do considered od:
A Contribution to the Programming Calculus

by

Eric C.R. Hehner

Computer Systems Research Group
University of Toronto

Technical Report CSRG-75

November 1976

Abstract - The utility of the repetitive DO construct is challenged. Recursive refinement is claimed to be semantically as simple, and superior for programming ease and clarity. Some programming examples are offered to support this claim. The relation between the semantics of predicate transformers and the mathematical semantics of Scott and Strachey is presented.

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their applications. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the National Research Council of Canada.

Introduction

A major advance toward a useable programming calculus has been made by Dijkstra [1,2]. His syntactic tool is "guarded command sets", from which he constructs an alternative, or IF, statement, and a repetitive, or DO, statement. His semantic tool is "predicate transformers", which specify, for a given statement S and post-condition R, the weakest pre-condition guaranteeing that S will establish R. In this paper, we shall assume that the reader is familiar with the above.

Our purpose is to offer some constructive criticisms of Dijkstra's approach. In particular, we challenge the utility of the repetitive DO statement, and offer, in its place, the notion of recursive refinement. Before the reader flees in panic from the "sledgehammer" tactics of replacing something as simple as repetition by something as complicated as recursion, let us make our motivation plain. The semantics of DO involve a recurrence relation on the semantics of IF; that is by far the most complicated part of Dijkstra's rudimentary language. Our purpose is to avoid complication as much as possible. By contrast, we shall claim that recursive refinement introduces less semantic complication to the language. Even more important, we shall claim that programs composed using recursive refinement are simpler and clearer than programs composed of DO statements. To support this claim, we shall present some of the programming examples of [1]. It is intended that our programs be compared with those in [1]. For the reader's convenience, we include the latter in an appendix, but we suggest that the reader refer instead to [1], which contains a charming and illuminating commentary.

The Language

For our programming language, we mainly adopt Dijkstra's notations. The empty statement will be denoted by "*skip*". An assignment will be denoted by "x:=E" where x is any identifier, and E is any expression. Composition (sequencing) of statements S_1 and S_2 is denoted by "$S_1;S_2$". The alternative, or IF, statement is denoted by

$$\underline{\text{if}}\ B_1 \rightarrow SL_1 [\!] B_2 \rightarrow SL_2 [\!] \ \ldots\ [\!] B_n \rightarrow SL_n\ \underline{\text{fi}}$$

where the B_i are boolean expressions, and the SL_i are statement lists. Like Dijkstra, we shall not make use of the "*abort*" statement. Unlike Dijkstra, we shall not make use of the repetitive, or DO statement.

The programming technique known as "stepwise refinement", championed by Dijkstra [3] and others [4], has been used to great advantage in [1]. It involves the invention of a name for a portion of a program, using the name in place of the program portion, and specifying the text of the program portion elsewhere (possibly re-using the technique within the text of the program portion). We shall refer to the use of a name in place of some statements as a "call", and we shall refer to the specification of the statements as a "refinement". Thus, in Dijkstra's notation, a program may consist of the two calls

"action A"; "action B"

together with the refinements

"action A": ⟨statement list⟩

and

"action B": ⟨statement list⟩

Refinement is commonly considered to be an extra-language programming technique, as in [1]; the programmer is then required to assemble the various pieces of his program into their proper places to form the final product. We shall add

the call and refinement statements to Dijkstra's little language; we thus save the programmer from what, in many cases, is a purely clerical task, and we allow the final program to retain the intermediate design decisions. This is our only addition to the language. Since we have made it, allowing recursive calls does not make the language larger; on the contrary, disallowing them would require a special rule.

> *Note*. Although we have used the word "call", we urge the reader not to condemn our addition to the language because of some inefficient implementation of procedure calls in some other language. We do not intend to imply anything about the implementation, in particular, whether a refinement is compiled "out-of-line" with branching to and from it, or "in-line" in place of the call. We especially do not intend to imply stacking activity. We shall discuss implementation later. (*End of note*.)

Semantics

The meaning of the statements in our language may be given in either of two ways. One is called "operational semantics"; in this view, a statement is a command to change state (the state is a list of variables together with their values) from a given initial state to one of the desired final states. (As Dijkstra points out, the word "command" would have been preferable to "statement".) The operational semantics of a statement are instructions on how to execute the statement (perform the change in state). The other way of giving the meaning of a statement is called "mathematical semantics"; in this view, a statement describes a change in state (it is not a command to do anything). The description is less detailed than a complete mapping between initial and final states, for that would be equivalent to a set of instructions for changing the state. Instead, it is a mapping between sets of initial states and sets of final states. A set of states is characterized by a predicate on the program variables, thus the mathematical semantics of a statement are given as a predicate transformer.

Operational semantics are necessary for the implementation, or execution, of programs. But for programming, or understanding programs, they are too detailed. Since our interest in this paper is mainly programming, we shall shun operational semantics, and embrace mathematical semantics. In this respect, we follow Dijkstra [1] and many others [6,7,8]. Perhaps partly as a concession to our history of trying to understand programs by tracing their executions, Dijkstra gives both the operational and mathematical semantics of the statements he introduces. Unfortunately, the terminology of the text comes largely from the operational view, such as "control goes through the loop" [1, p.66], and "repetitive construct" or "repetition" [1, throughout].

Recognizing that programming is a goal-directed activity, Dijkstra gives the semantics of a statement S by a predicate transformer wp that tells us, for any post-condition R, the weakest pre-condition such that S establishes R. This is denoted by "wp(S, R)". The semantics of our chosen statements are defined as satisfying the following equations.

$$\text{wp("}skip\text{"}, R) = R$$

$$\text{wp("}x{:=}E\text{"}, R) = R_{x:=E}$$

$$\text{wp("}S_1;S_2\text{"}, R) = \text{wp}(S_1, \text{wp}(S_2, R))$$

$$\text{wp(IF}, R) = (\exists\, i{:}B_i) \text{ and } (\forall\, i{:}B_i \Rightarrow \text{wp}(SL_i, R))$$

In the above, "$R_{x:=E}$" denotes the predicate obtained from R by simultaneously changing all occurrences of "x" into "E". The existential and universal quantifiers take i over the integers in the range $1 \le i \le n$.

The call and refinement statements give us no new semantic equations. A call is given meaning by the details of its refinement. The act of naming some statements, and using the name in their place, is not semantically significant. This remains true even if some of the calls are recursive. In this case, however, the equations become recursive, raising the possibility that they may have more than one solution. To specify a particular solution, we define wp(S, R) as the strongest predicate satisfying the equations. Solving the equations may be difficult, with or without recursion; sometimes the solution may be expressed in a "closed form", and sometimes not (see the section titled "Solving the Semantic Equations"). Fortunately, as Dijkstra points out [1, p.17], we are not very interested in solving the equations. For programming, a predicate that is stronger than wp (and hence not a solution) will content us.

Problem Solving Methods

We now present three problem solving methods. The first may be called "divide and conquer". We want to establish R. If we cannot see how to do so directly, but we see that the conjunction of predicates P and Q implies R, then the refinement

"establish R":

"establish P";

"maintain P, establish Q"

will do the job. In the second call, predicate P is called an invariant. Clearly we can generalize to several subgoals.

The second problem solving method is "case analysis". Suppose we can find n predicates, B_1, B_2, ..., B_n, such that at least one must be true, and for each i, given B_i we can establish R. Our refinement is then

"establish R":

if B_1 → "given B_1, establish R"

[] B_2 → "given B_2, establish R"

[] ...

[] B_n → "given B_n, establish R"

fi

In each alternative we are, of course, free to establish subgoals and do further case analyses.

Suppose that, even with these methods, we do not see how to refine "given B_i, establish R" for some alternative. Then the third problem solving method may help. It is to delegate the task to a refinement after making some measurable progress toward the goal. "Measurable progress" is defined, as in [1], as a decrease in a monotonically decreasing integer-valued

function that is bounded below (or equivalently, an increase in a monotonically increasing integer-valued function that is bounded above). After that, the task may be delegated either to a new refinement, or to the current refinement. For example, if t is the aforementioned function, the refinement may be

"establish R":

if ...
$\quad [\!] B_i \rightarrow$ "decrease t"; "establish R"
$\quad [\!] \ldots$
fi

Having seen the method, we are now tempted to not bother with "decrease t", and to simply delegate the whole alternative to "establish R". The futility of this attempt can be seen from the equation for wp("establish R", R): the strongest solution implies <u>non</u> B_i. Thus the shortcut "works" only in alternatives that are unnecessary anyway! (For proof, and for an illustration that measurable progress is sufficient for recursion, see the section titled "Solving the Semantic Equations".) After making measurable progress, one need not even be aware of using recursion; this is fortunate, since an indirect recursion may escape notice.

The three programming techniques we have presented are not the only valuable problem solving methods, nor are they sufficient for all occasions. But they are sufficient to begin our programming examples.

Example 1

(Euclid's algorithm) [1, ch.7] Given two positive integers X and Y, print their greatest common divisor. Our program is as follows.

$x:=X$; $y:=Y$;

"print GCD(x,y)"

with the refinement

"print GCD(x,y)":

if $x=y \rightarrow print(x)$

[] $x>y \rightarrow x:=x-y$; "print GCD(x,y)"

[] $x<y \rightarrow y:=y-x$; "print GCD(x,y)"

fi

Here we have used all three problem solving methods. In the case analysis, the method of achieving the result in each alternative is independent of the methods used in the other alternatives. In the first alternative, we achieved our result directly; in the other two, we made measurable progress, then recursed.

As programmers, we may comfortably use a result that has been proved without being constantly aware of the proof's details. But for those who want to prove for each program separately that measurable progress is sufficient for recursion, we have some words of warning. The proof is, of course, an induction on the measure of progress. It is often easy to see that a recursive construct works for $n=0$, and that if it works for $n=k-1$, it will work for $n=k$. The common mistake is asking (or explaining) how it works for $n=k-1$. This mistake is made for one of two reasons: (a) failure to assume the inductive hypothesis; induction requires that we prove an implication, not the hypothesis of the implication. (b) curiosity about the implementation; an explanation of the implementation should come only after the semantics are understood, not as an explanation of the semantics. For either reason, this mistake leads to an

effort to understand by tracing, and to the poor man's induction: "If it works for $n=1$, 2, and 3, then that's good enough for me.".

To make our presentation of Euclid's algorithm more formal, we shall omit the printing; the problem is to establish

R: x=GCD(X,Y)

Our invariant is

P: $x>0$ and $y>0$ and GCD(x,y) = GCD(X,Y)

Our integer-valued monotonically decreasing function is

t: $x+y$

The invariant P implies that t is bounded below. We shall write predicates whose truth has been established as comments enclosed in braces.

$x:=X$; $y:=Y$; {P}

"maintain P, establish $x=y$" {R}

"maintain P, establish $x=y$":

if $x=y \rightarrow skip$ {P is maintained, $x=y$ is established}

[] $x>y \rightarrow x:=x-y$; {P is maintained, t is decreased}

"maintain P, establish $x=y$" {P is maintained, $x=y$ is established}

[] $x>y \rightarrow x:=y-x$; {P is maintained, t is decreased}

"maintain P, establish $x=y$" {P is maintained, $x=y$ is established}

fi

Following a DO construct, one is entitled to conclude that all guards are false. To put it more positively, one is entitled to conclude that the "missing guard" is true. The IF is more straightforward: the conclusion is established explicitly by each alternative. But then, the fact that IF has simpler semantics than DO was never in question; our point is that the recursive aspect is irrelevant to the predicate transformations.

Example 2

[1, p.57, Ex.3] Given integers $a \geq 0$ and $d > 0$ establish

$$R\colon\ 0 \leq r < d \ \underline{\text{and}}\ d \mid (a-r)$$

using only addition and subtraction. We obtain an invariant by weakening R to the more easily established

$$P\colon\ 0 \leq r \quad \underline{\text{and}}\ d \mid (a-r)$$

Our program is

$r := a$; $\{P\}$

"maintain P, establish $r < d$" $\{R\}$

At this point, using DO, one would require an inspiration as expressed by the phrase "it is hard to see how R can be established without a loop" [1 p.53]. We are unable to eliminate the need for inspiration, but, in the spirit of the programming calculus, we would like to minimise the need for it. With our method, one needs the inspiration at this point to use case analysis

"maintain P, establish $r < d$":

$\underline{\text{if}}\ r < d \rightarrow skip$

$[]\ r \geq d \rightarrow \ldots$

$\underline{\text{fi}}$

but one need not realise until the second alternative that recursion will be needed. Thus we have broken the required inspiration into two pieces. Choosing r as our measure of progress, the second alternative becomes

"maintain P, decrease r"; "maintain P, establish $r < d$"

On the other hand, our method requires that the program portion on which we recurse be named, whereas, if we use DO, there is no such necessity. We introduced the refinement as a freedom, to be used for better programming; but now that it is seen to be a necessity, it may seem to be an unfortunate burden. Perhaps so; our rebuttal is that the names add understandability. The issue is debatable.

The easiest refinement of "maintain P, decrease r" is simply "$r:=r-d$". However, following Dijkstra, we can speed up our program by introducing variable dd, and maintaining it as a multiple of d. Thus we can decrease r by multiples of d at a time.

"maintain P, decrease r":

dd:=d; {P and $d|dd$}

"maintain P and $d|dd$, decrease r"

"maintain P and $d|dd$, decrease r":

if $r<dd \rightarrow skip$

[] $r \geq dd \rightarrow r:=r-dd;\ dd:=dd+dd;$

"maintain P and $d|dd$, decrease r"

fi

We thus obtain a program that is computationally equivalent to Dijkstra's. But now, and in the examples to follow, we would like to show that our methods can do better than merely duplicate what is attainable with DO.

The major deficiency in the last refinement above is that it does not clearly achieve its purpose. It is obliged to decrease r; to see that it does so, we must see that the call occurs in a context such that $r \geq d$ and $d=dd$. Hence the alternative guarded by $r \geq dd$ will be selected. The minor deficiency is that, since we know $r \geq dd$, a test is made unnecessarily. In "loop" terminology, the DO construct is a generalization of the "while...do..."; the test is at the beginning. What is needed is a "repeat...until..."; using DO we are obliged either to duplicate some code or to make an unnecessary test. Instead of mechanically translating Dijkstra's program, we should have written

"maintain P and $d|dd$, decrease r":

r:=r-dd; dd:=dd+dd;

if $r<dd$ → *skip*

▯ $r\geq dd$ → "maintain P **and** $d|dd$, decrease r"

fi

It is now clear, without examining any external context, that the refinement does decrease r.

Example 3

[1, p.65, Ex.6] Given integers $X \geq 1$ and $Y \geq 0$ establish

$$R\colon\ z = X^Y$$

without using exponentiation. As before, we obtain an invariant by weakening R to something that is more easily established.

$$P\colon\ z * x^y = X^Y \text{ and } y \geq 0$$

The program is then

$x := X;\ y := Y;\ z := 1;\ \{P\}$

"maintain P, establish $y = 0$" $\{R\}$

with the refinement

"maintain P, establish $y = 0$":

if $y = 0 \rightarrow$ *skip*

[] $y > 0 \rightarrow$ "maintain P, decrease y";

"maintain P, establish $y = 0$"

fi

Note 1. The condition $y \geq 0$ was omitted in the invariant in [1], but is required so that a decrease in y will be measurable progress. (*End of note 1.*)

Note 2. Following Dijkstra's advice, that "all other things being equal, we should choose our guards as weak as possible" [1, p.57], I had originally written the second guard above as "$y \neq 0$". In Dijkstra's program, using DO, the guard is "$y \neq 0$". One reason is robustness. Suppose that, due to either machine malfunction or incorrect programming, the portion of the program that is intended to "maintain P, decrease y" should fail to maintain P by decreasing y below 0. The guard "$y > 0$" would lead to termination of the DO loop, giving no alarm, whereas the guard "$y \neq 0$" would lead to non-termination. The latter is a kind of alarm, and therefore preferable. In our program, using recursive refinement, the guard "$y \neq 0$" would also

lead to non-termination, should y be erroneously decreased below 0. But the invariant P tells us that we need consider --and therefore should consider-- only the cases "$y=0$" and "$y>0$". Then the case "$y<0$" will lead to immediate abortion, which is a better alarm. The moral is, all other things being equal, we should choose our guards as strong as possible. (*End of note 2.*)

The remaining refinement is easy, if only a correct solution is wanted. For efficiency, the following refinement contains two separate improvements.

"maintain P, decrease y":

```
if    2|y → y := y/2; x:= x*x;
                "maintain P, decrease y"
 [] non 2|y → y:= y-1; z:=z*x
fi
```

The first improvement is to divide the task into two cases, one of which gives a possibly large decrease in y. The second is to realize that, if $y>0$ and $2|y$, then after division by 2 we still have $y>0$ and hence we may decrease y further.

Example 4

[1, p.67, Ex. 7] Given an integer $n \geq 0$ and a function $f(i)$ defined on the domain $0 \leq i < n$, establish

$$R: allsix = (\forall\ i: 0 \leq i < n: f(i) = 6)$$

Our first problem solving method, "divide and conquer", suggests that we look for predicates P and Q such that P and $Q \Rightarrow R$. In the previous examples, we have found the invariant P by weakening R; what was left became Q. We now take the opposite approach: we shall weaken R to obtain Q; what is left will be P.

$$Q: allsix = (\forall\ i: j \leq i < n: f(i) = 6)$$

$$P: 0 \leq j \leq n \text{ and } (\forall\ i: 0 \leq i < j: f(i) = 6)$$

Fortunately, P is easily established. Our program, with refinement, follows.

$j := 0$; $\{P\}$

"maintain P, establish Q"

"maintain P, establish Q":

```
if j=n → allsix:= true
 □ j<n → if f(j) ≠ 6 → allsix:=false
         □ f(j) = 6 → j:=j+1; "maintain P, establish Q"
         fi
fi
```

Exits

Apparently, the problem in Example 3 has been cited as supporting the need for intermediate exits from loops. One of Dijkstra's purposes in presenting his solution is to remove this support. But one example is not proof; those who would support the need for exits may prefer Example 4, or some other. The argument against exits, in short, is clarity: following a DO that contains an exit one can not be sure that all guards are false. The argument in favour of exits is efficiency: arranging that all guards become false for termination may require the introduction of a boolean variable, an assignment, and many tests that would be unnecessary using an exit [9]. We shall not take a side in the argument; we are objecting to loops with or without exits. But we shall show the relationship between recursive refinement and exits.

The general DO construct may be modelled as follows.

"DO": if $B_1 \rightarrow SL_1$; "DO"
[] $B_2 \rightarrow SL_2$; "DO"
[] ...
[] $B_n \rightarrow SL_n$; "DO"
[] *else* → *skip*
fi

where *else* = non $(\exists\, i{:}B_i)$. (The proof that the above is equivalent to the DO construct is presented later.) This model gives us a way of translating programs with DO constructs into programs using recursive refinement. But the programs so constructed are not necessarily ones to be proud of; given the different facility, we may construct our programs differently. Notice that if we omit the recursive "DO" from some alternative, we have, in loop terminology, an intermediate exit. Programs using loops and exits have the curious property that when there is more to be done, one says nothing, but

when there is no more to be done, one says something: exit. The recursive version is more straightforward; when there is more to be done, one specifies it, and when there is not, one says nothing. Unlike the DO construct, our decision in one alternative is independent of our decision in the others; we specify further (possibly recursive) action precisely in those alternatives where further action is needed.

Deep exits have been proposed as a means of exiting several nested loops at once. The arguments for and against are the same as for intermediate exits: efficiency versus clarity. Once again, we claim to provide both. A common example is a search for a given value in a multi-dimensional array.

Given integers $n \geq 0$ and $m \geq 0$, and an array or function $a(i,j)$ defined on the domain $0 \leq i < n$ and $0 \leq j < m$, and a value x, establish the following.

$$R0: \ 0 \leq ix \leq n$$

$$R1: \ ix < n \Rightarrow a(ix,jx) = x$$

$$R2: \ ix = n \Rightarrow \underline{\text{non}}(\exists\, i : 0 \leq i < n : \exists\, j : 0 \leq j < m : a(i,j) = x)$$

In words: ix and jx are to indicate a position of value x in the array if it is present; setting ix to n will indicate that it is not.

Our solution uses two invariants.

$$P1: \ 0 \leq ix \leq n \ \underline{\text{and}} \ \underline{\text{non}}(\exists\, i : 0 \leq i < ix : \exists\, j : 0 \leq j < m : a(i,j) = x)$$

asserts that the value x does not occur prior to row ix. And

$$P2: \ 0 \leq jx \leq m \ \underline{\text{and}} \ \underline{\text{non}}(\exists\, j : 0 \leq j < jx : a(ix,j) = x)$$

asserts that in row ix, the value x does not occur prior to column jx. Our program follows.

```
ix:=0; {P1}
"maintain P1, establish R1 or ix=n" {R0 and R1 and R2}

"maintain P1, establish R1 or ix=n":
    if ix=n → skip
     [] ix<n → jx:=0; { P2 }
                "maintain P1 and P2, establish R1 or ix=n"
    fi

"maintain P1 and P2, establish R1 or ix=n":
    if jx=m → ix:=ix+1;{P2 may be destroyed}
                "maintain P1, establish R1 or ix=n" {⇒ P2}
     [] jx<m → if a(ix,jx)=x → skip
                 [] a(ix,jx)≠x → jx:=jx+1;
                                 "maintain P1 and P2, establish R1 or ix=n"
                fi
    fi
```

The reader is invited to write a solution using DO constructs, stating all invariants and subgoals as above, and compare it with the above. Without an exit, the solution will probably involve a boolean variable, say "*found*", that is tested many times. Perhaps the problem as stated is unfair; it may be more reasonable to represent the presence of value x by a boolean variable than by the expression "$ix<n$". If so, the two "*skip*" statements in our solution must be replaced by "*found*:=*false*" and "*found*:=*true*" respectively, but no extra testing is required.

Implementation

In general, a call may be implemented as "stack a return address, then branch to the start of a refinement". The refinement must end with a return: "unstack a return address, then branch to it". It is sometimes thought that stacking is unnecessary if recursion is prohibited. The usual FORTRAN implementation, for example, associates with each subroutine one location for storing a return address. But these locations are filled and consulted in "last in, first out" order. They therefore form a stack, although its elements are dispersed; there is no advantage in dispersing the stack. Lack of recursion tells us that the number of subroutines (or refinements) is an upper bound on the size of the return address stack; often the program structure determines a much smaller upper bound. But the general need for a stack comes with the call and refinement, independent of whether there are recursive calls.

There are (at least) two situations in which stacking activity is unnecessary. They are the "last action call" and the "only call". A statement is a "last action" of a refinement if (a) it is the last statement of the refinement, or (b) it is the last statement of an alternative in a last action IF statement. A last action call may be implemented simply as a branch. This is independent of whether the call is recursive [10].

A call for a refinement is an "only call" if all other calls for that refinement are last action. Assuming that last action calls have been implemented as branches, we may implement an only call as a branch, and the return from the refinement as a branch. Or, the code for the refinement

can simply replace the only call. The latter is known as "opening"; it may be done even if the replaced call is not an only call, by duplicating the code for the refinement (if one so wishes). Once again, this is independent of whether the replaced call is recursive; when it is, this is known as "unrolling".

In summary, we make two points. The first is that recursion is irrelevant to the implementation of call and refinement; it adds no complication. The second is that our programs, implemented as described above, are as efficient as those using DO, and sometimes more efficient (see the section titled "Exits"). In particular, none of the examples presented involves any stacking.

Solving the Semantic Equations

Given a statement or statement list S, we consider that we understand S when we know, for all R, $wp(S, R)$. We call this predicate transformer the semantics of S. It may be found as a solution to the semantic equation for S, as given in the section titled "Semantics". If S is not recursive, the equation will have a unique solution that may be found, assuming we know the semantics of its components, by an appropriate sequence of substitutions, compositions, and applications of basic formulae. If S is recursive, its semantic equation may have more than one solution. In that case, $wp(S, R)$ is defined as the strongest solution, i.e., that solution A such that, if B is any solution, then $A \Rightarrow B$. The existence of a strongest solution follows from the existence of a solution, together with the monotonicity property: if $Q \Rightarrow R$ then $wp(S, Q) \Rightarrow wp(S, R)$; in words, wp preserves implication. The existence of a solution is proved constructively; in fact, the construction gives us the strongest solution. This section concerns that construction.

For now, we confine our attention to direct recursion. With this restriction, a semantic equation has the form $wp = f(wp)$. The strongest solution may be found as the limit of the approximating sequence $wp_0, wp_1, wp_2, \ldots$ defined as follows:

$$wp_0 = F$$

$$wp_i = f(wp_{i-1})$$

where "F" is the predicate that is everywhere false; we shall use "T" for the predicate that is everywhere true. The sequence is monotonically weakening (or, more precisely, non-strengthening); this fact follows once again from the monotonicity property. It is bounded by T. Hence a limit

exists. The limit is unique; this fact is the continuity property, proved in [1, ch. 9]. The limit may be expressed as

$$wp = (\sqcup i : wp_i)$$

Note. The approach we are taking is exactly the "least fixed point" approach to formal semantics taken by Scott and Strachey [8]. For the set of all predicates over the program variables forms a continuous lattice whose partial ordering is implication. The limit of our approximating sequence is a "least upper bound" if we identify "bottom" with "*F*" and "top" with "*T*". Each wp_i that approximates wp is the exact semantics of a statement S_i that approximates S. If S is defined (recursively) as "S: $\mathcal{F}$(S)", then the S_i are as follows:

$$S_0: \textit{abort}$$

$$S_i: \mathcal{F}(S_{i-1})$$

(End of note.)

Let us look at some examples. Our first one is inspired by Dijkstra's example [1, p. 76] to illustrate that, for correct execution, if two guards are true, then either may be selected regardless of the past history of the computation.

"increment x by an arbitrary amount":

    if *true* → x:=x+1; "increment x by an arbitrary amount"
    [] *true* → *skip*
    fi

wp("increment...", R) = wp("x:=x+1", wp("increment...", R)) and R

$wp_0 = F$

wp_1 = wp("x:=x+1", F) and R

= F

We have immediate convergence: wp("increment...", R)=F. This tells us that no initial state will guarantee any final state, i.e., not even termination is guaranteed.

Our second example proves the statement of the section titled "Problem Solving Methods" that omitting the measurable progress is futile.

"establish R": if $B1 \rightarrow S$
[] $B2 \rightarrow$ "establish R"
fi

wp("establish R", R)
= ($B1$ or $B2$) and ($B1 \Rightarrow$ wp(S, R)) and ($B2 \Rightarrow$ wp("establish R", R))

$wp_0 = F$

wp_1 = ($B1$ or $B2$) and ($B1 \Rightarrow$ wp(S, R)) and ($B2 \Rightarrow F$)
= $B1$ and non $B2$ and wp(S, R)

wp_2 = ($B1$ or $B2$) and ($B1 \Rightarrow$ wp(S, R)) and
($B2 \Rightarrow$ ($B1$ and non $B2$ and wp(S, R)))
= $B1$ and non $B2$ and wp(S, R)

We have converged to the solution wp("establish R", R) = $B1$ and non $B2$ and wp(S, R), which implies that the second alternative must be unnecessary. The other alternative may be generalized to any number of alternatives without changing this conclusion.

The next example illustrates that measurable progress toward the goal is sufficient for a recursive alternative.

"maintain $t \geq 0$, establish R":
if $R \rightarrow$ *skip*
[] non R and $t>0 \rightarrow t:=t-1$;
"maintain $t \geq 0$, establish R"
fi

$$\text{wp}(\text{"..."}, R) = (R \text{ or } (\text{non } R \text{ and } t>0)) \text{ and}$$
$$(R \Rightarrow \text{wp}(\text{"skip"}, R)) \text{ and}$$
$$((\text{non } R \text{ and } t>0) \Rightarrow \text{wp}(\text{"}t:=t-1\text{"}, \text{wp}(\text{"..."}, R)))$$
$$= (R \text{ or } t>0) \text{ and } (\text{non } R \Rightarrow \text{wp}(\text{"}t:=t-1\text{"}, \text{wp}(\text{"..."}, R)))$$

By finding the first few approximations, we are led to the formula

$$\text{wp}_i = (\exists\, j:0\le j<i:t>j \text{ and } R_{t:=t-j})$$

This formula may be proved by induction. Thus

$$\text{wp}(\text{"..."}, R) = (\exists\, j:0\le j:t>j \text{ and } R_{t:=t-j})$$

The refinement will therefore establish R if R can be established by reducing t. The proof that measurable progress is sufficient in general will not be given here; it is equivalent to Dijkstra's proof that the execution of a DO construct will terminate if it makes measurable progress on each repeated execution.

The next example is the model of the DO construct given in the section titled "Exits".

"DO": if $B_1 \to SL_1$; "DO"
[] $B_2 \to SL_2$; "DO"
[] ...
[] $B_n \to SL_n$; "DO"
[] $else \to skip$
fi

where $else = \text{non}(\exists\, i:Bi)$. In this example, the range of all quantified variables is understood to be the integers from 1 to n, unless specified otherwise.

$$\text{wp}(\text{"DO"}, R) = (\exists i{:}B_i) \text{ or } else) \text{ and}$$
$$(\forall i{:}B_i \Rightarrow \text{wp}(SL_i, \text{wp}(\text{"DO"}, R))) \text{ and}$$
$$(else \Rightarrow \text{wp}(\text{"skip"}, R))$$
$$= (\forall i{:}\ B_i \Rightarrow \text{wp}(SL_i, \text{wp}(\text{"DO"}, R))) \text{ and } (else \Rightarrow R)$$

$$\text{wp}_0 = F$$

$$\text{wp}_k = (\forall i{:}B_i \Rightarrow \text{wp}(SL_i, \text{wp}_{k-1})) \text{ and } (else \Rightarrow R)$$

$$\text{wp} = (\exists k{:}k{\geq}0{:}\text{wp}_k)$$

In defining the semantics of the repetitive DO construct, Dijkstra defines an infinite sequence of predicates $H_0(R)$, $H_1(R)$, $H_2(R)$,..., as follows:

$$H_0(R) = else \text{ and } R$$

$$H_k(R) = (\exists i{:}B_i) \text{ and } (\forall i{:}B_i \Rightarrow \text{wp}(SL_i, H_{k-1}(R)) \text{ or } H_0(R)$$

He then defines

$$\text{wp}(\text{DO}, R) = (\exists k{:}k{\geq}0{:}H_k(R))$$

A little boolean algebra reveals that $H_0(R){=}\text{wp}_1$ and that the recurrence relation for $H_k(R)$ is identical to the one for wp_k. The two sequences are therefore identical, except for a shift in subscripts. Since $\text{wp}_0{=}F$, we could redefine $\text{wp}{=}(\exists k{:}k{>}0{:}\text{wp}_k)$; thus we have proved that our model is semantically equivalent to the repetitive DO. The advantage of basing our approximating sequence on $\text{wp}_0{=}F$ is that this base, and the recurrence relation $\text{wp}_k{=}\text{f}(\text{wp}_{k-1})$, are applicable to all constructs, whereas the $H_k(R)$ are specific to DO. The advantage of basing $H_0(R)$ as Dijkstra bases it is that, thinking operationally, the subscript corresponds to the number of times that control passes through the loop.

As Dijkstra has pointed out [1, p.17], for programming we are not interested in the complete semantics of a construct S; that is, we do not care about wp(S, R) for all R. We want S to establish a particular predicate P. In general, to find wp(S, P) we find wp(S, R) for all R and then substitute P for R. But in special cases, we can solve directly for wp(S, P). Whenever the recursions are last action calls, wp(S,) is applied to the same predicate throughout the equation; in that case, we can form an approximating sequence in terms of the particular predicate P. In the following example, although the recursion is not a last action, the equation can be manipulated into the required form.

"establish $r=n!$":

  if $n=0 \rightarrow r:=1$

  [] $n>0 \rightarrow n:=n-1$;

    "establish $r=n!$";

    $n:=n+1$;

    $r:=r*n$

  fi

(This program is not recommended for its efficiency; it serves only as an example for solving its semantic equation.) As the choice of refinement name states, we want to use this construct to establish $r=n!$.

wp("establish $r=n!$", $r=n!$)

= ($n\geq 0$) **and** (($n=0$) $\Rightarrow$ wp("$r:=1$", $r=n!$)) **and** (($n>0$) $\Rightarrow$ wp("$n:=n-1$",
wp("establish $r=n!$", wp("$n:=n+1$", wp("$r:=r*n$", $r=n!$)))))

= ($n\geq 0$) **and** (($n=0$) $\Rightarrow$ ($1=n!$)) **and** (($n>0$) $\Rightarrow$ wp("$n:=n-1$",
wp("establish $r=n!$", wp("$n:=n+1$", $r*n=n!$))))

= ($n\geq 0$) **and** (($n>0$) $\Rightarrow$ wp("$n:=n-1$", wp("establish $r=n!$", $r*(n+1)=(n+1)!$)))

= ($n\geq 0$) **and** (($n>0$) $\Rightarrow$ wp("$n:=n-1$", wp("establish $r=n!$", $r=n!$)))

The approximating sequence may now be formed, yielding the expected solution $n \geq 0$.

So far, we have confined our attention to direct recursion. The equations for indirect recursions can sometimes be put in the form wp=f(wp) by substitution. The program that searches for an element in a two-dimensional array, presented in the section titled "Exits", is such an example. It has roughly the following form.

D1: **if** *B1*→*S1*
▯ *B2*→*S2*; *D2*
fi

D2: **if** *B3*→*S3*; *D1*
▯ *B4*→*S4*; *D2*
fi

By substituting for wp(*D1*, *R*) in the equation for wp(*D2*, *R*) we eliminate the indirection. In general, when such a substitution is impossible, we must solve by forming several approximating sequences simultaneously.

The predicate transformer wp(*S*, *R*) gives, for statement(s) *S* and any post-condition *R*, the weakest pre-condition such that *S* establishes *R*. Dijkstra also introduces the "weakest liberal pre-condition" predicate transformer wlp(*S*, *R*), which gives the weakest pre-condition such that *S* does not establish **non** *R*. If *S* is deterministic, then wlp(*S*, *R*) = **non** wp(*S*, **non** *R*). In general, the semantic equations for wlp are as follows.

$$\text{wlp}(\text{"}skip\text{"},\ R) = R$$

$$\text{wlp}(\text{"}x{:=}E\text{"},\ R) = R_{x:=E}$$

$$\text{wlp}(\text{"}S_1;S_2\text{"},R) = \text{wlp}(S_1,\ \text{wlp}(S_2,\ R))$$

$$\text{wlp}(\text{IF},\ R) = (\forall i{:}B_i \Rightarrow \text{wlp}(SL_i,\ R))$$

When the equations are recursive, wlp is defined as the weakest solution. It may be found as the limit of the approximating sequence

$$\text{wlp}_0 = T$$

$$\text{wlp}_i = \text{f}(\text{wlp}_{i-1})$$

Note. Monotonicity and continuity are provable for wlp, hence a unique limit exists. Once again, this is the Scott-Strachey "least fixed point" approach [8] if we turn our lattice upside-down and identify "bottom" with "T" and "top" with "F". Each wlp_i that approximates wlp is the unique solution of an equation for a statement S_i that approximates S. The S_i are the same for wlp as for wp. (*End of note.*)

A Final Alternative

According to the semantic equation for the IF construct, every IF contains the implicit guarded command "*else*→*abort*"; the condition $(\exists i{:}B_i)$ is a simplification of "*else* ⇒ wp("*abort*", R)". As shown in the previous section, the semantics of DO are those of IF with two exceptions: there is an implicit recursion after each explicit alternative; and the implicit guarded command is "*else*→*skip*". Our interest in this section is in the implicit guarded command.

As we noted in Example 3, our programs, using IF, are more robust than those using DO. The extra robustness is due entirely to the fact that the "*else*" alternative is "*abort*" in IF, and "*skip*" in DO. The DO could have been defined with extra robustness by making the implicit "*else*" alternative "*abort*"; in that case the termination condition would have to be stated explicitly, perhaps by the guarded command "B_{n+1}→*exit*". The IF could have been defined without extra robustness by making the implicit "*else*" alternative "*skip*"; this would be similar to Algol's one-tailed "if...then...".

The reader who has compared our programming examples with those of Dijkstra will have noticed that Dijkstra's programs are more compact. There are two reasons for this: one is that the call and refinement statements require the introduction of names for portions of our programs; the other is the presence of "*skip*" alternatives. Under the second suggestion of the preceeding paragraph, our programs would be more compact. Had Dijkstra chosen to add robustness by making his guards strong and including an "*abort*" alternative, his programs would be less compact. The tradeoff is clear: robustness versus compactness. We favour robustness, so we prefer the semantics of IF as Dijkstra has defined them. But we point out the alternative.

A Notational Ideal

We assume in this section that wp is the only predicate transformer of interest. It is wp that defines the meaning or semantics of a statement or construct. To give the semantics of a particular statement S, one must give $\mathrm{wp}(S, R)$ for all predicates R. Thus there is a certain asymmetry in the use of the arguments. As a first suggestion, consider denoting the weakest pre-condition such that S establishes R by "$S(R)$". That is, we shall consider each statement to be a predicate transformer.

In the semantic equation for a compound statement "$S1$; $S2$", it is pleasant to write the component statements in the same order on both sides of the equation:

(1) $$"S1;\ S2"(R) = S1(S2(R))$$

This results from Dijkstra's "goal oriented" approach, i.e., transforming post-conditions to pre-conditions, together with our traditional prefix notation for functions. But when the component statements are assignments, it is annoying to see them occurring in reverse order as subscripts. This may be remedied by using a prefix subscript:

(2) $$"x:=E"(R) = {}_{x:=E}R$$

so that

$$"x:=E;\ y:=D"(R) = {}_{x:=E}({}_{y:=D}R)$$

Having made these suggestions, we are struck by how trivial, and purely cosmetic, are the differences between the left and right sides of equations (1) and (2). To increase the similarity, we can drop the parentheses, since we are

dealing with only unary, prefix operators. We are struck also by the similarity between Dijkstra's notation for a guarded command, and the implication that it gives rise to, in a semantic equation.

There is no point in having two notations for the same thing. As a notational ideal, we suggest that each statement should denote its meaning. A programming language should be designed so that one need not, separately, give the semantic equations. Rather, every program should denote its semantics.

Conclusion

We have suggested that refinement deserves a place in a programming language, to allow programs to retain design decisions that would otherwise be lost. Given refinement, we see no reason to enlarge the language with a special rule to prohibit recursion; on the contrary, recursive refinement is a more flexible programming tool, allowing us to produce clearer and more efficient programs, than the "do guarded-command-set od" construct.

There are (at least) two reasons that recursion has been considered a difficult programming tool. First, it has been tied to two other language issues: parameters, and local scope (i.e., recursive procedures). It is, however, a separable concern. Second, almost every programming text explains recursion, as it explains all language constructs, by explaining how to trace an execution according to some implementation. And that implementation invariably involves a stack, even though a stack is frequently unnecessary. But a program should not be understood in terms of any particular implementation, and cannot be understood by tracing an execution. The basis of our understanding of recursion, or of loops, must be the principle of mathematical induction.

It is sometimes objected that an average person cannot be expected to understand the principle of induction, or to apply it to programming. If that were true, it would not be an argument against the use of induction in programming, but against the use of average people as programmers. In fact, average people understand the principle perfectly well, although informally. Given a positive integer, and enough time, an average person believes he can count from 1 to the given integer. For large enough integers, that belief is not based on the experience of having done so, but on an implicit

understanding of induction. And finally, we remark that a programmer can often use a result, such as "measurable progress is sufficient for recursion", whose proof is an induction, rather than using induction directly.

Our programming examples were chosen from among those that Dijkstra used [1] to exhibit the DO construct. The ones omitted were not omitted to hide problems that arose; on the contrary, nothing new arose in them. The reader is invited to come to this happy conclusion for himself by trying the remaining examples. (Dijkstra's example 8 [1, p.68] is particularly good. Hint 1: a repeat...until... would have improved Dijkstra's program, as in our example 2. Hint 2: there is an error in one of the invariants.) Only through practice with both can we judge the relative merits of DO and recursive refinement. Initially, the latter will be at a disadvantage; our judgement is, of course, affected by what we are used to, and most of us (myself included) are used to operational semantics and loops. But computer programming is barely 30 years old; let us not be too set in our ways at so young an age.

Appendix

This appendix contains Dijkstra's solutions to the four example problems of this paper. The presentation here is devoid of the reasoning, justifications, and other commentary that ought to accompany these solutions. It is intended only as a reference, and not as a substitute for [1].

Example 1 [1, p.45]

```
x:=X; y:=Y;
do x>y→x:=x-y
 [] x<y→y:=y-x
od;
print (x)
```

Example 2 [1, p.58]

```
r:=a;
do r≥d→dd:=d; do r≥dd→r:=r-dd; dd:=dd+dd od
od
```

Example 3 [1, p.66]

```
x:=X; y:=Y; z:=1;
do y≠0→do 2|y→y:=y/2; x:=x*x od; y:=y-1; z:=z*x
od
```

Example 4 [1, p.68]

```
allsix:=true; j:=0;
do j≠n and allsix→allsix:=f(j)=6; j:=j+1 od
```

Acknowledgement

I gratefully acknowledge discussions with Jim Horning and Art Sedgwick. Jim Horning suggested the title, and pointed out an error in the first draft. For the painstaking typing I thank J. Wood.

References

[1] E.W. Dijkstra. A Discipline of Programming. Prentice-Hall, New Jersey, 1976.

[2] E.W. Dijkstra. Guarded commands, non-determinacy, and formal derivation of programs. CACM 18(8) p.453, August 1975.

[3] E.W. Dijkstra. A Short Introduction to the Art of Programming. Report EWD316, Technological University of Eindhoven, August 1971.

[4] N. Wirth. Program development by stepwise refinement. CACM 14(4) pp. 221-227, April 1971.

[5] H.D. Mills. On the development of large reliable programs. IEEE Symposium on Computer Software Reliability, pp. 155-159, New York, April 1973.

[6] C.A. R. Hoare. An axiomatic basis for computer programming. CACM 12(10) pp. 576-580, October 1969.

[7] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. Acta Informatica 2, pp. 335-355, 1973.

[8] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. J. Fox (ed.), Computers and Automata, Wiley, pp. 19-46. 1972.

[9] H.F. Ledgard and M. Marcotty. A genealogy of control structures. CACM 18(11), November 1975.

[10] D.E. Knuth. Structured programming with go to statements. ACM Computing Surveys 6(4), December 1974.