THE CONSTRUCTION OF AN ALGOL TRANSLATOR FOR A SMALL COMPUTER

W. L. van der Poel

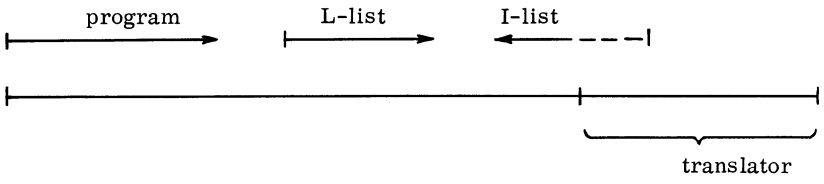
Dr. Neher Laboratory PTT,
Leidschendam, Netherlands

For the computer Zebra we have undertaken a project of constructing an Algol translator with the aim of being as complete as possible (with the only exception of own variable arrays). As a few new concepts are used in this design, it is thought to be useful to communicate them in this form.

Zebra is a two address machine with a very limited fast store and a reasonable drum store of 8192 words. The operation code is a rather peculiar one. Every bit of the 15 bits of the function part of an instruction is used separately for a different elementary function. The operations are in fact micro-operations and more complicated actions such as multiplication, division, list searching, etc. are not built in functions but can be readily programmed. This makes the code extremely powerful for just those operations which are required for an Algol translator. On the other hand it is not the appropriate object language for Algol. Therefore we decided to create a new object language within the machine which is used interpretatively. The structure of this object language (called Intermediate Code or IC) is fully adapted to the requirements of Algol. The analytic code of Zebra helps a great deal in making the interpreter fast and compact. Another reason why an interpretive code is not such a drawback is that the machine has no built-in floating point operations so that these have to be programmed anyway.

The system as a whole works on a load-and-go basis. First the translator is put into the machine, into the upper end of the store and the Algol program is then fed in, translated and built up in the lower part of the store. Then the interpreter overwrites the translator and the program can start working. We shall distinguish these phases by the names translate phase and run phase.

During translation the store is divided into three lists:
(a) the program to be built-up,
(b) the identifier list (I-list),
(c) the L-list.
The arrangement is as follows:



The I-list starts at the high end of the store and has an extension reading into the translator for the fixed (pre-declared) identifier such as sin, read, etc.

The program is built up in the lower part; the L-list is somewhere in between. The reason for making the I-list running backward is a very important one. For every occurrence of an identifier the I-list must be searched for this identifier and this searching process must of course be as fast as possible. In our analytic code this can be done by a repetition instruction, searching in ascending order through consecutive drum location. Thus a fully optimum timing is attained and the last identifier placed in the list is found first. This accounts for the local, non-local concept. For fast searching see (1).

We shall call an Algol symbol an identifier, a number or a delimiter. It is silently understood that the read routine already composes the complete identifiers and numbers and also determines the correct delimiter expressed by a word such as procedure. The read routine also skips all comment situations. We distinguish 3 such comment situations:

(a) the situation after comment
(b) the situation after end
(c) the situation after) followed by a letter.

The third kind of comment also copes with the parameter delimiter but far more than that. It is permitted in our translator to have any string not containing (. All comments are skipped only

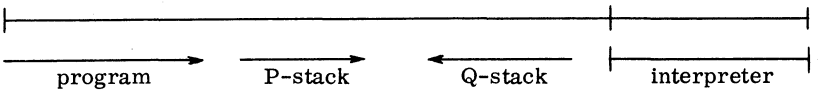
after treatment of the foregoing symbol by the translator thus obviating the difficulty of comment after the last end. Identifiers are read as numbers in radix 37. (26 letters, 10 figures and 1 spare). More than 6 characters give an overflow of the capacity of the word but this is allowed. Only a marking bit is kept which distinguishes an identifier of more than 6 characters from one with 6 or less. Hence there is a risk that two identifiers of more than 6 characters will be represented by the same word but we have taken that extremely remote chance thus being able to place no limitation on the number of characters in an identifier.

The L-list serves many different purposes. In the first place it holds these Algol symbols which cannot be translated directly because of their order of occurrence. E.g. in a + b * c + ... the read routine reads a, +, b* and on seeing * it knows that it has to go on reading until an operator of lower rank than * has been read. On reading the + it translates into take b, multiply c and only then remove a and + from the list by adding add a to the program.

It will be clear from this example that we have only obeyed in our translator to the rules of precedence for operators and we have not adhered to any rule for the order of evaluation of the primaries. The change of order into inverse polish notation is now well known so that I can leave details about this process to be read elsewhere.

In the second place the list L is used for string symbols like begin together with several stack pointers and variables as they were at the moment of reading begin. E.g. in the case of begin there are stored the address where the last instruction that has been translated has been stored, the last location of the identifier list and the last relative address given outside the block just being entered. Then after having translated the completed inside of the block everything what happened inside has disappeared from L and on finding end all pointers and variables are restored to their value before entering the block. When the program meets L or L meets I, the list L is shifted up or down wholesale.

During run time the translated program is already built up in the store. The translator is now overwritten by the interpreter (or for smaller Algol programs both can be held in the store together). The remainder of the store is again used for two stacks. The Q-stack is used for all simple variables and simple intermediate results. These intermediate results are stored in the true



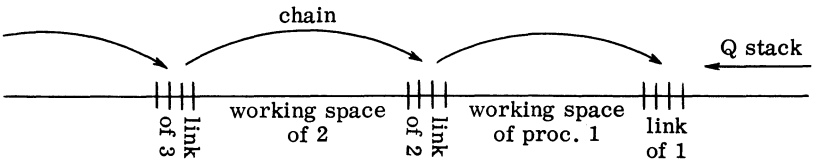
stack fashion on top and are exactly used in the reverse order in which they are put in the stack. Intermediate results are stored in unpacked form. The P-stack is used for arrays, fixed and variable. The allocation is fully dynamic and is only determined during run time.

Constants, own simple variables and own (constant bound) arrays are imbedded in the program itself. For a single constant the instruction using it automatically jumps over the constant placed consecutively. For own arrays the space for it is reserved at the time the declaration is translated.

The greatest advantages from the two stack system come from the allocation system during translation. There are two kinds of locations which cannot be given an absolute address during translation time; one is the variable array, which is dynamically allocated during run time; the other one is the space for formals in procedures.

As procedures can be used recursively a single absolute address is not sufficient for all levels of activation. The mechanism for allocations therefore is staged as follows: Every instruction has an operation part, an address part and a rank. As long as the translator is not translating the inside of a procedure declaration and hence is not dealing with any formals, all simples can be given absolute addresses. They get a rank 0. Also the entering of blocks does not necessitate other than absolute addresses. But as soon as a procedure is translated it is not known when or where or at what level this procedure is activated. Then all formals are given relative addresses, and the instruction referring to a formal gets an appropriate rank $\neq 0$. The first translated procedure gets a rank 1. When inside that procedure again another procedure is declared, this gets rank 2 etc. Hence the system of ranks is a static picture of the level of a procedure, not a dynamic one during run time. The top of the stack at run time when calling a procedure is put into the stack together with the link and some other quantities. Then the called procedure starts work and formals within the body are using the relative addresses given during translation relative to the beginning point determined during run time. When for example in a procedure of rank 20 a non-local quantity of rank 5 is

needed, the interpreter sees that the rank of the quantity needed is not the same as the present rank and then goes through the action of chain searching. The link of rank 20 contains the top of the stack at rank 19 and these quantities again contain the top of the stack at rank 18 etc.



The reason for not fixing down the beginning points of the relative addresses in procedures is that it can happen very well that in a procedure at level 20 a non-local quantity of level 5 is called which appears to be a new parameter-less procedure call rising to level 30 in itself and building up a completely new chain of relative addresses in the top of the stack from 5 to 30. This does not interfere with the old chain which still remains in the stack lower down. When this last procedure has been evaluated, everything needed for the evaluation has disappeared again from the stack and the former level 20 proceeds.

The process of chain searching is a very fast one in our computer as it can be done by underwater programming techniques. For a fuller account of this see (1).

To give a better insight into the coding of a procedure we shall give here an abbreviated form of the structure.
The procedure declaration is coded as:

X : An instruction calling the displacer subroutine. The displacer sets the link and makes the full formal actual correspondence.

D_1 : A specification pattern giving information about type, value, etc. of the first formal parameter

D_2 : Specification pattern of second parameter

.
.
.

D_k : Specification pattern of last parameter. All D's are positive.

-r : The rank of this procedure put there negatively so that the displacer can see the end.

body: Here follow all instructions of the procedure body in IC or machine code.

Y : The procedure body ends with a subroutine Y called the counter displacer which returns and restores the stack top pointers to their previous values in the calling program.

The procedure call looks as follows:

J : A jump to the X of the procedure to be called

s : A link address to where the procedure must return, which at the same time gives an indication of the place where to find the key of the first actual

Object progr. of Y_1: The object program for the evaluation of the first actual expression when this is not a simple variable. Absent when Y_1 is a simple

Object progr. of Y_2: etc.

Object progr. of Y_k:

0 A zero which signals the end of the retrogressive list of keys for actuals

Key of Y_k : The key of Y_k gives information about the kind of the actual, whether formal or actual. It can be a simple, an expression, a procedure, a label or switch, an array, a string or a formal

Key of Y_2

Key of Y_1

The displacer treats all actuals in turn and compares the pattern D_k with the keys, transports the quantities called by value and transforms type when the specified type is not the same as the type of the actual. In fact the type specification and the value list is the only information extracted from the specification. Procedure, label, switch, array, string automatically are clear from the parameter keys.

The structure of the instruction is worth reviewing for a while. The bits are numbered from left to right. Bit $q = \overline{I_0 - I_6}$ from the type of operation. When $I_0 = 0$ the instruction is a real machine code instruction and is not interpreted. When $I_0 = 1$ then $q > 64$.

The operations $64 \leqslant q < 96$ are the arithmetic, relational and Boolean operations such as add, equal, implies etc. Bit I_6 serves

the purpose of indicating whether a non-commutative operation is progressive or regressive i.e whether e.g. a subtraction has to form a-b or -a+b. Hence all operations such as division, implication, greater are only present in one version. The other version is produced by interchange of operands.

The operations $96 \leqq q < 104$ are the extractive operations such as take, take negatively, take inverse).

The operations $104 \leqq q$ are called the non-extractive operations. They contain the storing operations of different kinds (store accu, store address, store factor, store procedure) and many organizational operations as test, jump, pass, adjust (for the dynamic block begin), restore (for the restoration of stack pointers on leaving a block) and verify. The last one is a very peculiar one and is inserted where during translation no information about the kind can be extracted from the quantity itself as for example a constant as an actual parameter of a procedure which is itself called by name. Or in the case that a non-specified actual is a subscripted variable but where it is not known whether this is a designational expression or an arithmetical expression. The verify order is then substituted and during run time when of course the kind is clear from the actual keys the true operation is executed. Some operations are true machine code operations such as X, calling the displacer and Y, calling the counter displacer, restore (for labeled places for restoring stack pointers as entrance from jump leaving from inner blocks), return (for returning from an evaluation of an actual which is an expression), several kinds of for instructions for doing the for, step and while elements.

The bit I_7 indicates the type of the operation. $I_7 = 0$ indicated integer or Boolean type and $I_7 = 1$ indicates real type. It has an advantage to have Booleans represented by the integers 0 ——— 0 and 1 ——— 1. In that case the logical operations of intersection etc. on binary numbers can then be very simply done by applying Boolean operators on integers. But of course this must be hidden in a machine code body procedure looking like Algol on the inside as this sort of tricks does not belong to Algol.

The bit $I_8 = 1$ indicates in an extractive instruction that the type declared is not in correspondence with the type required. Then an automatic transfer of type takes place. For example in formal instructions the type bits I_7 and I_8 are copied from the parameter keys during the formal actual correspondence.

At last the bit $I_9 = 0$ indicates that a formal parameter is used, $I_9 = 1$ indicates that the operation concerns an actual.

$I_{10} - I_{19}$ denote the rank of the instruction which initiates the chain searching when the rank of the instr. at hand is not in line with the rank of the procedure at hand. $I_{20} - I_{32}$ is the absolute or relative address.

We have thus created within a machine not made for Algol an operation code and address system which comply very admirably with the requirements of Algol.

There is one further subject on which I should like to make some explanations. It is not required in our system to give procedure declarations or switch declarations in any prescribed order. It can very well happen that in a procedure A a procedure B is called while B has still to be declared, provided it is declared in the same block. Also a go to statement can lead to a label which is only declared later in the block or even can be declared later or earlier outside the block. (A label followed by colon is considered to be the declaration of a label). The solution has been found as follows: every occurrence of an actual or a label in a go to statement is counter-declared and placed in the I-list, also when already a declaration outside the block has occurred. Only on finishing the translation of a block all counter-declarations are treated and looked up whether they correspond with a normal declaration. They are then cancelled and all instructions concerned are adjusted accordingly. If not, they are shifted down in the identifier list and incorporated in the identifier list of the enclosing block. When then the declaration comes in the outer block they are treated there. At last no counter declarations may be left over otherwise there is an error. It follows that as switches are fully treated in the same way as labels, a switch declaration may in principle be given anywhere in the block.

Acknowledgement. A great many ideas for the realization of this translator have been contributed by Dr. G. v.d. Mey to whom I must express my warmest gratitude. Also to Mr. Mulders who coded a big part of the translator and brought the technique for underwater programming so peculiar for Zebra to a new height.

REFERENCES

1. W. L. van der Poel, Microprogramming and trickology, in Digitale Informationswandler, ed. W. Hoffmann Vieweg 1962.