Stanford Artificial Intelligence Laboratory
Memo AIM-307

Computer Science Department
Report No. STAN-CS-77-640

January 1978

STRUCTURED PROGRAMMING WITH RECURSION

by

Zohar Manna
Artificial Intelligence Lab
Stanford University
Stanford, CA.

Richard Waldinger
Artificial Intelligence Center
SRI International
Menlo Park, CA.

Research sponsored by

National Science Foundation
Office of Naval Research
Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-307

January 1978

Computer Science Department
Report No. STAN-CS-77-640

STRUCTURED PROGRAMMING WITH RECURSION

by

Zohar Manna
Artificial Intelligence Lab
Stanford University
Stanford, CA.

Richard Waldinger
Artificial Intelligence Center
SRI International
Menlo Park, CA.

This research was supported in part by the National Science Foundation under Grants DCR72-03737 A01 and MCS76-83655, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by a grant from the United States-Israel Binational Science Foundation (BS F).

There is a tendency in presentations of structured programming to avoid the use of recursion as a repetitive construct, and to favor instead the while statement, the guarded command, or other iterative loop constructs. For instance, in his recent book, "A Discipline of Programming" [2], Dijkstra bars recursion from his exemplary programming language; although he does not absolutely forbid recursion to the qualified practitioner, he warns that "I regard general recursion as an order of magnitude more complicated than just repetition," and declares that "I don't like to crack an egg with a sledgehammer, no matter how effective the sledgehammer is for doing so."

The method by which an iterative loop is to be constructed is clearly dictated: We are first to find an *invariant assertion,* a relation between the variables of the program, and a *termination function,* an expression involving the program's variables whose value is a nonnegative integer. The loop body is then constructed so as to reduce the value of the termination function while maintaining the truth of the invariant assertion upon each execution of the loop body. In this way, the correctness and termination of the resulting program are guaranteed by the nature of the construction process. The decision when to introduce a loop, and the choice of an appropriate invariant assertion and termination function, are not determined by the method; generally they are left to the Intuition of the programmer.

For example, in constructing a program to compute the exponential function $z = x^y$ of two integers x and y (where x is positive and y is nonnegative), Dijkstra recommends that we introduce new variables xx and yy, and take the invariant assertion to be

$$z \cdot xx^{yy} = x^y$$

and the termination function to be yy Itself, The invariant assertion is established initially by taking xx, yy, and z to be x, y, and 1, respectively; the task of the loop body is to maintain this invariant assertion while reducing the termination function yy to 0. Employing familiar properties of the exponential function, he derives the program *

```
$(xx\ yy\ z) \leftarrow (x\ y\ 1)$
while $yy \neq 0$
do $(yy\ z) \leftarrow (yy-1\ z \cdot xx)$ .
```

This program is transformed subsequently to a more efficient version.

---

*Actually, Dijkstra obtains the invariant in two stages: he first introduces a new variable h and attempts to maintain the invariant $h \cdot z = x^y$; he then replaces h by the term xx^{yy}. His final program is expressed in terms of the guarded command construct.

In discussing how such invariant assertions and termination functions are to be discovered, Dijkstra appeals to his "inventive powers" and uses phrases such as "my experience suggests . .." and "the trick is that . . . . ." Of course, the exponential is a familiar program, and these choices may-appear natural or even inevitable. But if we had never seen the program before, how would we know to select $z \cdot xx^{yy} = x^y$ as the invariant assertion while reducing the termination function yy to 0? Why not maintain $z+xx^{yy} = x^y$ while reducing xx to 0, or maintain $z^{yy} = x^y$ while reducing yy to 1, or even maintain $z^{(xx^{yy})} = x^y$ while reducing xx to 1 or yy to 0?

In general, at each stage in the derivation there are innumerable conditions and functions that could be adopted as the invariant assertion and termination function of a loop; only a few of these choices lead to the successful completion of a derivation. With so many plausible candidates to choose from, a correct selection requires an act of precognitive insight.

The answer, of course, is that we must defer the introduction of a loop until it Is forced upon us by the structure of the program's derivation. For this purpose, we have found recursion to be far more useful than any of the iterative constructs; a recursive call can be introduced when a recurrence is observed in the derivation. Applying this approach, we avoid the premature selection of an invariant assertion and termination function.

For example, let us again consider the construction of an exponential program $exp(x\ y)$, intended to achieve the goal of computing the expression x^y. Employing the same properties of the exponential function that Dijkstra applied in his derivation, we reduce our goal to the subgoal of computing the constant 1 in the case that y is 0, and to the subgoal of computing the expression $x \cdot x^{y-1}$ In the case that y Is positive. We observe that the subexpression x^{y-1} is an instance of the top-level goal expression x^y; at this point, we decide to introduce a recursive call $exp(x\ y-1)$ to compute this subexpression. This call cannot lead to a nonterminating computation, because the second argument $y-1$ of the recursive call is a nonnegative Integer less then the second input y. The final program obtained is thus

```
exp(x y) <= if y = 0
            then 1
            else x·exp(x y-1) .
```

This program, like its Iterative counterpart, is guaranteed to be correct by virtue of the way It was constructed, and can be transformed into a more efficient version In a subsequent optimization phase. This optimized verslon can be recursive or iterative.

In general, a recursive call is formed when a subgoal in the program's derivation is found to be an instance of a higher-level goal; the decision to introduce the recursive call, its form, and the choice of the termination function are all dictated by the structure of the derivation; the choice of the invariant assertion is avoided altogether.

Another example: In constructing a program to find the index of the maximum element of an array, we want to assign a value to a global variable i such that

$$a[i] \geq all(a[0:n]) \text{ and}$$
$$0 \leq i \leq n\ ,$$

where n is a nonnegative *Integer* and $a[0:n]$ is an array segment of $n+1$ numbers $a[0]$, $a[1]$, ..., $a[n]$. In other words, we need to achieve that $a[i]$ is greater than or equal to every element In the array segment and that i Is between 0 and n.

In approaching this problem, Dijkstra [2] produces the invariant assertion**

$$a[i] \geq all(a[0:j]) \text{ and}$$
$$0 \leq i \leq j \text{ and}$$
$$j \leq n\ ,$$

explaining: "A standard way of generalizing a relation is the replacement of a constant by a variable -- possibly with a restricted range," and adding that "the condition $j \leq n$ has been added in order to do justice to the finite domain" of the array segment $a[0:n]$. As to the termination function, he continues: "Again, my experience suggests to choose a monotonically decreasing function . . . n-j . . . In order to ensure thls monotonic decrease , , ., I propose to subject j to an increase by 1 . . . ." By application of the propertles of the natural numbers and the concept of the "weakest precondition," Dijkstra develops the program

```
$(i\ j) \leftarrow (0\ 0)$
while $j \neq n$
do it $a[i] \geq a[j]$
    then $j \leftarrow j+1$
    else $(i\ j) \leftarrow (j\ j+1)$ .
```

In our approach, we want to construct a program $maxi(a\ n)$, whose goal Is again to assign a value to i such that

-----

**Again we take certain liberties with Dijkstra's notation.

$$a[i] \geq all(a[0:n]) \; and$$
$$0 \leq i \leq n \,.$$

In the case that n is 0, this condition is satisfied by taking f to be 0; in the other case, the condition $a[i] \geq all(a[0:n])$ decomposes into the conjunction of two subgoal conditions,

$$a[i] \geq all(a[0:n-1]) \; and \; a[i] \geq a[n] \,.$$

(Other decompositions are possible; the final program derived depends on which decomposition is chosen), The first of these subgoals is an instance of the condition $a[i] \geq all(a[0:n])$, which is part of the top-level goal; we therefore attempt to achieve it with a recursive call $maxi(a\ n\text{-}1)$. Termination is ensured because the second argument n-1 of the recursive call is less than the second input n. The second subgoal $a[i] \geq a[n]$ is achieved without introducing any recursive calls. The final program obtained is

```
maxi(a n) <= if $n = 0$
             then $f \leftarrow 0$
             else maxi(a $n-1$)
                  if $a[i] < a[n]$ then $f \leftarrow n$ .
```

Note that our use of recursion as a repetitive construct in this program has not precluded the use of assignment statements or even global variables.

The *recursion-formation* technique illustrated by these simple examples is a basic principle of our approach to systematic program derivation. This approach, presented in detail in [6], was designed for automatic program-construction systems; therefore, even when applied by the human programmer, it cannot rely on any leaps of intuition. The recursion-formation approach does not always make the act of programming easy, but it does avoid the extraordinary feats of ingenuity characteristic of the invariant-assertion approach,

Not everyone concerned with programming methodology has been completely enamored of the invariant assertion as a means for program construction. For example, In [5], Knuth compares two iterative programs for a moderately complex task: one was developed by inventing an invariant assertion while the second was derived by first constructing a simple recursive program for the same task, and then transforming it. He observes that "the recursive program is trivially correct, and the transformations require only routine verification; by contrast, a mental leap is needed to invent [the invariant assertion]."

Some of the proponents of the "Structured Programming School" admit the use of recursion when it is especially called for; e.g., Wirth [7] advises that recursion is "primarily appropriate when the problem to be solved, or the function to be computed, or the data structure to be processed, are already defined In recursive terms." Some researchers, such as Gries [3] and Hehner [4], have praised the simplicity and clarity of recursive programs, while others, such as Burstall and Darlington [1], have found recursive programs to be easier to transform and manlpulate.

Our point here is different: recursive programs are not only simpler to understand and manipulate, but also are easier to construct, in that their formation does not require the premature invention of an invariant assertion. For all these reasons, recursion seems to be an ideal vehicle for systematic program construction. It is surprising that some of the advocates of structured programming have not adopted it with more enthusiasm: in their fidelity to iteration, they have been driven to resort to more dubious means.

REFERENCES

1. But-stall, R. M. and J. Darlington, *A transformation system for developing recursive programs*, JACM, Vol. 24, No. 1 (Jan. 1977), pp. 44-67.

2. Dijkstra, E. W., *A discipline of programming*, Prentice-Hall, Englewcod Cliffs, NJ, 1976.

3. Gries, D., *Recursion as a programming tool,* Technical Report, Department of Computer Science, Cornell University, Ithaca, NY, 1076.

4. Mehner, E. C. *R.*, do *considered* odt *A contribution to the programming calculus,* Technical Report, Computer Systems Research Group, University of Toronto, Toronto, Canada, March 1077.

5. Knuth, D. E., *Structured programming with* go to *statements*, Computlng Surveys, Vol. 6, No. 4 (Dec. 1974), pp. 261-301.

6. Manna, Z. and R. Waldinger, *Synthesis: dreams => programs.* Technical Report, Artificial Intelligence Lab., Stanford University, Stanford, CA and Artificial Intelligence Center, SRI International, Menlo Park, CA, Nov. 1977.

7. Wirth, N., *Algorithms + data structures = p:ograms*, Prentice-Hell, Englewood Cliffs, NJ, 1976.