Storage Management for ALGOL68

A. D. Birrell

Abstract This paper describes some of the techniques which can be used for managing the run time storage required for an ALGOL68 program. The emphasis is on stack storage, since garbage collection techniques would require another paper. The problems caused by some ALGOL68 constructs are described; the solutions given are mainly those adopted in the Cambridge ALGOL68C system.

1 Representation of objects.

ALGOL68 is a language concerned with internal objects and operations upon them. In designing the storage management for ALGOL68 one of the first questions to be faced is how to represent these objects. In other words, when a value of some mode (data type) is assigned, or yielded, what bit patterns are physically moved around the store of the machine.

For some modes, the choice of representation is straightforward:

int => appropriate (machine-dependent) bit pattern - typically a single word.

char => single byte (if possible).

real => floating point number.

Other modes can be built out of simpler parts:

struct(...) => concatenation of the fields, possibly with gaps for boundary alignment.

union(...) => (marker, value)

routine => (entry address, environment pointer)

Note that although the ALGOL68 report does not talk of values of mode union(...), a value which has been united is universally represented as the value with a small marker indicating its mode. The 'environment pointer' in a routine is used in addressing items in blocks outside the routine - this is considered later.

It should be pointed out here that it is possible to achieve considerable simplification of many of our problems by an indirection. By representing values of the more complicated modes by a pointer to data in a global storage area, the whole stack organization is simplified; this substitutes the problems

of managing the global storage area for the problems described below. I believe such a technique has been adopted in the Carnegie-Mellon implementation of ALGOL68S.

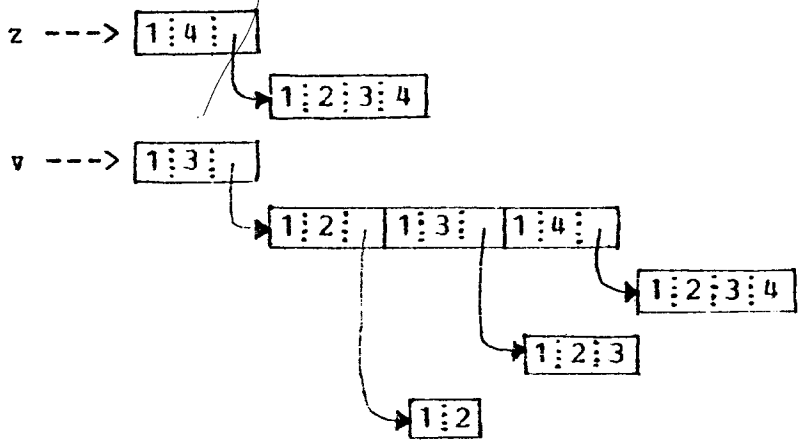
The representation of multiple values is a cause of many problems. There are several causes. Firstly, we must always know the bounds of an array (multiple value), so that we can generate code to copy the array, and for array bound checking. Often, situations arise such that we cannot know these bounds at compile time and so must store them as part of the run time representation. Secondly, the facility of trimming a multiple value allows the program to manipulate sub-arrays. Unless we copy the elements of an array when we trim (which seems needlessly expensive), we must store separately from the elements a block containing the bounds of, and a pointer to, the elements. Thirdly, when subscripting an array we require a uniform distance between the addresses of the elements. When the elements are themselves arrays, we must carefully consider how to achieve this.

For example:

```
[ ][ ]int v= ( [ ]int x=(1,2)
                      y=(1,2,3),
                      z=(1,2,3,4);
               (x,y,z)
             );
```

We require that (address for v[1]) - (address for v[2])
= (address for v[2]) - (address for v[3])

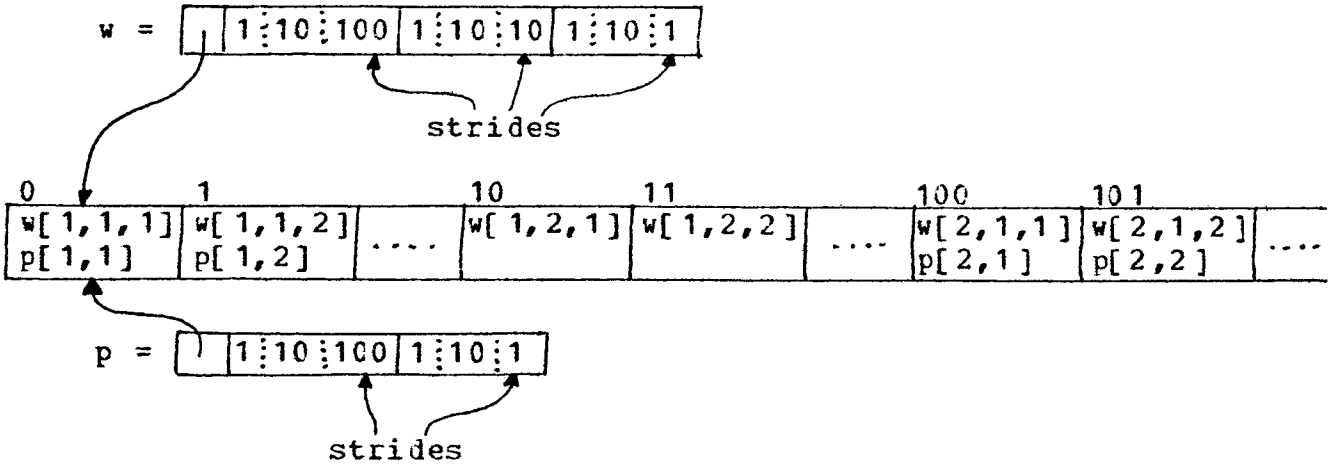
All these requirements can be satisfied by using a descriptor containing bounds and a pointer to the elements. Thus:



Additionally, the descriptor contains a stride for each dimension, to indicate the spacing between elements. For example, if we have

```
[,,]int w = (......);
[,]int p = w[,1,];
```

then the representations of 'w' and 'p' are:



A final step in the representation of such objects is usually that the pointer, instead of being the address of the first physical element, is the address which would be that of the element whose subscript is 0 in each dimension, even if this is not physically part of the array (or even a legal address). This simplifies subscripting, since the computation no longer involves the bounds (unless for checking purposes).

Thus in general objects involving array elements can be complicated tree structures. When such an object is assigned, or storage for it is generated, we must generate code to manage such tree structures. Innocuous looking declarations or assignments can generate considerable tracts of code, and the presence of such objects is the major reason for complexity in our stack management. In future we will term the first level of such an object the static part of the object (its size is known at compile time); the remainder of the object is the dynamic part (its size may not be known until run time).

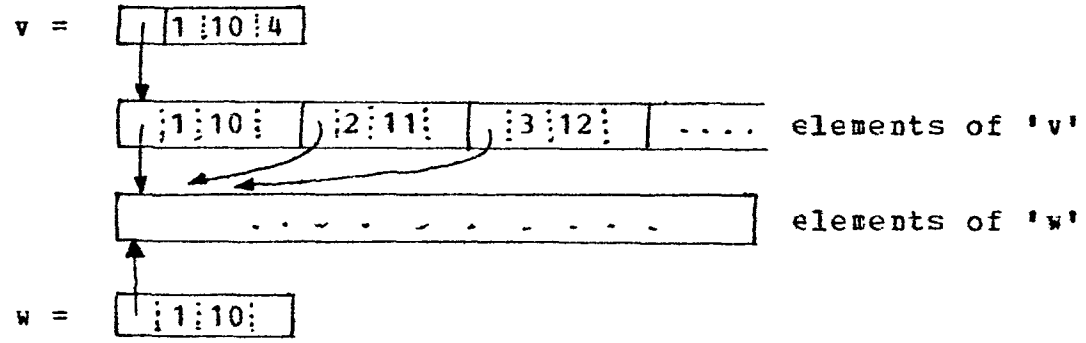
The representation of names, that is objects whose mode is of the form ref amode, is mainly straightforward - the address of an amode object. However, a complication arises if amode is of the form [...]bmode. This complication is caused by trimming; the scope of the name yielded by trimming is the same as that of the name being trimmed. Thus the descriptor produced can

be required to exist longer than the block in which the trimming occurs, and so cannot be allocated on the present stack frame. For example

```
[1:10] ref[ ] int v;
[1:20] int w;
for i to 10 do int j; ... ; v[i]:=w[i:i+10 at i] od;
```

Here, the descriptors for the arrays referred to by the elements of 'w' are created when trimming 'w' inside the loop, but their storage cannot be allocated at this point. For this reason, when allocating store for an object whose mode is of the

form ref[....]bmode, we also allocate store for a descriptor. For example the declaration of 'v' allocates store for 10 extra descriptors. An immediate optimisation is to represent such names as the descriptor, rather than its address. For example,



Note that this causes the implementor some tedium, since ref[...]bmode must always be handled as a special case. It also causes minor complications in handling identity relations for objects of such modes.

2 The stack

The stack organisation we will develop is based on the conventional ALGOL60 stack, which is summarised here.

The stack consists of a number of frames, one for each block which has been entered but has not yet terminated.

At any time there is a current display. A display consists of a number of display registers (or levels), each containing the address of some frame. The current display consists of registers addressing the current and each textually enclosing block.

To access a word of the stack, we use an address of the form

[display register] + offset

where the display register (but not its contents) and the offset are known at compile time.

Due to recursion, there may be more than one frame for each block.

On entry to a block, a new frame is started and an extra display register added to the display in order to address the frame.

On entry to a procedure, the complete display is reset to correspond to the block in which the procedure (not the call) occurred. To allow this operation, the representation of a

routine value contains an environment pointer indicating the values to load as the new display. This routine value is known as a closure; due to recursion, there may be different closures for a single routine text.

On exit from a block or procedure, the display is restored to its previous value.

In ALGOL68, there are several categories of data we wish to store on the stack.

a) A heading containing the address of the previous frame, subroutine link, information for setting up the display, etc.

b) values for definitions: for 'int i' the value referred to, for 'real x = random' the value itself.

c) anonymous results created during elaboration of the block.

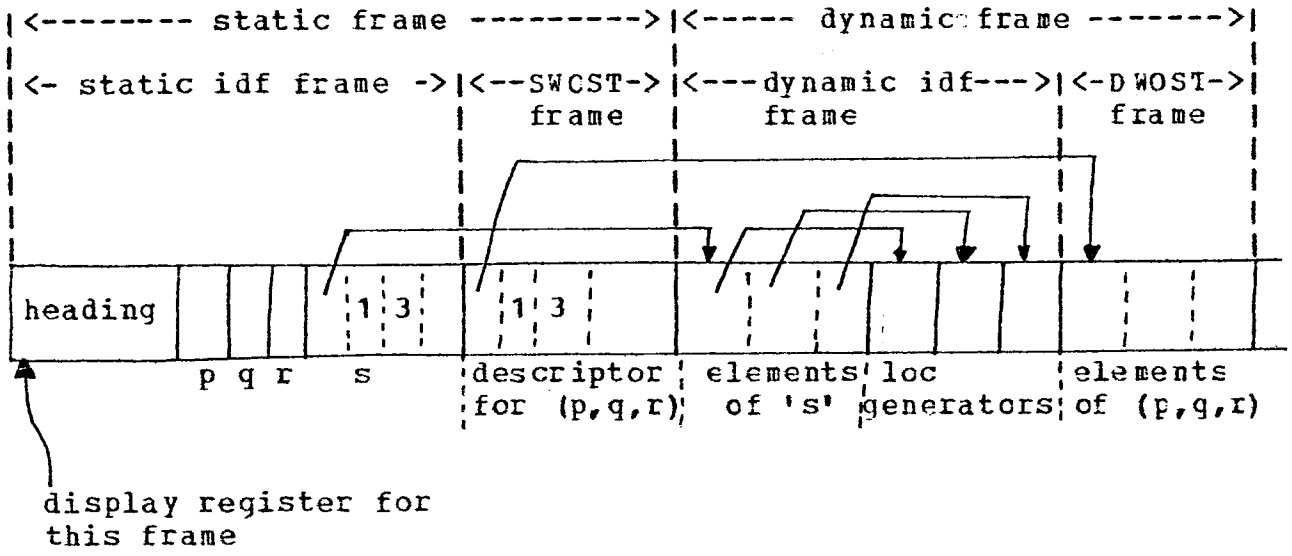
d) array elements

e) storage for explicit loc generators.

Of these (a), (b) and (c) are straightforward, but (d) and (e) are difficult since the amount of storage required may be large and may not be known at compile time.

It is common practice to store (a), (b), (c) at the start of the frame, with (d) and (e) at the end. This has two advantages: the offsets written in instructions are smaller (many machines place severe limitations on such offsets), and we always know the offsets for identifiers at compile time. We can thus consider each frame as being divided into a static frame containing (a), (b), (c), and a dynamic frame (possibly empty) containing (d) and (e). Further, it is convenient to treat (c) separately as the SWOST (static working stack) frame, calling the remainder of the static frame the static idf frame. Similarly, we can sub-divide the dynamic frame into the DWOST frame containing the dynamic parts of SWOST objects, and the dynamic idf frame. For example:

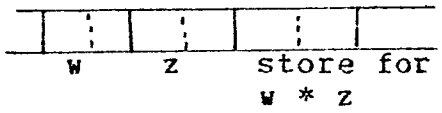
```
begin
   int p, q, r, [1:3]ref int s;
   for i to 3 do s[i] := loc int od;
   .
   .
   (p,q,r) #row display#
   .
   .
end
```



Management of the static frames is mainly straightforward. Storage is allocated on the static idf frame only at definitions. The structure of the language is such that at a definition, none of the anonymous results which have been created in the current block still exist, so the SWOST part of the frame is empty. Thus the static idf frame is contiguous storage starting at the end of the heading area. The SWOST, then, is always placed at the end of the static idf frame. However, situations can arise which produce holes in the SWOST; these are typically when we are constrained to produce the result of some action without overwriting its parameters. An example might be:

```
compl w := ...,  z := ..  ;

...  w * z ....;
```



It is always possible to avoid such holes by copying, and with sufficient care most holes can be avoided without copying. In ALGOL68C, we decided that the extra expense of allowing the holes was not great enough to justify the complexity (or expense, if we copy) of avoiding them. Accordingly, holes are allowed to occur on SWOST; however since SWCST for a block is always empty at a semicolon and before a definition, such holes are generally of short duration. It should be noted that all static frame offsets are known at compile time. No run time management is required.

Before considering the management of dynamic frames, we should look at an optimisation available to us. Within a single procedure, we know at compile time all the offsets inside each

static frame. If, then, we place all static frames first, followed by the dynamic frames, we will be able to address all the static frames with a single display register, pointing to the base of the first static frame. This optimisation is often described as having one frame per procedure, but this is not really an accurate description. In terms of when store is allocated and recovered, and in the interleaving of static idf frames with SWOST, we are still running one frame per block. The only alteration is to omit some display registers, and move the dynamic frames. The dynamic frames are still, in every sense, one per block. This optimisation gives us several gains. The number of display levels is drastically reduced, being limited to the textual nesting depth of procedures - in practice, we have never encountered depths greater than 5, although ALGOL68C allows for 64. The number of display registers required is in fact less than the textual nesting depth, since if an enclosing frame is not referenced from inside a procedure, it can be omitted from the display. (This is allowed by, and required by, the rules on the scope of routine values.) It is possible, instead of keeping the complete display, to keep only a pointer to the static frames of the current procedure, and store there a pointer to the frames of the enclosing procedure. Then accessing a frame of an enclosing procedure is achieved by indirecting down this static chain. Since the number of levels on the static chain is typically less than 5, these indirections never go very far. In ALGOL68C we maintain a pointer to the outermost level, and one to the current procedure; in this way only about 2% of static frame accesses require indirection down the static chain (and then, less than 4 indirections). These indirections can be further reduced by remembering which registers currently address outer levels. It should be noted that, using the above techniques, if a block requires no dynamic frame then no run time cost is incurred by block entry or exit. This means that the programmer can freely use begin/end for structuring his program without worrying about extra code being generated. The environment pointer of a routine is now a pointer to the frames of the enclosing routine, and is used for the static chain when the body of the routine is elaborated.

The mechanism used by ALGOL68C for run time management of dynamic frames is unusual. At first sight it appears too complicated, but by paying a little in in conceptual complexity we have attempted to minimize run time actions, and as far as possible to eliminate them completely for blocks or procedures with no dynamic frame. We define a drange to be any range (block) which, excluding inner ranges, allocates storage on a dynamic frame, and a droutine to be any routine containing a drange. For each dynamic frame we will require a pointer to the top of that frame - this we call the dsmd (dynamic stack management data) for the frame. ALGOL68C always keeps the dsmd stored on the static frame at an address known as the dsma - as will be seen, this simplifies our run time actions. With the stack organization as described above, we would perform the following actions; these will be modified in the light of changes to be described later.

a) On entry to the outermost drange of a droutine, we allccate a dsma and initialise its dsmd tc the top of the static frames.

b) On entry to an inner drange, we allocate a new dsma and initialise its dsmd to the previous dsmd.

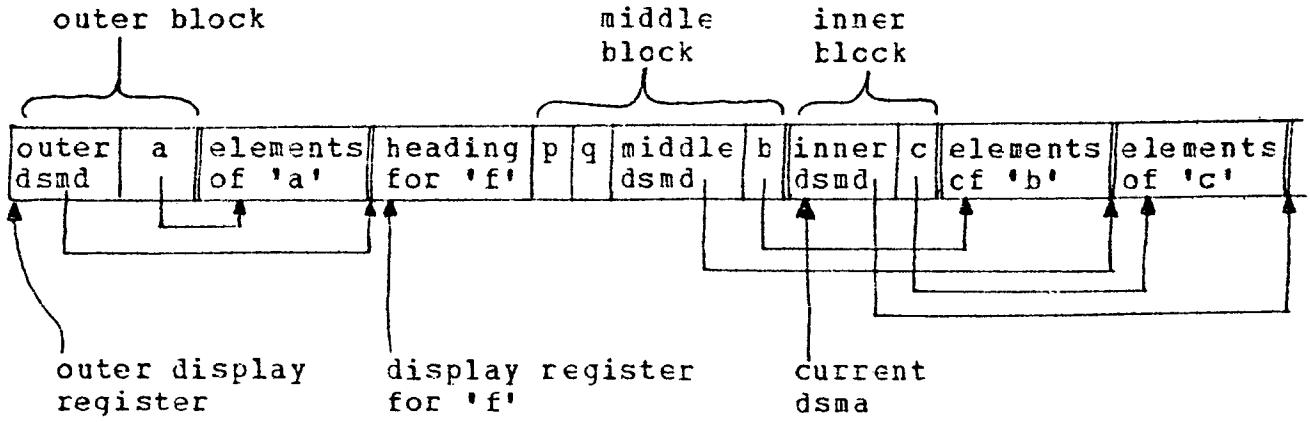
c) To allocate storage on a dynamic frame, we use and update the current dsmd (as addressed by the current dsma).

d) On exit from an inner drange, we revert tc the cuter dsma. Note that this is not a run time action, since we know the dsma (as a static frame offset) at compile time.

This mechanism is simpler at run time than the alternative of keeping a single dsmd and preserving/restoring it; it is the cnly tenable mechanism for the stack organization described belcw. Under this scheme, jumps present no problem - at the target label, we revert to the appropriate dsma. It is difficult to produce an alternative scheme which does not have tc preserve the dsmd at every call in case there is a jump cut cf a drange in some inner routine; such preservation has the effect that you pay for dynamic frames even if you do not use them. An example cf our stack organization would now be:

```
begin
   [1:10]int a;
   proc f = int:
      begin
         [1:10]int b;
         int p, q;
         begin
            [1:10]int c;
               .
               .
               .
         end
      end;
   f
end
```

The stack after declaring 'c' might be:



Considerable difficulty is presented by argument passing, when the arguments have dynamic parts allocated during their elaboration. For example:

(random < 0.5 | f | g)(a, loc[x:y]int, b)

Firstly, consider which display register to use for addressing the static parts of the arguments while inside the called routine.

a) Using the display register of the calling routine is not possible, since inside the called routine we would not know the offsets for the arguments.

b) We could use a separate display register solely for the arguments, but this would double the number of display levels. (This solution is quite commonly adopted by other implementors.)

c) The only other possibility is to address the static parts of the arguments using the display register of the called routine.

Assuming choice (c), then, we must consider where to place the dynamic frame allocated for the arguments.

a) We cannot place it before the static parts of the arguments, since we do not yet know its size.

b) We can place it after the static parts of the arguments only if we place it after the other static frames of the called routine, but in ALGOL68C we do not know the size of the called routine's static frames while we are elaborating the call. Some implementors do arrange to maintain this information at run time.

By this stage in the design of the stack we have accumulated (albeit implicitly) several problems.

1) Where to place the dynamic parts of arguments.

2) How, at the calling end, to address the static parts of the arguments since we do not at that stage have a display register for them.

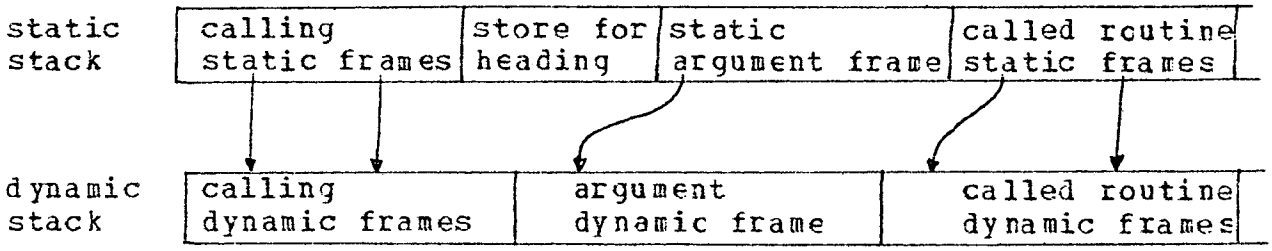
3) Storage is wasted since dynamic frames start at the high water mark of the static frames of the routine.

4) The ALGOL68C separate compilation mechanism would require a display register for each segment.

5) Any proposed solution of (1) to (4) with this form of stack organisation appears to be much too complicated.

In ALGOL68C, to solve these problems we made a drastic re-arrangement. Instead of continuing attempts to organize a single stack, we split the storage into two independent stacks. The static stack contains all static frames, and is addressed by display registers and offsets; the dynamic stack contains all dynamic frames, and is referred to from the static stack.

We can now have a simple solution to the argument passing problem. The static frame for the arguments is addressed at the calling end using the display register of the calling routine - since there are no intervening dynamic frames we know all the offsets at compile time. Inside the called routine, the arguments form the first static frame. The dynamic frame (if any) for the aguments is treated as any other dynamic frame, with no additional problems; if there is such a dynamic frame, the arguments will constitute a drange. Thus:



The wasted static frame storage is eliminated, and we do not need a separate display register for separately compiled segments.

With this revised organisation, we must revise the actions to be performed for managing the dynamic frames. Since we no longer know at compile time the base for the first dynamic frame of a droutine, this information must be passed with the call. Since we do not know at the call whether the called routine is, or will call, a droutine, the information must be passed with every call. To avoid this causing a run time action on every call, we always have the current dsma available at run time (in a particular register, say, or in a fixed store location). The actions to be performed are then:

a) On entry to any drange, allocate a new dsma, initialise its dsmd to the previous dsmd, and reset the run time dsma.

b) On exit from any drange, restore the run time dsma to its previous value.

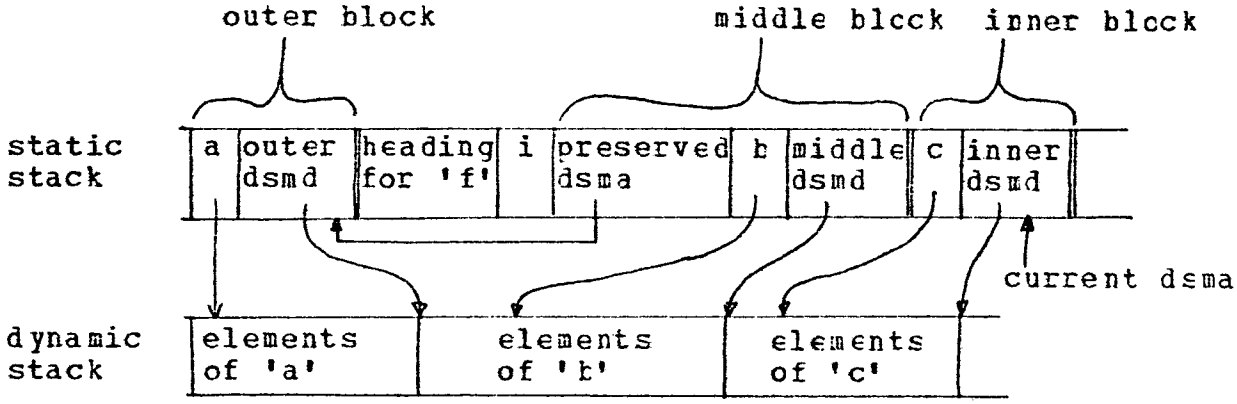
c) To permit (b) in the outer drange of a droutine, preserve the dsma on entry to a droutine.

To allow for labels and jumps, we include as a droutine any routine containing a label; at a label we reset the dsma to the appropriate value. Note that these arrangements still satisfy the dictum that if you don't use the dynamic stack then you

shouldn't pay for it (except at labels). Our example might now be as follows:

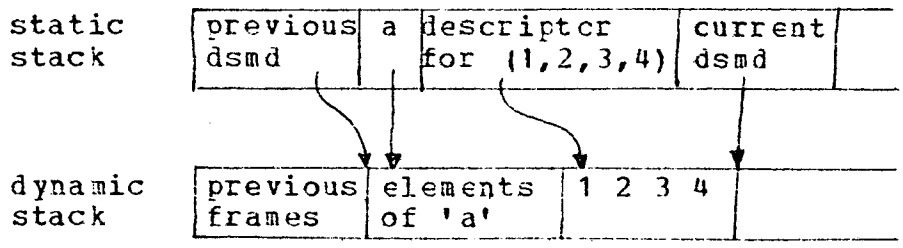
```
begin
   [1:10]int a;
   proc f = (int i)int:
      begin
         [1:10]int b;
         begin
            [1:10]int c;
             ( i <= 1 | 1 | i * f(i-1) )
         end
      end;
   print( f(1) )
end
```

The stacks after declaring 'c' would be:



One serious problem remains in our description of the stacks - this is the yielding of a result from a block or a procedure. The difficulty is that the result is constructed on stack frames, inside the block or procedure, which are about to be relinquished. For example:

```
begin
   [1:1000]int a;
   .
   .
   .
   (1,2,3,4)
end
```



There are basically two possibilities: either copy the value onto the outer SWOST and DWOST, or delay relinquishing the frames. However, copying can be very expensive (and sometimes very difficult), while delaying relinquishment can waste vast amounts of storage. An extensive analysis of this problem has been given by Branquart, and an algorithm which assists in copying has been given by Meertens. It is certainly best to delay the decision as to whether to copy or to avoid relinquishing, until as late as possible. At present ALGOL68C does not recover the storage in this situation - this is hardly satisfactory. With sufficient care, it is possible to achieve satisfactory results even in extreme examples such as:

```
op * = ([ ]int x,y)[ ]int: ... ,
   + = ([ ]int p,q)[ ]int: ... ;
[1:100]int a,b,c;

1: a * b + ( ... | c | goto 1 );
```

In particular, the compiler can treat some constructs as if they were dranges (though not actually ranges) to aid the recovery of dynamic stack.

3 Summary

The power and flexibility of the constructs available in ALGOL68 lead to considerable complexity in the objects being manipulated and in the management of the storage for them. By dividing the stack into two independent stacks we greatly simplify these problems, although on an unsegmented machine the need for three storage areas (the stacks and the heap) presents extra difficulties. An alternative solution, often adopted, is to place dynamic parts on the heap in times of difficulty - this we still must do when assigning objects of modes such as union([ ]int,[ ]real) - but this approach was discarded, because it is expensive and because it uses the heap behind the programmer's back. A full discussion of the problems of result passing would be outwith the scope of this paper, as are the techniques available for flex.

4 Acknowledgements

The ALGOL68C compiler was developed in Cambridge by a team led, until January 1975, by S.R.Bourne. Since then it has been maintained and further developed by C.J.Cheney for the University of Cambridge Computing Service. Throughout the development of the compiler much advice and much work has been given by M.J.T.Guy, I.Walker, and myself. Much of the work has been funded by the Science Research Council, and the maintenance is now supported by the Computer Board. Help has been given by our various users and by I.Wand of York University. Much of our terminology, and some of the ideas, are based on those of P.Branquart.

5 References

[1] A. van Wijngaarden, et al, "Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica, Vol. 5, pts 1,2,3, (1975).

[2] S.R.Bourne, A.D.Birrell, I.Walker. "ALGOL68C Reference Manual", Cambridge University Computer Laboratory, (1975).

[3] P.Branquart, et al, "An Optimized Translation Process and its Application to ALGOL 68", Report R204, M.B.L.E., Brussels, (1974).

[4] P.Knueven, "The Foundation of a Flexible Run-time System for ALGOL 68S", in "Experience with ALGOL 68", Proceedings of the Liverpool University Conference, April 1975, Ed. C.C.Charlton and P.H.Lang.

[5] L.G.L.T.Meertens, "A Space-saving Technique for Assigning ALGOL 68 Multiple Values", Mathematisch Centrum, Amsterdam, (1976).