PROGRAMMING BY INDUCTION

by

David Gries[†]
Cornell University

TR 71 - 106

September 1971

Computer Science Department
Cornell University
Ithaca, New York 14850

[†] This research was supported by NSF grant GJ-28176

PROGRAMMING BY INDUCTION

by

David Gries

Abstract

A technique for creating programs, called <u>programming by induction</u>, is described. The term is used because of the similarity between programming by induction and proving a theorem by induction.

PROGRAMMING BY INDUCTION

by

David Gries

This paper discusses and gives an example of a programming technique which we call programming by induction. It is a technique not only for proving that a program is correct, but a technique for actually developing or creating the programs.

Suppose we are given a problem involving a fixed integer n, where n might be the number of elements in a one-dimensional array, the number of arguments of a function, and so forth. We generally "program" the problem by considering the general case first and developing a program for it. We then prove to ourselves (or attempt to convince ourselves by running test cases) that the program does actually work the way it is supposed to. The hardest part is often to convince ourselves that it works for the "boundary" cases n=1 and $n = n_{max}$.

Programming by induction can be briefly described as follows. We first of all create a program for the case n=1 and convince ourselves that it is correct. Then we proceed essentially as we might when proving a theorem by induction. Create a program for the case n=2. Attempt to relate it to the original program for the case n=1; see if it is possible to construct the program for n=2 by systematically changing the program for n=1. This may of course requires changes in the program for n=1. If successful, this process may lead to an idea for a general "induction step": given a program for the case $n \geq 1$ with certain properties and structure, one can show how to construct a program for the case n+1 which satisfies the same properties and has the same structure. Hopefully, the construction will be simple enough so that the proof of correctness will be relatively easy.

Programming by induction may not produce the most efficient program - with respect to either time or space. It is certainly not a general method of creating programs like Dijkstra's structured programming [2] or equivalently Wirth's stepwise refinement [3], and cases of its usefulness will undoubtedly be few and far between. Its value lies mainly in that the "proof" of correctness of a program may be easier than if one programmed the conventional way. We do present below a problem which came up in our research [1] in which programming by induction was the only way we could see to solve it.

An Example. We will program the following process by induction, and then given a second program for it which was created in the conventional manner.

(1) We are given a single value v_0, a predicate P with one argument, a fixed integer $n > 0$, and a function f with n arguments. We are to write an ALGOL-like (compound) statement to generate possible values using v_0 and f, in any order we wish, until a value v_j is found such that $P(v_j)$ = false. (v_0 should also be tested.) We know nothing about P - it may be that $P(v)$ = false for only a single value v. Hence we must be sure that any given value v is generated using v_0 and f in a finite amount of time (this will become clearer later on).

Note that the infinite set E of all possible values can be described by the equation

$$E ::= v_0 \quad | \quad f(E, \ldots, E)$$

For n=1, the possible values are

$$v_0, \; v_1 = f(v_0), \; v_2 = f(v_1), \; \ldots, \; v_m = f(v_{m-1}) \; , \quad \ldots$$

A statement to perform (1) in the case n=1 is easily written:

```
(2)  BEGIN VALUE := v0;
       WHILE P(VALUE) DO
          VALUE := f(VALUE)
     END
```

The case n=2 is harder, mainly because we have to generate "all possible" values. What we mean by this is as follows. Suppose we have at some point generated and tested values $v_0, v_1, \ldots, v_m$ where $m \geq 0$. Then we must be sure that every value $f(v_i, v_j)$, $0 \leq i, j \leq m$, is generated and tested in a finite amount of time. For if not, then we may "miss" the (possibly) single value v such that P(v) = false.

A possible ordering for the generated values in the case n=2 is

$v_1 = f(v_0, v_0)$

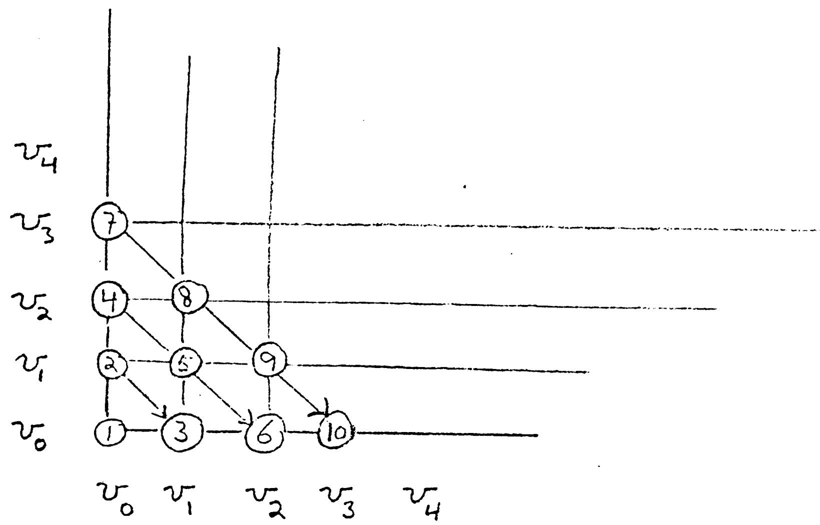
$v_2 = f(v_1, v_0), \quad v_3 = f(v_0, v_1)$

$v_4 = f(v_2, v_0), \quad v_5 = f(v_1, v_1), \quad v_6 = f(v_0, v_2), \quad \ldots$

$\vdots$

We describe this in diagram (3). Each row (column) specifies the value used for the first (second) argument of f, while the numbers at the grid points indicate the order in which new values are generated. Clearly, if we can "program" this ordering, we will generate and test "all possible" values.

(3)

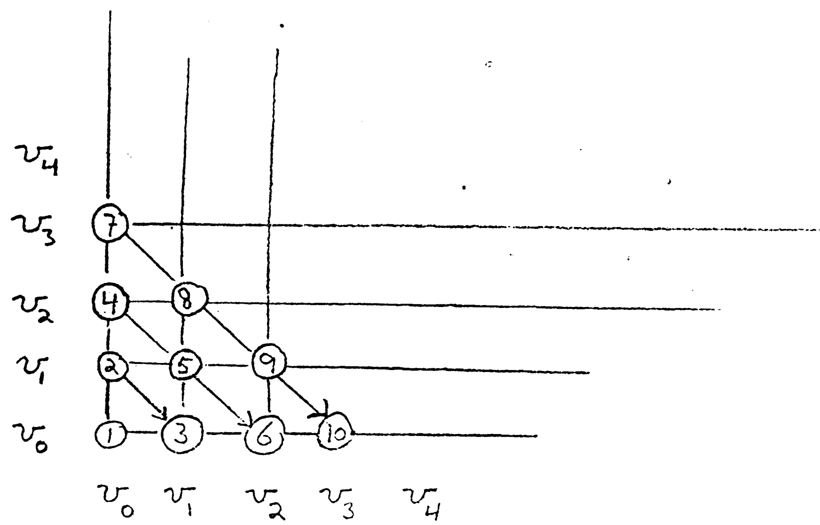


The program for the case n = 2 will obviously need at least a one-dimensional array A to store the values. [We assume that the lower bound of the subscript range of any one-dimensional array is 0, and that there is no upper bound.] A single variable I will indicate both how many values are in A and which value to test next using P. We can use two simple variables ROW and COL to indicate which values in A are to be used as arguments to f. Note that if we create v_i = f(A[ROW], A[COL]), then according to diagram (3), the next value to be generated is either

$$v_{i+1} = f(A[ROW-1], A[COL+1]) \text{ or } v_{i+1} = f(A[COL+1], A[0])$$

depending on whether ROW ≠ 0 or ROW = 0. The reader should convince himself that statement (4) does solve problem (1) for the case n = 2.

(3)



The program for the case n = 2 will obviously need at least a one-dimensional array A to store the values. [We assume that the lower bound of the subscript range of any one-dimensional array is 0, and that there is no upper bound.] A single variable I will indicate both how many values are in A and which value to test next using P. We can use two simple variables ROW and COL to indicate which values in A are to be used as arguments to f. Note that if we create v_i = f(A[ROW], A[COL]), then according to diagram (3), the next value to be generated is either

$$v_{i+1} = f(A[ROW-1], A[COL+1]) \text{ or } v_{i+1} = f(A[COL+1], A[0])$$

depending on whether ROW $\neq$ 0 or ROW = 0. The reader should convince himself that statement (4) does solve problem (1) for the case n = 2.

```
(4)  BEGIN
       I := 0; A[0] := v0;                          [Put in initial value.
       ROW := 0; COL := -1;                         [Initialize subscript
                                                     counters.
       WHILE P(A[I]) DO
         BEGIN
           IF ROW = 0 THEN                          [Get to next grid pt:
             BEGIN ROW:=COL+1; COL:=0 END            Start new diagonal or
           ELSE BEGIN ROW:=ROW-1; COL:=COL+1         go down current
                END;                                 diagonal.
           I := I+1; A[I] := f(A[ROW], A[COL])      [Insert new value.
         END
     END
```

Statement (4) gives us an idea for the induction step. Suppose we have a program for some integer n, n $\geq$ 1, in which the only way new values are introduced into A is in one place, through execution of

(5) I := I+1; A[I] := f(v_1, v_2, ..., v_n)

where the v_i are variables. This means that, "all possible" n-tuples of values must appear in (v_1, v_2, ..., v_n) as execution progresses. To perform the induction, we can replace (5) by a series of statements which

(a) Save the values (v_1, v_2, ..., v_n) in new arrays A1, A2, ..., An, by executing something like

J := J+1; A1[J] := v_1; ...; An[J] := v_n;

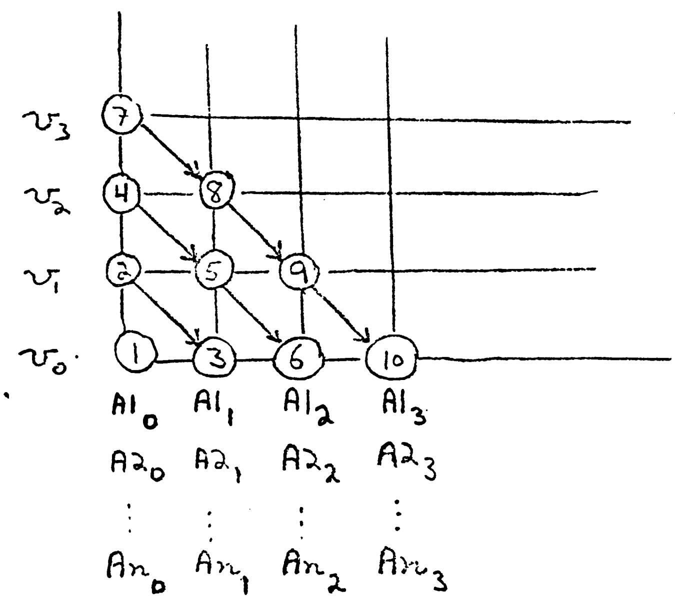
where J indicates how many n-tuples have been stored.

(b) Perform a second "diagonalization" using new subscript counters ROW1 and COL1 to reference an (n+1)-tuple by

```
I := I+1;
A[I] := f(A[ROW1], A1[COL1], ..., An[COL1]);
```

Thus we are using diagram (6), where the rows represent single values, and the columns n-tuples of values. Since "all possible" n-tuples are put in arrays A1, ..., An, we see that "all possible" (n+1)-tuples will be used as arguments to f.

(6)



With this idea in mind, let us proceed with the exact formulation of the induction step. First, in order to get the same "structure" for the cases n = 1 and n = 2, we rewrite statement (2) for the case n=1 to use an array:

(7)
```
BEGIN I := 0; A[0] := v0;          [Put in initial value.
  WHILE P(A[I]) DO
    BEGIN
      I := I+1; A[I] := f(A[I-1])   [Insert new value.
    END
```

Now suppose we have a statement for the case $n > 0$ with the form (8) which has the following properties;

1) $S_1, \ldots, S_k, S_p, \ldots, S_q$ are assignment statements, or conditional statement which contain only assignment statements.

2) P is not referenced in statements $S_1, \ldots, S_k, S_p, \ldots, S_q$.

3) I is never changed by statements $S_1, \ldots, S_k, S_p, \ldots, S_q$.

4) The array A is not changed by $S_1, \ldots, S_k, S_p, \ldots, S_q$.

5) Statement (8) works as desired for case n. This implies that "all possible" n-tuples of generated values appear in the (subscripted) variables $v_1, v_2, \ldots, v_n$ at some time.

```
(8) BEGIN I := 0; A[0] := v0;                  [Put in initial value.
      S1; ...; Sk;
      WHILE P(A[I]) DO
        BEGIN Sp; ...; Sq;
          I := I+1; A[I] := f(v1, ..., vn)   [Insert new value.
        END
    END
```

Note that statement (7) for the case $n = 1$ satisfies these conditions; no statements $S_1, \ldots, S_k, S_p, \ldots, S_q$ appear at all. To construct a statement which works for the case n+1, we perform the following. Use new arrays A1, ..., An and three new simple variables J, ROW, and COL, and rewrite (8) as

```
(9) BEGIN I := 0; A[0] := v0;                        As before.
      S1; ...; Sk;
      ROW := 0; COL := -1;                           Initialize subscript
                                                     counters.

      J := -1;                                       Initialize counter for
                                                     new arrays.
      WHILE P(A[I]) DO                               As before.
        BEGIN Sp; ...; Sq;
          J := J+1;                                  Put n-tuple into
          A1[J] := v1; ...; An[J] := vn;             arrays A1, ..., An.
          IF ROW = 0 THEN                            Get to next grid pt:
               BEGIN ROW:=COL+1; COL:=0 END          start new diagonal, or
          ELSE BEGIN ROW:=ROW-1; COL:=COL+1          go down current
               END;                                  diagonal.
          I := I+1;                                  Add new value.
          A[I] := f(A[ROW],A1[COL],...,An[COL])
        END
    END
```

Note that the new statement (9) has form (8), and satisfies at least the stated properties 1-4. To see that is satisfies (1) for the case n+1, note first of all that the arrays A1, ..., An will contain all possible n-tuples of values that can be generated, and that A contains all possible values. Then note that (9) does indeed generate the values as described by diagram (6).

<u>A conventional program for (1)</u>. The reader may complain that (1) could have been programmed more easily using conventional techniques. Indeed, I believe that statement (10) also performs (1) (I have not <u>proved</u> it completely, but I am inclined to think it will work). Array elements S[1], ..., S[n] are used to hold subscript values to reference n-tuples A[S[1]], ..., A[S[n]], and an n-dimensional "diagonalization" scheme is used to vary the subscript values.

Note that (10) was not <u>created</u> by induction; we tried to write a single, general statement which holds for any n. We may have to <u>prove</u> that it is correct by induction on n, however, Statement (10) uses 2 arrays, for any fixed integer n, while statement (8) uses (n-1)+(n-2)+(n-3)+...+1 arrays!

```
(10)  BEGIN
        I := 0; A[0] := v0;                  [Put initial value in.
        S[1] := 0; ...; S[n-1] := 0;         [Initialize array of
        S[n] := -1;                          [subscript values.
        WHILE P(A[I]) DO
          BEGIN S[n] := S[n]+1;               [Fix array of subscript
            J := n-1;                         [values for next
            WHILE J > 0 ∧ S[J] = 0 DO         [function
              BEGIN S[J] := S[J+1];           [evaluation.
                    S[J+1] := 0;
                    J := J-1
              END;
            IF J > 0 THEN S[J]:=S[J]-1;
            I := I+1;                         [Insert new value.
            A[I] := f(A[S[1]],...,A[S[n]])
          END
      END
```

<u>A more difficult example</u>. Actually, our original need for designing programming by induction in [1] was caused by a condition imposed on the form of the statement which performs (1):

(11) Within the statement which performs (1), no "testing" can be performed (except of course the test P(A[I])). Thus, for the case n = 2, statement (4) is not allowed, since it contains a test "IF ROW = [illegible] ...".

At first it was not at all clear that a statement which performs (1) and also satisfies (11) could even be programmed. A neat programming trick, the original version of which was due to Constable [1], gives us a statement for the case n = 2. This statement still generates values in the order described by diagram (3). We can then use the same idea to perform the induction step; this is quite easy, and we leave it to the reader.

We were not able to create a statement for (1) satisfying (11) using conventional programming methods (we had to be sure the statement worked, and running test cases was not enough).

Now let us change (4), the statement for the case n = 2, so that it satisfies (11). We will do this in three steps, so that things remain clear. Remember, values will be generated in the same order, as indicated by diagram (3).

First of all, it will be advantageous to move the incrementation of COL till <u>after</u> a new value is generated within the WHILE loop. This requires us to change the initialization of COL and the use of COL within the conditional statment "IF ROW = 0 THEN ...". Secondly, we use an array DOWN which will satisfy the property DOWN[j] = j-1 for $0 \leq j \leq I$. This allows us to replace ROW := ROW-1 by ROW := DOWN[ROW]. These changes yield statement (13), which still performs (1).

```
(13)  BEGIN
        I := 0; A[0] := v0;              [Initialize fisrt value
        DOWN[0] := -1;                   [and auxiliary array.
        ROW := 0; COL := 0;              [Initialize subscripts.
        WHILE P(A[I]) DO
          BEGIN
            IF ROW = 0 THEN              [Get to next row:
                 BEGIN ROW:=COL; COL:=0 END  [start a new diagonal,
            ELSE ROW := DOWN[ROW];       [go down current one.
            I := I+1;                    [Add a new value and
 4:         A[I] := f(A[ROW], A[COL]);   [fix corresponding
            DOWN[I] := I-1;              [DOWN value.
            COL := COL+1;                [Increase column count.
          END
      END
```

The second step is to consider COL to be an array, instead of a simple variable. We want to replace the line labeled 4:

A[I] := f(A[ROW], A[COL])

by A[I] := f(A[ROW], A[COL[ROW]]).

Thus, when ROW indicates the first argument to f, COL[ROW] indicates the second. For any fixed value "*row*" of ROW, the values produced using as the first argument to f are, in order,

f(A[*row*], A[0])

f(A[*row*], A[1])

f(A[*row*], A[2])

⋮

(See diagram (3).) Hence we must initialize COL[*row*] to 0 when we put a value into A[*row*], and after each use of A[*row*] as the first argument, we must add 1 to COL[*row*]. We wind up with the equivalent statement (14), where the labeled lines were changed.

```
(14)  BEGIN
        I := 0; A[0] := v0;                           Initialize first value
        DOWN[0] := -1;                                and auxiliary arrays, and
      1: ROW := 0; COL[0] := 0;                       Init. row counter
        WHILE P(A[I]) DO
          BEGIN
            IF ROW = 0 THEN                           Get to next row.
      2:          ROW := COL[0]
            ELSE ROW := DOWN[ROW];
            I := I+1;                                 Add new value to A
      4:    A[I] := f(A[ROW],A[COL[ROW]]);            and fix corresponding
            DOWN[I] := I-1;                           auxiliary array
      5:    COL[I]  := 0;                             elements.
      6:    COL[ROW]:= COL[ROW]+1;                    Fix ROW's column count.
          END
      END
```

We are finally ready to delete the test "IF ROW = 0 ....". Note that the affect of these is to put either COL[0] or DOWN[ROW] into ROW, depending on whether ROW = 0 or not. Execut following two statements has the same effect:

(15) DOWN[0] := COL[0]; ROW := DOWN[ROW];

The other effect of executing these is to put a quantity into DOWN[0]. Since DOWN[0] is never referenced except in this context when ROW = 0, this assignment has no other effect on the outcome of the statement (14). Hence we can replace the conditional statement in (14) by (15), yielding statment (16).

```
(16)  BEGIN
        I := 0; A[0] := v0;                        Put in initial value and
        DOWN[0] := -1;                             fix auxiliary arrays.
        ROW := 0; COL[0] := 0;                     Initialize row counter.
        WHILE P(A[I]) DO
          BEGIN
            DOWN[0] := COL[0];                     Get to next row.
            ROW := DOWN[ROW];
            I := I+1;                              Add a new value
            A[I] := f(A[ROW], A[COL[ROW]]);        to A.
            DOWN[I] := I-1;
            COL[I] := 0;
            COL[ROW] := COL[ROW]+1;                Fix ROW's columns.
          END
      END
```

REFERENCES

[1] Constable, R., and Gries, D. On classes of program schemata. TR 71-105, Computer Science Dept., Cornell University, August 1971.

[2] Dijkstra, E. W. Notes on structured programming. EWD 249, Technical University Eindhoven, The Netherlands, 1969.

[3] Wirth, N. Program development by stepwise refinement. Comm. of the ACM 14 (April 1971), 221-227.