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.

```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 }

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

```pascal
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.

```pascal
function halts1 (p, i: string): string;
{ return 'yes' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to halts1 (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 halts1 (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 halts1 (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 halts1 := 'not applicable'
  else if (p refers to halts1 directly or indirectly)
    then halts1 := '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 looks like a transitive closure algorithm, but it is problematic because, theoretically, there can be an infinite chain of calls. Thus we may be able to compute halting for this limited set of procedures, but not determine whether a procedure is in this limited set. The last case may not be easy, but at least it is free of the reason it has been called incomputable: that it cannot cope with

```pascal
procedure diag1 (s: string); { execution terminates if and only if halts1 (s, s) ≠ 'yes' }
begin
  if halts1 (s, s) = 'yes' then diag1 (s)
end
```

Procedure \( \text{diag1} \) refers to \( \text{halts1} \) by calling it, so \( \text{halts1} \) is not required to determine the halting status of \( \text{diag1} \). Therefore \( \text{halts1} (\text{diag1'}, \text{diag1'}) = 'maybe' \), and execution of \( \text{diag1} (\text{diag1'}) \) is terminating.

Calling is one kind of referring, but not the only kind. In the specification of \( \text{halts1} \), the name \( \text{halts1} \) appears, and also in the body. These are self-references, whether or not \( \text{halts1} \) calls itself. We exempt \( \text{halts1} \) from applying to procedures containing any form of reference to \( \text{halts1} \). We might try to circumvent the proscription by writing another function \( \text{halts2} \) that is identical to \( \text{halts1} \) but renamed (including in the specification, the return statements, and any recursive calls).
function halts2 (p, i: string): string;
{ return 'yes' if p represents a Pascal procedure with one string input parameter }
{ that does not refer to 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 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 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 halts2:= 'not applicable'
  else if (p refers to halts2 directly or indirectly)
    then halts2:= 'maybe'
  else (return halting status of p , either 'yes' or 'no')
end;

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

Now halts2 ('diag2', 'diag2') = 'maybe' because diag2 refers to halts2 by calling it. And halts1 ('diag2', 'diag2') = 'yes' because diag2 does not refer to halts1 and execution of diag2 ('diag2') terminates. Even though halts2 is obtained from halts1 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 problem of determining whether a Pascal procedure refers to halts_ disappears, along with the 'maybe' option.

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 \texttt{halts\_} we do not use any underscores in any other identifiers, and we do not use the identifier \texttt{halts}. Then we can easily obtain a Pascal function \texttt{halts} just by deleting the underscore from the \texttt{halts\_} identifier. We thus obtain a Pascal function with the same functionality: \texttt{halts}(s) = \texttt{halts\_}(s) for all \texttt{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. Here is a Pascal\_ procedure that prints its own name.

\begin{verbatim}
procedure A_; {this procedure prints its own name}
begin print ('A_') end
\end{verbatim}

Translating this procedure to Pascal, we face a dilemma. We could translate it as

\begin{verbatim}
procedure A; {this procedure prints its own name}
begin print ('A.') end
\end{verbatim}

arguing that the two procedures have the same output, but clearly this translation does not preserve the intention. The Pascal\_ procedure \texttt{A\_} meets its specification; the Pascal translation \texttt{A} does not. Or we could translate it as

\begin{verbatim}
procedure A; {this procedure prints its own name}
begin print ('A') end
\end{verbatim}

arguing that we have preserved the intention, but clearly the two procedures do not have the same output. Translating from \texttt{halts\_} to \texttt{halts} has the same problem. We cannot preserve the intention because the specification at the head of \texttt{halts\_}, which is perfectly reasonable for a Pascal\_ function, becomes inconsistent when placed at the head of a Pascal function. If we just use the same Pascal\_ procedure but delete the underscores, we obtain a Pascal procedure that no longer satisfies the specification.

There is another argument that, at first sight, also seems to refute the possibility of computing the halting status of all Pascal procedures just by programming in another language. In Pascal, we can write an interpreter for Pascal\_ programs. So if we could write a halting function \texttt{halts\_} in Pascal\_ for all of Pascal, we could feed the text of \texttt{halts\_} to this interpreter, and thus obtain a Pascal function to compute halting for all Pascal procedures. 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.

The reason this argument fails is the same as the reason the previous argument fails. The interpreter interpreting \texttt{halts\_} is just like the translation of \texttt{halts\_} into Pascal by deleting underscores. The interpreter interpreting \texttt{halts\_} can be called by another Pascal program; \texttt{halts\_} cannot be called by a Pascal program. That fact materially affects their behavior. Pascal\_ program \texttt{halts\_} can be applied to a Pascal procedure \texttt{d} that calls the interpreter interpreting \texttt{halts\_} applied to \texttt{d}, and it will produce the right answer. But the interpreter
interpreting \textit{halts\_} applied to \textit{d} calls the interpreter interpreting \textit{halts\_} applied to \textit{d}, and execution will not terminate.

\textbf{the Barber}

A town named Russellville consists of some men (only men). Some of the men shave themselves; the others do not shave themselves. A barber is a person who shaves all and only those men in Russellville who do not shave themselves. There is a barber; his name is Bertrand\_ and he lives just outside the town, in the Greater Russellville Area (called Russellville\_). Without any difficulty, he satisfies the specification of barber.

One of the men in Russellville, whose name is Bertrand, decided that there is no need to bring in a barber from outside town. Bertrand decided that he could do the job. He would shave those men whom Bertrand\_ shaves, and not shave those men whom Bertrand\_ does not shave. If Bertrand\_ is fulfilling the role of barber, then by doing exactly the same actions as Bertrand\_, Bertrand reasoned that he would fulfill the role of barber. But Bertrand is wrong; those same actions will not fulfill the role of barber when Bertrand performs them. To be a barber, Bertrand has to shave himself if and only if he does not shave himself. A specification that is perfectly consistent and doable for someone outside town becomes inconsistent when it has to be performed by someone in town.

And so it is with the halting specification, and for the same reason. For Bertrand\_, the barber specification has no self-reference; for Bertrand, the barber specification has a self-reference. For \textit{halts\_}, the halting specification has no self-reference; for \textit{halts}, the halting specification has a self-reference (indirectly through \textit{diag} and other procedures that call \textit{halts} ).

\textbf{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 character 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, as I have done in the preceding sections.

I applied \textit{halts} to \textit{diag} by encoding the name as a character string: \textit{halts} (\textit{\char'108diag}, \textit{\char'108diag}). I could equally well have encoded the procedure, like this:

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

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

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

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

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

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

Within \textit{diag}, the meaning of the call to function \textit{halts}, like the meaning of any call, is its specification, not its implementation. So we can replace the name \textit{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.