stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 103/78 DECEMBER

H.J. BOOM

CODE GENERATION IN ALGOL 68H: AN OVERVIEW

Preprint

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).

AMS(MOS) subject classification scheme (1970): 68A15

ACM-Computing Reviews-categories: 4.12, 4.22

Code generation in Algol 68H: an overview *)

by

H.J. Boom

ABSTRACT

Algol 68H uses the Janus intermediate language to generate object code for the IBM 370.

The general strategy used in the code generation process, and the mutual impact of Algol 68H and Janus on each other are described.

KEY WORDS & PHRASES: *Algol 68H, Janus, object code generation, storage allocation, intermediate code, portability.*

---

*) This report will be submitted for publication elsewhere.

1. CODE GENERATION IN ALGOL 68H: AN OVERVIEW

Algol 68H translates Algol 68 programs without formats into 370 machine code. It does so by first producing an in-core parse tree, including all identifications and coercions, then generating Janus [1] object code from this parse tree, and finally letting a separate Janus translator produce machine code. This paper describes the latter two phases, from the parse tree until executable code. To make the discussion meaningful, the general outline of the run-time system are also presented. We use the convention that the translator from Algol 68 to Janus is called a "compiler", and the translator from Janus to assembler or to machine code is merely a "translator".

The development of the compiler was heavily influenced by flaws in the support software used. It was written in Algol W [2], which provided run-time security through much of the development process, but several fixed-size tables within the Algol W compiler proved to be extremely annoying. Specifically, only 14 record classes could be declared by the programmer (a "record class" corresponds to a Pascal record type or an Algol 68 structured mode), causing error-prone mechanisms to be used to make single record classes do the job of many. Furthermore, late in the development process, a limit on the size of Algol W's block table was discovered. This forced a rewrite of a large part of the Algol 68 compiler.

One component of the present running Algol 68H system, namely the Janus translator, should be considered as scaffolding. Although it does work, it is extremely slow, and produces object code of atrocious inefficiency. It was thrown together quickly using Spitbol. This Janus translator translates from Janus to 370 assembler, which is then further translated to machine code by IBM's F assembler. Outside fixed initialization costs (which don't increase for large programs), the Janus to machine code phase takes about 81% of the total compilation CPU time. This simplest part of the translation has nothing specific to do with Algol 68, yet takes enough CPU time to render performance unacceptable. It is being replaced by a new version in Algol W, which produces 370 object modules directly. With this change, Algol 68H will reach usable compilation speed.

Parts of the Algol 68H run-time system are themselves written in Algol 68. Some unusual interfacing is performed to connect Algol 68 to its garbage collector without the garbage collector itself requiring a further garbage collector. Several unsafe language extensions are provided to enable the arbitrary interpretation of memory contents. This is used in garbage collection, moving and copying of arrays, and straightening of transput values. All such unsafe language extensions are available only when the pragmat "aleph" is specified; in that case the question mark is interpreted as a letter (the letter aleph) instead of as a special character. The keywords introducing such unsafe features all contain question marks.

The run-time system itself is based on extensive use of the heap. Even the procedure-calling stacks are ultimately placed on the heap, although activation records are grouped into "stack frames" to speed up calling. Within a stack frame, storage is allocated in a normal stack-like manner. No

scope checking is performed, if some execution safety is nonetheless required, the system can be put in a state in which every procedure activation record is individually placed on the heap.

2. THE SUPPORT HARDWARE AND SUPPORT SOFTWARE

Algol 68H was developed using two different computer systems. From 1972 to 1974 a 360/67 running under the Michigan Terminal System (MTS) was used, and from 1975 to 1978 a 370/158 under OS/VS/TSO and later MVS/TSO. The 360/67 was used on-site, but the 370 was used via a long-distance 300 baud line and the mail.

Of these various systems, MTS was easiest to use. Its editor acted directly on the file being edited; this meant that little work was lost on the rare occasions that the system crashed or communication was broken. The command language was extremely simple. It was not necessary under MTS to maintain a file of JCL recipes for commonly used operations. This was necessary under OS.

During the last year of use (1974) the MTS system became heavily overloaded, rendering response time unacceptable during parts of the day. MTS must still be commended for its high reliability despite an almost abusive load at times. During the three years of use, it could be considered unreliable only one month, when new CDC "plug-compatible" disk drives were installed. They were later removed and service became reliable again. These drives, according to rumour, occasionally failed to signal i/o completion properly. As a result, every task in the system that used that disk unit stopped execution. Sooner or later, a system task would fail and the system would slowly crash. Except for this month, only two or three crashes occurred that adversely affected my work (in over two years).

The OS/VS system in Delft was less reliable, with noticeable crashes perhaps once a month. The TSO subsystem was restricted to editing and remote job entry, which provided tolerable (but rarely good) response time. The facility slowly became worse through the years because the load increased. Toward the end of 1978, MVS and an extra CPU were installed. After that, response time was usually good, the job queue was adequately small, but crashes began to occur approximately once a week. It is not possible to pin the blame on any specific component, since the local rumour mill is not accessible to an off-site user.

As mentioned above, communication with the OS/VS system was accomplished using a 300-baud long-distance telephone line and the mail, giving two-day turn-around time for printed listings. This placed a premium on the ability to selectively examine output for error messages and other important features before having it printed. This experience has led to several ideas how compiler output should be organized. Every line containing any part of an error report should contain a distinctive string to enable it to be easily located. If an error report relates to some specific line in the program, it should contain a source coordinate, expressed using the same numbering system as that used by the rest of the operating system. This is necessary even for messages interspersed in the program listing. Algol W,

for counterexample, numbers source lines with a count of begins and semicolons, which is unrelated to all other means of locating lines in both MTS and TSO and therefore makes syntax errors tricky to find on old listings. All listings used were delayed at least a few days in the mail.

The programming systems used for developing Algol 68H were Algol W, assembler, and Spitbol. Spitbol was used for the scaffolding version of the Janus translator (on which entirely too much time was spent making minor improvements in the extremely bad generated assembler code). Assembler was used for certain small run-time routines, mostly those concerned with the operating-system interface. Algol W was used for the compiler proper.

Algol W was chosen for several reasons. The author was familiar with its behaviour on an earlier program of about 2000 lines, and did not expect any serious changes in its qualitative behaviour when a really large program was to be attempted. It had a separate compilation facility and a Fortran-subroutine interface which could be used as an escape hatch if trouble were to show up. Algol W was type-secure, had a garbage collector, and did proper checking on pointer compatibility and use of null pointers. Its data structure facilities were convenient for expressing mode tables, symbol tables, parse trees, etc., of arbitrary size (but, alas, not of arbitrary complexity because of the restriction to 14 user-defined record classes). Finally, the semantics of Algol W were such that mechanical translation from Algol W to either Algol 68 itself or to Algol 60 would be possible without unreasonable circumlocution. These translations could be efficient and preserve program structures, and so they could be used if Algol 68H were ever to be transported to another computer.

There were two serious problems encountered with Algol W during the writing of Algol 68H. The first was that run-time null pointer checking was not secure on the 370, because it relied on interrupts for invalid addresses, and the second, far more serious, was that Algol W had a fixed-size BEGIN table.

Minor changes have also been made in ALGOLX, the run-time monitor for Algol W, to eliminate annoying default limits on CPU time and printed output, and to change ddnames to prevent ddname conflict if the various parts of Algol 68H were ever to be run within one jobstep.

3. THE BEGIN SHORTAGE

A peculiar difficulty in writing Algol 68H has been the begin shortage. The only available implementation of Algol W constructs a table of all the begins in the program during its first pass. If the block associated with a begin contains declarations, their identifiers and types are chained onto the entry for the begin. The second pass uses this begin table to identify applied occurrences of identifiers (using dead reckoning to synchronize with the first pass, causing chaos after difficult syntax error recoveries). This is an adequate mechanism, except that there is an upper limit on the number of begins in the program caused by fixed storage allocation for the begin table.

The first version of the code generator translated the parse tree directly into 370 object modules, and far exceeded the permissible number of begins. Although separate compilation of Algol W procedures is possible, access to record parameters is very difficult, since they cannot be declared as parameters without first declaring their record classes, and record class declarations cannot precede a separately compiled procedure. Since records were used rather often in Algol 68H, and there were no large independent procedures, within pass 5, it was decided not to try to split off the most beginful procedures in the object code generator for separate compilation. Instead, the listing was cut in two, the semantics of the communication between the two halves was seen to be close to the level of the machine-independent intermediate language Janus, and the decision was made to use Janus as intermediate code. It is estimated that the rewrite has cost perhaps two years above what might have been necessary without the rewrite; but the new version is cleaner, Janus has been tested for a second programming language, and it may lead to a truly machine-independent Algol 68 some time in the future (using a mechanical translation from Algol W to Algol 68).

Expanding the block table was briefly considered. The source code for Algol W was written in PL/360, and was quite incomprehensible. The section that seemed to deal with allocation of compile-time tables was long and full of shifts, arithmetic and unusual constants. The size of the block table seemed to be in some way connected with the machine instructions used to access it and the other variables stored after it. I decided not to try to rewrite the Algol W compiler.

(As things turned out, about a year or two after this decision I received a list of changes that could be applied to the MTS adaptation of Algol W to increase the block table by 50%. A number of these were in parts of the compiler I had had no idea were relevant. It was not clear whether the same changes would work under OS.)

Both the new code generator and Janus translator again ran into trouble with the size of the block table. This time it was less severe than the previous time, and a simple device was found for eliminating many of the begins. Compound statements with two or three statements, such as

```
BEGIN A; B; C END
BEGIN A; B END
```

could be replaced with procedure calls:

```
BLOCK(A, B, C)
BLOCK(A, B,)
```

The procedure BLOCK closely resembles Lisp's prog2 construction:

```
PROCEDURE BLOCK(
        PROCEDURE X, Y, Z);
    BEGIN
    X;
    Y;
    Z
    END BLOCK;
```

Nevertheless, the begin shortage has continued to cramp Algol 68H's style, and as a result a number of straightforward optimizations have never been installed even though sufficient information is available.

4. THE BASIC CODE GENERATION PROCESS

The semantics of Algol 68 is nondeterministic in that the precise order of actions with collateral elaboration is undefined. Any compiler for a sequential machine will determine some specific order of execution, because at run time, instructions will happen to occur in some specific order. Selecting a proper specific order can lead to greater efficiency, and Algol 68H has some simple mechanisms to take advantage of this freedom.

Elaboration of Algol 68 programs can be viewed as a tree-branching process - whenever constructs as elaborated collaterally, new branches arise for the elaboration of those constructs. This tree of actions can be mapped onto the parse tree of the program by mapping each action onto the construct it elaborates. This may be a many-to-one mapping, since a single construct may be elaborated many times. If two actions are collateral, the implementation may choose their (possibly interleaved) order of elaboration. Algol 68H does so by choosing the order in which code for their constructs are compiled. This design principle reduces the indeterminacy considerably because the subconstructs of any construct will now always be elaborated in the same order (except for explicit parallel-clauses, of course).

The elaboration of most Algol 68 constructs requires some sort of initiation, the collateral elaboration of subconstructs, and finally a termination that combines the results of the subconstructs. Each of these phases may require action. During compilation, we get an "active front" of modelled execution, which starts by initiating the root of the tree, propagates to the leaves, and eventually returns to the root. At a branch-point, propagation downwards and returning can occur independently for the branches, but returning to the construct above the branch-point does not occur until all the branches have returned.

Algol 68H further restricts indeterminacy by usually elaborating constructs from left to right, except that dereferencing, field selection, construction of constants, and copying may be delayed or advanced from the normal left-to-right elaboration order. This is done because on many computers, including the 370, multiple field selections can be combined, field-selection and dereferencing together can be combined with arithmetic in single instructions, and constants can often receive special treatment. Combining such operations reduces the amount of code that need be generated, and can achieve results similar to peephole optimization.

To enable the code generator to use a strict tree-walk while deferring and advancing some operations, two mechanisms are used. First, "advice" is generated to advise the compilation of a construct how its computed value will be used, and secondly, compile-time data structures ("mvalues") are used to indicate which operations must still be performed to obtain a value.

It is of interest that deferring operations and combining them with later operations must be done differently in the Janus to 370 machine-code translator than in the Algol 68 to Janus compiler, because Janus does not have collateral elaboration. The Janus translator must continually check for possible conflict of deferred actions with new ones.

The compiler thus contains a central routine, COMPILE, which compiles constructs into Janus. This routine is organized with a central driver which examines a construct to be compiled and then calls a specific compilation routine for the specific kind of construct involved. Such specific routines usually have names beginning with "COMPILE", such as "COMPILECOLLATERALCLAUSE", which compiles a collateral-clause.

COMPILE yields an "mvalue" as result, which is a compile-time model of a run-time value. An mvalue indicates which operations must still be generated to access the value, and also which extra operations have already been gratuitously performed on the value in the hope that these will later be required. Such extra operations will be generated only if COMPILE receives advice that they would be appreciated. The operations which can be deferred are field-selections, dereferencing, and constant evaluation. The operations which can be performed extra upon advice are conditional jumps, storage into a specific storage location, and loading onto the Janus temporaries stack. In any case, advice can be ignored, and then their value can be placed in main store, or can be a known constant. Other alternatives (jumping or stack) are permissible only if advised.

Since compilation of most constructions passes through COMPILE, it is possible to suppress or alter advice in a systematic manner by altering COMPILE. This can be used to patch bugs caused by incorrect advice. It has also occasionally been done during debugging, to reduce the complexity of some otherwise bewildering situations.

Consideration has also been given to adding a further kind of advice to indicate that a yielded value will be voided anyway, and that there is therefore no point in computing it. This was not done, because on closer examination it appeared that all the situations whose efficiency could be improved by such advice were situations which were difficult to imagine to be useful in a real program, such as voiding a denotation. It might be worthwhile to issue a warning in such cases, but that is more properly the task of coercion. Optimizing nonsense is not sensible.

The advice to place a value a specific storage location has eliminated many loads, stores, amd moves. The advice to jump conditionally depending on a boolean value eliminates explicit construction of an explicit boolean value in all conditional and while-clauses with simple conditions.

It is not always possible to pass advice from superconstructs down to subconstructs uncritically. It depends on the precise collateral semantics of a construction. For example, when one compiles a structure display with the advice that it should be placed in a specific storage location, one might think that the units of the display should be compiled using the advice that their values should be placed in the fields of that storage location. The following counterexample shows that this will not work:

```
compl z;
...
z:= (im of z, re of z)
```

If im of z were to be compiled with the advice that its value should be placed in re of z (the first field of the destination), it would demolish the re of z which was still to be computed. Using such advice is actually performing part of the assignment while evaluating im of z. But the assignment must be performed after the fields have been determined; the assignment of the first field cannot be performed collaterally with the evaluation of the second. To avoid trouble, COMPILESTRUCTUREDISPLAY creates and uses its own private structured temporary and ignores advice. Its advice to the field compilations is to place their results into the fields of the temporary.

5. RANGES AND DECLARATIONS

Algol 68H recognizes six kinds of declarations:

identity declarations, operator declarations
variable declarations
mode declarations
priority declarations
parameter declarations
definition module declarations
dummy declarations.

The code generator ignores priority declarations, and treats identity and operator declarations alike. Each declaration (except priority) may have associated with it an mvalue called its "place" that describes the the value being declared. The value associated with a mode declaration is its bounds elaborator, and that with a definition module is the procedure to be called to invoke the definition module. These mvalues are associated with their declarations at range entry, and the values they describe are overwritten when the declarations are elaborated. It is thus possible to have use before definition, and the value will be undefined only if the declaration has not been elaborated when the applied occurrence is elaborated.

The Report does not mention any action for elaborating a mode declaration, but instead merely specifies what is to be done when an applied occurrence is elaborated. This may well be before elaboration of the mode declaration takes place in normal order. To prevent difficulty, Algol 68H constructs the various bounds elaborators at range entry. The complete range entry sequence, insofar as it relates to declarations, is thus:

- Allocate storage for the places of the declarations, and associate the mvalues describing the places with the declarations (the DPLACE field of a declaration at compile time).
- Compile the bounds elaborators for the various mode declarations in the range, and generate code to place the procedure values in the proper places. This can only be done after places have been allocated for all indicators, because of possible forward references made by actual bounds.
- Begin normal elaboration of the range contents.

When an elaboratable declaration is compiled (other than a mode or priority declaration), code is generated to overwrite the place of the declaration with the value to be ascribed.

Upon range exit, nothing special is done to erase ascribed values. This may result in some storage being unnecessarily retained by the garbage collector. Since the Janus translator takes care never to reuse storage within an activation record, no incorrect execution can occur. (It is an Algol 68H defect that it relies on such a bizarre property of a specific Janus translator. A better solution would be to perform more careful bookkeeping when constructing the garbage collector's templates.)

Because the place of a declaration is actually used as a storage location (it is read to obtain a value and even overwritten during ascription), it might have been wiser to use an MSTORE for it instead of an MVALUE. This would have been conceptually cleaner, and an mvalue could have been made from the mstore whenever the value was required. The present scheme was designed to make constant propagation easier (values of constants reside in mvalues); however, none is performed anyway because of lack of begins.

6. GENERATORS AND ACTUAL DECLARERS

A generator is used to allocate storage for variables. It contains an actual declarer, which may refer to other actual declarers elsewhere via mode declarations. The actual declarers determine the mode and array bounds of the variable, and the generator then causes storage allocation. The object code for an actual declarer consists of code that elaborates the bounds and stores them in a standard format. The object code for a generator consists of the code for its actual declarer, code that allocates the static part of the variable, and code that uses the bounds to "fulfil" the variable by allocating the necessary array elements and filling in the array descriptors. If an actual declarer is the actual parameter of a mode definition, it is compiled within a special routine. This routine is called from the code for an applied occurrence of the mode indicator. The "value" of a mode declaration is thus a bounds elaboration procedure.

The actual declarer compilation routine is recursive on the syntax of the declarer; the fulfil routine is recursive on the mode of the object it must construct.

If a mode has no dynamic part, no bounds elaborator is ever generated or called.

7. DIFFICULTIES WITH JANUS TEMPORARIES

Janus temporaries are intended for short term storage of values which will only be accessed explicitly. Access to such values can be optimized by placing them in registers, rearranging computations, avoiding explicit store instructions, etc. There are two kinds of Janus temporaries, the anonymous temporaries and the named temporaries. Anonymous temporaries function as a stack, and are therefore ideally suited for the evaluation of arithmetic expressions. The anonymous temporaries are used as implicit operands in many Janus instructions, thereby giving Janus a conveniently uniform syntax (instructions accept zero or one explicit operands). Named temporaries are used in other situations. After being declared, a named temporary receives a value from a STORE instruction, can be used any number of times thereafter, and loses its value when it is used after being named in a RELEASE instruction. All Janus temporaries, whether named or unnamed, lose their values at a label or upon successful execution of a jump. This last restriction ensures that temporaries can be processed with straightforward basic block optimization techniques and that the set of active temporaries at any point can be determined without flow-of-control analysis.

These conventions work well for Pascal, the first language implemented using Janus, but cause difficulties for Algol 68, as well as for other languages in the Algol family (such as Algol 60, Algol W, and SIMULA). Algol 68 has two features which make the use of temporaries difficult: conditional expressions, and garbage collection.

The trouble with a conditional expression is that it may require a jump while an arithmetic expression is being evaluated. This will void the temporaries stack, and so elaborate precautions must be taken to save it or to ensure by other means that it happens to be unused at the jump. This is no problem in Pascal, which has no conditional expressions. The same difficulty arises in Algol 68 for any construction whose implementation requires a jump, because such a construction can be nested within a serial clause within an expression.

The second problem is garbage collection. At any operation involving dynamic storage allocation, the garbage collector may be invoked. Guided by templates, it must be able to find all pointers for preservation and updating. If any pointers reside in temporaries, they must be updated too. The locations of named and unnamed temporaries are, however, unknown. Once again, we get a severe restriction on the use of temporaries.

Algol 68H has no central administration for its values, temporary or other. Instead, its model of the run-time machine is spread throughout the compile-time data structure. Mvalues are attached to declarations, are the values of compile-time variables, and are even yielded as the values of compile-time procedures. It is not possible to scan the set of temporaries in its entirety in order to store them at strategic moments. This posed no difficulties in an early version of the code generator which translated

directly to machine language, but does not work well with Janus. Algol 68 temporaries are kept in permanent variables, and not in Janus temporaries. This has adverse effects on object-time efficiency. The Janus translator cannot optimize permanent variables to any great extent, because the access restrictions on temporaries do not apply to permanent variables (they may, for example, be read or overwritten via indirect addresses).

At a number of places, Janus takes special precautions to avoid difficulties with temporaries. For example, there is a special TRAP instruction for performing run-time checks. Without this instruction, a conditional jump around a call to an error routine might have been necessary, which would destroy temporaries. This instruction seems to make Janus adequately preserve temporaries for Pascal. It is difficult to see how the same can adequately be done for Algol 68 without preserving temporaries at most jumps (not at all jumps! Explicit Algol 68 jumps cannot only void temporaries, bur even terminate procedures).

8. INTERNAL LANGUAGE EXTENSIONS

Several language extensions to Algol 68 and Janus have been made to enable parts of the run-time system to be written in Algol 68H. These language extensions are "internal" in the sense that they are intended for use only within the Algol 68H system. They can be used only by a programmer who can write aleph symbols. An aleph symbol can be written as a question mark if the pragmat .pr aleph .pr has occurred earlier in the program text.

To represent values of arbitrary mode, a new mode ?hyperunion has been introduced. This mode is a kind of a union of all modes, and is implemented as a pair of pointers, one to a mode descriptor and one to the value itself. The mode descriptors used here are the same as those used by the garbage collector.

To enable storage to be interpreted as if it contained a value of arbitrary data type, a "hyping" operation has been added.

```
NEST MODE1 hype: virtual MODE1 declarer, ?ex symbol,
        NEST MODE2 cast.
```

The value of the MODE2 cast is simply interpreted as if it were of mode MODE1 instead. A cast is required to eliminate any ambiguity as to the a priori mode. The ?ex symbol is written as ".?ex".

There is an extra mode ?addr. This represents a machine address of unknown mode, is updated by the garbage collector if the storage it points to moves, is not followed by the collector's marking phase, and is changed to nil if the storage pointed at is freed. Great care is taken not ever to refer to a value of some mode of the form ref ref amode as if it were of mode ref ?addr. If the garbage collector were to see it as of mode ref ?addr first, it would mark and not follow the ?addr. The value of mode amode might then be lost.

To enable copying and allocation of values of an arbitrary statically unknown mode, two operations have been added, BLOT and ALLOC. BLOT takes three arguments, a destination address, a source address, and a length. It copies. ALLOC receives a "size" (as a number of bytes), and the address of an "allocator" for some mode (see the description of the garbage collector). It allocates that much storage for values of the mode indicated by the allocator, using the storage allocation system currently in effect.

Finally, there are the kludge declarations. A kludge declaration is a declaration whose actual parameter is a "kludge" instead of a normal unit.

kludge: ?kludge symbol, string denotation.

The ?kludge symbol is written ".?kludge". The string is interpreted by the code generator to determine the code to be generated. Some kludges of particular importance are listed below:

.?kludge " opcode"	The operation is directly implemented by the Janus operator "opcode".
.?kludge "x extsym"	The procedure has been written and compiled separately, probably in another language.

9. BASIC RUN-TIME SYSTEM

The "basic run-time system" is the run-time system excluding normal preludes. It does not include transput, sines, cosines, and such.

Instead, the basic run-time system consists of the routines and conventions for representing data in memory, calling procedures managing main storage, and processing arbitrary objects. Some of the routines are written in Algol 68; all of them have the property that the compiler must treat either them or their calls in a special manner.

Copying and assigning arrays requires fairly complex processing. It is necessary to have a number of nested loops for the various dimensions. Array elements can themselves be arrays, which must be similarly processed. Flexible arrays may require reallocation of storage. If the source and destination overlap, extra processing must be performed to prevent wipeout. (This overlap problem does not exist with values which do not contain arrays - other values are either identical or disjoint!) Furthermore, it may be convenient to implement initialized declarations as if they were assignments, and in this case, the destination must acquire its bounds from the source.

To handle all these cases by in-line code would require ridiculous complexity. The decision was made to centralize copying of all values involving array components in a single central routine. This routine is not called for values with no dynamic parts. The central move routine accepts a destination and source of arbitrary data type, and two Booleans to indicate whether the destination is being initialized and whether it may be flexed.

10. STORAGE ALLOCATION

Storage must be allocated when a procedure is called, when a generator is elaborated, when an array is initially or flexibly assigned, or when ALLOC is explicitly invoked. All of these cases except for the procedure call involve an explicit Janus ALLOC operation. An ALLOC operation is itself treated as a peculiar procedure call. An allocation routine is obtained (more about this later), and called using a separate stack for the call.

When a normal procedure is called, control may also reach the allocation routine, but by a roundabout scheme. This scheme is built into the special procedure-calling semantics of the Algol 68H Janus translator. The called procedure first checks whether sufficient storage remains on the stack for its activation record. If so, all is well; if not, it return jumps to an interrupt routine whose function is to extend the stack or allocate a new one and then return just as if all was already OK and it need never have been called. (The 370 Janus implements this as L 15,=V(ST@KOFLO); BALR 14,15.) The interrupt routine obtains the separate ALLOC stack, arranges things so that an operating system dump will give consistent information should it ever occur, and calls the allocation routine.

Since the garbage collector and storage allocator are themselves written in Algol 68, some other storage manager must be around when they are being executed. This storage allocator must be able to reclaim all necessary storage without garbage collection and without explicit freeing (Algol 68 has no feature for explicit freeing, and we would prefer not to add one) except for ordinary stack processing. To achieve this, the garbage collector is written so as not to allocate any storage for its own use during collection except for the activation record stack. Any variables it needs are permanently allocated when the garbage collector is initialized. Since the garbage collector is called only when the heap is exhausted, storage allocation during collection is not practical anyway. The work area for the garbage collector is of bounded size, and is allocated when the heap is constructed. The garbage collector does not contain any recursive routines, since these could require an amount of stack storage which is not determinable in advance.

The garbage collector examines mode descriptors (also called "templates") to interpret storage properly. A mode descriptor is of mode .mdesc, which is described in Appendix 3. There is no need to make the mode .mdesc unwriteable by the normal programmer, since he cannot gain access to any mode descriptors anyway. Each mode descriptor (except for activation record descriptors and stack descriptors) contains an "allocator", which indicates the size and alignment of values of the given mode. Some poor implementation decisions have been made in this area. Janus has been extended with a special constant declaration for making mode descriptors, and Algol 68 implements allocators as structures with three integers. The situation should be reversed, since mode descriptors are language-dependent and allocators are machine-dependent. A new primitive mode should have been added to Janus for allocators, together with appropriate operators.

The various clever compressing garbage collectors that have appeared recently could not be used because they assume that every address points to the beginning of an allocated cell, never to the middle, and that the mode of a cell is obvious from its address. The extra overhead required to achieve these properties would make addresses unacceptably large.

The garbage collector uses a separate bit table (on the 370, one thirty-second of memory) to mark words as "white" or "black". When the marking phase starts, all memory is coloured white. The following iterative routine ensures that all memory accessible from a "root" becomes black:

```
colour the root black
while there is a black pointer in memory pointing to an area
      which is not all black
do    colour at one white word in the area black
od
```

The search for black, white-pointing pointers can be done by a linear search of memory. To speed things up, a small conventional marking stack is used, and the linear scan is started only if stack overflow has occurred, thus:

```
stackoverflowed:= false

markfrom(root);

while stackoverflowed

do stackoverflowed:= false;

   for each word W in memory
   do  if W is black and W contains a traceable pointer
       then markfrom(W)
       fi
   od
od
```

```
proc markfrom(W):
     A normal recursive marking algorithm with a fixed size stack.
     If stack overflow occurs, set 'stackoverflowed':= true,
     discard part of the stack, and go on.
```

Notice that "markfrom(W)' will fail to mark all memory accessible from 'W' only if it sets stackoverflowed:= true. The condition 'while stackoverflowed' ensures that termination occurs only when all accessible memory has been marked. The process will terminate if the stack is of a certain minimum size (the size necessary to find pointers within at least one call and examine the colour of the other cells they point to) since then the for-loop will have found an marked at least one new word in memory if stack overflow occurs. The collector will not be fast if there are many long linked lists of long linked lists, but it will remain correct, and is simple.

REFERENCES

[1] HADDON, B.K. & W.M. WAITE, The Universal Intermediate Language Janus (Draft Definition), SEG-78-3, Software Engineering Group, Dept. of Electrical Engineering, University of Colorado, 1978.

[2] SITES, R.L., Algol W reference manual, Stanford University, 1971.

Appendix 1

Overall structures and statistics.

Component and function		language used	size in lines
Algol 68 to Janus compiler			
GLOBAL	calls other passes.	Algol W	2500
PASS1	does lexical scan, builds tables of corrals and mode, operator, priority, and module declarations.	Algol W	2000
PASS2	does lexical conglomeration of operators depending on the operator and priority declarations. (no longer necessary because of standard representation and standard set of operator symbols), complete context-free parse, builds mode tables.	Algol W	3000
PASS3	mode equivalencing	Algol W	1000
PASS4	identification of all indicators, coercion.	Algol W	2700
PASS5	generates Janus object code.	Algol W	5000
Janus translator, two versions			
(1)	translates Janus to assembler. Very slow, and the IBM assembler is also very slow.	Spitbol	3500
(2)	translates Janus to object modules. Not yet debugged, probably fast.	Algol W	5500
Run-time system			
OS interface.			
	stack, memory allocation	Assembler	321
	primitive input	Assembler	500
	primitive output	Assembler	450
Definition module invocation finder		Algol 68H	34
Move/copy routine (not yet debugged)		Algol 68H	217
Garbage collector		Algol 68H	425
Transput (obtain from Algol 68 Support Committee)		Algol 68H	?
Standard prelude mathematical routines (steal from Fortran).		?	?

Appendix 2 Compile-time data structures need for modelling values.

```
RECORD MVALUE(

   REFERENCE(MODE) VMODE; COMMENT the mode of the value;

   INTEGER VLAYER;
      COMMENT When currentlayer < vlayer, we may discard
         the mvalue;

   INTEGER VREFCOUNT;
      COMMENT The compile-time reference count. When
         VREFCOUNT <= 0, we may discard this mvalue;

   LOGICAL VREADONLY,
           VINVARIANT;
      COMMENT If 'READONLY', this mvalue does not let you
                change its value.
         If 'VINVARIANT', no one else will change it
            'VINVARIANT' must be false if the mvalue is based
               upon a nonVINVARIANT mvalue;

   INTEGER VDISPLAY; COMMENT non-zero for a display element.
      If non-zero, it indicates which display element the
      mvalue represents.
      PARAM(i) is represented by 2 * 1 + 2.
      DISP(i) is represented by 2 * i + 3.
      VDISPLAY ]= 0 may as well imply 'VINVARIANT';

   LOGICAL VISCONSTANT;
      COMMENT 'VISCONSTANT' may as well imply 'VINVARIANT';

   LOGICAL VINSTORE;

   STRING(1) VJUMPING; COMMENT "T", "F", or " ";

   LOGICAL VSQUASHED;
      COMMENT If VSQUASHED, the value is a reference, and
         furthermore, VSTORE and VPERMANENT are about the
         other value referred to, not about the reference
         itself
         'VSQUASHED' implies '(VINSTORE AND (VSTORE ]= NULL))';

   LOGICAL VONSTACK;
      COMMENT 'VONSTACK' implies ']VINVARIANT' and ']VSQUASHED';

   REFERENCE(GENE, FINGER) VCONSTANT;
   INTEGER VVALUE;
      COMMENT One of these fields gives the value of a constant.
         If 'VCONSTANT' = 'NULL',
         then 'VVALUE' contains an integer or a boolean.
            Boolean values are encoded 0=false, 1=true.
```

```
        Otherwise, the value resides in 'VCONSTANT';

    REFERENCE(MSTORE, FINGER) VSTORE;
       COMMENT where the value is to be found
          If 'VJUMPING' is "T" or "F", then 'VSTORE' indicates
              the jump target ;

    REFERENCE(MVALUE) VNEXT

    );

RECORD MSTORE(

    REFERENCE(MODE) SMODE;

    INTEGER SREFCOUNT;

    REFERENCE(MVALUE) SBASE;
       COMMENT the value SBASE is either
          - in the current display, or
          - in storage based on a value in the local stack segment;

    REFERENCE(FINGER, LIST) SDISP;
       COMMENT A finger, or a list of them to be added together;

    REFERENCE(MSTORE) SMASTER;
       COMMENT If this record is represents a field of another value
          then SMASTER points to the larger mvalue
          else SMASTER is null ;

    REFERENCE(MSTORE) SNEXT);
```

Appendix 3 Mode descriptors.

```
.mode .mdesc = .struct(.allocator alloc, .mchoice mode),
      .allocator = .struct(.int size, mod, rem),
      .mchoice = .union(
         .maddr, .mhyper, .mactrec, .mrefact,
         .marray, .mbunch, .mflex, .mproc, .mref,
         .mstruct, .munion,
         .mprim),
      .maddr = .struct(.vvoid addr),
      .mhyper = .struct(.vvoid hyper),
      .mactrec = .struct(.vvoid actrec),
      .mrefact = .struct(.vvoid refact),
      .mprim = .union(.mlbits, .mbool, .mlbytes, .mchar,
         .mlint, .mlreal, .mlcompl),
      .marray = .struct(.int dim, .ref .mdesc elem),
      .mbunch = .struct(.int count, .ref .mdesc elem),
      .mflex = .struct(.ref .mdesc deflex),
      .mproc = .void #???# ,
      .mref = .struct(.ref .mdesc deref),
      .mstruct = .struct(.ref{}.fdesc fields),
         .fdesc = .struct(.ref .mdesc mode, .int disp),
      .munion = .struct(.ref{}.fdesc options),
      .mlbits = .struct(.vvoid lbits),
      .mbool = .struct(.vvoid bool),
      .mlbytes = .struct(.vvoid lbytes),
      .mchar = .struct(.vvoid char),
      .mlint = .struct(.vvoid lint),
      .mlreal = .struct(.vvoid lreal),
      .mlcompl = .struct(.vvoid lcompl),
      .vvoid = .char # an actual void declarer would be nice #;
```