GOTO (CONSIDERED HARMFUL)^n, n IS ODD

P.H.Welch, Computing Laboratory, The University, Canterbury, KENT— CT2 7NF

SOME CORRESPONDENCE

You have either been following, or else have just missed, an entertaining series of letters in the columns of the Communications of the ACM. The subject was the GOTO statement and its contribution to simple clear programming. The debate was passionate. For an occamist, it was somewhat alarming to see all those GOTOs rising to the surface with such vigour. Just when we thought it was safe to commit ourselves to working with a language that didn't have the things, there they were again demanding attention and threatening to bite!

It started with a serious letter [0] in the March 1987 CACM. This argued that a literal acceptance of Dijkstra's 19 year old letter to the same journal, ''GOTO Statement Considered Harmful'' [1], leads to gratuitously contorted, inefficient and often incorrect algorithms. The author described his astonishment at the almost universal acceptance of Dijkstra's proposition — the points in its favour being merely ''academic'' — and claimed that this had cost the software industry ''hundreds of millions of dollars in excess development and maintenance costs, plus the hidden cost of programs never developed due to insufficient resources''!

Whilst the author agreed that unbridled use of the GOTO was a bad thing, he maintained that it was quite safe — indeed, most beneficial — to give yourself the occasional fix to get out of some logic into which you had gotten yourself jammed. Moderate use of the GOTO, therefore, was a good thing, gave much needed relief and would not turn you into a hardened addict.

In order to illustrate this thesis, the following problem was specified. ''Let X be an N x N matrix of integers. Write a program that will print the number of the first all-zero row of X, if any''.

That letter stimulated an enormous response (see the May, June, July and August CACMs). Clearly, there were a lot of people out there nursing large grudges against the accepted wisdom that forbade them the GOTO. Numerous ''solutions'' were offered, both with GOTOs and without. Many contained errors (chiefly array index violations, but also failing to work if the all-zero row was the last one, if N were zero or other silly end effects that are the cause of so many software problems). Virtually no attempts were made to reason — even informally — about the correctness of the algorithms.

Eventually, Professor Dijkstra responded [2] and pointed out the obvious: that the set problem is a double instance of a standard problem (the ''bounded linear search'') for which there is a standard algorithm (involving a while loop, indexing and a boolean flag) and a standard theorem to prove its correctness. To solve the problem, do a bounded linear search through the rows of the matrix seeking an all-zero row. To seek an all-zero row, do a bounded linear search through its elements looking for a non-zero. Correct code is produced by applying the theorem twice with no further proofs necessary.

The trouble is that people look at Dijkstra's code in isolation from its derivation and say: ''Crumbs, I can't follow that — I still prefer that GOTO solution as being snappier and more direct.'' This is despite the fact that we don't really need to look at Dijkstra's code and understand it, since it was generated automatically (i.e. we may consider it as ''object'' code, not really for human consumption). It seems that we have a psychological need to look at real code, feel comfortable about it and not rely on those funny ''academic'' theorems.

BUT IN OCCAM ...

In occam, of course, we have no choice — there is no GOTO to tempt us from the true path. But the true path now offers complete relief since occam allows a direct algorithm to implement the bounded linear search (that is far simpler than the standard one employed by Dijkstra and has a trivial correctness proof).

Let's formalise the specification (just a little bit). The ''bounded linear search'' is simply to find the smallest i from the set {0, ..., n-1} such that some predicate, P(i), holds — if any. The simple solution is to test the predicates P(0), ..., P(n-1) sequentially until one holds or they all fail. This, of course, is directly expressed as a single IF statement :-

```
IF
  P(0)
    ... answer is 0
  P(1)
    ... answer is 1
  .
  .
  P(n-1)
    ... answer is n-1
  TRUE
    ... search fails
```

The correctness of this implementation follows directly from the semantics of the IF construct (which evaluates its conditions sequentially until one turns out to be TRUE).

For variable n, occam has a syntactic abbreviation for this lengthy (but trivial) statement :-

```
IF
  IF i = 0 FOR n
    P(i)
      ... answer is i
  TRUE
    ... search fails
```

(Note that if n is zero, the inner IF clause replicates zero times and the search fails — no ‘‘silly bugs’’ here!)

To the occamist, this is elementary stuff. There is nothing clever here — that is the important point. There is a sense of wonder that anybody would wish to employ some WHILE loop with tricky flag variables — let alone anything so insecure as a GOTO!

To solve the set problem, there are two nested bounded linear searches. The inner search checks on a particular row :-

```
BOOL FUNCTION all.zero (VAL []INT row)
  BOOL answer:
  VALOF
    IF
      IF j = 0 FOR SIZE row
        row[j] <> 0
          answer := FALSE
      TRUE
        answer := TRUE
    RESULT answer
:
```

The main search then becomes :-

```
BOOL, INT FUNCTION first.all.zero.row (VAL [][]INT X)
  BOOL found:
  INT index:
  VALOF
    IF
      IF i = 0 FOR SIZE X
        all.zero (X[i])
          found, index := TRUE, i
      TRUE
        found, index := FALSE, -1
    RESULT found, index
:
```

This completes the solution. I would claim that it is clear, correct and trivial — indeed, more so than any incorporating a GOTO. But that, of course, is a subjective claim.

SECURITY AND EFFICIENCY

Three points about occam to forestall some criticism of the above code. Firstly, VAL parameters may be passed by value or reference or in-line name substitution (no assignments to or aliasing of them are allowed by the language — these are enforced by the compiler). The transputer implements vector parameters by reference — i.e. we get both security and efficiency (occam FUNCTION parameters must be VAL data types).

Secondly, although the final RESULT clause of the VALOF expression forces explicit declaration of some transient variables (e.g. answer, found and index), its compulsion enables the compiler to check that FUNCTION results do get defined — i.e. we get security at the cost of some mild inconvenience. Not allowing the arbitrary placement of a RESULT clause also prevents its use as a sneaky break out of loops within FUNCTION bodies (as if you ever thought of trying such a thing!).

Thirdly, an objection was made in some of the CACM correspondence to the abstraction of the all.zero logic into a separate FUNCTION. I would have hoped that such thinking would have been abandoned long ago as counter-productive. However, if people really worry about this, occam FUNCTION calls are defined semantically in terms of their in-line substitution. The transformation which removes this abstraction is trivial and precisely defined in the language definition [3] (and may be done by compiler optimisers).

In summary, we have a simple, clear, complete and efficient solution in a simple, clear, complete and efficient language.

In the August CACM, the author of the original letter renewed his challenge: ‘‘It is easy to find problems where the best known solution with GOTOs permitted is simpler and/or faster than the best known solution with GOTOs forbidden. The opposite is impossible’’! With occam, we can refute this and we can do so using his own problem.

SUMMARY

Occam is a rare programming language in that it obeys (and was designed to obey) a rich set of simple mathematical laws [4, 5]. This allows clear thinking.

In all the excitement about occam’s power for expressing parallel logic, we should not underestimate its contribution for sequential reasoning. For instance, the fact that expression evaluation is mathematically pure (i.e. no side-effects can take place), whilst offending the hardened C hacker, does lead to code that can be reasoned about. [It also would permit a parallel implementation of expression evaluation ... but that is quite another story!]

This article has pointed out how a standard occam sequential construct (the replicated IF) gives a trivial solution to a standard old problem (the bounded linear search) which is frequently raised to support the contention that GOTOs are needed for simplicity and efficiency. [Actually, if anyone knows of another use for the replicated IF, please tell me!!]

CONTENTIOUS FOOTNOTE

Many of my colleagues are getting excited about the impending arrival of robust C and FORTRAN-77 compilers for the transputer. ‘‘At last,’’, they say, ‘‘transputers we can use!’’.

This puzzles me. I see such developments only for providing necessary short-term solutions to enable existing software take advantage of (single) transputer performance. For instance, individual alien-language jobs could profitably be submitted to some batch-processing transputer ‘‘farm’’. However, I would not want to use C as the implementation language for the farm software.

For anything new, I want to use occam to program transputer networks because :-

- the occam model of concurrency is simple, powerful, mathematically consistent and built into the language as a central feature of its original design;
- the occam model of sequential logic is simple, mathematically consistent, certainly safer and, in some ways, more powerful than those of traditional sequential languages.

I have little trust in, and no enthusiasm for, traditional sequential languages to which parallel constructs have been grafted as an afterthought. The occam approach enables a comprehensive view to be developed of a concurrent system as a single integrated software/hardware structure.

So, the next time anyone offers you a quick GOTO and an alien language — just say: ‘‘... go replicate an IF!’’.

REFERENCES

[0] Rubin, F., ‘‘ ‘GOTO Considered Harmful’ Considered Harmful’’, Commun. ACM 30, 3 (Mar. 1987), 195-196.

[1] Dijkstra, E.W., ‘‘GOTO Statement Considered Harmful’’, Commun. ACM 11, 3 (Mar. 1968), 147-148.

[2] Dijkstra, E.W., ‘‘GOTO One More Time’’, Commun. ACM 30, 8 (Aug. 1987), 661-662.

[3] May, D., ‘‘Occam 2 Language Definition’’, INMOS Limited, 1000 Aztec West, Almondsbury, BRISTOL, BS12 4SQ, ENGLAND.

[4] Hoare, C.A.R. et al., ‘‘Laws of Programming’’, Commun. ACM 30, 8 (Aug. 1987), 672-686.

[5] Roscoe, A.W. and Hoare, C.A.R. et al., ‘‘Laws of Occam Programming’’, Technical Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, OXFORD, OX1 3QD, ENGLAND.