Programs, Specifications, and Halting

Eric C.R. Hehner
Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Question and Answers

The simple question “What is a procedure?” is not so simple to answer, and its answer has far-reaching consequences throughout computer science. By “procedure” I mean any named, callable piece of program; depending on the programming language, it may be a procedure, or function, or method, or something else. To illustrate my points, I will use the Pascal programming language, but the points I make apply to any modern programming language.

Here is a little piece of Pascal programming.

function binexp (n: integer): integer; { for 0≤n<31 , \( binexp (n) = 2^n \) }

procedure toobig; { if \( 2^{20} > 20000 \) , print 'too big' ; otherwise do nothing }
begin
  if binexp (20) > 20000 then print ('too big')
end

Only the header of function \( \text{binexp} \) appears. In place of the body there is a comment to specify the result of the function. But \( \text{toobig} \) is there in its entirety. Now I ask: Is \( \text{toobig} \) a Pascal procedure? And I offer three possible answers.

Program Answer: No. We cannot compile and execute \( \text{toobig} \) until we have the body of \( \text{binexp} \), or at least have a link to the body of \( \text{binexp} \). It is not a procedure until it can be compiled and executed. (We may not have the body of \( \text{print} \) either, and it may not even be written in Pascal, but the compiler does have a link to it, so it can be executed.) Since \( \text{toobig} \) calls \( \text{binexp} \), whose body is missing, we cannot say what is the meaning of \( \text{toobig} \). The specification of \( \text{binexp} \), which is just a comment, is helpful documentation expressing the intention of the programmer, but intentions are irrelevant. We need the body of \( \text{binexp} \) before it is a Pascal function, and when we have the body of \( \text{binexp} \), then \( \text{toobig} \) will be a Pascal procedure.

Specification Answer: Yes. \( \text{toobig} \) conforms to the Pascal syntax for procedures. It type-checks correctly. To determine whether \( \text{binexp} \) is being called correctly within \( \text{toobig} \), we need to know the number and types of its parameters, and the type of result returned; this information is found in the header for \( \text{binexp} \). To determine whether \( \text{print} \) is being called correctly, we need to know about its parameters, and this information is found in the list of built-in functions and procedures. To understand \( \text{toobig} \), to reason about it, to know what its execution will be, we need to know what the result of \( \text{binexp} (20) \) will be, and what effect \( \text{print} \) (‘too big’) will have. The result of \( \text{binexp} (20) \) is specified in the comment, and the effect of \( \text{print} \) (‘too big’) is specified in the list of built-in functions and procedures. We do not have the body of \( \text{binexp} \), and we probably cannot look at the body of \( \text{print} \), but we do not need them for the purpose of understanding \( \text{toobig} \). Even if we could look at the bodies of \( \text{binexp} \) and \( \text{print} \), we should not use them for understanding and reasoning about \( \text{toobig} \). That’s an important programming principle; it allows programmers to work on different parts of a program independently. It enables a programmer to call functions and procedures written by
other people, knowing only the specification, not the implementation. There are many ways
that binary exponentiation can be computed, but our understanding of toobig does not depend
on which way is chosen. Likewise for print. This important principle also enables a
programmer to change the implementation of a function or procedure, such as binexp and
print, but still satisfying the specification, without knowing where and why the function or
procedure is being called. If there is an error in implementing binexp or print, that error
should not affect the understanding of and reasoning about toobig. So, even without the
bodies of binexp and print, toobig is a procedure.

Who Cares Answer: Who cares? What we have and don’t have is clear; what we can and
cannot do is clear. We have the header and specification of binexp, but not its body. We can
understand and reason about toobig, and we know what its execution should be, but we cannot
compile it and see this execution. There’s no mystery. The only question is whether to use the
word “procedure” to describe toobig. That’s not a substantive question; it’s a question of
terminology. To answer, we don’t need an investigation; we just need to decide.

The semantics community has decided on the Program Answer. For them, the meaning of a
function or procedure is its body, not its specification. They do not assign a meaning to toobig
until the bodies of binexp and print are provided.

The verification community has also decided on the Program Answer. To verify a program that
contains a call, they insist on seeing the body of the procedure/function being called. They do
not verify that ‘too big’ is printed until the bodies of binexp and print are provided.

It may be of some use, perhaps for debugging, to be able to execute programs with unresolved
calls. A Pascal implementation can provide a stub for every unresolved call. Perhaps the
integer 0, or a random integer, could be supplied for an unresolved function call that returns an
integer. Perhaps the empty body could be supplied for an unresolved procedure call. Then
toobig can be executed even though the body of binexp has not been written. The execution
will not satisfy the specification, but it may be of some help, and the Program Answer says the
specification is irrelevant anyway. I am neither advocating nor warning against this suggestion;
I am saying that the presence of the body of binexp is not necessary for some kind of
execution of toobig.

Halting Problem

The Halting Problem is widely considered to be a foundational result in computer science.
Here is a modern presentation of it.

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); { execution terminates if and only if halts (s, s) ≠ ‘yes’ }
begin
  if halts (s, s) = ‘yes’ then diag (s)
end

For function halts, only its header and specification appear; I cannot write its body. Function
halts is called within diag. This raises the question of whether diag is a Pascal procedure.
The result of applying \textit{halts} to \textit{diag} depends on the answer to this question.

Here is the “textbook proof” that \textit{halts} is incomputable.

Assume the body of function \textit{halts} has been written according to its specification. Does execution of \textit{diag} (\textit{\textquotedbl\text{\texttt{diag}}}\textquotedbl) terminate? If it terminates, then \textit{halts} (\textit{\texttt{diag}}, \textit{\texttt{\textquotedbl\text{\texttt{diag}}}\textquotedbl}) returns \textit{‘yes’} according to its specification, and so we see from the body of \textit{diag} that execution of \textit{diag} (\textit{\texttt{diag}}) does not terminate. If it does not terminate, then \textit{halts} (\textit{\texttt{diag}}, \textit{\texttt{\textquotedbl\text{\texttt{diag}}}\textquotedbl}) returns \textit{‘no’}, and so execution of \textit{diag} (\textit{\texttt{diag}}) terminates. This is a contradiction (inconsistency). Therefore the body of function \textit{halts} cannot have been written according to its specification; \textit{halts} is incomputable.

The “textbook proof” begins with the computability assumption: that the body of \textit{halts} can be written, and has been written. The assumption is necessary for advocates of the Program Answer to say that \textit{diag} is a Pascal procedure, and so rule out ‘not applicable’ as the result of \textit{halts} (\textit{\texttt{diag}}, \textit{\texttt{\textquotedbl\text{\texttt{diag}}}\textquotedbl}). If we suppose the result is ‘yes’, then we see from the body of \textit{diag} that execution of \textit{diag} (\textit{\texttt{diag}}) is nonterminating, so the result should be ‘no’. If we suppose the result is ‘no’, then we see from the body of \textit{diag} that execution of \textit{diag} (\textit{\texttt{diag}}) is terminating, so the result should be ‘yes’. Thus all three results are eliminated, we have an inconsistency, and advocates of the Program Answer blame the computability assumption for the inconsistency.

Advocates of the Program Answer must begin by assuming the existence of the body of \textit{halts}, but since the body is unavailable, they are compelled to base their reasoning on the specification of \textit{halts} as advocated in the Specification Answer.

Advocates of the Specification Answer do not need the computability assumption. According to them, \textit{diag} is a Pascal procedure even though the body of \textit{halts} has not been written. What does the specification of \textit{halts} say the result of \textit{halts} (\textit{\texttt{diag}}, \textit{\texttt{\textquotedbl\text{\texttt{diag}}}\textquotedbl}) should be? the Specification Answer eliminates ‘not applicable’. As before, if we suppose the result is ‘yes’, then we see from the body of \textit{diag} that execution of \textit{diag} (\textit{\texttt{diag}}) is nonterminating, so the result should be ‘no’; if we suppose the result is ‘no’, then we see from the body of \textit{diag} that execution of \textit{diag} (\textit{\texttt{diag}}) is terminating, so the result should be ‘yes’. Thus all three results are eliminated. But this time there is no computability assumption to blame. This time, the conclusion is that the body of \textit{halts} cannot be written due to inconsistency of its specification.

Both advocates of the Program Answer and advocates of the Specification Answer conclude that the body of \textit{halts} cannot be written, but for different reasons. According to advocates of the Program Answer, \textit{halts} is incomputable, which means that it has a consistent specification that cannot be implemented in a Turing-Machine-equivalent programming language like Pascal. According to advocates of the Specification Answer, \textit{halts} has an inconsistent specification, and the question of computability does not arise.

\textbf{Simplified Halting Problem}

The distinction between these two positions can be seen better by trimming away some irrelevant parts of the argument. The second parameter of \textit{halts} and the parameter of \textit{diag} play no role in the “textbook proof” of incomputability; any string value could be supplied, or the parameter could be eliminated, without changing the “textbook proof”. The first parameter of \textit{halts} allows \textit{halts} to be applied to any string, but there is only one string we apply it to in the “textbook proof”; so we can also eliminate it by redefining \textit{halts} to apply specifically to \textit{\texttt{\textquoteleft\texttt{diag}\textquoteright}}. Here is the result.
function halts: string;
{ return 'yes' if diag is a Pascal procedure whose execution terminates; }
{ return 'no' if diag is a Pascal procedure whose execution does not terminate; }
{ return 'not applicable' if diag is not a Pascal procedure }

procedure diag; { execution terminates if and only if halts ≠ 'yes' }
begin
  if halts = 'yes' then diag
end

The “textbook proof” that halts is incomputable is unchanged.

Assume the body of function halts has been written according to its specification. Does execution of diag terminate? If it terminates, then halts returns 'yes' according to its specification, and so we see from the body of diag that execution of diag does not terminate. If it does not terminate, then halts returns 'no', and so execution of diag terminates. This is a contradiction (inconsistency). Therefore the body of function halts cannot have been written according to its specification; halts is incomputable.

Function halts is now a constant, not depending on the value of any parameter or variable. There is no programming difficulty in completing the body of halts. It is one of three simple statements: either halts:= 'yes' or halts:= 'no' or halts:= 'not applicable'. The problem is to decide which of those three it is. If the body of halts is halts:= 'yes', we see from the body of diag that it should be halts:= 'no'. If the body of halts is halts:= 'no', we see from the body of diag that it should be halts:= 'yes'. If the body of halts is halts:= 'not applicable', advocates of both the Program Answer and the Specification Answer agree that diag is a Pascal procedure, so again that's the wrong way to complete the body of halts. The specification of halts is clearly inconsistent; it is not possible to conclude that halts is incomputable. The two parameters of halts served only to complicate and obscure.

Printing Problems

The “textbook proof” that halting is incomputable does not prove incomputability; it proves that the specification of halts is inconsistent. But it really has nothing to do with halting; any property of programs can be treated the same way. Here is an example.

function WhatTwistPrints: string;
{ return 'yes' if twist is a Pascal procedure whose execution prints 'yes'; }
{ return 'no' if twist is a Pascal procedure whose execution does not print 'yes'; }
{ return 'not applicable' if twist is not a Pascal procedure }

procedure twist; { if WhatTwistPrints = 'yes' then print 'no'; otherwise print 'yes' }
begin
  if WhatTwistPrints = 'yes' then print ('no') else print ('yes')
end

Here is the “textbook proof” of incomputability, adapted to function WhatTwistPrints.

Assume the body of function WhatTwistPrints has been written according to its specification. Does execution of twist print 'yes' or 'no'? If it prints 'yes', then WhatTwistPrints returns 'yes' according to its specification, and so we see from the body of twist that execution of twist prints 'no'. If it prints 'no', then WhatTwistPrints returns 'no' according to its specification, and so we see from the body of twist that execution of twist prints 'yes'. This is a contradiction.
(inconsistency). Therefore the body of function $\text{WhatTwistPrints}$ cannot have been written according to its specification; $\text{WhatTwistPrints}$ is incomputable.

The body of function $\text{WhatTwistPrints}$ is one of $\text{WhatTwistPrints} := \text{'yes'}$ or $\text{WhatTwistPrints} := \text{'no'}$ or $\text{WhatTwistPrints} := \text{'not applicable'}$ so we cannot call $\text{WhatTwistPrints}$ an incomputable function. But we can rule out all three possibilities, so the specification of $\text{WhatTwistPrints}$ is inconsistent. No matter how simple and clear the specification may seem to be, it refers to itself (indirectly, by referring to $\text{twist}$, which calls $\text{WhatTwistPrints}$) in a self-contradictory manner. That's exactly what the $\text{halts}$ specification does: it refers to itself (indirectly by saying that $\text{halts}$ applies to all procedures including $\text{diag}$, which calls $\text{halts}$) in a self-contradictory manner.

The following example is similar to the previous example.

**function** $\text{WhatStraightPrints}: \text{string} ;$
{ return 'yes' if $\text{straight}$ is a Pascal procedure whose execution prints 'yes' ; }
{ return 'no' if $\text{straight}$ is a Pascal procedure whose execution does not print 'yes' ; }
{ return 'not applicable' if $\text{straight}$ is not a Pascal procedure }

**procedure** $\text{straight};$
{ if $\text{WhatStraightPrints} = \text{'yes'}$ then print 'yes' ; otherwise print 'no' }
begin
if $\text{WhatStraightPrints} = \text{'yes'}$ then print ('yes') else print ('no')
end

To advocates of the Program Answer, $\text{straight}$ is not a Pascal procedure because the body of $\text{WhatStraightPrints}$ has not been written. Therefore $\text{WhatStraightPrints}$ should return 'not applicable', and its body is easily written: $\text{WhatStraightPrints} := \text{'not applicable'}$. As soon as it is written, it is wrong. Advocates of the Specification Answer do not have that problem, but they have a different problem: it is equally correct for $\text{WhatStraightPrints}$ to return 'yes' or to return 'no'.

The halting function $\text{halts}$ has a similar dilemma when applied to

**procedure** $\text{what} (s: \text{string});$
{ execution terminates if and only if $\text{halts} (s, s) = \text{'yes'}$ }
begin
if $\text{halts} (s, s)$ not= 'yes' then $\text{what} (s)$
end

**Resolution**

The Who Cares Answer has not yet received due consideration. A terminology problem can be resolved simply by choosing more precise terms. Let us say “executable procedure” for a procedure that a computer can execute. An executable procedure does not include any non-executable specification or reference to non-executable specification, except possibly as comments that are ignored because they do not affect execution. The meaning of an executable procedure is taken from the executable code; its purpose or intent is irrelevant. An executable procedure is the procedure of the Program Answer.

Let us say “intentional procedure” for a procedure whose purpose is specified, or can be derived from its code and the specifications of procedures and functions it invokes. The meaning of an intentional procedure is its stated or derived purpose. An intentional procedure may or may not also be executable. An intentional procedure is the procedure of the Specification Answer.
In the Halting Problem, we need to be more specific about what kind of procedures \( \text{halts} \) applies to. In the simplified version, we might apply \( \text{halts} \) to executable procedures.

\[
\text{function } \text{halts}: \text{string};
\]
\begin{align*}
\{ \text{ return 'yes' if } \text{diag} \text{ is an executable procedure whose execution terminates; } \} \\
\{ \text{ return 'no' if } \text{diag} \text{ is an executable procedure whose execution does not terminate; } \} \\
\{ \text{ return 'not applicable' if } \text{diag} \text{ is not an executable procedure } \}
\end{align*}

Since we have specified but not programmed \( \text{halts} \),

\[
\text{procedure } \text{diag}; \{ \text{ execution terminates if and only if } \text{halts} \neq 'yes' \}
\begin{align*}
\text{begin} \\
\text{ if } \text{halts} = 'yes' \text{ then } \text{diag}
\end{align*}
\]
\text{end}

is an intentional procedure, but not an executable procedure. So function \( \text{halts} \) should, according to its specification, return 'not applicable', and the execution of \( \text{diag} \) should, according to its intention, terminate.

We can make \( \text{diag} \) an executable procedure by programming \( \text{halts} \). Suppose we program it according to this latest specification.

\[
\text{function } \text{halts}: \text{string};
\]
\begin{align*}
\{ \text{ return 'yes' if } \text{diag} \text{ is an executable procedure whose execution terminates; } \} \\
\{ \text{ return 'no' if } \text{diag} \text{ is an executable procedure whose execution does not terminate; } \} \\
\{ \text{ return 'not applicable' if } \text{diag} \text{ is not an executable procedure } \}
\end{align*}
\[
\text{begin} \\
\text{halts} := 'not applicable'
\end{align*}
\text{end}

Then \( \text{diag} \) is an executable procedure, and its execution terminates. If we replace 'not applicable' with 'yes' or with 'no', then again \( \text{diag} \) is an executable procedure, and we can determine whether its execution terminates. In all three cases, the body of \( \text{halts} \) does not satisfy its specification, but specifications are irrelevant comments for executability.

Suppose we make \( \text{halts} \) more specific the other way, applying it to intentional procedures.

\[
\text{function } \text{halts}: \text{string};
\]
\begin{align*}
\{ \text{ return 'yes' if } \text{diag} \text{ is an intentional procedure whose execution terminates; } \} \\
\{ \text{ return 'no' if } \text{diag} \text{ is an intentional procedure whose execution does not terminate; } \} \\
\{ \text{ return 'not applicable' if } \text{diag} \text{ is not an intentional procedure } \}
\end{align*}

Even without programming \( \text{halts} \), \( \text{diag} \) is an intentional procedure, making 'not applicable' the wrong result. And as argued previously, both 'yes' and 'no' are also wrong, making the specification inconsistent.

Although I have used the simplified version of \( \text{halts} \) in this section, the same arguments and same conclusions apply to the two-parameter version of \( \text{halts} \), but the arguments are cluttered and obscured by the parameters.
Conclusion

The question “What is a procedure?” has at least two defensible answers. If we adopt the answer that a procedure must be executable, then the “textbook proof” of the incomputability of halting cannot be made. That is because the assumption that \( \text{halts} \) is computable and has been programmed does not give us the program; so we have no meaning for \( \text{halts} \), and cannot say whether execution of \( \text{diag} \) terminates. On the other hand, if we adopt the answer that we have a procedure when we know its intention, and know its execution from the specifications of the functions and procedures that it calls, then the specification of \( \text{halts} \) is inconsistent. Either way, the “textbook proof” does not show us a (consistently specified) mathematical function that is incomputable.