What is a Procedure?

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

Question and Answers

What is the meaning of a procedure? This question 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, designed at ETH Zürich 40 years ago by my academic grandfather, Niklaus Wirth. I think it is an appropriate choice for celebrating the history of software engineering at ETH. But the points I make apply to any programming language.

Here are two Pascal procedures.

```
procedure A; { this procedure prints 'A' }
begin
    print ('B')
end;

procedure AA; { this procedure prints 'AA' }
begin
    A; A
end
```

What is the meaning of procedure *A* ? Is it a procedure that prints 'A' as its specification (the comment) says, or is it a procedure that prints 'B' as its implementation (the body) says? Perhaps I should instead ask: Is *A* implemented correctly? Clearly it is not, though we cannot say whether the specification or the implementation is at fault. Is *AA* implemented correctly? This time I want to say yes: its specification says it prints 'AA' , and to do so it twice calls a procedure whose specification says it prints 'A' . The error is in procedure *A* , not in procedure *AA* .

Now consider this example.

```
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 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 two 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 principle of software engineering; 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.

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.

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 *binexp* and *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.

Theory of Programming

In my theory of programming (sometimes called “predicative programming”, sometimes called UTP), we do not specify programs; we specify computation, or computer behavior. The nonlocal (free) variables of the specification represent whatever we wish to observe about a computation (initial state, final state, all states, interactions, execution time, space occupied). Observing a computation provides values for those variables. A specification is a binary (i.e. boolean) expression because, when you instantiate its variables with values obtained from observing a computation, there are two possible outcomes: either the computation satisfies the specification, or it doesn't. If you write anything other than a binary expression as a specification (for example, a pair of predicates), you must say what it means for a computation

to satisfy a specification, and to do that formally you must write a binary expression anyway.

A program is an implemented specification. It is a specification of computer behavior that you can give to a computer and get the specified behavior. I also refer to any statement in a program, or any sequence or structure of statements, as a program. Since a program is a specification, and a specification is a binary expression, therefore a program is a binary expression. For example, if the state (or program) variables are x and y, then the program $x := x+y$ is the binary expression $x'=x+y \wedge y'=y$ where unprimed variables represent the values of the state variables before execution of the assignment, and primed variables represent the values of the state variables after execution of the assignment.

$$x := x+y \;=\; x'=x+y \wedge y'=y$$

Similarly for a conditional program

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q \;=\; b \wedge P \vee \neg b \wedge Q$$
$$\;=\; (b \Rightarrow P) \wedge (\neg b \Rightarrow Q)$$

Sequential composition is a little more complicated

$$P;Q \;=\; \exists x'', y'' \cdot (\text{in } P \text{ substitute } x'', y'' \text{ for } x', y') \wedge (\text{in } Q \text{ substitute } x'', y'' \text{ for } x, y)$$

but fortunately we can prove the Substitution Law, which doesn't involve quantification:

$$x := e; P \;=\; (\text{for } x \text{ substitute } e \text{ in } P)$$

For example,

$$x := x+y;\; x+y < 5 \;=\; (x+y)+y < 5$$

To say "specification P refines specification Q" means that all behavior satisfying P also satisfies Q. Formally, that's just implication: $P \Rightarrow Q$. For example,

$$x' < x \;\Leftarrow\; x := x-1$$

says that specification $x' < x$ is implied by, or refined by, or implemented by program $x := x-1$, and it is trivial to prove. As a second example,

$$x' \le x \;\Leftarrow\; \textbf{if } x>0 \textbf{ then } x' < x$$

From those two examples, we conclude

$$x' \le x \;\Leftarrow\; \textbf{if } x>0 \textbf{ then } x := x-1$$

and that's how stepwise refinement works.

A complete explanation can be found in the book *a Practical Theory of Programming* and the online course Formal Methods of Software Design.

Loop Semantics

Equating programs with binary expressions gives meaning to straight-line and branching programs; but how shall we give meaning to loops? There are two answers: the Program Answer and the Specification Answer. The Program Answer is the standard answer: by a construction axiom and an induction axiom, also known as a least-fixed-point.

while-construction (fixed-point) axiom:

$$\textbf{while } a \textbf{ do } B \;=\; \textbf{if } a \textbf{ then begin } B; \textbf{ while } a \textbf{ do } B \textbf{ end}$$

while-induction (least-fixed-point) axiom (σ is the prestate and σ' is the poststate):

$$(\forall \sigma, \sigma' \cdot\; S = \textbf{if } a \textbf{ then begin } B; S \textbf{ end}) \Rightarrow (\forall \sigma, \sigma' \cdot\; S \Rightarrow \textbf{while } a \textbf{ do } B)$$

Construction says that a **while**-loop equals its first unrolling. Induction says that of all specifications satisfying the construction axiom, the **while**-loop is the weakest (least deterministic). Least-fixed-points are difficult to use for program verification, so the verification community has gone part way toward the Specification Answer, by using invariants.

The Specification Answer requires an implementable specification. Specification S is implementable if $\forall \sigma \cdot \exists \sigma' \cdot S$. The refinement

S $\Leftarrow$ **while** a **do** B

means, or is an alternate notation for

S $\Leftarrow$ **if** a **then begin** B; S **end**

In this unrolling, following the body B, we do not have the **while**-loop, but rather the specification S. Any refinement is a sort of small procedure, and this refinement is a small procedure with a recursive call, just like

procedure S; **begin if** a **then begin** B; S **end end**

and its execution is just like

S: **if** a **then begin** B; **goto** S **end**

For the recursive call, according to the Specification Answer, we take the meaning of the procedure to be the specification. And so also for loops, with the same benefits. Here is an example in one integer state variable x. To prove

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ **while** $x{>}0$ **do** $x{:=}\,x{-}1$

prove instead

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ **if** $x{>}0$ **then begin** $x{:=}\,x{-}1;\ x{\geq}0 \Rightarrow x'{=}0$ **end**

That means proving

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ $x{>}0 \wedge (x{:=}\,x{-}1;\ x{\geq}0 \Rightarrow x'{=}0) \vee x{\leq}0 \wedge x'{=}x$

Inside the parentheses we use the Substitution Law, and get

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ $x{>}0 \wedge (x{-}1{\geq}0 \Rightarrow x'{=}0) \vee x{\leq}0 \wedge x'{=}x$

Now we have no more programming notations; the proof is just binary and number laws.

For proof purposes, the Specification Answer is much easier to use than the Program Answer. But the biggest advantage of the Specification Answer is during programming. We start with a specification, for example $x{\geq}0 \Rightarrow x'{=}0$, and we refine it. The obvious refinement is

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ $x{:=}\,0$

but to obtain the same computation as in the preceding paragraph, we can refine it as

$x{\geq}0 \Rightarrow x'{=}0$ $\Leftarrow$ **if** $x{>}0$ **then** $x{>}0 \Rightarrow x'{=}0$ **else** $x{=}0 \Rightarrow x'{=}0$

Now we have two more specifications to refine.

$x{>}0 \Rightarrow x'{=}0$ $\Leftarrow$ $x{:=}\,x{-}1;\ x{\geq}0 \Rightarrow x'{=}0$

$x{=}0 \Rightarrow x'{=}0$ $\Leftarrow$ **begin end**

And we're done. We never refine to a loop construct, so we never need any fixed-points, nor any proof rules concerning loops, nor any invariants. But we form loops by reusing specifications.

For execution time, we just add a time variable t, and increase it wherever we need to account for the passage of time. To count iterations, we place $t{:=}\,t{+}1$ inside the loop. And we can write specifications about execution time. For example,

$x{\geq}0 \Rightarrow t'{=}t{+}x$ $\Leftarrow$ **while** $x{\neq}0$ **do begin** $x{:=}\,x{-}1;\ t{:=}\,t{+}1$ **end**

which means, according to the Specification Answer,

$x{\geq}0 \Rightarrow x'{=}t{+}x$ $\Leftarrow$ **if** $x{\neq}0$ **then begin** $x{:=}\,x{-}1;\ t{:=}\,t{+}1;\ x{\geq}0 \Rightarrow x'{=}t{+}x$ **end**

That means proving

$x{\geq}0 \Rightarrow x'{=}t{+}x$ $\Leftarrow$ $x{\neq}0 \wedge (x{:=}\,x{-}1;\ t{:=}\,t{+}1;\ x{\geq}0 \Rightarrow x'{=}t{+}x) \vee x{=}0 \wedge x'{=}x \wedge t'{=}t$

Inside the parentheses we use the Substitution Law twice, and get

$x{\geq}0 \Rightarrow x'{=}t{+}x$ $\Leftarrow$ $x{\neq}0 \wedge (x{-}1{\geq}0 \Rightarrow x'{=}t{+}1{+}x{-}1) \vee x{=}0 \wedge x'{=}x \wedge t'{=}t$

Now we have no more programming notations; the proof is just binary and number laws.

We can just as easily prove

$x{<}0 \Rightarrow t'{=}\infty$ $\Leftarrow$ **while** $x{\neq}0$ **do begin** $x{:=}\,x{-}1;\ t{:=}\,t{+}1$ **end**

which means

$x{<}0 \Rightarrow t'{=}\infty$ $\Leftarrow$ **if** $x{\neq}0$ **then begin** $x{:=}\,x{-}1;\ t{:=}\,t{+}1;\ x{<}0 \Rightarrow t'{=}\infty$ **end**

That means proving

$x{<}0 \Rightarrow t'{=}\infty$ $\Leftarrow$ $x{\neq}0 \wedge (x{:=}\,x{-}1;\ t{:=}\,t{+}1;\ x{<}0 \Rightarrow t'{=}\infty) \vee x{=}0 \wedge x'{=}x \wedge t'{=}t$

Inside the parentheses we use the Substitution Law twice, and get

$x{<}0 \Rightarrow t'{=}\infty$ $\Leftarrow$ $x{\neq}0 \wedge (x{-}1{<}0 \Rightarrow t'{=}\infty) \vee x{=}0 \wedge x'{=}x \wedge t'{=}t$

Now we have no more programming notations; the proof is just binary and number laws.

The Specification Answer is a general recipe for all kinds of loops. Departing momentarily from Pascal, here is a more complicated structure using 1- and 2-level exits.

```
loop
        A;
        exit 1 when b;
        C;
        loop
                D;
                exit 2 when e;
                F;
                exit 1 when g;
                H
        end;
        I
end
```

The Specification Answer requires a specification for each loop. If they are P and Q for these two loops, then what we must prove is

$P \Leftarrow A$; **if not** b **then begin** C; Q **end**
$Q \Leftarrow D$; **if not** e **then begin** F; **if not** g **then begin** H; Q **end else begin** I; P **end end**

Note that specifications P and Q are used, rather than the loop constructs, on the right sides of these reverse implications; that's the Specification Answer.

The literature on loop semantics is large, and entirely according to the Program Answer. But the Specification Answer has advantages: it makes proofs much easier, and program derivation much much easier. If we include time, we have more than total correctness, without any least-fixed-points or invariants.

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 *halts*, but not its body. Then we have procedure *twist* in its entirety, and *twist* calls *halts*. This is exactly the situation we had with function *binexp* and procedure *toobig*. Usually, *halts* gives two possible answers: 'stops' or 'loops'; for the purpose of this essay, I have added a third: 'not applicable'.

```
function halts (p, i: string): string;
{ return 'stops' if p represents a Pascal procedure with one string input parameter }
{       whose execution terminates when given input i ; }
{ return 'loops' 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 twist (s: string);  { execution terminates if and only if halts (s, s) ≠ 'stops' }
begin
        if halts (s, s) = 'stops' then twist (s)
end
```

We assume there is a dictionary of function and procedure definitions that is accessible to *halts*, so that the call *halts* ('*twist*', '*twist*') allows *halts* to look up '*twist*', 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 *twist* ('*twist*') terminate? If it terminates, then *halts* ('*twist*', '*twist*') returns 'stops' according to its specification, and so we see from the body of *twist* that execution of *twist* ('*twist*') does not terminate. If it does not terminate, then *halts* ('*twist*', '*twist*') returns 'loops', and so execution of *twist* ('*twist*') 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 *twist* is a Pascal procedure, and so rule out 'not applicable' as the result of *halts* ('*twist*', '*twist*'). If we suppose the result is 'stops', then we see from the body of *twist* that execution of *twist* ('*twist*') is nonterminating, so the result should be 'loops'. If we suppose the result is 'loops', then we see from the body of *twist* that execution of *twist* ('*twist*') is terminating, so the result should be 'stops'. 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, contrary to the Program Answer.

Advocates of the Specification Answer do not need the computability assumption. According to them, *twist* 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* ('*twist*', '*twist*') should be? The Specification Answer eliminates 'not applicable'. As before, if we suppose the result is 'stops', then we see from the body of *twist* that execution of *twist* ('*twist*') is nonterminating, so the result should be 'loops'; if we suppose the result is 'loops', then we see from the body of *twist* that execution of *twist* ('*twist*') is terminating, so the result should be 'stops'. 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 *twist* 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 '*twist*'. Here is the result.

function *halts*: **string**;
{ return 'stops' if *twist* is a Pascal procedure whose execution terminates; }
{ return 'loops' if *twist* is a Pascal procedure whose execution does not terminate; }
{ return 'not applicable' if *twist* is not a Pascal procedure }

procedure *twist*; { execution terminates if and only if *halts* ≠ 'stops' }
begin
    if *halts* = 'stops' **then** *twist*
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 *twist* terminate? If it terminates, then *halts* returns 'stops' according to its specification, and so we see from the body of *twist* that execution of *twist* does not terminate. If it does not terminate, then *halts* returns 'loops' , and so execution of *twist* 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*:= 'stops' or *halts*:= 'loops' or *halts*:= 'not applicable' . The problem is to decide which of those three it is. If the body of *halts* is *halts*:= 'stops' , we see from the body of *twist* that it should be *halts*:= 'loops' . If the body of *halts* is *halts*:= 'loops' , we see from the body of *twist* that it should be *halts*:='stops' . If the body of *halts* is *halts*:= 'not applicable' , advocates of both the Program Answer and the Specification Answer agree that *twist* 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 'A' if *twist* is a Pascal procedure whose execution prints 'A' ; }
{ return 'B' if *twist* is a Pascal procedure whose execution does not print 'A' ; }
{ return 'not applicable' if *twist* is not a Pascal procedure }

procedure *twist*; { if *WhatTwistPrints* = 'A' then print 'B' ; otherwise print 'A' }
begin
    if *WhatTwistPrints* = 'A' **then** *print* ('B') **else** *print* ('A')
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 'A' or 'B' ? If it prints 'A' , then *WhatTwistPrints* returns 'A' according to its specification, and so we see from the body of *twist* that execution of *twist* prints 'B' . If it prints 'B' , then *WhatTwistPrints* returns 'B' according to its specification, and so we see from the body of *twist* that execution of *twist* prints 'A' . 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*:= 'A' or *WhatTwistPrints*:= 'B' 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 *twist* , which calls *WhatTwistPrints*) in a self-contradictory manner. That's exactly what the *halts* specification does: it refers to itself (indirectly by saying that *halts* applies to all procedures including *twist* , which calls *halts*) in a self-contradictory manner.

The following example is similar to the previous example.

function *WhatStraightPrints*: **string**;
{ return 'A' if *straight* is a Pascal procedure whose execution prints 'A' ; }
{ return 'B' if *straight* is a Pascal procedure whose execution does not print 'A' ; }
{ return 'not applicable' if *straight* is not a Pascal procedure }

procedure *straight*; { if *WhatStraightPrints* = 'A' then print 'A' ; otherwise print 'B' }
begin
    if *WhatStraightPrints* = 'A' **then** *print* ('A') **else** *print* ('B')
end

To advocates of the Program Answer, *straight* is not a Pascal procedure because the body of *WhatStraightPrints* has not been written. Therefore *WhatStraightPrints* should return 'not applicable' , and its body is easily written: *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 *WhatStraightPrints* to return 'A' or to return 'B' .

The halting function *halts* has a similar dilemma when applied to

procedure *straight* (*s*: **string**); { execution terminates if and only if *halts* (*s*, *s*) = 'stops' }
begin
    if *halts* (*s*, *s*) **not**= 'stops' **then** *straight* (*s*)
end

The specification of *halts* may sound all right, but we are forced by the examples to admit that the specification is not as it sounds. In at least one instance (*twist*), the *halts* specification is overdetermined (inconsistent), and in at least one instance (*straight*), the *halts* specification is underdetermined.

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 *stop* (*s*: **string**); **begin end**

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

is easy. Perhaps we can ask for a Pascal function *halts1* to compute the halting status of all Pascal procedures that do not refer to *halts1* , neither directly nor indirectly. Here is its header, specification, and a start on its implementation.

```
function halts1 (p, i: string): string;
{ return 'stops' 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 'loops' 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 halts directly or indirectly)
            then halts1:= 'maybe'
            else (return halting status of p , either 'stops' or 'loops' )
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

```
procedure twist1 (s: string);  { execution terminates if and only if halts1 (s, s) ≠ 'stops' }
begin
        if halts1 (s, s) = 'stops' then twist1 (s)
end
```

Procedure *twist1* refers to *halts1* by calling it, so *halts1* ('*twist1*', '*twist1*') = 'maybe' , and execution of *twist1* ('*twist1*') is terminating.

Calling is one kind of referring, but not the only kind. In the specification of *halts1* , the name *halts1* appears, and also in the body. These are self-references, whether or not *halts1* calls itself. We exempt *halts1* from having to determine the halting status of procedures containing any form of reference to *halts1* ; the result is 'maybe' . We might try to circumvent the limitation by writing another function *halts2* that is identical to *halts1* but renamed (including in the specification, the return statements, and any recursive calls).

```
function halts2 (p, i: string): string;
{ return 'stops' 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 'loops' 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  'stops'  or  'loops' )
end
```

Of course, *halts2* has its own nemesis:

```
procedure twist2 (s: string);
begin
    if halts2 (s, s) = 'stops' then twist2 (s)
end
```

The point is that *halts2* can determine halting for procedures that *halts1* cannot, and *halts1* can determine halting for procedures that *halts2* cannot. For example,

halts1 ('*twist1*', '*twist1*') = 'maybe' because *twist1* calls *halts1*
halts2 ('*twist1*', '*twist1*') = 'stops' because execution of *twist1* ('*twist1*') terminates
halts2 ('*twist2*', '*twist2*') = 'maybe' because *twist2* calls *halts2*
halts1 ('*twist2*', '*twist2*') = 'stops' because execution of *twist2* ('*twist2*') terminates

But there are procedures that refer to both *halts1* and *halts2* , for which both *halts1* and *halts2* say 'maybe' . The most interesting point is this: even though *halts1* and *halts2* are identical except for renaming, they produce different results when given the same input, according to their specifications, as the above four examples show.

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 all identifiers must end with an underscore. Pascal_ is neither more nor less powerful than Pascal: they are both Turing-Machine-equivalent. 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  'stops'  if  p_  represents a Pascal procedure with one string input parameter }
{       whose execution terminates when given input  i_ ; }
{ return  'loops'  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  'stops'  or  'loops' )
end
```

If it is possible to write a Pascal function to compute the halting status of all Pascal procedures that do not refer to this 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. If we can write *halts_* in Pascal_, then we can easily obtain a Pascal function *halts* just by deleting the underscore from the Pascal_ identifiers. We thus obtain a Pascal function with the same functionality. 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. Even though *halts_* fulfills the specification, telling the halting status of all Pascal procedures, and *halts* is obtained from *halts_* by renaming, *halts* does not fulfill the specification. The next two sections explain why.

How Do We Translate?

If I say "My name is Eric Hehner.", I am telling the truth. If Margaret Jackson says exactly the same words, she is lying. There is a self-reference ("My"), and the truth of that sentence depends on who says it.

Here is a Pascal_ procedure that prints its own name.

procedure *A_*; { this procedure prints its own name }
begin *print_* ('*A_*') **end**

How do we translate this procedure to Pascal? There are two answers, and here is the Program Answer.

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

Ignoring the specification, which is just a comment, the Program Answer is a procedure that performs the same action(s). The original and the translation have the same output, but clearly this translation does not preserve the intention. The Pascal_ procedure *A_* meets its specification; the Pascal translation *A* does not.

The Specification Answer is

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

This translation preserves the intention, meets the same specification, but it does not have the same output. Translating from *halts_* to *halts* has the same problem. We cannot preserve the intention because the specification at the head of *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 from the ends of identifiers, 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 *halts_* in Pascal_ for all of Pascal, we could feed the text of *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 *halts_* is just like the translation of *halts_* into Pascal by deleting underscores. The interpreter interpreting *halts_* can be called by another Pascal program; *halts_* cannot be called by a Pascal program. That fact materially affects their behavior. Pascal_ program *halts_* can be applied to a Pascal procedure *d* that calls the interpreter interpreting *halts_* applied to *d* , and it will produce the right answer. But the interpreter interpreting *halts_* applied to *d* calls the interpreter interpreting *halts_* applied to *d* , and execution will not terminate.

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 for Russellville is a person who shaves all and only those men in Russellville who do not shave themselves. There is a barber for Russellville; his name is Bertrand_ and he lives in the neighboring town of Russellville_. Without any difficulty, he satisfies the specification of barber for Russellville.

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_ (translation by the Program Answer), 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 for Russellville, Bertrand has to shave himself if and only if he does not shave himself. A specification that is perfectly consistent and possible for someone outside town becomes inconsistent and impossible 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 *halts_* , the halting specification has no self-reference; for *halts* , the halting specification has a self-reference (indirectly through *twist* and other procedures that call *halts*).

Conclusion

The question “What is the meaning of a procedure?” has at least two defensible answers, which I have called the “Program Answer” and the “Specification Answer”. The Program Answer says that the meaning of a procedure (or any other unit of program) is its body; the Specification Answer says that the meaning of a procedure is its specification. These two answers have quite different consequences throughout computer science.

To find the meaning of a procedure that contains calls to other procedures, the Program Answer requires the bodies of those other procedures; and if they contain calls, then also the bodies of those procedures; and so on, transitively. For that reason, the Program Answer does not scale up; large software must be analyzed as a whole.

The Specification Answer gives the meaning of a procedure directly, without looking at its body. But this answer raises a different question: does the body satisfy the specification? If the body contains calls to other procedures, only the specifications of those other procedures are used as the meanings of the calls. There is no transitive closure. So the Specification Answer does scale up.

The Program Answer can be used to verify whether some software has a certain property, giving the answer "yes" or "no". The Specification Answer can do more: if there is an error, it isolates the error to a specific procedure.

The meaning of loops, and the methods for verifying loops, have the same two answers as procedures. The Program Answer uses least-fixed-points as the meaning of loops, but they are difficult to find, difficult to use in verification, and useless for program construction. The Specification Answer says that the meaning of a loop is a specification, and verification is a single unrolling. The Specification Answer enables programming by refinement, without invariants.

For translation between languages, the Program Answer says that behavior should be preserved, and the Specification Answer says that intention should be preserved. Surprisingly, the two answers give different results. Preserving behavior may not preserve intention. A specification that is consistent and satisfiable in one language may be inconsistent and unsatisfiable in another.

In the Halting Problem, the Program Answer requires the computability assumption; *halts* must have a body to be a function with a meaning, and for *twist* to be a procedure whose execution can be determined. But the assumption that *halts* has a body does not give us the body, so we still have no meaning for *halts*, and cannot reason about the execution of *twist*. The Specification Answer says that we know the meaning of *halts* from its specification, and we can reason about the execution of *twist*. We don't need the computability assumption, and we reach the conclusion that the specification of *halts* is inconsistent.

The standard proofs that halting is incomputable prove only that it is inconsistent to ask for a halting function for a Turing-Machine-equivalent language in which that same halting function is callable. 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. In any case, we do not yet have a proof that it is impossible.

I hope that the Specification Answer will become the standard for software engineering.

References

E.C.R.Hehner: *a Practical Theory of Programming* (book)

E.C.R.Hehner: the Halting Problem (papers)

E.C.R.Hehner: Formal Methods of Software Design (online course)