IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 4/73 JUNE

IA

ANDREW S. TANENBAUM
DESIGN AND IMPLEMENTATION OF AN ALGOL 68
VIRTUAL MACHINE

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

ACM - Computing Reviews - category: 4.1, 4.2

ABSTRACT

A virtual machine specifically designed for running ALGOL 68 programs is proposed. The instructions and addressing of this machine are discussed in detail. A method of implementing the run time organization for this machine, based upon use of descriptors, is given. Memory organization, garbage collection, procedure and range entry and exit, and parallel processing are among the topics covered.

The machine has been designed such that a hardware implementation of it could have a single instruction for all assignations, and a single instruction for garbage collection.

Table of Contents

Introduction

Any attempt to produce a portable compiler for ALGOL 68, or any other language, must solve two fundamental problems: insuring that the compiler itself can be moved from machine to machine easily, and insuring that the object programs produced by the compiler can be moved from machine to machine easily. The compiler itself can be made portable by writing it either in a language that has already been implemented on many computers, such as ANSI standard FORTRAN, or by writing it in a very simple language, a compiler or macro processor for which can be implemented from scratch in a short time, due to its simplicity.

The problem of object code portability is more difficult. Consider, for example, a compiler written in ANSI standard FORTRAN which produces object code for the IBM 370. Although the compiler can be moved to a CDC Cyber 70 with almost no work at all, it will continue to produce object code for the 370, which is, of course, totally useless on the Cyber 70. This report discusses a solution to the object code portability problem based upon the concept of having the compiler produce object code for some virtual machine, which can be implemented on various existing machines, called the target machines.

There are at least four ways to implement a virtual machine on a "real" computer: by macro expansion, by procedure calls, by compilation, or by interpretation. A virtual machine (assembly) language program consists of a series of statements in prefix form, that is, the operator preceeds the operands, as is customary in assembly languages. For example, A :=B might be MOVE A,B, but not A MOVE B. Putting the operator (in this case MOVE) first simplfies the syntax of the language.

An implementation of a virtual machine using macros consists of defining a macro for each virtual machine instruction, with the macro body consisting of zero or more statements in some language already implemented on the real computer. Thus by string substitution, a program in the virtual machine language is converted into a program in some other language, which can be assembled or compiled on the existing machine. To run ALGOL 68 programs on a new computer requires writing macro definitions for the new computer.

An implementation of a virtual machine using procedure calls requires writing a procedure for each virtual instruction. These procedures are written in some language available on the target machine. The virtual machine program is regarded as a list of procedure calls, which together with the procedures form a valid program in some language available on the target machine. This method may require some trivial syntax changes to the virtual machine program, for example MOVE A,B may have to be changed to MOVE (A,B). Implementing the virtual machine on a new target machine requires writing a new set of procedures, unless the previous set is in a language available on the new machine.

A third possibility for implementing a virtual machine is to write a compiler for the target machine which compiles from virtual machine language. Such a compiler might be available with options to do more or less optimizing. A new compiler is needed for each new target machine.

The fourth way of implementing a virtual machine is to provide an interpreter or emulator for the target machine, which successively fetches, examines and carries out the virtual instructions, possibly after first having compiled them to a syntactically simpler intermediate form. An interpreter itself may be portable.

These methods have various advantages and disadvantages with respect to execution time, execution space required, ease of implementation, ease of debugging ALGOL 68 programs etc. They may also be mixed to combine the advantages of several methods.

In summary, the proposal is to implement the ALGOL 68 compiler as a two part process. The first part consists of writing a portable compiler which compiles ALGOL 68 programs to virtual machine language programs. The second part consists of making a (machine dependent) implementation of the virtual machine on each existing target machine. This method has several virtues.

1. Several implementations of the virtual machine may be available on the same target machine, for example, an interpreter written quickly while the other implementations were being developed, a highly optimizing multipass compiler for production jobs, an interpreter for student jobs designed to

detect as many run time errors as possible, such as unitialized variables, subscript range errors etc.

2. The problem of writing the compiler is simplified by breaking it up into two parts, each of which is more manageable than the original. The syntactic and semantic analysis of ALGOL 68 is in the compiler, and the machine dependent optimization is in the virtual machine implementation.

3. The separation of the machine independent and machine dependent parts will be very clean, and the compiler will be highly portable.

One of the design parameters is the "level" of the virtual machine. The virtual machine instructions could be designed to have the "flavor" of ALGOL 68, e.g. instructions like enter range, select and generate. Alternately they could be designed to have the flavor of a third generation computer, e.g. load register, add two integers, and branch on zero. Of course, intermediate designs are also possible. In general, the higher the level, the less trouble idiosyncrasies of particular target machines cause, the more work needed to implement the virtual machine, and the more efficient the system will be. The lower the level, the easier it will be to implement the virtual machine, but the less likely it will be that an efficient implementation will be produced. In essence, machine independence can be traded off against efficiency.

In this report, a high level virtual machine is proposed. The principal reason for this is to keep open the possibility of actually constructing the virtual machine from hardware, or at the very least providing an emulator for it on a microprogrammable computer. Such a hardware or microprogrammed virtual machine would be very efficient compared to the other implementation methods, particularly if the architecture of the virtual machine were designed for ALGOL 68.

It is exceedingly difficult to design an architecture independent virtual machine at the present state of the art. For example, consider the question of memory organization. A design intended for a machine with a small address space will run on a machine with a segmented virtual memory, but possibly very inefficiently, not only because program and data will straddle page and segment boundaries arbitrarily, but also because the software will be simulating features either provided for in, or made unnecessary by, the hardware. A design intended for a machine intended for a segmented address space will likewise cause difficulties on a machine lacking this feature. Compromises and tradeoffs made here tend to favor the future over the past.

This report is both preliminary and incomplete. Preliminary in the sense that some of the solutions are not optimal, and incomplete in the sense that some aspects of the virtual machine are not mentioned at all, notably transput. Furthermore, optimization is not treated in much detail.

The report contains two general parts: a description of the ALGOL 68 virtual machine, and a discussion of an ALGOL 68 run time design. The run time design could be used to implement the virtual machine, or it could be used in a traditional design where the compiler produces code for a specific target machine with no attempt at portability. Similarly, the virtual machine described may be implemented by a technique other than the one given here. Because the memory organization is critical to the virtual machine, the memory organization will be described first, including the concept of descriptors. Then the virtual machine will be described in detail. Following that, aspects of the run time design, i.e. the virtual machine implementation are covered.

The reader of this report is assumed to be somewhat familiar with ALGOL 68 Implementation, edited by J.E.L. Peck, North Holland Publishing Co., 1971.

Descriptors

One of the main goals of this design is to permit the construction or emulation of an "ALGOL 68 machine". Various studies have shown that 50 to 70 percent of the statements in higher level languages are assignment statements. Therefore it would be highly desirable to be able to build an "assignment instruction" into the hardware or microprogram. Such an instruction should be able to perform an assignment of values of any mode in a single instruction. Because a majority of statements are assignment statements, this will greatly improve the machines performance compared to a conventional machine. In particular, the need to fetch a large number of instructions from main memory will be eliminated. In a conventional computer, assigning a 1000 element array will probably require fetching at least 3000 instructions, 1000 each of load register, store register and conditional jump. If an assignment instruction were built into the hardware, these could all be eliminated. If the assignment instruction were microprogrammed, the micro-instructions could be fetched from a high speed read only control store.

Another important advantage of a built in assignment statement is the possibility of performing assignments of elements of an array or fields of a structure in parallel. With the recent appearance on the market of a general purpose digital computer with a 12 microsecond cycle time, stack hardware an instruction repertoire of over 40 instructions, and a cost of under $100, the possibility of having hundreds or even thousands of processors performing parallel assignments must be taken seriously.

Because there are an infinite number of possible modes in ALGOL 68, such an assignment instruction will have to find out the mode of the source from the data itself. This suggests the use of run time descriptors, discussed shortly.

Another feature of ALGOL 68 that has a profound effect upon the run time design is the use of the heap and its attendent storage management problems. Unlike objects created by local generators, objects created by heap generators cannot be accommodated on a stack. They require a garbage collection type of storage management. The garbage collector must be aware of all run time objects in order to trace and mark all pointers.

Finding most pointers presents no problem since they are located on or are pointed to by the identifier stack, which contains, roughly speaking, the variables. However, the results of many virtual instructions are put on a different stack, the working stack, which is used for temporary storage. For example, if the results of an operator is a heap ref ref [] int, the first pointer in the chain will not be traceable from the identifier stack.

In order to trace all objects pointed to by the working stack, it must be possible for the garbage collector to identify the mode of each object on the working stack. One way of accomplishing this is to have the compiler produce a list of all possible configurations of the working stack and give each one a number. During run time, a global variable giving the current configuration of the working stack would be maintained.
Whenever the configuration changed, this number would have to be updated. Since nearly every ALGOL 68 statement affects the working stack, this would be not only time consuming, but wasteful of space as well, not only the space used to store the configurations (templates), but the instructions used to update the configuration number as well.

An alternate method of providing the garbage collector with the configuration of the working stack is to maintain the mode information on the working stack itself. The descriptors needed for the assignment instruction provide exactly the correct information needed by the garbage collector. Thus the use of descriptors eliminates the need for updating the working stack configuration number, and in fact, dispenses with the entire idea of a working stack configuration. At any point in time the garbage collector can find out what is on the working stack simply by looking there.

As a consequence of this, it is possible to design a general purpose garbage collector. A hardware instruction "garbage collect" can be provided or emulated. Such an instruction would eliminate the need for fetching instructions from main memory during garbage collections, since all the instructions could come from a high speed read only control store, or be built into the timing circuits themselves. Furthermore, parallelism in the microprocessor or hardware can be exploited. This design does not preclude either a recursive garbage collector or a non recursive one, so the type most appropriate can be chosen. If the target machine is microprogrammable, the best solution would be a recursive garbage collector using a paged control store for the stack. When the stack filled up it could be paged into main memory, which would be used as a backup memory. Conceivably the main memory itself could be paged onto a disk or drum if need be.

In summary, two significant advantages of using descriptors are the possibility of building a hardware or microprogrammed general purpose assignment instruction, something not possible without descriptors due to the infinite number of potential modes it may have to deal with, and the possibility of a hardware or microprogrammed garbage collector without the necessity of maintaining configuration numbers or templates. Other advantages will be discussed later.

The descriptors referred to earlier will now be explained in detail. The reader should avoid confusing them with the multiple value descriptors used in the ALGOL 68 report. Historically, the use of the word "descriptor" in the sense used here predates the ALGOL 68 report by at least a decade.

A descriptor is an object present in the machine at run time. It consists of three fields

type	scope	address

The number of bits required for a descriptor is machine dependent, but for a given target machine all descriptors have exactly the same size. This is a crucial property, without which the entire method to be presented would not work. Every instance of a value, be it a plain value, a structured value, a multiple value, or other value has associated with it a descriptor. In addition, certain objects which are not values, such as unions and parallel control blocks also have descriptors.

Values are accessed only via their descriptors. All instructions addressing objects contain not the address of the value, but the address of the descriptor. The address field of the descriptor gives either the address of the value, or the address of another object from which the value can be found.

The scope field of a descriptor contains the scope of the value.

The type field of a descriptor provides mode related information. The types are

int	long int	reference
real	long real	row
bool	short int	structure
char	short real	union
bits	long bits	void
bytes	long bytes	empty
proc	long long int	semaphore
format	etc	parallel control block

Note that these are related to modes but are not modes, for example proc is not a mode, but it is a type. Additional types for longer and shorter plain values may be provided. The number of bits in the type field is determined by the number of lengths of plain values provided, an implementation dependent parameter.

Every identifier occurring in the program is associated with a descriptor. For every procedure in the program there is a unique set of N identifiers used within all the ranges of that procedure. This set includes the formal parameters. When a procedure is entered, space is reserved on a stack called the identifier stack for a descriptor for each identifier.

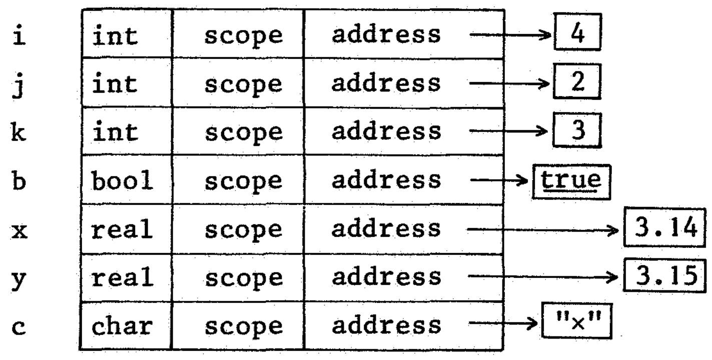
One important optimization will be used. Values of modes not beginning with ref will be stored as if they had been declared as ref modes. This may seen peculiar at first, but the distinction between an int and a ref int at run time is of little value unless one assumes the existence of immediate operand instructions, something we will not due because of their lack of universality. Keeping that distinction turns out to be quite costly however. In other words, storage for i in int i=4 is allocated just as though it had been declared as int i :=4. This causes no problems because any run time operation that need be performed on i can still be performed. This optimization does not change the syntax or semantics of an ALGOL 68 program, of course, any more than using the same representation for integers and reals, as many ALGOL 60 compilers do.

The address field of a plain value descriptor points to a cell containing the value.
For example, consider

```
proc p= (int i,j, ref int k, bool b) void:
begin  real x,y; char c;
       skip
end
```

The identifier stack for this procedure contains:

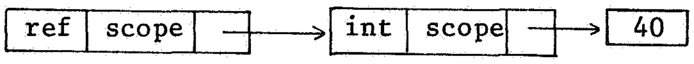
variable



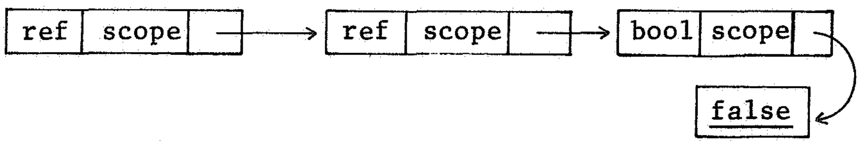
Note that the representation of i and k are the same, despite their differing modes. This is the optimization discussed. Thus as far as the run time system is concerned, it is possible to use i as a destination in an assignation, but of course no such instructions will ever be generated by the compiler. All machine instructions know whether they need a reference in front of the mode or not, so no ambiguity arises. For example, if the standard prelude integer addition operator is given i and j, it knows that it needs two integers, not two ref int's. Similarly, an assignation with i as source and j as destination automatically implies that i must be an int and j must be a ref int.

All values of mode ref m for m a mode of the form ref m1 , have a "reference" type descriptor, with the address field pointing to the descriptor for m. Some examples follow:

ref ref int



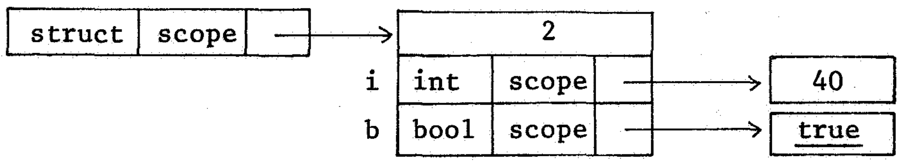
ref ref ref bool



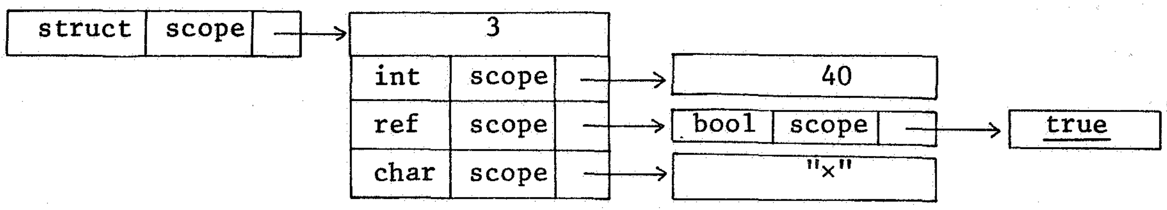
At this point the optimization should become clear. If it were not done, a an extra descriptor would be needed for ref int etc., and since it is expected that ref int will be more common than int, this price seems too high.

Descriptors for structured values have type "struct", with the address pointing to a block of memory of size N+1 descriptors, where N is the number of fields in the structured value. The first descriptor space in the block consists of the integer N. The remaining descriptor contain descriptors for the fields. For example,

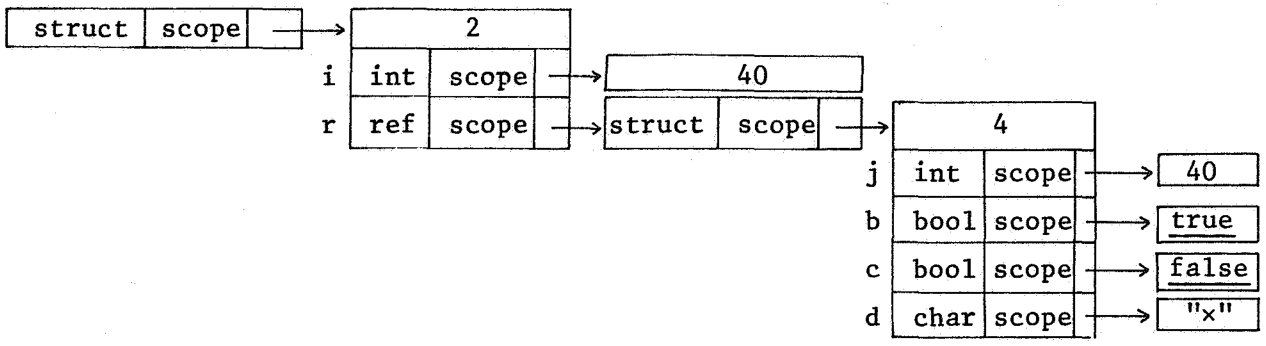
struct (int i, bool b)



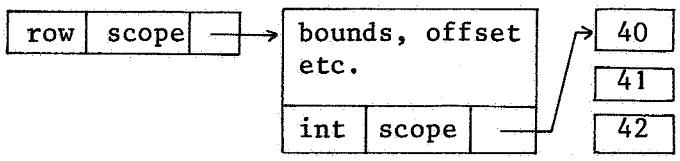
struct (int j, ref bool bb, char c)



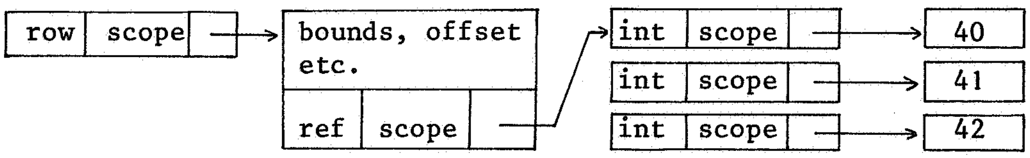
struct (int i, ref struct (int j, bool b,c, char d)r)



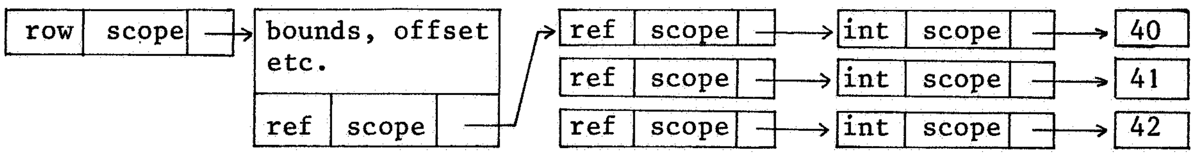
Descriptors for multiple values have type "row" with the address pointing to an ALGOL 68 multiple value descriptor, as discussed in the report. In addition to the fields described by the report, and certain other fields required for garbage collection, the multiple value descriptor must contain a descriptor of the type appropriate for the elements. For example, [1:3] int



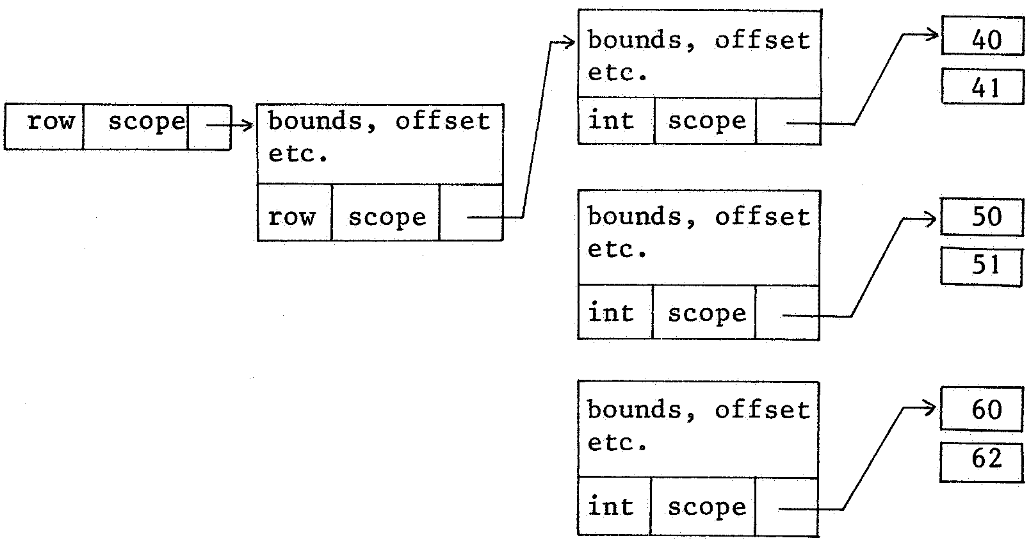
[1:3] ref int



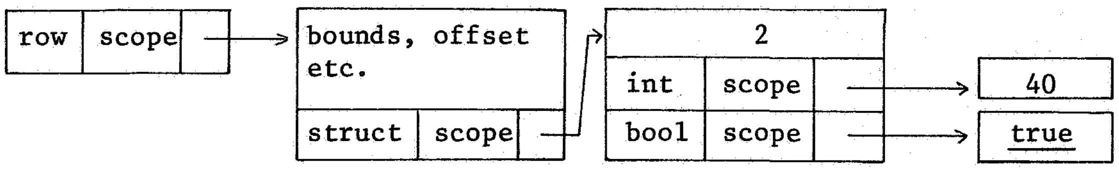
[1:3] ref ref int



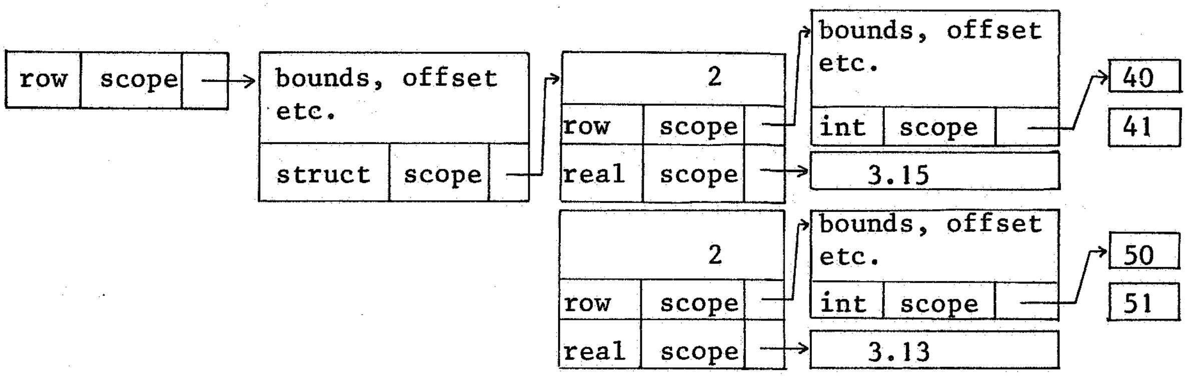
[1:3] [1:2] int



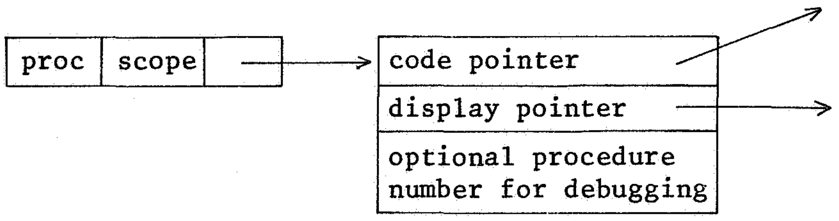
[1:1] struct (int i, bool b)



[1:2] struct ([1:2] int ii, real r)

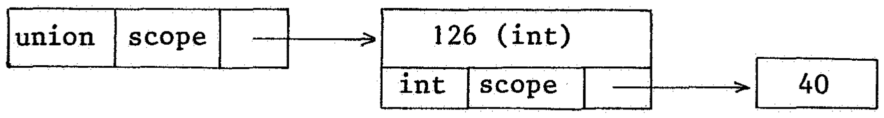


Descriptors for procedures have type "proc" with the address pointing to a block containing a code pointer, a display pointer, and (possibly) a procedure number. For example,

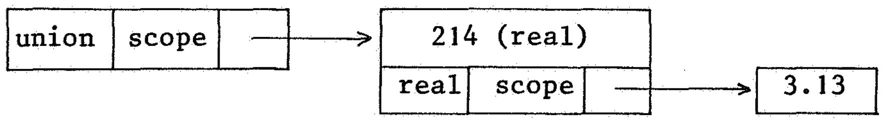


Descriptors for unions have type "union" with the address pointing to a block similar to that of a proc. The first word of this block contains a mode number and the second word contains a descriptor for the value. The mode numbers may be assigned in any convenient way. For example,

<u>union</u> (<u>int</u>, <u>real</u>)



or



As with the primitive modes, the run time representation of a structure and reference to a structure are the same, i.e. The initial reference is suppressed as an optimization. The same arguments apply to rows and unions as well.

<u>Advantages of a descriptor based machine</u>.

1. Assignations can be carried out directly in hardware, or in any event, at a level below that of the virtual machine language. Since the majority of statements in higher level languages are assignment statements, this will be a highly efficient method of carrying them out.

2. Garbage collection can be carried out directly in hardware, because all the information needed is in the machine at run time.

3. The compiler is freed from having to generate instructions to update the working stack configuration number, and the running program is spared the time and space required to carry out this updating. Furthermore the compiler need not waste time trying to optimize this calculation, since it is no longer necessary.

4. Because descriptors have a uniform size, a highly efficient form of addressing is possible. This saves both space and time. It will be discussed in detail later.

5. Unions have no residuals in this representation. In other representations, the unused static part of a union causes complications for the garbage collector.

6. There is no need to implement references to local objects and heap objects differently. In some representations, <u>int</u> i; and <u>heap</u> <u>int</u> j; would be handled quite differently. For the local variable, i, space would be reserved on the identifier stack for an integer, whereas for the heap variable j, the identifier stack would contain a pointer to the heap. Thus not only is the amount of space required different, but the garbage collector must be informed of which integers are pointers and which are not. With descriptors it can just look.

7. Only descriptors are passed parameters. This makes the procedure mechanism simpler. The called procedure itself can examine.
   The actual parameters' descriptors to see if copying is needed. In some cases the decision to copy or not copy can only be made at run time.

8. Uninitialized variables can be easily detected in many cases by having the address field of an uninitialized descriptor set to 0. Because there are no extra bits in integers and reals and bits and bytes in general, this information would be difficult to provide in other representations.

9. Scope cheeking can be provided in hardware, and furthermore, it can be done in parallel with other actions, such as copying.

Memory organization.

Memory is conceptually divided into a number of independent segments:

1. the program
2. the administration
3. the identifier stack
4. the working stack
5. the local generator stack
6. the heap
7. the garbage collector's stack
8. constants and environment inquiries.

The program segment contains the executable code constituting the program. Little more need be said about it.

The administration stack contains information saved when a procedure is called, and which must be later restored, including

1. all the stack pointers
2. the display
3. the return address
4. information for dumping and tracing (debugging)
5. information for managing storage upon range entry or exit.

The identifier stack contains an entry for each identifier appearing in an identity declaration, and for a few compiler created internal identifiers.

The working stack can be used for or by:

1. collateral clauses
2. row displays
3. structure displays
4. assignations and identity declarations
5. operands of standard prelude operators
6. results of ENCLOSED clauses
7. results of procedure calls

8. results of operators
9. results of deproceduring
10. results of certain coercions, e.g. widening
11. subscripts
12. format elaboration
13. generators
14. selections

as well as other constructions when that is convenient, e.g. identity relations.

The local generator stack is used for local generators, excepting their descriptors. The identifier stack contains the descriptor for each identifier and the local generator stack or heap contains the rest of the object. The heap is used by heap generators.

The garbage collector stack is used by the garbage collector. Garbage collection will be discussed later.

It is not necessary to maintain a separate segment for each of these conceptually separate items. On a machine such as the IBM 370 with a moderate sized address space (2^{24} bytes) it may be better to have a few large segments than a number of small ones. The merging of segments is not accomplished without some cost in complexity however, and this is reflected in the size and performance of the compiler and program. At the present time the computer industry is undergoing a transition from computers without virtual memory to computers with virtual memory, and this design is something of a compromise, with an emphasis on the future, rather than the past.

The program, constants and environmental inquiries are all static and can be merged without problems. The administration stack, identifier stack and local generator stack can also be merged easily, giving rise to the range stack.

It is desirable to split the working stack into two parts. The descriptor working stack (DWS) contains only descriptors. The object working stack (OWS) is used for the objects themselves. The descriptors on the DWS may point to the local generator stack, object working stack or heap.

Whenever the result of an operator, procedure etc. must be created and put onto the working stack, the descriptor goes on the DWS and the rest goes on the object working stack.

The reason for maintaining the DWS as a separate segment is as follows. This arrangement enables the garbage collector to find all objects not pointed to by the identifier stack. Every object has a descriptor either on the working stack or the identifier stack (or both).

The origin of the DWS is globally available, and the top of it is kept in a register, the DWS pointer, so the garbage collector can trivially locate all objects pointed to by the working stack, regardless of which procedure activation they occur in.

Space on both the DWS and object working stack are recovered dynamically as objects are removed from the working stack. When an object is popped off the DWS, the DWS stack pointer is reset, and if the object is on the object working stack, its stack pointer is reset too.

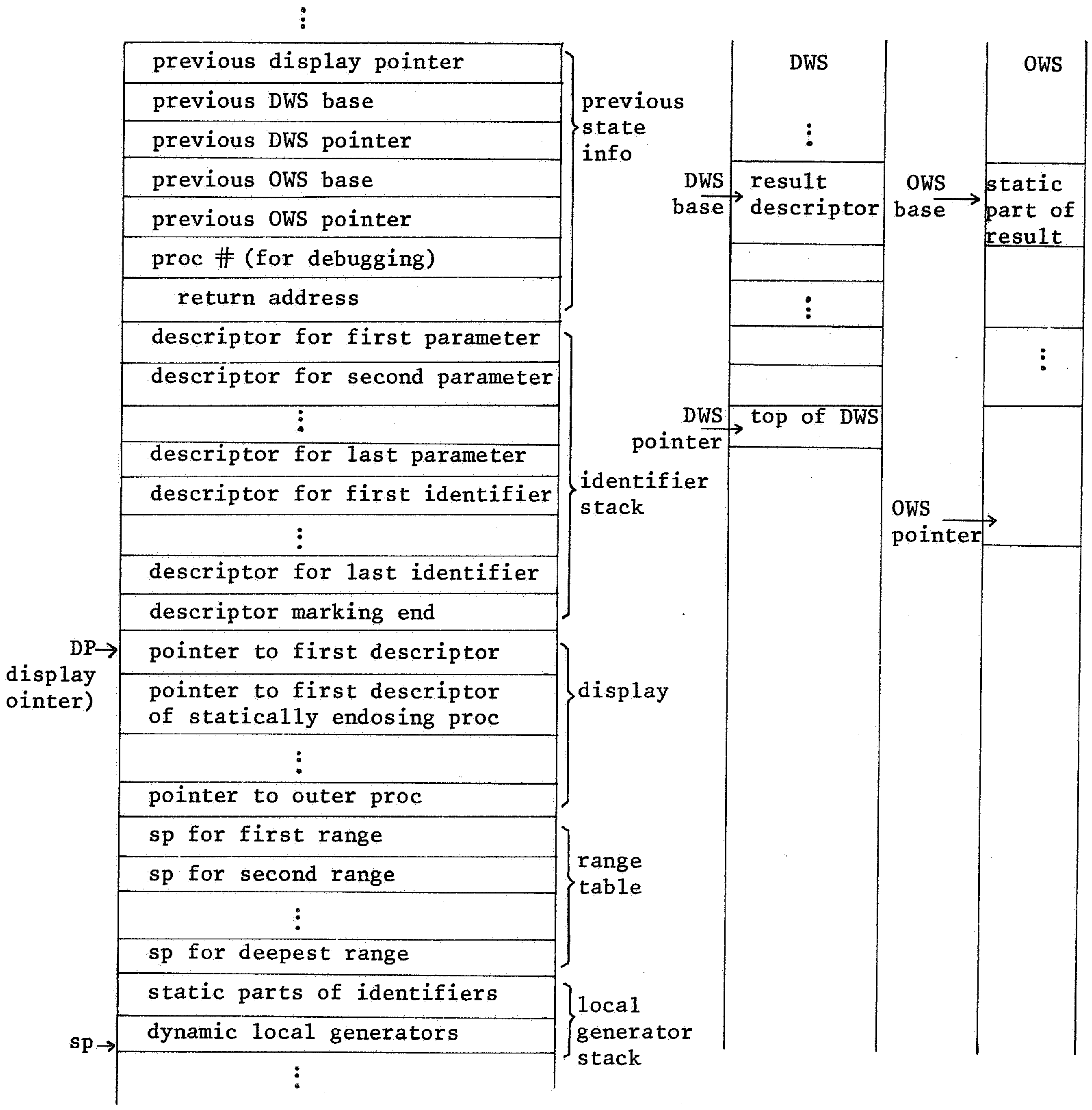
In addition, because all objects on the DWS have the same size, the DWS can be indexed into to find the N-th object from the top. If objects were of different sizes, this would not be possible.

Furthermore, although the working stack could be merged with the range stack, this would introduce complications into the compiler. Because of the rules concerning the scope of local generators, the compiler would otherwise have to reserve local generator space in advance, and figuring out when to do this and when it could be optimized out increases the complexity of the compiler.

Now we turn to the problem of implementing the memory organization. There are 5 segments to be implemented: the program, constants, and range stack in the first, and the two working stacks as the second and third, and the heap and garbage collector stack as the fourth and fifth.

The range stack is organized as a series of frames, one for each procedure entered, but not yet left. The main program is regarded as a procedure for these purposes. The structure of a frame is shown below.

One frame of the range stack



The range stack frame consists of 5 parts:

1. the previous state information
2. the identifier stack
3. the display
4. the range table
5. the local generator stack.

The previous state information contains all the pointers that need to be saved. The identifier stack contains one descriptor for each formal parameter and identifier. The display contain pointers to the start of the identifier stack for the frame itself, and all the statically enclosing procedures. The range table has as many entries as the maximum depth of range nesting. Whenever a range is entered the value of SP is saved in the corresponding position so that it can be restored when the range is left. Thus although space for descriptors is reserved for the entire time a procedure is active, local generator space is released upon range exit and need not wait for procedure exit. The local generator stack follows the range table.

On a machine with a segmented memory, the program, range stack, DWS, OWS, and heap would be separate segments, each free to grow and contract independent of one another. On a machine such as the IBM 370, with a linear address space of moderate size, a portion of the address space would have to be allocated to each segment, e.g. addresses between 15,000,000 and 16,000,000 for the DWS. If the program actually used a quarter of a million 32 bit descriptors, it would run out of address space, but it seems exceedingly unlikely that such a situation would ever occur except in programs written for the purpose of seeing what happened when it occurred. It should be noted that the DWS is never pointed to, and therefore it is completely relocatable. The entire DWS could be moved in the address space and only the DWS pointers would have to be changed. If a hardware register were available to dynamically relocate all DWS references, only this register would have to be changed when the DWS was moved.

If the target machine is a second generation machine with no virtual memory and a small address space, and a programmer writes a large, highly dynamic program, in which, for example, the operands of a formula frequently invoke other operators whose operands invoke still other operators etc., the programmer will discover that the machine is finite. There is no magic way to allow programs that need an arbitrarily large amount of space to run on a small finite machine. It is true that by merging stacks the program might have been able to run a bit longer, but it is also known that when programs using dynamic storage allocation are stopped because storage is exhausted, they do so in a most unpleasant way. In particular, near the end they cause successively more frequent garbage collections, each recovering successively less space. Thus the merged stacks would have an obvious advantage for programs that needed only the space available elsewhere, but the obvious disadvantage of prolonging the death throes of programs that were not going to make it anyway, and few computer centers give refunds for CPU time used to discover that there was not enough memory for the program.

Specification of the ALGOL 68 virtual machine.

Each virtual machine instruction consists of an optional label field, an opcode and zero or more addresses. Instructions are terminated by semicolons. Eight kinds of addresses are used

1. dereferenced identifier
2. non dereferenced identifier
3. environment inquiry
4. constant table
5. working stack reference
6. integer
7. label
8. null.

These will now be discussed in turn.

1. Dereferenced identifier.

Dereferenced identifiers will be written as a triple in the form:

(1, display level, id number)

where display level is 0 for the current static level, 1 for the enclosing level etc. The higher the number the more global the range of the level. Each identifier within a display level (which in this design is a procedure) has a unique number, with numbering beginning at 0. The actual parameters of a procedure are also accessed using this mechanism, with the first parameter having the address (1,0,0), the second parameter having the address (1,0,1) etc. The numbering of the identifiers follows that of the parameters.

2. Non dereferenced identifier.

These addresses are written in the form:

(2, display level, id number)

The difference between type 2 addresses and type 1 addresses is that the latter are not dereferenced and the former are. Dereferencing is provided as an explicit instruction, therefore type 1 addresses are not strictly necessary, but are useful for optimization. If the first identifier in display level 0 has mode reference-to-integral, then a push operation with address (1,0,0) will push an integral onto the working stack, whereas a push operation with address (2,0,1) will push the address of an integral onto the working stack.

3. Environment inquiry.

Addresses of the form:

(3, index, length)

are used to make environment inquiries. The list of objects in the standard environment of the revised report is not known as this is being written (1973), but for the original report these are: int lengths, L max int, real lengths, L max real, L small real etc. These are numbered starting at 0. The second parameter, length, is 0 if no longs or shorts are used, 1 for long, 2 for long long etc. For short, short short etc. the values -1, -2 etc. are used. For example (3,1,2) is the address of long long max int.

4. Constant table.

This form of addressing is used to access constants preloaded by the compiler. The constant table is global to an entire program, to avoid having small constants such as 0, 1 and <u>true</u> stored in many procedures. An address of the form:

(4, index)

accesses the constant table, entry index, where index is a zero origin counter.

5. Working stack.

An address consisting of a 5 pops one object off the working stack. The object popped off may be of any mode or size, and the stack pointer is reset.

6. Integer.

An address may be an integer. This form of addressing is only used in situations where no ambiguity arises. Integer addresses are commonly used for counting, e.g. the number of fields in a structure, or numbering, e.g. a mode number.

7. Label.

Labels are used for internal jumps. For reasons of efficiency it would be desirable for the compiler to generate all labels of the form "ln" where n is an integer.

8. Null.

It occasionally occurs that an address is optional in a certain instruction. The loop instruction has an address field for the controlled variable. When compiling <u>to</u> 3 <u>do</u> newline it is nevertheless necessary to provide an address in that field. In such cases 8 is used.

Some instructions have multiple addresses. It is important to note that the order in which the addresses are evaluated is significant. As an example of this consider the evaluation of the formula (i+j) - (k+1). This will be compiled as code to elaborate the left operand and push it onto the working stack, code to elaborate the right operand and push it onto the working stack followed by the (conceptual) instruction sub 5,5 which proceeds in 4 steps:

1. The right operand is popped from the stack
2. The left operand is popped from the stack
3. The subtraction is performed
4. The result is pushed onto the stack.

Note that the right operand is popped before the left operand. Thus we have the rule: address fields are elaborated right-to-left. This holds for all virtual instructions.

Virtual instructions.

Each virtual instruction has a mneumonic given below, although once the compiler has passed the debugging stage these should be mapped onto the integers for efficiency. Instructions may have 0 or more addresses, as described earlier. A few of the virtual instructions are really pseudo-instructions in that they provide information but generate no object code.

The virtual machine has been designed in such a way as to ease the construction of a hardware or microprogrammed ALGOL 68 machine. In particular, an implementation involving descriptors is not precluded. Each instance of a value would have associated with it a fixed sized descriptor, giving information about the mode of the value, the scope of the value, and the name of the value. In many cases these descriptors could be manipulated instead of the values themselves. For example, the instructions for standard prelude operators might put descriptors rather than values on the working stack. Also, in the case of multiple values this might save considerable time in many cases.

Below is a listing of the virtual instructions. A detailed discussion follows.

List of ALGOL 68 Virtual Instructions.

Group 1. Declarations and structuring

1.	begin	
2.	end	
3.	mode	mode nr, type, details, bounds
4.	constant	index, mode nr, value
5.	procedure	proc nr, depth, nr of params, nr of ids, range depth, modes
6.	endproc	proc nr

Group 2. Generators.

7.	idgen	index, mode nr, range
8.	locgen	mode nr
9.	heapgen	mode nr
10.	skipgen	mode nr
11.	nilgen	

Group 3. Assignment, pushing, popping etc.

12.	assign	mode nr, source, destination
13.	iddecl	mode nr, source, destination
14.	push	mode nr, source
15.	pop	mode nr, destination

Group 4. Operators, coercions, etc.

16.	dyop	op nr, length, left operand, right operand
17.	monop	op nr, length, operand
18.	deref	mode nr, source
19.	widen	type, length, source
20.	idrel	mode nr, left source, right source
21.	monbound	mode nr, type, source
22.	dybound	mode nr, type, left source, right source
23.	random	length

Group 5. Selecting and slicing.

24. select	mode nr, primary, field nr
25. refselect	mode nr, primary, field nr
26. subscript	mode nr, primary, subscripts
27. refsubscript	mode nr, primary, subscripts
28. slice	mode nr, primary, result dims, bounds
29. refslice	mode nr, primary, result dims, bounds
30. descrip	mode nr, primary, result dims, bounds

Group 6. Creating objects.

31. createproc	mode nr, label, display, range
32. createunion	union mode nr, value mode nr, source
33. createstruct	mode nr, fields
34. createrow	mode nr

Group 7. Flow of control.

35. enterrange	range nr, nr of ids
36. exitrange	range nr, nr of ids
37. call	mode nr, primary, parameters
38. deprocedure	mode nr, primary
39. return	mode nr, proc nr
40. jump	label
41. dirtyjump	label, proc nr, range nr
42. jumptrue	source, label
43. jumpfalse	source, label
44. loopsetup	variable, counter, from, by, to, label
45. while	source, label
46. loop	variable, counter, by, label
47. fork	nr of branches, control block, successor, labels
48. join	label
49. internalcall	label
50. internalreturn	
51. conform	mode nr, source, nr of alternatives, outlabel, alternatives

Group 8. Miscellaneous.

52.	nilcheck	mode nr, source
53.	check	type
54.	copy	mode nr, source

Semantics of the virtual instructions.

1. begin

The first virtual instruction of each program is begin. This serves to identify the start of a program. A comment may appear after begin.

2. end

The last virtual instruction of a program is end. This serves to identify the end of a program.

3. mode   mode nr, details, bounds

Mode is a pseudo instruction used to convey information about modes to the virtual machine. Mode statements are not executable and do not generate object code. The term mode as used in this context is similar to, but not identical with, the ALGOL 68 concept of an actual mode declarer. In particular actual bounds are considered to be part of a mode. Thus as far as the virtual machine is concerned, [1:n] int and [1:m] int are distinct modes. A mode virtual instruction provides enough information for a local or heap generator to generate an object. To avoid confusion, the term "virtual mode" will be used when an ambiguity might otherwise result. Examples of distinct vitual modes are

[1:10] int	ref int
[1:5] int	ref long long int
[1:n] int	union (int, bool)
bool	struct (int [1:2], bool)

Note that field selectors are not part of the virtual mode, because they are not used at run time. Fields are selected by position.

Each virtual mode is assigned a unique number by the compiler, presumably consecutively from 20, since 0-19 are reserved. The ordering is arbitrary. Virtual mode numbers are used to identify virtual modes. For example, the virtual heap generate instruction specifies the number of the virtual mode of the object to be created on the heap. The first address field of the mode instruction contains this compiler generated identifying number.

The second address field, type, contains an integer specifying the outermost structure, a row of something, a structure with some fields, a union of something, a reference to something, a long something, etc. The following types are available:

0. int
1. bool
2. char
3. real
4. bits
5. bytes
6. format
8. compl
9. string
10. sema
11. void
12. long
13. short
14. row
15. ref
16. proc
17. union
18. struct
19. parallel processing control block

Virtual modes 0-11 are predefined as above and should not be explicitly declared. Virtual modes 12-19 should be avoided to prevent confusion on the part of the human beings doing the implementation.

The third address field, details, continues the specification of the virtual mode. In the case of the primitive modes, with types 1-11, the type provides all the needed information, therefore the details and bounds fields are omitted. For other virtual modes, more information is needed, for example, if the type is 15, reference to, the details field specifies the virtual mode referred to The interpretation of the details field depends on the type field, as follows.

For types 12-15 (long, short, row, ref) the details field contains a virtual mode number. These modes have the form:

long m
short m
[] m or [,] m or [,,] m etc.
ref m

The details field contains the virtual mode number of m. Note that for modes of the form [][] m, the details field contains the virtual mode of [] m.

For types 16-18 (proc, union, struct) several virtual modes must be specified, and thus the details field contains a list of virtual modes separated by commas. For proc, these are the virtual modes of the parameters, followed by the virtual mode of the result. For union these are the virtual modes from which it is united. For struct these are the virtual modes of the fields.

The details field for a parallel processing control block is an integer equal to the number of constituent units in the parallel clause it describes.

The fourth address field, bounds, is omitted unless the type field contains a 14 (row). For rows, this field is used to specify where the actual bounds are to be found when a generator of this mode is elaborated. The bounds field consists for an N-dimensional multiple value contains 2N+1 subfields. The first subfield contains the number of dimensions. For [] m this is 1, for [,] m this is 2, etc. The remaining subfields specify the lower bound of the first dimension, upper bound of the first dimension, lower bound of the second dimension, etc. Each bound may be any of address kinds 1-5 as described earlier. Thus an actual bound may be in a variable, in the constant table, or on the working stack. In the case of a bound which is a closed clause, the compiler will generate code for evaluating the closed clause, possibly as an internal procedure, with the value being left on the working stack. The subfields are evaluated right to left, for the same reasons given earlier.

As an example of the use of the mode virtual instruction, consider the following modes.

```
 (1) bool
 (0) int
(20) long long real
(21) ref ref ref bool
(22) [1: if random <.5 then n else m fi] int
(23) proc (int, int) bool
(24) struct (long long real, proc (int, int) bool
(25) union (ref ref ref bool, long long real, char)
```

If these modes were all used in the same program, the following virtual instructions might be generated. The symbol ¢ will be used to denote comments to aid the reader.

```
mode 20, 12, 26;                ¢ mode 20 is long long real ¢
mode 21, 15, 27;                ¢ mode 21 is ref ref ref bool ¢
mode 22, 14, 0, 1, (4,0), 5;    ¢ mode 22 is a row. ¢
mode 23, 16, 0, 0, 1;           ¢ mode 23 is proc (int, int bool) ¢
mode 24, 18, 20, 23;            ¢ mode 24 is a structure
mode 25, 17, 21, 20, 2;         ¢ mode 25 is a union ¢
mode 26, 12, 3;                 ¢ mode 26 is long real ¢
mode 27, 15, 28;                ¢ mode 27 is ref ref bool ¢
mode 28, 15, 1;                 ¢ mode 28 is ref bool ¢
```

One thing should be noted about mode 22. It is assumed that entry 0 in the constant table is 1, and that whenever an object of mode 22 is to be generated, the upper bound will have been elaborated and pushed onto the working stack. Thus upon encountering

... heap [1: if random <.5 then n else m fi] int ...

the compiler will first arrange to have the value of n or m on the working stack, either by in line code or an internal procedure call, then generate a virtual heapgen instruction with address 22.

4. constant   index, mode nr, value

There is a single constant (denotation) table loaded along with the object program in which constants of all modes and all ranges are kept. The table is global to the entire program, and can be accessed from anywhere within it. Each constant has a number, by which it is accessed, with the numbering beginning at 0. The address (4,i) references the i-th entry in this table.

The field "index" is the number of the constant, presumably 0 for the first one, 1 for the second one, etc. The field "mode nr" is the number of the mode of the constant. The "value" field contains the constant itself. String constants are enclosed within quotation marks. The boolean denotations are T and F respectively.

For each unique constant occurring in the source program, one "constant" pseudo instruction will be generated, no matter how many times that constant occurs in the source program. Some examples follow.

```
constant 0, 0, T;
constant 1, 3, 3.14;
constant 2, 2, x;
```

5. procedure   proc nr, depth, nr of params, nr of ids, range depths. modes

This non-executable statement is used to indicate the beginning of a procedure or operator definition. At run time, no distinction is made between procedures and operators. For simplicity, we will refer to "procedure" in this report to include both procedures and operators. Each procedure is assigned a number by the compiler, which is the first address field.

"Depth" is the static depth of nesting of the procedure. Depth = 1 corresponds to procedures declared in the outermost range. Depth is needed to determine the number of entries in the display, including the outermost one. It is intended that only procedures are counted in nesting, not ranges. For example, in

```
begin      int i; skip;
    begin      int j; skip;
        begin      proc p= (int i,j) bool: (int k;skip); skip
        end
    end
end
```

procedure p has depth 1.

The decision to organize memory by procedures has been made for two reasons. First, to reduce the size of the display, thus saving not only space, but time as well, since copying it will be faster. Second, to speed up range entry and exit, because range entry and exit will be much simpler than procedure entry and exit. This strategy will not release all the space used in a range until the containing procedure is left, but if descriptors are used, only the space for a descriptor will be held onto, and not the space for the object itself. This will be discussed later.

The third field contains the number of parameters. This information is needed for storage allocation, as the actual parameters are copied onto the called procedure's identifier stack.

The fourth field is the number of identifiers in the procedure. Each of these is accessed via the identifier stack, i.e. address forms (1,i,j) and (2,i,j). This number is needed for storage allocation purposes.

The fifth field, "range depth" is the depth of range nesting at the deepest point in the procedure.

The sixth field, "modes" consists of a list of the mode numbers of the parameters, if any, followed by the mode number of the result.

6. endproc   proc nr

This statement delimits a procedure definition. The number specifies which procedure definition is finished. The pseudo-instructions "procedure" and "endproc" are used to bracket procedure definitions.

7. idgen   index, mode nr, range

For every identifier occurring in a procedure, there is one idgen instruction. These are used to allocate space for the descriptors (or static parts) of all objects, including procedures of course, the static part of a procedure being a few pointers, rather than code. The "index" is a number used for accessing the object. All identifiers in a procedure are numbered consecutively, regardless of which range they occur in. If the procedure has N formal parameters, then these are numbered 0, 1,... N-1, and identifiers are numbered beginning at N. Presumably the idgen statements will occur in consecutive order. They must appear before the first "enterrange" and before the first executable statement. Mode nr is the mode of the object. Range is in the range in which the object was declared. As an example, consider the program

```
begin   proc p= (int i,j) void:
    begin  int k; real x,y; skip
    end;
    p(19,29)
end
```

This will be compiled into

```
begin;
mode        20, 16, 0, 0, 11;           ¢ mode 20= proc (int,int) void ¢
constant    0, 0, 19;                   ¢ integer 19 ¢
constant    1, 0, 29;                   ¢ integer 29 ¢
idgen       0, 20, 0;                   ¢ p is only identifier here ¢
procedure   1, 1, 2, 3, 1, 0, 0, 11;    ¢ 2 parameters, 3 identifiers ¢
idgen       2, 0, 0;                    ¢ k ¢
idgen       3, 3, 0;                    ¢ x ¢
idgen       4, 3, 0;                    ¢ y ¢
:
endproc     1;
call        20, (1,0,1), 2, (4,0), (4,1);
end;
```

within the procedure, the first actual parameter can be accessed using the address (2,0,0) and x can be accessed using the address (2,0,3) or (1,0,3), depending whether the address or contents of x is needed. Within p, p itself can be accessed using (2,1,0), for a recursive call, for example.

It should be noted that the actual storage for a dynamic object will be reserved within the procedure itself using a locgen or heapgen instruction. Idgen just reserves space for the descriptor and the static part, i.e. the part whose storage depends only upon the mode.

Idgen is also used for temporary variables, e.g. loop counters.

8. locgen   mode nr

This instruction generates an object of the specified mode on the local generator stack. The descriptor for the generated object (or address, if descriptors are not used) is pushed onto the working stack.

9. heapgen   mode nr

This instruction generates an object of the specific mode on heap. The descriptor for the generated object (or address, if descriptors are not used) is pushed onto the working stack.

10. skipgen   mode nr

This instruction generates a new instance of some value of the specified mode. The object so generated is placed on the working stack. The virtual machine implementer is free to choose the value.

11. nilgen

A nil is pushed onto the working stack.

12. assign   mode nr, source, destination

This instruction is used for all assignations.
Mode nr specifies the mode of the source, not the mode of the destination. The second and third fields specify the source and destination respectively. If the mode is 0 (integer), then the source must be an int and the destination must be a ref int. The details of the assignment are left open for the virtual machine implementer.

It should be noted that this instruction leaves the compiler considerable freedom, because the source may accessed via the working stack, the constant table, a display address etc. The compiler could always insure that the source was on the working stack, for example. Or alternately, the compiler could always insure that objects of certain modes and not other modes were on the working stack.

13. iddecl   mode nr, source, destination

This instruction is used for identity declarations, where necessary. It is expected that contracted identity declarations will be handled using idgen, which is much more efficient. If the identity declaration is not contracted, idgen is still needed to reserve descriptor space on the identifier stack, and in addition, iddecl is needed to put a descriptor there. Iddecl is similar to assign, except that mode nr is the mode of both the source and the destination, since they must both have the same mode (after coercions). The compiler can, of course, treat all noncontracted identity declarations as though they had been contracted, for convenience, if it wishes e.g. int i=4 can be treated as int i:= 4 if appropriate dereferencing is provided. This is purely an implementation point. The programmer will never know the difference.

14. push   mode nr, source

This instruction pushes an object of the specified mode onto the working stack. Source is the address of the object. Normally the source will be an identifier stack entry, dereferenced or not, a constant table entry, or an environment enquiry. It hardly makes sense to pop the top item off the working stack and then push it back onto the working stack.

15. pop   mode nr, destination

This instruction is provided for symmetry with push. It is in fact a special case of assign, with the source on the working stack. Pop removes the top item from the working stack and moves it to destination. If the destination is address type 8, the item popped is discarded, thus providing an instruction for voiding.

16. dyop   op number, length, left operand, right operand

This instruction is used for all the standard prelude, dyadic operators. Each operator has a number, which together with its length uniquely identifies it. Length = 1 is <u>long</u>. length = 2 is <u>long</u> <u>long</u> etc. Length = -1 is <u>short</u>, length = -2 is <u>short</u> <u>short</u> etc. Length = 0 is the standard length.

The third and fourth fields specify the operands. The result of the operation is placed on the working stack. The operator numbers, based upon the unrevised report follow. If the revised report contains changes to the standard prelude dyadic operators, this list will have to be modified.

The "and becomes" operators are not included. They are compiled as though the expanded form had been written.

<u>DYADIC OPERATORS</u>

R. 10.2.2	R. 10.2.4	(real,int)	54. ≤	(int,compl)	R. 10.2.8
(bool,bool)	(real,real)	38. <	55. =	74. +	(bits,bits)
1. ∨	19. <	39. ≤	56. ≠	75. −	91. =
2. ∧	20. ≤	40. =	57. >	76. ×	92. ≠
3. =	21. =	41. ≠	58. ≥	77. /	93. ∨
4. ≠	22. ≠	42. >	59. +	(real,compl)	94. ∧
	23. ≥	43. ≥		78. +	95. ≤
R. 10.2.3	24. >	(int,real)	R. 10.2.7	79. −	96. ≥
(int,int)	25. −	44. <	(compl,compl)	80. ×	97. ↑
5. <	26. +	45. ≤	60. =	81. /	98. □
6. ≤	27. ×	46. =	61. ≠	(compl,int)	
7. =	28. /	47. ≠	62. −	82. ↑	R. 10.2.9
8. ≠	29. ⊥	48. >	63. +	83. =	(bytes,bytes)
9. ≥		49. ≥	64. ×	84. ≠	99. <
10. >	R. 10.2.5	(real,int)	65. /	(compl,real)	100. ≤
11. −	(real,int)	50. ⊥	(compl,int)	85. =	101. =
12. +	30. +	(int,real)	66. +	86. ≠	102. ≠
13. ×	31. −	51. ⊥	67. −	(int,compl)	103. >
14. ÷	32. ×	(real,int)	68. ×	87. =	104. ≥
15. ÷:	33. /	52. ↑	69. /	88. ≠	105. □
16. /	(int,real)		(compl,real)	(real,compl)	
17. ↑	34. +	R. 10.2.6	70. +	89. =	
18. ⊥	35. −	(char,char)	71. −	90. ≠	
	36 ×	53. <	72. ×		
	37. /		73. /		

R. 10.2.10	(string,char)
(string,string)	125. +
106. <	(char,string)
107. ≤	126. +
108. =	
109. ≠	
110. >	
111. ≥	
(string,char)	
112. <	
113. ≤	
114. =	
115. ≠	
116. >	
117. ≥	
(char,string)	
118. <	
119. ≤	
120. =	
121. ≠	
122. >	
123. ≥	
(string,string)	
124. +	

17. monop op nr, length, operand

This instruction is used for all standard prelude monadic operators. Each operator has a number, which together with its length, uniquely identifies it. The length field is interpreted as in dyop. The result of the operation is pushed onto the working stack. The list of monadic operators (and procedures) from the unrevised report follows. The revised report may require some modifications.

MONADIC OPERATORS

R. 10.1	212. +	R. 10.2.9
(char)	213. abs	(string)
200. abs	214. leng	231. ctb
(int)	215. short	
201. repr	216. round	R. 10.3
	217. sign	(real)
R. 10.2.2	218. entier	232. sqrt
(bool)		233. exp
202. ¬	R. 10.2.7	234. ln
203. abs	(compl)	235. cos
	219. re	236. arcos
R. 10.2.3	220. im	237. sin
(int)	221. abs	238. arsin
204. -	222. conj	239. tan
205. +	223. -	240 arctan
206. abs	224. +	
207. leng	225. leng	R. 10.4
208. short	226. short	(int)
209. odd		241. /
210. sign	R. 10.2.8	(sema)
	(bits)	242. ↓
R. 10.2.4	227. abs	243. ↑
(real)	228. bin	
211. -	229. btb	
	230. ¬	

18. deref   mode nr, source

This instruction is used for explicit dereferencing. In many cases, use of addresses of the form (1,i,j) will provide the required dereferencing, but in those cases where the dereferencing must be explicit, for example if a ref ref ref ref bool must be coerced to a bool, this instruction can be used. Mode nr is the number of the mode before dereferencing. Source specifies where the object to be dereferenced is. The dereferenced object is placed on the working stack.

19. widen   type, length, source

This instruction performs widening. The "type" field selects the type of widening desired:

0 = int	to	real
1 = real	to	compl
2 = bits	to	[] bool
3 = bytes	to	[] char

If other types of widening are added to the revised report, they can be added to this list. The "source" field specifies the address of the object to be widened. "Length" specifies the length of the object, with length = 1 meaning long, length = 2 meaning long long, length = -1 meaning short etc. The result is put on the working stack.

20. idrel   mode nr, left source, right source

This instruction performs identity relations. Mode nr is the mode of the objects. The second and third fields specify their addresses. The result of this instruction consists of true or false being pushed onto the working stack.

21. monbound   mode nr, type, source

This instruction is needed because lwb and upb are not operators, thus the mode of the multiple value must be given explicitly. Monbound is a monadic operator, not a dyadic operator. The second field is:

0 for lower bound
1 for upper bound

Source specifies the object whose bound is desried. The result is pushed onto the working stack.

22. dybound   mode nr, type, left source, right source

This instruction performs dyadic lwb and upb pseudo-operations, such as 2 upb x. Other than that, it is the same is monbound.

23. random   length

This instruction is needed because random is neither a dyadic operator nor a monadic operator. Thus it forms a special category by itself. A random number is pushed onto the working stack. "Length" determines the length of the random number.

24. select   mode nr, primary, field nr

This instruction is used for selecting fields from structures. Mode nr specifies the mode of the structure. Primary specifies the address of the structure. Field nr specifies which field is desired, with 0 for the first one. The selected field is put on the working stack.

25. refselect   mode nr, primary, field nr

This instruction is needed as a consequence of section 2.2.3.5b of the unrevised report. It selects from a reference to a structure and puts a reference to one of the fields on the working stack. Consider the following examples

```
    struct (int i,j) s;       int k:= i of s;
    ref struct (int i,j) s;   i of s:= 2;
    ref struct (int i,j) s;   int k:= i of s;
```

In the first case "select" is used. In the second case "refselect" is used. In the third case, s is dereferenced, perhaps by address type 1, and select is used. Like select, the result of refselect is put on the working stack.

26. subscript   mode nr, primary, subscripts

This instruction is used to access a single element of a multiple value whose mode is specified by mode nr, and whose address is specified by primary. The last field contains all the subscript addresses, each address being a display address (1,i,j), or (2,i,j) a constant or environment enquiry, or a working stack reference. These addresses are elaborated right to left, as ususal, for reasons relating to the operation of the stack described earlier. The element chosen by the subscripts is pushed onto the working stack.

27. refsubscript   mode nr, primary, subscripts

This instruction bears the same relation to subscript as refselect bears to select. The need for this instruction arises from section 2.2.3.5c of the unrevised report. It is used in those cases where a <u>ref</u> [] <u>m</u> is subscripted to yield a <u>ref</u> <u>m</u>. Other than that, it is the same as subscript.

28. slice   mode nr, primary, result dims, bounds

This instruction creates a slice and leaves it on the working stack. The mode nr and primary pertain to the multiple value to be sliced. Result dims is the dimensionality of the slice. Bounds is a list of addresses, arranged in groups of 3. The first group of three is for the first trimscript, the second group of three is for the second trimscript etc. Each group of three consists of the lower bound, the upper bound, and the new lower bound. If any one (or more) of the 3 was omitted in the source program, it should be translated as address 8. This instruction creates a new multiple value descriptor and a new multiple value on the working stack.

29. refslice   mode nr, primary, result dims, bounds

This instruction bears the same relation to slice as refsubscript does to subscript, i.e. it is a result of section 2.2.3.5c of the unrevised report. It could be used, for example, when a ref [,,] m is sliced to yield a ref [] m. the result is left on the working stack.

30. descrip   mode nr, primary, result dims, bounds

This instruction is intended for creating a new multiple value descriptor, without creating a new multiple value itself. For example, in a[1:5]:= b[1:5]. A slice is made of b, and the multiple value descriptor and multiple value itself are copied to the working stack. A new multiple value descriptor is needed for a[1:5] but no new copy of the values themselves are needed. This instruction provides that. The parameters are the same as for refslice. The multiple value descriptor will normally be used as the destination of a subsequent assign or iddecl instruction.

31. createproc   mode nr, label, display, range

This instruction is used to create a routine from a routine text plus scope information. Mode nr specifies the mode of the procedure to be created. "Label" is the address of the executable code for the procedure. The scope of the procedure must be some range within some procedure (if any). The third field specifies which procedure, 0 for current one, 1 for enclosing one etc. Range is the range within the procedure whose scope is the scope of the procedure. The result is put on the working stack.

32. createunion   union mode nr, value mode nr, source

This instruction carries out uniting coercions. The first address gives the mode of the union to be created. The second address gives the mode of the object from which the union is to be made. The third address specifies where that object is. The union is left on the working stack.

33. createstruct mode nr, fields

This instruction takes a list of fields and from them creates a structure. The first address specifies the mode of the structure to be created. "Fields" consists of a list of addresses, specifying the fields. The fields are elaborated right to left, as usual, in other words, if all the fields are on the working stack, the last field is popped off first, etc. The result is left on the working stack.

34. createrow mode nr

This instruction creates a multiple value, including its multiple value descriptor, and leaves them on the working stack. The mode of this multiple value is given by mode nr. Unlike createstruct, where the number of fields is known at compile time, the number of elements in the row is not known. Therefore the elements themselves are not individually addressed, but rather are taken from the working stack. Thus this instruction removes 0 or more elements from the working stack, and replaces them by a single multiple value. The order of the elements on the working stack must be the order implicitly specified by the unrevised report, section 8.6.1.2. This may remove the need for copying in some cases.

35. enterrange range nr, nr of ids

The execution of this instruction invokes the administration necessary for entering a new range. "Range nr" is the level of the range being entered, with the outermost range of each procedure being 0. A procedure, or the main program, may have many ranges at level i, disjoint of course. The field "nr of ids" is the number of identifiers in the range, and is used for storage management.

36. exitrange range nr, nr of ids.

This is the inverse of enterrange. It is used to release space when a range is exited. The addresses have the same meaning as in enterrange.

37. call mode nr, primary, parameters

This instruction calls a procedure (or an operator). It supplies the mode of the procedure called and the address of its primary. The third field is a list of the addresses of all the actual parameters, in the same order they are listed in the source program, although they are elaborated right to left here.

38. deprocedure mode nr, primary

This is the same as call, but for procedures with no parameters.

39. return mode nr, proc nr

This instruction terminates execution of a procedure and returns control to the calling procedure. The two parameters are the mode and number of the procedure being terminated.

40. jump label

This instruction causes a jump to label. This form may only be used when the label is within the same procedure and range as the goto statement.

41. dirtyjump label, proc nr, range nr

This instruction is used to jump out of a procedure or range. Such jumps require storage administration actions to be performed. Most virtual machines will handle dirtyjump by interpretation, unraveling the stack range by range. This will be slow and deservedly so.

42. jumptrue source, label

This conditional jump instruction tests the source, and if it is _true_ jumps to label. This may not be used to exit a procedure or range. If that is needed , a conditional jump to a dirtyjump should be used.

43. jumpfalse source, label

This conditional jump instruction tests the source, and if it is _false_ jumps to label. It may not be used to exit a procedure or range.

44. loopsetup variable, counter, from, by, to, label

This instruction is used to set up _for_-loops.
"Variable" is the display address of the controlled variable. "Counter" is an internal variable used only in this statement and the corresponding loop statement. "From", "by" and "to" are from the _for_ statement, which may be display addresses, constants, working stack etc. This instruction performs:

```
variable  := from;
counter   := if by = 0 then -1
             elsf (by > 0  ∧ from ≤ to) then (to-from) ÷ by + 1
             elsf (by < 0  ∧ from ≥ to) then (from-to) ÷ by + 1
             else 0
             fi;
if counter = 0 then goto label fi;
```

Counter is an internal variable initially set to the number of times the loop is to be executed, if finite, and -1 if infinite as in : _while_ b _do_ S;

45. while source, label

This instruction is identical to jumpfalse. It is included for reasons of optimization.

46. loop   variable, counter, by, label

This instruction is used at the bottom of loops to jump back to the top. The meaning is:

```
variable +:= by
if (counter -:= 1) ≠ 0 then goto label fi;
```

47. fork   nr of branches, control block, successor, labels

This statement is executed whenever a parallel clause is begun. "Nr of branches" is the number of units in the parallel clause. "Control block" is the display address of the control block where the administration for the parallel ckause is kept. "Successor" is the label following the parallel clause, i.e. the place to jump to when the clause is finished. "Labels" are the labels for the constituent units.

48. join   label

This is a backwards jump to the corresponding fork at label. It is used at the end of a parallel clause. At the end of each unit is a jump to here.

49. internalcall   label

In addition to the procedures present in the program, there may be internal procedures used to eliminate the need for inline coding. For example, in

```
mode m = [0: (read(n); 100 + n × n)] int
```

the closed clause can be compiled as an internal procedure leaving its result on the working stack. Whenever a local or heap m is needed, the compiler can generate an internal call to this procedure. Internal procedures are only used for non recursive code. If the code is not known to be non recursive, then the full procedure call machinery is needed. Furthermore, internal procedures do not call any other procedures, internal or otherwise. This makes the administration simple.

50. internal   return

Return for the internal procedure call

51. conform   mode nr, source, nr of alternatives, outlabel, alternatives

This instruction is used for case conformity clauses. The mode nr and address of the object whose mode is to be tested are the first two parameters. The number of alternatives in the in part follows. That is followed by the label to jump to if there is no match. "Alternatives" consist of a list of pairs, mode, label. If the source has the mode of the i-th pair, a jump is made to the i-th label. For this statement the ALGOL 68 mode and not the virtual mode is the important one, so bounds in multiple values are ignored.

52. nilcheck   mode nr, source

If the source is nil, true is pushed onto the working stack, otherwise false is pushed onto it.

53. check   type

This pseudo instruction enables run time checking. Type specifies the type of checking

1. sopes
2. subscripts
3. integer overflow
4. real overflow
5. real underflow
6. assignment to nil
7. use of skip where improper

8. use of flexible submane, if relevant

A program may contain more than 1 check statement.

54. copy   mode nr, source

A copy of the object specified by source is made and its address or descriptor put on the working stack.

Example of an Algol 68 program and its translated form

```
( proc    compsqrt= (compl z) compl:
  begin   real x = re z , y = im z;
          real rp= sqrt((abs x + sqrt(x↑2 + y↑2))/2);
          real ip= (rp = 0|0|y/(2xrp));
          (x≥0 |rp ⊥ ip |abs ip ⊥ (y≥0 |rp|-rp))
  end;
  compsqrt(-1)
)
```

```
begin;
mode           20, 16, 11, 11;           ¢ proc (compl) compl ¢
constant       0, 0, 0;                  ¢ integer 0 ¢
constant       1, 0, 2;                  ¢ integer 2 ¢
constant       2, 0, -1;                 ¢ integer -1 ¢
idgen          0, 20;                    ¢ compsqrt ¢
procedure      0, 1, 1, 4, 1, 8, 8;
enterrange     0, 4;
idgen          1, 3;                     ¢ x ¢
idgen          2, 3;                     ¢ y ¢
idgen          3, 3;                     ¢ rp ¢
idgen          4, 3;                     ¢ ip ¢

monop          219, 0 (2,0,0);
iddecl         3, 5, (2,0,1);            ¢ real x = re z ¢
monop          220, 0, (2,0,0);
```

```
     iddecl        3, 5, (2,0,2);             ¢ real y = im z ¢
     monop         213, 0, (1,0,1);           ¢ abs x ¢
     dyop          52, 0, (1,0,1), (4,1);     ¢ x↑2 ¢
     dyop          52, 0, (1,0,2), (4,1);     ¢ y↑2 ¢
     dyop          26, 0, 5, 5;               ¢ x↑2 + y↑2 ¢
     monop         232, 0, 5;                 ¢ sqrt(x↑2 + y↑2) ¢
     dyop          26, 0, 5, 5;               ¢ abs x + sqrt(x↑2 + y↑2))/2 ¢
     dyop          33, 0, 5, (4,1);           ¢ (abs x + sqrt)x 2 + y 2))/2 ¢
     monop         232, 0, 5;                 ¢ sqrt ( ) ¢
     iddecl        3, 5 (2,0,3);              ¢ real rp = sqrt ( ) ¢
     dyop          40, 0, (1,0,3), (4,0);     ¢ rp = 0 ¢
     jumpfalse     5, L1;                     ¢ if rp ≠ 0 then goto L1 fi ¢
     push          0, (4,0);                  ¢ push 0 ¢
     widen         0, 0, 5;                   ¢ real (0) ¢
     jump          L2;
L1:  dyop          36, 0, (4,1), (1,0,3);     ¢ 2xrp ¢
     dyop          28, 0, (1,0,2), 5;         ¢ y/(2xrp) ¢
L2:  iddecl        0, 5, (2,0,4);             ¢ real ip = (rp=0|0|y/(2xrp)) ¢
     dyop          43, 0, (1,0,1), (4,0);     ¢ if x≥0 ... ¢
     jumpfalse     5, L3;
     dyop          29, 0, (1,0,3), (1,0,4);¢ rp ⊥ ip ¢
     jump           4;
L3:  monop         213, 0, (1,0,4);           ¢ abs ip ¢
     dyop          43, 0, (1,0,2), (4,0);     ¢ y≥0 ... ¢
     jumpfalse     L5;
     push          3, (1,0,3);                ¢ rp ¢
     jump          L6;
L5:  push          3, (1,0,3);                ¢ rp ¢
     monop         211, 0, 5;                 ¢ -rp ¢
L6:  dyop          29, 0, 5, 5;               ¢ abs ip ⊥ (   ) ¢
L4:  exitrange     0, 4;
     return        20, 0;
     endproc       0;
     call          20, (2,0,0), (4,2);
     end;
```

<u>Instruction design and addressing</u>.

The number of bits per second that can be fetched from a memory is a parameter of the engineering design of the memory. If the bit transfer rate of a memory is N bits per second, and the average instruction length is L bits, then the machine will not be able to execute instructions at a rate exceeding N/L instructions per second. If the amount of time necessary to decode and execute an instruction is much less than L/N, then the memory bandwidth is the limiting factor in the machine speed. At the present time this is the case, and furthermore, memory is usually far more expensive than the CPU, so having several memories running in parallel is an expensive way to increase the speed of the whole system.

An alternate approach is to decrease the average instruction length. This can be done not only by clever addressing techniques, but also by providing sophisticated instructions that replace two or more simpler instructions. An ALGOL 68 machine might provide special instructions, in addition to the ones described above, for handling certain commonly occurring cases. As an example we will examine an instruction add d, s1, s2 which performs d:= s1 + s2 for integers. This would be of value in an implementation of the virtual machine consisting of an optimizing compiler, which compiled virtual machine code to the machine language of an ALGOL 68 machine. The add d, s1, s2 instruction would be the result of replacing

```
    dyop    12, 0, s1, s2          ¢ s1, s2, d represent display addresses ¢
    assign 0, 5, d
```

in the virtual machine code.

A very important point about descriptors is that they all have exactly the same size. This means that it is possible to specify the number of a descriptor and have the hardware (or microprogram, of course) find the object by a simple indexing operation. An important observation is that most procedures reference variables declared within the procedure (including the parameters) and variables declared global to the entire program far more often than they reference variables declared in intermediate levels. This suggests the use of a frequency dependent addressing mechanism.

As an example of such a technique, consider an 8 bit address field, appropriate for byte oriented machines. The first 2 bits of the 8 bit field would determine the type of address as follows:

00	identifier stack, current level
01	identifier stack, outermost level
10	working stack
11	constant table (including environment inquiries)

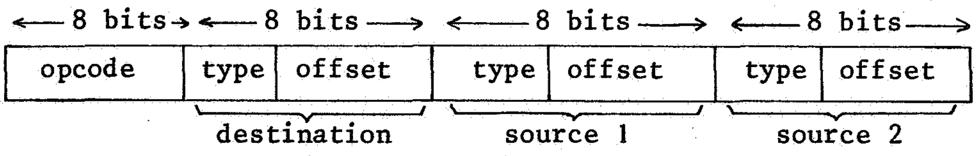
For 00 and 01, the remaining 6 bits would specify one of up to 64 descriptors at the selected level. Thus for eaxample, if a procedure had 32 parameters, all of which were structures or arrays, and 32 local variables, all of which were structures or arrays, it would still be possible to address any parameter or local variable in only 8 bits. This is in strong contrast to a more conventional design in which the variables themselves are on the identifier stack, and in which far longer addresses would be needed to address them. The saving here comes from the fact that descriptors are all of the same size, even if that size is several machine addresses, since finding the i-th descriptor is a simple operation.

For address type 10, the remaining 6 bits could be an index into the DWS, with 1 as the top item and 63 as the 63rd item from the top. Six zero bits could be reserved for something special, such as popping the top item from the stack and resetting the stack pointer(s).

For address type 11 an entry in the constant table would be referenced, with the remaining 6 bits specifying which one. Environment inquiries would be stored as constants. Constants would use the same descriptor mechanism as other objects.

In summary, an 8 bit address could access any one of 64 local variables, 64 global variables, 64 constants, or the working stack. Although this is likely to suffice for most programs, an escape convention must be provided for cases. where this is not adequate.

One possible format for instructions of the form add d, s1, s2, as discussed above would be an 8 bit opcode, and three 8 bit addresses. This would allow d:= s1 + s2 to occupy a total of 32 bits, as shown below.



It is instructive to compare this to machines designed for assembly language, rather than for ALGOL 68. On the IBM 360/370, d:= s1 + s2 would require

```
L     R1, S1        (32 bits)
A     R1, S2        (32 bits)
ST    R1, d         (32 bits)
                    ─────────
                    (96 bits total)
```

On the CDC Cyber 70 series it is even worse

```
SA1   S1        (30 bits)
SA2   S2        (30 bits)
IX6   x1 + x2   (15 bits)
SA6   d         (30 bits)
                ─────────
                (105 bits total)
```

If memory cycle time were the limiting factor in processor execution time, the ALGOL 68 machine could run three times as fast as either of the above using the same technology memory hardware, due to the fact that it was designed to run ALGOL 68 and not assembly language.

Although 32 bit instructions of the format discussed above would be widely used, other formats would be needed as well. For example, a 24 bit format consisting of an opcode and two 8 bit addresses would be useful for instructions with two operands, and whose result was left on the working stack. Furthermore, a 16 bit address could be provided, with the first two bits interpreted as above. The remaining 14 bits of type 00 could be broken up into a display level and descriptor offset perhaps 4 bits display and 10 bits offset, allowing up to 1024 variables at each of 16 levels. For even more flexibility, the hardware might provide a way to switch from 4-10 to 3-11 or 5-9 or another combination of display and offset sizes.

The 01 type address would provide for 16384 distinct global variables and the 11 type address would provide for 16384 distinct constants, surely sufficient for all but the most voracious programs. Since instructions would have 0, 1, 2, or 3 addresses and two possible address lengths would be provided (8 and 16 bits), a total of 1+2+4+8 = 15 different formats would be needed. This could be accomodated by reserving opcodes 0 to 240 for the standard format, 8 bit addresses, with the number of addresses op-code dependent. Opcodes 241-255 would signal an extended opcode, with the number 241 to 255 specifying the instruction format, and the succeeding 8 bits specifying the actual opcode. A maximum of 3 16 bit addresses could follow.

Garbage collection.

The use of descriptors makes garbage collection particularly simple. Although a nonrecursive garbage collector is possible, it is known to be slower than a straightforward recursive one. The garbage collector can either use a specially reserved area of memory for its stack, or swap part of the program to secondary memory during garbage collection. If a segmented virtual memory is available, the garbage collector's stack can occupy a special segment.

The garbage collector is designed so that a single hardware instruction, collect garbage, could be hardwired or microprogrammed. In both cases the sequence of instructions needed to carry out the garbage collection could come from a high speed read only memory instead of the slower main memory. A few hundred words of very high speed (and very expensive per word) read/write memory could be provided for the garbage collector's stack. In the event that the garbage collector encountered a tangled object that caused it to recurse more than a few hundred times, an unlikely event to say the least, the garbage collector's stack could be swapped to main memory or secondary memory if need be.

The garbage collector has three distinct phases. The first phase marks the objects on the heap still in use. The second phase compacts the heap, squeezing out the garbage. The third phase updates pointers (descriptors).

The garbage collector must be able to find all descriptors that point to the heap. Because the heap occupies a separate segment or area of the address space, an address can be checked to see if it points to the heap in one or two comparisons. Every object on the heap that can be accessed by the program must have a descriptor either on the identifier stack, or on the descriptor working stack. The garbage collector must systematically check every descriptor in every frame of the range stack, and every descriptor on the descriptor working stack.

The current display pointer allows the garbage collector to find the display. From the display entry for the current level, the origin of the descriptors in the current frame can be found. All the descriptors for the parameters and local variables are consecutively stored, terminated by a specia "empty" descriptor, marking the end. From the previous display pointer (the first item in the frame, located a fixed distance ahead of the first descriptor, whose location is known) the display for the previous frame can be found. From this display, the descriptors for that frame can be located. This process can be repeated until the entire identifier stack has been traced.

Tracing of the working stack is even easier, The origin of the DWS is a known constant, i.e. address N in segment K, and the top of the DWS is pointed to by a global pointer. The garbage collector systematically checks every descriptor between these pointers. This process is simplified by the fact that the DWS contains only descriptors, and no holes.

The garbage collector has a bit table, one bit per heap address. On a machine with metabits (e.g. parity bits) these metabits could be used. The bit table is initially set to all zeroes. Whenever a heap address is discovered to be active, the corresponding bit is set to 1. After the tracing phase is finished, all heap addresses whose bits are set to zero contain garbage to be squeezed out of the heap, and all heap addresses whose bits are set to one contain useful information that must be preserved.

The first phase of the garbage collector consists of a part responsible for the administration of finding all the descriptors. Every time it finds a descriptor it calls an internal procedure, trace descriptor, passing the descriptor as a parameter. In due course of time, trace descriptor will be called with every descriptor on the identifier stack and every descriptor on the DWS as parameter. Thus if trace descriptor correctly traces a descriptor, all active objects on the heap will be located.

Trace descriptor works as follows:

1. if the descriptor is a primitive, e.g. <u>int</u>, <u>long</u>, <u>real</u>, <u>bool</u>, <u>proc</u>, etc. and it points into the heap, mark the place pointed to. Otherwise do nothing.

2. if the descriptor is a reference pointing into the range stack, ignore it. If it is a reference pointing into the heap, mark it if need be and go process the descriptor pointed to. This need not and should not be recursive.

3. if the descriptor is a structure, mark it if need be and follow it to the table of descriptors to which it points. Obtain the count from the first field of this block and set up a loop to call trace descriptor for each descriptor in the block.

4. if the descriptor is a union, follow the pointer, pick up the descriptor and process it. Mark the union descriptor if it is on the heap.

5. if the descriptor is a row first check to see if it is a slice. If it is, find the multiple value descriptor (the kind discussed in the ALGOL 68 report) for its ultimate parent, the multiple value of which it is a slice. The multiple value descriptor for a slice must contain a pointer to its ultimate parent, the unsliced multiple value from which it is descended. The ultimate parent is to be put on a chain of multiple values to be marked at the end of the tracing phase. Set up counters and trace all elements of the row using trace descriptor.

At the end of the tracing phase, all the multiple values on the chain of ultimate parents must be marked, but not traced. Thus if x is declared by [1:10] ref m x, and only x[10] is currently active, the heap objects originating from x[1] to x[9] must not be traced and marked.

If at any point in the tracing, trace descriptor is passed a descriptor that has already been marked, it does not have to trace it again. It simply returns, indicating tracing of the descriptor is complete.

Compacting consists of squeezing out the unused garbage from the heap. Although an algorithm exists for performing the compacting without using any extra storage, it is slow and therefore not recommended on any machine where it can be avoided. The compacting begins at one end of the heap and works toward the other end. Whenever a block of useful information is encountered, it is moved towards the end where the compacting began, and an entry is made in a table indicating the former and new addresses of the block. This process is repeated until the entire heap has been compacted.

The third phase consists of tracing and marking, the same as the first phase, but in addition whenever a heap pointer is found, the old address is looked up in the table of old and new heap addresses, and the old address is replaced by the new address. An associative memory would be helpful here.

The control blocks used for parallel processing must also be traced and marked. These will be described later.

It should be pointed out that garbage collection an virtual memory computers, especially computers with a very large virtual address space, is somewhat different than on a computer without virtual memory. In the case of MULTICS, for example, a program that acquired one word of heap storage every 10 microseconds would have to run for 24 hours before it filled up the virtual address space. Thus few programs would actually require garbage collection.

However, as the heap becomes very spread out there will be many page faults, and thrashing may set in, so garbage collection may still be desirable. Because space does not suddenly run out, it may be difficult to determine when garbage collection should be intiated. Carefully monitoring the history of page faults, and initiating garbage collection when performance begins to degrade appreciably may be one approach. Unfortunately performance may degrade for reasons unrelated to heap usage, e.g. a rapidly changing working set, so this is tricky. This whole area is not well understood at present.

It is most important to keep a proper perspective on garbage collection and virtual memory. Garbage collection was invented as a method of running programs with large storage requirements on small machines. If at some point in the future salt crystals provide 10^{23} bit memories, garbage collection may crease to be an interesting subject. Virtual memory is a step in the direction of getting rid of garbage collection altogether, by providing a larger address space. One should be careful about not coming to regard garbage collection as a desirable end in itself.

Virtual memory does simplify the garbage collection process in some ways, nevertheless. In particular, garbage collection can often be postponed more or less indefinitely, until it is convenient. With a very large virtual memory it is possible to make and justify assumptions like "garbage collection will never never occur while actual parameters are being elaborated" by simply postponing garbage collection until all actual parameters of pending calls have been elaborated.

The three phase garbage collector described above can be used with virtual memory, of course, but the tracing may cause large numbers of page faults if the program is very deep into recursion and there are many frames on the range stack. Another possible strategy is to mark entire pages as used or not, rather than individual addresses. This produces a list of free virtual pages. Instead of compacting and updating, the garbage collection is terminated. Subsequent heap generators use the list of free virtual pages. This leads to a heap scattered over a wide range in virtual addresses, and a more complicated mechanism for heap generators, but it may greatly speed up garbage collection. Another virtue of this method is that a bit table with 1 bit per word for 10^{10} words will require a large, probably paged, bit table, whereas 1 bit per page helps by about 3 orders of magnitude. More experience with ALGOL 68 on virtual memory systems will be needed before this subject can be statisfactorily resolved.

Procedure call mechanism.

When a procedure is called (or a deproceduring takes place or an operator is invoked) a new frame is set up both on the range stack and the working stack (both parts). Part of this set up is done by the calling procedure (caller) and part by the called procedure (callee).

When a procedure p1 calls another procedure p2, either p2 is visible at the point of call, i.e. p2 is directly accessible to the point of call according to the ALGOL 68 rules scopes, or it is not. In the latter case, p2 must either be accessible via a parameter or global variable.

First consider case 1. Either p1 is declared inside p2 or p2 is declared inside p1. As an example of the former consider the following skeleton:

```
proc  x = ( ) m :
begin proc p2 = ( ) m2 :
      begin proc p1 = ( ) m1 :
            begin skip;
                  p2  ¢ point of call. static depth = 3¢
            end; skip
      end; skip
end
```

As an example of p2 declared inside p1, consider the following skeleton:

```
proc  x = ( ) m :
begin proc p1 = ( ) m1 :
      begin proc p2 = ( ) m2 :
            begin skip;
            end;
            p2  ¢ point of call. static depth = 2¢
      end; skip
end
```

The static depths before and after the call of p2 in both these examples are:

	p1 declared inside p2	p2 declared inside p1
static depth before call	3	2
static depth after call	2	3

A procedure call may increase or decrease the static depth, and hence the size of the display. However each routine knows how large its own display is, and therefore how many display entries to copy. To make a copy, it needs the old display pointer, which is available. The display entry for the new level must point to the descriptor for the first parameter, or local if there are no parameters.

Now consider the case where p2 is not visible at the point of call, for example:

```
begin proc p1 = (proc void p2) void :
      begin  skip;
             p2  ¢ point of call  ¢
      end;

      proc p3 = void :
      begin  skip
      end;
      p1 (p3).
end
```

In this case p2 is a formal parameter. The effect of the call p2 is to activate p3 from inside p1, although a direct call to p3 there would be prohibited by the scope rules of ALGOL 68. Calls of this type are implemented by using the descriptor for p2 to find the code and display pointers. Once these have been found the procedure p3 can be started.

When a procedure is called, the following steps must be carried out by the caller and callee.

caller: 1. push a descriptor for the result onto the DWS

2. reserve space on the OWS for the static part of the result (pointed to by the DWS)

3. save all the stack pointers on the range stack
4. save the return address on the range stack
5. push descriptors for actual parameters onto range stack
6. set SP to point to descriptor for last actual parameter
7. jump to called procedure

callee: 1. set SP to point to last word in range table
2. set up identifier stack descriptors
3. put "empty" descriptor after identifier stack
4. set display pointer to point to place where display will go
5. copy display into place
6. update DWS and OWS base and top pointers
7. copy parameters if need be
8. enter first range, if any

When a procedure is left, the following steps must be carried out by the callee (the running procedure) and the caller (the procedure returned to).

callee: 1. reset working stack pointers
2. reset display pointer
3. reset local generator stack pointer
4. return

caller: 1. move part of result, if need be.

Range entry and exit.

The storage allocation mechanism described so far allocates space for descriptors when a procedure is entered, not when a range is entered. Space for the objects themselves is allocated and released when ranges are entered or left, however. The advantage of this method is to reduce the size of the display and minimize the amount of time needed to enter a range, at the price of a small increase is storage during outer ranges. Since space for multiple and structured values is released on a range basis, rather than a procedure basis, the total amount of extra storage used by this method will be comparatively small, except for procedures with very large numbers of variables.

It should be noted that in programs in which all procedures are declared in the outermost range, a very common occurrence, the display will be of length 2, independent of how complicated the range structure is. Thus this method will save time in copying displays, by reducing the size of the displays.

When a range is entered, the following steps are carried out:

```
range entry: 1. SP is saved in the range table
             2. space is claimed for the objects in the range to the
                extent that is known at range entry time

range exit : 1. SP is restored from the range table, releasing the space
                used in the range.
```

The ALGOL 68 scope rules cause certain problems with implementing storage management for slices. These problems are closely related to the claiming and releasing of storage when a range is entered or left. One manifestation of the problem is the case of a slice of a multiple value created in a subrange of the range of the multiple value itself. The scope of the slice is the scope of the multiple value itself, and therefore the slice must not disappear until the range containing the multiple value has been left. However, no space will have been reserved for the multiple value descriptor in the range of the multiple value. As an example of this problem, consider the following program:

```
begin ref [ ] int xx;
      [1:4] int x;  flex [1:3] int y;
      begin  read(n);  [1:n] int z;
             xx:= if random < .5 then y else x[1:2] fi
      end;
      L: ¢ at this point xx may refer to x[1:2] ¢
end
```

The question is: where does the multiple value descriptor for x[1:2] go? It must be in existence at L, but when the inner range was entered it was not known whether it was needed or not. One possibility would be to reserve space for it in the outer range, whether needed or not. However, this is not sufficient. At the time of this writing it is not known whether flexibility will remain in ALGOL 68 or not, and if so, in what form. If it does remain in, then the action to be taken depends upon the source in the assignment to xx. If the source is x[1:2], a multiple value descriptor must be created in the place reserved for it in the outer range, and the descriptor for xx on the identifier stack must be set to point (indirectly) to it. If, on the other hand, the source is y, only copy of its multiple value descriptor may exist, and it can not be copied to the place reserved. If more than one copy of the multiple value descriptor of a flexible array existed, trouble would arise when only one version of it were changed. A run time test is needed to see which case prevails.

A completely different strategy is simply to put the unwanted multiple value descriptor on the heap, with the proper scope of course. If flexibility vanishes from the language, this problem will be simplified, in any case. Rowed coercends present a similar problem, and can be handled by similar methods.

Jumping out of a procedure.

There are 2 cases to be distinguished.

Case 1. The jump is to an explicit label in the environment. The compiler knows the machine address of the label, the procedure number, and the range number in which the label is contained. The jump is performed by a call to an interpretive routine internal to the run time system. It's parameters are the procedure of the procedure containing the label, and the range # containing the label.

The jump interpreter then begins searching backwards on the stack, following the chain of pointers, until it finds a proc with procedure nr equal to its first parameter. The pointers are reset, restoring the range stack and local generator stack to the right positions. The working stack must also be reset if needed. The working stack will always be empty at a label.

Case 2. The jump occurs as a result of deproceduring one of the formal parameters, or as a result of elaborating a jump contained in a procedure whose execution was initiated, directly or indirectly, by deproceduring a formal parameter or by using one as the primary of a call. The simple method of case 1 fails in the case of recursive procedures. As an example, consider

```
begin  proc  p = void:
     begin proc x = (int n, proc void label) void:
          begin  int  depth = n;
                 if n-:= 1 <  0 then label
                              else if random < .5 then x(n,1)
                                                   else x(n,label)
                 fi;          fi
          1:  print (depth)
          end;
          x (int k:= 10, proc void (goto 11));
     11:  print ("monkeys")
     end;
     p
end;
```

If the random numbers are:

.35, .15, .45, .15, .55, .95, .25, .65, .55, .85, .55

it should print: 4 monkeys.

The case of deproceduring a formal parameter which is a jump can be handled by a traceback routine that uses not only the proc nr, but the display pointer of the procedure as well. In fact only the display pointer is needed if a different routine handles this case than the routine that handles case 1. Frames are removed from the stack until the frame containing the procedures display pointer is reached. The descriptor for the formal parameter points to a block containing the display pointer.

Jumping out of an actual parameter may cause difficulties unless one is careful, since possibly a new frame is only partially constructed. The traceback routines must be able to deal with this situation e.g.

```
begin  proc p = (int i,j,k) void: print ((i,j,k));
       p (3, if random < .5 then goto 1 else 2, 1); exit;
       1:  print ("it jumped")
end
```

Jumping out of a range or display is similar to jumping out of a procedure in the sense that pointers must be restored e.g.

```
begin  [1:5] int i;  int  n:= 10;
       1: if n-:= 1 <  0 then goto stop else  print ("x") fi;
       i:= (0,1,2,3, if random < .2 then 4 else goto 1 fi)
end
```

Dynamic scope checking.

Each descriptor contains a scope field. The scope is the scope of the object described, i.e. the range in which it was declared. Because only one display entry is used per procedure, the display cannot be used for scope information. In fact, even if every range were given its own display entry, it would not work, as shown by this example:

```
int  n:= 0;
proc  p = (ref ref int i) void:
begin  int j;
       proc  p1 = (ref ref int k, ref int m) void:
             (k:= m    ¢ scope check here ¢);

       proc  p2 = void: (ref int ii; n+:= 1; p(ii));

       if n = 0 then p2 else p1 (i,j) fi
end
```

In the above, the display of p1 obvious does not provide information about the scopes of k and m, since both are accessed via the current level, yet the assignation is clearly wrong.

The actual numerical value of the scope is unimportant. All that matters is that an ordering of all scopes is maintained. One method is to use the range tables as a basis for assigning scopes. Both the address and contents preserve the ordering, but the address is known at procedure entry time, whereas the contents is only known at range entry time.

Another method is to have a global range counter, initially 1. Whenever a range is entered, this counter is incremented by 1. Whenever a range is exited, this counter is used for all local scopes. Global scopes are all 1. This method is similar to the first one, except that smaller numbers will be needed. This allows the number of bits in the descriptor to be smaller.

In both methods, a larger number means a less global scope, in other words if the scope of i is less than the scope of j, that means that i was declared first. The condition that an assignment is valid is

scope of destination ≥ scope of source

This quarantees that the source will not be unstacked before the destination. At worst they will be unstacked together.

The scope of a ref m is the range in which it is created. The scope of a plain value is the program. The scope of a structure can only be determined by examining the scopes of its fields. The scope of the structure is the smallest scope of any of its fields, i.e. the largest numerical value. The scope in the struct descriptor can be invalidated by changing any of the fields. The hardware could help here by automatically setting the scope field of the struct descriptor to 0 whenever a field was changed. This might save some checking. The hardware could also catch changes in the structure that made its scope numerically larger (less global), but not vice versa, at least not easily. Similar remarks apply to multiple values, where the scope of the multiple value is the scope of the element with the largest numerical scope.

Parallel processing.

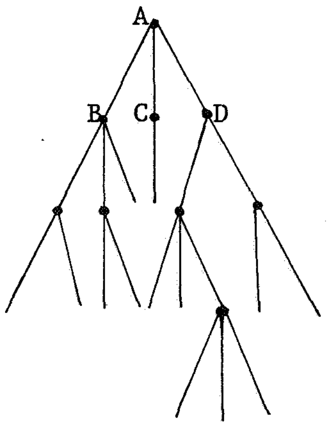
This discussion primarily describes how to implement the ALGOL 68 par feature on a machine with a small address space and a single processor. The generalization to true parallel processing follows from it, however.

A constituent unit of a parallel clause has some environment, consisting of all the ranges in which it is nested. To run the unit, the environment must be made available.

example:

```
          ( int i,j;
               par ( par (( int k,l; x: skip), (int m,n; y: skip)))
```

At point x, the environment consists of i, j, k, and l, whereas at y it consists of i, j, m, and n. In general the stack will take the form of a tree where each node represents the (dynamic) execution of a par symbol, and the branches emanating from each mode correspond to the constituent void clauses. Each of these units may in turn also split into several parts. A piece of code along with all the state and environmental information needed to execute it is called a process.



The problems introduced by parallel processing are of two sorts:

1. The envrionment of a process must be stored upon occasion and reconstructed later.

2. A process may stop running, either as a result of a semaphore operation or as a result of finishing its work. It is therefore necessary to have a systematic way to find other processes to run.

When a procedure that contains 1 or more par symbols is entered, a descriptor is put on the idstk for each par symbol, as though it were an object. The address of each such descriptor points to the local generator stack for that procedure where a control block is kept. The control block contains information for the parallel clause as a whole, as well as a structured entry for each of the constituent strong void units contained in the parallel clause.

The global information is:

1. The number of constituent void units in the parallel clause.

2. A pointer to the control block of the father of this control block. In the above figure A, B, C, D are all control blocks. A has 3 constituent units, B, C, and D. The first constituent unit of A later also splits, yielding another control block on the stack. The father of this control block is the control block for A.

3. An index into the control block pointed to by item 2. i.e. if this control block was spawned by the $i\underline{th}$ constituent unit in the <u>par</u> clause pointed to by 2, the index is i.

From any control block it is possible to retrace the path back to the main program. The father of a <u>par</u> clause not nested within another <u>par</u> clause is 0 to indicate that it is a root. Since the program may contain many such independent trees, the data structure needed is a forest of stacks (i.e. ≥ 0 trees).

The entry for each constituent void clause contains the following information:

1. status
   1 = finished running
   2 = blocked by a semaphore operation
   3 = not yet started
   4 = ready to run (it was in state 2, but the semaphore was upped)
   5 = forked. During the elaboration of the clause, another parallel clause was encountered, so the status of this one depends on the status of its children.

Statii 3 and 4 may be combined, but in a particular implementation it may be desirable to make a distinction between them for purposes of scheduling.

2. link field.
   This holds a pointer to a control block and an index into it, i.e. a unique process is specified. Associated with each semaphore is a linked list of all the processes blocked on it. The link field is used to thread them together. In this way no heap space is required for the lists.

3. restart address.
   If it is decided to restart this process, the address where to restart it must be available. That address is kept here.

4. stack hiding place.
   While the process is not being run, its private stacks (both range and working) are kept on the heap. By private stack is meant those frames that came into existence as a result of some action by the process. A process that did nothing would have no private stack, even though its environment contained a large stack. This field contains the heap address of the private stack. The range stack is stored first, then working stack.

5. Size of private range stack.
   Used when process is to be run, i.e. how much stack must be loaded.

6. Size of private working stack.
   See 5.

7. Pointer to control block (if forked).
   If the process itself encounters a parallel clause, the elaboration of the clause ceases, and the elaboration of one of its sons begins. This field points to the control block for the sons, so 1 of them can be chosen. Although the clause itself may contain many parallel

clauses, only 1 of them, at most, can be active at any instant, so 1 field is enough.

8. Origin of private stack.
   While the process is running, the origin of its private stack must be remembered somewhere, namely here. Note then this field is only needed while the process is running (including its sons). Field 4 is only needed while the process is not running. A clever implementer might use the same field in the control block.

Each processor keeps an internal record of which process it is processing i.e. a pointer to its control block and the index within the block.

The ALGOL 68 representation of an n clause control block is:

```
mode control block =
     struct (int count,
             ref control block father,
             int index of father,
             [1:n] struct (int status,
                           ref control block link field,
                           int link index,
                           int restart address,
                           int heap address,
                           int range stack size,
                           int working stack size,
                           ref control block sons
                           int stack origin
                          )
            );
```

Note that this is a doubly linked tree. From a given control block one can find both the ancestors and the descendants.

There are 5 semantic actions associated with parallel processes:

1. Forking (beginning elaboration of a parallel clause).
2. Halting a process because it is finished.
3. Resuming a process.
4. Performing an up on a semaphore.
5. Performing a down on a semaphore.

These will now be discussed in turn.

Forking.

The control block for the fork is initialized at the time the procedure containing it is entered. A process is chosen from the control block for the new parallel clause, and it is started. The status of its father is set to 3:

An algortihm for choosing a process from a control block.

The entries in the control block are examined in turn. The first one whose status is 3 or 4 is selected and run. If none of the processes have status 3 or 4, a search is made for a process with status 5 (forked). If ne is found,

1. The stacks (range + working) needed are brought in from the heap and put in place after the present stacks. The stack origin field is updated.
2. The choose-a-process algorithm is called using the control block pointed to by the forked entry. (i.e. one of the control blocks just brought in is used)

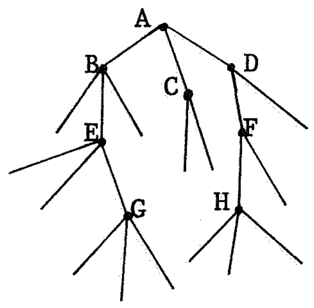
If none of the processes in the original control block have status 3, 4, or 5, the father block will be examined. This implies that the stack associated with the current process (which among other things contains the current control block) will have to be moved to the heap and the entries updated (see halting a process).

After the stack is removed to the heap, the father is examined for a candidate using the choose-a-process algorithm.

If it is noticed while searching a control block that all the processes have status 1, the entry in the father's control block that points to it (and which has status 5) should be changed to status 1.

End of choose-a-process algorithm.

Several points concerning this algortihm deserve mention. First, the copying of stacks needn't actually be done immediately. Consider fig. 6. The control block for B is being examined for a process to run. Imagine that C is the only unblocked process. The algorithm will first bring in E's and later G's stack. Then it will remove them again. This moving can be eliminated by keeping track of needed moves but not doing them until a ready process has been found.



Second, the algorithm is a tree traverser. One might think at first that a stack or mark bits in each node are needed, but this is not so because the tree is doubly linked. If the searching algorithm begins always at process 1 and works consecutively to process N, no stack or mark bits are needed. Consider what happens in the above figure when it is discovered that entry 2 of B is a fork. The path is followed and eventually nothing is found. B is retried when the algorithm finds nothing interesting in E. Since the pointer in E says "entry 2 in B is my father", the algorithm continues searching with entry 3 in B. If the pointer from E to B merely said "B is my father" then a stack or counter would be needed to remember where to continue from in B.

Restarting a process.

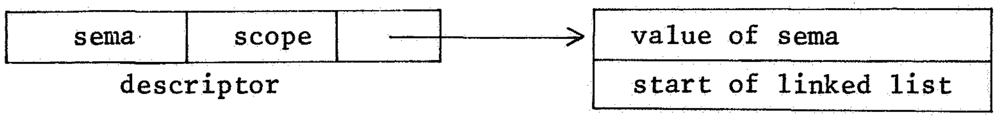
First a process to restart is chosen. This implies that its stack will have already been brought in from the heap. The processor remembers the control block address and index for the process. The process begins executing at the restart address. Just before it begins interpreting the process, the processor performs an up on the processor scheduling semaphore to let another processor begin scheduling.

Halting a process (due to its finishing).

To halt a process, the following steps are performed.

1. The stack origin is looked up in the control block
2. The stack length, N, is computed from the stack origins and SP's.
3. A block of N words is requested from the heap manager
4. The private stack of this process, N words long, is copied into the N word block provided by the heap manager
5. The heap address and stack lengths are entered into the control block. The status is made 1
6. A new process is chosen from the control block and run.

Performing operations on semaphores.



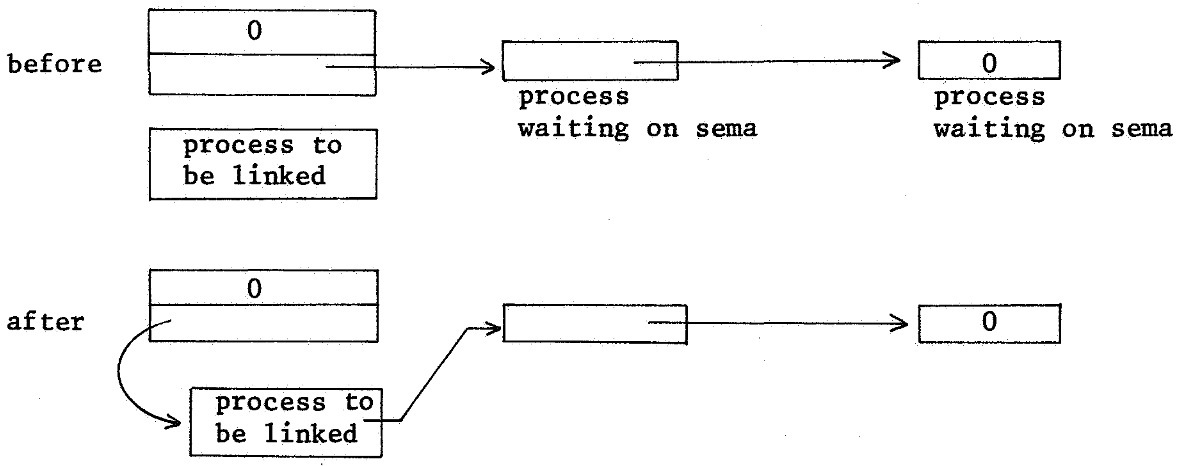
A semaphore consists of the usual descriptor and a two entry object to which it points. The first entry is the value of the semaphore. The second is either 0 (no processes are waiting on the semaphore) or the address of a control block plus an index into it. If processes are waiting on the semaphore, a linked list of all of them is maintained. The second field in the semaphore points to one process. The link field in the process points to the next, etc. The last one is indicated by a 0 in the link field.

A process can be added to a semaphore list in 2 steps:

1. The value in word 2 of the semaphore is stored in the link field of the new process.

2. The control block address and index are stored in word 2 of the semaphore.

This is illustrated below.



A process can be removed by:

1. Finding the process pointed to by the semaphore
2. Removing the link field
3. Storing the link field in the 2nd word of the semaphore.

The down operation on a semaphore is performed by:

1. Examine the semaphore. If it is 0, add the process to the waiting list and set its status to 2. Call the choose-a-process-to run procedure with the control block of the halted process as parameter.
2. If the semaphore ≠ 0, sema -:= 1.

The up operation on a semaphore is performed by:

1. Examine the semaphore. If it is 0 and there is someone waiting on it, set his status to 4. Continue running. The semaphore is not changed in value. The process set to status 4 is removed from the waiting list.
2. If the semaphore ≠ 0, sema +:= 1.

Initially there are no control blocks active and only 1 processor is running. Eventually a parallel clause is encountered. Some more processors can be started. Each processor runs until either

1. its process finishes
2. its process halts on a semaphore
3. its process forks.

In cases 1 and 2, it then chooses a process from the control block of the halted process. In case 3 it chooses a process from the control block associated with the fork.

To avoid race conditions among the processors, each of which schedules itself, there is a global semaphore used by all the processors called processor scheduling semaphore. Initially it is 1. When a processor wants to pick a process it must perform a down on that sema. When it is finished scheduling it performs an up on the sema. This insures that only 1 processor may be in the act of scheduling at a given time. The first processor does a down and sets it to 0 when the first par is encountered.

When there really are multiple processors, each private stack should occupy a separate segment. This eliminates the need for shuttling pieces of stack back and forth from the heap. Because all identifier stack references occur via the display, the fact that different parts of the it are in different segments will be transparent.