Jumping About and Getting into a State

L. V. Atkinson
Department of Computer Science, University of Sheffield, Sheffield S10 2TN, UK

A number of recently published letters have presented alternative styles for programming a simple multi-exit loop. Some comments are made upon these styles and a further alternative is suggested—that of using state indicators.

1. INTRODUCTION

Five letters[1–5] published in recent issues of *The Computer Journal* form a chain of references leading back to a paper by Arblaster *et al.*[6] and, in particular, to one example drawn from Knuth.[7] Knuth adopted an *ad hoc* Algol-like language and presented the following program fragment.

Knuth1

```
FOR i := 1 STEP 1 UNTIL m DO
  IF a[i] = x THEN GOTO found FI;
not found: i := m + 1;   m := i;
          a[i] := x;   b[i] := 0;
   found: b[i] := b[i] + 1
```

Its purpose is to seek a value x within the first m elements of an array a, dimensioned from 1 to *max*. If x is not present then it is to be inserted as an additional entry. Each element $b[i]$ of the array b records how many times the value $a[i]$ has been sought.

The opinion expressed by Arblaster and colleagues is that the alternative versions of this program, with the explicit jump removed, are no easier to understand. This view is not shared by some of the authors of the recent letters. Each of the first three letters criticizes the previous solution and suggests a 'superior' alternative. These alternatives are shown in Fig. 1. The authors did not adopt the same syntax conventions and so, to standardize notation, the fragments are presented here in Pascal. Missala and Rudnicki took m to indicate the first free slot rather than the last one occupied; for compatability, their fragment has been modified in Fig. 1.

Hill makes the point that his fragment is easier to understand by virtue of its having only one entry point: the top. Knuth's original fragment, on the other hand, has three possible entry points because, in addition to natural entry from the top, control can enter via either of the two labels *found* and *not found*. Thus one cannot be certain of the effects of Knuth's fragment without examining the whole section of program for which the two labels are in scope.

Knuth presents the following alternative to his original version:

Knuth2

```
i := 1;
WHILE i ≤ m AND a[i] ≠ x DO i := i + 1;
IF i > m THEN m := i;   a[i] := x;   b[i] := 0 FI;
b[i] := b[i] + 1
```

This assumes the existence of a sequential conjunction operator *AND* which evaluates its second operand only if the first is *true*. Such a definition is necessary because, if the array a is full ($m = max$) but does not contain x, an attempt will be made to access an element of a which does not exist. Knuth's original notation is retained because a direct transliteration into Pascal would not produce a portable program; the Pascal Standard[8] deems the order of evaluation of operands of a dyadic operator to be implementation dependent. The relative merits of the sequential conjunction definition of the Boolean operator *AND* have been debated by Barron and Mullins[9] and Sale.[10] Robinson's suggestion, which is almost equivalent to **Knuth2**, avoids this implementation dependence but, as pointed out by Missala and Rudnicki, fails to cater for the case when a is empty ($m = 0$).

It is perhaps worth noting that the need for a sequential conjunction operator does not arise in a language which provides conditional expressions. For example, in Algol 60, the while-loop of **Knuth2** can be written as follows:

while (if $i \leq m$ **then** $a[i] \neq x$ **else** *false*) **do** $i := i + 1$

However, such constructs, especially when extended to include several tests, can impede program transparency.

Hill

```
i := 1;   notfound := true;
WHILE notfound AND (i < = m) DO
BEGIN
  notfound := a[i] < > x;
  IF notfound THEN i := i + 1
END;
IF notfound THEN
BEGIN
  m := i;   a[m] := x;   b[m] := 1
END ELSE
  b[i] := b[i] + 1
```

Robinson

```
i := 0;
REPEAT
  i := i + 1
UNTIL (a[i] = x) OR (i = m);
IF a[i] = x THEN b[i] := b[i] + 1 ELSE
BEGIN
  m := m + 1   a[m] := x;   b[m] := 1
END
```

Missala & Rudnicki

```
a[m + 1] := x;   b[m + 1] := 0;   i := 0;
REPEAT
  i := i + 1
UNTIL a[i] = x;
IF i > m THEN m := i;
b[i] := b[i] + 1
```

Figure 1

CCC–0010–4620/84/0027–0042$02.50

Linear search of a linked list, an application analogous to linear search of an array, is considered by Jensen and Wirth.[11] We are given a (possible empty) list of records (of type *person*) and wish to locate some specified key (a social security number *n*). Each record has the following form:

```
ptrtoperson = ↑person;
person = RECORD
           ss: integer;
           . . .
           next: ptrtoperson
         END {person};
```

Recognizing the possibility that the sought key may be absent, a solution requiring a sequential conjunction operator is suggested:

```
VAR
  p, first: ptrtoperson;
  . . .
p := first;
WHILE (p < > NIL) AND (p↑ . ss < > n) DO
  p := p↑ . next
```

However, because such a Pascal solution is implementation dependent, two alternatives are suggested. One is effectively equivalent to Knuth's first version of the array search and the other is as follows:

```
p := first;  b := true;
WHILE (p < > NIL) AND b DO
  IF p↑ . ss = n THEN b := false ELSE p := p↑ . next;
IF b THEN {key absent} . . . ELSE {key found} . . .
```

Hill has adopted this same technique in his program to search an array. Interestingly, like Wirth, Hill has chosen to use a Boolean variable which adopts the value *true* when the value sought has *not* been found. This involves a double negative because, when determining the condition under which control reaches the statement

```
b[i] := b[i] + 1
```

one concludes that this is when the sought value is NOT *notfound*—in other words, when it is found! This aspect has been criticized elsewhere[12] and the suggestion made that program transparency is improved when the *truth* of a Boolean variable implies a *successful* outcome. A version of Hill's fragment applying *positive thinking* follows.

```
i := 1;  found := false;
WHILE NOT found AND (i < = m) DO
  IF a[i] = x THEN found := true ELSE i := i + 1;
IF found THEN b[i] := b[i] + 1 ELSE
BEGIN
  m := 1;  a[m] := x;  b[m] := 1
END
```

2. USE OF A SENTINEL

Turning to considerations of efficiency, Knuth illustrates use of the sought value as a data sentinel in the array.

Knuth3

```
a[m + 1] := x;  i := 1;
WHILE a[i] ≠ x DO i := i + 1;
IF i > m THEN
  m := i;  b[i] := 1
ELSE b[i] := b[i] + 1 FI
```

The fragment presented by Missala and Rudnicki is similar to this and Inglis makes the interesting observation that Missala and Rudnicki criticize Robinson for not catering for the case where *a* is empty and yet, themselves, present a fragment which fails when *a* is full!

3. A FULL ARRAY

None of the fragments discussed considers the possibility that *a* may be full but, as Knuth points out, each program must include a check of the form

```
IF m = max THEN error
```

before attempting to reference *a*[*max* + 1] or *b*[*max* + 1]. Including this test in the positive version of Hill's fragment, we get the following:

Hill*

```
i := 1;  found := false;
WHILE NOT found AND (i < = m) DO
  IF a[i] = x THEN found := true ELSE
    i := i + 1;
IF found THEN b[i] := b[i] + 1 ELSE
  IF m = max THEN error ELSE
  BEGIN
    m := i;  a[m] := x;  b[m] := 1
  END
```

4. TRANSPARENCY OF MULTI-EXIT LOOPS

Our concern here is not to compare one particular search algorithm with another, such as linear search with and without a sentinel. Rather, we concentrate upon the transparency and security of implementation of multi-exit loops, particularly when two loop termination tests are dependent in that the validity of one depends upon the outcome of another. To this end, the linear search of an array, without the use of a sentinel, will suit our purpose initially. There are two reasons for exiting the loop (the sought entry has been found or an unsuccessful exhaustive search has been performed) but, when the process is programmed as in **Knuth2**, the two loop termination tests are dependent. To avoid subscript overflow, values of *a*[*i*] and *x* can be compared only if $i \leq m$.

Using Pascal, there is one further approach, applicable to all multi-exit loops and which we have not yet examined—the use of *state indicators*.

5. STATE INDICATORS

Knuth discusses the application of *event indicators* as described by Zahn.[13] Using Pascal's enumerated types, we can simulate Zahn's event indicators by using a variable to record any *state transition* of interest. The application of such state indicators has been suggested by the present author[12,14,15] with reference to a number

of examples including linear search of a list and of an array as well as to earlier work by Arblaster's colleagues.

For a loop with n possible reasons for exit, there are $n + 1$ states of interest during execution of the loop: either the loop is still repeating or one of the n reasons for stopping has been encountered. Using Pascal's enumerated types, a name can be given to each of the states. For the array search example there are three states of interest:

(i) I haven't found it yet but I'm still looking;
(ii) got it;
(iii) I've looked everywhere but it's not here.

We can introduce a state variable

```
VAR
  state: (searching, gotit, givenup);
```

and produce the following fragment:

Atkinson1

```
IF m = 0 THEN state := givenup ELSE
BEGIN
  state := searching;  i := 1;
  REPEAT
    IF a[i] = x THEN state := gotit ELSE
      IF i < m THEN i := i + 1 ELSE
        state := givenup
  UNTIL state < > searching
END;
CASE state OF
  gotit: b[i] := b[i] + 1;
  givenup:
    IF m = max THEN error ELSE
    BEGIN
      m := m + 1;  a[m] := x;  b[m] := b[m] + 1
    END
END {case}
```

Of course the constants *gotit* and *givenup* do not suffer from the drawbacks of normal statement labels, as described by Hill. These are constant values of an enumeration type and access to a limb of the case-statement can only be via the selector *state*.

As a general technique for programming a multi-exit loop, this approach has a number of advantages. The intent of the loop is readily apparent and subsequent processing, upon exit from the loop, is more transparent. The case-statement could, of course, be removed and its actions inserted at the appropriate places within the loop body but the separation is to be preferred; each action defined by the case-statement is performed only once and so does not logically form part of a loop.

This format also has the advantage that further states of interest can easily be accommodated. Any number of tests can be included within the loop and can be arranged in any order; this avoids the need to organize while-loops (or repeat-loops) so that the test is made at the beginning (or at the end).

For instance, in the above example, we might decide to qualify the state

'I've looked everywhere but it's not here'

as either

'It's not here but there's room for it'

or

'It's not here and there's no room for it'.

We simply replace the state *givenup* by two others, *extend* and *noroom*, and include a further test ($m < max$) within the loop. This test will not be made more than once and so will not affect efficiency:

Atkinson2

```
IF m = 0 THEN state := extend ELSE
BEGIN
  state := searching; i := 1;
  REPEAT
    IF a[i] = x THEN state := gotit ELSE
      IF i < m THEN i := i + 1 ELSE
        IF m < max THEN state := extend ELSE
          state := noroom
  UNTIL state < > searching
END;
CASE state OF
  gotit: b[i] := b[i] + 1;
  extend:
    BEGIN
      m := m + 1;  a[m] := x;  b[m] := b[m] + 1
    END;
  noroom: error
END {case}
```

6. MINIMAL SUBRANGING

One further advantage of the state indicator approach is of particular significance to Pascal. With the exception of Knuth's initial version, all the array search programs not using a sentinel require the range of i to be 1 greater than the index range of the array a. It is a general case, with state transition loops involving arrays, that the range of the subscripting variable need be no greater than the index range of the array. This *minimal subranging* has been commented upon by the present author[16] and is important for three reasons.

First, because Pascal permits any ordinal type to be used as an array index type, it may be impossible to extend the range of the index type; for example, this would be the case if the index type were *char*. Secondly, one of Pascal's greatest assets is the degree of compile-time security it affords, and this is particularly beneficial for subscript checking. When the range of a subscripting variable does not exceed the index range of the array, many possible subscript range errors can be detected by the compiler and, when the variable is used as a subscript, no run-time subscript checking is necessary. This leads to a third benefit—efficiency. Welsh[17] considers a number of situations and shows how minimal subranging aids compiler optimization and improves efficiency.

Efficiency of the state transition linear search compares reasonably with the alternative implementations of the same algorithm. **Knuth1** and **Knuth2** both make two tests each time round the loop and the state transition loop performs the same two tests and, in addition, tests *state* < > *searching*. However, this additional test is simply a jump on zero, usually achievable with a single machine code order, and so the run-time overhead is minimal. This extra test is equivalent to the inclusion of the Boolean variable in the loop test of Hill's program.

7. NUMERICAL ITERATION

As a final illustration of the general application of state indicators to multi-exit loops, we consider an example from the field of numerical methods. Many numerical processes involve repeating some process until convergence is achieved and the process we shall examine is the Newton iteration to find a zero of a non-linear function $f(x)$.

Given an initial value x_0, the basic iteration has the form

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

and we shall terminate the iteration when the relative difference between two successive iterates falls within some specified tolerance. This gives the following initial schema

```
REPEAT
  oldx := x;  x := x - f(x)/fdashed(x)
UNTIL abs((x - oldx)/oldx) < = tolerance
```

In case the process does not converge satisfactorily, it is advisable to bound the permitted number of iterations and so the loop acquires a second possible reason for exit. As with the previous examples, the loop can be controlled by a three-state scalar. This is illustrated by the program fragment of Fig. 2.

In addition, there are two places at which division by zero can occur: when a new iterate is computed ($f'(x)$ small) and when the relative error test is made (x small). Including tests for these eventualities gives the loop four reasons for exit and leads to the adoption of a five-state scalar variable. This is illustrated by the program fragment of Fig. 3.

There are even two further possibilities of division by zero—within the evaluations of $f(x)$ and $f'(x)$. A fragment offering complete protection in this respect is given in Fig. 4.

Many other numerical iterations, controlled by state indicator loops, are discussed by Atkinson and Harley.[18]

8. CONCLUSION

Little of the information presented in this paper is entirely new but no apology is made for this. Recent correspondence to *The Computer Journal* has shown that disagreement still exists over the most transparent style to adopt for a multi-exit loop and the amount of debate generated by one simple example shows that the issue is not trivial.

Choosing Pascal as our implementation language, we have attempted to show a particular approach which, in

```
CONST
  tolerance = . . .;  maxits = . . .;
VAR
  x, oldx: real;
  itcount: 0..maxits;
  state: (iterating, converged, maxitsreached);
  . . .
itcount := 0;  state := iterating;
REPEAT
  itcount := itcount + 1;
  oldx := x;  x := x - f(x)/fdashed(x);
  IF abs ((x - oldx)/oldx) ≤ tolerance THEN state := converged ELSE
    IF itcount = maxits THEN state := maxitsreached
UNTIL state < > iterating;

CASE state OF
  converged: . . .;
  maxitsreached: . . .
END {case}
```

Figure 2

```
CONST
  tolerance = . . .;  maxits = . . .;  assumedzero = . . .;
VAR
  x, oldx, fd: real;
  itcount: 0. . maxits;
  state: (iterating, converged,
          maxitsreached, flatspotmet, xtoonearzero);
  . . .
itcount := 0;  state := iterating;
REPEAT
  IF abs(oldx) < = assumedzero THEN state := xtoonearzero ELSE
  BEGIN
    fd := fdashed(x);
    IF abs(fd) < = assumedzero THEN state := flatspotmet ELSE
    BEGIN
      itcount := itcount + 1;
      oldx := x;  x := x - f(x)/fd;
      IF abs((x - oldx)/oldx) < = tolerance THEN state := converged ELSE
        IF itcount = maxits THEN state := maxitsreached
    END
  END
UNTIL state < > iterating;
CASE state OF
  converged: . . .;
  maxitsreached: . . .;
  xtoonearzero: . . .;
  flatspotmet: . . .
END {case}
```

Figure 3

```
CONST
  tolerance = . . .;  maxits = . . .;  assumedzero = . . .;
VAR
  x, oldx, fx, fd: real;
  itcount: 0 . . maxits;
  zerodivattempted: boolean;
  state: (iterating, converged,
          zerodivinf, zerodivinfdashed,
          maxitsreached, flatspotmet, xtoonearzero);
PROCEDURE Evaluatef(x: real;  VAR f: real;  VAR zerodiv: boolean);
  . . .
PROCEDURE Evaluatefdashed (x: real;  VAR f: real;  VAR zerodiv: boolean);
  . . .
  . . .
itcount := 0; state := iterating;
REPEAT
  IF abs(oldx) < = assumedzero THEN state := xtoonearzero ELSE
  BEGIN
    Evaluatef(x, fx, zerodivattempted);
    IF zerodivattempted THEN state := zerodivinf ELSE
    BEGIN
      Evaluatefdashed(x, fd, zerodivattempted);
      IF zerodivattempted THEN state := zerodivinfdashed ELSE
        IF abs(fd) < = assumedzero THEN state := flatspotmet ELSE
        BEGIN
          itcount := itcount + 1;
          oldx := x;  x := x - fx/fd;
          IF abs((x - oldx)/oldx) < = tolerance THEN state := converged ELSE
            IF itcount = maxits THEN state := maxitsreached
        END
    END
  END
UNTIL state < > iterating;

CASE state OF
  converged: . . .;
  maxitsreached: . . .;
  xtoonearzero: . . .;
  zerodivinf: . . .;
  zerodivinfdashed: . . .;
  flatspotmet: . . .
END {case}
```

Figure 4

simulating Zahn's event indicators, abides by the doctrines of structured programming, conforms to Pascal's structured control constructs, maintains program transparency and yet produces efficiency comparable with equivalent versions using explicit jumps. The use of Pascal in this way is gaining support. In teaching texts[18–20] the present author has proposed the use of state indicators as the recommended approach for multi-exit loops, particularly when the validity of one test is dependent upon the outcome of another. Recently, this recommendation has received support from Wilson and Addyman.[21]

It is hoped that this paper will urge more people to adopt this technique and thereby produce programs which are structured, transparent and efficient.

Acknowledgement

I am indebted to the referee for pointing out an omission in the original draft of this paper.

REFERENCES

1. I. D. Hill, Jumping to some purpose (letter). *The Computer Journal* **23**, 94 (1980).
2. J. Inglis, Jumping to some purpose (letter). *The Computer Journal* **25**, 495 (1982).
3. M. Missala and P. Rudnicki, Jumping to some purpose (letter). *The Computer Journal* **25**, 286 (1982).
4. G. L. Robinson, Jumping to some purpose (letter). *The Computer Journal* **23**, 288 (1980).
5. G. L. Robinson, Jumping to some purpose (letter). *The Computer Journal* **26**, 95 (1983).
6. A. T. Arblaster, M. E. Sime and T. R. G. Green, Jumping to some purpose. *The Computer Journal* **22**, 105–109 (1979).
7. D. E. Knuth, Structured programming with goto statements. *Computing Surveys* **6**, 261–301 (1974).
8. *BS6192: 1982 Specification for Computer Programming Language Pascal*, BSI, London (1982).
9. D. W. Barron and J. M. Mullins, What to do after a while. *Pascal News* **11**, 48–50 (1978).
10. A. H. J. Sale, Compiling boolean expressions. *Pascal News* **11**, 76–78 (1978).
11. K. Jensen and N. Wirth, *Pascal—User Manual and Report*. Springer-Verlag, New York (1978).
12. L. V. Atkinson, Should **if** . . . **then** . . . **else** . . . follow the dodo? *Software—Practice and Experience* **9**, 693–700 (1979).
13. C. T. Zahn, A control statement for natural top-down structured programming. *Symposium on Programming Languages*, Paris (1974).
14. L. V. Atkinson, Know the state you're in. *Pascal News* **13**, 66–69 (1978).
15. L. V. Atkinson, Pascal scalars as state indicators. *Software—Practice and Experience* **9**, 427–431 (1979).
16. L. V. Atkinson, A contribution to minimal subranging. *Pascal News* **15**, 60–62 (1979).
17. J. Welsh, Economic range checks in Pascal. *Software—Practice and Experience* **8**, 85–97 (1978).
18. L. V. Atkinson and P. J. Harley, *An Introduction to Numerical Methods with Pascal*. Addison-Wesley, London (1983).
19. L. V. Atkinson, *Pascal Programming*. Wiley, Chichester (1980).
20. L. V. Atkinson, *A Student's Guide to Programming in Pascal*. Wiley, Chichester (1982).
21. I. R. Wilson and A. M. Addyman, *A Practical Introduction to Pascal—with BS6192 (2nd edition)*. Macmillan, London (1982).

Received January 1983