Acta Informatica 11, 287–304 (1979)

Acta Informatica

do Considered od: A Contribution to the Programming Calculus*

Eric C.R. Hehner

Computer Systems Research Group, University of Toronto, Toronto M5S 1A4, Canada

Summary. The utility of repetitive constructs 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 "least fixed point" semantics is presented.

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 are 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

* This work was partially supported by the National Research Council of Canada

0001-5903/79/0011/0287/$03.60

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 is denoted by "*skip*" and is executed by doing nothing. The "*abort*" statement is executed presumably by printing an error message and then halting; like Dijkstra, we shall not use it. An assignment statement is denoted by "$x := E$" where x is any variable and E is any expression of the appropriate type; we leave its further description and its execution to the reader's experience. Sequencing is denoted by a semi-colon; to execute "$S_1; S_2$", first execute S_1, and then execute S_2. The IF statement is denoted by

$$\textbf{if}\ B_1 \to SL_1 \,[]\, B_2 \to SL_2 \,[]\, \ldots \,[]\, B_n \to SL_n\ \textbf{fi}$$

where the B_i are boolean expressions called "guards", and the SL_i are statement lists called "alternatives". To execute an IF statement, execute any one of the alternatives whose guard is true. The DO statement, denoted by

$$\textbf{do}\ B_1 \to SL_1 \,[]\, B_2 \to SL_2 \,[]\, \ldots \,[]\, B_n \to SL_n\ \textbf{od}$$

is executed as follows: repeatedly, as long as some guard is true, execute any one of the alternatives whose guard is true. Unlike Dijkstra, we shall not make use of the DO statement.

The programming technique known as "stepwise refinement", championed by Dijkstra [3] and others [7], 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". A call consists of a name enclosed in quotation marks; a refinement consists of the quoted name, followed by a colon, followed by a statement list:

"name": SL

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 the 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 implemen-

tation of procedure calls in some other language. We do not intend to imply 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 the implementation of call and refinement later. *(End of note.)*

The meaning of the statements in the language may be given in either of two ways; one is called "operational semantics", the other "mathematical semantics". According to operational semantics, we view a program as a description of a sequence of activities. 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 perform the change in state. This view is of interest to an implementer, and to a programmer when he or she is concerned with efficiency. The language description in the preceding paragraphs, though incomplete and informal, is of the operational kind. The words "repetition" and "termination" belong to operational semantics.

According to mathematical semantics, we view a program as a mapping between initial (input) states, and final (output) states. A statement describes a change in state (it is not a command to do anything). The description adopted by Dijkstra 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. This view is of interest to a programmer when he or she is concerned with writing correct programs.

Recognizing that programming is a goal-directed activity, Dijkstra gives the mathematical semantics of a statement S by a predicate transformer 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 by the following equations.

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

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

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

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

In the above, "$R_{x:=E}$" denotes the predicate obtained from R by simultaneously changing all free occurrences of "x" into "E". The existential and universal quantifiers take i over the integers in the range $1 \leqq i \leqq n$. The call gives us no semantic equation; a call is given meaning by the details of its refinement. A refinement gives us the trivial equation

$$\text{wp}(\text{“name”}, R) = \text{wp}(SL, R).$$

When some calls are recursive, the equations become recursive, raising the possibility that they may have more than one solution. But there is always one

solution that implies all (other) solutions (for proof, see the section titled "Solving the Semantic Equations"); that one defines the semantics of a recursive construct. Solving the equations may be difficult, with or without recursion; sometimes the solution may be expressed in a "closed form", and sometimes not. Fortunately, as Dijkstra points out [1, p. 17], we are often not interested in solving the equations. For programming, a predicate that is stronger than $\mathrm{wp}(S, R)$ (and hence not a solution) will content us.

Note. Predicate transformers have been given at the statement level so that statements may be viewed as mathematical objects, rather than as a sequence of activities. Predicate transformers have not been given at the expression level under the assumption that expressions are mathematical already. Any language feature that would require a particular ordering in the evaluation of sub-expressions, such as **cand** (**c**onditional **and**), is thus excluded. *(End of note.)*

Problem Solving Methods

The mathematical semantics suggest four problem solving methods, one leading to each of *skip*, assignment, IF, and sequencing. The problem to be solved will be written as "given G, establish R" where G, the given information, and R, the desired result, are expressed as first-order predicates.

Method 1. If $G \Rightarrow R$, then we can solve as follows.

"given G, establish R": *skip*

Method 2. If $G \Rightarrow R_{x:=E}$, then we can solve as follows.

"given G, establish R": $x := E$

Method 3, case analysis. If we can find predicates $B_1, B_2, \ldots, B_n$ such that $G \Rightarrow (B_1$ **or** B_2 **or** $\ldots$ **or** $B_n)$ then we can solve as follows.

"given G, establish R":
if $B_1 \rightarrow$ "given G **and** B_1, establish R"
$[]$ $B_2 \rightarrow$ "given G **and** B_2, establish R"
$[]$ $\ldots$
$[]$ $B_n \rightarrow$ "given G **and** B_n, establish R"
fi

Method 4, divide and conquer.

"given G, establish R":
"given G, establish P";
"given P, establish R"

Two particular choices of P are worth special mention. If we can find H and P such that $(G$ **and** $H) \Rightarrow P$, then we write

"given G, establish R":
"maintain G, establish H";
"given P, establish R"

In the first call, the word "maintain" is not fully appropriate; it means only that G is both given and to be established. Also, in the same inappropriate sense adopted from proofs of loop programs, we call G an "invariant" in the first call. The other choice of P makes it an invariant in the second call. If we can find P and Q such that $(P \textbf{ and } Q) \Rightarrow R$, then we write

"given G, establish R":
"given G, establish P";
"maintain P, establish Q"

Methods 3 and 4 generate subproblems that may be solved by these same four methods. To ensure correctness, we shall require that, whenever a refinement contains a call, some kind a progress be made. There are two kinds of progress: informational progress, and computational progress. In case-analysis, informational progress is made in the ith case if $G \not\Rightarrow B_i$. In divide-and-conquer, informational progress is made in the first call if $R \Rightarrow P$ and $P \not\Rightarrow R$, and the second call if $P \Rightarrow G$ and $G \not\Rightarrow P$. As long as informational progress is being made, no recursions (loops) can be formed. When informational progress is not made, and this is how the programming task terminates, the call may be to a refinement that has already been written. Each loop thus formed must contain computational progress, defined as in [1], as a net decrease in a function that is integer-valued and bounded below.

In the programming examples that follow, we shall abbreviate "given" as "gvn", "maintain" as "mtn", and "establish" as "est".

Programming Example 1A

[1, p. 57, Ex. 3] Given integers $a \geqq 0$ and $d > 0$, find the remainder r when a is divided by d using only addition and subtraction.

Since a and d are not variables, the given information is always maintained, and we shall not bother to say in each refinement. We are asked to establish

$$R: 0 \leqq r < d \textbf{ and } d|(a-r).$$

We begin with divide-and-conquer, weakening R to the more easily established

$$P: 0 \leqq r \qquad \textbf{and } d|(a-r).$$

Our program is

"est R": "est P"; "mtn P, est $r < d$"

We have made informational progress in each part, so our program is correct (so far). The first refinement is easy.

"est P": $r := a$

For the second, 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 want to minimize the size of the required doses. The DO construct is a combination of case-analysis and divide-and-conquer with computational progress. At this point, we need only the inspiration to use case analysis.

"mtn P, est $r<d$":
  if $r<d \rightarrow$ "gvn $r<d$, mtn P, est $r<d$"
  ▯ $r \geq d \rightarrow$ "gvn $r \geq d$, mtn P, est $r<d$"
  fi

Then, we refine each alternative independently, with the full range of methods available. The first alternative is trivial.

"gvn $r<d$, mtn P, est $r<d$": *skip*

One way to refine the second alternative is

"gvn $r \geq d$, mtn P, est $r<d$":
  "gvn $r \geq d$, mtn P";
  "mtn P, est $r<d$"

In this use of divide-and-conquer, informational progress is made in the first call, but not in the second. The second call has already been refined; we must ensure that the first call makes computational progress.

"gvn $r \geq d$, mtn P": $r := r-d$

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 computational progress is sufficient for recursion, we have some words of warning. The proof is, of course, an induction on the decreasing function. 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."

So far, we have constructed only what could also have been constructed with DO; after the next section, we shall return to this example problem to demonstrate that recursive refinement is computationally superior. We have already demonstrated a methodological advantage. 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.

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 [4].

If a call is the only one for a particular refinement, it and the return from the refinement may be implemented as branches. 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".

The implementation of last action calls and only calls are not independent; we may implement either as stated above, but then the other is restricted. Suppose all last action calls are implemented as branches. Refinements are thus connected into groups. If a call is the only one to a particular group (except for the last action calls), then it and all returns from the group may be implemented as branches. Alternatively, suppose all only calls and corresponding returns are implemented as branches. Then a last action call may be implemented as a branch if the returns of the calling and called refinements are implemented identically.

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 our programming examples involves any stacking.

Programming Example 1B

We now return to the problem of finding the remainder r when a is divided by d, using only addition and subtraction. Whereas we previously displayed at length the sequence of individual methods, giving each refinement a name, we shall

now sometimes apply methods together in one step. On the other hand, our methods require that certain program portions 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. (By failing to name and separate a refinement, we have occasionally missed an opportunity to improve an algorithm.)

Following Dijkstra, we can speed up our program by introducing variable dd as a multiple of d. Thus we can decrease r by multiples of d at a time. To obtain a program that is computationally equivalent to Dijkstra's, we must refine as follows.

"gvn $r \geqq d$, mtn P, est $r < d$":
  $dd := d$;
  "gvn $d \leqq dd \leqq r$ **and** $d|dd$, mtn P, est $r < d$"

"gvn $d \leqq dd \leqq r$ **and** $d|dd$, mtn P, est $r < d$":
  if $dd > r \rightarrow$ "mtn P, est $r < d$"
  ▯ $dd \leqq r \rightarrow r := r - dd$; $dd := dd + dd$;
    "gvn $d \leqq dd \leqq r$ **and** $d|dd$, mtn P, est $r < d$"
  fi

The minor deficiency in the last refinement above is that it is not efficient; it begins by making an unnecessary test, since we are given $dd \leqq r$. The major deficiency is that it is not obviously correct; contrary to its name, the final call occurs in a context where $dd \leqq r$ is not assured. 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** ...". Instead of mechanically translating Dijkstra's program, we should have written

"gvn $d \leqq dd \leqq r$ **and** $d|dd$, mtn P, est $r < d$":
  $r := r - dd$; $dd := dd + dd$;
  if $dd > r \rightarrow$ "mtn P, est $r < d$"
  ▯ $dd \leqq r \rightarrow$ "gvn $d \leqq dd \leqq r$ **and** $d|dd$, mtn P, est $r < d$"
  fi

It is now clear that the refinement is correct.

Programming Example 2

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

$$R: z = X^Y$$

without using exponentiation. (We can include $X = 0$ if we define $0^0 = 1$.) As before, we obtain an invariant by weakening R to something that is more easily established.

P: $z*x^y = X^Y$ **and** $y \geqq 0$.

The program is then

"est R": $x := X$; $y := Y$; $z := 1$; "mtn P, est $y=0$"
"mtn P, est $y=0$":
  if $y=0 \rightarrow$ *skip*
  ▯ $y>0 \rightarrow$ "gvn $y>0$, mtn P, est $y=0$"
  fi
"gvn $y>0$, mtn P, est $y=0$":
  $y := y-1$; $z := z*x$; "mtn P, est $y=0$"

Note. For the DO construct, Dijkstra advises that "all other things being equal, we should choose our guards as weak as possible" [1, p. 57]. Thus he writes "$y \neq 0$" where correspondingly we have written "$y>0$". One reason for the advice is robustness. Suppose that, due to either machine malfunction or incorrect programming, the final refinement 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; the guard "$y \neq 0$" would lead to non-termination, which is a kind of alarm, and therefore preferable. In our program, using recursive refinement, the guard "$y \neq 0$" would also lead to nontermination, 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, if ever y is less than 0, we have immediate abortion, which is a better alarm. We should therefore choose our guards as strong as possible. *(End of note.)*

Let us now revise the final refinement to make it more efficient. The idea is to divide the task into two subcases, one of which allows us to make a possibly large decrease in y.

"gvn $y>0$, mtn P, est $y=0$":
  if non $2|y \rightarrow$ "gvn **non** $2|y$, mtn P, est $y=0$"
  ▯ $2|y \rightarrow$ "gvn $2|y$ **and** $y>0$, mtn P, est $y=0$"
  fi
"gvn **non** $2|y$, mtn P, est $y=0$":
  $y := y-1$; $z := z*x$; "mtn P, est $y=0$"
"gvn $2|y$ **and** $y>0$, mtn P, est $y=0$":
  $y := y/2$; $x := x*x$; "mtn P, est $y=0$"

We can further improve by realizing that, if $2|y$ and $y>0$, then after division by 2 we still have $y>0$ and we may call a more appropriate refinement to make $y=0$, avoiding an unnecessary test. We therefore revise:

"gvn $2|y$ **and** $y>0$, mtn P, est $y=0$":
  $y := y/2$; $x := x*x$; "gvn $y>0$, mtn P, est $y=0$"

Our program is now computationally equivalent to Dijkstra's, but its form and development are dramatically different. In particular, notice that the last change,

which for us was small, corresponds to the introduction of an inner DO loop. For even more efficiency, we can rewrite the next-to-last refinement as follows:

"gvn **non** 2|y, mtn P, est $y=0$":
  $y:=y-1$; $z:=z*x$; "gvn 2|y, mtn P, est $y=0$"
"gvn 2|y, mtn P, est $y=0$":
  if $y=0 \rightarrow$ *skip*
  ▯ $y>0 \rightarrow$ "gvn 2|y **and** $y>0$, mtn P, est $y=0$"
  fi

If we later realize that our program will be used to compute X^Y for odd exponents more often than for zero exponents, we should begin our program by testing for oddness. Rather than requiring a major structural change, this means revising only one refinement.

"mtn P, est $y=0$":
  if non 2|$y \rightarrow$ "gvn **non** 2|y, mtn P, est $y=0$"
  ▯ 2|$y \rightarrow$ "gvn 2|y, mtn P, est $y=0$"
  fi

The efficiency we have gained may not be important, but the ease and security with which we can modify a program is important.

Exits

Whenever a language contains loops, the proposal is inevitably made that it should contain an "intermediate exit" to allow escape from a loop without retesting the guards. The argument against exits, in short, is clarity: following a DO that contains an exit one cannot 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 [5]. 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* $\rightarrow$ *skip*
  fi

where *else* = **non** $(\exists i: B_i)$. (The proof that the above is equivalent to the DO construct is presented in the section titled "Semantics Example 4". 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. Programming Example 3 will illustrate this point.

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. We leave illustration of this point to the reader.

Programming Example 3

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

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

In words, set logical variable *allsix* to indicate whether the function value is always 6. With the abbreviation

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

our program follows.

"est R": $j := 0$; "gvn P, est R"

"gvn P, est R":
if $j = n \rightarrow allsix := true$
$\square$ $j < n \rightarrow$ "gvn P **and** $j < n$, est R"
fi

"gvn P **and** $j < n$, est R":
if $f(j) = 6 \rightarrow j := j + 1$; "gvn P, est R"
$\square$ $f(j) \neq 6 \rightarrow allsix := false$
fi

Notice that the program sets the value of *allsix* once, when its value is finally known, and does not test its value. If we receive the new information that $n \geqq 1$, our program becomes

"est R": $j := 0$; "gvn P **and** $j < n$, est R"

and no other changes are needed.

Solving the Semantic Equations

Given a statement or statement list S, we consider that we understand S when we know, for all predicates R, the predicate $\text{wp}(S, R)$. The semantics of S, then, is the predicate transformer awkwardly denoted by $\text{wp}(S,)$. We shall take the notational liberty of denoting this predicate transformer by wp whenever S is understood. So that we can use the calculus of predicates, rather than a calculus of predicate transformers, we shall consider that wp is applied to the free predicate variable R unless stated otherwise.

We can find wp 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 is defined as the strongest solution, i.e., that solution A such that, if B is any solution, then $A \Rightarrow B$. The existence and uniqueness of a strongest solution are proved using properties of wp proved in [1]; specifically, the monotonicity property:

if $A \Rightarrow B$ then $\text{wp}(S, A) \Rightarrow \text{wp}(S, B)$
in words: wp preserves implication,

and the continuity property:

if $C_0, C_1, C_2, \ldots$ is a (finite or infinite) sequence of predicates
such that $(\forall i: C_i \Rightarrow C_{i+1})$
then $\text{wp}(S, (\exists i: C_i)) = (\exists i: \text{wp}(S, C_i))$
in words: wp preserves limits.

This section presents a construction that gives us the strongest solution.

For now, we confine our attention to direct recursion. With this restriction, a semantic equation has the form $\text{wp} = f(\text{wp})$. The strongest solution may be found as the limit of the approximating sequence $\text{wp}_0, \text{wp}_1, \text{wp}_2, \ldots$ defined as follows:

$\text{wp}_0 = F,$
$\text{wp}_i = \text{f}(\text{wp}_{i-1})$

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).

Proof. Induction base: $\text{wp}_0 \Rightarrow \text{wp}_1$ (trivial).
Induction step: assume $\text{wp}_{i-1} \Rightarrow \text{wp}_i$.
Then, by monotonicity, $\text{f}(\text{wp}_{i-1}) \Rightarrow \text{f}(\text{wp}_i)$, i.e., $\text{wp}_i \Rightarrow \text{wp}_{i+1}$.
Therefore, $\forall i: \text{wp}_i \Rightarrow \text{wp}_{i+1}$. (*End of proof.*)

The sequence is bounded by T. Hence the limit

$\text{wp}_\infty = (\exists i: \text{wp}_i)$

exists. The limit is a solution.

Proof. $f(wp_\infty) = f(\exists i: wp_i)$
by continuity $= \exists i: f(wp_i)$
$= \exists i: wp_{i+1}$
$= wp_\infty$. *(End of proof.)*

Furthermore, it is the unique strongest solution.

Proof. Suppose x is a solution. Then $x = f(x)$.
Induction base: $wp_0 \Rightarrow x$ (trivial).
Induction step: assume $wp_{i-1} \Rightarrow x$.
Then, by monotonicity, $f(wp_{i-1}) \Rightarrow f(x)$, i.e., $wp_i \Rightarrow x$.
Therefore, $\forall i: (wp_i \Rightarrow x)$. Therefore, $(\exists i: wp_i) \Rightarrow x$.
Therefore, $wp_\infty \Rightarrow x$. *(End of proof.)*

The approach we are taking is exactly the "least fixed point" approach to computation taken by Scott [6]. For the set of all predicates over the program variables forms a complete, 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 "$S: \mathscr{F}(S)$", then the S_i are as follows:

S_0: *abort*
$S_i: \mathscr{F}(S_{i-1})$

The predicate transformer wp 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, which gives the weakest pre-condition such that S does not establish **non** R. If S is deterministic, then $wlp(S, R) = \textbf{non}\ wp(S, \textbf{non}\ R)$. In general, the semantic equations for wlp are as follows.

$wlp(\text{"}skip\text{"}, R) = R$,
$wlp(\text{"}x := E\text{"}, R) = R_{x:=E}$,
$wlp(\text{"}S_1; S_2\text{"}, R) = wlp(S_1, wlp(S_2, R))$,
$wlp(\text{IF}, R) = (\forall i: B_i \Rightarrow wlp(SL_i, R))$.

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

$wlp_0 = T$,
$wlp_i = f(wlp_{i-1})$.

Monotonicity and continuity are provable for wlp, hence the limit $wlp_\infty = (\forall i: wlp_i)$ is the unique weakest solution. Once again, this is Scott's "least fixed point" approach [6] 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.

Semantics Example 1

Our first semantics example is inspired by Dijkstra's example [1, p. 76] to illustrate that a finite program must not require a machine to make an unbounded number of non-deterministic choices.

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

wp("increment ...", R) = wp("$x:=x+1$", wp("increment ...", R)) **and** R
$\text{wp}_0 = F$
$\text{wp}_1 = \text{wp}(\text{"}x:=x+1\text{"}, F)$ **and** R
$= F$.

We have immediate convergence : wp("increment ...", R) = F.

The semantics of IF tell us that, for correct execution, if two guards are true, then the statement list corresponding to either one may be selected regardless of the past history of the computation. With a straightforward implementation, execution of the above may not terminate.

Semantics Example 2

In our second example, one of the alternatives of a case-analysis fails to make informational or computational progress.

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

wp("est R", R)
$= (B1$ **or** $B2)$ **and** $(B1 \Rightarrow \text{wp}(S, R))$ **and** $(B2 \Rightarrow \text{wp}(\text{"est } R\text{"}, R))$
$\text{wp}_0 = F$
$\text{wp}_1 = (B1$ **or** $B2)$ **and** $(B1 \Rightarrow \text{wp}(S, R))$ **and** $(B2 \Rightarrow F)$
$= B1$ **and non** $B2$ **and** $\text{wp}(S, R)$
$\text{wp}_2 = (B1$ **or** $B2)$ **and** $(B1 \Rightarrow \text{wp}(S, R))$ **and**
$(B2 \Rightarrow (B1$ **and non** $B2$ **and** $\text{wp}(S, R)))$
$= B1$ **and non** $B2$ **and** $\text{wp}(S, R)$.

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

Semantics Example 3

The next example illustrates that computational progress is sufficient for a recursion.

"mtn $t \geqq 0$, est R": **if** $R \rightarrow skip$
$\square$ **non** R **and** $t > 0 \rightarrow t := t-1$; "mtn $t \geqq 0$, est R"
fi

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

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

$$\text{wp}_i = (\exists j: 0 \leqq j < i: t \geqq j \textbf{ and } R_{t:=t-j}).$$

This formula may be proved by induction. Thus

$$\text{wp}(\text{"mtn } t \geqq 0, \text{ est } R\text{"}, R) = (\exists j: 0 \leqq j: t \geqq j \textbf{ and } R_{t:=t-j}).$$

The refinement will therefore establish R if R is true already, or can be established by reducing t. The proof that computational progress is sufficient for recursion 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 computational progress on each repeated execution.

Semantics Example 4

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

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

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

$\text{wp}(\text{"DO"}, R) = (\exists i: B_i)$ **or** $else)$ **and**
$(\forall i: B_i \Rightarrow \text{wp}(SL_i, \text{wp}(\text{"DO"}, R)))$ **and**
$(else \Rightarrow \text{wp}(\text{"}skip\text{"}, R))$
$= (\forall i: B_i \Rightarrow \text{wp}(SL_i, \text{wp}(\text{"DO"}, R)))$ **and** $(else \Rightarrow R)$
$\text{wp}_0 = F$
$\text{wp}_k = (\forall i: B_i \Rightarrow \text{wp}(SL_i, \text{wp}_{k-1}))$ **and** $(else \Rightarrow R)$
$\text{wp}_\infty = (\exists k: k \geqq 0: \text{wp}_k)$.

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:

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

He then defines

$$\mathrm{wp}(\mathrm{DO}, R) = (\exists k: k \geqq 0: H_k(R)).$$

A little boolean algebra reveals that $H_0(R) = \mathrm{wp}_1(\mathrm{DO}, R)$ 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 $\mathrm{wp}_0 = F$, we could redefine $\mathrm{wp}_\infty = (\exists k: k > 0: \mathrm{wp}_k)$; thus we have proved that our model is semantically equivalent to the repetitive DO. The advantage of basing our approximating sequence on $\mathrm{wp}_0 = F$ is that this base, and the recurrence relation $\mathrm{wp}_k = \mathrm{f}(\mathrm{wp}_{k-1})$, are applicable to all constructs, whereas the $H_k(R)$ are specific to DO.

General Recursion

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 $\mathrm{wp}(S, R)$ for all R. We want S to establish a particular predicate 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 general, to find $\mathrm{wp}(S, P)$ we find $\mathrm{wp}(S, R)$ for all R and then substitute P for R.

So far, we have confined our attention to direct recursion. The equations for indirect recursions can sometimes be put in the form $\mathrm{wp} = \mathrm{f}(\mathrm{wp})$ by substitution. In general, when such a substitution is impossible, we must solve by forming several approximating sequences simultaneously.

A Final Alternative

According to the semantic equation for the IF construct, every IF contains the implicit guarded command "*else* $\rightarrow$ *abort*"; the condition $(\exists i: B_i)$ is a simplification of "*else* $\Rightarrow$ 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* $\rightarrow$ *skip*". Our interest in this section is in the implicit guarded command.

As we noted in Programming Example 2, 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 "$B_{n+1} \rightarrow$ *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 preceding 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.

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**" (or any loop) 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 "computational 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. Only through practice with both DO and recursive refinement can we judge their relative merits. Initially, the latter will be at a disadvantage; our judgement is, of course, affected by what we are used to, and most of us (author 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 three 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. 58]

$r := a;$
do $r \geqq d \rightarrow d\,d := d;$
do $r \geqq d\,d \rightarrow r := r - d\,d;\ d\,d := d\,d + d\,d$ **od**
od

Example 2 [1, p. 66]

$x := X;\ y := Y;\ z := 1;$
do $y \neq 0 \rightarrow$ **do** $2|y \rightarrow y := y/2;\ x := x * z$ **od**;
$y := y - 1;\ z := z * x$
od

Example 3 [1, p. 68]

$allsix := true;\ j := 0;$
do $j \neq n$ **and** $allsix \rightarrow allsix := f(j) = 6;\ j := j + 1$ **od**

Acknowledgement. I am grateful to Jim Horning. Art Sedgewick and Ed Ashcroft for discussions, to Edsger Dijkstra and Bob Tennent for letters, and to all of the above for pointing out errors in the first draft. Jim also suggested the title. I also wish to thank an anonymous referee (whose initials are EWD) for many valuable suggestions.

References

1. Dijkstra, E.W.: A Discipline of Programming. New Jersey: Prentice-Hall 1976
2. Dijkstra, E.W.: Guarded commands, non-determinacy, and formal derivation of programs. CACM **18**(8) p. 453, August 1975
3. Dijkstra, E.W.: A Short Introduction to the Art of Programming. Report EWD316, Technological University of Eindhoven, August 1971
4. Knuth, D.E.: Structured programming with go to statements. ACM Computing Surveys **6**(4), December 1974
5. Ledgard, H.F., Marcotty, M.: A genealogy of control structures. CACM **18**(11), November 1975
6. Scott, D.: Outline of a mathematical theory of computation. Proceedings of Fourth Annual Princeton Conference on Information Science and Systems, pp. 169–176, 1970
7. Wirth, N.: Program development by stepwise refinement, CACM **14**(4) pp. 221–227, April 1971

Received January 30, 1978/Revised January 5, 1979