Science of Computer Programming 14 (1990) 133–158
North-Holland

A PRACTICAL THEORY OF PROGRAMMING

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

Received January 1990

Abstract. Programs are predicates, programming is proving, and termination is timing.

1. 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.

We shall begin with a very simple model of computation. There are some observable quantities, called variables x, y, ..., each with some domain. Our input to a computation is to provide initial values for the variables. After a time, the output from the computation is the final values of the variables. Later we shall add communication during the course of a computation, but at first we shall consider only an initial input and a final output. Also, for exposition, we start by ignoring timing, and therefore also termination.

2. Specifications

We consider a specification of computer behavior to be a predicate in the initial values $x, y, \ldots$ and final values $x', y', \ldots$ of some variables (we make no typographic distinction between a variable and its initial value). From any initial state $\sigma = [x, y, \ldots]$, a computation satisfies a specification by delivering a final state $\sigma' = [x', y', \ldots]$ that satisfies the predicate. Here is an example. Suppose there are two variables x and y each with domain *int*. Then

$$(x' = x + 1) \wedge (y' = y)$$

specifies the behavior of a computer that increases the value of x by 1 and leaves y unchanged.

Suppose we have an implementation of specification S. We provide the input σ; the computer provides an output σ' to satisfy S. For a specification to be implementable, then, there must be at least one satisfactory output for each input: A specification S is called *implementable* iff $\forall\sigma.\exists\sigma'.S$.

In the same variables, here is a second specification:

$$x' > x$$

This specification is satisfied by a computation that increases x by any amount; it may leave y unchanged or may change it to any integer. The first specification is deterministic, and the second nondeterministic, for each initial state.

At one extreme, we have the specification *true*; it is the easiest specification to implement because every implementation satisfies it. At the other extreme is the specification *false*, which is not satisfied by any implementation. But *false* is not the only unimplementable specification. Here is another:

$$(x \geq 0) \wedge (y' = 0)$$

If the initial value of x is nonnegative, the specification can be satisfied by setting variable y to 0. But if the initial value of x is negative, there is no way to satisfy the specification. Perhaps the specifier has no intention of providing a negative input, but to the implementer, every input is a possibility. The specifier should have written

$$(x \geq 0) \Rightarrow (y' = 0)$$

For nonnegative initial x, this specification still requires variable y to be assigned 0. If we never intend to provide a negative value for x, then we do not care what would happen if we did. That is what this specification says: for negative x any result is satisfactory.

For our specification language we will not be definitive or restrictive; we allow any well understood predicate notations. Often this will include notations from the application area. When it helps to make a specification clearer and more understandable, a new notation may be invented (and defined) on the spot. In addition, we provide four more specification notations:

$$\begin{aligned}
&ok \\
&= (\sigma' = \sigma) \\
&= (x' = x) \wedge (y' = y) \wedge \cdots \\
&(x := e) \\
&= (\text{substitute } e \text{ for } x \text{ in } ok) \\
&= (x' = e) \wedge (y' = y) \wedge \cdots \\
&(\textbf{if } b \textbf{ then } S \textbf{ else } R) \\
&= (b \wedge S) \vee (\neg b \wedge R) \\
&= (b \Rightarrow S) \wedge (\neg b \Rightarrow R) \\
&(S; R) \\
&= \exists\sigma''.(\text{substitute } \sigma'' \text{ for } \sigma' \text{ in } S) \wedge (\text{substitute } \sigma'' \text{ for } \sigma \text{ in } R)
\end{aligned}$$

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 satisfied by a machine that does nothing. In the assignment notation, e is any expression. For example,

$$(x := x + y)$$
$$= (x' = x + y) \wedge (y' = y)$$

specifies that the final value of x should be the sum of the initial values of x and y, and the value of y should be unchanged. The **if** and semi-colon notations combine specifications to make a new specification. They apply to all specifications, not just implementable specifications. They are just logical connectives, like $\wedge$ and $\vee$. But they have the nice property that if their operands are implementable, so is the result. The specification **if** b **then** S **else** R can be implemented by a computer that behaves according to either S or R depending on the initial value of b. The specification $S; R$ is the relational composition (or relational product) of S and R. It can be implemented by a computer that first behaves according to S, then behaves according to R, with the final values from S serving as initial values for R. It is therefore sequential composition.

Notational note

The identity relation *ok* is called *skip* by some. We have used *ok* because it is shorter, less active, and not in conflict with usage in other programming languages. Our operator precedence is as follows:

0	(juxtaposition)	index or argument left-to-right
1	$+ \; - \; \neg$	prefix
2	$\times \; / \; \wedge$	infix /left-to-right
3	$+ \; - \; \wedge$	infix
4	$\forall \; \exists$	right-to-left
5	$= \; \neq \; < \; > \; \leq \; \geq \; \Leftarrow \; \Rightarrow$	infix continuing
6	$:= \; !$ **if-then-else**	right-to-left
7	;	infix
8	$\|\|$	infix
9	::	infix

On level 5, the operators can be used in a continuing fashion. This means, for example, that $a = b = c$ is neither grouped to the left nor grouped to the right, but means $(a = b) \wedge (b = c)$.

3. Refinement

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

$$\begin{aligned}&(S :: R)\\ &= \forall\sigma.\forall\sigma'.(S \Leftarrow R)\end{aligned}$$

Thus refinement simply means finding another specification that is everywhere equal or stronger. Here are two examples of refinement:

$$x' > x :: (x' = x+1) \wedge (y' = y)$$
$$(x' = x+1) \wedge (y' = y) :: x := x+1$$

In each case, the left-hand side is implied by the right-hand side for all initial and final values of all variables.

The *precondition* for S to be refined by R is $\forall\sigma'.(S \Leftarrow R)$.

The *postcondition* for S to be refined by R is $\forall\sigma.(S \Leftarrow R)$.

For example, although $x' > 5$ is not refined by $x := x+1$, we can calculate (in one integer variable):

$$\begin{aligned}&\text{(the precondition for } x' > 5 \text{ to be refined by } x := x+1)\\ &= \forall x'.((x' > 5) \Leftarrow (x := x+1))\\ &= \forall x'.((x' > 5) \Leftarrow (x' = x+1))\\ &= (x+1 > 5) \qquad\qquad \text{One-point Law}\\ &= (x > 4)\end{aligned}$$

This means that a computation satisfying $x := x+1$ will also satisfy $x' > 5$ if and only if it starts with $x > 4$. If we are interested only in starting states such that $x > 4$, then we should weaken our specification with that antecedent, obtaining the refinement:

$$(x > 4) \Rightarrow (x' > 5) :: x := x+1$$

Precondition Law. If A is the precondition for S to be refined by R, then $A \Rightarrow S :: R$.

There is a similar story for postconditions. For example, although $x > 4$ is unimplementable in general:

$$\begin{aligned}&\text{(the postcondition for } x > 4 \text{ to be refined by } x := x+1)\\ &= \forall x.((x > 4) \Leftarrow (x := x+1))\\ &= \forall x.((x > 4) \Leftarrow (x' = x+1))\\ &= (x'-1 > 4) \qquad\qquad \text{One-point Law}\\ &= (x' > 5)\end{aligned}$$

This means that a computation satisfying $x := x+1$ will also satisfy $x > 4$ if and only if it ends with $x' > 5$. If we are interested only in final states such that $x' > 5$, then we should weaken our specification with that antecedent, obtaining the refinement:

$$(x' > 5) \Rightarrow (x > 4) :: x := x+1$$

For easier understanding, it might be better to use the contrapositive law to rewrite the specification $(x' > 5) \Rightarrow (x > 4)$ as the logically equivalent specification $(x \leq 4) \Rightarrow (x' \leq 5)$.

Postcondition Law. If A' is the postcondition for S to be refined by R, then $A' \Rightarrow S :: R$.

Let P be the specification "make x be further from zero". Formally,

$$\begin{aligned} P = {} & ((x > 0) \Rightarrow (x' > x)) \wedge \\ & ((x = 0) \Rightarrow (x' \neq 0)) \wedge \\ & ((x < 0) \Rightarrow (x' < x)) \end{aligned}$$

or equivalently,

$$P = (x' > x > 0) \vee (x' \neq x = 0) \vee (x' < x < 0)$$

Then

$$\begin{aligned} & (\text{the precondition for } P \text{ to be refined by } x := x + 1) \\ = {} & \forall x'.(P \Leftarrow (x' = x + 1)) \\ = {} & (x + 1 > x > 0) \vee (x + 1 \neq x = 0) \vee (x + 1 < x < 0) \\ = {} & (x \geq 0) \end{aligned}$$

We find that $x := x + 1$ specifies that x becomes further from zero if and only if x starts nonnegative. Also,

$$\begin{aligned} & (\text{the postcondition for } P \text{ to be refined by } x := x + 1) \\ = {} & \forall x.(P \Leftarrow (x' = x + 1)) \\ = {} & (x' > x' - 1 > 0) \vee (x' \neq x' - 1 = 0) \vee (x' < x' - 1 < 0) \\ = {} & (x' \geq 1) \end{aligned}$$

Therefore $x := x + 1$ specifies that x becomes further from zero if and only if x ends positive.

4. 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 may vary from one implementation to another, and from time to time. 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:

$$x' = 0 :: \textbf{if } x = 0 \textbf{ then } ok \textbf{ else } (x := x - 1; x' = 0)$$

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.

5. Common laws

A law is a sentence form such that all sentences of the form are theorems. In other words, it is a theorem schema. The laws we present are not axioms (postulates); they can be proved from the definitions we have given. It is essential for programming that the laws hold for all specifications, not just for programs. That way we can use them for program construction, not just for verification.

Transformation laws

The following laws hold for all specifications P, Q and R, and boolean expressions b:

$$(ok; P) = (P; ok) = P$$
$$(P; (Q; R)) = ((P; Q); R)$$
$$(\textbf{if } b \textbf{ then } P \textbf{ else } P) = P$$
$$(\textbf{if } b \textbf{ then } P \textbf{ else } Q) = (\textbf{if } \neg b \textbf{ then } Q \textbf{ else } P)$$
$$((\textbf{if } b \textbf{ then } P \textbf{ else } Q); R) = (\textbf{if } b \textbf{ then } (P; R) \textbf{ else } (Q; R))$$

Reformation laws

The following laws hold for all specifications P, Q and R, variables x, expressions e and boolean expressions b:

$(x := e; P) = (\text{for } x \text{ substitute } e \text{ in } P)$ Substitution Law
$P = (\textbf{if } b \textbf{ then } b \Rightarrow P \textbf{ else } \neg b \Rightarrow P)$ Case Law
$(P \vee Q; R \vee S) = (P; R) \vee (P; S) \vee (Q; R) \vee (Q; S)$
$(\textbf{if } b \textbf{ then } P \textbf{ else } Q) \vee R = (\textbf{if } b \textbf{ then } P \vee R \textbf{ else } Q \vee R)$
$(\textbf{if } b \textbf{ then } P \textbf{ else } Q) \wedge R = (\textbf{if } b \textbf{ then } P \wedge R \textbf{ else } Q \wedge R)$

In the Substitution Law, P must use only standard predicate notations, not programming notations. Here are some examples of the Substitution Law:

$(x := y+z; y' > x') = (y' > x')$
$(x := x+1; (y' > x) \wedge (x' > x)) = (y' > x+1) \wedge (x' > x+1)$
$(x := 1; (x = 1) \Rightarrow \exists x.(y' = 2 \times x)) = ((1 = 1) \Rightarrow \exists x.(y' = 2 \times x))$
$(x := y; (x = 1) \Rightarrow \exists y.(y' = x \times y)) = ((y = 1) \Rightarrow \exists k.(y' = y \times k))$
$(x := y+z; y' = 2 \times x) = (y' = 2 \times (y+z))$

State laws

Let P and Q be any specifications, and let A be a predicate of the prestate, and let A' be the corresponding predicate of the poststate. Then:

$A \wedge (P; Q) :: A \wedge P; Q$
$(P; Q) \wedge A' :: P; Q \wedge A'$
$A \Rightarrow (P; Q) :: A \Rightarrow P; Q$
$(P; Q) \Rightarrow A' :: P; Q \Rightarrow A'$
$P; A \wedge Q :: P \wedge A'; Q$
$P; Q :: P \wedge A'; A \Rightarrow Q$

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:

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

Another simple and very useful law is the following:

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

These two laws are so important for programming that we shall require all our program constructors to be monotonic. So far we have only two, **if** and sequential

composition, and they are both monotonic in both their specification operands. (In fact, they are pointwise monotonic.)

6. Program development

Here is an example of program development. It is not expected to convince the reader that program development is aided by a good theory; that has been demonstrated many times in the literature using Dijkstra's *wp* theory and Jones' VDM theory. The example is intended only to help the reader become familiar with the theory presented in this paper.

Let n be a natural variable. The problem is to reduce n modulo 2 using addition and subtraction. The first step is to express the desired result as clearly and as simply as possible:

$$n' = mod\ n\ 2$$

We refine it using the Case Law.

$$n' = mod\ n\ 2 ::$$
$$\textbf{if } n<2 \textbf{ then } (n<2) \Rightarrow (n' = mod\ n\ 2)$$
$$\textbf{else } (n \geq 2) \Rightarrow (n' = mod\ n\ 2)$$

We consider that we have solved the original problem, but now we have two new problems to solve. One is trivial:

$$(n<2) \Rightarrow (n' = mod\ n\ 2) :: ok$$

The other can be solved using the Substitution Law:

$$(n \geq 2) \Rightarrow (n' = mod\ n\ 2) :: n := n-2\,;\, n' = mod\ n\ 2$$

The final specification has already been refined, so we have finished programming (except for timing). Each refinement is a theorem; programming is proving.

The last refinement could have been written as

$$(n \geq 2) \Rightarrow (n' = mod\ n\ 2) ::$$
$$(even\ n' = even\ n)\,;\, n' = mod\ n\ 2$$

in order to leave room for improvement. Proceeding at a faster pace,

$$even\ n' = even\ n ::$$
$$p := 2\,;\, even\ p \Rightarrow even\ p' \wedge (even\ n' = even\ n)$$
$$even\ p \Rightarrow even\ p' \wedge (even\ n' = even\ n) ::$$
$$n := n-p\,;\, p := p+p\,;$$
$$\textbf{if } n<p \textbf{ then } ok \textbf{ else } even\ p \Rightarrow even\ p' \wedge (even\ n' = even\ n)$$

When a compiler translates a program into machine language, it treats each refined specification as just an identifier. Our program looks like:

$$A :: \textbf{if } n < 2 \textbf{ then } B \textbf{ else } C$$
$$B :: ok$$
$$C :: D ; A$$
$$D :: p := 2 ; E$$
$$E :: n := n - p ; p := p + p ; \textbf{if } n < p \textbf{ then } ok \textbf{ else } E$$

to a compiler. Because **if** and sequential composition are monotonic, a compiler can compile the calls to B, C and D inline (macro-expansion) creating:

$$A :: \textbf{if } n < 2 \textbf{ then } ok \textbf{ else } (p := 2 ; E ; A)$$

There is no need for the programmer to perform this mechanical assembly, and we need not worry about the number of refinements used. Also, the recursive calls to A and E can be translated as just branches (as can 99% of calls), so we need not worry about recursion being inefficient.

7. Timing

So far, we have talked only about the result of a computation, not about how long it takes. To talk about time, we just add a time variable. We do not change the theory at all; the time variable is treated just like any other variable, as part of the state. The state $\sigma = [t, x, y, \ldots]$ now consists of a time variable t and some space variables x, y, ... The interpretation of t as time is justified by the way we use it.

We use t for the initial time, i.e. the time at the start of execution, and t' for the final time, i.e. the time at the end of execution. To allow for nontermination we take the domain 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, i.e.,

$$\forall t.(t \leq \infty)$$

and it absorbs any addition:

$$\infty + 1 = \infty$$

Time cannot decrease, therefore a specification S with time is *implementable* iff

$$\forall \sigma . \exists \sigma' . S \wedge (t' \geq t)$$

And for every program P,

$$t' \geq t :: P$$

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

7.1. 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 the program as follows:

Step 1. Replace each assignment $x := e$ by

$$t := t+u\,;x := e$$

where u is the time required to evaluate and store e.

Step 2. Replace each conditional **if** b **then** P **else** Q by

$$t := t+v\,;\textbf{if } b \textbf{ then } P \textbf{ else } Q$$

where v is the time required to evaluate b and branch.

Step 3. 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 "inline", this time will be zero. For a call that is executed last in a refinement, it may be just the time for a branch. Sometimes 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.

Step 4. Each refined specification can include time. For example, let f be a function of the initial state σ. Then

$$t' = t+f\sigma$$

specifies that $f\sigma$ is the execution time,

$$t' \leqslant t+f\sigma$$

specifies that $f\sigma$ is an upper bound on the execution time, and

$$t' \geqslant t+f\sigma$$

specifies that $f\sigma$ is a lower bound on the execution time.

We could place the time increase after each of the programming notations instead of before. By placing it before, we make it easier to use the Substitution Law.

In an earlier section, we considered the example

$$x' = 0 :: \textbf{if } x = 0 \textbf{ then } ok \textbf{ else } (x := x-1\,;x' = 0)$$

Suppose that the conditional, the assignment, and the call each take time 1. Using P for the specification, the refinement becomes

$$P :: t := t+1\,;$$
$$\quad \textbf{if } x = 0 \textbf{ then } ok \textbf{ else } (t := t+1\,;x := x-1\,;t := t+1\,;P)$$

This is still a theorem when $P = (x' = 0)$ as before, but we can now include time in the specification. The refinement with time is a theorem when

$$\begin{aligned} P = {} & (x' = 0) \wedge \\ & ((x \geq 0) \Rightarrow (t' = t + 3 \times x + 1)) \wedge \\ & ((x < 0) \Rightarrow (t' = \infty)) \end{aligned}$$

Execution of this program always sets x to 0; when x starts with a nonnegative value, it takes time $3 \times x + 1$ to do so; when x starts with a negative value, it takes infinite time. It is strange to say that a result such as $x' = 0$ is obtained at time infinity. This is really just a way of saying that a result is never obtained. We could change our theory of programming to prevent any mention of results at time infinity, but we do not for two reasons: it would make the theory more complicated, and we will need to distinguish among infinite loops later when we introduce communications.

7.2. 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:

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

which is a theorem when

$$\begin{aligned} P = {} & (x' = 0) \wedge \\ & ((x \geq 0) \Rightarrow (t' = t + x)) \wedge \\ & ((x < 0) \Rightarrow (t' = \infty)) \end{aligned}$$

As a second example, consider the refinement

$$Q :: \textbf{if } x = 1 \textbf{ then } ok \textbf{ else } (x := div\, x\, 2; t := t + 1; Q)$$

where

$$\begin{aligned} Q = {} & (x' = 1) \wedge \\ & ((x \geq 1) \Rightarrow (t' \leq t + lb\, x)) \wedge \\ & ((x < 1) \Rightarrow (t' = \infty)) \end{aligned}$$

(lb = binary logarithm; div = division and round down). Execution of this program always sets x to 1; when x starts with a positive value, it takes logarithmic time; when x starts nonpositive, it takes infinite time. The proof breaks into six pieces:

thanks to the Partwise Refinement Law, it is sufficient to verify the three conjuncts of Q separately; and for each there are two cases in the refinement. In detail,

$$x' = 1 :: (x = 1) \wedge (x' = x) \wedge (t' = t)$$
$$x' = 1 :: (x \neq 1) \wedge (x' = 1)$$

$$(x \geq 1) \Rightarrow (t' \leq t + lb\, x) :: (x = 1) \wedge (x' = x) \wedge (t' = t)$$
$$(x \geq 1) \Rightarrow (t' \leq t + lb\, x) ::$$
$$\quad (x \neq 1) \wedge ((div\, x\, 2 \geq 1) \Rightarrow (t' \leq t + 1 + lb\, (div\, x\, 2)))$$

$$(x < 1) \Rightarrow (t' = \infty) :: (x = 1) \wedge (x' = x) \wedge (t' = t)$$
$$(x < 1) \Rightarrow (t' = \infty) :: (x \neq 1) \wedge ((div\, x\, 2 < 1) \Rightarrow (t' = \infty))$$

Each is an easy exercise, which we leave to the reader.

Our examples were of direct recursion, but recursions can also be indirect. For example, the refinement of a specification A may contain a call to B, whose refinement contains a call to C, whose refinement contains a call to A. The general rule is that in every loop of calls, at least one call must be charged at least one time unit.

7.3. 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 next section 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 theory; 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.

7.4. Termination

A customer arrives with the specification

$$x' = 2 \tag{1}$$

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, including recursive time, is:

$$x' = 2 :: t := t + 1 ; x' = 2$$

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 want a computation that finishes at a finite time, and specifies it thus:

$$(x' = 2) \wedge (t' < \infty) \tag{2}$$

We reject (2) because it is unimplementable: $(2) \wedge (t' \geq t)$ is unsatisfiable for $t = \infty$. It may seem strange to reject a specification just because it cannot be satisfied with nondecreasing time when started at time ∞. After all, we never want to start at time ∞. But consider the sequential composition $P; Q$ with P taking infinite time. Then Q starts at time ∞ (in other words, it never starts), and the theory must be good for this case too. An implementable specification must be satisfiable with nondecreasing time for all initial states, even for initial time ∞. So the customer weakens his specification a little:

$$(x' = 2) \wedge ((t < \infty) \Rightarrow (t' < \infty)) \tag{3}$$

He says he does not care how long it takes, except that it must not take forever. I can refine (3) with exactly the same construction as (1)! Including recursive time, my refinement is:

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

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

$$(x' = 2) \wedge (t' = t) \tag{4}$$

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 (1), 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 (3), he can complain about a computation if it delivers a final state in which $x' \neq 2$ or if 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 (3) has not been met are exactly the same as for specification (1). It is therefore entirely appropriate that our theory allows the same refinement constructions for (3) as for (1). Specification (4) gives a time bound, therefore more circumstances in which to complain, therefore fewer refinements.

As this example shows, it is meaningless to request or to promise results in a "finite but unbounded" time. One might suggest that a time bound of a million years is also untestable in practice, 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.

8. Local variable

The ability to declare a new variable within a local scope is so useful that it is provided by every decent programming language. A declaration may look something like this:

$$\textbf{var}\ x : T$$

where x is the variable being declared, and T, called the type, indicates what values x can be assigned. There must be a rule to say what scope the declaration has, that is, what it applies to. We shall consider that a variable declaration applies to what follows it, up to the next enclosing right parenthesis. In program theory, it is essential that each of our notations apply to all specifications, not just to programs. That way we can introduce a local variable as part of the programming process, before its scope is refined.

We can express a variable declaration together with the specification to which it applies as a predicate in the initial and final state:

$$(\textbf{var}\ x : T\,;\, P) = \exists x : T.\exists x' : T.P$$

Specification P is a predicate in the initial and final values of all global (already declared) variables plus the newly declared local variable. According to this predicate, the initial value of the variable is an arbitrary value of the type. Suppose the global variables are integer variables y and z. Then

$$\begin{aligned}&(\textbf{var}\ x : int\,;\, y := x)\\ &= \exists x : int.\exists x' : int.(x' = x) \wedge (y' = x) \wedge (z' = z)\\ &= (z' = z)\end{aligned}$$

which says that z is unchanged. Variable x is not mentioned because it is not one of the global variables, and variable y is not mentioned because its final value is

unknown. However

$$(\textbf{var}\ x : int ; y := x - x)$$
$$= (y' = 0) \wedge (z' = z)$$

In some languages, a newly declared variable has a special value called "the undefined value" which cannot participate in any expressions. To write such declarations as predicates, we introduce the elementary expression *undefined* but we do not give any axioms about it, so nothing but trivialities like *undefined* = *undefined* can be proved about it. Then

$$(\textbf{var}\ x : T ; P)$$
$$= \exists x : undefined . \exists x' : T, undefined . P$$

9. In-place refinement

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-hand side says, hence the three lines together are equal to:

$$y' = z' > 5$$

We are not allowed to use the way $y' > x$ is refined and conclude

$$(x' = 5) \wedge (y' = z' = 6).$$

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

$$(x \geqslant 0) \Rightarrow (y' = x!) \wedge (t' = t + x) ::$$
$$\quad y := 1 ;$$
$$\quad ((x \geqslant 0) \Rightarrow (y' = y \times x!) \wedge (t' = t + x) ::$$
$$\qquad \textbf{if}\ x = 0\ \textbf{then}\ ok$$
$$\qquad \textbf{else}\ (y := y \times x ; x := x - 1 ; t := t + 1 ;$$
$$\qquad\quad (x \geqslant 0) \Rightarrow (y' = y \times x!) \wedge (t' = t + x)))$$

There are two theorems to be proved here. The first is:

$$(x \geqslant 0) \Rightarrow (y' = x!) \wedge (t' = t + x) ::$$
$$\quad y := 1 ; (x \geqslant 0) \Rightarrow (y' = y \times x!) \wedge (t' = t + x)$$

Note that the body of the loop is not used in this theorem. The other is:

$$(x \geqslant 0) \Rightarrow (y' = y \times x!) \wedge (t' = t + x) ::$$
$$\quad \textbf{if}\ x = 0\ \textbf{then}\ ok$$
$$\quad \textbf{else}\ (y := y \times x ; x := x - 1 ; t := t + 1 ;$$
$$\qquad (x \geqslant 0) \Rightarrow (y' = y \times x!) \wedge (t' = t + x))$$

With in-place refinement we can indent the inner refinements and make our programs look more traditional. This is not a good reason for using them. The only advantage they offer is to save us from writing a specification one extra time.

10. Concurrency

We shall define the parallel composition $\|$ of specifications P and Q so that $P \| Q$ is satisfied by a computer that behaves according to P and the same time, in parallel, according to Q. The operands of $\|$ are called *processes*. The connective $\|$ is similar to $\wedge$ (conjunction), but weaker, so that

$$P \| Q :: P \wedge Q$$

Conjunction is not always implementable, even when both conjuncts are. For example, in variables x and y,

$$\begin{aligned}&(x := x+1) \wedge (y := y+1)\\ &= (x' = x+1) \wedge (y' = y) \wedge (x' = x) \wedge (y' = y+1)\\ &= \mathit{false}\end{aligned}$$

We shall define parallel composition so that $P \| Q$ is implementable whenever P and Q are. In particular,

$$(x := x+1 \| y := y+1) = (x' = x+1) \wedge (y' = y+1)$$

The program *ok* will be the identity for parallel composition

$$(\mathit{ok} \| P) = (P \| \mathit{ok}) = P$$

just as for sequential composition, unlike conjunction.

If we ignore time, or if we use the recursive time measure, we have:

$$(x := x+1 ; x := x-1) = \mathit{ok}$$

Therefore, according to the Law of Transparency (substitution of equals),

$$(x := x+1 ; x := x-1 \| y := x) = (\mathit{ok} \| y := x)$$

According to the right-hand side of this equation, the final value of y is the initial value of x. According to the left-hand side of the equaiton, it may seem that $y' = x+1$ should also be a possibility: the right-hand process $y := x$ may be executed in between the two assignments $x := x+1$ and $x := x-1$ in the left-hand process. We face a simple choice: give up the Law of Transparency and with it all familiar mathematics, or give up the possibility that one process may use the intermediate states of another process. We choose the latter; our processes will not communicate with each other through shared variables, but through communication channels that do not violate the Law of Transparency. We postpone communication to the next section, and in this section consider non-interacting processes.

Variable x is *independent* of specification P if

$$x' = x :: P$$

According to P, the final value of x will be its initial value. Variable x is *dependent* on specification P if it is not independent:

$$\neg(x' = x :: P)$$

This says that x might change (is not guaranteed not to change) its value under specification P. The precondition for independence

$$\forall \sigma'.((x' = x) \Leftarrow P)$$

tells us in which initial states x does not change its value under P. Its negation

$$\begin{aligned} &\neg \forall \sigma'.((x' = x) \Leftarrow P) \\ &= \exists \sigma'.(x' \neq x) \wedge P \end{aligned}$$

tells us in which initial states x might change its value under P.

For any initial state, the *dependent space* of a specification consists of all space variables that might change value. Let σ_P denote the dependent space of P, and let σ_Q denote the dependent space of Q. Then

$$(P \parallel Q) = (\exists \sigma'_Q.P) \wedge (\exists \sigma'_P.Q)$$

Parallel composition is just conjunction, except that each conjunct makes no promise about the final values of variables that might be changed by the other process.

A parallel composition can be executed by executing the processes in parallel, but each process makes its assignments to private copies of variables. Then, when both processes are finished, the final value of a variable is determined as follows: if both processes left it unchanged, it is unchanged; if one process changed it and the other left it unchanged, its final value is the changed one; if both processes changed it, its final value is arbitrary. This final rewriting of variables does not require coordination or communication between the processes; each process rewrites those variables it has changed. In the case when both processes have changed a variable, we do not even require that the final value be one of the two changed values; the rewriting may mix the bits.

In a commonly occurring special case, there is no need to copy variables. When both processes are expressed as programs, and neither assigns to any variable appearing in the other, copying and rewriting are unnecessary. For the sake of efficiency, it is tempting to make this case the definition of parallel composition. But we must define our connectives for all specifications, not just for programs. We must be able to divide a specification into processes before deciding how each process is refined by a program.

Common laws

$$(P :: Q) \Rightarrow (P \parallel R :: Q \parallel R) \qquad \text{monotonic}$$
$$(P :: Q) \Rightarrow (R \parallel P :: R \parallel Q) \qquad \text{monotonic}$$
$$(P \parallel Q) = (Q \parallel P) \qquad \text{symmetric}$$
$$(P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R) \qquad \text{associative}$$
$$(P \parallel ok) = (ok \parallel P) = P \qquad \text{identity}$$
$$(P \parallel Q \vee R) = (P \parallel Q) \vee (P \parallel R) \qquad \text{distributive}$$
$$(P \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } R)$$
$$= (\textbf{if } b \textbf{ then } (P \parallel Q) \textbf{ else } (P \parallel R)) \qquad \text{distributive}$$

Examples

Let the state consist of three integer variables x, y and z. Here are two unsurprising examples:

$$(x := z \parallel y := z) = (x' = y' = z' = z)$$

$$(x := y \parallel y := x) = (x' = y) \wedge (y' = x) \wedge (z' = z)$$

Here is a less obvious example:

$$(x := y \parallel x := z)$$
$$= ((x = y) \Rightarrow (x' = z)) \wedge ((x = z) \Rightarrow (x' = y)) \wedge (y' = y) \wedge (z' = z)$$

To see this, we must find the dependent spaces. Variable x is in the dependent space of $x := y$ when:

$$\exists \sigma'.(x' \neq x) \wedge (x := y)$$
$$= \exists x'.\exists y'.\exists z'.(x' \neq x) \wedge (x' = y) \wedge (y' = y) \wedge (z' = z)$$
$$= (x \neq y)$$

Variable y is in the dependent space of $x := y$ when

$$\exists \sigma'.(y' \neq y) \wedge (x := y)$$
$$= \exists x'.\exists y'.\exists z'.(y' \neq y) \wedge (x' = y) \wedge (y' = y) \wedge (z' = z)$$
$$= \mathit{false}$$

Variable z is in the dependent space of $x := y$ when

$$\exists \sigma'.(z' \neq z) \wedge (x := y)$$
$$= \exists x'.\exists y'.\exists z'.(z' \neq z) \wedge (x' = y) \wedge (y' = y) \wedge (z' = z)$$
$$= \mathit{false}$$

In other words, when $x = y$ the dependent space of $x := y$ is empty, i.e.,

$$(x = y) \Rightarrow (\sigma_{x:=y} = [\mathit{nil}])$$

and when $x \neq y$ the dependent space of $x := y$ is x, i.e.,

$$(x \neq y) \Rightarrow (\sigma_{x:=y} = [x])$$

Similarly for $x := z$. Thus we have four cases:

$$
\begin{aligned}
&(x := y \parallel x := z)\\
&= ((x = y = z) \Rightarrow (x := y) \wedge (x := z)) \wedge\\
&\quad ((x \neq y) \wedge (x = z) \Rightarrow (x := y) \wedge (\exists x'.(x := z))) \wedge\\
&\quad ((x = y) \wedge (x \neq z) \Rightarrow (\exists x'.(x := y)) \wedge (x := z)) \wedge\\
&\quad ((x \neq y) \wedge (x \neq z) \Rightarrow (\exists x'.(x := y)) \wedge (\exists x'.(x := z)))
\end{aligned}
$$

Simplifying, we obtain the answer stated previously.

10.1. Parallel timing

The definition of parallel composition uses dependent spaces to say which conjunct determines the final value of each variable. For each initial state, we place the time variable t in the dependent space of the process that takes longer. If neither process takes longer than the other (they take the same time), then it is in neither dependent space. With this placement of the time variable, the time for a parallel composition is the maximum of the individual process times.

11. Communication

Until now, the only input to a computation has been the initial state of the variables, and its only output has been the final state. Now we consider input and output during the course of a computation. We allow any number of named communication channels through which a computation communicates with its environment, which may be people or other computations running in parallel.

Communication on channel c is described by a list s_c called the channel script, and two extended natural variables r_c and w_c called the read and write cursors. The script is the sequence of all messages—past, present and future—that pass along the channel. The script is a constant, not a variable. At any time, the future meassages on a channel may be unknown, but they can be referred to as items in the script. The read cursor is a variable saying how many messages have been read, or input, on the channel. The write cursor is a variable saying how many messages have been written, or output, on the channel. Neither the script nor the cursors are programming notations, but they allow us to specify any desired communications. If there is only one channel, or if the channel is known from context, we may omit the subscripts on s, r and w.

Here is an example specification. It says that if the next input on channel c is even, then the next output on channel d will be 0, and otherwise it will be 1. Formally, we may write:

$$\textbf{if } even\ (s_c r_c) \textbf{ then } s_d w_d = 0 \textbf{ else } s_d w_d = 1$$

or more briefly

$$mod\ (s_c r_c)\ 2 = s_d w_d$$

We provide three programming notations for communication. Let c be a channel. Then $c?$ describes a computation that reads one input on that channel. We use the channel name c to denote the message that was last previously read on the channel. The notation $c!e$ describes a computation that writes the output message e on channel c. Here are the formal definitions (omitting the obvious subscripts):

$$c? = (r := r+1)$$

$$c = s(r-1)$$

$$c!e = (sw = e) \wedge (w := w+1)$$

The predicate *computation* is the strongest predicate that describes all computations. If we are considering time, and there is one channel c, then

$$computation = (t' \geq t) \wedge (r'_c \geq r_c) \wedge (w'_c \geq w_c)$$

A computation cannot take less than zero time, it cannot unread, and it cannot unwrite. There are similar conjuncts for other channels. If there are no channels and we are not considering time, then *computation* is the empty conjunction *true*. In general, specification S is *implementable* iff

$$\forall \sigma . \exists \sigma' . S \wedge computation$$

For every program P

$$computation :: P$$

Examples

Input numbers from channel c, and output their doubles on channel d. Formally, the specification is:

$$S = \forall n : nat . (s_d(w_d + n) = 2 \times s_c(r_c + n))$$

We cannot assume that the input and output are the first input and output ever on channels c and d. We can only ask that from now on, starting at the initial read cursor r_c and initial write cursor w_d, the outputs will be double the inputs. This specification can be refined as follows:

$$S :: c?; d!2 \times c; S$$

The proof is as follows:

$$\begin{aligned} & (c?; d!2 \times c; S) \\ = & (r_c := r_c + 1; (s_d w_d = 2 \times s_c(r_c - 1)) \wedge (w_d := w_d + 1); S) \\ = & (s_d w_d = 2 \times s_c r_c) \wedge \forall n : nat . (s_d(w_d + 1 + n) = 2 \times s_c(r_c + 1 + n)) \\ = & S \end{aligned}$$

For reasons stated earlier, parallel processes cannot communicate and cooperate through variables. They must communicate through channels. Here is a first example, though not a convincing one:

$$c!2 \parallel c?; x := c$$

The dependent space of the left-hand process is w. The dependent space of the right-hand process is r, x. This example is equal to

$$(sw = 2) \wedge (w' = w + 1) \wedge$$
$$(r' = r + 1) \wedge (x' = sr) \wedge (\text{other variables unchanged})$$

We do not know that initially $w = r$, so we cannot conclude that finally $x' = 2$. The parallel composition may be part of a sequential composition, for example,

$$c!1; (c!2 \parallel c?; x := c); c?$$

and the final value of x may be the 1 from the earlier output, with the 2 going to the later input. In order to achieve useful communication between processes, we shall have to introduce a local channel.

11.1. Local channel

Channel declaration is similar to variable declaration; it defines a new channel within some local portion of a program or specification. Each language must have rules to say what the scope of the declaration is. We shall consider that a channel declaration applies to what follows it, up to the next enclosing right parenthesis. Here is a syntax and equivalent predicate.

$$(\textbf{chan } c : T; P)$$
$$= \exists s_c : {}^*T.(\textbf{var } r_c : xnat := 0; \textbf{var } w_c : xnat := 0; P)$$

The type T says what communications are possible on this new channel. The declaration introduces a script, which is a list of items from T; it is not a variable (in the programmer's sense), but a constant of unknown value. It also introduces a read cursor with initial value 0 to say that initially there has been no input on this channel, and a write cursor with initial value 0 to say that initially there has been no output on this channel.

A local channel can be used without concurrency as a queue, or buffer. For example,

$$(\textbf{chan } c : int; c!3; c!5; c?; x := c; c?; x := x + c)$$
$$= (x := 8)$$

Here are two processes with a communication between them:

$$(\textbf{chan } c : int; (c!2 \parallel c?; x := c))$$
$$= \exists s : {}^*int.(\textbf{var } r : xnat := 0; \textbf{var } w : xnat := 0;$$
$$(sw = 2) \wedge (w' = w + 1) \wedge (r' = r + 1) \wedge (x' = sr) \wedge$$
$$(\text{other variables unchanged}))$$
$$= \exists s : {}^*int.(s0 = 2) \wedge (x' = s0) \wedge (\text{other variables unchanged})$$
$$= (x' = 2) \wedge (\text{other variables unchanged})$$
$$= (x := 2)$$

Replacing 2 by an arbitrary expression in this example, we have a general theorem equating communication on a local channel with assignment.

11.2. Communication timing

Here is an example of two communicating processes:

$$\textbf{chan}\ c : int ; \textbf{chan}\ d : int ; (P ; c!0 ; d? ; R \parallel c? ; Q ; d!1)$$

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

$$t' = t + p :: P$$
$$t' = t + q :: Q$$

then we modify the program as follows:

$$\textbf{chan}\ c : int ; \textbf{chan}\ d : int ;$$
$$(P ; c!0 ; t := t + q ; d? ; R \parallel t := t + p ; c? ; Q ; d!1)$$

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:

$$\textbf{chan}\ c : int ; \textbf{chan}\ d : int ; (c? ; d!2 \parallel d? ; c!3)$$

In the two processes, inserting a wait before each input yields

$$t := t + u ; c? ; d!2 \parallel t := t + v ; d? ; c!3$$

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 $t + u$. 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: $u = v = \infty$. 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 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-hand side can send its output to the left at that same time, which justifies our original assumption. Of course, $u = v = \infty$ is also a solution. Spontaneous simultaneous 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 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.

11.3. Synchronous communication

The communication we have described is asynchronous. For synchronous communication, we need the same script as for asynchronous communication, but only one cursor. The programming notations are defined the same way, but variables r and w become one variable. Synchronous communication timing requires a time increment before input and output because execution of an output must wait until the corresponding input can be executed simultaneously. The theory of synchronous communication is not very different from the theory of asynchronous communication, but in practice the extra delays for output mean more simultaneous equations to solve.

11.4. Time dependence

Our examples have used the time variable as a ghost, or auxiliary variable, never affecting the course of a computation. It was used as part of the theory, to prove something about the execution time. Used for that purpose only, it did not need representation in a computer. But if there is a readable clock available as a time source during a computation, it can be used to affect the computation. Both

$$x := t$$

and

if $t < 1989;06:26:09:30.000$ **then** ... **else** ...

are allowed. 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". Let us invent a notation for it, and define it formally as

wait until w
$= (t := max\ t\ w)$

This specification is easily refined. Including recursive time,

wait until w ::
  if $t \geqslant w$ **then** *ok* **else** $(t := t+1$; **wait until** $w)$

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 we should use real values and the real time measure. The change is an easy exercise.)

12. Recursive construction

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**, **var**, $\|$ and **chan** apply 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. The syntax

identifier :: *specification*

would be appropriate. A solution to a refinement is 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 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 $P_0 = \mathit{computation}$. 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.

In our approach, we ask a programmer to specify what she intends her 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.

13. Conclusion

In our theory, a program is a specification whose implementation is provided automatically. Thus programming notations can be freely mixed with other specification notations to allow a smooth transition in small steps from specification to program.

A specification is a predicate, hence a program is a predicate. This is simpler than Dijkstra's *wp*, in which a program is (or can be mapped to) a function from predicates to predicates. It is simpler than Jones' VDM in which a program is (or can be specified by) a pair of predicates. And we have a simpler notion of refinement: universally quantified implication.

Our theory is also more general than its competitors, applying to sequential and concurrent computations, communicating processes, and infinite interactive computations. For the latter purposes, it is much simpler than theories involving sets of interleaved sequences, or those involving temporal logic.

The decision to include time was made when the following facts became evident. Time is just an ordinary variable in the theory, distinguished only in that its changes of value correspond to the passage of time. Thanks to the Partwise Refinement Law, time can be ignored at first, and conjoined later. Proof of termination is essentially the discovery of a time bound, so we may as well call it that. Without a time variable (or other termination indicator), our logic would be more complicated. For example, sequential composition would be

$$\begin{aligned} P;Q \;=\; & \textbf{if } P \text{ requires termination} \\ & \textbf{then } \text{relational composition of } P \text{ and } Q \\ & \textbf{else } P \end{aligned}$$

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

$$\begin{aligned} c? \;=\; & \textbf{if } \text{there is or will be a communication on channel } c \\ & \textbf{then } \text{advance the cursor} \\ & \textbf{else } \text{loop forever} \end{aligned}$$

With time, input is just a 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.

Acknowledgement

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.