# Retrospective and Prospective for Unifying Theories of Programming

Eric Hehner

Department of Computer Science, University of Toronto,
Toronto ON M5S 2E4 Canada
hehner@cs.utoronto.ca

**Abstract** This paper presents a personal account of developments leading to Unifying Theories of Programming, and some opinions about the direction the work should take in the future. It also speculates on consequences the work will have for all of computer science.

## 0 UTP and me

My introduction to formal methods was the book *a Discipline of Programming* [2] by Edsger Dijkstra in 1976. I wrote a small contribution in a paper named **do** *considered* **od** [14] in that same year. In that paper I proposed recursive refinement as a way of composing programs, and a different way of generating the sequence of approximations for loop semantics that is more general than the one in Dijkstra's book, applying to all looping constructs, including general recursion.

It was standard in semantics work then (and for some people, it remains so today) to use a meaning function (sometimes written as double square brackets) that maps program text to its meaning. In Dijkstra's book, he used the $wp$ function to map a program text and postcondition to a precondition. If $S$ is some program text, and $R$ is a postcondition, then $wp(S, R)$ is the exact precondition[1] for execution of

[1] The English meaning of "precondition" is "something that is necessary beforehand". Dijkstra was using it to mean "something that is sufficient beforehand". In Dijkstra's use, the "weakest precondition" $wp$ was the weakest sufficient condition, *i.e.* the necessary and sufficient condition. To avoid misusing the word "precondition", I am saying "exact precondition" to mean the condition that is both necessary and sufficient.

$S$ to terminate and establish postcondition $R$. In my 1976 paper, I made the proposal that we should stop thinking of programs as mere text, and start thinking of them as mathematical expressions in their own right. We should not need a function to map a program to its meaning. My proposal was that, like any mathematical expression, a program can stand for its meaning all by itself. So, in that paper, program $S$ is a function that maps a postcondition $R$ to a precondition, written $S(R)$. Sequential composition (semicolon) is just function composition. I proposed that the arrow in a guarded command is a lifted implication, that the box connecting guarded commands is a lifted conjunction, and that the **if fi** brackets are a "totalizer". That proposal seems tame today, but in 1976 it was apparently bizarre, causing rejection of the paper in its first submission; and in its second submission the referees insisted that it be removed from the paper for publication. Since there were other contributions of the paper that I really wanted published, I obeyed the referees and removed it from the paper for publication in *Acta Informatica* in 1979. But it remains in the 1976 technical report version. Fortunately for all of us, Ralph Back in Finland read the technical report, adopted the proposal, and began the work called "Refinement Calculus", culminating in a wonderful book with that same name in 1998 [0].

Meanwhile, I made an amazing discovery: that Dijkstra's book was not the first work on formal methods; from the lack of references in the book, I had supposed it was. But it owed a lot to a paper by C.A.R.Hoare in 1969 [16]. That paper, and another in 1972 on data abstraction [17], and some lectures by Tony on CSP, convinced me to spend my sabbatical in 1981 in Oxford. It was an intellectually lively place, including Jean-Raymond Abrial, Steve Brookes, Peter Henderson, Cliff Jones, Lockwood Morris, David Park, Bill Roscoe, Dana Scott, Ib Sorensen, Joe Stoy, Bernard Sufrin, Jim Woodcock, and others. But it was not a good year for the Hoare family.

I had two projects while I was in Oxford. I had started writing a book in the year before going there, but then David Gries started writing the same book, and sending me the chapters for comment. David wrote much faster than I did, and quickly overtook me. So I decided to put my effort into comments on David's book, and abandon mine. But in Oxford, Tony persuaded me that there is room for another book, especially if I pitch my book at a different level than David's. So I resumed writing, aiming for a more advanced audience. The book [12] was published in Tony's series in 1984, and in it programs were predicate transformers.

My other project in Oxford in 1981 was to find a good model for CSP. I decided that in this project, programs were not predicate transformers, but predicates. Each day I would go into Tony's office before he arrived, and fill his board with my latest

formulas, hoping that would catch his attention. I guess it worked. The paper [13] was a technical report in 1981, and published in *TCS* in 1983.

I liked the idea of using predicates for programs so much that I decided to apply it beyond CSP to a wide variety of programming constructs. My principles of "predicative programming" were:

- a specification is a predicate[2]
- refinement is implication
- a program is an implemented specification

In 1982, IFIP Working Group 2.3 (Programming Methodology) allowed me an extraordinary 4 hours to present these ideas. For the most part, the presentation went very well, but there was one point that went badly. I wanted a "total correctness" formalism (who wouldn't?), and I achieved it by borrowing the weakest specification (given an initial state, any final state is satisfactory) to represent possibly nonterminating computations. I "justified" it by saying that if you don't care what the result is, then you don't care if there is a result (I am not defending that argument any more). I gained "total correctness" at the cost of making sequential composition (semicolon) almost but not quite associative. I had a theorem saying that if the state space is infinite (one integer variable makes it infinite) then semicolon is associative for all programs. I had another theorem saying that if there is at least one variable not appearing in any program (one unused boolean variable is enough), then again semicolon is associative for all programs. I remember Butler Lampson saying that I should just assume there's one extra variable, and get on with it. I also remember that Tony was unhappy; for him, associativity of semicolon had to be unqualified.

In 1982 March, Tony came to Toronto for a week, in part so that we could resolve the problem. I was opposed to adding an extra boolean variable that would not appear in any program, but would burden nearly every non-program specification. Tony was opposed to any qualification on associativity. In the end, Tony convinced me to add the alternative he preferred as an appendix to my paper. I tried to give the variable some physical motivation, so I called it $s$ for "start/stop", saying that $s$ with "initial" decoration means "the computation has started" and that $s$ with "final" decoration means "the computation has stopped". That variable

---

[2] I am using the word "predicate" here as I used it back then, and as some people still use it today, to mean a boolean expression, particularly one that contains or may contain subexpression(s) of other type(s), and/or quantifiers. I now say "boolean expression" no matter what types its subexpression have, and no matter what operators are used within. I now use "predicate" to mean a function whose result is boolean.

became the *ok* variable in UTP [15]. "Predicative Programming" was published [11] in *CACM* in 1984.

There was a growing number of theories of programming. I had predicative programming; Edsger Dijkstra had *wp* ; Cliff Jones had VDM; David Parnas had limited domain relations, and a second theory that he called "standard semantics"; and there was another theory of partial relations proposed by Bill Robison and independently by Bernard von Stengel, that later became the refinement semantics of Z. These were not just notationally different; they had substantive differences in expressive power and in their treatment of termination. So I set out to compare these different theories in a paper called "Termination Conventions and Comparative Semantics", published in *Acta Informatica* in 1988 [10]. The basis of the comparison was a translation from each of them to my own predicative semantics, and vice versa. I included a catalogue of semantics for all the aforementioned theories, expressed both with and without the extra boolean variable that indicates proper termination. So, in my mind at least, this paper was very much a forerunner of the UTP work.

One more idea that may have influenced UTP is parallel by merge. In 1990, Theo Norvell was my PhD student, and he suggested parallel by merge as a way of defining parallel composition that is both implementable and insensitive to frame. That is the form of parallelism in the 1993 edition of my book *a Practical Theory of Programming* [5], and in UTP in 1998. I have since abandoned it, and for the 2002 edition and onward I have returned to simple conjunction (to keep implementability, it must be sensitive to frame) that I had used from 1981 to 1989 [11, 13].

Since publication of UTP, both it and my work have been extended to include probabilistic computation [6], but I think neither work influenced the other; perhaps we were both influenced by the same source [18].

This history of work leading to UTP has been a personal one, and I am not sure it accords well with a history that Tony Hoare and He Jifeng might tell. My doubt comes from the fact that none of the work I have mentioned, except for the predicate model of CSP [13], was referenced in the UTP book.

## 1 UTP without me

I am pleased to think that I made some contribution to the UTP project. But there is an important point on which I have tried hard and so far failed to have any influence. I think the point is inevitable, so I will now make another attempt.

The tradition in programming theories is not to speak directly about execution time. To refer to theories that talk about whether a computation terminates, and the

result upon termination, we commonly use the words “total correctness”, suggesting that nothing else is of interest. I suspect the sentiment was (and maybe still is) that execution time is too dependent on factors (compiler, hardware) beyond the program. There are circumstances when time is important, called real-time or reactive programming. And theories have been invented [1], and new ones are still being invented [3], to reason about execution time. An entire logic, called temporal logic, was invented to specify and reason about timing. But none of that is necessary. All you need to do is add a time variable, placing increments ( $t := t + something$ ) in the program wherever they are needed to account for the time required by the other operations in the program. Then you reason about the time variable exactly the same way you reason about the other variables, using exactly the same theory you were using before you added time. I presented this position in a paper [9] published in 1989 January.

Then in June of that year, in the opening address of the first MPC [8], I presented a more compelling reason for the inclusion of a time variable. To calculate the exact execution time, the time variable can be real-valued, and the increments should be exactly the execution time of the instructions compiled for the machine that will execute the instructions. A more abstract, machine-independent measure of time uses an integer-valued time variable, counting iterations of loops and recursive calls, ignoring all else. A still more abstract measure of time uses a boolean-valued time variable that just distinguishes finite execution time (termination) from infinite execution time (nontermination); that is the $ok$ variable of UTP. But there's a big difference between numeric (real- or integer-valued) time and boolean time: the former can tick, and the latter cannot. We can do arithmetic on the time variable if it is a numeric type, but not if it is boolean. And that has a profound effect on the semantics and proof rules of the programming language, as I shall explain.

## 2 If and When

The acceptance of 0 as a number has taken a long time, and is still incomplete. In English, no-one quite knows whether to treat 0 as singular or plural, does he? On your keypad or telephone it is placed after 9 , which is mathematically silly. In the 1991 Toronto phone book, there is a page that helpfully gives the time difference to various places in the world; to the U.K. it says “+5”, and to Costa Rica it says “–1”. But to Cuba it says “NA”, and the legenda explains “time difference not applicable”. By 1996 they tried to correct it; for Cuba it says “=”, with the same explanation. In

1997 they discovered the number $0$, but they felt the need then, and still do today, to explain that $0$ means "no time difference".

When we say "There are a number of issues to discuss.", we don't mean there might be $0$ of them. When $0$ really is a possibility, people often add the phrase "if any", as in "Please put all the leftovers in the fridge, if there are any.". They create a case analysis, when none was needed. For example, the 1991 Canadian census asked the question "How many persons who have a usual home somewhere else in Canada stayed here overnight between 1991 June 3 and 4?", then offers a place to tick if there were none, and a box to fill with the number of persons if there were some. The people designing the form probably know perfectly well that the box is sufficient, but without the place to tick "if none" they would be overwhelmed by people complaining that they can't answer the question.

The Fortran language of 1955 had a loop construct, but its body had to be executed at least once; I suppose it seemed senseless to have a loop whose body might be executed $0$ times. The error was corrected in Algol in 1958, and in PL/I, and in Pascal, in part: iteration might be $0$ times, but the data structure over which one is iterating, the array, had to have at least one element. In Pascal that meant there was no null string. And that put the algebra of data structures back where the algebra of natural numbers was prior to 1930. We learn, but slowly; two steps forward, one step back.

The authors of UTP might have chosen to include a boolean variable to distinguish executions that take $0$ time from those that take positive time. This variable would complicate the semantics to no advantage, and it would infect all specifications, causing the authors to invent "designs", which are specifications with this variable suppressed but still implicit. I commend the authors for not making this mistake. Perhaps someday, in English (or its successor), we won't feel the need to ask for "the number, if any"; we will simplify by just asking for "the number", accepting $0$ as an answer.

In English, one sometimes hears the phrase "if ever", or "if and when", as in "I'll deal with that if and when it happens.". If we just say "I'll deal with that when it happens.", undoubtedly someone would immediately ask: "What if it never happens?". But it seems to me that case is already covered: if it never happens, I'll deal with it never. We simplify by eliminating the case analysis, and to do that we must learn to accept $\infty$ as an answer to the question "when?". We are not bothered by the different grammar in the two sentences "I don't have any bananas." and "I have $0$ bananas."; one uses a negative verb and the other a positive verb, but we take them to mean the same thing. Likewise we should take "It never happens." and "It happens at time $\infty$." to mean the same thing. Perhaps someone in the future will

show some census forms in which the case $\infty$ was separated off unnecessarily, and that speaker's audience can all have a good laugh at their ancestors' unwillingness to accept $\infty$ as a number.

The authors of UTP have chosen to include a boolean variable $ok$ to distinguish executions that take finite time from those that take infinite time. This variable complicates the semantics to no advantage, and it infects all specifications, causing the authors to invent "designs", which are specifications with this variable suppressed but still implicit. Worse than that, this variable causes duplication of work. Suppose I want to show that a computation involving loops delivers a certain result within a certain time bound. The next section shows that the work necessary to prove $ok'$ is equivalent to finding an upper time bound, which I must repeat using a time variable in order to prove an upper time bound.

## 3 What is the meaning of loops?

There are two usual ways to give meaning to loops (and recursions) in a "total correctness" semantics: the limit of a sequence of approximations, and a least fixpoint. To find the meaning of $b*S$ using the limit of approximations, define

$$W_0 = \textbf{true}$$

$$W_{n+1} = (S;\ W_n) \lhd b \rhd II$$

Then

$$b*S = (\forall n \cdot W_n)$$

where the quantification may need to continue past the naturals and through the transfinite ordinals. As an example, we can find the semantics of

$$(x{\neq}1) * (x{:=}\ x\ \textbf{div}\ 2)$$

in one integer variable $x$. We find

$$W_0 = \textbf{true}$$

$$W_1 = (x{:=}\ x\ \textbf{div}\ 2;\ \textbf{true}) \lhd x{\neq}1 \rhd II$$

$$= (x{=}1 \Rightarrow x'{=}1)$$

$$W_2 = (x{:=}\ x\ \textbf{div}\ 2;\ x{=}1 \Rightarrow x'{=}1) \lhd x{\neq}1 \rhd II$$

$$= (1{\leq}x{<}4 \Rightarrow x'{=}1)$$

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

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

And so

$$(x\neq 1) * (x := x \textbf{ div } 2)$$
$$= (\forall n \cdot 1 \leq x < 2^n \Rightarrow x' = 1)$$
$$= (1 \leq x \Rightarrow x' = 1)$$

A sequence of approximations introduces an integer-valued time variable in disguise: it is the subscript $n$. $W_n$ is the strongest specification of behavior that is observed before time $n$, in the measure that counts iterations. If we have an integer-valued time variable, it is unnecessary to introduce another one for the same purpose, and we can simplify the semantics of loops.

The other usual way to define loops is as a least fixpoint.

$$b * S = \mu X \cdot (S; X) \lhd b \rhd II$$

This is closely analogous to defining the natural numbers $\mathbb{N}$ as a least fixpoint.

$$\mathbb{N} = \mu X \cdot \{0\} \cup \{n+1 \mid n \in X\}$$

For more familiarity, we can remove $\mu$ by replacing the definition with two axioms called construction and induction. Loop construction

$$b * S = (S; b * S) \lhd b \rhd II$$

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

$$X = (S; X) \lhd b \rhd II$$

It is analogous to natural construction

$$\mathbb{N} = \{0\} \cup \{n+1 \mid n \in \mathbb{N}\}$$

which says that $0$ is a natural number, and if $n$ is a natural number, so is $n+1$ ($II$ is analogous to $0$, $\lhd b \rhd$ is analogous to $\cup$, and unrolling is analogous to adding 1). Stated differently, it says that $\mathbb{N}$ is a fixpoint of an equation. Loop induction

$$(\forall \sigma, \sigma' \cdot X = (S; X) \lhd b \rhd II) \Rightarrow (\forall \sigma, \sigma' \cdot X \Rightarrow b * S)$$

where $\sigma$ is the state variables, says that $b * S$ is as weak as any fixpoint, so it is the weakest (least strong) fixpoint. It is analogous to natural induction, which can be written in a nontraditional form (to make the analogy clearer), replacing predicate satisfaction with set membership, as follows:

$$\forall X \cdot X = \{0\} \cup \{n+1 \mid n \in X\} \Rightarrow \mathbb{N} \subseteq X$$

which says that $\mathbb{N}$ is a subset of any fixpoint, so it is the smallest fixpoint. Once again, if we lack an arithmetic time variable, then the loop semantics must compensate by introducing a kind of loop-arithmetic. If we have an arithmetic time variable, this is unnecessary, and we can simplify the semantics of loops.

Programming from specifications by means of refinement replaces the question "what does this program mean?" with the question "does this program refine that specification?" [4]. All a programmer needs to know about the meaning of program

$P$ is: for what specifications $S$ is $S \Leftarrow P$ a theorem? What a programmer needs to know about $II$ is

$$\sigma' = \sigma \quad \Leftarrow \quad II$$

In UTP (as in my work), $II$ is defined by strengthening that refinement to equality, but for programming, all we need is the implication. The same comment applies to assignment, conditional, and sequential composition.

I am content to form loops by recursive refinement (as in my 1976 paper [14]). For example, if the specification (in one integer variable $x$) is $x \geq 1 \Rightarrow x' = 1$, I can refine it as follows:

$$(x \geq 1 \Rightarrow x' = 1) \quad \Leftarrow \quad II \lhd x = 1 \rhd (x := x \textbf{ div } 2;\ x \geq 1 \Rightarrow x' = 1)$$

With this refinement, we can now execute the specification $x \geq 1 \Rightarrow x' = 1$ by executing what refines it, and when specification $x \geq 1 \Rightarrow x' = 1$ is encountered again, it is again executed by executing what refines it. That's a loop. Knowing what $II$, assignment, conditional, and sequential composition refine is sufficient for proof of this refinement; we do not need any further theory for loops.

If we are interested in execution time, we include a time variable. Let's make it integer-valued, and count iterations. We can prove

$$(x \geq 1 \Rightarrow t' \leq t + log\ x) \quad \Leftarrow \quad II \lhd x = 1 \rhd (x := x \textbf{ div } 2;\ t := t+1;\ t' \leq t + log\ x)$$

which says that for positive $x$, the execution time is bounded above by $log\ x$. We can also prove

$$(x < 1 \Rightarrow t' = \infty) \quad \Leftarrow \quad II \lhd x = 1 \rhd (x := x \textbf{ div } 2;\ t := t+1;\ x < 1 \Rightarrow t' = \infty)$$

which says that for nonpositive $x$, the execution time is infinite. And for free we get the conjunction of all that we proved previously: execution satisfies

$$(x \geq 1 \Rightarrow x' = 1) \wedge (x \geq 1 \Rightarrow t' \leq t + log\ x) \wedge (x < 1 \Rightarrow t' = \infty)$$

It is extremely useful to be able to prove partial properties separately, and specifically to be able to prove results and timing separately, and then to combine them for free.

Although I am content to form loops without any loop syntax and without any theory that pertains to loops, apparently some people feel the need for loop syntax and theory. So UTP provides the syntax $b*P$. All we need to say about it is that

$$S \quad \Leftarrow \quad b*P$$

is syntactic sugar for

$$S \quad \Leftarrow \quad (P;\ S) \lhd b \rhd II$$

We do not attribute any meaning to $b*P$, but only to the refinement $S \quad \Leftarrow \quad b*P$. We do not need a limit of a sequence of approximations. We do not need least fixpoints. If we want to know about time (including termination), we add a time variable, but we don't have to complicate the semantics of loops.

My example recursive refinement has the form of a * loop, but recursive refinement works for any loop structure, including loops with intermediate exits and deep exits, and for general recursion, not just tail recursion (see [5]).

## 4 What can we prove about loops?

The two traditional ways of defining loop semantics (limit of a sequence of approximations, least fixpoint) for "total correctness" are too complicated to be used in proofs, and in practice they never are used. Instead, those who use formal methods split the problem into a "partial correctness" proof and a termination argument. "Partial correctness" of

$$(x{\geq}1 \Rightarrow x'{=}1) \quad \Leftarrow \quad (x{\neq}1) * (x{:=}\, x \textbf{ div } 2;\ x{\geq}1 \Rightarrow x'{=}1)$$

is exactly

$$(x{\geq}1 \Rightarrow x'{=}1) \quad \Leftarrow \quad (x{:=}\, x \textbf{ div } 2;\ x{\geq}1 \Rightarrow x'{=}1) \lhd x{\neq}1 \rhd \mathit{II}$$

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 variant is again time in disguise; they are showing that the execution time is bounded by $x$ in the measure that counts iterations. Then they throw away the bound, retaining only the one bit of information that there is a bound, and hence termination. In the example, this corresponds to a proof of

$$(x{\geq}1 \Rightarrow t'{\leq}t{+}x) \quad \Leftarrow \quad (x{:=}\, x \textbf{ div } 2;\ t{:=}\, t{+}1;\ x{\geq}1 \Rightarrow t'{\leq}t{+}x) \lhd x{\neq}1 \rhd \mathit{II}$$

This linear time bound is rather loose; for about the same effort we prove a logarithmic time bound. And in exactly the same way, we prove nontermination when $x{<}1$. More generally, we can prove useful lower time bounds; we are not limited to the existence of an upper bound, which is what "total correctness" provides.

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

$$x'{\geq}x \quad \Leftarrow \quad b * (x'{\geq}x)$$

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

$$x'{\geq}x \quad \Leftarrow \quad (x'{\geq}x;\ t{:=}\, t{+}1;\ x'{\geq}x) \lhd b \rhd \mathit{II}$$

as I advocate, then the proof of the invariance property is easy.

## 5 What can we prove about infinite loops?

What can we prove about an infinite loop? According to the least fixpoint semantics, nothing. According to that semantics, **true****P* is equivalent to **true** , which is completely arbitrary behavior. It does not imply $\neg ok'$ ; the behavior may be nonterminating, or terminating. If we add a time variable, we cannot prove $t'=t+\infty$ . If the body of the loop includes communications (interactions), we cannot prove they happen. My way, to prove $S$ , we must prove

$$S \Leftarrow (P;\ t:= t+1;\ S) \lhd \mathbf{true} \rhd II$$

or more simply

$$S \Leftarrow (P;\ t:= t+1;\ S)$$

(which is what I would write in the first place). Taking $II$ as body for the moment, we cannot prove $t'\le t+n$ for finite $n$ ; that would require proving

$$t'\le t+n \Leftarrow t'\le t+1+n$$

which is not so. But we can prove $t'>t+n$ . We can also prove $t'=t+\infty$ ; that requires proving

$$t'=t+\infty \Leftarrow t'=t+1+\infty$$

and since, in my algebra, $\infty$ absorbs finite additions ( $\infty$ is a fixpoint of *tick* ), that refinement is a theorem. If the body of the loop includes communications, we can prove that they do indeed happen.

A specification $S$ is implementable (in UTP terminology, "healthy") if and only if for all initial states (including time) there is a final state (including time) that satisfies the specification with nondecreasing time (and non-undoable communications, but I'll omit that for now):

$$\forall \sigma \cdot \exists \sigma' \cdot\ S \wedge t'\ge t$$

Refinement by a program is proof of implementability. For recursive refinement, we need to know separately that the specification is implementable. Although

$$\mathbf{false} \Leftarrow (P;\ t:= t+1;\ \mathbf{false})$$

is a theorem, we reject **false** because it is unimplementable; we have not implemented a miracle.

Disturbingly, we can prove both of the implementable specifications $x'=2$ and $x'=3$ . Both

$$x'=2 \Leftarrow (t:= t+1;\ x'=2)$$
$$x'=3 \Leftarrow (t:= t+1;\ x'=3)$$

are theorems. There is no inconsistency here. My theory of programming is sound in the following sense: if $S$ is an implementable specification, and $F$ is a program (possibly with call sites), and we can prove the refinement $S \Leftarrow F(S)$ , then no

observation of the corresponding computation will ever contradict $S$. The point is that observations are made at finite times, whereas the results $x'=2$ and $x'=3$ happen at time $\infty$ (never). For exactly the same reason, we can prove both

$$\neg ok' \Leftarrow (t:= t+1;\ \neg ok')$$
$$ok' \Leftarrow (t:= t+1;\ ok')$$

If this is at first disturbing, consider it the price to pay for the ability to prove results and timing separately, and combine them for free.

Perhaps more disturbingly, we can also prove

$$t<\infty \Rightarrow t'<\infty \Leftarrow (t:= t+1;\ t<\infty \Rightarrow t'<\infty)$$

which seems to say that if the computation starts at a finite time, it will end at a finite time. But without a time bound, the specification offers no opportunity for complaint that the computation is taking too long. The theory should allow, and does allow, any computation whose observation does not contradict the specification.

The theory is incomplete in the following sense. Even if $S$ is an implementable specification, and observations of the computation(s) corresponding to $S \Leftarrow F(S)$ never (in finite time) contradict $S$, the refinement might not be provable. But in that case, there is another implementable specification $R$ such that the refinements $S \Leftarrow R$ and $R \Leftarrow F(R)$ are both provable. In that weaker sense, the theory is complete. There cannot be a theory of programming that is both sound and complete in the stronger sense.

## 6 the Problem with Halting

The halting function (predicate) is defined to tell whether a program's execution terminates. I will make two simplifications to the standard formulation, neither of which changes anything essential. We need to encode programs as data so we can apply the halting function to something that represents a program. In the standard formulation, programs are numbered, so we can apply the halting function to a number representing a program. Instead, I use a more transparent encoding: a program is represented by its text (character string). (That is how a program is presented to a compiler or interpreter.) The other simplification is to eliminate all mention of initial state (input). One way to do that is to define the halting function applied to program text $p$ as saying whether "$p$ halts from all initial states" or "$p$ fails to halt on some initial state". Another way to do it is to pick some initial state as the one where execution of any program always starts; if you want some other initial state, just start the program with some initializing assignments to create the state you

want. Define predicate $H$: $\mathbb{T} \to \mathbb{B}$, where $\mathbb{T}$ is the text data type and $\mathbb{B}$ is the boolean data type, so that

$H(\text{“}II\text{”}) = \textbf{true}$

$H(\text{“}\textbf{true}{*}II\text{”}) = \textbf{false}$

and so on. Define text $P$ as follows:

$P = \text{“}(\textbf{true}{*}II) \lhd H(P) \rhd II\text{”}$

If we assume $H$ is a functional program, then $P$ represents a program. Now we ask: what is the result of $H(P)$? If the execution of $P$ terminates, then $H(P)$ is **true**, and $P$ represents a program that is equivalent to $\textbf{true}{*}II$, so execution of $P$ does not terminate. And if the execution of $P$ does not terminate, then $H(P)$ is **false**, and $P$ represents a program that is equivalent to $II$, and so execution of $P$ does terminate. Conclusion: $H$ cannot be a program; it's an incomputable function. That's the orthodox argument, and the orthodox conclusion, first made by Turing, and now found in many textbooks.

In UTP, programs are a special case of specification, so let me generalize $H$ to apply to all specification texts, not just to program texts. In particular,

$H(\text{“}ok'\text{”}) = \textbf{true}$

$H(\text{“}\neg ok'\text{”}) = \textbf{false}$

And this time, we don't make any assumption that $H$ is a functional program (computable function). Define specification text $S$ as follows:

$S = \text{“}\neg ok' \lhd H(S) \rhd ok'\text{”}$

Now we ask: what is the result of $H(S)$? If $S$ specifies terminating behavior, then $H(S)$ is **true**, and so $S$ specifies nonterminating behavior. And if $S$ specifies nonterminating behavior, then $H(S)$ is **false**, and so $S$ specifies terminating behavior. What do you conclude from that?

This argument about specifications has exactly the same form as the orthodox argument about programs. Both arrive at a self contradiction. We look for a way out by looking for an assumption that was wrong. In the argument about programs, the assumption was made that $H$ is a program, so we withdrew that assumption. But from the argument about specifications, we see that the problem is still there, even without that assumption.

My conclusion is that we cannot consistently say the sentence “$H$ tells us, for all specification texts $s$, whether $s$ specifies terminating behavior.”. The inconsistency is not immediately apparent, but the above argument shows us that it's there. This is similar to saying that the sentence “The barber, who is a man, shaves all and only the men in his town who do not shave themselves.” is not obviously self-contradictory, but a short proof or argument shows it to be so. And from the first

version of the story about $H$ applied to program codes, I do not conclude that $H$ is a perfectly well defined but incomputable function; I conclude there also that there is an inconsistency in the definition of $H$.

Let me try to make the inconsistency in the definition of $H$ more apparent. Within $S$, $H(S)$ and $ok'$ have the same rôle. So $S$ represents

$$\neg ok' \lhd ok' \rhd ok'$$

which says, as directly as possible, that if execution terminates, then it doesn't terminate, and if it doesn't terminate then it does. There is nothing wrong with having a primed variable between the conditional triangles; for example, the specification

$$x'{=}2 \lhd even(x') \rhd x'{=}3$$

says quite reasonably that if the final value of $x$ is even, then it should be $2$, and if odd it should be $3$; it is equivalent to $x'{=}2 \vee x'{=}3$. However, the specification $\neg ok' \lhd ok' \rhd ok'$ is equivalent to **false** (independent of the interpretation of $ok'$), so it is unimplementable (unhealthy). We are asking $H$ to tell us the termination status of an unimplementable specification.

Returning to the "program" example

$$P \;=\; \text{“}(\mathbf{true}{*}II) \lhd H(P) \rhd II\text{”}$$

is $P$ an implementable specification? If we assume it is, then it might seem reasonable to ask $H$ about its termination status (without any assumption that $H$ is computable), and we are led into the same contradiction as before. If we assume it isn't and don't ask $H$ about its termination status, we lose the very specification we were using to demonstrate that $H$ is incomputable.

The problem with $H$ doesn't stop there. If we could define $H$ consistently on just the implementable specifications, then we could consistently extend its definition to all specifications by, for example, saying $H(s){=}\mathbf{false}$ for all unimplementable specifications $s$. If $P$ is unimplementable, then $P$ is equivalent to $II$, which is implementable. There is no way out.

The situation is exactly the same as for an interpreter of boolean expressions (also known as a prover). Suppose we try to define $I\colon \mathbb{T}\to\mathbb{B}$ so that, when we apply $I$ to a text representing a boolean expression, we get the result of evaluating the boolean expression. Now define

$$Q \;=\; \text{“}\mathbf{false} \lhd I(Q) \rhd \mathbf{true}\text{”}$$

or instead, to simplify, define

$$Q \;=\; \text{“}\neg I(Q)\text{”}$$

Applying $I$ to $Q$ yields inconsistency. This is exactly Gödel's incompleteness theorem: $Q$ is saying that $Q$ is not a theorem. Either we leave $I$ incompletely

defined (specifically, it does not interpret $Q$), or we suffer inconsistency. (I note with some irony that an interpreter is a meaning function, which I began this paper by eliminating!)

Wait a minute: there is a way out. Interpreter $I$ is a program, and $H$ is just a simplification of $I$: $I$ tells us the result of evaluating, and $H$ just tells us whether there is a result. So $H$ really is a program. Applying $H$ to $P$ and to $S$ results in an infinite loop (as does application of $I$ to $Q$). We could say that $H$ does deliver a result for $P$ and for $S$, and $I$ does deliver a result for $Q$, but only at time $\infty$. The "incomputable" function $H$ is nothing but a program whose execution, for some input, is nonterminating. Such programs are common, and some of them are useful. This way out is a great mathematical simplification.

## 7 the Problem with Vacuum Cleaners

Here's a "proof" that a vacuum cleaner is unbuildable. If you could build one, then you could use it to clean out its own bag. But that's a self contradiction (making the bag empty makes the bag full, and vice versa), so a vacuum cleaner is unbuildable.

The "proof" that a vacuum cleaner is unbuildable is like the "proof" that the halting function is incomputable in the following ways. It accepts without question that a vacuum cleaner is at least a meaningful, consistent concept, just as the standard incomputability proof accepts without question that a halting function is at least a meaningful, consistent concept. Then the vacuum cleaner is applied to itself, just as the halting function is applied to itself. And, most importantly, time is not considered in the argument: in each case, there is no static solution, so we have inconsistency. To restore consistency, we seem to have three options.

The first option, à la Turing, is to remain steadfast in the belief that the vacuum cleaner and halting function are at least meaningful (consistent) concepts, but to label them as "unbuildable" and "incomputable" respectively. That withdraws an assumption made in the argument, but it was an irrelevant assumption. If you could just specify (never mind build) a vacuum cleaner, you arrive at the same contradiction. If you could just specify (never mind compute) the halting function, you arrive at the same contradiction. This option is not a way out. Neither "unbuildability" nor "incomputability" serve the purpose for which they were invented: to restore consistency.

The second option, à la Gödel, is to say that the definition of a vacuum cleaner, and the definition of the halting function, are inconsistent unless we leave them

incomplete, and we do not apply them to the example that gives rise to the contradiction.

The third option is to add a time variable. Then we can ask what really does happen (over time) if we apply them to the troublesome examples. A vacuum cleaner really is buildable, and the halting function really is programmable. What really happens if someone uses a vacuum cleaner to clean out its own bag is that they create an infinite loop, blowing dirt forever around a circular hose. But that's not an inconsistency. Indeed, there are physical systems built intentionally as infinite loops; for example, pumping electrons around a circuit, doing useful work as they go. Likewise, applying the halting function to its troublesome example is an infinite computation, not a self contradiction.

A simpler, but maybe less visual, example, is the problem of the NOT gate. If we could build one, then we could use it in a closed circuit that includes just one NOT gate, and nothing else. If we ignore time, we find an inconsistency: assuming either final state of the circuit leads to a contradiction. The inconsistency is not eliminated by labeling NOT gates "unbuildable" or "incomputable". The problem is eliminated if we outlaw this particular use (and all similar uses) of the NOT gate. But the best solution is to admit that a NOT gate takes time; we look at the circuit's behavior over time, and we do not worry about what its final state might be. It is a useful circuit called an oscillator. (A practical oscillator is more complicated, but at its heart there is a NOT gate in a loop.)

## 8 What is a time bound?

I have argued that a claim of termination should be accompanied by a time bound. Now I ask: what is acceptable as a time bound?

Finding the execution time of any program can always be done by transforming the program into a function that expresses the execution time. To illustrate how, let us again look at the example

$$(n{\neq}1) * (n{:=}\ n\ \mathbf{div}\ 2)$$

in natural variable $n$. The first step in expressing the execution time is, not surprisingly, to get rid of the loop notation in favor of recursive refinement.

$$n'{=}1 \ \Leftarrow\ II \lhd n{=}1 \rhd (n{:=}\ n\ \mathbf{div}\ 2;\ n'{=}1)$$

The next step is to add a time variable, and choose a timing policy. We express the execution time as $f(n)$, where function $f$ must satisfy

$$t'{=}t{+}f(n) \ \Leftarrow\ II \lhd n{=}1 \rhd (n{:=}\ n\ \mathbf{div}\ 2;\ t{:=}\ t{+}1;\ t'{=}t{+}f(n))$$

which can be simplified to

$$f(n) = 0 \lhd n{=}1 \rhd (1 + f(n \textbf{ div } 2))$$

From this recursive definition of $f$, we see

$$f(1) = 0$$
$$f(2) = 1 + f(1) = 1$$
$$f(3) = 1 + f(1) = 1$$
$$f(4) = 1 + f(2) = 2$$

and so on. We also see

$$f(0) = 1 + f(0)$$

which has no finite solution, but according to my axioms for numbers [5], it has solution $\infty$ (because $\infty$ absorbs finite additions). This is exactly the right answer for how long the computation takes when $n$ is $0$. It would have been a duplication of effort to worry first about termination before calculating execution time.

Now consider this famous program whose execution time is considered to be unknown:

$$(n{\neq}1) * ((n{:=}\, n/2) \lhd even(n) \rhd (n{:=}\, 3{\times}n + 1))$$

where $n$ is a natural variable. It is not even known whether the execution time is finite for all $n{>}0$. Following the same steps as before, we find

$$f(n) = 0 \lhd n{=}1 \rhd ((1 + f(n/2)) \lhd even(n) \rhd (1 + f(3{\times}n + 1)))$$

or, more readably,

$$f(n) = \textbf{if } n{=}1 \textbf{ then } 0 \textbf{ else if } even(n) \textbf{ then } 1 + f(n/2) \textbf{ else } 1 + f(3{\times}n + 1)$$

Thus we have an exact definition of the execution time. So why is the execution time considered to be unknown?

If the execution time of some program is $n^2$, we consider that the execution time of that program is known. Why is $n^2$ accepted as a time bound, and $f(n)$ as defined above not accepted? The reason is not that $f$ is defined recursively; the square function is defined in terms of multiplication, and multiplication is defined recursively. The reason cannot be that $n^2$ is well behaved (finite, monotonic, and smooth), while $f$ jumps around wildly and might sometimes be infinite-valued; every jump and change of value in $f$ is there to fit the original program's execution time perfectly, and we shouldn't disqualify $f$ because it is a perfect bound. One might propose the length of time it takes to compute the time bound as a reason to reject $f$. Since it takes exactly as long to compute the time bound $f(n)$ as to run the program, we might as well just run the original program and look at our watch and

say that's the time bound. But $log\ log\ n$ is accepted as a time bound even though it takes longer than $log\ log\ n$ to compute $log\ log\ n$.

Could the reason be that function $f$ is unfamiliar? that it has not been well studied and we don't know much about it? If it were as well studied and familiar as square, would we accept it as a time bound?

Consider the linear search program to find the first occurrence of a given item $x$ in a given list $L$, and report its position as the final value of variable $h$. Suppose that $L$ is infinitely long, and we are told that there is at least one occurrence of $x$ in the list. We can prove that the execution time (counting iterations) is $h'$.

$$t'=t+h' \Leftarrow h:=0;\ t'=t+h'-h$$

$$t'=t+h'-h \Leftarrow II \lhd Lh=x \rhd (h:=h+1;\ t:=t+1;\ t'=t+h'-h)$$

Is this acceptable as a time bound? It gives us no indication of how long to wait for a result. On the other hand, there is nothing more to say about the execution time. The defect is in the given information: that $x$ occurs somewhere, with no indication where.

## 9 Conclusion

When I began programming, I put my program, punched onto a deck of cards, in the "in" basket; hours later, the computer operator fed it into the computer, and put the output in the "out" basket, where I retrieved it. Computing involved an initial input and a final output, with no possibility of interaction. A "total correctness" theory is based on this out-of-date paradigm: without interaction, termination is essential. With the addition of interactive communication, nonterminating computations can be useful, so a semantics that does not insist on termination is useful. Furthermore, for some programs, for some inputs, we might well want to guarantee nontermination, which a "total correctness" formalism does not do. The operating system, even when I began programming, was an interacting, nonterminating computation. These days, every program I use terminates its execution when I click on "quit". Of course, each response to me must be a terminating computation; more than that, each response must come within the limit of my patience.

Throughout this paper, I have used annoying quotation marks around "total correctness" in order to provide some protection against the appeal of the phrase. It sounds like something very desirable, but it's a bad deal. It requires a complicated semantics of loops (either limit of a sequence of approximations, or least fixpoint) that is not easily used in proofs. To prove termination, you must do all the work of

finding time bounds, but without the reward. And you must prove termination before you can conclude anything about results or time bounds. And when you have proven termination, you have proven something worthless, because no observation of a computation can falsify it (nontermination is unobservable). It is time to retire the concept of "total correctness", and to terminate our obsession with termination.

## 10 References


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

1 P.Caspi, N.Halbwachs, D.Pilaud, J.A.Plaice: LUSTRE: a Declarative Language for Programming Synchronous Systems, 14th ACM Symposium on Principles of Programming Languages p.178-189, 1987

*2* E.W.Dijkstra: *a Discipline of Programming*, Prentice-Hall, 1976

3 I.J.Hayes: Reasoning about Real-Time Repetitions, Terminating and Nonterminating, *Science of Computer Programming* v.43 n.2-3 p.161-192, 2002

4 E.C.R.Hehner, A.M.Gravell: Refinement Semantics and Loop Rules, FM'99 World Congress on Formal Methods, Toulouse, LNCS 1709 p.1497-1510, 1999

5 E.C.R.Hehner: *a Practical Theory of Programming*, first edition Springer 1993, current edition www.cs.utoronto.ca/~hehner/aPToP

6 E.C.R.Hehner: Probabilistic Predicative Programming, Mathematics of Program Construction, Stirling Scotland, 2004 July 12-14, and Springer LNCS v.3125 p.169-185, 2004

7 E.C.R.Hehner: Specifications, Programs, and Total Correctness, *Science of Computer Programming*, v.34 p.191-205, 1999

8 E.C.R.Hehner: Termination is Timing, International Conference on Mathematics of Program Construction, Enschede, The Netherlands, 1989 June (opening address, invited); chapter in van de Snepscheut(ed.): *Mathematics of Program Construction*, Springer LNCS v.375 p.36-47, 1989

9 E.C.R.Hehner: Real-Time Programming, *Information Processing Letters* v.30 p.51-56, 1989 January 16

10 E.C.R.Hehner, A.J.Malton: Termination Conventions and Comparative Semantics, *Acta Informatica* v.25 n.1 p.1-14, 1988 January

11 E.C.R.Hehner: Predicative Programming, *Communications ACM* v.27 n.2 p.134-151, 1984 February

12 E.C.R.Hehner: *the Logic of Programming*, Prentice-Hall International Series in Computer Science (ed. C.A.R.Hoare), London, 1984

13 E.C.R.Hehner, C.A.R.Hoare: a More Complete Model of Communicating Processes, University of Toronto Technical Report CSRG-134, 1981 September, and *Theoretical Computer Science* v.26 p.105-120, 1983 September

14 E.C.R.Hehner: **do** considered **od**: a Contribution to the Programming Calculus, University of Toronto Technical Report CSRG-75, 1976 November, and *Acta Informatica* v.11 p.287-304, 1979

15 C.A.R.Hoare, J.He: Unifying Theories of Programming, Prentice-Hall International Series in Computer Science (ed. C.A.R.Hoare), London, 1998

16 C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *Communications ACM* v.12 n.10 p.576-580, 583, 1969 October

17 C.A.R.Hoare: "Proof of Correctness of Data Representations", *Acta Informatica* v.1 n.4 p.271-282, 1972

18 C.C.Morgan, A.K.McIver, K.Seidel, J.W.Sanders: "Probabilistic Predicate Transformers", *ACM Transactions on Programming Languages and Systems* v.18 n.3 p.325-353, 1996 May