The last case may not be easy, perhaps it is impossible, but at least it is free of the reason it has been called impossible to program: that it cannot cope with procedure diag . Procedure diag refers to halts by calling it, so halts is not required to determine its halting status, halts ('diag','diag') = 'maybe' , and execution of diag ('diag') is terminating.

Calling is one kind of referring, but not the only kind. In the specification of halts , the name halts appears, and also in the body. These are self-references, whether or not halts calls itself. We exempt halts from applying to procedures containing these and all other forms of reference to halts .

We might try to circumvent the proscription by writing another function halts1 that is identical to halts but renamed (including in the return statements and any recursive calls).

function halts1 (p, i: string): string;
{ return 'yes' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to halts (neither directly nor indirectly) }
{ and whose execution terminates when given input i ; }
{ return 'no' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to halts (neither directly nor indirectly) }
and whose execution does not terminate when given input \( i \); }
{ return 'maybe' if \( p \) represents a Pascal procedure with one string input parameter }
{ that refers to \( \text{halts} \) (either directly or indirectly); }
{ return 'not applicable' if \( p \) does not represent a Pascal procedure }
{ with one string input parameter }
begin
{ if \( p \) does not represent a Pascal procedure with one string input parameter)
    then \( \text{halts1} := 'not applicable' \)
{ else if \( p \) refers to \( \text{halts} \) directly or indirectly)
    then \( \text{halts1} := 'maybe' \)
{ else (return halting status of \( p \), either 'yes' or 'no' ) }
end

Now we define a procedure \( \text{diag1} \) that \( \text{halts1} \) cannot cope with.

procedure \( \text{diag1} (s: \text{string}); \)
begin
{ if \( \text{halts1} (s, s) = 'yes' \) then \( \text{diag1} (s) \)
end

Now \( \text{halts1} \) cannot tell the halting status of \( \text{diag1} \), and since \( \text{halts1} \) is identical to \( \text{halts} \), therefore \( \text{halts} \) also does not tell the halting status of \( \text{diag1} \), even though \( \text{diag1} \) does not call \( \text{halts} \). But \( \text{diag1} \) calls \( \text{halts1} \) and \( \text{halts1} \) refers to \( \text{halts} \), so both \( \text{halts1} ('\text{diag1}', '\text{diag1}') \) and \( \text{halts ('diag1')} \) return 'maybe', and execution of \( \text{diag1} ('\text{diag1}') \) terminates.

Perhaps the renaming should have included the references in the specification and body. So let's try again.

function \( \text{halts2} (p, i: \text{string}): \text{string}; \)
{ return 'yes' if \( p \) represents a Pascal procedure with one string input parameter }
{ that does not refer to \( \text{halts2} \) (neither directly nor indirectly) }
{ and whose execution terminates when given input \( i \); }
{ return 'no' if \( p \) represents a Pascal procedure with one string input parameter }
{ that does not refer to \( \text{halts2} \) (neither directly nor indirectly) }
{ and whose execution does not terminate when given input \( i \); }
{ return 'maybe' if \( p \) represents a Pascal procedure with one string input parameter }
{ that refers to \( \text{halts2} \) (either directly or indirectly); }
{ return 'not applicable' if \( p \) does not represent a Pascal procedure }
{ with one string input parameter }
begin
{ if \( p \) does not represent a Pascal procedure with one string input parameter)
    then \( \text{halts2} := 'not applicable' \)
{ else if \( p \) refers to \( \text{halts2} \) directly or indirectly)
    then \( \text{halts2} := 'maybe' \)
{ else (return halting status of \( p \), either 'yes' or 'no' ) }
end

procedure \( \text{diag2} (s: \text{string}); \)
begin
{ if \( \text{halts2} (s, s) = 'yes' \) then \( \text{diag2} (s) \)
end

Now \( \text{halts2} ('\text{diag2}', '\text{diag2}') \) = 'maybe' because \( \text{diag2} \) refers to \( \text{halts2} \) by calling it. And \( \text{halts ('diag2', 'diag2')} \) = 'yes' because \( \text{diag2} \) does not refer to \( \text{halts} \) and execution of
diag2 (‘diag2’) terminates. Even though halts2 is obtained from halts by renaming, they produce different results.

**How to Compute Unlimited Halting**

In Pascal, as originally defined, identifiers cannot contain underscores. I now define a new programming language, Pascal_, which is identical to Pascal except that identifiers can contain underscores. In this new language, perhaps we can write a function named halts_ that determines the halting status of all Pascal procedures. Pascal procedures are syntactically prevented from referring to halts_, so the 'maybe' option disappears.

```pascal
function halts_(p, i: string): string;
{ return 'yes' if p represents a Pascal procedure with one string input parameter }
{ whose execution terminates when given input i; }
{ return 'no' if p represents a Pascal procedure with one string input parameter }
{ whose execution does not terminate when given input i; }
{ return 'not applicable' if p does not represent a Pascal procedure }
{ with one string input parameter }
begin
  if (p does not represent a Pascal procedure with one string input parameter)
    then halts_:= 'not applicable'
  else (return halting status of p , either 'yes' or 'no')
end
```

It should be clear that, by allowing underscores in identifiers, we do not increase the computing power of Pascal_ beyond the Turing-Machine-equivalent power of Pascal. If it is possible to write a Pascal function to compute the halting status of all Pascal procedures that do not refer to this halting function, then by writing in another language, we can compute the halting status of all Pascal procedures.

There is an argument that, at first sight, seems to refute the possibility of computing the halting status of all Pascal procedures just by programming in another language. Suppose that in writing halts_ we do not use any underscores in any other identifiers. Then we can easily obtain a Pascal function halts just by deleting the underscores from the halts_ identifier. We thus obtain a Pascal function with the same functionality: halts (s) = halts_ (s) for all s. But there cannot be a Pascal function that computes the halting status of all Pascal procedures. Therefore, the argument concludes, there cannot be a Pascal_function to do so either.

As compelling as the previous paragraph may seem, it is wrong. If I say “My name is Eric Hehner.” I am telling the truth. If Margaret Jackson says exactly the same words, she is lying. When I say it, there is a self-reference; when Margaret Jackson says it, there is no self-reference. The truth of that sentence depends on who says it. We can also write programs whose behavior is dependent on the name of the program. In the C language on the UNIX platform, the name of a program is accessible as argv[0], so we can write a program that prints its own name. Due to the self-reference, just renaming a program can change its behavior. Even though halts and halts_ are textually identical except for name, they do not have the same behavior when applied to diag due to the fact that there is a self-reference in one case and not in the other. When halts is applied to diag , there is an indirect self-reference, and its result is wrong; when halts_ is applied to diag , there is no self-reference, and it gives the correct result.
Naming

My presentation of the halting problem has relied heavily on the fact that functions and procedures in Pascal have names. Turing Machine procedures do not have names. For each procedure there is a numeric code, and the Universal Turing Machine is the decoder. In modern terms, the code for a procedure is its text, or string, and the Universal Turing Machine is an interpreter. The more convenient modern equivalent is to give a procedure a name, and to call it by invoking its name.

I applied \texttt{halts} to \texttt{diag} by encoding the name as a character string: \texttt{halts (\textquotesingle diag\textquotesingle, \textquotesingle diag\textquotesingle)}. I could equally well have encoded the procedure, like this:

\texttt{halts (procedure diag (s: string); begin if halts (s, s) then diag (s) end, procedure diag (s: string); begin if halts (s, s) then diag (s) end)}

The presentation would have been less clear, but the arguments would be unchanged. The fact that Pascal procedures have names makes it easy to tell the story, but the names are not necessary. If Pascal had nameless procedures, I would have written

\texttt{halts (procedure (s: string); begin if halts (s, s) then loop: goto loop end, procedure (s: string); begin if halts (s, s) then loop: goto loop end)}

The name \texttt{diag} is unnecessary; in effect, the string representing the procedure is its name.

Within \texttt{diag}, the meaning of the call to function \texttt{halts}, like the meaning of any call, is its specification, not its implementation. So we can replace the name \texttt{halts} with its specification. It is the specification, not the nonexistent body, that is used to reach the inconsistency conclusion. The assumption that there is a body, which is the computability assumption, is of no use in reaching that conclusion.

Conclusion

By weakening the specification a little, reducing the domain from “all procedures” to “all procedures that do not refer to the halting function”, we obtain a specification that may be both consistent and computable. Equivalently, we may be able to compute the halting status of all procedures in a Turing-Machine-equivalent language by writing a halting function in another Turing-Machine-equivalent language, assuming that the procedures of the first language cannot refer to the halting function written in the second language.