Numerische Mathematik 2, 312—318 (1960)

Recursive Programming*

By

E. W. DIJKSTRA

The Aim

If every subroutine has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need *simultaneously*, and the available memory space is therefore used rather uneconomically. Furthermore—and this is a more serious objection—it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on.

We intend to describe the principles of a program structure for which these two objections no longer hold. In the first place we sought a means of removing the second restriction, for this essentially restricts the admissable structure of the program; hence the name "Recursive Programming". More efficient use of the memory as regards the internal working spaces of subroutines is a secondary consequence not without significance. The solution can be applied under perfectly general conditions, e.g. in the structure of an object program to be delivered by an ALGOL 60 compiler. The fact that the proposed methods tend to be rather time consuming on an average present day computer, may give a hint in which direction future design might go.

The Stack

The basic concept of the method is the so-called *stack*. One uses a stack for storing a sequence of information units that increases and decreases at one end only, i.e. when a unit of information that is no longer of interest is removed from the stack, then this is always the most recently added unit still present in the stack. For example, one can construct a stack as follows: a number of successive storage locations are set aside for the stack and also an administrative quantity, the "stack pointer", that always points to the first free place in the stack (i.e. the value of the stack pointer may be defined as the address of the first free location in the stack; if the stack is empty to start off with, the stack pointer is equal to the start address of the portion of the memory reserved for the stack). When an information unit is added to the stack the stack pointer indicates where this unit must be stored and when this has been done, the value of the stack pointer is accordingly increased. To remove one or more information units from the stack the value of the stack pointer is suitably decreased.

* Report MR 33 of the Computation Department of the Mathematical Centre, Amsterdam.

If we mark off, on a time axis, the moments when a unit is added to or removed from the stack, by using an opening bracket for the addition of a unit and a closing bracket for its removal, then we obtain a correctly nested bracket structure, in which opening and closing brackets form pairs in the same way as they do in a normal algebraic expression involving brackets. This is closely related to the circumstance that we can use a stack for storing the intermediate results formed in the evaluation of an arbitrary algebraic expression by means of elementary algebraic operations. In this case our interest is always restricted to the most recent element in the stack. As the intermediate results are used only once, use of an element implies its removal from the stack.

A simple example may be given in illustration; the successive stack locations are indicated by $v_0, v_1, v_2, \ldots$ etc. The evaluation of

$$A + (B - C) \times (D/E + F)$$

can be split up into: $v_0 := A$; $v_1 := B$; $v_2 := C$; $v_1 := v_1 - v_2$; $v_2 := D$; $v_3 := E$; $v_2 := v_2/v_3$; $v_3 := F$; $v_2 := v_2 + v_3$; $v_1 := v_1 \times v_2$; $v_0 := v_0 + v_1$; the required result is formed in v_0. The reader will be aware that the operations used are of only two different types. If we denote the value of the stack pointer by k, they are:

1. selecting a new (explicitly mentioned) number, say X, which process is described by $v_k := X$; $k := k+1$; and

2. performing an arithmetic operation, say OP, which consists of

$$k := k - 1; \qquad v_{k-1} := v_{k-1}\, OP\, v_k.$$

If we refer to type 1 by the name of the selected variable and to type 2 by the operator in question, then we can also record the program by means of the symbol sequence:

$$A, B, C, -, D, E, /, F, +, \times, +.$$

In this description the v's and, what is more important, specific values of the stack pointer no longer appear. Here it is unnecessary to specify the values of the stack pointer in the text, as the computer can keep track of its value during execution of the programm: the v's have become completely *anonymous* again, just as anonymous as they originally were.

The above is well known (see for instance [*1*]) and so elegant that we could not refrain from trying to extend this technique by consistent application of its principles[1]. Let us consider for a moment the operation for the selection of an argument, e.g. the third "$v_2 := C$". This operation can be executed without further claim for memory space, as we assume that the numerical value of C

[1] Without doubt, in view of the vivid interest in the construction of compilers, at least some of these extensions have been envisaged by others but the author was unable to trace any publication that went further than [*1*]. The referee was so kind to send him a copy of the report „Gebrauchsanleitung für die ERMETH" of the Institut für Angewandte Mathematik der ETH, Zürich, a description by Heinz Waldburger of a specific program organisation in which similar techniques as developed by Professor H. Rutishauser in his lectures are actually incorporated. The author of the present paper thinks, however, that there the principle of recursiveness has not been carried through to this ultimate consequences which leads to logically unnecessary restrictions like the impossibility of nesting intermediate returns and the limitation of the order of the subroutine jump (cf. section F 44 of the report).

can already be found in the memory. If, instead of C, a compound term had occured in the expression, e.g. $C=\big(P/(Q-R+S\times T)\big)$, then we would have used v_2 up to v_5 for the calculation of this subexpression, but the nett result of this piece of program would still be $v_2:=P/(Q-R+S\times T)$ or $v_2:=C$. In other words, it is immaterial to the "surroundings" in which the value C is used, whether the value C can be found ready-made in the memory, or whether it is necessary to make temporary use of a number of the next stack locations for its evaluation. When a function occurs instead of C and this function is to be evaluated by means of a subroutine, the above provides a strong argument for arranging the subroutine in such a way that it operates in the first free places of the stack, in just the same way as a compound term written out in full.

Stacked Reservations

In the evaluation of an algebraic expression as described above, we stack *numerical information:* the next place in the stack only becomes occupied when we actually fill in an intermediate result, it becomes vacant as soon as its contents have been used. We now consider the case that this expression—which occurs in what we shall refer to as the (relative) main program—contains a function that is to be evaluated by means of a subroutine. As far as the main program is concerned, *all* variables that the subroutine introduces for its own internal use are anonymous, and they should be placed in the next free places of the stack. Within the subroutine, however, we can make distinction between three types of quantities.

The parameters. We use the name "parameters" for all the information that is presented to the subroutine when it is called in by the main program; function arguments, if any, are therefore parameters. The data grouped under the term "link" are also considered as parameters; the link comprises all the data necessary for the continuation of the main program when the subroutine has been completed. Should one wish to do so one can leave the first free place in the stack open, so that the subroutine can place its "function value" there. Thereafter the next places in the stack are used for the parameters. (In some respects it is convenient if all parameters occupy the same number of places in the stack; if, as in ALGOL 60, a parameter may be given in the form of an arbitrarily complicated expression, the limited parameter space in the stack can refer to a point in the memory where further specification of the parameter is given. The amount of information for the link can also be regarded as being constant. Hence the amount of stack space used for the parameters is constant and known for every call.)

The local variables. In general the subroutine itself is a piece of program in which explicit reference is made to a number of quantities introduced by the subroutine. Their values are no longer of interest as soon as the subroutine has been completed. In the course of the execution of the subroutine they can take on a number of different values in succession, and their values can be used more than once (this is the reason why they cannot remain anonymous in the subroutine itself). We allow the amount of memory space occupied by these local variables to be dependent on one or more parameters, e.g. one of the input parameters can be the length of a local vector. We restrict ourselves to the case that the amount of storage space occupied by the local variables becomes known as soon as the input parameters are given, and that it remains constant during that particular activation of the subroutine. (This restriction is in accordance with ALGOL 60; from a logical point of view the restriction is not essential but it simplifies matters considerably.) We therefore know at the beginning of the subroutine, how much memory space the local variables will require this time, and can see to it that they will be stored in the stack immediately after the parameters.

The "most anonymous" intermediate results. During the execution of the subroutine intermediate results that are anonymous even in the text of the subroutine also play a role: for, in general, expressions will have to be evaluated there too, and the subroutine will in its turn call in one or more subroutines itself. These "most anonymous" intermediate results are placed in the stack in the same way as those of the main program, but we must now begin at the first free place following the last local variable.

Consequences

The main program used the stack exclusively for intermediate numerical results that were formed and added to the stack once, and were later used once and removed. Until then there was no need to store the stack in a random access memory, for our interest was at all times restricted to the youngest element in the stack. In principle we could have used a small magnetic tape that would have to move one place forward in writting and one place backward in reading. The fact that random access is not necessary there, is a direct consequence of the fact that these stack places as such need not be explicitly mentioned in the description of the computation.

Inside the subroutine we store the most anonymous intermediate results in the "top" of the stack in just the same way. Every reference to a local quantity, however, implies that one is interested in a place that is situated deeper within the stack, and *here* one is interested in random access to the stack places, in other words, we must be able to give the places deeper in the stack some kind of address. The point in the stack from which the latter is available for a subroutine, is handed over to the subroutine at the moment it is called in. We assume a static, i.e. constant description of the subroutine to be present in the memory: as a result, this description must be sensitive to the dynamic specification, as described above, of the reference point in the stack. The static reference (static addressing) of the local variables can only occur in terms of a fixed position with respect to the reference point. The value of this reference point is derived from the value of the stack pointer at the moment of the call, and will therefore generally vary from call to call.

The Link

We must now investigate which data are to be stored in the stack under the heading "link". For the sake of simplicity we regard our computer as consisting of two parts. We refer to the memory space required for the storing of the program and of the stack as "the memory", and we will call the rest "the arithmetic unit". The following considerations—suitably interpreted—apply to both a built-in and a programmed arithmetic unit.

We regard the operation "$v_2 := C$" of our example as an elementary operation of the arithmetic unit. As a result of this operation information in the memory has been modified, but other changes have occured too. Before the execution of this operation, the state of the arithmetic unit was such that the order "$v_2 := C$" was the next to be obeyed, after its execution the state is such that the next order is to be obeyed. Part of the arithmetic unit therefore stores information specifying its state (it contains something equivalent to an order counter). If the text of the program does not specify explicitly the values of the stack pointer, and the arithmetic unit is therefore obliged to keep track of it, the value of the stack pointer is another aspect of the state of the arithmetic unit. One can say quite generally that every instruction is carried out correctly, provided that the arithmetic unit is initially in the appropriate state, and part of the execution of an order is the modification of the state of the arithmetic unit in such a way that the next order will, in its turn, be executed correctly. We now consider the case that the value of C cannot be found ready-made in the memory, but must be calculated by means of an function subroutine. As we are looking for an arrangement in which it is immaterial to the surroundings where C comes from as long as it arrives in the desired stack location, it is necessary that after completion of "$v_2 := C$", the arithmetic unit is left behind in exactly the same state as if C had been transported from the memory by means of an elementary operation. This must hold regardless of the complexity of the subroutine used to calculate C. As the subroutine is in general a piece of program that takes full advantage of the flexibility of the arithmetic unit, its execution obviously implies a series of changes within the arithmetic unit, and we must therefore be willing to record sufficient data regarding the state of the arithmetic unit in order to be able to reconstruct it later. These are the data that must be stored in the stack under the name "link" when a subroutine is called in. (It may be possible that some of the disturbances will be annihilated automatically, e.g. the filling of the stack, which is, in principle at any rate, emptied again. For such "reversible" disturbances the reconstruction requires no extra measures.)

It is clear that some form of "return address" is part of the link data. What the link data should furthermore include depends on the number of different aspects of the state of the arithmetic unit, and whether their changes are intrinsically irreversible. Filling the stack is a reversible process, except when the language permits—as ALGOL 60 does—that a subroutine under execution remains unfinished on account of some criterion, and is left once for all via an exit other than its normal "return". A function subroutine will generally be called in during the evaluation of an expression, so that some stack places are filled with anonymous intermediate results that will never be used on account of the unusual exit from the function routine. In that case one must be able to reconstruct up to which point the stack contents are still of interest. This facility requires an additional record in the link.

Furthermore there exists a certain hierarchy in the information specifying the state of the arithmetic unit. If we regard the state of the arithmetic unit between two orders, there is certain information (order counter) that sees to the transition to the next order. During the execution of an order, however, we can distinguish between different "sub-states": the information specifying these substates is of no significance at the moment of transition from one order to the next, and there is therefore no need to record this information in the link, as long as the subroutine mechanism only operates between orders. The specification of the substates should be recorded in the link, however, as soon as one allows the subroutine mechanism to operate during the course of an elementary operation, i.e. when (for the execution of one of its sub-operations) a subroutine must be called in that may possibly make use of the full flexibility of the arithmetic unit.

When a subroutine, say subroutine A, is activated, it must operate in the stack starting at a point that only becomes known at the moment of call. The parameters (including the link) and the local variables of A have a fixed positioning with respect to each other, but their position as a whole is only determined at the call: the quantity specifying the position of the whole block, is called a *parameter pointer*. The program for subroutine A can only be executed correctly if the current value of the parameter pointer is at the disposal of the arithmetic unit, in other words, the parameter pointer can be regarded as one of the aspects of the state of the arithmetic unit, and it must therefore be recorded amongst the link data. Now consider the case that subroutine A calls in subroutine B. We call the value of the parameter pointer indicating the parameters and local variables of A and B respectively, PPA and PPB respectively. If subroutine B is called in by A, the value PPA is recorded as part of the link formed at this call. The place in the stack where this link is recorded is deduced from the value of the stack pointer at that moment and is, by definition, recorded in the parameter pointer, which now assumes the value we called PPB. When we return from B to A the return address is found in the stack under control of PPB, as well as the new value PPA of the parameter pointer; finally, at the operation "return", the new value of the stack pointer, which suddenly decreases, can be deduced from the old value of the parameter pointer (in our case from PPB)

In this process *nothing* forbids A from being identical with B. The subroutine only has to appear in the memory once, but it may then have more than one simultaneous "incarnation" from a dynamic point of view: the "innermost" activation causes the same piece of text to work in a higher part of the stack. Thus the subroutine has developed into a defining element that can be used completely recursively.

Recursive Techniques and ALGOL 60

Every procedure is regarded as a subroutine in the sense described above. For the sake of simplicity a block that is not a procedure can also be treated as a subroutine, be it that this subroutine is only called in at one point.

One of the complications of ALGOL 60 is that not only local variables may be used in each block, but that explicit reference may be made in every block to variables that are local in a lexicographically enclosing block. When a subroutine is called in, the link contains *two* parameter pointer values for this purpose. Firstly, the youngest parameter pointer value corresponding to the block in which the *call* occurs (as described in the previous section), secondly, the value of the parameter pointer corresponding to the most recent, not yet completed, activation of the first block that lexicographically encloses the *block* of the subroutine called in. As already mentioned, the first parameter pointer value plays a vital role in the return at the end of the subroutine, the second is indispensable in localizing the global variables in the stack. As the second parameter pointer, by definition, points to a link in the stack, which in its turn contains a second parameter pointer value corresponding to the next enclosing block, the arithmetic unit can trace this "chain" and, in doing so, will find all parameter pointer values that may be necessary for localizing any global variable in which it may be interested.

One can assign a so-called *block number* to each block, indicating the number of blocks which enclose it lexicographically: the main program therefore has a block number $=0$. If the program refers to a global variable it is obviously necessary to specify the block in which the global variable was declared; the block number serves this purpose and, under control of this block number, the arithmetic unit can find the parameter pointer value it now needs. Further, the reference to a global variable will specify which variable of this block is required: its position with respect to the parameter pointer value just found achieves this. The introduction of the block numbers also makes it possible that the arithmetic unit has immediate access to all the parameter pointer values it may need. They can be stored in order of increasing block number in a so-called "display" (e.g. a series of index registers, numbered 0, 1, 2, 3, ... etc., as many as the maximum value of the block number). By tracing the second chain of parameter pointers one can, when necessary (amongst others at the return, in the call of a formal procedure, and at the beginning of the evaluation of a non-trivial formal parameter), bring the display up to date. This needs —and can in general—only be done for block numbers not exceeding the number of the block we are about to enter. It will not surprise the reader that the block number is to be regarded as specifying the state of the arithmetic unit; in consequence it has to be stored amongst the link data in the stack.

Acknowledgements. The author is especially indebted to Professor A. VAN WIJNGAARDEN and J. A. ZONNEVELD, co-members of the team that is engaged in the construction of a compiler for ALGOL 60 at the moment of writing. During the course of this work many aspects, which he had tentatively written down at an earlier stage, became clearer, and their consequences became apparent under the pressure of the circumstances.

Furthermore, the author had the great privilege of having some inspiring conversations on this subject with Professor H. D. HUSKEY, when he was the guest of the Mathematical Centre, Amsterdam, in the summer of 1959.

References

[1] BAUER, F. L., and K. SAMELSON: Sequentielle Formelübersetzung. Elektronische Rechenanlagen **1**, H. 4, 176—182 (1959).

Mathematical Centre
Amsterdam-O/Niederlande
2e Boerhaavestraat 49

(Received Mai 11, 1960)