Termination is Timing

Eric C.R. Hehner
Department of Computer Science, University of Toronto, Toronto M5S 1A4 Canada

To be presented at the International Conference on the Mathematics of Program Construction, Enschede, the Netherlands, 1989 June 26.

Summary Termination is treated as a special case of timing, with the result that the logic of programming is simplified and generalized.

Introduction

Our formalism for the description of computation is divided into two parts: logic and timing. The logic is concerned with the question: what results do we get? The timing is concerned with the question: when do we get them? One possible answer to the latter question is: never, or as we may say, at time infinity. Thus we place termination within the timing part of our formalism. Nontermination is just the extreme case of taking a long time.

Specification

For now we ignore timing (and therefore also termination). We consider a specification of computer behavior to be a predicate in the initial values x, y, ... and final values x', y', ... of some variables. From any initial state s = [x, y, ...] , a computation satisfies a specification by delivering a final state s' = [x', y', ...] that satisfies the predicate. A specification P is called *implementable* iff for every initial state s there is a satisfactory final state s' :

$$\forall s\ \exists s'\ P$$

For our specification language we will not be definitive or restrictive; we allow any well-defined predicate notations. To them we add the following four.

ok	=	$s'=s$
	=	$x'=x \wedge y'=y \wedge \ldots$
x:= e	=	ok(subst e for x)
	=	$x'=e \wedge y'=y \wedge \ldots$
if b **then** P **else** Q	=	$b \wedge P \vee \neg b \wedge Q$
	=	$(b \Rightarrow P) \wedge (\neg b \Rightarrow Q)$
P; Q	=	$\exists s''$ P(subst s'' for s') $\wedge$ Q(subst s'' for s)

The notation ok is the identity relation between pre- and poststate; it specifies that the final values of all variables equal the corresponding initial values (it is sometimes called skip). In the assignment notation, x is any variable and e is any expression (we are ignoring questions of type and initialization). The **if** notation requires a boolean expression b and two specifications P and Q , and forms a new specification. Finally, the sequential composition P;Q is just the relational composition of specifications P and Q.

Refinement

Specification P is *refined* by specification Q iff all computer behavior satisfying Q also satisfies P . We denote this by P:: Q and define it formally as

$$P :: Q \quad = \quad \forall s \, \forall s' \; P \Leftarrow Q$$

Thus refinement simply means finding another specification that is everywhere equivalent or stronger.

> *Aside.* The *precondition* for P to be refined by Q is $\forall s' \; P \Leftarrow Q$. The *postcondition* for P to be refined by Q is $\forall s \; P \Leftarrow Q$. With these definitions we can relate our formalism to those formalisms based on pre- and postconditions. But we do not pursue that here. *End of aside.*

Refinement is a partial ordering of specifications. A function from specifications to specifications is called *monotonic* if it respects the refinement ordering. Thus, trivially, we have the

> Stepwise Refinement Theorem (refinement by steps): If f is a monotonic function on specifications, and P:: Q , then fP:: fQ .

Another simple and very useful theorem is the

> Partwise Refinement Theorem (refinement by parts): If f is a monotonic function on specifications, and P:: fP and Q:: fQ , then $(P \wedge Q) ::$ f$(P \wedge Q)$.

Programs

A program is a specification of computer behavior; it is therefore a predicate in the initial and final state. Not every specification is a program. A program is an "implemented" specification, one that a computer can execute. To be so, it must be written in a restricted notation. Let us take the following as our programming notations.

(a) ok is a program.

(b) If x is any variable and e is an "implemented" expression then x:= e is a program.

(c) If b is an "implemented" boolean expression and P and Q are programs, then **if** b **then** P **else** Q is a program.

(d) If P and Q are programs then P;Q is a program.

(e) If P is an implementable specification and Q is a program such that P:: Q then P is a program.

In (b) and (c) we have not stated which expressions are "implemented"; that set will vary from one implementation to another. Likewise the programming notations (a) to (d) are not to be considered definitive, but only an example. Part (e) states that any implementable specification P is a program if a program Q is provided such that P:: Q . To execute P , just execute Q . The refinement acts as a procedure declaration; P acts as the procedure name, and Q as the procedure body; use of the name P acts as a call. Recursion is allowed; in part (e) we may use P as a program in order to obtain program Q.

Here is an example. Let x be an integer variable. The specification x'=0 says that the final value of variable x is zero. It becomes a program by refining it, which can be done in many ways. Here is one.

x'=0:: **if** x=0 **then** ok **else** (x:= x-1; x'=0)

In standard predicate notations, this refinement is

$$\forall x\ \forall x'\ \ x'=0 \ \Leftarrow\ x=0 \wedge x'=x \ \vee\ x\neq 0 \wedge \exists x''\ x''=x-1 \wedge x'=0$$

which is easily proven. (Helpful fact: x:= e; P = P(subst e for x).)

Timing

We have already seen the semantic formalism: it is similar to [2] but simplified by the omission of any consideration of termination. We now upgrade it from "partial correctness" to "total correctness" by adding a time variable. We do not change the formalism at all; the time variable is treated just like any other variable, as part of the state. Its interpretation as time is justified by the way we use it.

Let t be the initial time, i.e. the time at the start of execution; let t' be the final time, i.e. the time at the end of execution. To allow for nontermination we take the type of time to be a number system extended with ∞ . The number system we extend can be the naturals, or the integers, or the rationals, or the reals. The number ∞ is maximum

$$\forall t\ \ t\leq\infty$$

and it absorbs any addition

$$\infty + 1 = \infty$$

If t is time, it cannot decrease, therefore a specification P with time is implementable iff

$$\forall s\ \exists s'\ P \wedge t' \geq t$$

Time increases as a program is executed. There are many ways to measure time. We present just two: real time, and recursive time.

Real Time

Real time has the advantage of measuring the real execution time, and is useful in real-time programming. It has the disadvantage of requiring intimate knowledge of the implementation (machine and compiler).

To obtain the real execution time of a program, modify it as follows.

- Replace each assignment $x := e$ by
  $t := t+u;\ x := e$
  where u is the time required to evaluate and store e.
- Replace each conditional **if** b **then** P **else** Q by
  $t := t+v;$ **if** b **then** P **else** Q
  where v is the time required to evaluate b and branch.
- Replace each call P by
  $t := t+w;\ P$
  where w is the time required for the call and return. For a call that is implemented "in-line", this time will be zero. For a "tail-recursion", it may be just the time for a branch. In general, it will be the time required to push a return address onto a stack and branch, plus the time to pop the return address and branch back.
- Each refined specification can include time. For example, let f be a function of the initial state s. Then
  $t' = t + fs$
  specifies that fs is the execution time,
  $t' \leq t + fs$
  specifies that fs is an upper bound on the execution time, and
  $t' \geq t + fs$
  specifies that fs is a lower bound on the execution time.

In our earlier example, suppose that the conditional, the assignment, and the call each take time 1. The refinement becomes

P:: $t := t+1;$ **if** $x=0$ **then** ok **else** $(t := t+1;\ x := x-1;\ t := t+1;\ P)$

which is a theorem when

$$P \ = \ x'=0 \ \wedge \ (x \geq 0 \Rightarrow t'=t+3x+1) \ \wedge \ (x<0 \Rightarrow t'=\infty)$$

Execution of this program always sets x to zero; when x starts with a nonnegative value, it takes time $3x+1$ to do so; when x starts with a negative value, it takes infinite time. We shall see how to prove a similar example in a moment.

Recursive Time

To free ourselves from having to know implementation details, we allow any arbitrary scheme for inserting time increments t:= t+u into programs. Each scheme defines a new measure of time. In the recursive time measure,

- Each recursive call costs time 1.
- All else is free.

This measure neglects the time for "straight-line" and "branching" programs, charging only for loops.

In the recursive measure, our earlier example becomes

P:: **if** x=0 **then** ok **else** (x:= x-1; t:= t+1; P)

which is a theorem when

$P = x'=0 \wedge (x\geq 0 \Rightarrow t'=t+x) \wedge (x<0 \Rightarrow t'=\infty)$

As a second example, consider the refinement

Q:: **if** x=1 **then** ok
else if even x **then** (x:= x/2; t:= t+1; Q)
else (x:= (x-1)/2; t:= t+1; Q)

where

$Q = x'=1 \wedge (x\geq 1 \Rightarrow t'\leq t+\text{lb } x) \wedge (x<1 \Rightarrow t'=\infty)$

Execution of this program always sets x to one; when x starts with a positive value, it takes logarithmic time (lb = binary logarithm); when x starts nonpositive, it takes infinite time. The proof breaks into nine pieces: thanks to the theorem of refinement by parts, it is sufficient to verify the three conjuncts of Q separately; and for each there are three cases in the refinement. In detail, $\forall x\ \forall x'\ \forall t\ \forall t'$

$x=1 \wedge x'=x \wedge t'=t \Rightarrow x'=1$

$x\neq 1 \wedge \text{even } x \wedge x'=1 \Rightarrow x'=1$

$x\neq 1 \wedge \text{odd } x \wedge x'=1 \Rightarrow x'=1$

$x=1 \wedge x'=x \wedge t'=t \Rightarrow (x\geq 1 \Rightarrow t'\leq t+\text{lb } x)$

$x\neq 1 \wedge \text{even } x \wedge (x/2 \geq 1 \Rightarrow t'\leq t+1+\text{lb}(x/2)) \Rightarrow (x\geq 1 \Rightarrow t'\leq t+\text{lb } x)$

$x\neq 1 \wedge \text{odd } x \wedge ((x-1)/2 \geq 1 \Rightarrow t'\leq t+1+\text{lb}((x-1)/2)) \Rightarrow (x\geq 1 \Rightarrow t'\leq t+\text{lb } x)$

$x=1 \wedge x'=x \wedge t'=t \Rightarrow (x<1 \Rightarrow t'=\infty)$

$x\neq 1 \wedge \text{even } x \wedge (x/2 < 1 \Rightarrow t'=\infty) \Rightarrow (x<1 \Rightarrow t'=\infty)$

$x\neq 1 \wedge \text{odd } x \wedge ((x-1)/2 < 1 \Rightarrow t'=\infty) \Rightarrow (x<1 \Rightarrow t'=\infty)$

Each is an easy exercise, which we leave to the reader. Note that the use of recursion does not require the proof to be an induction.

Variant

A standard way to prove termination is to find a variant (or bound function) for each loop. A variant is an expression over a well-founded set whose value decreases with each iteration. When the well-founded set is the natural numbers (a common choice), a variant is exactly a time bound according to the recursive time measure. We shall see in the section titled "A Void Obligation" why it is essential to recognize that a variant is a time bound, and to conclude that a computation terminates within some bound, rather than to conclude merely that it terminates.

Another common choice of well-founded set is based on lexicographic ordering. For any given program, the lexicographic ordering used to prove termination can be translated into a numeric ordering, and be seen as a time bound. To illustrate, suppose, for some program, that the pair [n, m] of naturals is decreased as follows: [n, m+1] decreases to [n, m] , and [n+1, 0] decreases to [n, fn] . Then we consider [n, m] as expressing a natural number, defined as follows.

$$[0, 0] = 0$$
$$[n, m+1] = [n, m] + 1$$
$$[n+1, 0] = [n, fn] + 1$$

Well-founded sets are unnecessary in our formalism; time can be an extended integer, rational, or real. Consider the following infinite loop.

R:: x:= x+1; R

We can, if we wish, use a measure of time in which each iteration costs half the time of the previous iteration.

R:: x:= x+1; t:= t+2^{-x}; R

is a theorem when R = t'=t+2^{-x} . The theory correctly tells us that the infinite iteration takes finite time. We are not advocating this time measure; we are simply showing the generality of the theory.

A Void Obligation

A customer arrives with the specification

(a) x'=2

He evidently wants a computation in which variable x has final value 2. I provide it in the usual way: I refine his specification by a program and execute it. The refinement I choose is

x'=2:: x'=2

and execution begins. The customer waits for his result, and after a time becomes impatient. I tell him to wait longer. After more time has passed, the customer sees the weakness in his specification and decides to strengthen it. He wants a computation that takes finite time, and specifies

it thus:

(b) $x'=2 \wedge t'<\infty$

He says he does not care how long it takes, except that it must not take forever. (We must reject (b) as unimplementable since (b) $\wedge$ $t' \geq t$ is unsatisfiable for $t=\infty$, but for the sake of argument let us proceed.) I can refine (b) with exactly the same construction as before! Including recursive time, my refinement is

$x'=2 \wedge t'<\infty$:: $t:= t+1$; $x'=2 \wedge t'<\infty$

Execution begins. When the customer becomes restless I again tell him to wait longer. After a while he decides to change his specification again:

(c) $x'=2 \wedge t'=t$

He not only wants his result in finite time, he wants it instantly. Now I must abandon my recursive construction, and I refine

$x'=2 \wedge t'=t$:: $x:= 2$

Execution provides the customer with the desired result at the desired time.

Under specification (a) the customer is entitled to complain about a computation if and only if it terminates in a state in which $x' \neq 2$. Under specification (b) he can complain about a computation if it delivers a final state in which $x' \neq 2$ or it takes forever. But of course there is never a time when he can complain that a computation has taken forever, so the circumstances in which he can complain that specification (b) has not been met are exactly the same as for specification (a). It is therefore entirely appropriate that our theory allows the same refinement constructions for (b) as for (a). Specification (c) gives a time bound, therefore more circumstances in which to complain, therefore fewer refinements.

I have argued elsewhere [3] that termination by itself is meaningless. Dijkstra has termed it a "void obligation" [1]. One might suggest that a time bound of a million years is also a void obligation, but the distinction is one of principle, not practicality. A time bound is a line that divides shorter computations from longer ones. There is no line dividing finite computations from infinite ones.

In the predicate transformer theory, the formula wp(S, true) is widely quoted as being the precondition for termination of (computations described by) program S. It is not. It is the precondition for the existence of a time bound [0]. To illustrate, consider S to be

do $x>0 \rightarrow x:= x-1$
[] $x<0 \rightarrow$ "$x \geq 0$"
od

where "$x \geq 0$" specifies that x takes a natural value. Even if we allow

wp("$x \geq 0$", true) = true

we find

wp(S, true) = $0 \le x$

In the theory presented in this paper, using the recursive time measure, we have

S:: **if** x=0 **then** ok
else if x>0 **then** (x:= x-1; t:= t+1; S)
else ($x' \ge 0 \wedge t'=t$; t:= t+1; S)

when

$S = x'=0 \wedge (x \ge 0 \Rightarrow t'=t+x) \wedge (x<0 \Rightarrow t'>t)$

When $x<0$ the most we can say about execution time is that it is positive.

Recursive Constructions

We have not introduced a loop programming notation; instead we have used recursive refinement. When convenient, we can refine "in-place". For example,

x:= 5;
(y'>x:: y:= x+1);
z:= y

The middle line means what its left side says, hence the three lines together are equal to

$y' = z' > 5$

Although we can see how $y'>x$ is refined, we are not allowed to use that information and conclude $z'=6$.

In-place refinement can, of course, be recursive, and thus serve as a loop. For example,

$x \ge 0 \Rightarrow y'=x! \wedge t'=t+x$::
y:= 1;
($x \ge 0 \Rightarrow y'=y \times x! \wedge t'=t+x$::
if x=0 **then** ok
else (y:= y×x; x:= x-1; t:= t+1;
$x \ge 0 \Rightarrow y'=y \times x! \wedge t'=t+x$))

A popular way to define the formal semantics of a loop construct is as a least fixed-point. An appropriate syntax would be

identifier = program

where the identifier can be used within the program as a recursive call. (A fixed-point is a solution to such an equation, considering the identifier as the unknown. The least fixed-point is the weakest solution.) There are two difficulties.

We defined sequential composition as a connective between arbitrary specifications, not just programs. This is essential for programming in steps: we must be able to verify that P:: Q;R without referring to the

refinements of Q and R . Similarly **if** applies to arbitrary specifications. For the same reason, we must define a loop construct, if we choose to have one, for all specifications:

identifier = specification

Unfortunately, such equations do not always have solutions. Fortunately, we are not really interested in equality, but only in refinement. So we propose the syntax

identifier:: specification

A solution to a refinement as called a pre-fixed-point. Since true is always a solution to a refinement, a pre-fixed-point always exists. But obviously we cannot be satisfied with the least pre-fixed-point as the semantics. The greatest (strongest) pre-fixed-point cannot be used either because it is often unimplementable, even when the specification is a program. How about the greatest implementable pre-fixed-point? Unfortunately, greatest implementable pre-fixed-points are not unique, even when the specification is a program.

The second difficulty is that a least fixed-point, or greatest implementable pre-fixed-point, is not always constructible. (With a constructive bias, I would complain that it does does not always exist.) Even when it is constructible, the construction process is problematic. One forms a (Kleene) sequence of predicates, then one takes the limit of the sequence as the solution. (The sequence starts with P_0 = true (without timing) or $P_0 = t' \geq t$ (with timing). Then P_{k+1} is formed by substituting P_k into the loop body in place of the unknown identifier.) Finding the "general member" requires an educated guess (induction hypothesis) and induction. The limit may not be expressible in the specification language. Due to discontinuity, the limit may not be a solution to the original problem. (It has been suggested [0] that the construction process should pass over any discontinuity, using the limit from one phase as the starting point for the next.)

In our approach, we ask a programmer to specify what he intends his loop to accomplish, and then to provide a refinement. We neatly avoid the Kleene sequence, the induction, the limit, and the continuity problem. Of course, these problems do not disappear, but we prefer to avoid them when possible.

Communication Timing

To our repertoire of programming notations we now add input c?x , output c!e , and parallel composition P||Q . Just as assignment is a predicate, so are input and output. Just as sequential composition is a predicate connective, so is parallel composition. We do not present the logic of concurrency and communication in this paper, but content ourselves with their timing. We move all consideration of deadlock and livelock from the logic to the timing,

and that makes as great a simplification as did the movement of loop termination from the logic to the timing.

Parallel processes require separate timing variables. The time for the parallel composition is the maximum of the individual process times. Thus

```
t0:= t;  t1:= t;
( P0 (using t0)
|| P1 (using t1) );
t:= max t0 t1
```

is the general scheme.

Here is an example of two communicating processes.

```
P; c!0; d?y; R  ||  c?x; Q; d!1
```

The first process begins with a computation described by P, and then outputs a 0 on channel c. The second process begins with an input on channel c. Even if we decide that communication is free, we must still account for the time spent waiting for input. We therefore modify the program by placing a time increment before each input. For example, if

```
t0'=t0+p::  P
t1'=t1+q::  Q
```

then we modify the program as follows:

```
P; c!0; t0:=t0+q; d?y; R  ||  t1:=t1+p; c?x; Q; d!1
```

In general, the time increments in each process depend on the computation in the others.

It is sometimes reasonable to neglect the time required for a communication, as in the previous example. But it is not reasonable to neglect communication time when there are communication loops. Here is a simple deadlock to illustrate the point.

```
c?x; d!2 || d?y; c!3
```

The logic (which we did not present) tells us that this program refines $x'=3 \wedge y'=2$, but now we investigate "when". Inserting a wait before each input yields

```
t0:= t0+u; c?x; d!2  ||  t1:= t1+v; d?y; c!3
```

On the left, input is awaited for time u; then input is received and output is sent. We assign no time to the acts of input and output, so the output is sent at time $t0+u$. But we charge one time unit for transit, hence the output is available as input on the right after waiting time $v = u+1$. By symmetry, we can also say $u = v+1$. This pair of equations has a unique solution: $u=\infty$, $v=\infty$. The theory tells us that the wait is infinite.

If we had not charged for communication transit time, we would be solving $u=v$, and we would have to conclude that the communication could happen at any time. Assume that, after waiting time 100, an input is received on the left. Then the output can be sent at that same time (since we charge zero for

the acts of input and output), and received on the right at that same time. And then the right side can send its output to the left at that same time, which justifies our original assumption. Of course, $u=\infty$, $v=\infty$ is also a solution. This "spontaneous" communication is unphysical, so communication loops, like refinement loops, should always include a charge for time.

Livelock is an infinite communication loop on a local (hidden) channel. Without a time variable, our logic treats livelock as though it were ok . With a time variable we find that livelock leaves all other variables unchanged (like ok) but takes infinite time (like deadlock). The infinite time results from the infinite loop (recursive measure), even if communication is free.

Time Dependence

Our examples have used the time variable as a ghost, or auxiliary variable, never affecting the course of a computation. But it can be used to affect the computation. Both

x:= t

and

if t < 1989:06:26:09:30:000 **then** ... **else** ...

are allowed, with no harm to the theory. We can look at the clock, but not reset it arbitrarily; all clock changes must correspond to the passage of time.

We may occasionally want to specify the passage of time. For example, we may want the computation to "wait until time w". Formally, this is

t:= max t w

This specification is easily refined

t:= max t w::
  if t≥w **then** ok **else** (t:= t+1; t:= max t w)

and we obtain a busy-wait loop. (For simplicity we have considered time to be integer-valued, and used the recursive time measure. For practicality perhaps we should use real-values and the real-time measure. The change is an easy exercise.)

Conclusion

We have a simple form of specification: a predicate in the initial and final values of some variables. And we have a simple notion of refinement: universally quantified implication. Without a time variable (or other termination indicator [3]), our logic would be more complicated. For

example, sequential composition would be

P;Q = **if** P requires termination
then relational composition of P and Q
else P

in order to ensure "strictness" (we cannot have an infinite computation first, and then something more). But with a time variable, P;Q is just relational composition, and strictness is just the fact that $\infty + t = \infty$. Without time, input would be

c?x = **if** there is or will be a communication on channel c
then assign it to x and advance the cursor
else loop forever

With time, input is just an assignment and cursor advance, perhaps at time ∞.

The mathematics we need is just simple logic and arithmetic, with no special ordering or induction rules. In effect, our computational inductions are buried in the ordinary arithmetic on the time variable. We find it simpler to treat termination as a part of timing, and we find that timing is more easily understood and more useful than termination alone.

Acknowledgements

Je veux remercier mes amis à Grenoble, Nicolas Halbwachs, Daniel Pilaud, et John Plaice, qui m'ont enseigné que la récursion peut être contrôlée par le tic-tac d'une horloge. J'apprends lentement, mais j'apprends. I also thank IFIP Working Group 2.3, Theo Norvell, and Andrew Malton for criticism. Support is provided by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

[0] H.J. Boom: "A Weaker Precondition for Loops", ACM TOPLAS v.4 n.4 p.668-677 (1982)

[1] E.W. Dijkstra: "Position Paper on Fairness", EWD1013 (October 1987)

[2] E.C.R. Hehner, L.E. Gupta, A.J. Malton: "Predicative Methodology", Acta Informatica v.23 p.487-505 (1986)

[3] E.C.R. Hehner, A.J. Malton: "Termination Conventions and Comparative Semantics", Acta Informatica, v.25, n.1, p.1-14 (1988)