A NOTE ON ITERATION

David Gries

TR77-323

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

A Note on Iteration

by

David Gries
Computer Science Department
Cornell University
September 1977

The iterative statement is an essential feature of most programming languages, and virtually every program uses it. Yet, iteration is more difficult to understand than assignment, sequencing and alternation, because it requires the use of mathematical induction in one form or another. This importance of iteration and relative difficulty in understanding it suggests that we be careful in choosing a particular notation to use in developing iterative algorithms. Some of the criteria that could be used to compare notations are the following:

> The particular form of iteration should be an aid in developing algorithms -- there should be a systematic methodology to guide program development.
>
> The amount of formal reasoning needed to prove an iterative statement correct should be as small as possible. This is based on the assumption that formal reasoning is directly related to informal reasoning about programs.
>
> The iterative notation should allow the concise expression of most -- if not all -- of the algorithms we wish to write.

This work was supported by the National Science Foundation under grant MCS76-22360. The discussion following a presentation of this material at an IFIP WG2.3 meeting in August 1977 prompted me to put these comments in writing. Thanks go to Gary Levin who commented on a draft of this note, and who helped develop the second example while we were trying to understand the program used in "proving" the four-color conjecture.

This note presents some arguments and examples to support my opinion that Dijkstra's guarded command loop [1] is superior to more conventional forms of iteration (e.g. the <u>while</u> loop, with or without exits).

Dijkstra's calculus for the derivation of programs certainly provides the systematic methodology mentioned in our first criterion listed above. With respect to the second criterion, we will see that at least some algorithms that require nested loops in conventional notation can be systematically and effectively developed as a single loop using guarded commands. Thus, only one loop invariant (rather than two or three) is necessary, and it is easier to show that the single invariant is indeed invariant.

We note that the conventional loop <u>while</u> B <u>do</u> S can be written in guarded command notation as <u>do</u> B→S <u>od</u> ; hence the guarded command notation subsumes the <u>while</u> loop notation without exits. The examples given below are intended to convince the reader that the awkward <u>exit</u> statement, which is often found to be necessary when using the <u>while</u> loop, is usually unnecessary when using guarded commands. We also show, with an old example, the use of nondeterminism in developing concise algorithms in guarded command notation.

<u>An example</u>

In searching a two-dimensional array b for the presence of a value x, we are attempting to determine whether the following predicate is true or false:

(1) $\underline{E}i$: $\underline{E}j$: x=b(i,j)

Typically, the program is required to store in two simple variables i and j the row and column number where x appears, or to give some indication if x is not in the array. A precise formulation of this problem is:

(2) For fixed n,m$\geq$0, x, and b(0:n-1,0:m-1), establish the truth of
(0$\leq$i<n <u>and</u> 0$\leq$j<m <u>and</u> x=b(i,j)) <u>or</u> (i=n <u>and</u> x$\notin$b)

This task is usually performed by a program consisting of two nested loops, using either an exit statement or a variable <u>present</u> to aid in terminating the loops.

To develop the algorithm, we assume iteration will be used and develop the invariant for the loop, based on the idea that we will search row 0 for x, then row 1 and so on. We traditionally search rowwise rather than columnwise, perhaps because we talk of "rows and columns" rather than "columns and rows", and because predicate (1) gives preference to the row number i. During "execution" of the loop, b(i,j) will be the next element of b to test, and x will not appear "before" b(i,j) in the array. We express this in the three-part invariant

(3) (1) $0 \leq i \leq n$ <u>and</u> $0 \leq j \leq m$
    (2) $x \notin b(0:i-1, 0:m-1)$ (x is not in rows 0 through i-1)
    (3) $x \notin b(i, 0:j-1)$ if $i<n$ (x is not "before" position j in row i)

The relation $j \leq m$ is needed, rather than $j<m$, because we would like to initially establish the invariant using i,j:=0,0, and m might be zero.

The purpose of each command of a guarded command of a loop is to proceed towards termination, while the purpose of the guard is to describe those states in which execution of the command will keep the invariant true. Obvious possibilities for commands are j:=j+1 (proceed to next column in row i) and i,j:=i+1,0 (proceed to the next row). The corresponding guards are easily determined, yielding the algorithm

```
(4)  i,j:= 0,0;
     do i≠n cand j≠m cand x≠b(i,j)  → j:= j+1
      ▯ i≠n cand j=m                → i,j:= i+1,0
     od
```

It is easy to see that the complement of the guards together with invariant (3) imply the desired result (2).

Should one wish to, one could use the decreasing function $n(m+1) - i(m+1) - j$ in order to prove termination; I have not done so because termination was not really a problem here.

The algorithm does require the use of "conditional and" <u>cand</u>, which can be defined as

u <u>cand</u> v ≡ <u>if</u> u <u>then</u> v <u>else</u> false

I am quite willing to use <u>cand</u> (and also <u>cor</u>), for with this small addition we have achieved a notation which does not require nested loops or exit statements in this case. Note that both the Hoare-Wirth contribution to the development of ALGOL [2] and EUCLID [3] support <u>cand</u> instead of <u>and</u>.

The first guard in the loop of algorithm (4) could have been written as $(i \neq n$ <u>and</u> $j \neq m)$ <u>cand</u> $x \neq b(i,j)$. The overspecification was used in (4) to make the guard easier to understand.

<u>A slight modification of example 1</u>

Suupose that each row is known to be nonempty -- that $m>0$ instead of $m \geq 0$. With this extra information we might use the same invariant, but with $j<m$ instead of $j \leq m$. The algorithm would then be

```
i,j:= 0,0;
do i≠n cand j≠m-1 cand x≠b(i,j) → j:= j+1
 ▯ i≠n cand j=m-1 cand x≠b(i,j) → i,j:= i+1,0
od
```

<u>A second example</u>

We were recently led to consider writing an algorithm for a problem which can be simplified as follows. Consider triples $(\tilde{p},\tilde{q},\tilde{r})$ where $0 \leq \tilde{p} < pt$, $0 \leq \tilde{q} \leq qt$, $0 \leq \tilde{r} \leq rt$, and where $0 \leq pt$, $0<qt$, $0 \leq rt$. An algorithm is required to store in variable y the value of the predicate

(5) $\quad \underline{E}\tilde{p}\colon \underline{A}\tilde{q}\colon \underline{E}\tilde{r}\colon P(\tilde{p},\tilde{q},\tilde{r})$

where P is a fixed predicate and $\tilde{p},\tilde{q},\tilde{r}$ have the ranges given above. (Unless otherwise stated, all quantifiers on $\tilde{p}$, $\tilde{q}$, and $\tilde{r}$ have these ranges.) Thus, execution of the algorithm should establish the truth of

(6) $\quad y = (\underline{E}\tilde{p}\colon \underline{A}\tilde{q}\colon \underline{E}\tilde{r}\colon P(\tilde{p},\tilde{q},\tilde{r}))$

If we view P as a three-dimensional array with rows, columns and, let's say, files, then the purpose of the algorithm is to determine whether or not there exists a row such that for each column in that row there is at least one file which contains the value true. With this formulation, the reader will notice the similarity between this problem and the first one -- to determine whether or not $(\underline{E}i\colon \underline{E}j\colon x=b(i,j))$ is true.

One idea is to examine the triples $(\tilde{p},\tilde{q},\tilde{r})$ in essentially lexicographic order -- $(0,0,0),\ldots,(0,0,rt),(0,1,0),\ldots$ -- until the truth or falsity of (5) is determined. Thus we use three variables p, q, r to denote the current triple to be examined, and we expect to initialize them with p,q,r:=0,0,0. Possible commands to progress towards termination are r:=r+1, q,r:=q+1,0 and p,q,r:=p+1,0,0.

Once the answer is determined, the loop should terminate. Thus the invariant should indicate that based just on the elements examined so far, no answer could be determined. The invariant is

(7) (1) $0\leq p\leq pt,\ 0\leq q\leq qt,\ 0\leq r\leq rt$

(2) $\not{\underline{E}}\tilde{p}\colon 0\leq\tilde{p}<p\colon\ \underline{A}\tilde{q}\colon \underline{E}\tilde{r}\colon P(\tilde{p},\tilde{q},\tilde{r})$

(3) $\underline{A}\tilde{q}\colon 0\leq\tilde{q}<q\colon \underline{E}\tilde{r}\colon P(p,\tilde{q},\tilde{r})$

(4) $\not{\underline{E}}\tilde{r}\colon 0\leq\tilde{r}<r\colon P(p,q,\tilde{r})$

With this invariant and proposed commands we readily develop the guards, and end up with the algorithm

```
(8)   p,q,r:= 0,0,0;
      do r≠rt and not P(p,q,r)            → r:= r+1
       ▯ q≠qt and P(p,q,r)                → q,r:= q+1,0
       ▯ p≠pt and r=rt and not P(p,q,r)   → p,q,r:= p+1,0,0
      od;
      {P(p,q,r) ≡ predicate (5)}
      y:= P(p,q,r)
```

The guards were fairly easily determined; for example the command r:=r+1 could only affect parts (1) and (4) of the invariant since r does not appear in the other two.

The reader will note that the algorithm to establish (6), but with any quantifier $\underline{E}$ or $\underline{A}$ replaced by $\underline{A}$ or $\underline{E}$ respectively, will have exactly the same form and commands; only the guards will change.

<u>A third example</u>

This problem (but not the solution) was first shown to me and others in July 1975 by Dijkstra. I include it as an excellent illustration of the use of the calculus for the derivation of programs and of nondeterminism.

Consider three functions f, g, h defined on the nonnegative integers, with the property

$f(i) \le f(i+1),\ g(i) \le g(i+1),\ h(i) \le h(i+1)$ for $0 \le i$.

It is known that there exists at least one value x, together with three integers i, j, k, such that $x=f(i)=g(j)=h(k)$. Call the least such value $\bar{x}$, and the corresponding smallest possible integers $\bar{i},\bar{j},\bar{k}$. The problem is to find $\bar{x}$ -- that is, to establish the truth of

(9) $f(i)=g(j)=h(k)=\bar{x}$ <u>and</u> $i=\bar{i}$ <u>and</u> $j=\bar{j}$ <u>and</u> $k=\bar{k}$,

where i,j and k are the program variables whose values are to be determined. Note that $\bar{x},\bar{i},\bar{j}$ and $\bar{k}$ can be used in the proof of correctness, but not in the program itself, since they are the unknowns to be found.

Because we want to find the least such value x, it makes sense to begin at the beginning -- with i,j,k equal to zero -- and to increase them in some manner until $\bar{x}$ is detected. Thus, the simplest decreasing function to be used in proving termination of the algorithm is

$(\bar{i}+\bar{j}+\bar{k}) - (i+j+k)$.

One possible way to develop an invariant for a loop to perform the desired function is to weaken the result assertion (9). We do this by replacing the equalities of (9) by "weaker" comparisons, yielding the following invariant P:

$f(i) \le \bar{x}$ <u>and</u> $g(j) \le \bar{x}$ <u>and</u> $h(k) \le \bar{x}$ <u>and</u> $0 \le i \le \bar{i}$ <u>and</u> $0 \le j \le \bar{j}$ <u>and</u> $0 \le k \le \bar{k}$

A possible command to decrease the function given above is i:=i+1. We determine the precondition such that execution of i:=i+1 will yield the invariant P as follows:

wp("i:=i+1", P) = ($f(i+1) \le \bar{x}$ <u>and</u> $g(j) \le \bar{x}$ <u>and</u> $h(k) \le \bar{x}$ <u>and</u>
$0 \le i+1 \le \bar{i}$ <u>and</u> $0 \le j \le \bar{j}$ <u>and</u> $0 \le k \le \bar{k}$

Knowing that P is itself a precondition of each command of the loop, we determine that the following guard will suffice:

$f(i)<g(j)$ <u>or</u> $f(i)<h(k)$

That is, P together with this guard yields ($f(i)<g(j) \le \bar{x}$ <u>or</u> $f(i)<h(k) \le \bar{x}$) <u>and</u> P, which implies wp("i:=i+1",P). By symmetry, we end up with the algorithm

```
i,j,k:= 0,0,0;
do f(i)<g(j) or f(i)<h(k) → i:= i+1
 [] g(j)<h(k) or g(j)<f(i) → j:= j+1
 [] h(k)<f(i) or h(k)<g(j) → k:= k+1
od
```

Using the decreasing function $(\bar{i}+\bar{j}+\bar{k}) - (i+j+k)$ we immediately see that the algorithm will terminate. It remains to show that upon termination (9) does indeed hold. Suppose all guards are false. Then the falsity of the first part of all guards yields

(10) $f(i) \geq g(j) \geq h(k) \geq f(i)$

and so $f(i)=g(j)=h(k)$. This, together with the invariant and the fact that $\bar{x}$ is the least value in the range of all three functions, yields the result.

Note, however, that only the first part of each guard was used in proving that the result held upon termination; hence we may discard the second part of each guard, thereby strengthening the guards, yielding the algorithm

```
i,j,k:= 0,0,0;
do f(i)<g(j) → i:= i+1
 [] g(j)<h(k) → j:= j+1
 [] h(k)<f(i) → k:= k+1
od
```

The nondeterminacy is crucial for expressing the algorithm so elegantly and concisely. (I once saw an ALGOL 68 programmer write this algorithm using two procedures, in about 30 lines of program.)

Discussion

Each of the algorithms presented would require less formal reasoning than would their equivalent algorithms in a more conventional notation. What is also important is the systematic method that was applied in developing the algorithms - a method which I feel we will be able to teach to students at some point. I don't mean to say that this is the only way to develop an algorithm.

One may argue that the nonexistence of an implementation of guarded commands is a serious disadvantage, so much so as to bar its use. However, rather than develop an algorithm in conventional notation in a conventional _ad hoc_ way, I would rather attempt to use a systematic method to guide me to a correct, readable, and perhaps elegant algorithm, and then translate it into the language at hand if necessary.

We should not let the languages that are currently implemented hinder algorithmic development and the communication of algorithms, any more than they have to.

With some thinking, one could probably implement the guarded command loop effectively. For example, a compiler should be able to translate algorithm (4) into the equivalent pair of nested loops that programmers now write. Thus, the _while_ loop with exits, etc. would become an implementation technique rather than a program development tool.

Finally, paper [4] has discussed various control structures in an attempt to evaluate the effectiveness of "structured programs". The fact that the guarded command notation was not even mentioned in [4] leads me to question the relevance of the findings in that paper.

References

1. Dijkstra, E. W. _A Discipline of Programming_. Prentice Hall, Englewood Cliffs, N. J., 1976.
2. Wirth, N. and Hoare, C.A.R. A contribution to the development of ALGOL. CACM 9 (June 1966), 413-432.
3. London, R.L. et al. Proof rules for the programming language EUCLID. USC Inf. Sciences Institute, 1977.
4. DeMillo, R.A, Eisenstatt, S.C. and Lipton, R.J. Can structured programs be efficient? SIGPLAN Notices 10 (Oct 1976), 10-18.