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.

```pascal
function binexp (n: integer): integer; { for 0≤n<31 , \(2^n\) }
begin
  if binexp (20) > 20000 then print ('too big')
end
```

Only the header and specification of function `binexp` appear; the body is missing. But `toobig` is there in its entirety. Now I ask: Is `toobig` a Pascal procedure? And I offer three answers.

Program Answer: No. We cannot compile and execute `toobig` until we have the body of `binexp`, or at least a link to the body of `binexp`. `toobig` is not a procedure until it can be compiled and executed. (We may not have the body of `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 `toobig` calls `binexp`, whose body is missing, we cannot say what is the meaning of `toobig`. The specification of `binexp`, which is just a comment, is helpful documentation expressing the intention of the programmer, but intentions are irrelevant. We need the body of `binexp` before it is a Pascal function, and when we have the body of `binexp`, then `toobig` will be a Pascal procedure.

Specification Answer: Yes. `toobig` conforms to the Pascal syntax for procedures. It type-checks correctly. To determine whether `binexp` is being called correctly within `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 `binexp`. To determine whether `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 `toobig`, to reason about it, to know what its execution will be, we need to know what the result of `binexp (20)` will be, and what effect `print ('too big')` will have. The result of `binexp (20)` is specified in the comment, and the effect of `print ('too big')` is specified in the list of built-in functions and procedures. We do not have the body of `binexp`, and we probably cannot look at the body of `print`, but we do not need them for the purpose of understanding `toobig`. Even if we could look at the bodies of `binexp` and `print`, we should not use them for understanding and reasoning about `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 \texttt{toobig} does not depend on which way is chosen. Likewise for \texttt{print}. This important principle also enables a programmer to change the implementation of a function or procedure, such as \texttt{binexp} and \texttt{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 \texttt{binexp} or \texttt{print}, that error should not affect the understanding of and reasoning about \texttt{toobig}. So, even without the bodies of \texttt{binexp} and \texttt{print}, \texttt{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 \texttt{binexp}, but not its body. We can understand and reason about \texttt{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 \texttt{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 \texttt{toobig} until the bodies of \texttt{binexp} and \texttt{print} are provided.

Most of the verification community has 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 \texttt{binexp} and \texttt{print} are provided.

I would like the Software Engineering community to embrace the Specification Answer. That answer scales up to large software; the Program Answer doesn't. The Specification Answer allows us to isolate an error within a procedure (or other unit of program); the Program Answer doesn't. The Specification Answer insists on having specifications, which are the very best form of documentation; the Program Answer doesn't.

**Halting Problem**

The Halting Problem is widely considered to be a foundational result in computer science. Here is a modern presentation of it. We have the header and specification of function \texttt{halts}, but not its body. Then we have procedure \texttt{diag} in its entirety, and \texttt{diag} calls \texttt{halts}. This is exactly the situation we had with function \texttt{binexp} and procedure \texttt{toobig}. Usually, \texttt{halts} gives two possible answers: 'yes' or 'no'; for the purpose of this essay, I have added a third: 'not applicable'.

\begin{verbatim}
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) \neq 'yes' }
begin
  if halts (s, s) = 'yes' then diag (s)
end
\end{verbatim}

We assume there is a dictionary of function and procedure definitions that is accessible to \texttt{halts}, so that the call \texttt{halts ('diag', 'diag')} allows \texttt{halts} to look up 'diag', and subsequently 'halts', in the dictionary, and retrieve their texts for analysis. Here is the “textbook proof” that
halts is incomputable.

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

This “textbook proof” begins with the computability assumption: that the body of halts can be written, and has been written. The assumption is necessary for advocates of the Program Answer to say that diag is a Pascal procedure, and so rule out 'not applicable' as the result of halts('diag', 'diag'). If we suppose the result is 'yes', then we see from the body of diag that execution of diag('diag') is nonterminating, so the result should be 'no'. If we suppose the result is 'no', then we see from the body of diag that execution of diag('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 halts, but since the body is unavailable, they are compelled to base their reasoning on the specification of halts as advocated in the Specification Answer.

Advocates of the Specification Answer do not need the computability assumption. According to them, diag is a Pascal procedure even though the body of halts has not been written. What does the specification of halts say the result of halts('diag', 'diag') should be? The Specification Answer eliminates 'not applicable'. As before, if we suppose the result is 'yes', then we see from the body of diag that execution of diag('diag') is nonterminating, so the result should be 'no'; if we suppose the result is 'no', then we see from the body of diag that execution of diag('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 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 halts cannot be written, but for different reasons. According to advocates of the Program Answer, 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, halts has an inconsistent specification, and the question of computability does not arise.

**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 halts and the parameter of 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 halts allows 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 halts to apply specifically to 'diag'. Here is the result.

```pascal
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 WhatTwistPrints cannot have been written according to its specification; WhatTwistPrints is incomputable.

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

The following example is similar to the previous example.

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

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

To advocates of the Program Answer, \texttt{straight} is not a Pascal procedure because the body of \texttt{WhatStraightPrints} has not been written. Therefore \texttt{WhatStraightPrints} should return 'not applicable', and its body is easily written: \texttt{WhatStraightPrints} := '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 \texttt{WhatStraightPrints} to return 'yes' or to return 'no'.

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

\begin{verbatim}
function halts: string;
{ return 'yes' if \texttt{diag} is an executable procedure whose execution terminates; }
{ return 'no' if \texttt{diag} is an executable procedure whose execution does not terminate; }
{ return 'not applicable' if \texttt{diag} is not an executable procedure }
\end{verbatim}

\textbf{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 \texttt{halts} applies to. In the simplified version, we might apply \texttt{halts} to executable procedures.

\begin{verbatim}
function halts: string;
{ return 'yes' if \texttt{diag} is an executable procedure whose execution terminates; }
{ return 'no' if \texttt{diag} is an executable procedure whose execution does not terminate; }
{ return 'not applicable' if \texttt{diag} is not an executable procedure }
\end{verbatim}
procedure diag; { execution terminates if and only if halts ≠ 'yes' }
begin
  if halts = 'yes' then diag
end

Since we have specified but not programmed halts, diag is an intentional procedure, but not an executable procedure. So function halts should, according to its specification, return 'not applicable', and the execution of diag should, according to its intention, terminate.

To make diag an executable procedure we need to program halts. Suppose we program it according to this latest specification.

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

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

Suppose we make halts more specific the other way, applying it to intentional procedures.

function halts: string;
{ return 'yes' if diag is an intentional procedure whose execution terminates; }
{ return 'no' if diag is an intentional procedure whose execution does not terminate; }
{ return 'not applicable' if diag is not an intentional procedure }

Even without programming halts, 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 halts in this section, the same arguments and same conclusions apply to the two-parameter version of 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 halts is computable and has been programmed does not give us the program; so we have no meaning for halts, and cannot say whether execution of 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 halts is inconsistent. Either way, the “textbook proof” does not show us a (consistently specified) mathematical function that is incomputable.

other papers on halting