A SPACE-SAVING TECHNIQUE FOR ASSIGNING ALGOL 68 MULTIPLE VALUES*

Lambert MEERTENS
Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam 1005, The Netherlands

Received 9 June 1976

ALGOL 68, optimization

1. Introduction

The straightforward way of elaborating an assignation $d := s$ in an ALGOL 68 implementation is [1]:
- elaborate the destination d and the source s, yielding a name N and a value V;
- assign the value V to the name N.

In many cases, the elaboration of s will consist of the creation of a copy on the working stack of an already existing value V. This value is assigned to N by copying it once more into the memory location(s) whose address is given by N. In most of these cases, simple compile-time optimization techniques permit a translation which does not make the extra copy on the working stack and instead copies the source directly into the memory location(s) of the destination [2].

Such an optimization is of special interest if a multiple value (array) is being assigned, since an extra copy of a large multiple value might take up more space than is available. A multiple value might be copied by setting up a loop copying the individual elements one by one. However, a difficulty is encountered here: the destination and the source may overlap in memory.

An example: After the declaration [1:3, 1:3] REAL a, consider the assignation a[,3] := a[1,], which must assign the first row of a to the third column. Simple-minded application of the optimization would result in code amounting to

a[1,3] := a[1,1],
a[2,3] := a[1,2],
a[3,3] := a[1,3].

But this is wrong! The original value of a[1,3] is superseded before it is assigned to a[3,3]. Performing the assignments in reverse order would give the correct results in this case, but it is not difficult to construct examples where neither the "normal" nor the reverse order will do. (For example, after the declaration [1:3, 1:3, 1:3] REAL b, the assignation b[2, ,] := b[, ,2].)

A technique which can be applied at run time to determine a safe order for assigning the elements is presented below. The use of this technique may entail some overhead in execution time. It assumes that the destination is not "flexible", so that the old value it refers to occupies the same amount of space as the new value it is to receive**.

2. Preliminaries

A multiple value of n dimensions has a descriptor of the form $((l_1, u_1), (l_2, u_2), ..., (l_n, u_n))$, where l_i and u_i are the ith lower and upper bounds. If $u_i < l_i$ for any i, then the descriptor is "flat". This case must be treated as a special case, because of the so-called ghost element, but there is no need to make any actual copy. Otherwise, the multiple value has $(u_1 - l_1 + 1) \times (u_2 - l_2 + 1) \times ... \times (u_n - l_n + 1)$ elements, each of which is selected by a specific "index" $(r_1, ..., r_n)$, where $l_i \leq r_i \leq u_i$.

* This paper is registered at the Mathematical Centre as IW 56/76.

** If the destination is flexible and the old value occupies less space than the new one, it can be shown that no overlap can occur. It is not clear how this fact can be used. If the old value takes at least as much space, the technique described here could be used. In that case there should be a means to release the extra space.

It is assumed here that selection makes use of a linear address calculation method, so that the address a corresponding to an index $(r_1, ..., r_n)$ is given by some formula of the form

$$a = c + r_1 \times d_1 + r_2 \times d_2 + ... + r_n \times d_n.$$

Moreover, it will be assumed that the last subscript runs fastest. More precisely, if we take the lexicographic ordering on the indices, defined by

$$(r_1, r_2, ..., r_n) < (r_1', r_2', ..., r_n') \text{ iff } r_1 < r_1' \text{ or}$$
$$r_1 = r_1' \text{ and } (r_2, ..., r_n) < (r_2', ..., r_n'),$$

then the mapping $(r_1, ..., r_n) \to c + r_1 \times d_1 + ... + r_n \times d_n$ is strictly monotonic increasing.

This property holds, for example, if for each newly created multiple value, $d_1, ..., d_n$ are chosen to satisfy $d_n \geqslant 1$ and $d_{i-1} = (u_i - l_i + 1) \times d_i$ for $i = n, n-1, ..., 2$. It is left invariant by all operations on multiple values provided by ALGOL 68 (slicing, selecting and rowing). If matrix transportation were added, it would no longer hold! Adapting the technique to the case where the first subscript runs fastest should present no problems.

3. The technique

For the assignment to be defined, the multiple values corresponding to the destination and the source must have the same descriptor $((l_1, u_1), ..., (l_n, u_n))$. Let the respective address calculation vectors be $(c, d_1, ..., d_n)$ and $(c', d_1', ..., d_1', ..., d_n')$. The following algorithm assigns the elements one by one in a safe order:

```
FOR $r_1$ FROM $l_1$ TO $u_1$
DO
  .
    .
    FOR $r_n$ FROM $l_n$ TO $u_n$
    DO INT $\lambda = c + r_1 \times d_1 + ... + r_n \times d_n$,
       INT $\rho = c' + r_1 \times d_1' + ... + r_n \times d_n'$;
       IF $\lambda < \rho$ THEN assign $(\lambda, \rho)$ FI
    OD
    .
  .
OD;
FOR $r_1$ FROM $u_1$ BY $-1$ TO $l_1$
DO
  .
    .
    FOR $r_n$ FROM $u_n$ BY $-1$ TO $l_n$
    DO INT $\lambda = c + r_1 \times d_1 + ... + r_n \times d_n$,
       INT $\rho = c' + r_1 \times d_1' + ... + r_n \times d_n'$;
       IF $\lambda > \rho$ THEN assign $(\lambda, \rho)$ FI
    OD
    .
  .
OD.
```

It can be seen that the algorithm consists of two nested loops. The first one runs through the indices in ascending order and performs the assignments of those individual elements for which the direction of transport in memory is from high to low; whereas the second one runs through the indices in descending order and performs the assignments in the opposite direction.

Remark: The computations involved in the address calculations can be optimized in an abvious way; they are here presented as they are only for the sake of clarity.

4. Correctness proof

Let $\lambda(I)$ and $\rho(I)$ denote the addresses corresponding to an index $I = (r_1, ..., r_n)$. The algorithm defines a sequence of statements

assign $(\lambda(I_1), \rho(I_1))$;
assign $(\lambda(I_2), \rho(I_2))$;
⋮
assign $(\lambda(I_z), \rho(I_z))$.

Each index gets its turn, either in the first or in the second nested loop, depending on whether $\lambda(I_k) < \rho(I_k)$ or $\lambda(I_k) > \rho(I_k)$. Those indices for which $\lambda(I_k) = \rho(I_k)$ are left out; in that case the copying is a dummy action.

Let the indices which get their turn in the first nested loop be $I_1, ..., I_m$ (so that those getting their turn in the second one are $I_{m+1}, ..., I_z$). We have

$I_1 < I_2 < \ldots < I_m$,

$I_{m+1} > I_{m+2} > \ldots > I_z$,

$\lambda(I_k) < \rho(I_k)$ for $k \leqslant m$,

$\lambda(I_k) > \rho(I_k)$ for $k > m$,

if $\lambda(I) < \lambda(I')$, then $I < I'$,

if $\rho(I) < \rho(I')$, then $I < I'$.

It must be shown that an address which occurs both as that of a source element and as that of a destination element is first used as source and thereafter only as destination. More formally, we must show:

if $\rho(I_s) = \lambda(I_t)$ for some s and t, then $s < t$.

We distinguish four cases:

Case A: $s \leqslant m$ and $t \leqslant m$. We have $\rho(I_s) = \lambda(I_t) < \rho(I_t)$, so $I_s < I_t$, and therefore $s < t$.

Case B: $s \leqslant m$ and $t > m$. It immediately follows that $s < t$.

Case C: $s > m$ and $t \leqslant m$. From $\rho(I_s) = \lambda(I_t) < \rho(I_t)$, we deduce $I_s < I_t$. From $\lambda(I_s) > \rho(I_s) = \lambda(I_t)$, we deduce $I_s > I_t$. Clearly, this is impossible, so this case cannot arise.

Case D: $s > m$ and $t > m$. We have $\rho(I_s) = \lambda(I_t) > \rho(I_t)$, so $I_s > I_t$, and therefore $s < t$.

References

[1] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens and R.G. Fisker (eds), Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica 5 (1975) 1–236.

[2] P. Branquart, J.-P. Cardinael, J. Lewi, J.-P. Delescaille and M. Vanbegin, An optimized translation process and its application to ALGOL 68, Lecture Notes in Computer Science 38 (Springer Verlag, Berlin etc., 1976).