Specifications, Programs, and Total Correctness

Eric C.R. Hehner
University of Toronto

Abstract

This paper argues the following positions: that a formal specification is a boolean expression, that a program is a specification, and that total correctness is a poor choice of semantics.

Keywords: specification; program; total correctness

I have three opinions to present: first, that a formal specification is a boolean expression; second, that a program is a formal specification; and third, that total correctness is a mistake. I am discussing what are sometimes called "formal methods" of programming, but I am not debating the usefulness of formal methods to programmers; that debate has been well aired here and elsewhere. I am debating the direction formal methods research has taken. My concern is first to find a satisfactory theoretical foundation for programming, and ultimately to create useful tools to aid programmers.

Formal methods researchers have invented a fascinating variety of formalisms, and have probed into their far corners, finding some esoteric things, such as havoc and angelic nondeterminism. Although this work is theoretically interesting, I believe we are now in a position to say in a simple and clear way, independent of all special-purpose notations, what constitutes a formal specification, and what is the relationship between specifications and programs.

Opinion: a formal specification is a boolean expression.

By "formal specification" I mean some kind of mathematical expression. I shall argue that the best kind of expression to use as a formal specification is a boolean expression rather than a set or predicate or pair of predicates or predicate transformer or any other kind of mathematical expression. For the purpose of arguing this opinion, it does not matter whether we are specifying computations, cars, or anything else.

First a word on my terminology. By "boolean expression" I mean an expression that evaluates to a boolean when values are provided for its (global or free) variables. I do not mean to be restrictive in the allowed operators; quantifiers are welcome. For example,

$$x{>}y \wedge (\exists z{\cdot}\ y = f(z))$$

is a boolean expression. I allow variables of any type, and any operators and functions. Terminology and notations that pertain to an application are encouraged. Sometimes it is helpful to invent a notation specifically for use in one particular expression.

By “predicate” I mean a function that results in a boolean when applied to values in its domain. For example,

$\lambda y \cdot y > 5$

I still say “predicate” when there are global variables, as in

$\lambda y \cdot y > x$

By “relation” I mean a function that results in a predicate when applied to values in its domain. For example,

$\lambda x \cdot \lambda y \cdot y > x$

which may also be written as a function of two variables

$\lambda x, y \cdot y > x$

I'm not concerned here whether variables are strongly-, weakly-, or un-typed; choose your favorite. And I'm not concerned here with definedness, completeness, computability, or order of evaluation. And I don't distinguish between an expression that evaluates to a boolean and a proposition that becomes true or false when values are provided for the variables. That's my terminology; now for my arguments.

It is the job of a specification to distinguish those things that satisfy it from those that don't. In the scientific tradition, we use variables for quantities that are of interest, and observation of something provides us with values for the variables. For example, we may decide that the prestate and poststate of memory are of interest. We may decide that communications during a computation are of interest. We may decide that the start time and stop time of a computation are of interest. I make no case here for any particular choice of quantities of interest. But I insist that when we have a specification and an observation, we have to be able to put them together to find out whether the observation satisfies the specification.

One might suppose that any type of mathematical expression can be used as a specification: whatever works. A boolean expression certainly works, since it provides one of two possible outcomes for each observation. Functions with boolean result work also; if there's only one variable, it's a predicate; if there are two variables, it's a relation. However many variables (quantities of interest) there may be, we can write a specification as a function, and apply it to the values provided by an observation to get one of two possible outcomes. The difference between a boolean expression and a function is language level. The function

$\lambda x, y \cdot y > x$

is equivalent to

$\lambda y, z \cdot z > y$

The names x, y, and z are irrelevant; only their positions are relevant when we apply the function. In other words, we are using addresses to identify our quantities of interest. We apply

$(\lambda x, y \cdot y > x)\ 2\ 3$

to find out that the ordered pair $2\ 3$ satisfies the specification. If instead we use the boolean expression $y>x$ as specification, we refer to the quantities of interest by variable names, and we substitute

$x := 2;\ y := 3;\ y > x$

(for x substitute 2 and for y substitute 3 in $y>x$) to find out that $x=2$ and $y=3$ satisfies the specification. The boolean expression is both simpler and higher level than the function.

A set also works, but suffers exactly the same criticisms as a function. The set of pairs

$\{x, y \mid y > x\}$

(which many people call a relation) is also syntactically more complicated than the boolean expression $y>x$, and the extra syntax serves to bind the names x and y, leaving only their positions (addresses) visible, and so lowers the language level.

The best known form of specifying a computation is a pair of boolean expressions: the pre- and postcondition. I'll borrow the notation $[P, Q]$ from [16] for a specification consisting of precondition P and postcondition Q. In this form, the assumption is that the initial and final values of the program variables are the only quantities of interest. In early work (1969-1980) the same identifiers were used for initial values and final values, so to say that x is increased, one had to introduce an extra variable and say awkwardly

$\forall X \cdot [x = X, x > X]$

Since we can observe both the initial value and final value, we should have a notation for each, as in VDM [13], Z [17], and sometimes the Refinement Calculus [0, 16]. I'll write x and x' as in Z, so to specify that x is increased we write more simply

$[true, x' > x]$

The specification $[P, Q]$ means, informally: if P is satisfied, then the computation terminates and satisfies Q. So termination is also considered observable, and we ought to introduce a boolean variable for it, say h (for "halt"). Then we can write the specification as a single boolean expression: $P \Rightarrow h \wedge Q$. To say we want termination with an increase in x we say $h \wedge x' > x$. To say we want nontermination we write $\neg h$. If we just want partial correctness, it's $P \wedge h \Rightarrow Q$. To make sequential composition simpler, we might like to replace h with two boolean variables f and f', where f means that computation begins at a finite time (not sequentially following an infinite loop), and f' means that computation ends at a finite time (terminates); then $[P, Q]$ becomes $f \wedge P \Rightarrow f' \wedge Q$. This last approach is the one taken in the

forthcoming book by Hoare and He [12], and suggested earlier in [6].

If we write anything other than a boolean expression (or predicate) as a specification, we must say what it means for an observation to satisfy the specification, and to do that we must write a boolean expression anyway. For example, when is the specification $[x>0, x'=2]$ satisfied by the observation $x=a$ and $x'=b$? The answer is that it is satisfied if and only if $a>0 \Rightarrow b=2$. We really might as well have written the specification as $x>0 \Rightarrow x'=2$ in the first place.

Refinement is the central idea in programming from specifications. Informally, specification S is refined by specification T if all computations satisfying T also satisfy S . Using a single boolean expression for each specification, that's just (reverse) implication:

$S \Leftarrow T$

Using a pair of boolean expressions for each specification, refinement is more complicated: specification $[P, Q]$ is refined by specification $[R, S]$ when

$P \Rightarrow R \wedge (Q \Leftarrow S)$

Or we can define refinement by giving a large number of laws; in effect, we have to learn two sets of laws, boolean laws and refinement laws, when one set will do. For example, the law "weaken precondition"

If $A \Rightarrow B$ then $[A, R]$ is refined by $[B, R]$

is no more than the boolean identity

$(A \Rightarrow B) \;\Rightarrow\; (A \Rightarrow B \wedge (R \Leftarrow R))$

And the law "strengthen postcondition"

If $R \Leftarrow S$ then $[A, R]$ is refined by $[A, S]$

is just the identity

$(R \Leftarrow S) \;\Rightarrow\; (A \Rightarrow A \wedge (R \Leftarrow S))$

Z uses a single boolean expression as specification, but in a strange way. It seems reasonable to me that weaker (truer) boolean expressions should be satisfied by more behaviors, and stronger (falser) boolean expressions should be satisfied by fewer behaviors, and at the extremes, *true* should be satisfied by any behavior and *false* by none. But in Z, *true* is satisfied only by terminating behaviors and *false* is satisfied by all nonterminating behaviors. The advantage is that the class of specifications fits the class of computations: there are no unsatisfiable specifications. To say that *true* is not satisfied by something and that *false* is satisfied by something makes refinement rather complicated. In Z, specification S is refined by specification T when

$\forall s \cdot ((\exists s' \cdot S) \Rightarrow (\exists s' \cdot T) \wedge (\forall s' \cdot (S \Leftarrow T)))$

where s is the prestate and s' is the poststate. Since refinement is what we must prove at each step in programming, it is best to make refinement as simple as possible.

It is very important that we be able to write a specification by parts; we cannot possibly write realistic specifications all at once. So we write several partial specifications, and we want behavior satisfying all of them. If we are using a pair of boolean expressions for each partial specification, then it is not obvious how to put them together; putting $[P, Q]$ together with $[R, S]$ yields

$$[P \vee R, (P \Rightarrow Q) \wedge (R \Rightarrow S)]$$

For a discussion of the problem, see [15]. In Z, the problem is even worse; you can put S and T together if and only if

$$(\exists s' \cdot S \wedge T) \vee \neg(\exists s' \cdot S) \vee \neg(\exists s' \cdot T)$$

and you get

$$((\exists s' \cdot S \wedge T) \Rightarrow S \wedge T) \wedge (\neg(\exists s' \cdot S) \Rightarrow T) \wedge (\neg(\exists s' \cdot T) \Rightarrow S)$$

The PhD thesis of Frappier [3] is largely an attempt to solve the problem of specification by parts in a Z-like setting. But there shouldn't be a problem; if you write a specification as a boolean expression in the way I am advocating, and you want both behavior S and behavior T, then you want behavior $S \wedge T$; you put specifications together with ordinary conjunction.

One advantage sometimes claimed for predicate transformers over boolean expressions is the ability to express angelic nondeterminism, in which choices are made by an all-knowing and benevolent angel. No-one claims that angelic nondeterminism is observable or physical, but in a very nice, enjoyable paper [18], Ward and Hayes claim that angelic nondeterminism is a useful calculational device. The use is in obtaining backtracking computations. But angelic nondeterminism is not necessary to obtain backtracking computations. They can be obtained more simply using boolean expressions. For example, we might want

$$x := 0 \textbf{ or } x := 1;\ \textbf{ensure } x=1$$

where "**or**" says do either one, but then "**ensure**" says which one it had to be, so if the wrong one was chosen, go back and choose the other. As a boolean expression in one state variable, "**ensure** $x=1$" is $x=1 \wedge x'=x$, so the previous specification becomes

$$x'=0 \vee x'=1;\ x=1 \wedge x'=x$$

which is equivalent to

$$\exists x'' \cdot (x''=0 \vee x''=1) \wedge x''=1 \wedge x'=x''$$

which can be simplified to

$$x'=1$$

Although $x=1 \wedge x'=x$ is unimplementable by itself (in other terminology, "infeasible" [16], "miraculous" [0], or "magic" [1]), in combination with disjunction we get something that is implementable by backtracking. No predicate transformers are necessary.

Since I am not being restrictive concerning the types and operators appearing within a boolean expression, no other kind of expression is more expressive or more "powerful". If you really

want or need to use predicate transformers or anything else, you can do so within a boolean expression. I do not argue against the use of sets, functions, or any other kind of mathematical expression; they have many good uses, even within specifications. I argue only that the specification as a whole should be a boolean expression. There is, however, another side to the argument. Observations can be represented either by adding a variable for each observable quantity (as advocated here), or by adding structure to specifications, for example by using pairs or tuples of boolean expressions, or by using higher-order functions as specifications. We have already seen an example of this in the representation of termination. For another example, the Refinement Calculus [1] presents specifications as contracts between two agents (an angel and a demon). We could accommodate this view of specification while still maintaining our view that a specification is a boolean expression by adding a boolean variable to record the current agent. But the Refinement Calculus uses the two directions of the lattice of predicate transformers to represent the two agents, and thus very neatly takes advantage of all its duality laws.

The forms of specification that use a pair of boolean expressions, or a pair of predicates, or predicate transformers, or relations, are concerned with an initial and a final state; in other words, batch computations. They are not concerned with interaction during a computation, nor with time constraints (though there has been some work to further complicate them to do so). But a boolean expression is concerned with any quantity that is of interest: you just use a variable for each. An interaction sequence, or time, are as easily accommodated as initial and final state. For details of such specifications, please see [8, 9].

One reason industry is reluctant to use formal methods may be that they correctly perceive that the methods offered are too complicated for the benefits conferred. One of those complications is the form of specification. A boolean expression gives you the concepts of satisfaction and refinement in their simplest form. And boolean expressions are already in use by every programmer who ever wrote an **if**-statement.

First conclusion: A specification together with an observation yields yes or no; a boolean expression together with values for its variables yields yes or no. A specification is a boolean expression whose variables represent quantities of interest. Refinement is just reverse implication. Partial specifications are put together by conjunction. If instead we use a pair of predicates, or a function from predicates to predicates, or anything else, we make our specifications in an indirect way, we make satisfaction obscure, and we make refinement and specification by parts more complicated. All of these complications may be fun for researchers, but they are unhelpful for software engineers.

Opinion: a program is a specification.

I now want to argue that a program is a specification, and that one's programming language should be a part of one's specification language.

A program tells a computer what to do. If we look at it in a different mood, a program is a description, or prescription, of computer behavior. A computer executes a program by behaving according to the program. People often confuse programs with computer behavior. They talk about what a program "does"; of course it just sits there on the page or screen; it is the computer that "does" something. They ask whether a program "terminates"; of course it does; it is the execution that may not terminate. Furthermore, a computer may not behave according to a program for a variety of reasons: a disk head may crash, a compiler may have a bug, or a resource may become exhausted (stack overflow, number overflow), to mention a few. Then the difference between a program and computer behavior is obvious. We can have a program without executing it, or without having a computer. A program is not behavior, but a specification of behavior.

We often say we are specifying programs when we aren't. To specify a program we could say what programming language it should be in, how the indentation should be done, maybe how long it can be. When we say what relation we want between inputs and outputs, and what execution time we want, we are specifying computer behavior, not specifying a program. Our bad habits of speech have resulted in some strange debates, such as whether programs should have specifications.

A specifier should write the clearest, most understandable specification they can; a programmer's job is to refine it to obtain other specifications, the last of which is a program. Sometimes the clearest, most understandable specification is already a program. When that is so, there is no need for any other specification, and no need for refinement. (Note: if performance (time and space bounds) is of interest, it should be in the specification. A too inefficient program cannot serve as the entire specification, though it could serve as part of the specification.) Sometimes the clearest, most understandable specification is not a program, so the programming language should not be the entire specification language. Sometimes the clearest specification uses notations from the application area, or notations invented by the specifier for this one specification, so the specification language should be open to any useful additions.

There was a great debate a little while ago about whether specifications should be executable. The pro side [4] cited the benefit of being able to test a specification to see if it is the right one. The con side [5] preferred to have a view orthogonal to execution. In my opinion, the initial

specification should be as clear as possible; to make it so, it may be executable, totally nonexecutable, or partly both. The final specification is a program, and so it is executable.

If you put this opinion, that programs are specifications, together with my previous opinion, that specifications are boolean expressions, you get that programs are boolean expressions. For example, if the observable variables are x and y, then the program

$x := x+y$

is just another notation for

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

and the program

if $x>y$ **then** $x := x+y$

is just another notation for

$x>y \wedge x' = x+y \wedge y' = y \quad \vee \quad x \leq y \wedge x' = x \wedge y' = y$

All programs can be treated this way. Even loops are no problem. For example, refining specification S by the loop **while** b **do** P,

$S \Leftarrow$ **while** b **do** P

is just another way of saying

$S \Leftarrow$ **if** b **then** $(P; S)$

As a quick example, let x be an integer variable, and suppose we want to prove that

while $x \neq 1$ **do** $x := x$ **div** 2

is a refinement of

$x \geq 1 \Rightarrow x' = 1$

We prove

$(x \geq 1 \Rightarrow x' = 1) \Leftarrow$ **if** $x \neq 1$ **then** $(x := x$ **div** $2;\ x \geq 1 \Rightarrow x' = 1)$

First we simplify the **then**-part by replacing x with x **div** 2 in $x \geq 1 \Rightarrow x' = 1$, and replace **if** with its boolean equivalent.

$=$ $\quad (x \geq 1 \Rightarrow x' = 1) \Leftarrow x \neq 1 \wedge (x$ **div** $2 \geq 1 \Rightarrow x' = 1) \vee x = 1 \wedge x' = x$

We can simplify x **div** $2 \geq 1$ to $x>1$, and make a boolean rearrangement.

$=$ $\quad (x>1 \wedge x' = 1 \Rightarrow x' = 1) \wedge (x = 1 \wedge x' = x \Rightarrow x' = 1)$

Both these implications are obvious. The laws employed were those of boolean algebra and arithmetic; no special refinement laws are needed.

It is necessary, both for clear specification and for stepwise refinement, that the programming connectives be defined for all specifications. For example, if S and T are any specifications (possibly but not necessarily programs), then $S;T$ is a specification that says "behave according to S, and then after according to T". If we refine specification S by a sequential composition $T;U$, we want to be able to prove that we have made a correct step

$S \Leftarrow (T; U)$

before we further refine T and U. Good refinement methods, such as VDM and the Refinement Calculus, allow all specifications to be composed by programming connectives.

I believe it is also useful to allow programs to be composed by specification connectives. For example,

$(x := x+1) \vee (x := x+2)$

specifies that it is acceptable to increase x by either 1 or 2. If programs are specifications and specifications are boolean expressions, then programs can be connected by ordinary disjunction, and it is unnecessary to invent a special new operator for nondeterministic choice. Similarly we can specify

$x>0 \Rightarrow (x := x+1)$

or use any other boolean connectives for programs. Those formalisms that do not accept programs as specifications find it necessary to duplicate operators and rules at each level.

I once wrote a paper whose abstract said "Programs are Predicates" [7] (by "predicate" I meant what I now mean by "boolean expression"). C.A.R.Hoare has written two different papers both with that same phrase as title: "Programs are Predicates" [10, 11]. It certainly alliterates nicely, and succinctly expresses the combination of my first two opinions.

Second conclusion: Programs specify computer behavior, so specifications should be able to make use of programming notations, as well as application notations and any other notations. We should be able to refine specifications to programs in steps, so that in the middle of this process we may have a mixture of programming and nonprogramming notations. Programs should be given meaning in the same way as specifications so that the mixture is meaningful.

Opinion: total correctness is a mistake.

One way to relate partial and total correctness is by the informal equation

partial correctness + termination = total correctness

Another way is by the informal inequation

partial correctness + time > total correctness

What I mean by this is the following. Start with a partial correctness formalism. Add a variable to stand for time. Sprinkle into your program or specification time increments as appropriate. Use the formalism to reason about the time variable exactly the same way you reason about the other variables. By finding the execution time you know more than just whether execution terminates, and it takes less effort!

The time variable can be continuous and the time increments can be exactly the execution times of the machine instructions; that way you can reason about real-time. Or, if you prefer, the time variable can be integer-valued, and the time increments can count iterations of loops, ignoring all else; that way you get a machine-independent measure. You can choose whatever measure of time you like, but the domain of the time variable should include an infinite number to allow for nontermination.

For the example program of the previous section, we might like to show that, when $x \geq 1$, the execution time is bounded by $log\ x$, where time is just iteration count. That means proving

$$(x \geq 1 \Rightarrow t' \leq t + log\ x) \Leftarrow \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ t := t+1;\ x \geq 1 \Rightarrow t' \leq t + log\ x)$$

The point is that there is no extra theory or proof rules to learn in order to prove the time bound, and hence to prove termination. We might like to show that when $x<1$ execution is nonterminating. That means proving

$$(x<1 \Rightarrow t'=\infty) \Leftarrow \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ t := t+1;\ x<1 \Rightarrow t'=\infty)$$

If we used a real-time increment, the calculation would be no harder. When we place a time increment $t := t+e$ in a program, the expression e can depend on the values of variables; it doesn't have to be a constant. If we cannot say precisely what the time increment is, perhaps we can say what its bounds are: $a \leq t'-t \leq b$. For specification, refinement, and proof, we used only the notations and concepts of programming, arithmetic, and boolean expressions.

There are two usual ways to give meaning to loops (recursions) in a total correctness semantics: one is a limit of a sequence of approximations, the other is a least fixpoint.

The limit of approximations works like this. Define

$$W_0 = \textbf{true}$$
$$W_{n+1} = \textbf{if } b \textbf{ then } (S;\ W_n)$$

Then

$$(\textbf{while } b \textbf{ do } S) = (\forall n \cdot W_n)$$

As an example, we can find the semantics of

$$\textbf{while } x \neq 1 \textbf{ do } x := x \textbf{ div } 2$$

in one integer variable x. We find

$$W_0 = \textbf{true}$$
$$W_1 = \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ \textbf{true})$$
$$= (x=1 \Rightarrow x'=1)$$
$$W_2 = \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ x=1 \Rightarrow x'=1)$$
$$= (1 \leq x<4 \Rightarrow x'=1)$$

Jumping to the general case, which we could prove by induction,

$$W_n = (1 \leq x<2^n \Rightarrow x'=1)$$

And so

$$\begin{aligned} & (\textbf{while } x\neq 1 \textbf{ do } x := x \textbf{ div } 2) \\ = \ & (\forall n \cdot 1\leq x<2^n \Rightarrow x'=1) \\ = \ & (1\leq x \Rightarrow x'=1) \end{aligned}$$

In effect, we are introducing a time variable in disguise: it is the subscript n. W_n is the strongest specification of behavior that can be observed before time n, in the measure that counts iterations.

The other usual way to define **while**-loops is as a least fixpoint. There are two axioms. The construction axiom

$$(\textbf{while } b \textbf{ do } S) \ = \ \textbf{if } b \textbf{ then } (S;\ \textbf{while } b \textbf{ do } S)$$

says that a while-loop equals its first unrolling. Stated differently, **while** b **do** S is a solution of the fixpoint equation (in unknown W)

$$W \ = \ \textbf{if } b \textbf{ then } (S;\ W)$$

The induction axiom

$$(\forall s, s' \cdot \ W \ = \ \textbf{if } b \textbf{ then } (S;\ W)) \ \Rightarrow \ (\forall s, s' \cdot \ W \ \Rightarrow \textbf{while } b \textbf{ do } S)$$

(where s is the state variables) says that **while** b **do** S is as weak as any fixpoint, so it is the weakest (least strong) fixpoint. These axioms introduce a new form of arithmetic, **while**-loop arithmetic, in place of the arithmetic of a time variable.

A total correctness semantics makes the proof of invariance properties difficult, or even impossible. For example, we cannot prove

$$x'\geq x \ \Leftarrow \ \textbf{while } b \textbf{ do } x'\geq x$$

which says, quite reasonably, that if the body of a loop doesn't decrease x, then the loop doesn't decrease x. The problem is that the semantics does not allow us to separate such invariance properties from the question of termination. If, in place of the above, we write

$$x'\geq x \ \Leftarrow \ \textbf{if } b \textbf{ then } (x'\geq x;\ t := t+1;\ x'\geq x)$$

then the proof of the invariance property is easy.

In practice, neither the limit of approximations nor the fixpoint axioms are usable for programming. Instead, those who use formal methods tend to split the problem into partial correctness and termination argument. Partial correctness of

$$(x\geq 1 \Rightarrow x'=1) \ \Leftarrow \ \textbf{while } x\neq 1 \textbf{ do } x := x \textbf{ div } 2$$

is

$$(x\geq 1 \Rightarrow x'=1) \ \Leftarrow \ \textbf{if } x\neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ x\geq 1 \Rightarrow x'=1)$$

as we have already seen. For termination they use a “variant” or “bound function” or “well-founded set”. In this example, they show that for $x>1$, x is decreased but not below 0 by the body $x := x \textbf{ div } 2$ of the loop. The bound function is again time in disguise; they are showing

that execution time is bounded by x. Then they throw away the bound, retaining only the one bit of information that there is a bound, and hence termination. In the example, showing that x is a variant corresponds to the proof of

$$(x{\geq}1 \Rightarrow t'{-}t{\leq}x) \quad \Leftarrow \quad \textbf{if } x{\neq}1 \textbf{ then } (x{:=}\, x \textbf{ div } 2;\ t{:=}\, t{+}1;\ x{\geq}1 \Rightarrow t'{-}t{\leq}x)$$

Thus we express the termination proof in exactly the same form as the partial correctness proof. This linear time bound is rather loose; for about the same effort, we proved a logarithmic time bound. And in exactly the same way, we proved nontermination when $x{<}1$. And we didn't require any extra theory or proof rules beyond boolean algebra and arithmetic.

We can even prove termination of an unboundedly long computation by finding a finite time bound! Let x and y be natural variables. Let $x{:=}\,?$ assign an arbitrary natural number to x, and similarly $y{:=}\,?$. Now consider the program

$x{:=}\,?;\ y{:=}\,?;$
while $\neg\ x{=}y{=}0$ **do**
  if $y{>}0$ **then** $y{:=}\, y{-}1$
  else $(x{:=}\, x{-}1;\ y{:=}\,?)$

Let $f{:}\ nat{\rightarrow}nat$. Let Σx mean $\Sigma i{:}\ 0..\, x{-}1{\cdot}\, f\, i$ (the sum of the first x function values). We say nothing more about function f; it is totally undetermined. Since x is changed to a fresh value just before $y{:=}\,?$, we can replace $y{:=}\,?$ by $y{:=}\, f\, x$. Function f is finite, and therefore proving that $\Sigma x{+}x{+}y$ is the execution time proves termination. We prove

$$\begin{aligned} t' = t{+}\Sigma x{+}x{+}y \quad \Leftarrow\!\!= \quad & \textbf{if } x{=}y{=}0 \textbf{ then } ok \\ & \textbf{else if } y{>}0 \textbf{ then } (y{:=}\, y{-}1;\ t{:=}\, t{+}1;\ t' = t{+}\Sigma x{+}x{+}y) \\ & \quad \textbf{else } (x{:=}\, x{-}1;\ y{:=}\, f\, x;\ t{:=}\, t{+}1;\ t' = t{+}\Sigma x{+}x{+}y) \end{aligned}$$

The proof is in three parts:

$$t' = t{+}\Sigma x{+}x{+}y \quad \Leftarrow \quad x{=}y{=}0 \wedge x'{=}x \wedge y'{=}y \wedge t'{=}t$$

$$t' = t{+}\Sigma x{+}x{+}y \quad \Leftarrow \quad y{>}0 \wedge (y{:=}\, y{-}1;\ t{:=}\, t{+}1;\ t' = t{+}\Sigma x{+}x{+}y)$$

$$t' = t{+}\Sigma x{+}x{+}y \quad \Leftarrow \quad x{>}y{=}0 \wedge (x{:=}\, x{-}1;\ y{:=}\, f\, x;\ t{:=}\, t{+}1;\ t' = t{+}\Sigma x{+}x{+}y)$$

In the second and third parts, make the substitutions indicated by assignments, and simplify. Incidentally, the limit of approximations definition and the fixpoint definition disagree on the meaning of this loop. According to the limit of approximations, the **while**-loop equals

$$x{=}0 \Rightarrow x'{=}y'{=}0$$

According to the fixpoint axioms, it equals

$$x'{=}y'{=}0$$

To bring the two back into agreement, we must enter the realm of transfinite numbers, extending the approximations beyond infinity, as though a loop could be iterated more than an infinite number of times; for further details, see [2]. From Gödel and Turing we know that a complete and consistent theory in which termination can be proven is impossible, so any total correctness theory will therefore be incomplete in its treatment of termination.

Similar to the technical difficulties with total correctness, there is a philosophical difficulty. (If you are allergic to philosophy, or perhaps I mean immune to it, skip ahead to the conclusion.) Suppose you are given some software and a specification of it. You are entitled to complain if you observe any behavior contrary to the specification when you execute the software. If the specification says $x'=2 \wedge t'-t<100$ (measuring time in seconds), then you can complain if either you get a final value of x other than 2 or the computation takes 100 seconds or longer. If the specification just says $x'=2$, you are entitled to complain if a computation delivers a final value of x other than 2, but you are not entitled to complain about the length of time it takes. If it takes forever, there is never a time when you can complain that it has taken too long. If the specification says $x'=2 \wedge t'-t<\infty$, promising finite execution time (termination) but giving no other time bound, and the computation takes forever (nontermination), you still cannot complain; there is never a time when you can say that the specification has been violated. A promise of termination without a time bound is a worthless promise.

Third conclusion: a so-called "total correctness" semantics is not worth its trouble. It is a considerable complication over a partial correctness semantics in order to gain one bit of information of dubious value. So-called "partial correctness" with a time variable provides more information at less cost. I propose that we drop the term "total correctness", since it isn't total in any sense. We can also drop the term "partial correctness", since it is not in contrast to anything. Although it is clear to me that total correctness formalisms are inferior to partial correctness plus time, I do not expect total correctness to be abandoned by those who have a deep commitment to it. It takes a generation to make such a change in a research community. My hope is that those who are not yet committed to total correctness will choose a better path.

Acknowledgment

I am grateful to Leslie Lamport for comments on a draft of this paper. We seem to be in agreement on all points, though Lamport downplays the advantage of boolean expressions over predicates, and says that time is most conveniently added in the form of temporal operators. Lamport's view can be found in [14].

References and Further Reading

[0] R.-J.R.Back: "a Calculus of Refinement for Program Derivations", *Acta Informatica*, v.25 p.593-624, 1988.

[1] R.-J.Back, J.vonWright: *Refinement Calculus: a Systematic Introduction*, Springer, 1998

[2] P.Cousot, R.Cousot: "Constructive Versions of Tarski's Fixed Point Theorems", *Pacific Journal of Mathematics* v.82 n.1 p.43-57, 1979.

[3] M.Frappier: *a Relational Basis for Program Construction by Parts*, PhD thesis, University of Ottawa, 1995.

[4] N.E.Fuchs: "Specifications are (preferably) executable", IEE/BCS Software Engineering Journal v.7 n.5 p.323-334, 1992.

[5] I.J.Hayes, C.B.Jones: "Specifications are not (necessarily) executable", IEE/BCS Software Engineering Journal v.4 n.6 p.330-338, 1989.

[6] E.C.R.Hehner: "Predicative Programming", *Communications of the ACM* v.27 n.2 p.134-151, 1984.

[7] E.C.R.Hehner: "Termination is Timing", International Conference on Mathematics of Program Construction, The Netherlands, Enschede, 1989 June; also J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science v.375, p.36-47, 1989. hehner.ca/TT.pdf

[8] E.C.R.Hehner: *a Practical Theory of Programming*, Springer, 1993; current edition hehner.ca/aPToP

[9] E.C.R. Hehner: "Abstractions of Time", chapter 12 in *a Classical Mind, Essays in Honour of C.A.R.Hoare*, edited by A.W. Roscoe, Prentice-Hall International Series in Computer Science, p.191-210, 1994; hehner.ca/AoT.pdf

[10] C.A.R.Hoare: "Programs are Predicates", in C.A.R.Hoare, J.C.Shepherdson: *Mathematical Logic and Programming Languages*, p.141-154, Prentice-Hall International, London, 1985, and in C.A.R.Hoare, C.B.Jones: *Essays in Computing Science*, p.333-349, Prentice-Hall International, 1989.

[11] C.A.R. Hoare: "Programs are Predicates", ICOT journal v.38, 1993.

[12] C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall International, 1998.

[13] C.B.Jones: *Systematic Software Development using VDM*, Prentice-Hall International, 1986 and 1990.

[14] L. Lamport: "an Old-Fashioned Recipe for Real Time", *ACM TOPLAS* v.16 n.5 p.1543-1571, 1994

[15] C. Morgan: "the Cuppest Capjunctive Capping, and Galois", pages 317-332 in *a Classical Mind, Essays in Honour of C.A.R.Hoare*, edited by A.W. Roscoe, Prentice-Hall International, 1994.

[16] C.Morgan: *Programming from Specifications*, second edition, Prentice-Hall International, 1994.

[17] J.M.Spivey: *the Z Notation: a Reference Manual*, Prentice-Hall International, 1989.

[18] N.Ward, I.Hayes: "Applications of Angelic Nondeterminism", University of Queensland, *Proceedings 6th Australian Software Engineering Conference*, 1991.