ON THE IMPLEMENTATION OF ALGOL 68

B. J. MAILLOUX

ON THE IMPLEMENTATION OF ALGOL 68

AKADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN AAN DE UNIVERSITEIT VAN AMSTERDAM, OP GEZAG VAN DE RECTOR MAGNIFICUS, MR. J. VAN DER HOEVEN, HOOGLERAAR IN DE FACULTEIT DER RECHTSGELEERDHEID, IN HET OPENBAAR TE VERDEDIGEN IN DE AULA DER UNIVERSITEIT (TIJDELIJK IN DE LUTHERSE KERK, INGANG SINGEL 411, HOEK SPUI) OP WOENSDAG 12 JUNI 1968 DES NAMIDDAGS TE 4 UUR

DOOR

BARRY JAMES MAILLOUX
GEBOREN TE LEAMINGTON (CANADA)

1968

MATHEMATISCH CENTRUM - AMSTERDAM

Promotor: Prof.Dr.Ir. A. van Wijngaarden

for Isobel

Acknowledgements

I should like to thank the Mathematical Centre for providing the stimulating atmosphere in which the research dealt with in this paper has been performed.

My very special thanks to Prof.Dr.Ir. A. van Wijngaarden for many, many discussions together about ALGOL, and for his constant help and encouragement.

My further thanks to Miss L.C. Bosma and Mrs. M.J. Nuis for typing this manuscript, and to Messrs. D. Zwarst and J. Suiker for their careful reproduction work.

CONTENTS

0. Introduction

The Draft Report on the Algorithmic Language ALGOL 68 [1] (in the sequel, we shall use a notation as, for instance, DR 8.4.1 to denote Section 8.4.1 of this Draft Report) defines a new language intended to be the official successor to ALGOL 60 [2]. ALGOL 68 is designed to express algorithms in such a way that they can easily be written by, taught to and communicated amongst human beings, and yet be executable in an efficient manner on presently available or foreseeable digital computing machinery. ALGOL 68 is not merely an extended version of ALGOL 60, although many of the good features, and experience gained in the practical use of ALGOL 60 have had their influence on the design of this new language.

The Draft Report defines ALGOL 68 in terms of "actions" performed by a "hypothetical computer" [DR 2.2]. However, any real computer which attempted to perform the actions literally as described would be hopelessly slow; fortunately, such a literal interpretation is not required [DR 1.1.6.e], and one will therefore translate ALGOL-68 programs into programs in the machine language of the computer. This translation itself is most conveniently accomplished by the computer itself with the help of a special program, called a compiler.

The first chapter of this thesis deals with several important aspects of the operation of a compiler for ALGOL 68. One of the tasks of any compiler is the gathering and the systematic storing of information about the source program for use in later stages of the translation; a new algorithm, suitable for ALGOL 68 and other languages with a "block structure", is given for the efficient management of an identification table. Further, an outline is given of a four-pass compiler, the first pass of which handles the identification of indications, the second pass that of identifiers, and the third that of operators; a fourth pass is then able to produce efficient machine code.

The second chapter deals with a few features of the object program which are peculiar to ALGOL 68. Apart from the well-known stack discipline, ALGOL 68 requires a free-storage (heap) system, in which storage reservation and de-reservation do not necessarily coincide with block entry and exit; an algorithm is given by means of which nonreserved storage cells in the heap can be recognized, in order to be reallocated for new uses. Finally, the machine code corresponding to several characteristic ALGOL-68 constructions is described and explained.

1. The General Form of a Compiler

The syntactic structure of ALGOL 68 is described by means of a "two-level" production scheme. This scheme has considerable advantages over others in clarity, conciseness and descriptive power, but, since it is context-sensitive (in fact Chomsky type 0 [3,4]), it is not amenable to the usual, context-free, analysis methods [5,6,7,8].

On the other hand, ALGOL 60, FORTRAN, COBOL and most (probably all) other "high-level" programming languages are in fact context-sensitive even though the formalism, if any, used to describe them is not. The context-sensitive elements of these languages are introduced by sentences in natural language (in the ALGOL 60 report, as part of the so-called semantics). The context sensitivity of ALGOL 68 is introduced partly in the formalized syntax (see e.g. DR 8.6.1.a, and the pragmatic remarks in DR 8.6.1) and partly by means of the context conditions (DR 4.4), expressed in English.

Most compilers for ALGOL 60, etc. work by analysing (parsing) the text according to the given context-free grammar, and by building up and consulting a "symbol table" or "name list" (we will use the term "identification table", since the words "symbol", "name" and "list" have technical meanings in ALGOL 68) to handle the context dependencies.

A similar technique is applicable to ALGOL 68, provided that we first simplify its defining syntax by removing all context-sensitive constructions; that is, we create a context-free grammar producing all ALGOL 68 programs, as well as a good many other texts. Thus, for instance, DR 8.4.1 might be simplified to

a) slice: primary, sub symbol, indexer, bus symbol.

b) indexer: trimscript; indexer, trimscript.

c) trimscript: ; formary; trimmer.

d) trimmer: actual bound, up to symbol, actual bound, new lower part.

e) new lower part: ; at symbol, formary.

The compiler can then recognize a construction like *b*[*3,4:5*] as a possible candidate for a slice and consult its identification table to ascertain the "dimension", of *b*. It must then scan through the trimscripts, checking that there are not more of them than the dimension of *b*, and counting the

number of trimscripts which are not trimmers (i.e. which are subscripts); the dimension of the construction as a whole is that of *b* minus this number of subscripts.

The process described in the previous sentence is completely obviated by the syntax of DR 8.4.1 and not at all by our simplified version. It is therefore to be hoped that further research may reveal efficient means by which such processes can be performed automatically, guided only by the two-level syntax.

In the meantime, we shall proceed to a discussion of identification tables and a rough plan for a compiler.

1.1. On Identification Tables

1.1.1. An Unsophisticated Approach

In languages in which declarations occur at one level only, as in FORTRAN, the identification table can have a particularly simple form. The table is initially empty; at each defining occurrence of an identifier (or, in general, a key - that is, in ALGOL 68 terms, identifier, indication or operator) it is added to the table at the first empty place; at an applied occurrence, the entries in the table are searched, one after another, from one end of the table until the desired key is found (or to the other end of the table, in which case the desired key has not been found).

Such a table and search process are very easily programmed, but it turns out that many compilers which work in this way spend nearly half their time searching in the table; this is not surprising, since, on the average, half of the entries in the table have to be searched at each applied occurrence.

1.1.2. Hash Addressing

A variant of this scheme, known as a "hash table" [9] has been in use for some time. It is based on the fact that the key is represented in the computer as a number or sequence of numbers; this sequence is transformed, by simple arithmetic operations, into a positive integer, called the "hash" or hash total, whose expected value is fairly evenly distributed over some

range. The identification table is chosen to have a length equal to the largest possible hash value. At a defining occurrence of a key, that key and associated information (mode, address, etc.) are inserted into the table at the entry indicated by its hash, provided that entry is still empty. If the entry has already been filled by a previous key having the same hash, then some other entry, for example, the next one in the table, is tried, and so on until the key has been placed. At an applied occurrence of a key, the entry in the table indicated by its hash is consulted to see whether it contains the key; if so, then the associated information has been found; if not, then the following candidate entry is consulted, etc. The average number of entries which must be consulted to insert or retrieve an item under such a discipline is approximately $(1 - \alpha/2) / (1 - \alpha)$, where α is the fraction of table entries which are filled [10 p.41]; even for a table which is 90% full, this amounts to only 5.5 probings, considerably less than the simpler strategy of 1.1.1. Even more advantageous variations [9,10] of this scheme exist.

1.1.3. A Strategy for ALGOL 68

Unfortunately, at least as far as these schemes go, ALGOL 68 (as well as ALGOL 60, PL/I and other) programs take the form of ranges (DR 4.1.1.e) nested within one another, and defining occurrences in a given range must be "invisible" to ranges enclosing and "parallel" with the given range. Thus, the schemes of 1.1.1 and 1.1.2 cannot be applied without considerable alteration.

The system used in most ALGOL 60 compilers has been a modification of that in 1.1.1 [11]. We shall not discuss it further here, except to mention that it also requires a high average number of probings and is therefore quite time-consuming.

Batson [12] describes a hash-total method, but, apart from an error in his algorithm, it is applicable only to one-pass compilers. We shall need several passes to translate unrestricted ALGOL 68 in a reasonable fashion, and shall therefore discuss in detail a class of identification-table systems suitable for multiple-pass translation.

To do so, we shall associate an integer, called the range number, with each point in a program. We can make use of ALGOL 68 itself to help

define this quantity by supposing that the compiler is also written in ALGOL 68. Let us suppose that the declarations

```
[1:100] int current ranges;
int range number (0), max range (0), range depth (0);
```

are present in the outermost range of the compiler. Whenever (in all passes!) the scanner of the compiler "enters" a range, the statement

```
range number := current ranges [range depth plus 1] :=
  maxrange plus 1
```

is elaborated, and, whenever it "exits" from a range, the statement

```
range number := current ranges [range depth minus 1].
```

This algorithm not only calculates a range number for each point in a program, but also "remembers", in the array *current ranges*, the range numbers of all ranges within which the given point is contained.

Armed with this background, we shall discuss the identification table itself. As in 1.1.2, a hash total is calculated from the key, and the information, this time including the range number, is inserted at the position indicated by the hash, if that position is empty, or, otherwise, at the first empty position after it. Since the information to be inserted is of variable length (e.g. the number of characters in an identifier), only the fixed-length part of the information will be inserted, along with the index in a second table at which the variable part is to be found. At an applied occurrence, a search is begun, starting at the position indicated by the hash, and continuing to the first position which is empty or contains the key with a range number greater than or equal to the range number at the applied occurrence; the required entry is then the last one encountered during this search which contained the given key and a range number contained, at that moment, in the array *current ranges*.

Many variations on this strategy, mentioned by Maurer [9] and Morris [10] in connection with tables of the sort mentioned in 1.1.2, remain applicable to this scheme. Whether these variations, or others, are in fact used, is a rather computer- and installation-dependent affair. The main point here is that gains of large factors are possible with hash addressing, provided all current range numbers are kept track of and

used as mentioned above.

1.2. A Four-pass Compiler

1.2.1. The First Pass

We shall here describe some of the salient features of a compiler for unrestricted ALGOL 68. To begin, we consider the following program fragment:

```
begin real x;
      proc p = expr begin abc x; x := 3.14 end; .
```

In a strictly left-to-right scan of this text, we do not know when we arrive at the text *abc x* whether this is a declaration of a local *x* with some as yet unknown mode or an application of some as yet unknown monadic operator *abc* to the more globally declared *x*; that is, until the defining occurrence of *abc* has been encountered, in either a mode-declatation [DR 7.2] or an operation-declaration [DR 7.5], we do not even know whether or not to enter *x* in the identification table a second time.

A suggested method for handling this problem is to dedicate the first pass of the compiler to the task of entering mode- and priority-indications and operators in the identification table, based on their defining occurrences; this can be accomplished completely, without leaving any loose ends, by a single pass though the text of the program.

1.2.2. The Second Pass

The following program fragment illustrates another problem, (which also occurs in ALGOL 60):

```
begin int i; ... ;
      begin proc p = begin i := 1 end;
            int i;
```

Again, in a left-to-right scan, we cannot be sure while inspecting the applied occurrence of *i* in *i := 1* whether it identifies the defining

occurrence in the first line, or whether there may yet be some other defining occurrence, as in the third line, which gets the preference.

The suggested solution to this problem is to dedicate a second pass to entering identifiers in the identification table, based on their defining occurrences (which have now become recognizable, thanks to the first pass) in identity-declarations [DR 7.4] and in labels [DR 6.1.1.i]. The second pass should also construct a list-structured table associating a unique integer with each mode encountered in the program.

The objection may be raised that "sensible" programmers "always" declare things before using them, and that these first two passes are actually superfluous, for practical programs. There is some considerable merit in this objection and it is quite easy to formulate an additional context condition [DR 4.4], defining a sublanguage [DR 2.3.c] of ALGOL 68, as follows:
"No proper program contains an applied occurrence of an identifier (indication-applied occurrence of a mode-indication, operator-applied occurrence of an operator) which is textually before its defining (indication-defining, operator-defining) occurrence.".

One of the arguments of principle against such a restriction of ALGOL 60 was that it thereby made it impossible for each of a number of procedures to call the others, since one of them would have to be declared first and then could not call any of the rest. ALGOL 68 at least leaves a loophole by means of which this restriction can be avoided. Inspection of the following ALGOL 68 program fragment should make the principle clear:

```
begin proc p1, q1;
      proc p = expr begin ...; q1; ... end;
      proc q = expr begin ...; p1; ... end;
      p1 := p; q1 := q; ... .
```

A practical argument against a context condition such as that mentioned above is that it is extremely difficult for a compiler to check whether the condition has been met. Many people place a very high value on rigorous checking by the compiler, and the cost of

policing this item is likely to be nearly as great as keeping the first two passes and the unrestricted language. We leave it to individual implementers to decide, taking the characterestics of their own machines and users into consideration, whether or not they wish to combine the first two passes with the third.

1.2.3. The Third Pass

A much more subtle problem than the identification of indications and identifiers is that of operators. Let us consider the formula

$$i + x.$$

To determine which of many plus signs is meant, we first have to determine what might be called the "a priori" modes of its operands. In this case, let us suppose that i is a reference-to-integral-identifier and x a reference-to-real-identifier; then the a priori modes are 'reference-to-integral' and 'reference-to-real', respectively. A search must now be made in the identification table for all (in general, there will be more than one) plus signs with the highest range number contained in *current ranges* (see 1.1.3). These are checked, one by one, to see if the modes of their declared operands can be adjusted from [DR 2.2.4.1.h] the a priori modes of the given operands. If not, the plus signs with the next-highest range number contained in *current ranges* are checked, and so on, until a match is finally found.

In the given example, it is simple enough to generate the machine code necessary to elaborate the operands i and x to their a priori modes, the code to adjust them as required, and that to apply the required operation. However, we can easily run into difficulties if the operands are slightly more complicated. Suppose we have, for instance,

random + *if* $x > 13$ *then* x *else* y *fi*.

In this case, we do not have the slightest clue what the a priori mode of the second operand might be until we have scanned through the condition and then-clause [DR 6.5], and we are still not certain until we have reached the terminating *fi*. The problem is this: if we produce

object coding for the second operand while scanning through it, then we can very easily find ourselves without any room left over to insert the code to adjust (in this case, call) the first operand. Even more complicated situations can be envisaged.

The suggested solution is a third pass which produces, instead of machine code, an expanded source program, containing, at appropriate places, some sort of annotations, each telling the mode to which the construction to which it is attached must be adjusted. These annotations can all be of equal length, and thus, no problem arises in reserving room for them.

It may be remarked that these are very costly measures to handle what was, in the previous generation of programming languages, a relatively simple matter. We can only reply by asserting that this extra pass is not so terribly expensive, and that it is, in any case, the price that is to be paid for the generality and facility of use of ALGOL 68; furthermore, the price will have been completely paid by the time the compilation is completed - the object program will consist of lean, compact and fast code.

In an attempt to make a virtue of necessity, let us further remark that the third pass can very effectively be used to collect special information for optimizing the object program, information which can only be obtained when the identity of all identifiers has been established. For example, the third pass could keep track of which routines call which other routines, and which routines are called collaterally. Based on this information, the fourth pass could produce nonreentrant code for most of the ordinary routines, and only give the slower, reentrant form in exceptional cases.

1.2.4. The Fourth Pass

The fourth pass is able, given all the information collected by the preceding passes, to produce the machine-code version of the ALGOL 68 program by more or less standard techniques.

2. The Form of the Object Program

2.1. General Storage Management

2.1.1. The Stack

One of the characteristic innovations introduced by ALGOL 60 is its so-called "block structure". ALGOL 60 programs consist of blocks nested inside one another. Upon entry to a block, dynamically, the declarations heading it are carried out; what this means in machine terms is that storage space is reserved for the variables declared by those declarations. Upon exit from the block, this storage space may be made free again (apart from *own* declarations).

The nested character of ALGOL 60 blocks, and hence of the reservation and de-reservation of storage has traditionally been mirrored in implementations by a so-called "stack" mechanism [13,14,15]. A stack consists of a collection of contiguous storage cells and a pointer. When space is required, it is obtained beginning at the cell to which the stack pointer points, and the stack pointer is then set to point to the first free cell beyond this. Storage is made free again by resetting the stack pointer to the value it had before that piece of storage was reserved; this can only be done if the previous value has somehow been remembered, and this is usually done by reserving a cell for it in the stack in which the value of the pointer is stored before incrementing.

In ALGOL 68, we have serial-clauses [DR 6.1] nested within one another. Some serial-clauses begin with declarations, and others do not [DR 6.1.1.b]; by analogy with ALGOL 60, we shall use the word "block" for those that do. The analogy is not quite exact, since serial-clauses do not contain the brackets *begin* and *end*; on the contrary, the serial-clause is designed to be placed within various other kinds of brackets as well, for instance, between *if* and *then*, *then* and *else* or *else* and *fi* (notice that *then* and *else* each functions as a closing bracket followed by an opening bracket [DR 6.5.1].).

Thus, the stack, perhaps with some of the variations described by

Wieland [16], is a suitable and efficient discipline for ALGOL 68 as well - from the programmer's point of view to handle the storage demands of "variable" declarations [DR 9.2.a] and of (the rather rare) undisguised local-generators [DR 8.5.1.b, DR 11.5.h].

2.1.2. The Heap

However, ALGOL 68 provides a completely different kind of storage allocator, the nonlocal-generator [DR 8.5.1.c]. Here, storage is reserved, in principle, permanently, but, in practice at least for some length of time which does not coincide with block entry and exit. This type of storage allocation is that usually associated with list processing as in IPL-V [17] or LISP [18,19]. For instance, given declarations like

```
mode struct cons =(u car, ref cons cdr);
     union u =(string, ref cons);
```

then *car of a* and *cdr of b* are the ALGOL 68 versions of CAR(A) and CDR(B) in LISP, and, more relevant to our present storage-allocation considerations, *heap cons:=(a,b)* corresponds to CONS(A B).

It would seem that such constructions could be handled very simply by constructing a sort of stack, which we shall call the "heap", at the other end of the computer's storage, which would grow toward the ordinary stack. The difficulty is that storage reserved in this way, when given free, is seldom at the end (bottom) of the heap, and thus, holes (traditionally called "garbage") tend to form in the heap; it may then occur that, although the total of the garbage is large, there may not be any single hole large enough to fill some new request for storage reservation. Fortunately, much thought has been given to this problem, and it can be said that the art and science of "garbage collection" is now highly developed.

The basic principle of all garbage collection systems is the same. That is, all names (pointers) in the stack are chained together, or in some other way made recognizable so that they can be inspected one after another. Then, the value referred to by the first name in the stack is "marked" as being nongarbage; next, the

value referred to by the first name in that marked value is marked, and so on. If, however, a name refers to a value which has already been marked, or if the name does not refer to any value at all (as nil in ALGOL 68 [DR 2.2.3.5.a, DR 8.3.1.e]), then the next name, if any, in the value or in the stack is handled in the same way; if there is no such next name in the given value, then the next name in that value one of whose names led to the given value is taken, etc, until all names in the stack have been handled. The resulting situation is then that every value which can be addressed (i.e., referred to) by any means at all (i.e., by following some chain of references) has been marked. At this point, everything in the heap which has not been marked is, by definition, garbage, and it subject to collection by the next stage of the garbage-collection operation.

Let us first notice, however, that the marking process as described above is recursive and requires some number of cells on the top of the stack in order to remember how it arrived at a given value and this number is virtually impossible to predict. The irony of the situation is that garbage collection is initiated only because the free space between the stack and the heap has gotten too small to satisfy requests for new storage reservations. Fortunately, a variant of the marking scheme discussed above, described by Schorr and Waite [20], avoids this problem very elegantly, at the cost of an extra marking bit, and slightly longer execution time.

The method of Schorr and Waite is not directly applicable to ALGOL 68, however, because of the fact that a name in ALGOL 68 may refer not only to a structured (multiple) value as a whole, but also to a field (element or subvalue) of it. This implies that a value as a whole may be unreachable (i.e., no reachable name refers to it) but that a part or parts of it may still be reachable. Thus, we may not allow the whole to be given free without checking to see whether some part of it must still be saved.

We give an example to show how such a situation can arise. Suppose we have the declarations

```
struct pqr = ([1:5] int p, real q, ref pqr r);
ref pqr a; ref int ii
```

and that the assignation

a := pqr ((1,7,5,4,6), 3.14, nil)

has been elaborated. It may then happen, for instance, that the variable

p of a [3]

is required a large number of times in a certain section of a program. A good programmer will then preface that section of program with the assignation

ii := p of a [3],

in order to pay the price of selecting and indexing only once. If he thereafter writes

a := nil,

then, assuming there were no other names referring to the multiple value, it thereby becomes forever unreachable, even though the third element of its first field can be reached through *ii*.

A strategy might be suggested whereby only the unreachable parts are given free, and the rest preserved. There are considerable difficulties with such a scheme, as the following example shows. Suppose we have the declarations

ref [,] int ii2, jj2

and the series of assignations

```
ii2 := [1:n, 1:m] int;
jj2 := ii2 [4:6, 3:3];
ii2 := nil.
```

Here, we have the problem that, when the multiple value as a whole is made unreachable in the third line, there are still three noncontiguous but equally spaced elements reachable through *jj2*. Thus, whatever else the garbage collection does, it will have to preserve the spacing between those elements. The complications involved in such an operation appear enormous, and so, on the theory that it is wrong to punish good programmers for the sins of the few poor programmers who

actually write such unlikely constructions, it would seem better to follow a strategy whereby the whole of an array is given free only when all of its subarrays are also unreachable. In this way, we follow the lines of what is likely to happen anyway in most normal programs, and we also conform to the law as laid down in the Draft Report. (In this respect, it is interesting to note that the Draft Report does not specifically mention garbage collecting at all; it is "merely" an opportunistic strategy necessary for practical reasons of storage economy, and is legalized only by the sufficiently vague wording of the last sentence of DR 1.1.6.e).

One of the problems in any garbage-collection scheme is that of finding a means of marking cells as nongarbage. In many implementations of LISP, for instance, the problem is not severe, since a number of bits are otherwise unused in each LISP cell. With the more general approach of ALGOL 68, however, this is no longer the case; for instance, a real variable in the heap is almost certain to fill an entire machine word (or, in some machines, two). A perfectly general solution, and the one we recommend, is to set aside a part of storage for a "bit table", i.e. a sort of boolean array each of whose elements contains the mark for one machine word. Given a word length of, say, 32 bits, and assuming that the bit table is packed with 32 bits per word, this implies sacrificing about 3% of the storage used by the heap for the table; we may consider this a relatively modest price for the convenience of having a heap at all.

We now sketch a marking algorithm satisfying the above requirements; it is an adaptation of that of Schorr and Waite.

As with their system, we begin a "forward scan" from a name in the stack by marking that name (If we assume that a name is stored in a machine word, but does not occupy all bits of the word, then one of these extra bits can be used to hold this mark.); for each word of the value referred to by the name, a mark is set in the corresponding bit of the bit table (This implies that we somehow know how many words the value occupies; a simple method for determining this will be discussed later.). In order to make the reverse scan possible at a later stage, the first name in the given value is remembered (perhaps

in a machine register) and is replaced by a pointer (name), pointing back to the name which led to the given value; furthermore, this pointer is marked. The forward scan is then continued in this way beginning at the remembered name, until it encounters a value without any names in it, or one which is already marked in the bit table.

Then, a reverse scan is begun which follows along the reversed pointers (In general, this is the first pointer in the value) and re-reverses them; this continues until a value is encountered which still contains an unmarked name. A forward scan is then re-initiated, just as before, beginning from the first unmarked name in the given value. When the reverse scan encounters a value all of whose names are marked, it removes the marks on all the names of that value and continues backwards in this way until it reaches the name in the stack from which the forward scan was initiated. Then a forward scan is begun from the following name in the stack until all names in the stack have served to start a forward scan.

We have omitted a detail related to the problem of subarrays discussed above. During a forward scan, if a name refers to a subarray, then the descriptor of the full array is to be consulted, and all elements described by it are to be marked in the bit table; however, in the case that the array elements are names, it is sufficient to start forward scans from only the elements of the subarray.

In this way, all reachable words and only reachable words will have been marked in the bit table, except that nonreachable elements of an array having a reachable subarray will also have been marked. Then, all holes in the heap can be chained together and used when new heap storage is requested, or, alternatively, and at the cost of a scan to update all names in both stack and heap, the holes can be compressed out of the heap.

2.2. The Machine Code Corresponding to Certain Constructions

2.2.1. Mode Declarations

We consider here only those mode-declarations [DR 7.2] which introduce structured modes; e.g.,

struct ex = (int a, ref ex ex, real b, ref real bb).

Quite apart from the effect of such a declaration on the identification table, it must leave a shadow of itself behind in the compiled program. This shadow, which we shall call a "template", consists of a contiguous piece of storage, containing a number of integers. The first integer is the number of words occupied by a value of the given mode, and the second is the number of fields which are names, or arrays of names. Then, for each such field, we must have its distance, in words (or some other unit, such as bytes) from the beginning of the structured value.

These templates are necessary, during garbage collection, in order to determine how many consecutive words in the bit table are to be marked, and in order to locate the names for the forward and reverse scans.

It will be convenient to associate a unique cardinal number with each different template; we can then speak of the "template number" of a template, and hence, of the template number of a given value. It will also be convenient to reserve the first few template numbers for special purposes, as, e.g., 0 for "simple value, nonname", 1 for "simple value, name", 2 for "array; see descriptor for details", etc.

2.2.2. Names

A name is, in principle, merely the address of some value in machine storage. We shall require, however, that the template number of the value to which the name refers be attached to the address, this for the sake of the garbage collector. In most modern computers, a single word can contain an address and still have a number of bits left over. Thus, if we reserve one of these bits for the marking process, we may

still have enough bits left for template numbers up to 127 or 255 or so, enough for all likely programs.

2.2.3. Multiple Values

The definition of multiple values (arrays) in DR 2.2.3.3 suggests a suitable implementation rather directly.

The offset in the descriptor will in general actually be the absolute address in storage of the first element of the array. The last stride, d_n, will be the length of each element, and not, as suggested by DR 7.1.2.c Step 6, always 1. The states, s_i and t_i, require only one bit each, and can conveniently be packed into a single word (at least, for arrays of moderate dimension); if the roles of one and zero are chosen oppositely to what is described in the Draft Report, then a simple test of this word for equality to zero will yield the answer to the question "Can this array be replaced exclusively by arrays with exactly the same lower and upper bounds?"

In connection with garbage collection, it is necessary that the template number of the elements of the array be known; it can probably be combined with the offset in one word, in the same way as in 2.2.2. It is also necessary to know whether a given descriptor describes a subset of some other array, and, if so, where the descriptor of that other array is to be found; hence, an additional word must be set aside in all descriptors to contain the address of the "main" descriptor of the array, or, if it is the main descriptor, some distinctive mark (e.g., the address zero).

The storage for the elements of an "ordinary" array should be allocated in the stack. Clearly, the storage for an array created by elaboration of a nonlocal-generator [DR 8.5.1.c] must be allocated in the heap. Furthermore, storage for the elements, but not necessarily the descriptor, of a multiple value with flexible bounds (i.e. with one or more states equal to zero) must be allocated in the heap.

2.2.3.1. Strings

Strings (that is, multiple values of the mode 'row of character')

require special treatment in any effective implementation, except for those without flexible bounds. This is because of the peculiar nature of the operations which are usually performed on strings, such as the concatenation of new elements on either end, the removal of elements from either end, and even the insertion and removal of new elements between two elements of the string.

If all elements of a string were made to be contiguous in storage, then most of the above-mentioned operations would require the copying of all (or nearly all) of the elements of the string from their previous position in the heap to a new position where sufficient room was available for all the new elements; this copying is, in itself, a fairly expensive affair, and it also generates large quantities of garbage.

The suggested solution is to divide a string into a number of "bundles" of characters, each bundle being some fixed number of words long (e.g., four or five), and each containing not more than some maximum number of characters. Each bundle should contain a pointer to the next bundle, and, preferably, a pointer back to the previous bundle. Then, many of the operations could be carried out locally, in a single bundle, or by attaching a new bundle to the chain or removing one.

This system does make the indexing of strings more difficult, since it is then not possible to calculate the address of, say, the i-th element, without following along the chain. On the other hand, if each bundle contains about 16 or 20 characters, and supposing an average of not more than about five bundles in a string, then the extra costs for this indexing are fairly modest.

2.2.4. United Variables

ALGOL 68 permits declarations like

union (int, real, bool) irb.

That is, the elaboration of this declaration must reserve storage for a value which at one moment may be an integer, the next moment a real number, and later, perhaps, a truth value. This can be implemented by

reserving a number of contiguous storage locations, the first of which is used to contain the integer associated with the mode of the value which at that moment is contained in the following location(s). The number of locations required is thus one more than that required for the largest of the possible values which need to be stored.

However, given a declaration like

union (int, struct (real a,b,c,d,e,f, compl g,h,i,j)) is,

the variation in the lengths is rather great. One might consider storing only the shorter values locally, and providing only a pointer to the longer values in the heap. In fact, such a strategy is virtually inevitable in a case like

union (int, [ ] real) irr,

since the real array could be of any length. Hence, the first word must also contain a mark telling whether the value itself, or only a pointer to it, is contained in the other word(s). Furthermore, such a "united value" must be inspected by the garbage collector in its marking phase.

2.2.5. Conformity Relations

ALGOL 68 provides a construction, the conformity-relation [DR 8.9], to determine the mode of a value, and, if desired, to retrieve that value [DR 8.9.2 Step 4]. This operation can be performed rather easily and rapidly if the compiler constructs the integers which characterize modes appropriately. This number should actually consist of two numbers, one characterizing the "basic mode" (i.e. that part of the mode following all initial 'reference to's), and the other being the number of initial 'reference to's.

Then, in cases like those in DR 11.11.q,r,s,etc., where the left-hand mode is known on syntactic grounds not to be a reference to a union, the conformity can be tested by first testing the basic modes for equality; if they are not equal, then the conformity fails (i.e., yields the value false); otherwise, if the number of 'reference to's on the left-hand side does not exceed that on the

right-hand side by more than one, then the conformity succeeds (i.e., yields the value true).

The conformity-relation has still another function which should also be kept in mind. Given the declarations

union (int, real) ir;
union (int, real, char) irc,

one is not permitted to write the "assignation"

irc := ir

since the mode of *val irc* is not united from [DR 2.2.4.1.h] that of *val ir*. However, one can write the conformity-relation

irc ::= ir.

The point to notice is that this conformity is certain to succeed; the compiler should therefore recognize such situations, omit the test for conformity, and compile only the necessary instructions to perform the required assignment.

Quite a different situation is the conformity-relation

ir ::= irc;

here, the conformity may succeed or fail, and a dynamic test must indeed be performed. This test is more complicated than the simple case, since the right-hand side has "two chances" to conform; each of them must therefore be tried.

References

[1] A. van Wijngaarden (editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Draft Report on the Algorithmic Language ALGOL 68, report MR93, Mathematisch Centrum, Amsterdam, 1968.

[2] P. Naur (editor), Revised Report on the Algorithmic Language ALGOL 60, C. ACM, 1963, pg. 1.

[3] N. Chomsky, On Certain Formal Properties of Grammars, Information and Control, 1959, pg. 137.

[4] M. Sintzoff, Existence of a Van Wijngaarden Syntax for Every Recursively Enumerable Set, Annals de la Société Scientifique de Bruxelles, 1967, pg. 115.

[5] S. Ginsburg, A Survey of ALGOL-like and Context-free Language Theory, in T.B. Steel (editor), Formal Language Description Languages for Computer Programming, North-Holland, 1966, pg. 86.

[6] E.T. Irons, A Syntax Directed Compiler for ALGOL 60, C. ACM, 1961, pg. 51.

[7] K. Samelson and F. Bauer, Sequential Formula Translation, C. ACM, 1960, pg. 76.

[8] R.W. Floyd, Bounded Context Syntax Analysis, C. ACM, 1964, pg. 62.

[9] W.D. Maurer, An Improved Hash Code for Scatter Storage, C. ACM, 1968, pg. 35.

[10] R. Morris, Scatter Storage Techniques, C. ACM, 1968, pg. 38.

[11] B. Randell and L.J. Russell, ALGOL 60 Implementation, Academic Press, 1964.

[12] A. Batson, The Organization of Symbol Tables, C. ACM, 1965, pg. 111.

[13] E.W. Dijkstra, ALGOL-60 Translation, ALGOL Bulletin Supplement nr. 10.

[14] P. Naur, The Design of the Gier ALGOL Compiler, BIT, 1963, pg. 123.

[15] F. Kruseman Aretz and B. Mailloux, The ELAN source text of the MC ALGOL 60 system for the EL X8, report MR84, Mathematisch Centrum, Amsterdam, 1966.

[16] H. Wieland, Speicherzuweisung für Variable in ALGOL-Objekt-programmen, Electronische Datenverarbeitung, 1967, pg. 3.

[17] A. Newell (editor), Information Processing Language-V Manual, Prentice-Hall, 1961.

[18] J. McCarthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I, C. ACM, 1960, pg. 184.

[19] C. Weissman, LISP 1.5 Primer, Dickenson Publ. Co., 1967.

[20] H. Schorr and W. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, C. ACM, 1967, pg. 501.

Samenvatting

Het Draft Report on the Algorithmic Language ALGOL 68 beschrijft een nieuwe programmeertaal waarvan het de bedoeling is dat zij de officiële opvolgster wordt van ALGOL 60. Deze nieuwe taal is ontworpen, zodanig dat algorithmen hierin geschreven, gemakkelijk zijn op te stellen door, te onderwijzen aan en mede te delen aan mensen, terwijl zij toch nog op efficiente wijze uitgevoerd kunnen worden door hetzij nu dan wel in de nabije toekomst beschikbare rekenautomaten. ALGOL 68 is niet slechts een uitgebreide versie van ALGOL 60 alhoewel vele goede eigenschappen in de ervaring gewonnen in de toepassing van ALGOL 60 het ontwerp van deze nieuwe taal hebben beïnvloed.

Het Draft Report definieert ALGOL 68 in termen van "handelingen" verricht door een "hypothetische rekenautomaat". Evenwel zou iedere werkelijke rekenautomaat, die trachtte deze handelingen letterlijk als beschreven uit te voeren, hopeloos langzaam zijn. Gelukkig is deze letterlijke interpretatie niet vereist en daarom zal men de ALGOL 68 programma's vertalen in programma's geschreven in de machinetaal van de rekenautomaat. Deze vertaling wordt het best verricht door de rekenautomaat zelf met behulp van een speciaal programma, vertaler geheten.

Het eerste hoofdstuk van dit proefschrift handelt over verschillende belangrijke aspecten van de werking van een ALGOL 68 vertaler. Een van de taken van elke vertaler is het vergaren en het systematisch opbergen van informatie betreffende het bronprogramma, te gebruiken in latere stadia van de vertaling. Een nieuwe algorithme, geschikt voor ALGOL 68 en andere talen met een blokstructuur, wordt gegeven voor het efficiente beheer van een identificatietabel. Bovendien wordt een schets gegeven van een viertrapsvertaler. De eerste trap behandelt de identificatie van indicaties, de tweede trap die van identifiers en de derde trap die van operatoren. De vierde trap kan daarna een efficient machinecodeprogramma produceren.

Het tweede hoofdstuk handelt over enkele trekken van het object-programma die eigen zijn aan ALGOL 68. Naast de bekende stapeltechniek vereist ALGOL 68 bovendien een hooptechniek waarin reservering en vrijmaking van geheugenruimte niet noodzakelijkerwijs samenvallen met blok-binnenkomst en blokverlating. Een algorithme wordt gegeven met behulp waarvan niet gereserveerde geheugencellen in de hoop worden onderkend om ze opnieuw te kunnen gebruiken. Ten slotte wordt de machinecode behorende bij enkele karakteristieke ALGOL-68 constructies beschreven en toegelicht.