The design of a state space with a useful structure.(I)

We now turn to the discussion of another problem that is already old; it has been solved in one way or another in each implementation of ALGOL 60. In retrospect it is surprising that the problem has been so hard to solve and that with very rare exceptions --Burroughs Corporation is the exception that comes to mind immediately!-- computer manufacturers have been very slow in recognizing the problem and in designing hardware suitable for its solution.

In EWD719 I introduced the transition from uni- to multiprogramming with the requirement that different programs using the same library subroutine could share the same bit pattern representing the subroutine body in store. This, of course, creates problems when information varying from call to call --such as the return information, to mention a very simple example-- is allocated within the bit pattern representing the subroutine body. This allocation was very much ingrained in people's minds, so much so that it found its way into the hardware --I remember a Siemens computer in which a subroutine jump with address n would store the return jump in location with address n and would substitute n+1 into the instruction counter!-- and for a while it was thought that, therefore, invocations of a shared subroutine would have to exclude each other mutually in time --a synchronization for which aforementioned Siemens machine lacked the necessary facilities-- . But the mutual exclusion was identified as a red herring as soon as recursion entered the picture: nested invocations are by their very nature not mutually exclusive.

In the case of subroutines the same piece of code can be invoked by different calls in the program. This implies that for each subroutine we can identify what we might call "its primary allocation commitment", i.e. that fixed part of the machine the state of which depends on the call and, therefore, in general will vary from call to call. In different machines different conventions for the primary allocation commitment have been chosen.

In the EDSAC organization we find a mixture of all conceivable conventions:

1) part of the primary allocation commitment consisted of storage locations allocated within the subroutine's bit pattern (such as the return jump and other variable instructions)

2) part of the primary allocation commitment consisted of the accumulator of the arithmetic unit (in which at call the return information was transmitted)

3) a few fixed storage locations (such as location 0) were used by many subroutines for parameter passing

4) for some subroutines part of the primary allocation consisted of an area of store to be chosen by the programmer and to be fixed with a preset parameter (the difference between the last two being that the first commitment was common to many subroutines, whereas the last one was usually specific to a single one).

In a machine like the IBM/360 with 16 general purpose registers in its arithmetic unit the convention of using a few fixed storage locations was usually not introduced.

In the EL-X1 the order code comprised 16 different subroutine jumps, each of them placing the return information in one of 16 consecutive storage locations (from 8 through 23); each subroutine had to be called by its proper subroutine jump and the corresponding location belonged therefore to its primary allocation commitment.

The first convention --a primary commitment to store allocated within the code-- was forced upon the users of the first von Neumann machines for lack of B-lines (= index registers). It became a standard to such an extent that subroutines in which it was not adhered to were for at least a decade honoured with a special name: they had "re-entrant code". Today it should not be a special feature anymore: subroutines stored in ROM (Read Only Store) are necessarily "re-entrant".

Confining ourselves to re-entrant code we still have have a choice: do we try to give different subroutines in principle the same or in principle

different primary allocation commitments? In the EL-X1 --and that is the reason why I mentioned it-- a mixed strategy was adopted. Subroutines were arranged in a hierarchy: all subroutines that called no others were placed on level 0 , a subroutine calling others was placed on level n+1 when n was the maximum level of the subroutines it called. Subroutines on level n used location 8+n for their return link. At the time we thought this a bright idea, but it was, of course, a mistake. The trouble is that the locations from 8 to 8+n now form part of what we might call the primary allocation commitment of the program as a whole, and if the machine were to be used for multiprogramming a switch from one program to another would imply saving and restoring the contents of those locations.

In the IBM/360 and its copies it is usual to give each subroutine a heavy primary allocation commitment among the general purpose registers of the arithmetic unit: for reasons of homogeneity usually all of them. For reasons of speed the registers are the coder's most beloved locations for storing frequently accessed information and the machine has indeed been designed under the assumption that the coder would use it that way. That was a mistake too, for the necessary saving and restoring of register contents now imposes a heavy overhead on the mechanism of subroutine call and return, so heavy as a matter of fact that for such machines so-called "macro expansion" --i.e. replacing the call by (an adapted version of) the whole subroutine body!-- has become a usual implementation technique. (And where the technique was not applied, as in some implementations of PL/I , the poor programmer got the advice to avoid using procedures!)

The most homogeneous solution gives all subroutines the <u>same</u> primary allocation commitment, but keeps this commitment as small as possible in order to reduce the overhead of call and return as much as possible.

<u>Note</u>. The time taken for the execution of a program or program part can be "described" --the quotes because the terms used will not be defined precisely-- by ii + w/v , where ii stands for "initial investment" and v for the speed with which the useful work w is done. In the case of a whole program, ii might stand for the time taken by the compilation and w/v for the time taken

by the execution of the compiled program. In the case of a subroutine, ii might stand for the overhead of call and return and w/v for the time taken for the execution of the subroutine proper. It is always tempting to increase ii if by doing so v can be increased considerably: it is the temptation to which is yielded by the introduction of optimizing compilers, for instance, which themselves can be very time-consuming indeed. It should be noted that yielding to the temptation can be very dangerous, viz. in all those cases in which it cannot be guaranteed that w will indeed be big enough to get a proper return for the investment: the cost/performance ratio $(ii + w/v)/w$ viewed as function of w is unbounded when $ii \neq 0$! Installations spending most of their time compiling, loading, linking etc. have indeed been quite common. Because in most procedures w , i.e. the amount of useful work to be done, depends on the values of the parameters supplied at the call, a small value of ii is most desirable; otherwise one can get most unpleasant surprises when for a large number of calls w turns out to be small. (End of Note.)

The scene changed in the late fifties, when the following observation was made. Consider in the execution of a sequential program the time sequence of calls and returns; it satisfies the well-known grammar

$$< \text{sequence} > ::= \{ < \text{call}_i > < \text{sequence} > < \text{return}_i > \}$$

i.e. it is a nicely nested sequence with the call of A and the corresponding return from A as a matching bracket pair. To any moment in time corresponds a place in this sequence and we only need to store the return information corresponding to the bracket pairs that surround this place: they correspond to the calls of which the corresponding returns have not taken place yet. So let us introduce a non-volatile nomenclature for the members of this floating population of the bracket pairs that surround the current place in the sequence. The only orderly way we know of is assigning the first free number whenever the current place passes a call in the sequence, i.e. when a new member is added to the population of surrounding bracket pairs. Thanks to the fact that the sequence is nicely nested this strategy gives rise to a closed nomenclature! Each bracket pair becomes identified with a an ordinal number equal to the number of bracket pairs surrounding it.

This discovery is, of course, too beautiful not to be exploited. We introduce for such ordinal numbers the term "invocation numbers"; note that the invocation numbers provide a terminology that is local to the execution of a sequential program. With the execution of a sequential program we associate a Current Invocation Number, "CIN" for short. When the program execution is started, the CIN is initialized at zero, at every call the increase CIN:= CIN + 1 takes place, while CIN:= CIN - 1 accompanies every return. In a table with the invocation number as selector the return information can be stored in the entries under control of CIN ; upon return the return information can be retrieved again under control of CIN. At any time the value of CIN gives the length of the table, which is used as a so-called "stack", in which the entries reside on a so-called LIFO (Last-In-First-Out) basis. In other literature the CIN is also referred to as "the current depth of calling".

A machine built in the early sixties (the English Electric KDF9) had this built in, but alas under the assumption that a maximum value of 16 for CIN would suffice, an assumption that is hard to defend in the case of recursion. At the time of the design of the KDF9, recursive routines were not regarded as a reasonable programming tool; the problem is of course that the subsequent existence of machines on which the implementation of recursive routines presents serious problems prolongs the lifetime of such a prejudice.

Recursion drove another message home. Each routine needs in general some "working space" to allocate its local variables in. In uni-programming and without recursion one can establish a one-to-one correspondence between a routine and its local workspace. A naive implementation would then play it safely, allocating store in such a way that the local workspaces of any two different routines would be disjoint. But such a naive allocation does not exploit the fact that the local workspace of a routine is only needed between its call and the subsequent return from it: during its "activation", as it was called. Hence so-called sophisticated implementations would perform an an extensive flow analysis of the program, trying to establish for any pair of routines whether their activations would always be disjoint in time. And then one can try to allocate working spaces, allowing as much overlap as pos-

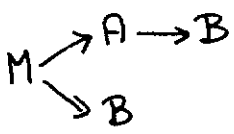
sible, but only between the working spaces of those routines for which it had been established that their activations would never overlap. This allocation problem is as bad as any graph colouring problem. In retrospect it is amazing that people have tried such solutions and even prided themselves on having done it. The problem should never have been solved! It was purely generated by regarding the routine, instead of the invocation, as the unit of thought. That in this connection the invocation was the proper thing to consider was the message that was driven home by recursion, where the same routine may enjoy any number of simultaneous activations (but then each corresponding to a different invocation).

With invocation numbers we have introduced a really new element: a nomenclature that grows and shrinks as the computation evolves. As such its introduction really implies a significant shift of the interface between the program text and its interpreter.

The following may clarify the role of the invocation numbers. Consider for simplicity's sake a program in which no procedure is handed over as a parameter. In that case we can easily define for that program a directed "activation tree" as follows:

the main program is the root of the activation tree, and each node of the activation tree has an outgoing edge for every routine it may call, with (an activation of) the routine called as its target node.

A nomenclature that distinguishes between all the nodes of this tree would provide a constant nomenclature that at any moment in time would distinguish between all "live" activations. Without recursion the tree is finite, e.g.



but we do already have the possibility that the same routine --for instance, B in the above example-- occurs at several nodes of the activation tree, i.e. the correspondence between routine and node is in general a one-to-many one. In the case of recursion the activation tree is infinite, and at least one routine occurs at an infinite number of nodes.

At any moment in time the current activation corresponds to one of the nodes of the activation tree. Its invocation number is its distance from the root. And we now exploit the fact that the local work spaces corresponding to the nodes on the unique path from the root to the current activation are the only ones that matter, and that in that subset, i.e. along that path, distance from the root (= invocation number) identifies the node (= activation) uniquely. The only special consequence of recursion is that along a path from the root the same routine may occur infinitely often instead of at most once.

I assume the local variables of a routine within its local work space identified by (small) integers, historically called "the displacements". (Think, to simplify matters, for the time being about a local work space only comprising a fixed number of variables, each requiring one word of storage; "fixed" means here: characteristic for the routine and independent of its activation.) To all intents and purposes we can treat the displacements in terms of which the routine body refers to its local variables as a closed nomenclature; the displacement acts as a selector from the local work space.

Within the program each stored variable is then during its lifetime fully identified by an ordered pair (invocation number, displacement). For the program as a whole we introduce a

Program Invocation Table, PIT for short

selector: invocation number

entry : return information,
starting address of local work space corresponding to this invocation.

Via the Program Invocation Table PIT we can derive from the ordered pair (invocation number, displacement) the address of the location in which the value of the variable in question is currently stored.

(To be continued.)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

21st November 1979

prof.dr. Edsger W. Dijkstra
Burroughs Research Fellow